**Technische Universität Berlin**

Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Datenbanksysteme und Informationsmanagement

Fakultät IV
Einsteinufer 17
10587 Berlin
https://www.dima.tu-berlin.de


Master's Thesis

# Fault Tolerant Distributed Window Aggregations For Internet of Things

Ankush Sharma

Matriculation Number: 0415479
15.08.2020

Supervised by
Prof. Dr. Volker Markl

Advised by
Philipp Grulich

Hereby I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Berlin, 15.08.2020

. . . . . .

*Ankush Sharma*

**Abstract**

With the rise of the Internet of Things (IoT) over the last few years and its predicted growth of upto 3.5 to 20 billion connected devices in the next few years, new network structures are required to efficiently handle IoT data. One new concept is Fog computing, which makes it possible to place computational nodes near IoT sensors. These computational nodes are called fog nodes, and are characterized by low storage, memory and network capabilities. The main advantage of such fog networks is the improved latency and network cost due to the close physical proximity of fog servers and IoT sensors. More over, IoT data is streaming in nature, and therefore lends itself to all the benefits that Stream Processing Engines(SPE) provides. In order to analyze continuous flow of raw events, SPE's use windowing to break down the streams into time-based chunks. In order to perform meaningful analysis on top of windows, data aggregations are needed to be performed. Data aggregations that can be performed in a distributed manner on distinct subsets of its parent multiset are known as Algebraic Aggregates. Examples of such aggregations include sum, count, average amongst others.

This thesis presents Fargo, a SPE for fault tolerant in-network window aggregations that can be applied for fog computing scenarios. Our model focuses on Sliding Windows and Algebraic Aggregate functions. Fargo uses data parallelism to distribute the aggregations on independent nodes. It detects node failures using Phi-Accrual Failure Detector, and provides fault tolerance by using Conflict-Free Replicated Data Types(CRDT) to replicate partial aggregates. Fargo is equipped with an Events Re-stream Protocol that is rendered when nodes performing the sub-aggregation fail. By replicating partial aggregates instead of raw events, Fargo saves time and minimizes Window latency by avoiding re-computation of events that had already taken place in the failed nodes. Additionally, we also provide CRDT specifications for multiple Algebraic Aggregate functions.

**Zusammenfassung**

Die Anzahl von Internet der Dinge (Internet of Things, IoT) basierten Geräten erfordert in den nächsten Jahren neue Netzwerkstrukturen und Ansätze für den effizienten Umgang mit IoT-Daten benötigt. Ein neues Konzept ist Fog-Computing, welches es ermöglicht, Rechenknoten in der Nähe von IoT-Sensoren zu platzieren. Diese Berechnungsknoten zeichnen sich durch geringe Speicher-, Rechen- und Netzwerkfähigkeiten aus. Der Hauptvorteil solcher Fog-Netowrks liegt in der verbesserten Latenzzeit und den geringeren Netzwerkkosten aufgrund der räumlichen Nähe von Fog-servern und IoT-Sensoren. Darüber hinaus sind IoT-Daten von Natur aus Streaming-fähig und eignen für die Verarbeitung mittels Stream Processing Engines (SPE). Um den kontinuierlichen Fluss von Rohdaten zu analysieren, verwendet SEPs Windows, um die Ströme in zeitbasierte Stücke zu zerlegen. Um aussagekräftige Analysen über Fenster durchzuführen, müssen Datenaggregationen durchgeführt werden. Datenaggregationen, die auf verteilte Weise an verschiedenen Untergruppen des übergeordneten Multisets durchgeführt werden können, werden als algebraische Aggregate bezeichnet. Beispiele für solche Aggregationen sind u.a. Summe, Anzahl und Durchschnitt.

In dieser Arbeit wird Fargo vorgestellt, ein SPE für fehlertolerante, netzwerkinterne Aggregationen, die für Fog Szenarien eingesetzt werden können. Dieses Modell konzentriert sich auf Sliding-Windows und algebraische Aggregatfunktionen. Fargo verwendet Datenparallelität, um die Aggregate auf unabhängige Knoten zu verteilen. Es erkennt Knotenausfälle mit dem Phi-Accrual Failure Detector und bietet Fehlertoleranz durch die Verwendung von Conflict-Free Replicated Data Types (CRDT) zur Replikation von Teilaggregaten. Fargo ist mit einem Replecation ausgestattet, das ausgeführt wird, wenn ein Knoten, der die Aggregation durchführt, ausfällt. Durch die Replikation von Teilaggregaten anstelle von Rohereignissen spart Fargo Zeit und minimiert die Fensterlatenz, indem es die Neuberechnung von Ereignissen vermeidet, die bereits in den abgestürzten Knoten stattgefunden haben. Zusätzlich bieten wir auch CRDT-Spezifikationen für mehrere algebraische Aggregatfunktionen. .

# Contents

# List of Figures

# 1 Introduction

With the rise of the Internet of Things (IoT) over the last few years and its predicted growth of upto 3.5 to 20 billion connected devices in the next few years [ris], new network structures are required to efficiently handle this IoT data. One new concept is Fog Computing, which acts as a bridge between Edge and Cloud Computing. While edge computing deals with the processing of individual data directly at the IoT devices [GLME+15], Cloud Computing deals with a centralized management of all data in Cloud data centers [VRMCL09]. In contrast, Fog Computing deals with management, storage, and processing of data on network nodes between the IoT devices and the data center [ARC+19]. The main advantage of such fog networks is the improved latency and network cost due to the close physical proximity of fog nodes and IoT sensors.

As data is emitted continuously by IoT sensors at a specified rate, the analysis of this kind of data is a natural fit for distributed streams processing. IoT data is characterized by high-volume and high-velocity, given that IoT sensors can be numerous in number ranging from a hundred thousand to millions. The sort of frameworks that are equipped to handle such a volume and speed of events are Stream Processing Engines(SPE). Modern SPE's such as Apache Flink[CKE+15] and Apache Spark[Spa] have seen massive adoption in the industry in the past decade. These SPE's are fault tolerant and provide a rich collection of libraries that the user can use to perform ad-hoc queries over data streams. However, one caveat is that these SPE's are meant to be deployed in a single data center. As IoT sensors can be spread across a wide geographic region such as multiple cities, a country, or even a continent, the cost of streaming the events emitted by IoT sensors to a single data center can be a limiting factor when it comes to real-time analysis of these streams[VCG+15]. This problem can be solved by moving the computation from the data center to near the sensors themselves.

In order to run analysis over streams, SPE's employ the use of a technique called Windowing to discretize the stream into time-based chunks. A user then performs their query on top of Windows. In most cases, queries are some sort of aggregations such as sum, average or count, amongst others. For instance, in a smart home setting, a user might be in finding the average temperature across all homes in the last five minutes. A user interested in performing such a query will define the Window size, the time unit to create Windows, and a function that takes as input the window and returns an aggregated average as output.

As the main focus of Edge Computing is to perform computations near the source, these streaming windows need to be created and processed on a network path between the source to the querying node. In other words, there is a need for SPE's that can create window on multiple independent nodes, perform sub-aggregations in parallel, merge sub-aggregations and ensure fault tolerance. The three key characteristics that we desire

from such fault tolerance are (1) detecting failures swiftly,(2) aggregate correctness in the face multiple node failures and (3) minimum performance impact. By aggregate correctness, we mean that the final result of a Windowed Aggregation in the face of multiple node failures must be equal to the Windowed Aggregation when there were no node failures. Furthemore, by minimum performance impact, we mean that there should be very minimal to no impact on latency and throughput in case nodes fail.

One such SPE that provides in-network window aggregations but without providing fault tolerance is Disco[Dis]. Disco is a SPE that focuses on complex in-network distributed window aggregations. The topology of Disco is based on a hierarchical-tree where the bottom-most layer contains sensors that constantly emit events. The layer above the sensors contain Child Nodes that process these raw events that emit partial aggregates. Data is then aggregated as it moves up the tree, where each layer of the tree then merges the partial aggregates received from the layer below it. Another such framework is Frontier[OSP18]. Frontier is a distributed and resilient SPE for IoT. To our knowledge, Frontier is the only SPE for Edge Computing that provides fault tolerance. It does so by deploying multiple instances of replicated operators, called a Replicated Dataflow Graph. Its algorithm for fault tolerance is dependent on buffering data on the device itself and re-sending it to a replicated instance of the operator in case of operator failure. The one shortcoming of this approach is that in case an operator goes down, Frontier misses on the opportunity to exploit partial computations already done by the crashed operator.

Most practical applications of data streams processing work with one or more large data streams such as click-streams or streams of stock prices. In the realm of Edge and Fog Computing, this situation is reversed. Instead of one or very few large streams, we have multiple IoT sensors emitting events at a steady rate. In such a scenario, there is a need for a SPE that can not only efficiently handle multiple streams, but also ensure fault tolerance. In summary, what is needed are answers to the following questions

1. How to create and merge Windows across independent nodes?
2. How to detect failures of nodes that perform sub-aggregations independently?
3. How to ensure fault tolerance when multiple nodes fail in a way that does not degrade the performance of the entire system?

## 1.1 Motivation Example

In this section, we give an intuition behind what it means to perform in-network aggregations and the questions that need to be answered when the nodes performing the sub-aggregations fail.

In Fig 1.1, we illustrate an in-network aggregation that performs a simple distributed count aggregation. This is meant for illustration purpose only, different in-network aggregations frameworks might have a different architecture. In our example, we have a hierarchical tree-based topology with the height of the tree being three. The leaf nodes

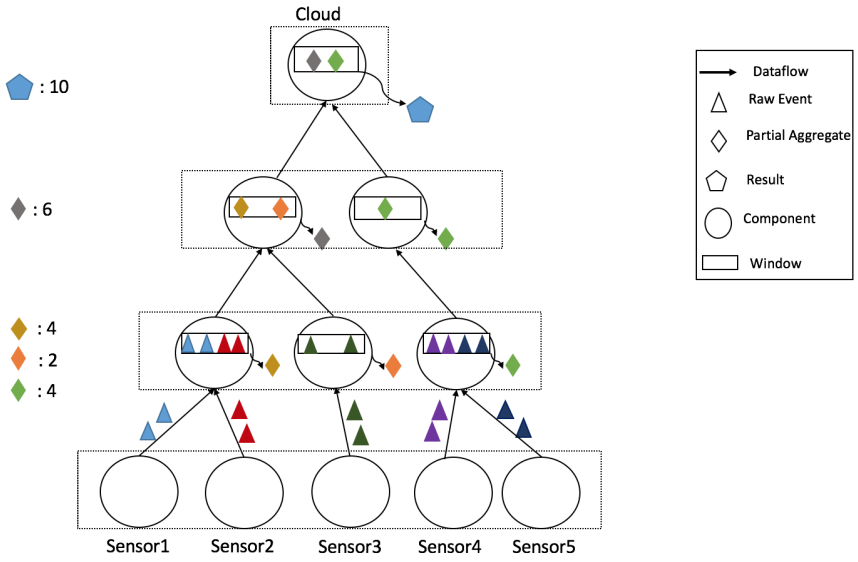



Figure 1.1: In-Network Aggregation

represent sensors. Each sensor emits two events depicted by triangles of different colors. For the sake of simplicity, let us assume that we have a Tumbling Window on all the layers above the sensors. Each one of these Tumbling Windows have the same start and end time. The sensors emits events to the layer above it, and each layer puts an event in a Tumbling Window based on some time-based parameters. When the window finishes, each window emits a *partial aggregate*, depicted by the quadrilateral. We see that in the layers above the sensors, the three nodes emit partial aggregates with counts 4, 2 and 4, as highlighted by the quadrilateral with different colours. Upon the completion of the Tumbling Window on each of these nodes, they emit this partial aggregate up the tree to the layer above it. The nodes at the layer below the root node essentially perform the same count aggregation, and emit more partial aggregates. The same steps are repeated in every layer of the tree until it reaches the root, which we call as the Cloud. The root merges the partial aggregates received from its immediate child nodes, and emits the true aggregate for the Tumbling Window, which is 10, as depicted by the blue polygon. In this kind of topology, the correctness of the true aggregate computed by the root node is guaranteed only if none of the nodes fail.

In Fig 1.2, we illustrate a scenario of an in-network aggregation wherein multiple nodes have failed in different layers of the tree. We label these nodes as N1 and N2. We assume that the nodes at height one and two failed sometime during the start and end time of the Tumbling Window. Keeping our discussion for Fig 1.1 as our baseline in terms of aggregate correctness and performance impact, the following questions need to be answered in order to ensure fault tolerance :

1. As N1 has failed, what happens to the events emitted by Sensor1 and Sensor2?

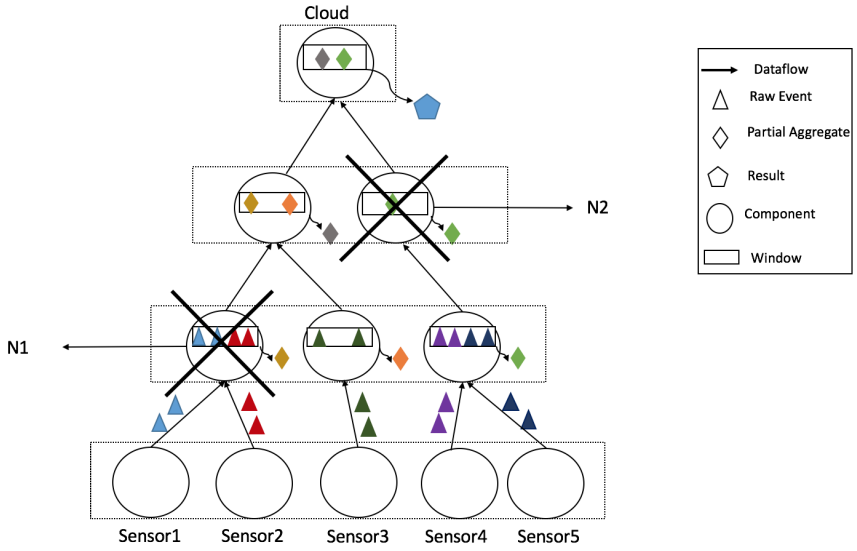



Figure 1.2: Failed Nodes in In-Network Aggregations.

2. How and who detects N1 to have failed?

3. In order to ensure aggregate correctness, the events emitted by Sensor1 and Sensor2 need to be re-streamed. How and who will re-compute the partial aggregate for these events?

4. How and who detects N2 to have failed?

5. As N2 has failed, what happens to all the partial aggregates that were computed on the sub-tree below it?

In Chapter 4, we look at different scenarios in which failures such as ones listed above can happen and how to react to such failures.

## 1.2 Contribution

In order to address the problem of fault tolerance in a decentralized and distributed streams processing setting, we propose Fargo, a Stream Processing Engine for fault tolerant distributed in-network window aggregations. Fargo supports basic commutative and associative aggregations called Algebraic Aggregations, that allow for computing aggregates on distinct subsets of the parent multiset. To achieve fault tolerance, we leverage state-of-the-art approaches in two sub-fields of distributed computing, namely Failure Detection and Eventual Consistency. To detect failures, Fargo uses Phi-Accrual Failure Detector[HDYK04], which is a network adaptive failure detector that outputs a suspicion value on whether a process has crashed. To achieve Fault Tolerance, Fargo

uses Conflict-Free Replicated Data Types(CRDT)[SPBZ11] to asynchronously replicate partial computations to other nodes. In summary, we contribute the following :

1. We propose Fargo, a new fault tolerant window aggregations framework that allows for efficient and fault tolerant, in-network and incremental data aggregations in a streams processing setting. We show that our framework provides fault tolerance with at least-once-delivery semantics.

2. We evaluate Fargo, showing that our framework is tolerant of multiple failed nodes and in most cases there is no increase in latency or decrease in throughput.

3. We provide Delta-based CRDT specifications for the following aggregations : sum, average, max, min, and range. We formally prove that our specification for the sum aggregation data structure is in fact a state based CRDT.

## 1.3 Scope and Delimitation

The main focus of our work is to investigate whether lightweight fault tolerance is possible for simple Algebraic Aggregate functions. To do this, we limit our scope to only Sliding and Tumbling Windows. To achieve this, we built a prototypical distributed Streams Processing Engine Fargo. Moreover, we limit our incremental aggregations only on real numbers. Similar reasoning can be used to perform incremental aggregations on complex numbers as well. Furthermore, unlike most modern SPE's, we do not consider complex features such as handling out-of-order streams or coping with backpressure. Our prototype is built on the assumption that the underlying communication channel is fair-lossy, i.e messages can be dropped or duplicated. For communication between distinct processes in our system, we use a custom string-based encoding scheme. Furthermore, we only focus on non-byzantine fault tolerance. Lastly, as soon as our topology is instantiated, we assume that nodes can fail but new nodes cannot enter the topology.

## 1.4 Thesis Outline

We now present an outline for our thesis.

**Chapter 2**
In Chapter 2, we introduce the technical concepts upon which our framework is built. We start by discussing Fog Computing, Stream Processing Engines and Windowing Operators. We then proceed to formally define and discuss distributed fault tolerance concepts such as failure detection, the meaning of time and order in a distributed system, and end our discussion with CRDT's.

**Chapter 3**
In Chapter 3, we introduce Fargo's dataflow and topology. We first discuss the high level architecture and explain each component of the architecture. We then proceed

to explaining in detail the different protocols that ensure fault tolerance. We end this chapter by providing by formally proving that our data structure for the sum aggregation is a CRDT.

**Chapter 4**

In Chapter 4, we evaluate Fargo. We perform two experiments. The first experiment measures throughput that a eight node cluster can achieve in a Tumbling Window of 10 seconds. As a baseline, this experiment computes throughput of a eight node cluster performing a count aggregation. We then evaluate the impact on throughput with different replication and heartbeat rates. The second experiment measures the impact on latency and throughput when we manually crash one, two and three nodes out of our eight node cluster over a Tumbling Window of 10 seconds with eight sensors emitting raw events. We fail our nodes in the beginning, middle and the very end of our window to measure the impact on latency and throughput.

**Chapter 5**

In Chapter 5, we present an overview of related work in the area of SPE's, in-network aggregations and Replicated Data Types.

**Chapter 6**

In Chapter 6, we summarize our work and findings and give an outlook towards future work that be done in Fargo.

# 2 Background

In this section, we cover core concepts upon which our contribution is build upon. We start our discussion with Stream Processing Engines(SPE) in Section 2.1. In order to perform aggregations over data streams, SPE's use a technique known as Windowing, which is discussed in Section 2.2. Furthermore, we also expand upon different windowing techniques that enable us to perform data aggregations. In Section 2.3, we discuss various types of Failure Detectors and touch upon the Phi-Accrual Failure Detector. In Section 2.4, we discuss the meaning of time and order in a distributed system. Finally, we finish our discussion in Section 2.5 on Conflict-Free Replicated Data Types.

## 2.1 Stream Processing Engines

A Stream Processing Engine(SPE) is an application that performs computations over potentially unbounded and continuous data streams. A SPE running a computation can either do it centralized on a single thread or distribute it over multiple threads or even multiple physical machines. In order to run a computation using a SPE, a user needs to define a streaming query, which is defined as a chain of streaming operators. With the rise in unbounded and global datasets such as web logs, mobile usage statistics, and sensor networks, companies are are increasingly using SPEs in their day to day business. [IAM+19]. For several application scenarios such as fraud detection in financial transactions and healthcare analytics involving digital sensors and IoT, the answers to queries over data streams must be processed with very low latency [dAVB17]. The primary reason for this is because stock values, credit card transactions, traffic conditions, time-sensitive patient data, and trending news rapidly depreciate in value if not processed instantly. In other words, real time analytics has become crucial for modern data-driven organizations, and SPEs play a key role in achieving this.

In this remainder of this section, we discuss some core concepts of SPEs in Section 2.1.1, and end with streaming operators and streaming queries in Section 2.1.2 and 2.1.3 respectively.

### 2.1.1 SPE Concepts

This section is devoted to discussing fundamental concepts in Stream Processing Engines. We discuss the techniques employed by modern SPEs to perform data processing such pipelining as micro-batching and distributed streams processing,

**Record-at-a-time processing vs Stream-Discretization**

By definition, an SPE processes data and performs computations over continously moving, fast and unbounded data streams. SPEs can do this in two ways : record-at-a-time processing or stream-discretization.

In stream-discretization(also called micro-batching), data is collected in small groups called batches. Micro-batching is a variant of the traditional model of batch processing wherein a computation is performed on the entire dataset at rest and the size and distribution of the dataset is known before hand. Using micro-batching, an SPE discretizes a continuous computation to into small batches. However, micro-batching is a technique that is not just limited to SPEs. A batch processing engine can use micro-batching to provide streaming capabilities. Amongst modern data processing systems, the biggest proponent of this technique is Apache Spark [CZC+13]. Spark treats each micro-batch as an atomic batch job, where each job can either succeed or fail. If a failure occurs when a micro-batch is being processed, Spark just recomputes the micro-batch altogether. An advantage of this approach is high throughput which is a result of lower communication overhead and better resource utilization. However, one major limitation of this approach is increased latency. This is due to the fact that records are not made available to the downstream operators as soon as they are processed by an upstream operator.

On the other hand, in record-at-a-time processing(also called pipelining), each event is processed individually. As soon as the event enters the SPE, it is processed and passed on. This usually leads to low latency, as an event can proceed from one stage of the pipeline to the next as soon as the upstream stages are done processing it. However, this may also lead to a reduced throughput, as single events may lead to higher overhead in communication and typically does not lend itself for optimum hardware utilization, unlike batch processing. We refer to previous work done in regards to the tradeoffs between the two approaches [KRK+18b].

In Fig 2.1, we see pipelining and micro-batching in action. In Fig 2.1(a), we see that events arriving from a stream are first buffered by the SPE. Then, each individual event flows through a series of operators. Each operator takes as input a single event, transforms it, and makes it available for processing for the next operator in the chain. In Fig 2.1(b), we see that the SPE reads from the buffer and creates a micro-batch of 3 events. Each operator works on a micro-batch of 3 events and only after the operator is done processing the micro-batch does it make the batch available for processing for the next operator in the chain. This is a simplistic explanation of of complex techniques, but the intuition behind these techniques should be understood.

### Centralized vs Distributed Streams Processing

As defined above, a SPE is an application that processes data streams. A SPE can either be implemented as a centralized or a distributed system. We define a centralized SPE as an application that runs on a single site. The stream data sources may or may not be distributed, but the computation performed by the SPE is centralized on a single site. A centralized SPE is easier to implement and reason about. One big limitation of such systems is the lack of fault tolerance. Since processing happens on a single site,

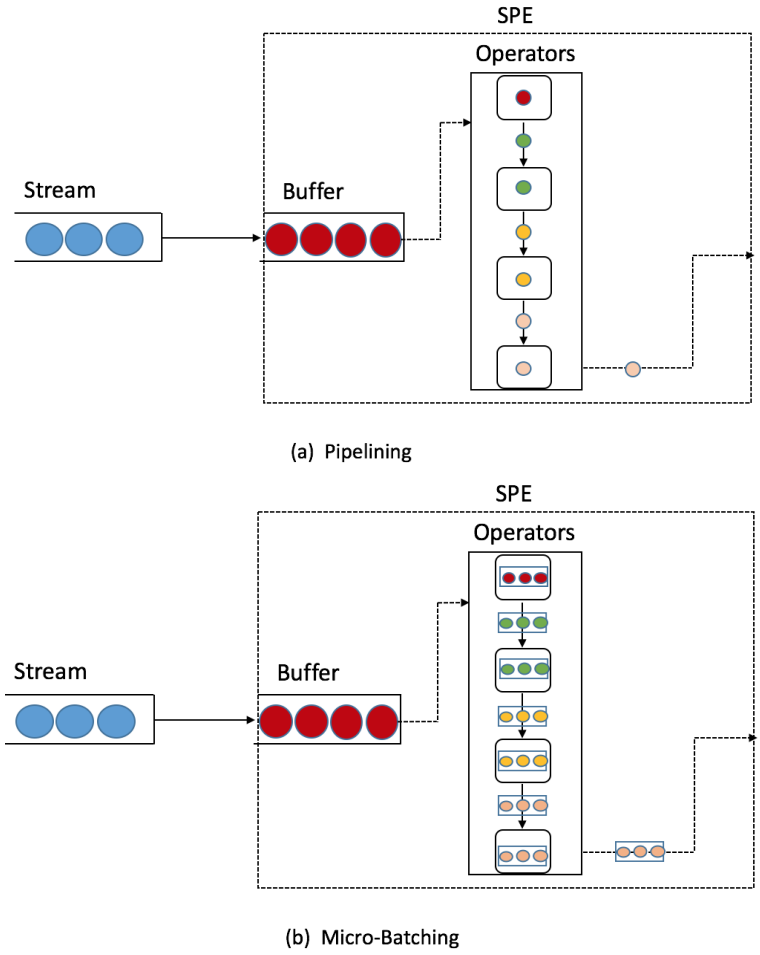



Figure 2.1: Pipelining vs Micro-Batching

these SPEs have to sacrifice high availability in the face of site failure. One such SPE that is well known in literature is Aurora [SBDBHGM+70]

A distributed SPE, on the other hand, runs on a cluster comprising of multiple sites. This is especially useful for organizations that are engaged in health-care, internet traffic monitoring, credit card fruit detection, smart cities, amongst others [MDCZFC+70]. These data sources are characterized by high volume stream of events that need to be collected in real time and also usually under a high velocity rate. This necessitates the need to parallelize computations on multiple sites, as no single site can handle such a high volume of streaming events without shedding load or incurring an increase in latency. The stream data source is usually partitioned and each processes a partition independently and in parallel. Usually, each event is equipped with a key, and for stateless operators, events with the same key can be processed in parallel. However, if some computation is needed to be performed all events with the same key, then shuffling

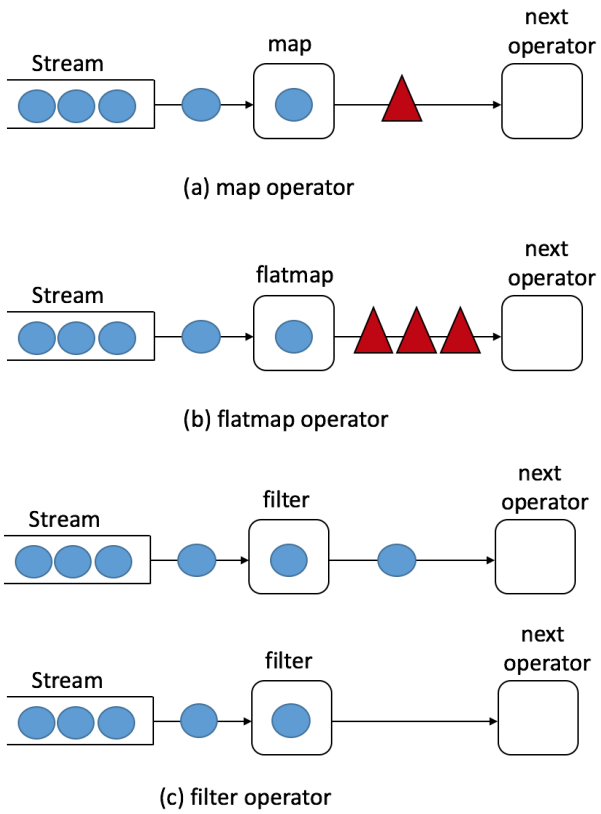



Figure 2.2: Map, Flatmap and Filter operators in a SPE

needs to take place across multiple sites. This results in tremendous communication and serialization/deserialization overhead for SPEs.

### 2.1.2 Streaming Operators

A computation(henceforth called query) in an SPE is modelled as a chain of operators. An operator is a function that takes as input an event from a stream, and emits zero or more results.

In Fig 2.2, we see common operators used by SPEs in action. In Fig 2.2(a), the map operator takes an event and applies a transformation. This transformation is one-to-one. A map operator always emits one transformed event. In Fig 2.2(b), the flatmap operator takes an event and emits one or more transformed events. In this sense, it is a one-to-many transformation and is a generalization of the map operator. The filter operator in Fig 2.2(c) takes an event and may or may not emit the same event. A filter operator is a predicate that will make the event available to the downstream operator only if the predicate returns true. This is by no means an exhaustive list of operators. An operator can be standard operators like the ones explained in the figure, but can also take the

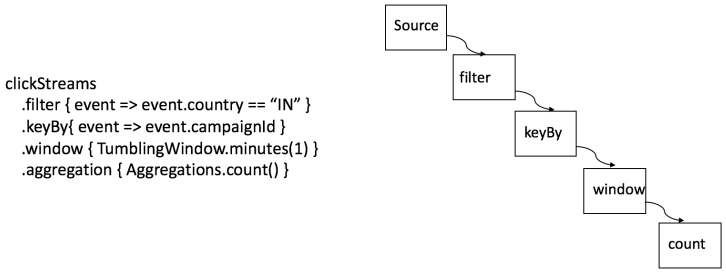



Figure 2.3: Logical graph of a streaming query based on a DAG implementation

form of an arbitrary user defined function(UDF).

### 2.1.3 Streaming Queries

A streaming query is a chain of operators. Events enter the SPE and flow through the chain with each operator emitting the result of the computation to the next operator in the chain. There are two phases in the implementation of a query. The first is a logical dataflow graph of the query, followed by the physical or execution dataflow graph. The logical graph constructed by SPEs such as Apache Flink [CKE+15] and Apache Spark [ZCD+12] is a DAG (directed acyclic graph), wherein each vertex in the graph is an operator and edges between the vertices represent the flow of streaming events. As the name suggests, the graph itself is acyclic, and the data can only flow in one direction. SPEs may further decide to modify the initial logical graph by applying optimizations [HSS+14]. After the logical graph is finally built, the SPE constructs an execution graph. This graph is the physical query plan that runs on top of a distributed runtime. In Apache Flink, for example, the vertices are tasks and the edges indicate input/output-relationships or partitions of data streams [CKE+15].

There are other SPEs that allow cycles in the logical graph and model a streaming query in the form of a directed cyclic graph. Most notable in literature is Naiad [MMI+13]. Naiad is an example of a distributed SPE for data-parallel cyclic dataflow programs. Because cycles are allowed in the dataflow graph, such SPEs lend themselves naturally to incremental and iterative algorithms used in machine learning and streaming graph algorithms [MIIM12].

In Fig 2.3, we illustrate the logical graph based on a DAG implementation. The query is run on top of click-streams, and gives the counts for campaigns grouped by key for all campaigns that rain in India broken down into 1 minute windows. The source itself is an operator that emits events continuous to downstream operators. The query *filters* out all clicks that belong to the country India, extracts the key from each event(in this case the campaignId), creates a tumbling window of 1 minute, and performs the count aggregation for each distinct key. On the right side is the logical DAG of this query.

## 2.2 Windowing and Aggregations in Data Streams

Data streams are continuous and unbounded sequence of events. In order to compute aggregation queries over unbounded streams, SPEs break down the infinite streams into smaller bounded subsets called Windows. Certain operators such map and filter do not require a bounded subset. For example, a map operator simply processes an individual event, transforms it, and passes it on. However, an aggregation such as median cannot be performed over an infinite stream, and therefore it becomes necessary to discretize the stream into small windows.

In this section, we discuss windowing in stream processing and distributed data aggregations, the central theme of our work. Section 2.2.1 is devoted to different Windowing measures that rely on time, followed by different types of windows in Section 2.2.2. We then discuss distributed data aggregation concepts in Section 2.2.3 and end our discussion with incremental aggregations in Section 2.2.4.

### 2.2.1 Window Measures

In order to discretize the stream into windows, there needs to be a notion of upper and lower bounds. These bounds can either be time based or count based. A count based window is a window $W$ with count $c$. A count based window discretizes a stream into fixed size windows of size $c$. Time based windows are created by discretizing a stream based on a notion of time.

A stream can be decomposed into windows in the past ten seconds, ten minutes or even one hour. That being said, the semantics of time in a streams processing setting is not as straight-forward and needs to be clearly defined. For instance, creating a window based on 'the last ten seconds' can be ambiguous as the 'last ten seconds' can refer to the time when the event was created by the data source or when it was processed by the SPE. To have more clarity, we distinguish between *event time* and *processing time* [ABC+15].

- **Event Time** is the time at which the event was generated at the source. It is the time at which the event occurred. Event time is an immutable artifact of an event, i.e it never changes

- **Processing Time** is the time at which the event was processed by the SPE. Usually, it is the system time of the SPE. Processing time varies depending where the event is in the pipeline of operators

In this work and in the remainder of the sections and chapters, we will be focusing only on time based windows based on event time.

Using event time as a measure to construct windows raises an important question : is it safe to assume that the event time is monotonic? In other words, if a SPE processes an event *e1* with event time *t1* followed by an event *e2* with event time *t2*, then is it safe to assume that $t1 \leq t2$ ? In reality, given the nature of distributed and networked systems, the monotonicity of event time is not guaranteed. Because of this, there is no guarantee

that the events will arrive in a SPE in the order they were created. This is known *out-of-order processing* and SPEs incur additional overhead to ensure correct behaviour if the ordering of events is relavant. To deal with out-of-order streams and lateness of events, SPEs employ the use of *watermarks*. Basically, a watermark is a threshold to specify how long the system waits for late events. If an arriving event lies within the watermark, it gets used to update a query. Otherwise, if it iss older than the watermark, it will be dropped and not further processed by the SPE [ABC+15].

In this work, for the sake of simplicity we assume that the streams are in-order and there is no lateness of events.

### 2.2.2 Window Types

We now present the most common window types that are employed by SPEs. Fig 2.4 shows these windows : Tumbling Windows in Fig 2.4(a), Sliding Windows in Fig 2.4(b) and session windows in Fig 2.4(c).

#### Tumbling Windows

Tumbling Windows require only a *size* parameter in order to be created. Using tumbling windows, a SPE can discretize a stream into non-overlapping chunks of a fixed *size*. This means that an event $e$ can belong to one window only. The *size* parameter is a defined in terms of time, e.g 5 seconds or 1 minute. As soon as an instance of a Tumbling Window finishes, the next one begins. Figure 2.4(a) depicts a Tumbling Window of size 10. As can be seen in the figure, there is no overlap and each event falls within a distinct Window.

#### Sliding Windows

A Sliding Window indow requires a *size* and a *slide*. The *slide* determines when a new window starts after the previous one. If the *slide* is smaller than the *size*, then sliding windows will overlap. Sliding Windows are a generalization of Tumbling Windows. If *size* is equal to *slide*, then a Sliding Window is nothing but a Tumbling Window as there is no overlap and the second Window starts right after the first one finishes. In figure 2.4(b), we see Sliding Windows in action. The figure shows a Sliding Window of *size* 10 and *slide* 5. The figure shows overlap between Windows, where each ball of the same colour belongs to two Windows. This redundancy may lead to increase memory usage and increase latency of computation being performed. We will not get into these optimization techniques and refer to earlier work done in this regard [TGC+19].

In this work, we will only be focusing on Sliding Windows.

#### Session Windows

Session windows are created by grouping all events into a window for a certain period of time. Session windows require the parameter *gap* that defines at what point in time a window finishes followed by no incoming events for *gap* time unit. In figure 2.4(c),

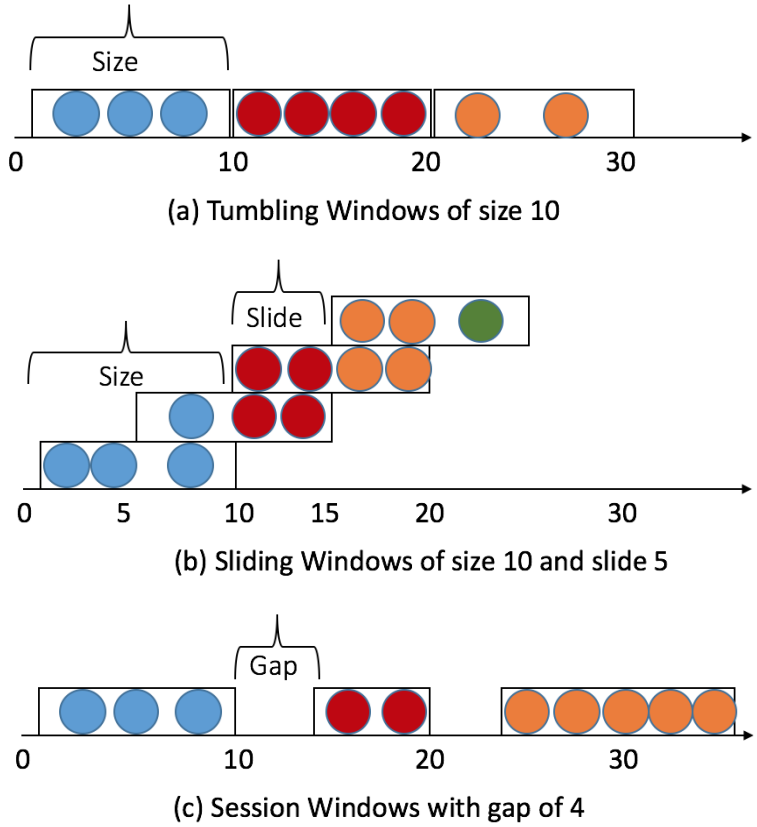



Figure 2.4: Tumbling, Sliding and Session Windows

after the first window finishes, there is a gap of 4 time units when there are no incoming events. When the event(red ball) appears, the second window is initiated.

### 2.2.3 Distributed Data Aggregations

To put it simply, data aggregation is the ability to summarize information [VR03]. For example, some of the most commonly used aggregations employed by SPEs are *MAX, MIN, COUNT, SUM, AVG* etc. However, the literature is not clear as to the upper bounds with regards to size of this summary. As a case in point, the identity function, which is a function that maps each event to itself is clearly not a good candidate to be called an aggregation. In practice, an aggregation should be logarithmic or of constant size with respect to the size of the input domain.

To formally define data aggregations and the different types of aggregation, we refer to the terminology introduced by Jesus et al[JBA11].

**Definition 1.** *An aggregation is a function from an multiset input domain $I$ to output domain $O$.*

$$f : I \to O$$

The input domain $I$ being a multiset emphasizes that an event may be repeated multiple times and the ordering of events is irrelevant. In this section, we discuss different types of data aggregations. Data aggregations fall into two categories, namely decomposable and holistic data aggregations. We now proceed to explain these categories

**Decomposable Aggregations**

An aggregation function $f$ is said to be decomposable if it can *decomposed* into multiple computations over sub-multisets of the primary multiset that is to be aggregated. To understand this concept better, consider the data aggregation *SUM* on the multiset of real numbers with the additive operator :

$$\mathbf{SUM}\{1,2,3,4,5\} = \mathbf{SUM}\{1,2,3\} + \mathbf{SUM}\{4,5\} = 15$$

On the left hand side, we see that the sum of the first five natural numbers is 15. Owing to the commutative and associative properties of the additive operator $+$, the *SUM* aggregation on the right hand side can be broken down into *SUM* over the multiset 1,2,3 and 4,5. Note that on the right hand side, the two sub-multisets can further be broken down into more sub-multisets and the result will still be the same. Such aggregations, wherein the aggregation(*SUM* in this case) can be operated upon any number of distinct sub-multisets and yield the same result as the aggregation on the primary multiset are called *Distributive Aggregate Functions*. To put it formally :

**Definition 2.** *Given non empty multisets X and Y, an aggregation function F is said to be a distributive aggregate if there exists a commutative and associative merge operator* $\bigoplus$ *in the output domain such that*

$$F(X \uplus Y) = F(X) \bigoplus F(Y)$$

In our *SUM* aggregation, X = {1,2,3}, Y = {4,5}. F(X) = SUM(X) = 6, F(Y) = SUM(Y) = 9. And finally, on the left hand side SUM(X $\bigoplus$ Y) = SUM({1,2,3} $\bigoplus$ {4,5}) = 15. Other aggregations such as *COUNT, MIN, MAX* are also distributive aggregate functions.

Then there are aggregations like *AVERAGE*, that embody the distributive nature of a distributive aggregate function, but need an auxiliary domain to hold the intermediate results. In our *SUM* aggregation, the output of *SUM* on the sub-multiset $X$ lied in the output domain of real numbers. To perform *AVERAGE* aggregation over the same multiset, each sub-multiset would be mapped to a pair *(sum, count)*, and the merge operator $\bigoplus$ computes the average by dividing the sum by the count. We call these types of aggregations as *Algebraic Aggregate Functions*.

**Definition 3.** *An aggregation function F is said to be Algebraic Aggregate if there exists a function G and a distributive aggregate function H such that*

$$F = G \circ H$$

Continuing with the same example as in the SUM aggregation, as per the definition of algebraic aggregate functions, the *AVERAGE* aggregation would take the form :

$$AVERAGE(X) = G(H(X))$$

$$H(\{x\}) = (x, 1)$$

$$H(X \uplus Y) = H(X) \bigoplus H(Y)$$

$$G((sum, count)) = sum/count$$

Here, H is the distributive aggregate function, and G is simply a function that returns the value of the true average by dividing the sum of values by the count of values seen. Another example of algebraic distributive function is the *RANGE* function in statistics, that gives the difference between the maximum and the minimum value.

**Holistic Aggregates**

An aggregation function that can only be performed as a unified single computation over the entire multiset, i.e the computation cannot be broken down into distinct sub-multisets are called holistic functions. Holistic aggregate functions are a case of centralized computations, wherein it is not possible to distribute the computation over distinct sub-multisets. An example of holistic aggregate functions is the *MEDIAN* aggregation. For the *MEDIAN* aggregation, we cannot split the multiset into subparts and merge their intermediate results.

$$\textbf{MEDIAN}\{1, 2, 3, 4, 5\} = 3$$

MEDIAN{1, 2, 3, 4, 5} requires us to see all the data to determine 3 as the median value. Other common holistic functions seen in statistics are *QUANTILES* , *RANK* and *COUNT DISTINCT*.

### 2.2.4 Incremental Aggregations

In this section, we introduce the concept of *Partial Aggregates* using the abstractions introduced by Tangwongsan et al. [THSW15]. An incremental aggregation is one that can be broken down into three functions : *Lift*, *Combine* and *Lower* as defined below.

**Definition 4.** *Lift is a function that maps a datum from an input domain IN to a datum called Partial Aggregate in the intermediate domain PARTIAL*

$$lift\colon IN \rightarrow PARTIAL$$

It is not always necessary that the input domain *IN* and intermediate domain *PARTIAL* are different. For distributive aggregate functions, the two domains are the same, and the *lift* function is just the identity function that maps an datum to itself. For an algebraic aggregate function operating on the multiset of real numbers like *AVERAGE*, it takes a datum {x} and maps it to the tuple (x, 1). For holistic functions such as *MEDIAN*, it returns a single-element list [x].

**Definition 5.** *Combine is a function takes two partial aggregates from domain PARTIAL and maps it to new partial aggregate value using a merge operator* $\bigoplus$

$$combine\colon PARTIAL \times PARTIAL \rightarrow PARTIAL$$

$$combine(X, Y) = X \bigoplus Y$$

Combine takes two partial aggregates and creates another partial aggregate using a merge operator in the domain *PARTIAL*. As an example, assume we have two partial aggregates called agg1 and agg2. For *SUM*, combine would simply add agg1 and agg2, i.e *combine(agg1, agg2) = agg1 + agg2*. For all decomposable functions, combine performs a partial aggregation towards the true global aggregate. For MEDIAN, combine merges two partial aggregates lists.

**Definition 6.** *Lower is a function that takes the a datum from domain PARTIAL and maps it to a datum from domain OUT. Lower is the final step of the aggregation and returns the true aggregate*

$$lower\colon PARTIAL \rightarrow OUT$$

Lower is the final result of the aggregation. Assuming that the partial aggregate is called partialAgg, lower for SUM would be the identity function, and for *AVERAGE* lower would perform *agg.sum* divided by *agg.count*. For *MEDIAN*, combine would sort the list and return the middle value.

## 2.3 Failure Detectors

A distributed system is a set of processes that perform computations independently and communicate with each other by message passing over a network. Over the years, researchers and the industry alike have come up with a list of eight fallacies that are typically made while designing a distributed system. The fallacies are [Fal]:

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

Five of these eight fallacies are directly related to the underlying communication channel between processes. Within the same cloud data center, there might be a power failure or someone trips on the network cord that connects different machines. Such a phenomena where the network devices connecting different processes fail is known as a *network*

*partition*. A study done by Alquraan et al. on the impact of network partitions in a cloud environment suggests that majority of the failures led to catastrophic effects, such as data loss, reappearance of deleted data, broken locks and system crashes [ATAAK18]. Things become even more complicated when a distributed system is connected via a network that is spread through different cities or even continents. In such a case, the network is connected through a plethora of network routers, cables and intercontinental submarines networks that constantly transmit and receive data. In such a heterogeneous environment, the probability of a network partition is high. Moreover, a process running on a machine may stop responding to other processes simply because the machine on which it is hosted just crashed. In such a scenario, it is impossible for other processes to know if the process has crashed or is just responding slowly. Be it a network failure, crash failure or software errors, we conclude that *faults* are a fundamental property of distributed systems. It is not a matter of if, but when they occur. Faults in a distributed system can be characterized as being one of the four types of failures[TB92]:

1. Byzantine Failure: Process produces unexpected results at arbitrary time and does not act as per specifications of the algorithm it is supposed to run. E.g: the control of a process being taken over by a malicious attacker
2. Crash Failure: Process stops after a failure or halts indefinitely
3. Timing Failure: Process responds correctly, but within a time frame that is outside the expected time interval
4. Omission Failure: Process either fails to send messages to other processes or receive messages from them

Fault tolerance can be defined as the property of a distributed system to continue functioning properly if one or more components fail. For a distributed system to be fault tolerant, it needs to isolate faults to the component where the fault occurred and then react on it. In this section, we discuss Failure Detectors, which are programs (or a distributed system by itself) that detects faults in a distributed system. In Section 2.3.1 and 2.3.2, we discuss the various properties that are used to classify Failure Detectors. These properties are called completeness and accuracy properties of failure detection. In Section 2.3.3, we list the eight classes of Failure Detectors based on the completeness and accuracy properties as identified by Chandra et Al[CT96]. We end our discussion with a probabilistic Failure Detector called Phi-Accrual Failure Detector[HDYK04]. We restrict our discussion to non-byzantine failure detection. Byzantine failure detection is a complex topic and out of scope for our work.

We assume a distributed system $s$ with a set of n non-byzantine processes $p = \{p_1, p_2,...,p_n\}$, where n $\geq$ 2 . We define a process $p_i \in p$ to be *faulty* if it experiences any one of more of the four modes of failures defined above. We define *failure* as the event at which $p_i$ becomes faulty. *Failure detection* is defined as the event at which $p_i$ is marked as suspected of being faulty.

We refer to the seminal work done by Chandra et al. [CT96] that lays out the different axes by which to judge the quality of a failure detector. These axes are *completeness* and *accuracy* properties of a failure detector. We provide only informal definitions and

give a summary of their contribution, curious readers are advised to refer to [CT96] for formal definitions and proofs. Processes in $p$ form a failure detection group wherein $p_i$ and $p_j$ exchange information about process failures within the group, $\forall\ p_i,\ p_i \in p$. Each $p_i$ is equipped with a local instance of the failure detector. The collective failure detection capabilities of the failure detection group is given by the global failure detector $\mathscr{F}$. Without loss of generality, we assume that $f = \{f_1, f_2,...,f_k\}$ are faulty processes and $c = \{c_{k+1}, c_{k+2},...,c_n\}$ are correct processes(i.e non faulty) within $S$.

### 2.3.1 Completeness Properties

**Definition 7.** *(Strong Completeness) A failure detector $\mathscr{F}$ is said to be strongly complete if $c_i$ permanently suspects $f_i$ as faulty, $\forall\ c_i \in c$ and $f_i \in f$*

**Definition 8.** *(Weak Completeness) A failure detector $\mathscr{F}$ is said to be weakly complete if $\exists\ c_i,\ c_j \in c$ and $f_s \in f$; $i \neq j$, such that $c_i$ has not permanently suspected $f_s$ whereas $c_j$ has permanently suspected all faulty processes in $f$*

In other words, if every failed process is correctly suspected to have been permanently failed by every correct process, then $\mathscr{F}$ is strongly complete. On the other hand, if there is even one correct process that has not permanently suspected each failed process as faulty, then $\mathscr{F}$ exhibits weak completeness. An important point to note that weak completeness implies that there exist at least one process that has successfully suspected each faulty process within the system, call it $c_j$. In essence, this means that a weak completeness can emulate strong completeness by having $c_j$ share the information about all failed processes within the failure detector group.

### 2.3.2 Accuracy Properties

Assume that a correct process $c_i \in c$ permanently suspects a faulty process $f_i \in f$ to have failed at time $c_{t_i}$. Further, assume that a faulty process $f_i \in f$ actually failed at time $f_{t_i}$. Let $T$ be the time domain.

**Definition 9.** *(Strong Accuracy) A failure detector $\mathscr{F}$ is said to have strong accuracy if $c_{t_i} \geq f_{t_i}$*

**Definition 10.** *(Weak Accuracy) A failure detector $\mathscr{F}$ is said to have weak accuracy if $\exists\ c_i \in c$ and $f_i \in f$ such that $c_{t_i} < f_{t_i}$*

In simpler words, if a faulty process is never suspected to have failed before it has actually failed, then $\mathscr{F}$ exhibits strong accuracy. On the other hand, if a faulty process is suspected to have failed before it actually failed, then $\mathscr{F}$ exhibits weak accuracy. Another way of looking at it would be that a strongly accurate Failure Detector would never suspect a correct process to have failed before it actually did, and a weakly accurate failure detector would suspect at least one correct process to have failed. Note that strong accuracy is a rather extreme property to hold true for a failure detector in practical scenarios. More over, the weak accuracy property does not allow for a wrongly suspected

correct process to be rectified at some point in the future. In order to relax these properties, Chandra et al. introduced the concept of *eventual accuracy*, that gives failure detectors more flexibility to *eventually* satisfy the accuracy properties.

**Definition 11.** *(Eventual Strong Accuracy) A failure detector $\mathscr{F}$ is said to eventually strongly accurate if $\exists\ t_0 \in T$ such that $\forall\ t \geq t_0$, $t \in T$, the strong accuracy property holds true*

**Definition 12.** *(Eventual Weak Accuracy) A failure detector $\mathscr{F}$ is said to eventually weakly accurate if $\exists\ t_0 \in T$ such that $\forall\ t \geq t_0$, $t \in T$, the weak accuracy property holds true*

Relaxing the accuracy properties by allowing them be eventually satisfied gives a degree of freedom to Failure Detectors. A Failure Detector $\mathscr{F}$ satisfies eventual strong accuracy property if after some point in time, it no longer suspects a correct process to be faulty before it actually failed. $\mathscr{F}$ satisfies eventual weak accuracy property if after some point in time, some correct process is never suspected.

Collectively, the properties of *completeness* and *accuracy* together aid in judging whether a failure detector detects actual faulty processes(completeness) and how well it avoids false detection(accuracy). The quality of the failure detector is therefore judged on the basis of the degree of *completeness* and *accuracy* that it satisfies. A perfect(strongly complete and strongly accurate) failure detector will suspect all faulty processes without any mistakes, whereas a weak failure detector(weakly complete and weakly accurate) will not suspect at least one faulty process and make numerous mistakes. Eventual failure detectors lie in between the spectrum of these extremes.

### 2.3.3 Eight Classes of Failure Detectors

Building on the definitions put above, Chandra et al. classify failure detectors into eight classes based on the completeness and accuracy properties that they satisfy. Below we list these classes and the degree of completeness and accuracy that holds true for them. The curious reader is advised to refer to the paper for formal definitions.

1. **Perfect Failure Detector** : Strong completeness and strong accuracy
2. **Eventually Perfect Failure Detectors**: Strong completeness and eventual strong accuracy
3. **Strong Failure Detectors**: Strong completeness and weak accuracy
4. **Eventually Strong Failure Detectors**: Strong completeness and eventual weak accuracy
5. **Weak Failure Detectors**: Weak completeness and weak accuracy
6. **Eventually Weak Failure Detectors**: Weak completeness and eventual weak accuracy
7. **Quasi-Perfect Failure Detectors**: Weak completeness and strong accuracy

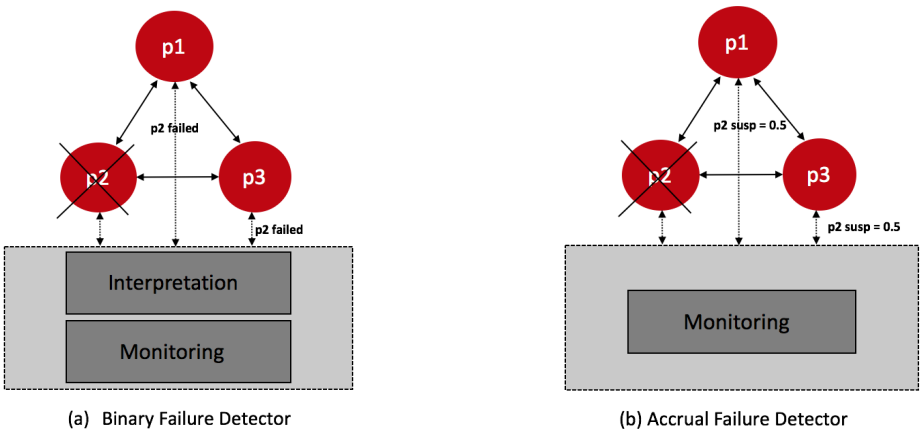



Figure 2.5: Binary and Accrual Failure Detectors

8. **Eventual Quasi-Perfect Failure Detectors**: Weak completeness and eventual strong accuracy

In practical scenarios, it should be acceptable for failure detectors to make initial mistakes. These mistakes can either be wrongly detecting a correct process to have failed or taking longer time to detect failure. These is what makes the Failure Detectors that satisfy the eventual accuracy property more practical, as by definition they can always adjust their mistakes and ensure that after a certain point in time mistakes in detection are not made and the speed of detection is quick.

### 2.3.4 Accrual Failure Detection

We now shift our attention to a type of eventually perfect failure detector called the Phi Accrual Failure Detector by Hayashibara et al. [HDYK04]. Binary failure detectors typically either tag a process to be correct vs suspect. An accrual failure detector, on the other hand, outputs a *suspicion level* on a continuous scale. What this does is that it decouples monitoring from interpretation. Whereas a binary failure detector will monitor processes and based on heartbeats sent at regular intervals by the processes it interprets whether the process is correct or faulty. An accrual failure detector allows monitoring to be separate from interpretation. Each process can react differently based on the *suspicion level* that is outputted by an accrual failure detector. In Fig 2.5, we see three processes *p1, p2 and p3* that exchange messages with each other. The dotted line represents *p1, p2 and p3* sending heartbeats at regular intervals to the failure detector. At a certain point in time, *p2* fails. In Fig 2.5(a), the binary failure detector(irregardless of implementation), that is monitoring all processes also interprets *p2* to have failed. It then sends a message to *p1 and p3* that *p2* has failed. On the other hand, the accrual failure detector in Fig 2.5(b) sends a *suspicion level* of 0.5 to *p1 and p3*. It is up to *p1 and p3* on how they interpret this suspicion value. This is a rather simplistic illustration, practical scenarios are more complex. However, the intuition behind binary and accrual

failure detectors should be understood.

This *suspicion level* is the degree of confidence that the process being monitored is faulty. For any process $p$ and time $t$, the $susp_p(t)$ is a probabilistic value that satisfies the following properties:

**Definition 13.** *(Asymptotic completeness) If a process f is faulty, then* $susp_p(t) \rightarrow \infty$ *as* $t \rightarrow \infty$*. In other words, if f is faulty, then as time increases, the degree of confidence that f has failed also increases*

**Definition 14.** *(Reset) If a process p is correct, then* $\exists\ t_0 \in T$ *such that* $susp_p(t) = 0$, $\forall\ t \geq t_0$*. In other words, if p is correct, then the failure detector should not suspect it at all*

**Definition 15.** *(Eventual monotonicity) If a process p is faulty, then* $\exists\ t_0 \in T$ *such that* $susp_p(t)$ *increases monotonically,* $\forall\ t \geq t_0$*. In other words, if p is faulty, then degree of confidence by which the failure detector can tag it as faulty will increase*

**Definition 16.** *(Upper Bound) If a process p is correct, then* $\exists\ r \in \mathbb{R}$ *such that* $susp_p(t) \leq r$*. That is, the degree of confidence of being faulty for a correct process p never goes above a certain threshold*

The property of asymptotic completeness and eventual monotonicity specify how an accrual failure detector should behave towards faulty processes, and the reset and upper bound properties specify how it behaves towards correct processes.

The $\Phi$ accrual failure detector outputs probabilistic estimate $\Phi$ that a process has failed based on the inter-arrival times of heartbeats received from other processes. Hayashibara et al. define the value of $\Phi$ using the equation [HDYK04] :

$$\Phi(t_{\text{now}}) \stackrel{\text{def}}{=} -\log_{10}(P_{\text{later}}(t_{\text{now}} - T_{\text{last}}))$$

Here, $t_{now}$ is the current timestamp and $T_{last}$ is the last timestamp at which the failure detector received heartbeat of the process being monitored. A Sliding Window of a fixed size is maintained, and the inter-arrival time of each heartbeat is stored in this window. As soon as a new heartbeat arrives, the oldest heartbeat is removed from the window to maintain the size. Two more variables, the mean $\mu$ and the variance $\sigma^2$ are maintained. The distribution of the inter-arrival times are assumed to follow the normal distribution. $P_{later}(t)$ is the probability that a given heartbeat will arrive t time units after the previous one and is given by :

$$P_{\text{later}}(t) = \frac{1}{\sigma\sqrt{2\pi}} \int_t^{+\infty} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx = 1 - F(t)$$

For a more complete understanding of the different parameters involved in Phi-Accrual Failure Detector, the curious reader is adviced to the work of Hayashibara et al[HDYK04].

## 2.4 Time and Order in a Distributed System

We continue with the same analogy of a distributed system $S$ with a set of n independent process $\{p_1, p_2,...,p_n\}$. The CAP theorem suggests that any distributed system can only chose a combination of two properties out of consistency(C), high availability(A) and partition tolerance(P) [CAP]. Consistency refers to strong consistency, which means that at point of time, all processes have the same view global view of the state. High availability is a property of a system wherein the system continues to function normally even if some processes have failed. A system is said to satisfy the property of partition tolerance if it continues to functional normally in the event of a communication failure between the processes. The communication failure can occur because of the underlying network channel that connects the processes, or the failure of a process to respond to another process because it has crashed. Therefore, as per the CAP theorem, $S$ would have to chose either CP(consistency and partition tolerance), AP(high availability and partition tolerance) or CA(consistency and high availability). Because of the constraints enforced by the CAP theorem, $S$ would have to forego one property out of consistency, availability and partition tolerance. Thus, we say that a system will either be AP, CP or CA. Abandoning partition tolerance can lead to catastrophic consequences for any distributed system, as per a study done by Alquraan et al.[ATAAK18] where they analyze the impacts of network partitions in a cloud based environment. Furthermore, by definition, in the absence of partition tolerance, a distributed system ceases to function as according to its specification. We restrict our discussion to AP systems only, and in the next chapter we will see how our work builds on these properties.

Even though an AP system is devoid of strong consistency, it still needs to guarantee a certain degree of consistency level. Modern AP systems relax the constraints enforced by the strong consistency model and instead offer a weaker form of consistency known as eventual consistency.

Without loss of generality, we assume that $C = \{c_1, c_2,...,c_k\}$ are correct processes(i.e non faulty) within $S$. We assume that each $c_i$ is a replica of $c_j$ is, in the sense that an update at $c_j$ is also replicated to $c_i$(without getting into the notion of *when* this replication takes place).

**Definition 17.** *(Eventual Consistency(EC)) A system $S$ is said to be eventually consistent if for an update $u$ occurring on a correct replica $c_i$, $\exists$ a point in time $t_0$ such that $u$ is eventually delivered to $c_j$, $\forall$ $c_i$, $c_j \in C$*

Note that the definition of eventual consistency says nothing about the state of the replicas once the update reaches the replicas. Even though the updates reach all correct replicas, an eventually consistent model of replication does not specify in what order those updates will be applied. In order to explain why this is so, we first need to understand ordering of events and the semantics of time in a distributed systems. In his seminal work on logical clocks [Lam78], Leslie Lamport observed that it is not always possible for a distributed system to say if one of the two events happened first in the absence of a global clock. At best, a distributed system with no global clock can only guarantee a *partial*

*ordering* of events. This kind of ordering is best defined explained by a happened-before relation. Lamport defines a happened-before as follows :

**Definition 18.** *(Happened-Before Relation) Let a and b be two events in that occur in S. Then, $\rightarrow$ is a happened-before relation that satisfies the following properties*

1. *If a and b occurred in the same process $c_i$, then $a \rightarrow b$ if occurrence of a preceded the occurrence of b*
2. *If a is the sending of a message from $c_i$ and b is the receiving of the message on $c_j$, then $a \rightarrow b$*

Logical clocks build on the happened-before relation to provide a partial ordering of time in a distributed system. We now define what it means for two events to be concurrent

**Definition 19.** *(Concurrent Events) Let a and b be two events in that occur in $c_i$ and $c_j$ respectively. Then, a and b are concurrent with one another if $a \not\rightarrow b$ and $b \not\rightarrow a$*

Going back to our original example, two replicas $c_i$ and $c_j$ that follow the eventually consistent mode of replication might exchange updates in which it is not possible to determine which update happened-before the other. In other words, there will always be some updates in which a causal order between these updates cannot be established, i.e they are concurrent. To simplify our example a bit more, we can assume that $c_i$ receives a update from $c_j$ that is concurrent to the updates that are already part of $c_i$, and vice versa. In such a scenario, without coordinating with one other, both $c_i$ and $c_j$ might end up having a conflicting ordering of these events. If these replicas are required to run a computation independently, the result of the computation will be different on both replicas as there was a conflict in the ordering of updates. We conclude that in the absence of a global clock, a distributed system has to rely on partial ordering to order events. Partial ordering can lead to concurrent events, in which it is not possible to state which event happened-before the other. And this in turn leads to conflicts between replicas.

## 2.5 Conflict Free Replicated Data Types

In this section, we discuss an important breakthrough done by Shapiro et al.[SPBZ11] called Conflict Free Replicated Data Types(CRDT). As the name suggests, these are distributed data types that are replication aware and free of conflicts. In this section, we define a CRDT and in subsequent Sections we discuss the different types of CRDT's and the underlying assumptions on which they are based upon. We will only be giving informal definitions, the curious reader is advised to refer to the work of Shapiro et al. [SPBZ11] [SPM+11] for formal definitions and proofs.

CRDT's are designed on a consistency model that Shapiro et al. call Strong Eventual Consistency(SEC). We first define what it means for a system to be strongly convergent.

**Definition 20.** *(Strong Convergence) A system is Strongly Convergent if for any two replicas $c_i$, $c_j \in C$, after applying the updates they received locally and also replicated to one another, the final state of $c_i$ and $c_j$ is equivalent*

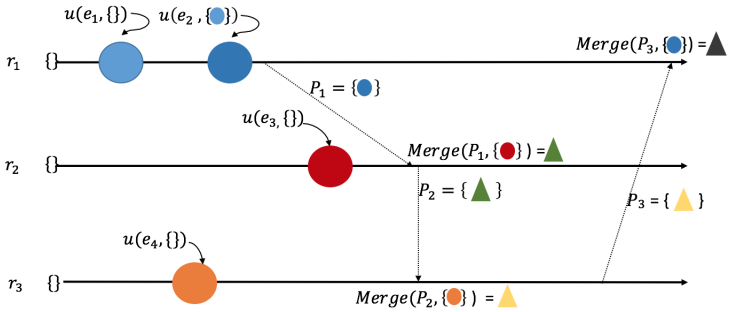



Figure 2.6: State Based CRDT

**Definition 21.** *(Strong Eventual Consistency) A system is Strongly Eventually Consistent if it provides Eventual Delivery of messages and is Strongly Convergent*

Informally, a CRDT is a distributed data structure comprising of replicas where:

1. Replicas observe updates that they receive locally and from other replicas
2. Replicas can be updated concurrently without any synchronization
3. It is mathematically possible to resolve any inconsistencies that arise between the replicas. In other words, they are conflict free

For the remainder of this section, we use the following terminologies : Let $S$ be a distributed object comprising of n correct replicas(i.e none of them are fauly) $R = \{r_1, r_1,....,r_n\}$. Let $X$ be any client that sends updates or queries $S$ . Each replica $r_i$ is equipped with a local state, called $Sr_i$. A replica $r_i$ can either receive updates that were sent directly to it by $X$(call it *local updates*) or a series of updates from another replica $r_j$ that are replicated to it(call it *replicated updates*). Any query by $X$ is executed against a correct replica $r_i$, and this query is performed against the local state $Sr_i$. Any update by $X$ on $r_i$ that needs to be replicated to $r_i$ is done asynchronously. Each $r_i$ and $r_j$ form a bi-directional replicas, i.e $r_i$ is a replica of $r_j$ and vice versa, $r_i, r_j \in S$.

### 2.5.1 State Based CRDT

In a state based CRDT, the entire local state of $r_i$ is replicated to $r_j$, periodically and asynchronously. In other words, the *replicated update* contains the entire local state of the replica. The underlying assumption of state based CRDT's is that the network is a fair-lossy channel. A fair-lossy channel is a weak channel that guarantees that if a sender sends a message infinite number of times to the receiver, the message will eventually be delivered.

We first look at Fig 2.6 to understand the intuition behind state based CRDT's. We illustrate a simple scenario with three replicas *r1, r2, r3*. The initial local state of these replicas is empty, denoted by {}. Whenever a replica receives an event, using that event the update function performs a state transition. When *r1* receives its first event, it applies the update function $u$, and the state transitions from {} to the ball in light

blue. *r1* receives yet another event, but this time the update function $u$ is applied on the previous state(ball in light blue), and the state transitions to the ball in dark blue. The replicas *r2, r3* also receive initial event and using the same logic, their local state transitions from empty to the balls in red and orange respectively. Next, *r1* decides to replicate it's local state(currently the ball in dark blue) to *r2*. Here, $P_1$ is the payload that encapsulates the current local state of *r1*. Upon receiving $P_1$, *r2* merges the remote state of *r1* with its own local state, transitioning to a new local state given by the green triangle. The same mechanism of replication takes place between *r2* and r3, with *r3* transitioning to a new local state denoted by the yellow triangle. Finally, *r3* replicates it's current local state to *r1*, thereby resulting in a state transition on *r1* to the triangle in black. The fact that each replica sends as payload it's entire current state over a fair-lossy channel is central to the idea of state based CRDT. More over, a state based CRDT is equipped with a merge function that merges two states and transitions to a new one. The merge function computes the *least upper bound* of the the local state and the replicated state. Furthermore, the merge function is also commutative, associative, and idempotent. Idempotency guarantees that if a replica $r_i$ receives a payload that is already equivalent to its local state, then it performs no state transitions. We formally define a state based CRDT as :

**Definition 22.** *(State Based CRDT) A state based CRDT of n replicas is a quintuple* $(\Lambda, \subset, \cup, u, q)$ *such that:*

1. $\Lambda$ *is a join semi-lattice of states*
2. $\subset$ *is a partial order on* $\Lambda$
3. $\cup$ *is a merge function that derives the least upper bound of two states in* $\Lambda$
4. $u$ *is the update that occurs locally on an individual replica*
5. $q$ *is the query that is either a direct view over the local state or a computation over the local state of a replica*

In summary, a state based CRDT is a distributed object $S$ equipped with an update function $u$, a query function $q$ and a merge function $m$. The update function is applied on the on a replica locally when it receives an event, resulting in a state transition. The merge function $m$ is applied whenever a replica receives as payload the state of another replica. The query $q$ can be initiated by a client at any point in time against any individual replica. This replica then will compute the query against its local state and output a result.

A clear disadvantage of state based CRDT is that the performance suffers when the size of local state of replicas increases. Infact, we will see in the upcoming section how Delta CRDT's solve this problem by decomposing the payload state into small deltas, and then replicating the deltas instead of the entire replica state. We now illustrate a concrete state based CRDT called GCounter to put the concepts explained above into perspective.

**GCounter**

**Algorithm 1** State Based GCounter Specification

**payload** : (P : Array[Integer]) ▷ of size n
  initial : [0,0,...,0]
**update** : incrementByOne()
  let id = *getId()*
  Set P[id] := P[id] + 1
**query** q : Return Integer v
  let v = $\sum_i$ P[i]
**merge** :(X: Payload, Y: Payload) : Returns Payload Z
  $\forall i \in [0, n-1] : Z.P[i] = \max(X.P[i], Y.P[i])$

A GCounter is a distributed counter that only supports one operation : incrementing a counter by one[SPBZ11]. Assume that our CRDT has n replicas.

The payload $P$ is the local state of the replica that is to be disseminated. The array is a version vector, where each index is an identifier for a specific replica. For instance, a very simple way of indexing replica identifiers is to assign an integer value to each replica, i.e replica 1 can be at index 0, replica 2 at index 1 and so on. The state of each replica is an array of size n with only zero as values. The update function is executed locally on each replica, and increments the counter of that replica by one. Note that as far as the update operation executed against a replica $r_j$ is concerned, only the value of this replica in the array is incremented. The query $q$ is executed against one replica, and simply sums the value in the array. The result of the query is the current count of the GCounter. The merge function takes two payloads : $X$, which is the payload sent by a replica, and $Y$, which is the current local state of the replica itself. Further, note that in the specification, we label both $X$ and $Y$ as a Payload. This is because, as stated before, in a state based CRDT, the local state of the replica is also the Payload that is disseminated to other replicas. In other words, the Payload and local state are one and the same. The merge function loops through both arrays simultaneously, computes the maximum at each index, and outputs an array $Z$, where each value at index $i$ in Z is the maximum of values of the arrays $X$ and $Y$ at that index.

### 2.5.2 Operation Based CRDT

Operation based CRDT's are the opposite of state based CRDT's. Whereas a state based CRDT ships the entire local state of a replica that is obtained after applying updates locally, an operation based CRDT is based on dissemination of operations. The update operations that are received by a replica $r_i$ are applied locally on the replica, and then shipped to other other replicas. The communication channel between the replicas is assumed to be stronger than the fair-lossy channel in state based CRDT's. In fact, the communication channel between the replicas should be capable of performing reliable causal broadcast [BAS17]. Having a middle-ware capable of reliable causal broadcast ensures that the causality between events is maintained after they are replicated. For example, assume that event *e1* occurred at $r_1$ and event *e2* occurred at $r_1$ and that *e1*

**Algorithm 2** Delta Based GCounter Specification

**payload** : (P : Array[Integer]) ▷ of size n
  initial : [0,0,...,0]
**update** : incrementByOne()
  let id = *getId()*
  Set P[id] := P[id] + 1
**$\delta$-mutator** : Return a (index, value) pair $\delta$
  let $\delta$ = { i → P[i] + 1 }
**query** q : Return Integer v
  let v = $\sum_i$ P[i]
**merge** :(X: DeltaGroup, Y: CurrentState) : Returns Payload Z
  let Z = $\{k \mapsto \max(X[k], Y[k]) \mid k \in DeltaGroup.Ids\}$

happened-before *e2*. If both $r_1$ and $r_2$ ship the events to $r_3$, $r_3$ can apply the update function with respect to the ordering of *e1* and *e2*, that is first on *e1* and then on *e2*. However, as we discussed in Section 3.2, it is not always possible to guarantee the happened-before relation between two events, i.e events may be concurrent. In order to circumvent the conflicts that can arise due to concurrent events, operation based CRDT's mandate that the update function also be commutative. As individual operations are disseminated to other replicas, operation CRDT's can lead to increased network traffic. More over, a network channel that guarantees exactly-once causal broadcast is not easy as large logs need to be maintained [ASB16].

As Fargo is based on a fair-lossy channel, we will not go in depth of operation based CRDT's.

### 2.5.3 Delta CRDT

If state based CRDT's are on one end of the spectrum, and operation based CRDT's on the other, then Delta CRDT lie somewhere in between. Both state based and operation based CRDT's are part of the original work done on CRDT's by Shapiro et al. [SPBZ11]. As we have explained already, the major pain point of a state based CRDT is that the entire state of a replica is disseminated to other replicas. In practical scenarios, it is not feasible to replicate an ever-growing state. Delta CRDT's are an optimization on top of state based CRDT's. A Delta CRDT defines what is called $\delta$- mutators that decompose the state into small deltas, typically a size much smaller than the full state[ASB16]. These deltas are then grouped together into a $\Delta$- group and the $\Delta$- group is shipped to other replicas, where they are merged with the local state of those replicas. Shipping a $\Delta$- group proves to be advantageous as multiple smaller $\delta$- mutators can be collapsed into one $\Delta$- group. Continuing with the same terminologies introduced in the definition of state based CRDT's, we formally define $\delta$- mutators.

**Definition 23.** *(Delta-mutator) Corresponding to an update function u, a $\delta$ mutator f : $\Lambda \to \Lambda$ is a function f($\lambda$) = $\delta$ such that $\lambda = \lambda_* \cup \delta$, for some $\lambda_* \in \Lambda$*

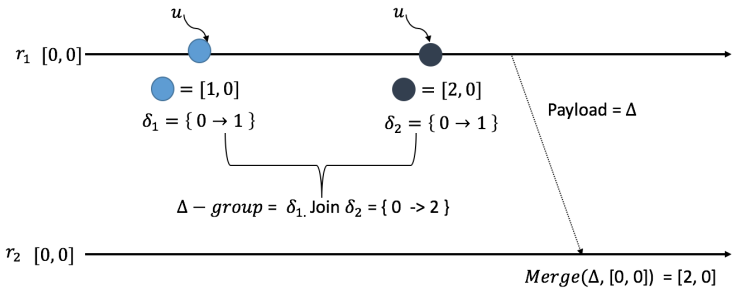



Figure 2.7: Delta Based GCounter with 2 replicas

A $\delta$- group is then a join of a single or multiple $\delta$- mutators. We now present a Delta Based GCounter.

**Delta Based GCounter**

The only new concept that is needed to be added to the state based GCounter specification is a $\delta$-mutator. This new concept further modifies the semantics of payload and the merge operator[ASB16].

As we see in Algorithm 2, a $\delta$-mutator for a GCounter is a single (key, value) pair, where the key is the index of the replica, and the value is the updated value at that index(i.e incremented value by one). Since the synchronization between replicas takes place periodically, a delta CRDT utilizes this time period to locally join all $\delta$-mutators into one $\Delta$-group. For instance, let us assume a scenario of a delta based GCounter with 2 replicas called *r1, r2*, as shown in Fig 2.7. The initial state of both replicas is marked as [0, 0]. For the sake of simplicity, we assume that the index for *r1* is 0 and for *r2* is 1. Assume further that *r1* receives 2 increment operations. As *r1* receives the first increment $u$, it's local state transitions to [1, 0]. The corresponding $\delta$-mutator for this particular update is the pair $\{0 \rightarrow 1\}$. Upon receiving an update again, the state transitions to [2, 0] and we get yet another $\delta$-mutator, same as before. We build a $\Delta$-group by simply joining the two $\delta$-mutators by adding their values at the corresponding index. In this particular case, $\Delta$-group becomes $\{0 \rightarrow 2\}$. This $\Delta$-group is then shipped to *r2*, and *r2* merges it with its local state, which is empty, yielding a state transition on *r2* to [2, 0].

### 2.5.4 Optimized Delta CRDT

In this Section, we will discuss the work done by Enes et al.[EABL18] that further optimizes Delta CRDT's. Enes et al. observed that synchronization algorithms used by Delta CRDT's can still disseminate much redundant state between replicas. This leads to a performance that, with time, is no better than state based CRDT's. Their approach identifies the sources of inefficieny of these synchronization algorithms and introduce the concept of *join decomposition* to obtain optimal $\Delta$-groups. Each $\Delta$-groups, before being propagated to other replicas, is added to a $\Delta$-buffer.

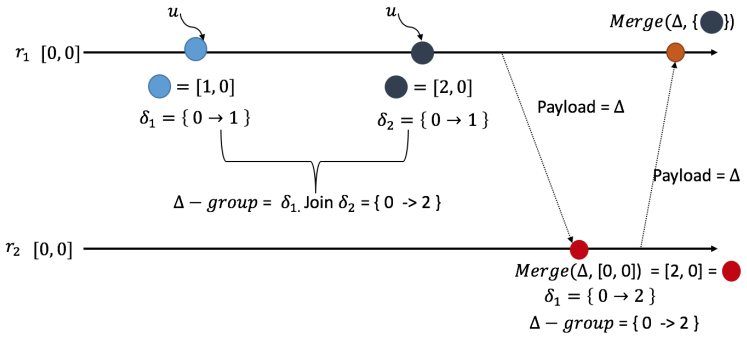



Figure 2.8: BP Optimization

The key idea behind their work can be broken down into three concrete steps : Using join decompositions to obtain optimal $\delta$-mutators and $\Delta$-groups, avoiding back-propagation of $\Delta$-groups(*BP optimization*), removing redundant state in received $\Delta$-groups(*RR optimization*).

### BP Optimization

The BP optimization, short for *avoid back-propogation of $\Delta$-groups* simply means that a replica that has already disseminated a $\Delta$-group to other replicas, should not receive it back. We continue with our illustration on delta based GCounter in Section 3.3.3. In Fig 2.8, *r2* receives a $\Delta$-group from *r1* with the value $\{0 \rightarrow 2\}$. After updating its local state obtained through the merge of the received $\Delta$-group with its own local state, *r2* then decides to synchronize again with *r1*. This time, it sends the $\Delta$-group with value $\{0 \rightarrow 2\}$ back to *r1*. However, this $\Delta$-group had already been propagated from *r1* to *r2* during their previous synchronization.

To avoid propagating $\Delta$-group back to their origin, the optimization Enes et al. proposed is as follows : each entry in the $\delta$-buffer is tagged with its origin. Whenever $r_i$ would need to synchronize with $r_j$ again, it would simply filter out all the entries in the $\delta$-buffer for replica $r_j$. In Fig 2.7, had the BP optimization been used, *r2* would tag the $\Delta$-group it received from *r1* in its $\delta$-buffer. The second synchronization shown in Fig 2.7 would thus, never take place.

### RR Optimization

The RR optimization is short for *removing redundant state in received $\Delta$-groups*. The intuition behind the idea is that a received $\Delta$-groups might contain state that has already been propagated to neighbouring replicas, or is already part of the $\Delta$-groups that is being propagated.

To understand this optimization better, we illustrate in Fig 2.9 a scenario of a delta based GCounter with four replicas *r1, r2, r3, r4*. The initial states of all replicas are [0, 0, 0, 0]. We assume the index of *r1* is 0, for *r2* is 1, for *r3* is 2 and for *r4* is 3 . *r1*

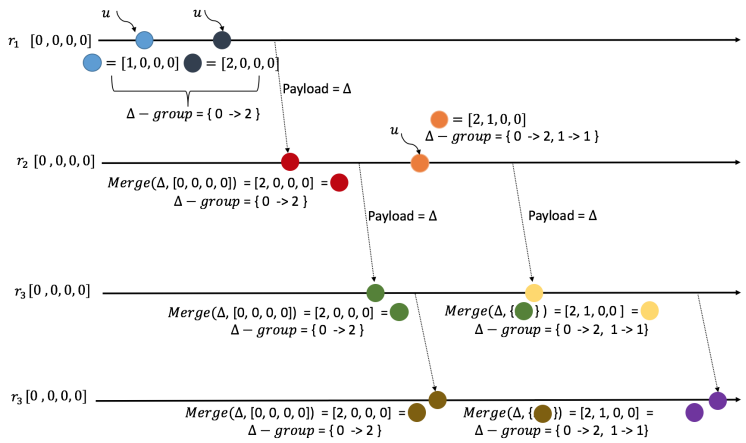



Figure 2.9: RR Optimization

receives two updates, and increments the counter twice, transitioning to a state in the ball in dark blue with value [2, 0, 0, 0]. The Δ-group is also maintained, the current value at *r1* being the key value pair $\{0 \rightarrow 2\}$. This Δ-group is propagated to *r2*, resulting in transition of state to [2, 0, 0, 0] on *r2* and *r2* further adds the received Δ-group to its own local Δ-buffer. *r2* then replicates its Δ-buffer to *r3*. In turn *r3* transitions to [2, 0, 0, 0] and adds the received Δ-group to its own Δ-buffer. As *r2* receives another update, it transitions to the ball in orange with value [2, 1, 0, 0], and adds the value of $\delta$-mutator $\{1 \rightarrow 1\}$ to it's existing Δ-buffer. The current value of the Δ-buffer of *r2* is now $\{0 \rightarrow 2, 1 \rightarrow 1\}$. *r3* initiates a synchronization with *r4*, and propagates it's Δ-buffer, making *r4* transition to [2, 0, 0, 0]. *r2* initiates yet another synchronization round with *r3*, this time shipping it's Δ-buffer with value $\{0 \rightarrow 2, 1 \rightarrow 1\}$ to *r3*. Finally, *r3* initiates a second synchronization round with *r4*, this time propagating the values $\{0 \rightarrow 2, 1 \rightarrow 1\}$. Note that in the second synchronization round between *r3* and *r4*, *r3* ships the value $\{0 \rightarrow 2\}$ yet again to *r4*, even though *r3* had done so already in the first synchronization round. In other words, *r3* has shipped as part of its Δ-buffer a redundant state that had already been propagated to *r4*.

This redundancy in state is what the RR optimization solves. If the optimization holds true, then *r3* will never send the value $\{0 \rightarrow 2\}$ again to *r4*, as that value had already been propagated from *r3* to *r4*. The RR optimization solves this problem as so: when *r3* receives the Δ-group of $\{0 \rightarrow 2, 1 \rightarrow 1\}$, it compares this Δ-group against it's own local state. It then computes the difference between the two states. If there is a difference, that difference is what inflates the local state, and this difference is added to the Δ-group. In our example, on *r3*, the difference between $\{0 \rightarrow 2, 1 \rightarrow 1\}$ and $\{0 \rightarrow 2\}$ is $\{1 \rightarrow 1\}$. With RR optimization, *r3* adds $\{1 \rightarrow 1\}$ to it's Δ-Buffer and propagates it down to *r4*.

# 3 Fault Tolerant In-Network Distributed Window Aggregations

In this Chapter, we present Fargo, a SPE for fault tolerant in-network distributed window aggregations that can be applied for fog computing scenarios. This poses a few challenges. First, fog nodes are very limited in terms of computational capacities, be it networking, CPU, memory or storage. Second, as we are in a streams processing setting and thus interested in real-time analysis of events, the latency with or without node failures must be absolutely minimal. Lastly, in case there are multiple node failures, the throughput of the system should not be impacted. Thus, the topology and protocols that we present in this chapter are modelled around hardware having very low computational capacities and a weak network channel connecting different processes of our system. To design a performant distributed SPE under such constraints is quite challenging with each decision leading to multiple trade-offs. Therefore, we desire a distributed SPE that can split a computation into parallel sub- computations, isolate node failures, react to node failures swiftly and accurately and ensure minimum to no impact on latency is the right way to go about approaching this problem. As will be seen in the upcoming sections, the different components in our topology are autonomous and participate in protocols that ensure not only low Window latency, but also lightweight fault tolerance.

Fargo is modelled as a k-partite graph where each partition of the graph is called a Group. Each Group contains nodes that we call components, with each Group serving a different purpose. Fargo supports data parallelism across different Groups in its topology, and distributes Algebraic Aggregations across multiple nodes. To achieve fault tolerance, Fargo uses Phi-Accrual Failure Detector as a flexible and network-adaptive Failure Detector to detect failures, and uses Conflict-Free Replicated Data Types(CRDT's) to replicate $\Delta$-groups between replicas using a Peer-to-Peer Full Mesh Topology. These $\Delta$-groups are the partial aggregations that Child Nodes have computed.

In this chapter, in Section 3.1, we first start with the Fargo DataFlow Topology and lay out the assumptions upon which it is based. In Section 3.2 we discuss the high level architecture of Fargo. In Section 3.3, we showcase the various components of Fargo's topology, namely Sensors, Child Nodes, Intermediate nodes and Root node. We adapt these terminologies from the work done by Benson et al.[Dis] in their SPE called Disco. In Section 3.4 we discuss the various protocols employed by Fargo that perform various orchestrations and ensure lightweight fault tolerance. Finally, in Section 3.5, we provide CRDT's for multiple Algebraic Aggregations, and formally prove that our data structure for Sum Aggregations is a state-based CRDT. We end this chapter with Section 3.6, where we argue how Fargo provides aggregate correctness in the face of nodes failure, and the circumstances in which it provides fault tolerance with atleast-once-delivery semantics.

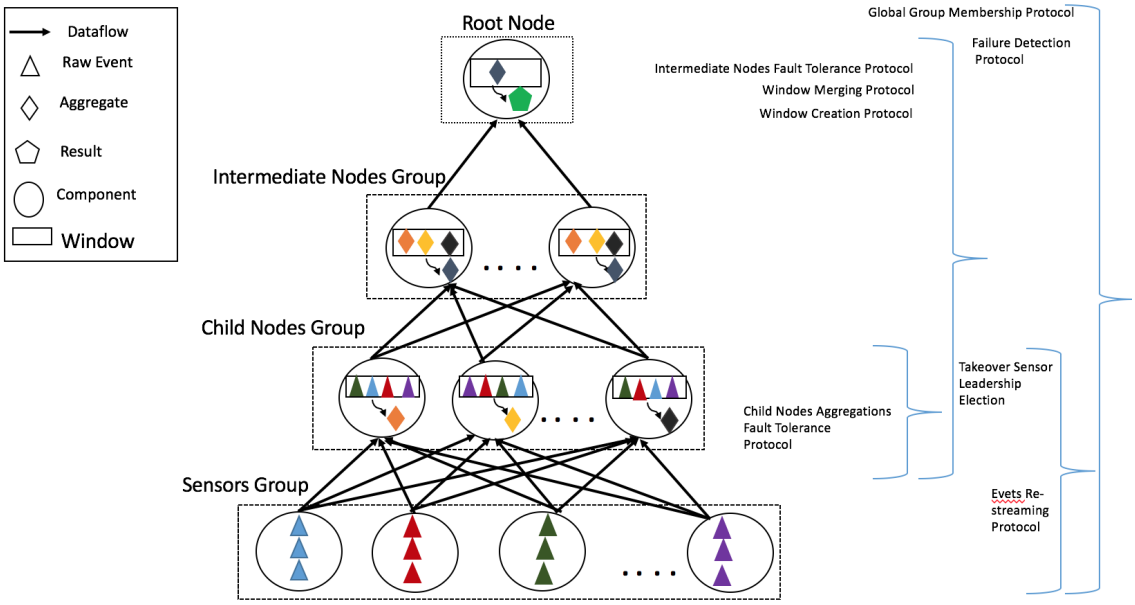



Figure 3.1: Fargo Dataflow Topology

## 3.1 Fargo DataFlow Topology

In this section, we discuss the dataflow of Fargo. We will discuss the components only briefly in this section. In the upcoming sections, we will explain in detail the different Groups and components, as well as the protocols that are followed by these different groups and components.

The dataflow of Fargo is a k-partite graph, wherein each partition of the graph is called a Group. Each Group is made up of components that serve a particular purpose. Different Groups or components within a Group participate in different protocols. The bottom-most Group of the graph is called a Sensor Group and is composed of Sensors. The Group right above the Sensor Group is called the Child Nodes Group, comprising of multiple Child Nodes. The Group above the Child Nodes Group is either the Intermediate Nodes Group or a Root Node. The Intermediate Nodes Group comprises of Intermediate Nodes. The Intermediate Nodes Groups are optional. The topology might have zero Intermediate Node Groups or more than one. We now formally define Fargo DataFlow Topology:

Definition

**Definition 24.** *(Fargo DataFlow Topology) Fargo DataFlow Topology(FDT) is a sextuple* $(\mathcal{G}, \mathcal{P}, \mathcal{R}, \mathcal{S}, \mathcal{CN}, \mathcal{IG})$ *such that:*

1. $\mathcal{G}$ *is k-bipartite graph where each partition is called a Group. The top-most partition is at said to be at the kth position, the one below that at (k-1)th and so on. The bottom-most partition is said to be at position 1*

2. *The k-th partition consists of a single node called the Root Node, represented by* $\mathcal{R}$
3. *The bottom-most partition, i.e the partition at k = 1 consists of nodes called Sensors. The entire partition is called Sensors Group and denoted by* $\mathcal{S}$
4. *The partition at k = 2, or the second partition consists of Child Nodes and the entire partition is called Child Nodes Group. It is represented by* $\mathcal{CN}$
5. *Each partition from the third partition on wards and below* $\mathcal{R}$ *is called Intermediate Nodes Group. Each node in this Group is called an Intermediate Node. Each of these partitions is optional in the topology. A single Intermediate Nodes Group is represented by* $\mathcal{IG}$
6. $\mathcal{R}$, $\mathcal{S}$ *and* $\mathcal{CN}$ *are mandatory in the topology*
7. $\mathcal{P}$ *are a set of Protocols that the different Groups or different components within a Group in* $\mathcal{G}$ *participate in*
8. $\mathcal{S}$ *and* $\mathcal{CN}$ *form a complete bipartite graph*
9. *If* $\mathcal{IG}$ *exists in the topology,* $\mathcal{CN}$ *and* $\mathcal{IG}$ *form a complete bipartite-graph. Further,* $\mathcal{IG}$ *and* $\mathcal{R}$ *form a bipartite-graph*

In Fig 3.1, we see the Fargo's Dataflow Topology(FDT) in action. The bottom-most Group contains Sensors that emit events upstream. The edge between each Group denotes the direction of data flow. Each triangle in the Sensor denotes an event. As a scenario, assume we have a Sensor $s$ in $\mathcal{S}$ that emits events *e1, e2, e3...*, that are consumed by a Child Node $\text{cn}_i$ in $\mathcal{CN}$. The sensor $s$ is connected to each $\text{cn}_i$ in $\mathcal{CN}$. The Sensor $s$ uses a partitioning scheme to send events to the upstream $\text{cn}_i$. Fargo does not impose any restrictions on the partitioning scheme. Without loss of generality, from here on forth, and unless stated otherwise, we assume that the partitioning scheme is round-robin. This means that if $s$ produces events *e1, e2, e3...*, and $\mathcal{CN}$ comprises of k number of Child Nodes, say $cn_1$, $cn_2$,...,$cn_k$, then $s$ sends *e1* to $\text{cn}_1$, *e2* to $\text{cn}_2$ and so on and so forth. Thus, each Child Node $\text{cn}_i$ is responsible for a partition of data emitted by $s$. This enables data parallelism, and ensures that one sensor does not overwhelm the computational capacities of a Child Node. An event consumed by $\text{cn}_i$ belongs to some time-based Sliding Window that has already been instantiated on $\text{cn}_i$. In Fig 3.1, the rectangle inside each Child Node represents a time-based Sliding Window. The result of the aggregation emitted by the Sliding Window is depicted by the quadrilateral. This aggregation is incremental in nature, as defined in the section on Incremental Aggregations in Chapter 2. If we assume that the start and end time of the window is denoted by $S_{start}$ and $S_{end}$, then this aggregated output is emitted right after $S_{end}$. This way, each $\text{cn}_i$ is responsible for a *partial aggregate* that is merged further upstream by each $\text{ig}_i$ in $\mathcal{IG}$ or $\mathcal{R}$ itself. Each $\text{cn}_i$ is connected to each $\text{ig}_i$ forming a complete bi-partite graph between the two Groups. In other words, there exists a directed edge from each $\text{cn}_i$ to each $\text{ig}_i$. If there is even one $\mathcal{IG}$ present in Fargo's Topology, the *partial aggregate* emitted by $\text{cn}_i$ is sent to each $\text{ig}_i$. Just like Child Nodes, each Intermediate Node already has an instantiated time-based Sliding Window. However, these Windows only *merge* different *partial aggregates* emitted by downstream Child Nodes. However, if the Fargo

Topology does not have any $\mathcal{IG}$, the partial aggregates outputted by each $\text{cn}_i$ in $\mathcal{CN}$ are sent directly to $\mathcal{R}$, where they are merged. The final result of the query is always outputted in $\mathcal{R}$, denoted by the green polygon. The green polygon is the true global aggregate.

There are a set of protocols that either different Groups or each component within a Group participate in. Some protocols in the topology are meant for fault tolerance, while others are meant for creating and merging Sliding Windows. We now give a brief on these protocols, and in Section 3.4, we go in great detail explaining how each of them work. On the level of the entire graph $\mathcal{G}$, we have the *Global Group Membership Protocol*, that describes the various orchestrations performed by each Group to be discoverable by the Group directly above it or each component within a Group to discover each other. The *Window Creation Protocol* outlines how the windows will be created on each node in $\mathcal{CN}$, $\mathcal{IG}$ and $\mathcal{R}$. The *Intermediate Nodes Fault Tolerance Protocol* dictates the steps taken by Fargo to ensure that fault tolerance is achieved in the event of an Intermediate Node crashing within a $\mathcal{IG}$. The *Window Merging* protocol are steps taken by Fargo to merge partial aggregates emitted by an instance of a Sliding Window running on each $cn_i$ in $\mathcal{CN}$. At the level of $\mathcal{CN}$, we have two protocols, namely *Failure Detection Protocol* and *Child Nodes Aggregations Fault Tolerance Protocol*. The *Failure Detection Protocol* outlines how Failure Detection takes place within each Child Node in $\mathcal{CN}$, and *Aggregations Fault Tolerance Protocol* outlines the steps taken by each Child Node to make the aggregations themselves are fault tolerant. Between $\mathcal{CN}$ and $\mathcal{S}$, there exists two protocols, namely : *Takeover Sensor Leadership Election*, and *Events Re-streaming Protocol*. The *Takeover Sensor Leadership Election* is the protocol that is rendered when any Child Node crashes. Since each Child Node is responsible for partition of the data emitted by each Sensor, in the event of a failure of a Child Node, this protocol ensures that a different Child Node take ownership of the partition that the failed Child Node was responsible for. In other words, the new Child Node becomes the leader of the partitions that the failed Child Node was consuming. The *Events Re-streaming Protocol* outlines the steps taken by the the new leader of the partition to re-stream the events from the partitions of different sensors that were consumed by the failed Child Node.

## 3.2 High Level Architecture

In this section, we describe Fargo's high level architecture and the various assumptions upon which it is built.

The topology of Fargo is a union of one or more Fargo DataFlow Topologies. We finally formally define Fargo Topology as :

**Definition 25.** *(Fargo Topology) Fargo Topology(FT) is a triplet ($\mathcal{G}$, FDTs, $\mathcal{R}$) such that:*

1. *FDTs are one or more distinct Fargo DataFlow Topologies that share the Root Node $\mathcal{R}$*
2. *$\mathcal{G}$ is a directed acyclic graph that is a union of FDTs*

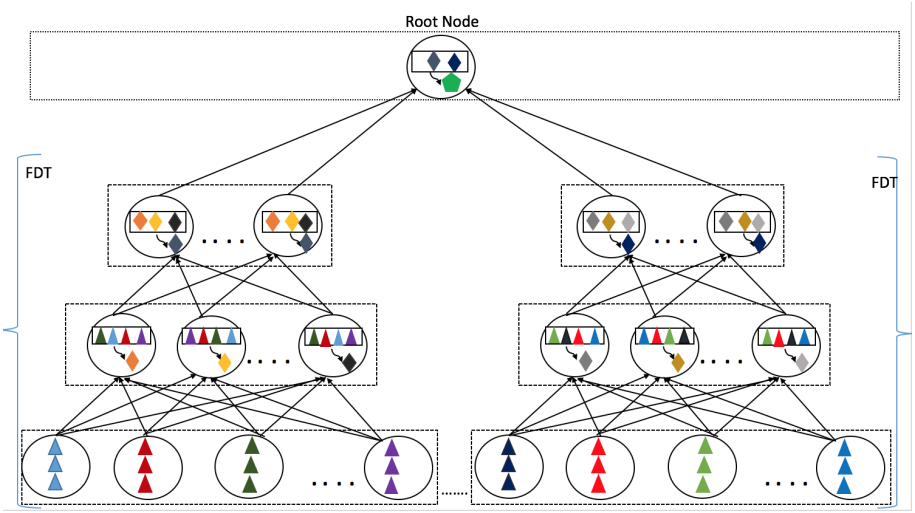



Figure 3.2: Fargo High Level Architecture

In Fig 3.2, we see Fargo's Topology. Note that each $\mathcal{IG}$ in the two distinct FDT's yield a quadrilateral of different colours. The Root Node merges these two partial aggregates produced by two distinct FDT's into the true global aggregate, shown as the green polygon. As a motivation as to why Fargo is based on the union of distinct FDT's, consider the scenario where we have a smart city where a temperature reading sensor is located in each house, and that there are a total of ten thousand houses in the city. We are interested in finding the average temperature reading across all houses broken down into Tumling Windows with a size of 30 seconds. Further, assume that the city is divided into ten zones, each zone having it's own fog infrastructure. We can deploy a FDT in each zone, and each FDT starts to read the temperature readings from the sensors independently. Each FDT yields it's share of the true global average, and the Root Node continually merges the partial results outputted by each FDT into the true global average.

### 3.2.1 System Assumptions

In this Section, we discuss the various assumptions upon which Fargo is designed. The architecture assumes that the communication channel between $\mathcal{CN}$ and $\mathcal{IG}$, and between $\mathcal{IG}$ and $\mathcal{R}$ is a fair-lossy. Thus, messages exchanged between different processes in the system can be dropped or duplicated. The point of interest of this research is to focus on fault tolerant distributed aggregations over a weak network channel. Thus, we are only interested in investigating how to make aggregations across $\mathcal{CN}$ and $\mathcal{IG}$ fault tolerant. The Sensors Group $\mathcal{S}$ and $\mathcal{R}$ are assumed to be highly available services and we only impose specifications on them, any fault tolerance aspects of $\mathcal{S}$ and $\mathcal{R}$ is outside the scope of this work. Between $\mathcal{CN}$ and $\mathcal{S}$, we do not pose a restriction on the underlying network channel and assume that it is strong. In other words, we assume that an event

that is emitted by a Sensor will always reach $\mathcal{CN}$ without a Child Node having to send an acknowledgement back. More over, we do not consider other aspects that a modern SPE deal with, such as backpressure or out-of-order streams. Out-of-order streams can handled using Watermarks as explained in Chapter 2. The Root Node $\mathcal{R}$, although part of Fargo's Topology, is assumed to be a black box that conforms to some specifications. First, $\mathcal{R}$ is assumed to be a highly available at all times. Second, the windowing queries are registered directly at $\mathcal{R}$, and it is $\mathcal{R}$ that makes the queries available for downstream Groups. The scalability and fault tolerance of $\mathcal{R}$ is out of scope for this work. We also do not enforce any restrictions where $\mathcal{R}$ exists, be it the cloud or part of the fog infrastructure itself. Next, we assume that the different kinds of components in Fargo will be deployed on hardware that are very limited in terms of computational power, storage, memory and network capabilities. This mimics fog nodes in fog infrastructures. Although we don't speak about physical query plans in our work, we do work under the assumption that the physical environment in which Fargo will be deployed is highly constrained. Furthermore, we focus strictly on non-byzantine fault tolerance(non-BFT). BFT is a complex subject in and of its own and out of scope of this work. Furthermore, Fargo only focuses on Sliding Windows and Algebraic Aggregate Functions. Holistic Aggregates are not considered, and are left for future work.

As explained in Chapter 2, Sliding Windows are created using a *size* parameter and a *slide*. For example, a Sliding Window of size 10 minutes with a slide of 5 minutes will create windows that have start and end time as : 10:00:00 - 10:10:00, 10:00:05 - 10:00:15 and so on. In literature, these are known as concurrent windows. For the reminder of this work, we assume that we are working with only one instance of Sliding Windows, i.e we do not consider concurrent windows. In other words, the *slide* parameter for us is not important. We do this without losing generality and only for simplicity, as in the upcoming sections we will see that the reasoning that applies for a single instance of a Sliding Window also applies for concurrent windows. Also, again for the sake of simplicity, we assume the existence of a global stream with a single key or no key at all. As explained in Chapter 2, in modern SPE's such as Apache Flink, a user first defines queries by passing Window Type(e.g Sliding Windows, Tumbling Windows), the Window Aggregation, and may or may not specify a key. If a key is specified, a separate window is created for each key, and the aggregation is run on top this window. We simplify this by assuming that we are working with non-keyed single input stream. Extending support for keyed-streams is left for future work.

Furthermore, again for the sake of simplicity and unless stated otherwise, we assume that Fargo's Topology comprises of exactly one FDT. The problem of how many and where FDT's have to be deployed on the fog infrastructure are left for future work. It is also assumed that the answers to "how many Sensors must be part of a $\mathcal{S}$ " or "how many Child Nodes should be part of $\mathcal{CN}$ " or similar concerns are based on practical intuitions. For instance, if there is a scenario wherein one thousand Sensors are depleting the performance of the system as a whole due to whatever cause, then instead of having only one FDT, there can be more than one FDT that accommodates some of these Sensors. In general, the problem of optimum resource utilization and operator placement

are not considered, and are left for future work. Fargo focuses strictly on achieving lightweight and quick fault tolerance for Algebraic Aggregations on Sliding Windows. Lastly, we assume that the size of $\mathcal{S}$ can have any number of Sensors, the size of $\mathcal{CN}$ is always less than or equal to each $\mathcal{IG}$, and there is exactly one $\mathcal{R}$.

## 3.3 Components

In this Section, we discuss the various components of Fargo. Fargo has four major components : Sensor, Child Node, Intermediate Node and the Root Node. We adapt these terminologies from Disco[Dis]. However, our components work in a quite different way than described in Disco. As explained before, each component belongs to a Group. In Section 3.3.1, we present Sensor, and the various modules within a Sensor. In Section 3.3.2, we define a Child Node with the various modules that make up a Child Node and the interactions between a Sensor and Child Node. We then proceed to Intermediate Node in Section 3.3.3, and finally end our discussion with Root Node in Section 3.3.4.

### 3.3.1 Sensor

A Sensor is the component that emits events. Before a Sensor starts to emit events, it already knows before-hand each Child Node that it will emit events to. How the Sensor has information of all Child Nodes is explained in Section 3.4.1. For the remainder of this section, we assume that the size of $\mathcal{CN}$ is $n$. In Fig 3.3, we see the different modules within a Sensor. We have an Event Generator that constantly emits data based on a *sendRate* parameter that tells it how many events to generate per specified time unit. We do not impose any restriction on this time unit. For the sake of simplicity, we assume that *sendRate* is always in seconds. We have multiple Event Buffers that temporarily buffer the events that have been emitted by the Event Generator. Lastly, we have a module called CNodes Group Membership Registry that registers and de-registers the various Child Nodes that the Sensor emits data to. We now proceed to explain each module.

**Event Generator**

When a Sensor is initialized, a parameter *sendRate* is passed to it. This parameter is an upper bound on the number of events that the Sensor can emit in one second. As the number of Child Nodes is already known to be $n$, and *sendRate* is the maximum number of events that will be emitted per second, the upper bound on the total number of events that a single Child Node will receive per second is $\lceil \frac{sendRate}{n} \rceil$. Upon generating the event, the event is tagged with an offset, which is an unique sequence number that is incremented by one for each event. The Event Generator sends the event tagged with the offset directly to the Network Channel, and puts the event directly into the Event Buffer allocated for that particular Child Node.

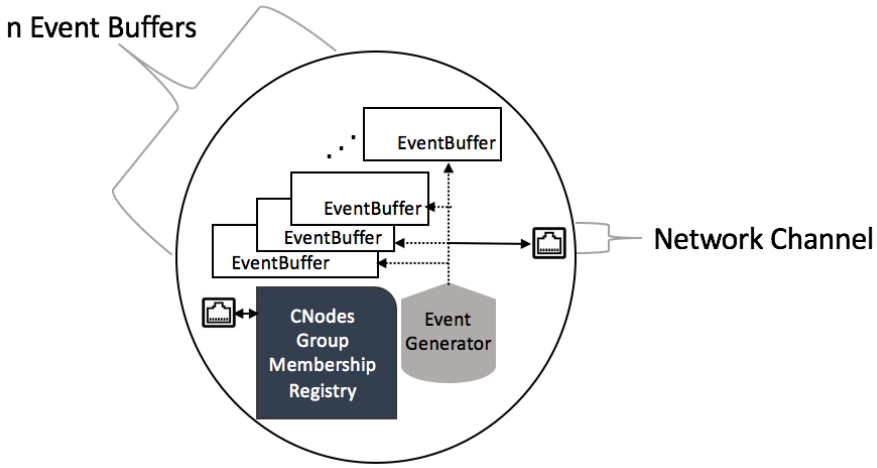



Figure 3.3: Sensor

**Event Buffer**

For each Child Node, a Sensor maintains an Event Buffer. The Event Buffer can be thought of as a queue, the size of which is constrained by the *bufferCapacity* parameter that is passed to the Sensor. The offset of an event denotes it's position in that queue. When an Event Buffer is empty, the value of the offset is zero, and is denoted by $l_0$(called last offset). Each event that is pushed to the Event Buffer has the last offset incremented by one. When the size of the Event Buffer reaches *bufferCapacity*, the first event is popped from the Event Buffer. For instance, if *bufferCapacity* = 2 and the Event Buffer is empty, and the events being pushed to the Event Buffer are *e1, e2, e3* in that order, then e1 has offset zero, *e2* has offset one, *e3* will have offset three and *e1* will be popped from the Event Buffer before *e3* is pushed to it. As the events are being pushed one by one, the value of $l_0$ is also incremented by one. Before *e1* is pushed, $l_0 = 0$. After *e2* is pushed, $l_0 = 1$, and after *e3* is pushed, $l_0 = 2$. '

The reason for constraining an Event Buffer to be of max size *bufferCapacity* is guided by the practical concern that without this contract, an Event Buffer and thereby a Sensor will need to have unlimited capacity. Our motivation to restrict size of the Event Buffers stems from the fact that Sensors in IoT are volatile devices with very limited memory capabilities.

Thus, a Sensor maintains as many Event Buffers as there are Child Nodes in $\mathcal{CN}$. An event that is emitted to a Child Node $\text{cn}_i$ is pushed to the Event Buffer allocated for that $\text{cn}_i$. The offset of each event gives Fargo the capability to re-read events from a particular offset on wards in a sequential manner. In fact, as we will explain in great detail later, the *Events Re-streaming Protocol* uses these offsets to start reading the events from a particular offset onwards in the event a Child Node crashes.

**CNodes Group Membership Registry**

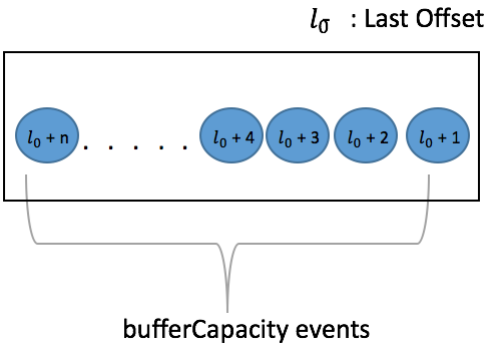



Figure 3.4: Event Buffer

This module maintains a registry of the Child Nodes that a Sensor is connected to. A Sensor uses this registry to know which Child Nodes it needs to emit events to. When a Sensor is initialized, it is connected to all the Child Nodes in $\mathcal{CN}$ and this registry contains the location of all Child Nodes. In the event of a failure of a Child Node, the Sensor removes the failed Child Node from the registry and stops emitting data to it. How the Sensor becomes aware of the failure of a Child Node will be covered in Section 3.4.3.

### 3.3.2 Child Node

In this section, we discuss the different functions of a Child Node. We only briefly touch upon the different modules within a Child Node, as the function of each module is made clear when we discuss Fargo Protocols in Section 3.4 . A Child Node is that component of Fargo that is the only one directly connected to a Sensor, i.e it is the only component that receives raw events that a Sensor emits. Upon receiving these events, the Child Node inspects the timestamp of the event, and puts it in an instance of a Sliding Window on the basis of the start and end time of the instance. Each Sliding Window runs one and only one incremental Algebraic Aggregation. Thus, a Child Node is that particular component of Fargo's topology that is the most computation-heavy as compared to other components as it performs computations over high-volume and high-velocity streams. Exploiting this incremental nature of aggregations, the Sliding Window in the Child Node periodically emits a Fargo Payload that is replicated to other Child Nodes in $\mathcal{CN}$. We explain Fargo Payload in Section 3.4.4. Fig 3.5 showcases the different modules within a Child Node. In brief, each Child Node is equipped with a Phi-Accrual Failure Detector that uses the *Failure Detection Protocol* to monitor other Child Nodes in $\mathcal{CN}$. The Replication Engine uses the *Child Nodes Aggregations Fault Tolerance Protocol* to replicate partial aggregates to other Child Nodes. After a Sliding Window is created on a Child Node using the *Windows Creation Protocol*, it is put in a Sliding Window Registry. In order to make itself discoverable to other components and

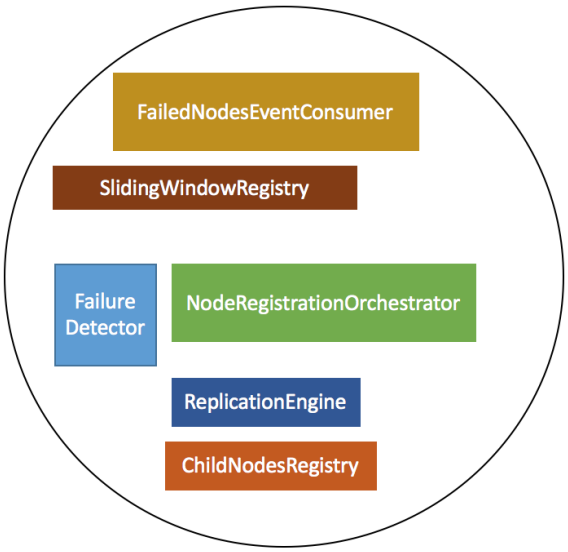



Figure 3.5: Child Node

Groups in Fargo's Topology, Child Node uses *NodeRegistrationOrchestrator* to register itself with the Root Node and get all the metadata it needs from the Root Node in order to start reading events, create Windows and perform incremental aggregations. Finally, the Failed Nodes Event Consumer is the module within a Child Node that participates in the *Events Re-streaming Protocol* to read the events of another failed Child Node. This way, this Child Node becomes what we call as a 'Leader' of all partitions of the failed Child Node. The *ChildNodesRegistry* maintains a registry of all other Child Nodes in $\mathcal{CN}$. Using the *Window Creation Protocol*, before creating any Windows or further instances of a Window, the we first query *ChildNodesRegistry* to discover other possible replicas for the Window. How this happens will be discussed at length in Section 3.4.7.

### 3.3.3 Intermediate Node

An Intermediate Node belongs to $\mathcal{IG}$ and is an optional component in Fargo's Topology. In the absence of $\mathcal{IG}$, the aggregates emitted at the end of each Sliding Window in the Child Node is sent directly to the Root Node.

Fig 3.6 showcases an Intermediate Node and it's three modules. The Sliding Window Registry and Node Registration Orchestrator performs similar functions as discussed in the section on Child Nodes. An Intermediate Node also has a Window Merger, that uses the *Window Merging Protocol* to merge the partial aggregates emitted by Child Nodes at the end of each Window. The merging of multiple partial aggregate is another partial aggregate that is then emitted upstream to either another Intermediate Node or directly to the Root Node.

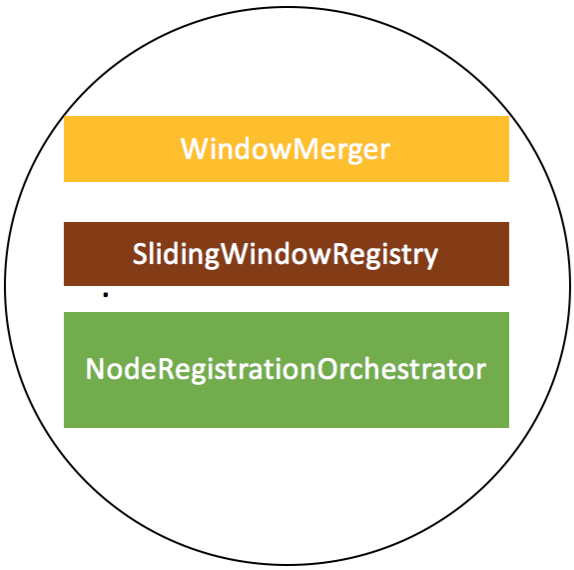



Figure 3.6: Intermediate Node

### 3.3.4 Root Node

The Root Node or $\mathcal{R}$ is the one node that sits atop every layer in our Topology. The Root Node is where the user registers their Windowing Aggregation query and provide a *replication factor*(rf). When the Intermediate and Child Nodes register themselves with the Root Node, it communicates with them all the metadata required to create windows, run aggregations, participate in replication amongst others. Although $\mathcal{R}$ is part of Fargo's Topology, we treat it as a black box/all knowing oracle that needs to conform to a few specifications, which were discussed in Section 3.2.1.

## 3.4 Fargo Protocols

In this section, we discuss the various Protocols that different Groups or components within a Group participate in Fargo. It is these protocols or a combination of them that makes Fargo fault tolerant in the first place. Some protocols are independent and do not rely on other protocols to run before them. On the other hand, there are some protocols like *Events Re-streaming Protocol* that assume that the the protocols *Takeover Sensor Leadership Election* and *Failure Detection* have already run before it. In the subsequent sections, we explain each protocol in detail.

### 3.4.1 Global Group Membership

This protocol concerns with the registration of Sensors with Child Nodes, and Child and Intermediate Nodes with the Root Node. The end result of this protocol is to create a single instance of an FDT. It does that by breaking the process of creating an instance of FDT into two steps : registration of Child and Intermediate Nodes with the Root Node, and subsequent registration of Sensors with Child Nodes. When Child and Intermediate Nodes connect to the Root Node, it responds with the metadata required for these Nodes to create Sliding Windows and run aggregations. The Root Node also clusters the Intermediate Nodes in $\mathcal{IG}$ and Child Nodes in $\mathcal{CN}$. When the Child Nodes and Intermediate Nodes are connected to the Root Node, we have a partial FDT. To make the Sensors discoverable, we take a slightly different approach. The most important Groups that actually perform the aggregations in a distributed fashion are $\mathcal{CN}$ and $\mathcal{IG}$. As soon as a partial FDT is deployed, our Sensors are ready to emit data. Each Sensor connects to any single Child Node, and as a response, this Child Node sends the location of all of it's replicas(i.e every other node in $\mathcal{CN}$) back to the Sensor. As an end result, we get a fully deployed instance of an FDT. This section is thus divided as follows : we discuss how a user can register a query at $\mathcal{R}$, then we discuss the orchestration required by Child and Intermediate Nodes to register themselves with $\mathcal{R}$. We end our discussion on this protocol by explaining how Sensors register with Child Nodes and how our topology enables flexibility in a Sensors Group.

#### Query Registration

The query is registered directly at the Root Node $\mathcal{R}$. The user specifies a *Sliding Window Aggregation Query* by passing three parameters : *Windowing Params*, *Replication Factor* (rf), and an *Aggregation Type*. The *Windowing Params* contains the *size* and *slide* parameters that are required to create a Sliding Window, along with a time unit which can be at the level of second, minute, hour or a day. We don't impose any restriction on the time unit, but for the sake of simplicity from here on forth we assume that the time unit is always in seconds. The *Replication Factor* is an integer value that is always greater than one, and it signals to $\mathcal{R}$ how many Child Nodes should be part of $\mathcal{CN}$ in a single FDT. For instance, if rf = 5, then $\mathcal{R}$ waits for five Child Nodes to register, and groups them together in $\mathcal{CN}$. Note that *Replication Factor* is only applicable to $\mathcal{CN}$, it does not impose any restrictions on how many Intermediate Nodes can be part of $\mathcal{IG}$ or Sensors in $\mathcal{S}$. As soon as the query is registered at $\mathcal{R}$, it waits for Intermediate Nodes and Child Nodes to register. Also, we do not put a restriction on how many queries a user can register. It can be one, two, ten or even one hundred. However, if more than one query is registered, Fargo Protocols require some way of identifying which instance of a Sliding Window an event belongs to. This will be explained in a later section. Therefore, $\mathcal{R}$ creates a unique identifier for each registered query, and this unique identifier is part of the metadata response that is disseminated to Intermediate and Child Nodes.

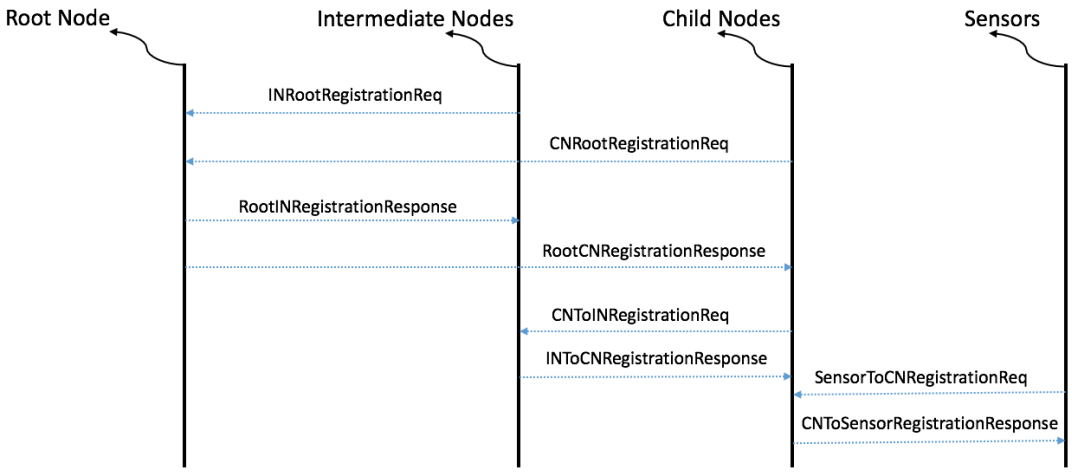



Figure 3.7: Registration and Response Orchestration

### Child and Intermediate Node Registration

In Fig 3.7, we have a sequence diagram where each vertical bar represents some kind of component. For the sake of illustration, we assume that rf = 2, that there are 3 intermediate Nodes and 5 Child Nodes. We do not put any restrictions on the number of Sensors for this discussion. Also, we assume that there is only one layer that corresponds to Intermediate Nodes, i.e the number of $\mathcal{IG}$ is one.

As a first step, the query is already registered at $\mathcal{R}$. Each Intermediate Node connects to $\mathcal{R}$ and sends a *INRootRegistrationReq*, which is short for IntermediateNodeRootRegistrationRequest. This request only contains the id of the Intermediate Node and it's location. Next, each Child Node registers with $\mathcal{R}$, and as part of it's *CNRootRegistrationReq*, it only sends it's location and it's id and nothing else. Since rf = 2, and rf is only concerned with $\mathcal{CN}$, the root node waits for two Child Nodes to register. As soon as the second Child Node registers with $\mathcal{R}$, $\mathcal{R}$ creates the group $\mathcal{CN}$ and sends to each Child Node a *RootCNRegistrationResponse*. It does the same with Intermediate Nodes by creating an $\mathcal{IG}$ and sends to each Intermediate Node a *RootINRegistrationResponse*. Both these responses share some attributes in common. For instance, the metadata regarding the query such as the unique identifier of the Sliding Window, the time unit, *WindowingParams* such as *slide* and *size* as well as the *AggregationType* are part of both these responses. Passing these attributes signifies to Intermediate and Child Nodes the type of Aggregation that they are supposed to run as well as the metadata for instantiating Sliding Windows. The *RootINRegistrationResponse* contains only an acknowledgement from $\mathcal{R}$ and nothing else. Thus, each Intermediate Node is connected to $\mathcal{R}$. However, *RootCNRegistrationResponse* contains more critical information. First, it includes the location of all Intermediate Nodes above it. Second, as we explained above, $\mathcal{R}$ waits till two Child Nodes have registered. As soon as this happens, it designates both Child

Nodes as a 'replica' of one another. Therefore, the *RootCNRegistrationResponse* received by both Child Nodes contains metadata for Windowing, Aggregation, location of Intermediate Nodes, but also the location of every other Child Node in $\mathcal{CN}$. In fact, as we will explain in detail in the section on *Child Nodes Aggregations Fault Tolerance*, each Window on each Child Node has exactly one CRDT associated with it, and each Child Node is part of a Full Mesh Replication Topology.

However, in the section on Fargo DataFlow Topology, we mentioned that Intermediate Nodes are an optional component in the topology. In other words, we might have a case in which $\mathcal{CN}$ is connected to $\mathcal{R}$ directly with no intermediaries in between. Thus, $\mathcal{R}$ needs a mechanism to know whether any Intermediate Nodes will be part of a single FDT. We solve this problem as follows: as soon as *rf* number of Child Nodes have registered with $\mathcal{R}$, $\mathcal{R}$ checks whether it has received a registration request from any Intermediate Node. If 'n' number of Intermediate Nodes have registered themselves with $\mathcal{R}$, we put all of them in $\mathcal{IG}$. If zero number of Intermediate Nodes have registered, we take this to be that there are not going to be any Intermediate Node in the topology. In this case, *RootCNRegistrationResponse* will contain the locations of other Child Nodes but the metadata regarding Intermediate Nodes will be an empty set.

Also, note that when we defined FDT, we put no restrictions on the number of $\mathcal{IG}$ layers a FDT can have. In such a case, we assume that $\mathcal{R}$ is equipped with a parameter called *maxIntermediateNodes*. $\mathcal{R}$ uses this parameter to place at the most *maxIntermediateNodes* in a single $\mathcal{IG}$. If the number of Intermediate Nodes that have registered with $\mathcal{R}$ is *m*, and m > *maxIntermediateNodes* , then $\mathcal{R}$ creates $\lceil \frac{m}{maxIntermediateNodes} \rceil$ number of $\mathcal{IG}$ layers, and at the bottom-most $\mathcal{IG}$ layer it puts *maxIntermediateNodes* number of Intermediate Nodes.

**Sensor Registration**

So far, what we have is a partial FDT, and now we discuss the orchestration of the missing component : Sensors. In our partial FDT, we have $\mathcal{CN}$ and $\mathcal{IG}$ layers already established. Now that these layers exist in our partial FDT, our sensors are ready to connect to Child Nodes and start emitting data. In order to register to all Child Nodes, a Sensor needs to connect with only one Child Node, and as a response, this Child Node sends forth the location and id's of all of it's replicas, i.e it sends the location of every other Child Node in $\mathcal{CN}$. A Sensor sends a *SensorToCNRegistrationReq*, that contains it's id and location. After the Sensor sends it's request, the Child Node responds with a *CNToSensorRegistrationResponse*, that contains the id's and location of itself and every other Child Node in $\mathcal{CN}$. The Sensor then initiates a connection using the underlying network channel with all Child Nodes, and once it has connected to each Child Node, it starts emitting events.

Note that initially, the Sensor Group $\mathcal{S}$ is empty. We say that a Sensor *s* belongs to $\mathcal{S}$ once it has successfully connected to each Child Node in $\mathcal{CN}$. As soon as one Sensor is added to $\mathcal{S}$, our FDT is finally complete. This gives us a great deal of flexibility as new Sensors can be added to $\mathcal{S}$ as and when needed.

### 3.4.2 Window Creation

In this section, we discuss the *Window Creation Protocol* that the layers $\mathcal{CN}$, $\mathcal{IG}$ and $\mathcal{R}$ participate in. We saw in the previous section that as part of it's response to group membership requests, $\mathcal{R}$ sends forth query information such unique identifier for a Window, *WindowingParams*, *AggregationType* and so on. For the remainder of this section, we refer to a node or nodes as being either a Child Node, Intermediate Node or the Root Node.

If we have a Sliding Window W with *size* of ten and *slide* as five with the time unit in minutes, the windowing operator will discretize a stream into the following time-based chunks : 12:00:00-12:00:10, 12:00:05-12:15:00,12:10:00-12:20:00 and so on. We call each of these time-based chunks as an *instance* of the Sliding Window with a start time given by $S_{start}$ and an end time given by $S_{end}$. In order to perform aggregations on Windows in a distributed setting such as ours, we need some terminologies to represent and identify instances of a Sliding Window. We are already equipped with a unique identifier for a Sliding Window that is sent to each Child Node and Intermediate Node by $\mathcal{R}$. Therefore, to uniquely identify each instance of a Sliding Window across the entire topology, we use the following parameters:

1. WindowId : The global unique identifier for the window provided by $\mathcal{R}$
2. $S_{start}$ : The start time for the instance of the window
3. $S_{end}$ : The end time for the instance of the window
4. NodeId : The unique identifier of the Node

The quadruple (WindowId, $\mathrm{S_{start}}$, $\mathrm{S_{end}}$, NodeId) can thus help Fargo identify an instance of a Sliding Window on that particular node. The triplet (WindowId, $\mathrm{S_{start}}$, $\mathrm{S_{end}}$) helps Fargo to identify the same instance of the Sliding Window across all nodes.

**Definition 26.** *(Sliding Window Instance Identifier) Sliding Window Instance Identifier is the triplet(WindowId, $S_{start}$, $S_{end}$) where WindowId is the unique identifier of the Sliding Window W received from $\mathcal{R}$, $S_{start}$ is the start time of the instance and $S_{end}$ is the end time of the instance*

A node uses the local time provided by the OS on which it runs to instantiate a Sliding Window. The instant the local time reaches $\mathrm{S_{end}}$, Fargo creates a new instance of the window by sliding it via the *slide* parameter. In summary, upon each *slide* a new instance of a Sliding Window is created and we identify each instance by the Sliding Window Instance Identifier defined above.

Lastly, all Child Nodes undergo an additional step as per this protocol. Before creating any instance of a Sliding Window, a Child Node first queries *ChildNodeRegistry*. Recall from Section 3.3.2, the *ChildNodeRegistry* is a registry that contains the locations and ids of all other Child Nodes in $\mathcal{CN}$. As we will see in Section 3.4.4, we treat each instance of a Window as a single unit that performs incremental aggregations and does replication to other Child Nodes. Therefore, it is important to know the addresses of all other Child Nodes, as the *Child Nodes Fault Tolerance Protocol* uses the number of Child Nodes to create CRDT replicas for each Window. This is explained in detail in Section 3.4.4.

### 3.4.3 Takeover Sensors Leadership Election

In this section, we discuss the *Takeover Sensors Leadership Election* that dictates the steps that need to be taken by all the correct Child Nodes in case a Child Node fails. When a Child Node fails, using the *Failure Detection Protocol*, one or more Child Nodes detect this failure. Since a Child Node has failed and there are Sensors that are constantly emitting events to this failed Child Node, what we require is a quick leadership election between the remaining correct Child Nodes. The outcome of this protocol is to elect a new leader for all the Sensor partitions of the failed Child Node.

Our protocol is based on the well known Bully Algorithm for Leadership Election[GM82]. There are many improvements that have been done over the past decades on improving the performance of the Bully Algorithm such as [SST+14], [MA12] and [AAKO19]. In this section, we explain the classical Bully Algorithm that the Child Nodes participate in[GM82]. Furthermore, we leave it to implementation details to decide which optimized version of the Bully Algorithm should be used.

We now explain the algorithm briefly. Assume that we have 10 number of Child Nodes, namely ChildNode1, ChildNode2,...,ChildNode10. Furthermore, assume that ChildNode1 has failed and this failure is detected first by ChildNode5. Additionally, assume that the unique identifiers of these nodes follow some ordering. For simplicity, we assume that ChildNode1 has the lowest identifier, ChildNode2 the second lowest and so on till ChildNode10, which has the highest identifier. Lastly, we assume the existence of a *leadershipElectionTimeout* parameter milliseconds. Next, recall that each Child Node has a unique identifier which is passed to it when it is initialized. Using the *Global Group Membership Protocol*, after every Child Node registers with the Root Node, the Root Node sends as a response the addresses of other Child Nodes as well as their unique identifiers. As ChildNode5 detects the failure, it initiates an election with the other remaining correct Child Nodes. ChildNode5 initiates this election with every Child Node with their unique identifier greater than its own. In other words, the election is initiated with ChildNode6, ChildNode7,...,ChildNode10. It waits for *leadershipElectionTimeout* milliseconds for each Child Nodes with which it initiated the election with to respond. If none of them reply within *leadershipElectionTimeout* time units, ChildNode5 elects itself as the new leader and sends a leadership broadcast to other all Child Nodes with the unique identifier lower than its own. That is, it sends an elected leader broadcast to ChildNode2, ChildNode3 and ChildNode4 letting them know that it is now a leader of all the partitions of ChildNode1. However, if a Child Node with a higher unique identifier does reply back to ChildNode5 acknowledging, say from ChildNode7, the election takes place again, but this time, ChildNode7 sends the election request to all the Child Nodes with unique identifier greater than its own. We do this recursively till we get a leader with the highest unique identifier.

Upon the completion of this protocol, the new leader Child Node initiates the *Events Re-streaming Protocol* to start re-streaming events across all Sensors that were initially meant for the failed Child Node.

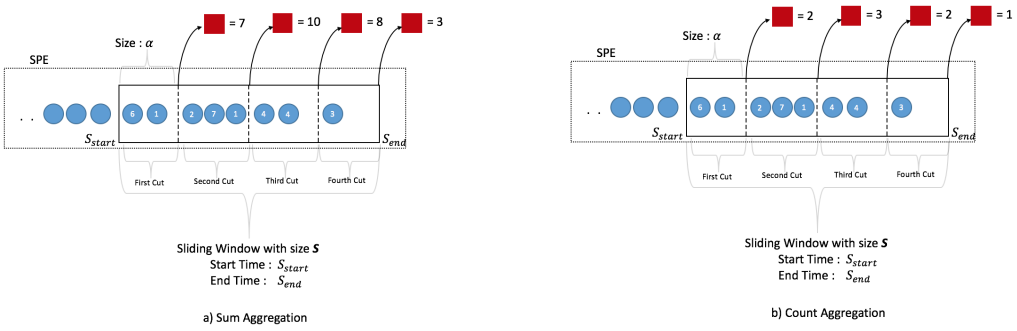



Figure 3.8: Cuts in a Sliding Window performing Sum and Count Aggregations

### 3.4.4 Child Nodes Aggregations Fault Tolerance

In this Section, we discuss perhaps one of the most important contributions of Fargo : making Algebraic Aggregations fault tolerant using Optimized Delta CRDT's. For the remainder of this section, we forego the use of the words Optimized Delta CRDT's and just use the shorthand CRDT's instead. This section is divided as follows : we first illustrate the concept of *Cuts* using the sum and count aggregations. Then, we generalize and formalize this concept to support any Algebraic Aggregations. We illustrate how for an instance Sliding Window, Fargo creates exactly one CRDT on a Child Node. We then illustrate the dataflow and the replication protocol employed by $\mathcal{CN}$ with the help of two Sensors and two Child Nodes.

#### Cuts

In Fig 3.8, we see the sum and count aggregations over a Sliding Window in a Child Node. The blue balls are events, and the text within the blue ball highlights the value that it holds. We have a Sliding Window that starts at time $S_{start}$ and ends at $S_{end}$. Fargo uses a parameter called $\alpha$ that partitions a Sliding Window into $\lceil \frac{S_{start} - S_{end}}{\alpha} \rceil$ cuts. $\alpha$ is the replication sync interval given in time units, and for the sake of simplicity, we assume that $\alpha$ is always in milliseconds. For instance, if the size of the Window is 10 seconds and $\alpha$ is 500 milliseconds, we have (10 * 1000) / 500 = 20 Cuts. We now provide a formal definition of Cuts :

**Definition 27.** *(Cuts) Given a time based parameter $\alpha$ and a Sliding Window $W$ with start time $S_{start}$ and end time $S_{end}$, a Cut $\mathcal{C}$ is a partition within $W$ with two temporal boundaries namely a start boundary and an end boundary such that:*

1. *There are $\mathcal{C}_{count} = \lceil \frac{S_{start} - S_{end}}{\alpha} \rceil$ number of Cut within $W$*
2. *$W = \bigcup \mathcal{C}$ and $\mathcal{C}_i \bigcap \mathcal{C}_j = \emptyset; \forall\ i, j \in \{1, 2, 3, .., \mathcal{C}_{count}\}$. In other words, the Window is a union of each Cut $\mathcal{C}$ and the each Cut is non-overlapping with each other*
3. *Let $\mathcal{C}_1, \mathcal{C}_2, ...., \mathcal{C}_{\mathcal{C}_{count}}$ be all Cuts within $W$ in that order. Then temporal boundaries are given by the left-open intervals $(S_{start}, S_{start} + \alpha]$, $(S_{start} + \alpha, S_{start} + 2 * \alpha]$, $(S_{start} + 2 * \alpha, S_{start} + 3 * \alpha], ..., (S_{start} + \alpha * (\mathcal{C}_{count} - 1), S_{end}]$ respectively.*

In Fig 3.8(a) and Fig 3.8(b), our Window is divided into four almost equal-sized cuts. At the end of each cut, depending on the aggregation, a partial aggregate is emitted, depicted by the red square. For the sum aggregation, the first cut has two events with values 6 and 1, and the partial aggregate emitted in this case is 6 + 1 = 7. Similarly, the partial aggregates emitted at the end of the second cut is 2 + 7 + 1 = 10, at the end of the third cut is 4 + 4 = 8, and the last cut has one value which is 3. Similarly, for the count aggregation, the partial aggregates emitted at the end of the first, second, third and last Cut are 2, 3, 2 and 1 respectively. The significance of Cuts is that it gives us temporal boundaries on *when* as to when Sliding Window should emit partial aggregates. We will use this concept of emitting a partial aggregate at the end of each Cut extensively in the upcoming sections. In fact, we use a Cut's end boundary to emit what we term as a Fargo Payload that encapsulates this partial aggregate, as we will see in the section on Fargo Payload. More over, if the end of a Cut boundary is the answer to '*when* the Fargo Payload should be emitted', the number of cuts is the answer to 'how many Fargo Payload be emitted'.

**Window and CRDT**

We assume a Sliding Window $W$ with a *size* and a *slide*, and a start time of $S_{start}$ and end time of $S_{end}$. We only consider a single instance of this Sliding Window. However, the same logic will apply to concurrent instances of this Window as well.

To make aggregations fault tolerant, Fargo employs the use of CRDT's that are tailored specifically towards the aggregation in question. In the Section on CRDT's in Chapter 2, we saw the GCounter CRDT. Assuming a GCounter CRDT with n replicas, we saw that each replica maintains as its local state a version vector of size n. When a replica gets an update request from a client, it increments the value in version vector where the replica is indexed by one. When a replica receives a replicated update from another replica, it merges this update with it's own local state. In this way, a client can query any replica and get the answer to the query 'what is the current count?'. This is the very intuition behind Fargo's approach. With each instance of a Sliding Window and an Algebraic Aggregation, we attach exactly one CRDT that is meant to work for only that aggregation. For instance, for the data aggregation count, we can use GCounter CRDT. In Section 3.5 , we provide and formally prove CRDT's for data aggregations sum, max, min, average amongst others.

**Fargo Payload**

So far, we have discussed that Child Nodes perform replication and that each Sliding Window running an Algebraic Aggregation has a corresponding CRDT tailored specifically for that aggregation. We now define the unit of replication, called *Fargo Payload*. We use the following terminologies : $\alpha$ is a parameter that is used to partition W into $\mathcal{C}_{\text{count}}$ number of Cuts, $\mathcal{S}_{\text{O}}$ is a map of key value pairs where the key is a Sensor ID and value is the last processed offset at the end of each Cut, and $\Delta$ is $\Delta$-group emitted by the underlying CRDT that is attatched to an instance of a Sliding Window $W$ running an Algebraic Aggregation $A$:

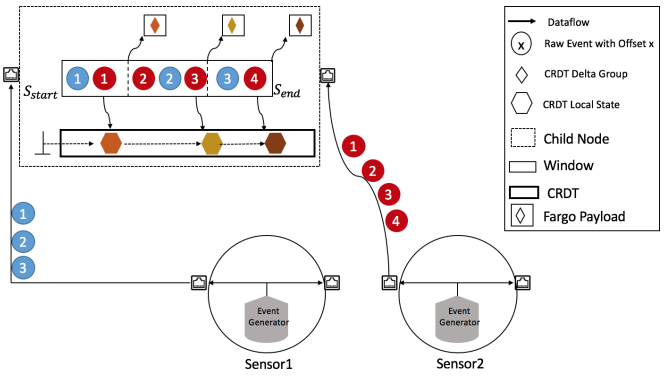



Figure 3.9: Window, CRDT and Fargo Payload

**Definition 28.** *(Fargo Payload) A Fargo Payload is a triplet* ($\Delta$, $\mathcal{S}_O$, *WindowInstanceId) such that :*

1. $\Delta$ *is the* $\Delta$*-group that is emitted by the the underlying CRDT that is attatched to an instance of a Sliding Window*
2. $\mathcal{S}_O$ *is a map of (key, value) pairs where the key is the ID of the Sensor and the value is the offset is the last offset of the event that was processed at the end of the Cut*
3. *WindowInstanceId is the Id of the instance of Sliding Window*

In Fig 3.9, we have two Sensors that are emitting events to a Child Node. The balls denote an event emitted by the Sensor, and the text within it specify the offset value of that event. The first Sensor emits the events represented as blue balls and the second Sensor emits events represented by the red balls. As the events reach the Child Node, their timestamp is inspected. Based on the timestamp, an event is put to a corresponding instance of a Sliding Window. This instance is already divided into three Cuts. Each event that is put in an instance of a Window is used as an update for the underlying CRDT. In other words, it is the CRDT itself that is performing the aggregation. In Fig 3.9, we see that the underlying CRDT goes through state transitions upon each event. For brevity, we only show three state transitions. In the meantime, when we reach the end of the first Cut, a Fargo Payload is emitted by the instance of the Window, represented by the square that contains the orange quadrilateral.

More over, at the end of the first Cut, the instance of our Window has processed events from two Sensors. The last processed offset from Sensor1 is 1 and from Sensor2 is 2. Using same terminologies as in Definition 5, we have $\mathcal{S}_O = \{$ "1" $\mapsto$ 1, "2" $\mapsto$ 2 $\}$, where the key in our map is the ID of the Sensor and the value is the last processed offset. At the end of the second Cut, $\mathcal{S}_O = \{$ "1" $\mapsto$ 2, "2" $\mapsto$ 3 $\}$, and at the end of the final Cut, $\mathcal{S}_O = \{$ "1" $\mapsto$ 3, "2" $\mapsto$ 4 $\}$.

In summary, as the Sliding Window progresses in time, a new Fargo Payload is emitted at the end of each Cut and is ready to be replicated to other Child Nodes. In our naive

illustration, we have three Fargo Payloads that are replicated to other Child Nodes one after the other. In the section on *Events Re-streaming Protocol*, we will explain in detail the reasoning behind embedding the metadata of the processed offsets of Sensors into a Fargo Payload. However, in brief, the idea is that in the event of a failure of a Child Node, a leadership election takes place and a different Child Node becomes the leader of Event Buffers in all Sensors that the Child Node was originally reading from. Because we embed the processed offsets of Sensors directly into Fargo Payload and replicated it already to all other Child Nodes, using the last processed offset our new leader will know *exactly* where the failed Child Node was in it's computation. Using *Events Re-streaming Protocol*, it queries all the Sensors that were connected to the failed Child Node, and starts streaming data that belonged to the failed Child Node from the last processed offset of these Sensors.

### $\mathcal{CN}$ Peer-to-Peer Full Mesh Replication Topology

We now put everything together and discuss the replication topology of the participating Child Nodes. Each Child Node runs multiple instances of Sliding Window, and each instance has exactly one CRDT that goes with it. Since a Fargo Payload encapsulates a $\Delta$-group and Sensors offsets metadata, the replication used by Child Nodes not only synchronizes the CRDT replicas located on different Child Nodes, but also makes a Child Node aware of *where* in its computation the other Child Nodes are at.

Consider Fig 3.10, where we have two Child Nodes that are receiving events from two Sensors. Let's call the Child Node on the left as ChildNode1, and the one on the right as ChildNode2. On ChildNode1, the balls in blue are events emitted by Sensor1, and the balls in red are events emitted by Sensor2. Similarly, on ChildNode2, the dark blue balls are events emitted by Sensor1 and the balls in purple are events emitted by Sensor2. The text in the balls is the offset of that event corresponding to the Sensor it originated from. On both ChildNode1 and ChildNode2, at the end of the each Cut, a Fargo Payload is emitted that contains the Sensors offsets given by $\mathcal{S}_{\mathrm{O}}$, and $\Delta$-group that from the underlying CRDT. Recall that Fargo Payload also encapsulates the Window metadata that contains the ID of the instance of the Sliding Window, which we called as WindowInstanceId. Therefore, the emitted Fargo Payload is replicated by ChildNode1 to ChildNode2 and from ChildNode2 to ChildNode1. Using WindowInstanceId, both Child Nodes are able to identify which instance of the Sliding Window the Fargo Payload is meant for. When ChildNode1 reaches the end of the first Cut, its Fargo Payload is shipped to ChildNode2. When ChildNode2 unpacks the Fargo Payload, it extracts the received $\Delta$-group and merges it with its own local state. This merge is depicted by the second orange Polygon in the CRDT on Child Node2. Furthermore, ChildNode2 also extracts $\mathcal{S}_{\mathrm{O}}$ from the Fargo Payload and stores it. This way, ChildNode2 is now aware how much progress ChildNode1 has made at the end of its first Cut. This bidirectional replication happens between ChildNode1 and ChildNode2 for as many Cuts as we have defined. As we have three Cuts in Fig 3.10, ChildNode1 sends three Fargo Payloads but also receives three Fargo Payloads from ChildNode2.

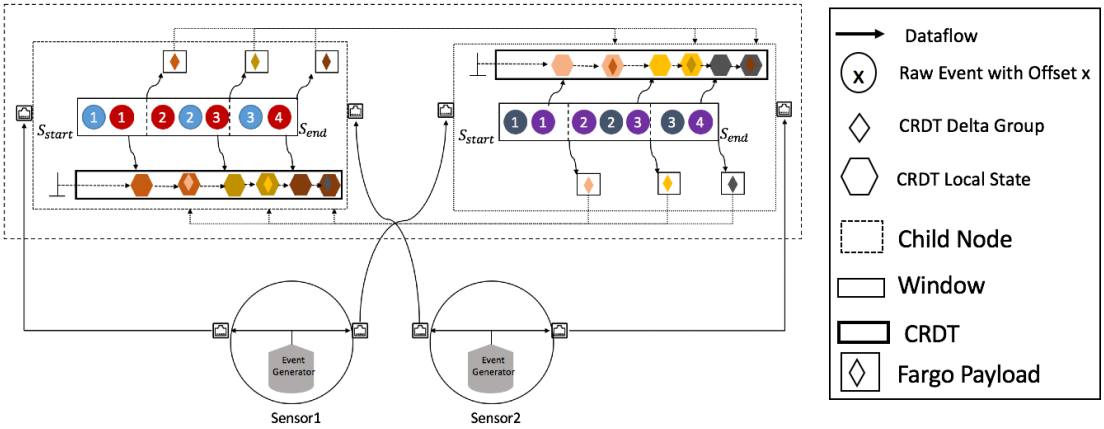



Figure 3.10: Replication

We now generalize our concept of replication. In Fig 3.11, we see four Child Nodes that participate in replication with each other. Each Child Node is connected with the other three. The arrow represents the direction of replication. In this example, ChildNode1 sends the same Fargo Payload to the other three nodes, and receives three distinct Fargo Payloads from other replicas. Extending this idea to support any number of Child Nodes, Fargo uses a Full Mesh replication topology. What this means is that each pair of Child Nodes in $\mathcal{CN}$ form bidirectional replicas. We now formally define $\mathcal{CN}$ replication topology:

**Definition 29.** *($\mathcal{CN}$ Peer-to-Peer Full Mesh Replication Topology) $\mathcal{CN}$ Peer-to-Peer Full Mesh Replication Topology is a directed graph $G = (C, E)$ such that :*

1. *Each vertex $c \in C$ is a Child Node*
2. *Each edge $e \in E$ between two vertices denotes the direction of replication*
3. *For any $c_1, c_2 \in C$, $\exists$ exactly two edges $e_1, e_2 \in E$ such that $c_1 \xrightarrow{e_1} c_2$ and $c_2 \xrightarrow{e_2} c_1$. In other words, $c_1$ is connected to $c_2$ via the edge $e_1$, and $c_2$ is connected to $c_1$ via the edge $e_2$*

### 3.4.5 Events Re-streaming

Child Nodes participate in the the *Events Re-streaming Protocol* after a Child Node fails and has this failure detected by some other Child Node. This other Child Node is now the leader of all partitions across all Sensors that the failed Child Node was responsible for.

In Fig 3.12, we illustrate the *Events Re-streaming Protocol* with a simple use case of two Child Nodes receiving events from a single Sensor. The two Child Nodes are called CN1 and CN2. The Sensor allocates two Event Buffers, one for each Child Node. The balls are the events, and the text within the balls is the offset of that event. The balls in blue are meant for CN1, and the balls in red are meant for CN2. The Sensor

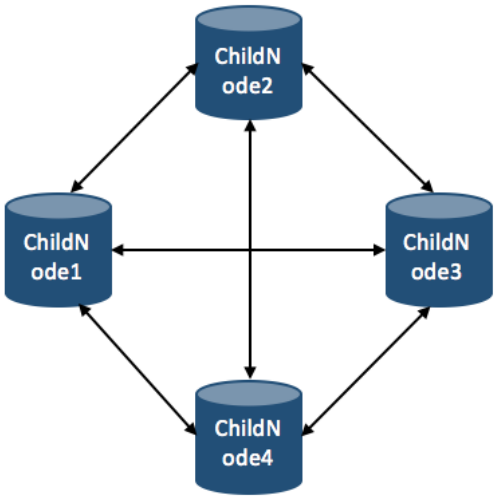



Figure 3.11: Four Child Nodes Replication

successfully sends two events to CN2, with offsets 1 and 2 respectively, after which CN2 crashes. Only for the sake of illustration and to keep it simple, we assume that CN2 had already replicated a Fargo Payload to CN1 the partial aggregate that it computed for the events with offsets 1 and 2. In other words, CN2 had progressed upto two events in it's computation and then crashes. Next, using *Failure Detection Protocol*, CN1 detects that CN2 has crashed. This is where *Events Re-streaming Protocol* is rendered, upon which CN1 does the following : it sends an Events Re-streaming Request to the Sensor, and the payload of this request contains the ID of CN2 and its last processed offset that CN1 was made aware of when CN1 received a Fargo Payload before from CN2 before it failed. In this example, the last processed offset of CN2 is 2. The Sensor receives this request, queries the CNodesGroupMembership module with the ID of CN2, and it starts to re-stream all those events in the EventBuffer for CN2 with offset greater than 2. In Fig 3.12, we see that the Sensor responds to CN1 request by re-streaming the events with offsets 3, 4, 5.

When CN1 receives these re-streamed events, it puts it in all those instances of Windows that were running while CN2 had failed. Most importantly, recall that the instances of Windows have their corresponding CRDT and these CRDT's are running the aggregation. This means that CN1 maintains the value held by CN2(because it is a CRDT). When the events are re-streamed to CN1, they are used as an update for CN2. Also, any subsequent events are not emitted to CN2 anymore by the Sensor. In fact, CN1 being the new leader of CN2 will receive all events that were supposed to be emitted to CN2 in the first place.

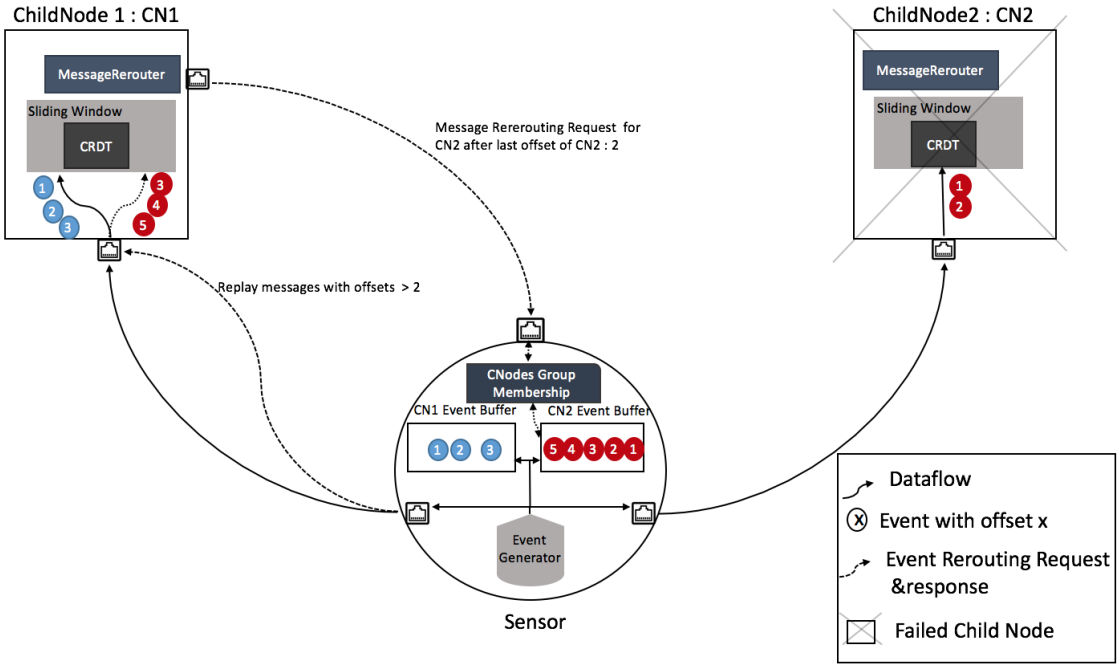



Figure 3.12: Events Rerouting Between Two Child Nodes

What has happened is that the input data stream for CN2 is now part of the input data stream for CN1. The fact that we did not reroute all the events in the EventBuffer has two advantages. First, we save network bandwidth as depending on the offset, we only reroute a part of the total EventBuffer. Second, CN1 does not recompute the aggregation that had already taken place in CN2 and replicated to CN1. This helps in minimizing overall latency.

Recall that in Section 4.3.1 on Sensors, we stated that an EventBuffer is constrained by a *bufferCapacity* parameter that restricts the size of an EventBuffer. If *bufferCapacity* is infinite, then we have nothing to worry about, as the requested offset will always be in the EventBuffer. However, in our toplogy, there is always a possibility of forever losing events if *bufferCapacity* is a small value. We illustrate this with the help of an example. In Fig 3.13, we assume that CN1 sends an Event Rerouting Request to the Sensor, but this time, the value of the offset is 100. The EventBuffer has a capacity for five events only. The EventBuffer on the left has has events with offsets 100, 101, 102, 103 and 104. In this case, our request succeeds as the Sensor responds with events having offsets 101, 102, 103 and 104. However, the smallest offset in the EventBuffer on the right is 201. In this case, the Sensor no events to respond with, as 201 $>$ 100(our requested minimum offset). In other words, we lost 201 - 101 = 100 messages.

**Definition 30.** *(Events Lost) Given a failed Child Node $cn_i \in \mathcal{CN}$, a Window $W$, a Sensor $S \in \mathcal{S}$, a requested offset $r_o$, the smallest offset $s_o$ in the EventBuffer in $S$ for $cn_i$, the Events Lost is given by $s_o$ - $r_o$ + 1, if and only if $s_o > r_o$*

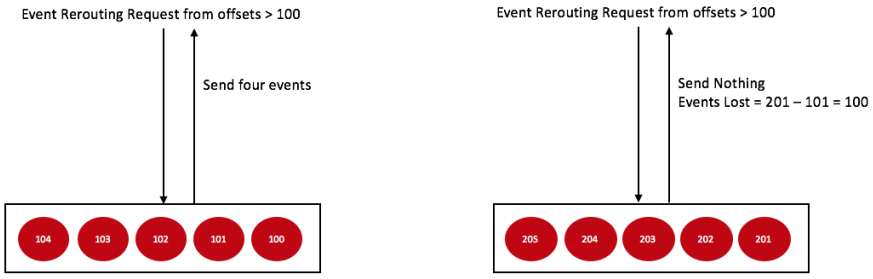



Figure 3.13: Querying EventBuffer with an Offset

Note that the definition of Events Lost is applicable to a single Sensor only. If there are multiple Sensors involved, and multiple Child Nodes that have failed, the Total Events Lost in that case would be a summation of all Events Lost for each failed Child Node and it's corresponding Sensors. Also, note that it will never be possible to have $s_o < r_o$. If this were to happen, this would mean that the Sensor has received a request for events after an offset that it has not even emitted yet. The ideal scenario will always be to have zero Events Lost. We postulate three cases in which Fargo can achieve zero Events Lost. First, if the *bufferCapacity* is small, then a low rate of event generation by the Sensor can circumvent the situation in $r_o$ is less than $s_o$. Second, the *bufferCapacity* is chosen carefully and is large enough to buffer large number of events. The third postulate has to do with the quality of failure detection itself. If the failure of a Child Node is not detected quickly, and if the rate of event generation by the Sensor is so fast that the EventBuffer is just en-queuing and popping.

### 3.4.6 Window Merging

This section is devoted to the *Window Merging Protocol* that either $\mathcal{CN}$ and $\mathcal{IG}$ participate in, or $\mathcal{IG}$ participates with $\mathcal{R}$. We first discuss the protocol from the point of view of $\mathcal{CN}$ and $\mathcal{IG}$.

This protocol works as follows : When an instance of a Sliding Window reaches it's end at an Intermediate Node, it waits/blocks till it receives the local state of the CRDT at each Child Node. We equip an Intermediate Node with the state-based version of the merge function of the CRDT's that are running for the instance of the Window, and an Intermediate Node uses the state-based merge function to merge the different states of Child Node replicas. Note that we did not create any CRDT on an Intermediate Node itself. Calling a merge function on the states of CRDT's replicas outside of a CRDT does not violate any correctness, as we are simply merging states as per the semantics of the CRDT in question.

In Fig 3.14, we have two Child Nodes that send the local state of a CRDT upstream to an Intermediate Node when an instance of a Sliding Window running on them finishes. The shaded box after $S_{end}$ means that the window has already reached the end time on the Intermediate Node, and is waiting for both Child Nodes to emit their CRDT states. When the two states are received at the Intermediate Node, as explained above it uses

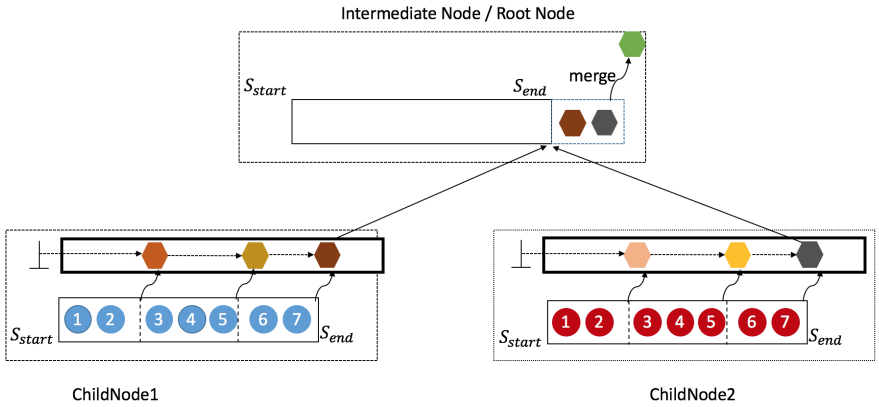



Figure 3.14: Windows Merging

the state-based CRDT merge function, and as a result we get a new state transition to the polygon in green in Fig 3.14. If there exists any Intermediate Node Group above the Intermediate Node that performed the state-based CRDT merge, we simply pass on the result of the state-based CRDT merge upstream.

Similarly, if $\mathcal{CN}$ is connected directly to the Root Node with no Intermediate Nodes in between, the state-based CRDT merge takes place directly at the Root Node. The Root Node will instantiate a hollow window just as we showed in Fig Fig 3.14.

### 3.4.7 Failure Detection

In this section, we discuss *Failure Detection Protocol* that Intermediate Nodes Group and Child Nodes Group participate in. This protocol enables Child Nodes and Intermediate Nodes to detect and reacts to failures of components within each Group. For the remainder of this section, we limit our discussion to how this protocol works for Child Nodes Group only. However, the same reasoning applies to the protocol working on Intermediate Nodes Group.

This protocol dictates that each Child Node be equipped with a Phi-Accrual Failure Detector. Assuming $\mathcal{CN}$ with n number of Child Nodes, each Child Node sends regular heartbeats to n-1 number of Child Nodes, thus forming a peer-to-peer Full Mesh Topology. Each Child Node expects a parameter called *heartbeatFrequency* in milliseconds. Every *heartbeatFrequency* milliseconds, a Child Node sends a heartbeat to other Child Nodes. As a user might register multiple Window queries with varied time intervals, the *heartbeatFrequency* is always set to be less than or equal to the the Window with the lowest size. For instance, if a user registers three queries with the size of the Sliding Window being three, five, and ten seconds, the *heartbeatFrequency* must always be less than three. A heartbeat is a mere ping to other Child Nodes to become aware that the Child Node sending the heartbeat is alive.

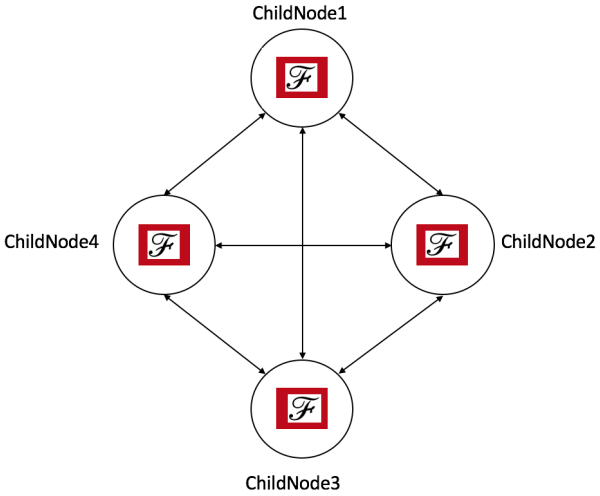



Figure 3.15: Peer-to-Peer Full Mesh Heartbeats Topology

In Fig 3.15, we have four Child Nodes, each equipped with a Phi-Accrual Failure Detector. At each *heartbeatFrequency*, ChildNode1 sends a heartbeat to ChildNode2, ChildNode3 and ChildNode4. In turn, ChildNode1 receives three heartbeats from Child Node2, ChildNode3 and ChildNode4. Assuming that the underlying network does not break and reliably deliver each heartbeat to its destination, in total, at every *heartbeatFrequency*, we have a maximum of 6 * 4 = 24 heartbeats flowing through $\mathcal{CN}$. To generalize, for n number of Child Nodes sending a heartbeat at every *heartbeatFrequency* milliseconds, the total number of heartbeats flowing through $\mathcal{CN}$ is given by n * (heartbeats send by a Child Node + heartbeats received a Child Node). We leave it to implementation details as to what the content of a heartbeat must be. We now formally define our Peer-to-Peer Full Mesh Heartbeats Topology :

**Definition 31.** *(Peer-to-Peer Full Mesh Heartbeats Topology) For an Intermediate Nodes Group or Child Nodes Group, the Peer-to-Peer Full Mesh Heartbeats Topology is a directed graph G = (V, E) such that :*

1. *Each vertex $v \in V$ is a Child Node or an Intermediate Node*
2. *Each edge $e \in E$ between two vertices denotes the direction of heartbeats*
3. *For any $v_1$, $v_2 \in V$, $\exists$ exactly two edges $e_1$, $e_2 \in E$ such that $v_1 \xrightarrow{e_1} v_2$ and $v_2 \xrightarrow{e_2} v_1$. In other words, $v_1$ is connected to $v_2$ via the edge $e_1$, and $c_2$ is connected to $v_1$ via the edge $v_2$*

Furthermore, in Section 3.4.6, we mentioned that the parent of $\mathcal{CN}$, be it $\mathcal{IG}$ or $\mathcal{R}$, waits for all Child Nodes to send their CRDT states. However, in the event that a Child Node has failed, there needs to be a way of making the parent Group of $\mathcal{CN}$ aware of that phenomenon. If the update of a Child Node failing is not sent upstream, the parent Group of $\mathcal{CN}$ will wait forever for the failed Child Node to send forth the CRDT state.

If this was to happen, the Window will never complete as the parent Group will block the completion of the Window. Therefore, in the event that a Child Node fails and is detected using this protocol, the new leader Child Node sends an update to the parent Group that a Child Node has failed. To illustrate what we mean, consider a scenario where there are five Child Nodes connected to a Root Node, i.e no Intermediate Nodes. Now assume that a Child Node called ChildNode1 has failed, and ChildNode2 is the new leader. In the scenario where there is no failure, the Root Node expects to receive CRDT states from five Child Nodes. However, as ChildNode1 has failed, the new leader ChildNode2 sends an update to the Root Node signalling that it is now the leader of all Sensors partitions of ChildNode1 and that the Root Node should now expect only four Child Nodes to send their CRDT states. Furthermore, note that in Section 3.4.5, we mentioned that when the new Leader Child Node re-streams the events belonging to the failed Child Node, it uses these events as an update for the failed Child Node. Therefore, when the leader sends its entire CRDT state upstream, this CRDT state also contains the updated values for the failed Child Node. In conclusion, the *Failure Detection Protocol* not only detects failures within $\mathcal{CN}$ but also makes the parent Group of $\mathcal{CN}$ aware that a Child Node has failed.

Lastly, any Child Node that detects the failed Child Node also sends sends a broadcast message to other correct Child Nodes making them aware of the failed Child Node. This message contains the id of the failed Child Node. Note that in most cases, depending on the parameters set in the Phi-Accrual Failure Detector, every other correct Child Node would have detected the failed Child Node anyway. If another Child Node detects the failed Child Node, but it had already received this broadcast message, it does not send any broadcast message. This does not hamper performance in any way, as for n number of Child Nodes, the Child Node that detects the failure sends a mere n - 1 broadcast messages.

### 3.4.8 Intermediate Nodes Fault Tolerance

In this Section, we discuss the *Intermediate Nodes Fault Tolerance* that is rendered in the event an Intermediate Node in any Intermediate Nodes Group fails.

In Fig 3.16, we illustrate a Fargo Dataflow Topology with a Sensors Group, Child Nodes Group and Intermediate Nodes Group with two Intermediate Nodes. As the sensor emits events, depicted by the triangle, the Child Nodes performs the aggregations and sends the final CRDT state to the two the Intermediate Nodes upstream. The CRDT state is sent when a Window instance reaches its end time. At some point during the duration of the Window instance, the first Intermediate Node fails. Using the *Failure Detection Protocol*, the other Intermediate Node detects the Intermediate Node to have failed, and sends a request to the Child Nodes downstream to stop sending CRDT states for other Windows to the failed Child Nodes. Furthermore, notice that all Child Nodes sent the same CRDT states to both Intermediate Nodes. As one Intermediate Node has failed, we still achieve fault tolerance as the second Intermediate Node also received the same CRDT states. Thus, we conclude that $\mathcal{IG}$ of size n can tolerate up to n-1 failures, as every Intermediate Node receives the same CRDT states from all downstream Child Nodes.

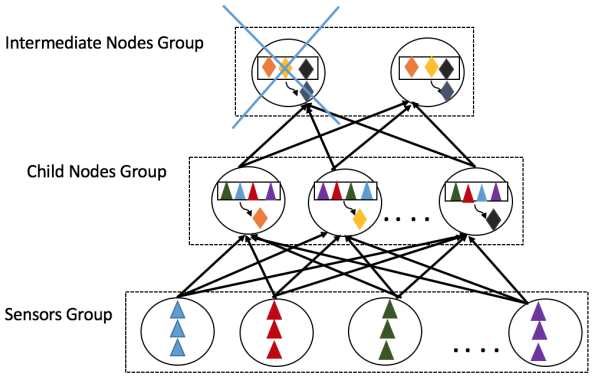



Figure 3.16: Child Nodes emitting CRDT states to Intermediate Nodes

## 3.5 Delta CRDT's for Algebraic Aggregations

In this section, we provide Delta CRDT's for the following Algebraic Aggregations : sum, average, max, min and range .Note that for the count aggregation, we use original work done by Shapiro et Al.[SPBZ11] that already contains a GCounter CRDT which we explained in Chapter 3. The GCounter CRDT is perfectly applicable for count aggregation. The data structures that we propose for our CRDT'S follow the same template more or less, and can be used to design a CRDT for more Algebraic Aggregations. We also prove that our design for the Sum Aggregation data structure is actually a state based CRDT. Proofs for other aggregations are similar, and therefore for the sake of brevity we omit them. Note that proving a data structure to be a state based CRDT is sufficient, as in an Optimized Delta CRDT, we only need to define the appropriate $\delta$-mutators.

Unless stated otherwise, we assume that the CRDT comprises of n replicas, and none of them are faulty, i.e all replicas are correct. We also restrict our discussion only to the domain of Real Numbers.

### 3.5.1 Sum Aggregation CRDT

We now provide a specification that our sum aggregation data structure. We use this specification to prove that our data structure is in fact a state-based CRDT. In Algorithm 3, the local state at each replica consists of two version vectors, denoted by *Pos* and *Neg*. The size of these version vectors is the number of replica's, which is given by n. Now, if a replica get's a value that is positive, the update gets the index of the replica, and increments the *Pos* vector at that index. Similarly, if the value is negative, the same step for *Neg*. If the value is positive, the $\delta$-mutator is simply the (key, value) pair (id, Pos[id] + value), where id is the identifier of the local replica. We call this $\delta$-mutator as $\delta_{\text{pos}}$. Similarly, if the value is negative, the $\delta$-mutator is given by (id, Neg[id] + value). We call this $\delta$-mutator as $\delta_{\text{neg}}$. The $\Delta$-group is an array of two maps pairs given by P and N. A map is a collection of (key, value) pairs. P is a map that contains as key the ids of those replicas that were received as a replicated update to be merged with the local

**Algorithm 3** Sum Aggregation Delta Based CRDT Specification

**state** : (Pos : Array[Integer], neg : Array[Integer]) ▷ both of size n
initial : Pos = [0,0,...,0], Neg = [0,0,...,0]
**update** : incrementOrDecrementBy(value : Double)
let id = *getId()*
if(value > 0) Set Pos[id] := Pos[id]+ value
if(value < 0) Set Neg[id] := Neg[id] + value
**$\delta$-mutator** : Return a (index, value) pair $\delta$
if(value > 0) let $\delta_{\text{pos}}$ := { i → Pos[id]+ value }
if(value < 0) let $\delta_{\text{neg}}$ := { i → Neg[id]+ value }
let $\delta$ = Either $\delta_{\text{pos}}$ OR $\delta_{\text{neg}}$
**Δ-group** : Returns two maps Δ
initial Δ, P, N = ∅ ▷ (k,v) pairs P for positive & N for negative
if(value > 0) P += $\delta$ ▷ $\delta$ will be $\delta_{\text{pos}}$
if(value < 0) N += $\delta$ ▷ $\delta$ will be $\delta_{\text{neg}}$
let Δ = [P, N]
**query** q : Return Integer v
let v = $\sum_i$ Pos[i] - $\sum_i$ Neg[i]
**merge** :(X: Δ-group, Y: CurrentState)
if(Δ-group.P is not ∅)
Z = $\{k \mapsto Pos[k] =: \max(Pos[k], P[k]) \mid k \in P.Ids\}$
if(Δ-group.N is not ∅)
Z = $\{k \mapsto Neg[k] =: \max(Neg[k], N[k]) \mid k \in N.Ids\}$

replica, and as value it contains the summation of all positive $\delta$-mutators. Similarly, N is a map for where all the value of each (key, value) pair is the summation of all negative $\delta$-mutators.

The query is simply the summation of all negative values subtracted from the summation of all positive values. The merge functions inspects whether the received Δ-group contains P and N. If P is non empty, this means that there are positive values that should be merged with local state. In the case where P is non empty, we loop over all the ids in P, and increment the old value in *Pos* for that id by the value P[k]. We perform a similar step in case N is non empty, but this time we increment *Neg* by the value N[k].

We now formally prove that $S_u$ = Summing CRDT is indeed a state-based CRDT. Each local state of $S_u$ consists of two version vectors *Pos* and *Neg*. As our aim is to prove that $S_u$ is a state-based CRDT, the payload for the merge function would comprise both arrays *Pos* and *Neg*. We equip a partial order $\leq$ on the set of all possible states *States* of $S_u$, where for any two states X and Y in *States*, X $\leq$ Y if and only if X.Pos *is-contained-in* Y.Pos and X.Neg *is-contained-in* Y.Neg. We say an array A *is-contained-in* in an array B(of equal size) if and only if A[i] is less than or equal to B[i], ∀ indices i ∈ {0, 1, 2,..., A.length - 1 }.

In order to prove that $S_u$ is a state-based CRDT, we need to prove the following[SPBZ11]:

1. The set of all possible states of $S_u$ form a semi-lattice with respect to the partial order $S_u$. A set L is said to be form a semi-lattice with respect to an operator $\bullet$ if and only there exists a least upper bound of any two values in L
2. The merge function $m$ is commutative, idempotent and associative
3. $m$ computes the least upper bound (L.U.B) of any two states that are passed to it
4. The local state of the replica is monotonically non-decreasing across all updates, i.e if *s2* is the state transition that happens on *s1* due to an update $u$, then s1 $\leq$ s2

**Theorem 1.** *$S_u$ is a state-based CRDT with respect to the partial order $\leq$ imposed on States and specifications given in Algorithm 3*

*Proof.* Let $s_1$, $s_2$ and $s_3 \in$ *States*.

We first prove that $m$ is idempotent, commutative and associative.
As max($s_1$.Pos, $s_1$.Pos) = $s_1$.Pos and max($s_1$.Neg, $s_1$.Neg) = $s_1$.Neg.
$\therefore m(s_1, s_1) = s_1 \implies m$ is idempotent
Let $s_1$.Pos = $[p_1^{'}, p_2^{'},..., p_n^{'}]$ and $s_2$.Pos = $[p_1^{''}, p_2^{''},..., p_n^{''}]$. As each value in these two arrays is a real number, we know that $\max(p_1^{'}, p_1^{''}) = \max(p_1^{''}, p_1^{'})$, $\max(p_2^{'}, p_2^{''}) = \max(p_2^{''}, p_2^{'})$ and so on.
Hence, $m(s_1.\text{Pos}, s_2.\text{Pos}) = [\ \max(p_1^{'}, p_1^{''})), \max(p_2^{'}, p_2^{''}),...., \max(p_n^{'}, p_n^{''})\ ] = [\ \max(p_1^{''}, p_1^{'}), \max(p_2^{''}, p_2^{'}),...., \max(p_n^{''}, p_n^{'})\ ] = m(s_2.\text{Pos}, s_1.\text{Pos})$. Similarly, $m(s_1.\text{Neg}, s_2.\text{Neg}) = m(s_2.\text{Neg}, s_1.\text{Neg})$. Basically, we exploited the fact that the *max* operation on real numbers is itself commutative.
$\therefore m(s_1, s_2) = m(s_2, s_1) \implies m$ is commutative
Similarly, $m(m(s_1, s_1), s_3) = m(s_1, m(s_2, s_3))$ because the max operation on real numbers is associative.
Hence, $m$ is idempotent, commutative and associative.

Next, we prove that $m$ computes the L.U.B of any two states in *States*. To prove this, we need to prove that $m$ computes the L.U.B of both *state.Pos* and *state.Neg* individually, $\forall$ state $\in$ *States*. We focus only on *state.Pos* as the proof for *state.Pos* is identical.
Let $m(s_1.\text{Pos}, s_2.\text{Pos}) = [p_1, p_2, ..., p_n]$. Further, let us assume the L.U.B of $s_1$.Pos and $s_2$.Pos is given by the array LUB($s_1$.Pos, $s_2$.Pos) = $[l_1, l_2,..,l_n]$. We need to show that $p_i \leq l_i$, $\forall$ i $\in$ {0,1,2,..,n-1}. However, this is also trivial as $p_i = \max(s_1[\text{i}], s_2[\text{i}])$, and the max of any two real numbers is the least possible number that is greater than the two numbers in question. As each $p_i$ itself is the L.U.B of the corresponding value at index i in $s_1$ and $s_2$. $\therefore$ $m$ computes the L.U.B for $s_1$.Pos and $s_2$.Pos . Similarly, $m$ computes the least upper bound for $s_1$.Neg and $s_2$.Neg.
$\therefore$ m computes the L.U.B for $s_1$ and $s_2$.

Next, we prove that the state transition is always monotonically increasing. Let $s_2$ be the state transition on $s_1$ after an update operation $u$. To do this, we just need to show that $s_1 \leq s_2$. An update operation $u$ will either cause an increment in $s_1$.Pos or $s_1$.Neg. We assume that the increment is in $s_1$.Pos. An increment in $s_1$.Pos means that

**Algorithm 4** Average Aggregation Delta Based CRDT Specification

**state** : (Pos : Array[Tuple[Integer]], neg : Array[Tuple[Integer]]) ▷ both of size n
Tuple[Integer] tuple : tuple.first = sum, tuple.second = count
initial : Pos = [(0,0),(0,0),...,(0,0)], Neg = [(0,0),(0,0),...,(0,0]
**update** : incrementOrDecrementBy(value : Double)
let id = *getId()*
if(value > 0) Set Pos[id] := Tuple(Pos[id].first + value, Pos[Id].second + 1)
if(value < 0) Set Neg[id] := Tuple(Neg[id].first + value, Neg[Id].second + 1)
**$\delta$-mutator** : Return a (index, value) pair $\delta$
if(value > 0) let $\delta_{\text{pos}}$ := { id → Tuple(Pos[id].first + value, Pos[Id].second + 1) }
if(value < 0) let $\delta_{\text{neg}}$ := { id → Tuple(Neg[id].first + value, Neg[Id].second + 1) }
let $\delta$ = Either $\delta_{\text{pos}}$ OR $\delta_{\text{neg}}$
**$\Delta$-group** : Returns two maps $\Delta$
initial $\Delta$, P, N = $\emptyset$ ▷ (k,v) pairs P for positive tuples & N for negative
if(value > 0) P += $\delta$ ▷ $\delta$ will be $\delta_{\text{pos}}$
if(value < 0) N += $\delta$ ▷ $\delta$ will be $\delta_{\text{neg}}$
let $\Delta$ = [P, N]
**query** q : Return Integer v
let v = ($\sum_i$ Pos[i].first - $\sum_i$ Neg[i].first) ÷ ($\sum_i$ Pos[i].second + $\sum_i$ Neg[i].second)
**merge** :(X: $\Delta$-group, Y: CurrentState)
if($\Delta$-group.P is not $\emptyset$)
$\{k \mapsto Pos[k] =: Tuple(Pos[k].first, P[k].first), \max(Pos[k].second, P[k].second)) \mid k \in P.Ids\}$
if($\Delta$-group.N is not $\emptyset$)
$\{k \mapsto Pos[k] =: Tuple(Neg[k].first, N[k].first), \max(Neg[k].second, N[k].second)) \mid k \in N.Ids$

at exactly one index in $s_1$.Pos where there was an increment. Let this index be j. Then, all other values in $s_1$.Pos are the same, but $s_1$.Pos[j] $\leq$ $s_2$.Pos[j]. $\therefore$ $s_1$.Pos *is-contained-in* $s_2$.Pos. Similarly, if the increment was in $s_1$.Neg, then $s_1$.Neg *is-contained-in* in $s_2$.Neg. Hence, state transition is always monotonically increasing.

Lastly, we prove that *States* is a semi-lattice with respect to the partial order $\leq$.

Consider $S_{max}$ = [max($s_1$[1], $s_2$[1]), max($s_1$[2], $s_2$[2]),...., max($s_1$[n], $s_2$[n])]. Then, $S_{max}$ is clearly an upper bound for both $s_1$ and $s_2$ as the value at each index of $S_{max}$ is greater than or equal to those of $s_1$ and $s_2$ at that index. Furthermore, as we argued above, the max of two real numbers $a$ and $b$ is the smallest number that is greatest than both $a$ and $b$. Therefore, $S_{max}$ is the L.U.B of $s_1$ and $s_2$.

□

### 3.5.2 Average Aggregation CRDT

We now present the Optimized Delta CRDT for average data aggregation in Algorithm 4. The local state of a replica consists of two version vectors Pos and *Neg*. The value at each index of the version vectors holds a tuple, where the first value of the tuple

is the sum, and the second value is the count. In Algorithm 4, we represent the sum as tuple.first and the count as tuple.second. The *Pos* version vector contains all those tuples for which tuple.first is a positive value. Similarly, *Neg* holds the tuples for which tuple.first is a negative value. The update function either increments or decrements the local state depending on the value passed to it. For instance, if the update value is 5, the tuple (5, 1) is pushed to *Pos*. Likewise, if the value is -5, (5, 1) is pushed to *Neg*. The $\delta$-mutator and the $\Delta$-group are computed the same way they are in Sum CRDT, the only exception is that for Average CRDT, a tuple is inserted instead of the raw value itself. The query follows the regular semantics of average operation, i.e the total sum is divided by the total count. In Algorithm 4, the sum is represented by the summation of negative values subtracted from the summation of positive values divided by the overall count. The merge functions inspects whether the received $\Delta$-group contains P and N. If P is non empty, we loop over all ids in P, and for that id we create a new Tuple. In this tuple, the first value is the max out of *Pos*[k].first and the received P[k].first. This first value represents the new sum at index k. The second value is the max out of *Pos*[k].second and P[k].second, and this value represents the total count. We perform similar steps for *Neg*.

### 3.5.3 Max Aggregation CRDT

The CRDT for Max aggregation follows a similar template to the one of GCounter. We present our specification in Algorithm 5. We have a version vector of size n, with initial values [0,0,...0]. The update function takes as parameter a value of type Double. The index for the replica is computed in the update function, and at that index, the max out of the current value at the version vector and the new update value is computed. This new max value is set to be the current value in the version vector at the said index. The $\delta$-mutator returns (key, value) pair where the key is the index and the value is the update value. The $\Delta$-group then returns that $\delta$-mutator that contains the max value. The query is simply the max of values at possible indices.

### 3.5.4 Min Aggregation CRDT

We present our specification for Min aggregation CRDT in Algorithm 6. The CRDT for Min aggregation is identical to the one of Max aggregation we described above, with two key differences. Firstly, for Min Aggregation, the partial order $\leq$ is the opposite. We say that a state X $\leq$ Y if and only if Y *is-contained-in* in X. Second, instead of taking a max of values, we take a min of values.

We have a version vector of size n, with initial values [0,0,...0]. The update function takes as parameter a value of type Double. The index for the replica is computed in the update function, and at that index, the min out of the current value at the version vector and the new update value is computed. This new min value is set to be the current value in the version vector at the said index. The $\delta$-mutator returns (key, value) pair where

**Algorithm 5** Max Aggregation Delta Based CRDT Specification

**payload** : (P : Array[Integer]) ▷ of size n
initial : [0,0,...,0]
**update** : update(value : Double)
let id = *getId()*
Set P[id] := max(P[id], value)
**$\delta$-mutator** : Return a (index, value) pair $\delta$
let $\delta$ = { id $\rightarrow$ max(P[id], value) }
**query** q : Return Integer v
let v = $\max_i$ P[i]
**$\Delta$-group** : Returns a map $\Delta$
let $\Delta$ = $\max_i$ $\delta$
**merge** :(X: $\Delta$-group, P: CurrentState)
$\{k \mapsto \max(X[k], P[k]) \mid k \in \Delta\text{-group.Ids} \}$

**Algorithm 6** Min Aggregation Delta Based CRDT Specification

**payload** : (P : Array[Integer]) ▷ of size n
initial : [0,0,...,0]
**update** : update(value : Double)
let id = *getId()*
Set P[id] := min(P[id], value)
**$\delta$-mutator** : Return a (index, value) pair $\delta$
let $\delta$ = { id $\rightarrow$ min(P[id], value) }
**query** q : Return Integer v
let v = $\min_i$ P[i]
**$\Delta$-group** : Returns a map $\Delta$
let $\Delta$ = $\min_i$ $\delta$
**merge** :(X: $\Delta$-group, P: CurrentState)
$\{k \mapsto \min(X[k], P[k]) \mid k \in \Delta\text{-group.Ids} \}$

the key is the index and the value is the update value. The $\Delta$-group then returns that $\delta$-mutator that contains the min value. The query is simply the min of values at possible indices.

### 3.5.5 Range Aggregation CRDT

The specification for range aggregation CRDT is presented in Algorithm 7. Range is the difference between the highest and the lowest value observed over a set of data points. Each replica maintains two arrays, *Max* and *Min*. Both arrays are initialized with all values set to zero. When an update is received by a replica, it computes the index of the replica. At that index, we take the max out of the update value and *Max[index]*. Likewise,

at that index, we take a min out of the updated value and *Min[index]*. Furthermore, we define our partial order $\leq$ as follows : a state X $\leq$ Y if and only if X.Max *is-contained-in* Y.Max and Y.Min *is-contained-in* X.Min. If the update changed the state of *Max*, $\delta$-mutator would be a (key, value) pair where the key is the id of the replica and the value will be the new max value. We call this $\delta_{\max}$. Similarly, if the update changed the state of *Min*, $\delta$-mutator would be a (key, value) pair where the key is the id of the replica and the value will be the new min value. We call this $\delta_{\min}$. The $\Delta$-group then picks that $\delta_{\max}$ that holds the maximum value and $\delta_{\min}$ that holds the minimum value. The query is simply the minimum of all values in *Min* subtracted from the maximum of all values in *Max*.

## 3.6 Discussion

In this section, we argue how Fargo provides aggregate correctness with or without nodes failure and the cases in which it provides fault tolerance with atleast-once-delivery semantics. Being an distributed in-network Window Aggregation SPE, Fargo must ensure that the final aggregate computed at the Root Node is always correct. We provide two arguments as to why this will mostly be the case and one argument in which we won't achieve it.

We now provide our first argument as to why aggregate correctness will be achieved. Recall that we have modelled our incremental computations as a CRDT. In Chapter 2, we discussed that in order to achieve Strong Eventual Convergence, three criterias have to be met. The criterias are (1) eventual delivery, (2) termination and (3) our data structure must be a CRDT. We satisfy eventual delivery as we replicate the $\Delta$-group within a Fargo Payload asynchronously in the background. As soon as an instance of a Sliding Window reaches its end time, we achieve termination, as no more updates will occur on the CRDT. In Section 3.5, we show cased CRDT's for different aggregations. The state-based merge that we perform on the Intermediate Nodes or the Root Node is the final state transition that our CRDT's go through. As the state-based merge is performed outside the replicas on Intermediate Nodes and the Root Node, we are playing lose with our analogy of state transitions of CRDT's. However, because of the termination clause we explained above, each of our CRDT replicas converge to the final state-based merge that is performed upstream. Therefore, we achieve Strong Eventual Convergence.

For our second argument as to how Fargo provides aggregate correctness, we assume that our failure detection is always accurate, i.e no correct processes have been tagged as being faulty. Using the *Child Nodes Fault Tolerance Protocol*, Child Nodes periodically emit Fargo Payloads that contains CRDT $\Delta$-groups, Window metadata and all the processed offsets observed for each Sensor. In the event a Child Node fails, using the *Event Re-streaming Protocol*, the new leader re-streams all the events from all Sensors that were not replicated by the failed Child Node. As long as the Sensors have enough *bufferCapacity*, we are guaranteed that no events will be lost and correct partial aggregates will be computed.

We now outline a scenario wherein aggregate correctness will not be achieved, but we

**Algorithm 7** Range Aggregation Delta Based CRDT Specification

**state** : (Max : Array[Integer], Min : Array[Integer]) ▷ both of size n
initial : Max = [0,0,...,0], Min = [0,0,...,0]
**update** : update(value : Double)
let id = *getId()*
if(value > Max[id]) Set Max[id] := value
if(value < Min[id]) Set Min[id] := value
**$\delta$-mutator** : Return a (index, value) pair $\delta$
if(update occurred on Max) let $\delta_{\max}$ := { i → Max[id] }
if(update occurred on Min) let $\delta_{\min}$ := { i → Min[id] }
let $\delta = \delta_{\max}$ AND/OR $\delta_{\min}$
**$\Delta$-group** : Returns two maps $\Delta$
initial $\Delta$, P, N = $\emptyset$ ▷ (k,v) pairs P for updates on Max & N for updates on Min
if($\delta_{\max}$ is not $\emptyset$) P := $\max_j$ $\delta_{\max}$
if($\delta_{\min}$ is not $\emptyset$) N += $\min_j$ $\delta_{\min}$
let $\Delta$ = [P, N]
**query** q : Return Integer v
let v = $\max_i$(Max[i]) - $\min_i$(Min[i])
**merge** :(X: $\Delta$-group, Y: CurrentState)
if($\Delta$-group.P is not $\emptyset$)
$Z = \{k \mapsto Max[k] =: \max(Max[k], P[k]) \mid k \in P.Ids\}$
if($\Delta$-group.N is not $\emptyset$)
$Z = \{k \mapsto Min[k] =: \min(Min[k], N[k]) \mid k \in N.Ids\}$

will still have fault tolerance with atleast-once-delivery semantics. This scenario occurs when a Child Node inaccurately detect a correct Child Node to have failed. In such an event, Fargo will conduct a leadership election and select a new leader Child Node. However, since the Child Node that is inaccurately detected to have been never failed in the first place, the new leader will use the *Events Re-streaming Protocol* to re-stream all events of the inaccurately detected Child Node. This leads to events being recomputed twice. Therefore, Fargo provides fault tolerance with atleast-once-delivery semantics.

# 4 Evaluation

In this section, we present our evaluation of Fargo. We start by discussing our methodology to measure throughput and latency along with our programming and testing environment in Section 4.1. In Section 4.2, we showcase the two experiments that we conducted to measure Window throughput and latency. We end this chapter by discussing the main findings of our evaluation in Section 4.3.

## 4.1 Setup and Methodology

In this section, we first discuss the programming and testing environment we used in 4.1.1. We then proceed to discuss the methodology we adopted from the work done by Karimov et al.[KRK+18a] in Section 4.1.1.

### 4.1.1 Setup

Our prototype for Fargo is a distributed SPE with three layers : Sensors, Child Nodes and a Root Node[1]. We have six Sensors in our Sensors Group, eight Child Nodes and a single Root Node. The Windows are created on all Child Nodes and the CRDT states emitted at the end of each Window by every Child Node is merged at the Root Node. Our experiments are run for only a single instance of a Tumbling Window of size ten seconds. Across all of our experiments, we performed a count aggregation. All of the components in our setup communicate with one another using TCP.

We programmed our Child Nodes and Root Node in the Scala programming language and used Akka[Akk] as the framework of choice. We modelled each Child Node and the Root Node as Akka actors, and each actor communicates with one another over TCP sockets. Our Sensors are programmed in the Groovy programming language and use the framework Vert.x[Ver] to communicate with the Child Node actors over TCP. Every Child Node, Sensor and the Root Node are executed as a separate jar process.

The Event Buffers in our Sensors are modelled as a Java TreeMap. Java's TreeMap is implemented as a Red-Black Tree and any search operation done on top of a TreeMap is guaranteed to be executed in *O(log(n))* time complexity. This means that any Child Node that queries an Event Buffer to request events from a specific offset on-wards is guaranteed to have fast access to the query results. Also, we make use of Vert.x scheduling capabilities to schedule the sending of events at an event level[Per]. In case a Child Node requests a Sensor to re-stream all events from a particular offset on-wards, we use Java's ExecutorService to schedule the re-streaming in a different thread.

[1] `https://github.com/ankushs92/fargo`

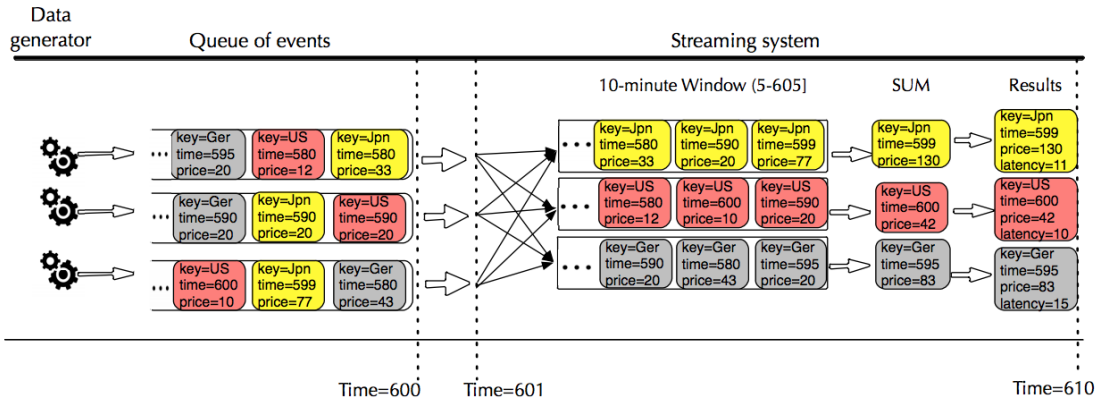



Figure 4.1: Measuring Window Latency [KRK+18a]

Each Sensor generated approximately 18,300 events per second. Thus, the total number of events emitted by all six Sensors each second is 18,300 * 6 = 1,09,800. Therefore, we expect that over a duration of ten seconds, all of our Sensors combined will produce approximately 1,09,800 * 10 which is approx 1.1 million events. We used round-robin partitioning scheme between the Sensors and the Child Nodes so that a Sensor distributes the events in a round-robin manner amongst all Child Nodes. Thus, each Sensor emits up to 18,3000/8 = 2290 or approx 2300 events to each Child Node per second.

Each event emitted by the sensor was serialized to a JSON that contained the following fields : (1) offset(Integer), (2) random value(Float), (3) timestamp(String, in UTC timezone), and (4) sensorId(String, the TCP port where the sensor was running). The size of each message always between 90-110 bytes.

Moreover, to detect Child Node failures, the Phi Accrual Failure Detector on each Child Node is set with a threshold of 1. We went with aggressive failure detection. Lastly, we did not use the leadership election that we discussed in Section 3.4.3. Our leadership election algorithm implemented in our prototype was quite naive and simple : once a Child Node fails, we simply elect a pre-defined Child Node that we hard-coded in our code. Therefore, the Window latencies that we discuss in Section 4.2 do not reflect the time it would take to perform leadership election as specified in 3.4.3.

All experiments were conducted on a MacBook Pro with 2,2 GHz Intel Core i7 processor and 16 GB RAM. However, for our experiments, we only had a free memory of approximately 10 GB to work with, as there were other processes also running on the system. Each experiment was conducted for a total of three times, and the evaluations that we present in this Chapter are an average of the three times we ran our experiments.

### 4.1.2 Methodology

In this section, we discuss the methodologies that we used to measure Window throughput and latency.

**Measuring Window Throughput**

To measure Window throughput, we first derive a baseline and then measure the impact of sending heartbeats and constant replication against this baseline. Our baseline gives us an upper bound on the throughput that our system can accommodate over a Tumbling Window of ten seconds. Our first experiment then measures the impact of increasing heartbeat rates and replication rates against this baseline.

**Measuring Window Latency**

In order to measure Window latency, we refer to the work done by Karimov et al.[KRK+18a]. In Fig 4.1, we have three Windows as shown in the left before Time = 600. Each of these Windows has three distinct keys, as shown in colours grey, pink and yellow. We limit our attention to the Window with key = Ger (the one in grey). On the right, we see that at time = 605, the Window finishes and the sum aggregation starts immediately after time = 605. Also, notice that the highest timestamp for our Window of interest is time = 595(the last grey box in the third window depicted on the right). The Window finishes the sum aggregation at time = 610. Therefore, the Window Latency in this case would be 610 - 595 = 15. Simply put, the Window latency is the time difference between the time when the computation of the window ends and the timestamp of the last event that contributed towards that computation.

In our case, since we are performing distributed and incremental data aggregation, along with the CRDT state, each Child Node sends an additional timestamp to the Root Node. This timestamp is the timestamp of the event that was the last event that contributed to the aggregation on that Child Node. When the Root Node receives the CRDT state and this last timestamp from each Child Node, it computes the Window Latency by subtracting the last timestamp from the time it processed the final CRDT state merge.

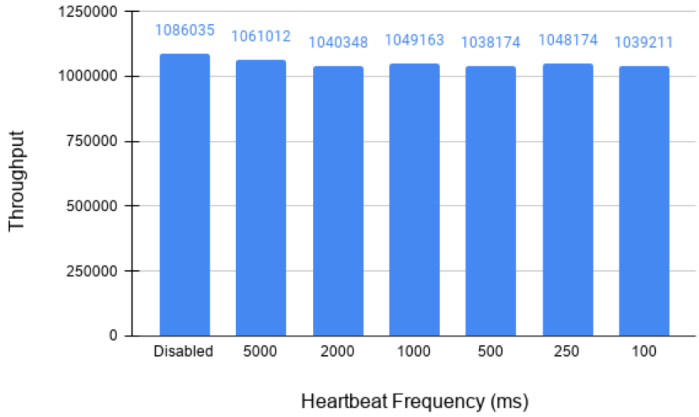
## 4.2 Experiments

In this Section, we present our evaluation of Fargo. Out first experiment measures the impact on Window throughput as we vary different heartbeat rates and replication sync intervals. Our second experiment measures the impact on Window latency as we manually fail multiple nodes.

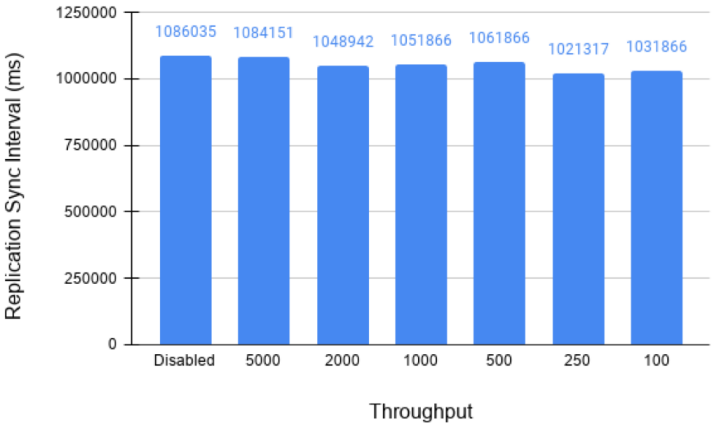
### 4.2.1 Window Throughput

We measure the maximum throughput that can be achieved by the Child Nodes Group. As our baseline, we measure the total throughput achieved by all Child Nodes with zero heartbeats sent and no replication being done. We then compare our baseline against varied heartbeat frequency rates and replication sync intervals.

In Fig 4.2, we see our evaluation with different heartbeat frequencies and replication sync intervals. Our baseline in both cases is depicted in the x-axis by 'Disabled'. In Fig




(a) Throughput with varied Heartbeat Frequencies




(b) Throughput with varied Replication Sync Intervals

Figure 4.2: Measuring Window Throughput

4.2(a), our baseline shows a throughput of 1,09,6035 with no heartbeats sent or received. We see that as we increase the heartbeat frequency rate from 5 seconds down to 2 seconds down to 100 ms, the throughput more or less remains the same. In other words, the Child Nodes Group is not impacted by the heartbeats that a Child Node sends and receives. If the heartbeat frequency is every 100 ms, a Child Node will send at most 10 * 7 = 70 heartbeats per second to seven other Child Nodes. It will also receive the same number from the seven other Child Nodes. Therefore, in total, at most (70 + 70) * 10 * 8 = 11,200 heartbeats will flow through the Child Nodes Group during the duration that our Window is running. The reason that we see virtually no increase in throughput can be explained by the fact that we model a heartbeat as a single bit exchanged over a TCP socket. This means that the total number of heartbeats contribute to 11,200 bits or roughly 1.4 KB, and thus does hog our network bandwidth.

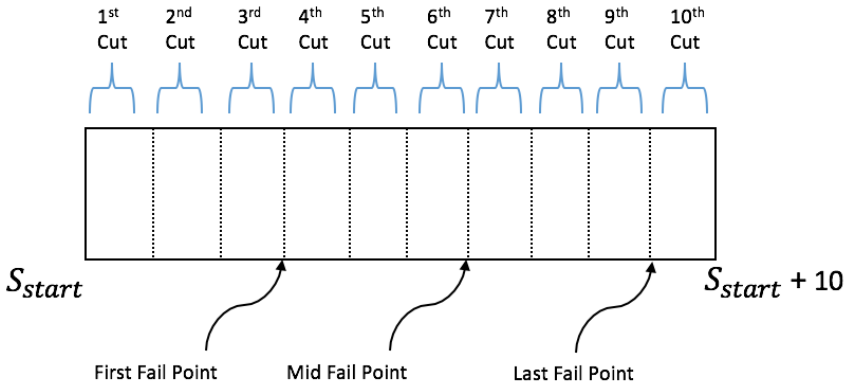



Figure 4.3: Tumbling Window of size ten seconds divided into ten Cuts [KRK+18a]

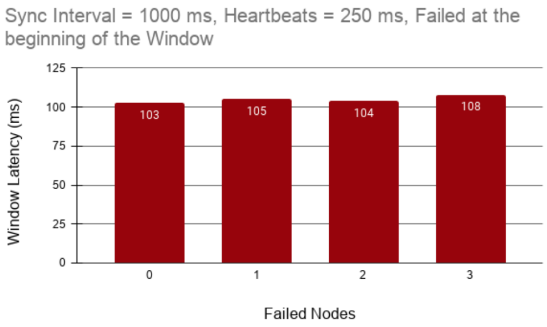
In Fig 4.2(b), we disable heartbeats and evaluate the throughput achieved by the Child Nodes Group with different replication sync intervals. Recall that a Child Node replicates a Fargo Payload to other Child Nodes. A Fargo Payload contains the latest processed Sensors offsets, the CRDT Δ-group and Window metadata. In our evaluation, we notice that there is virtually no drop in throughput as compared to our baseline as we vary the replication sync interval from 5 seconds down to one second. This is because the size of the Fargo Payload contains of a map of six Sensor ids to last processed offsets, a very small CRDT Δ-group, and Window metadata. Even when we consider the extreme case of replication every 100 ms, the computational capabilities required to serialize a Fargo Payload to JSON and the network bandwidth used to send it over the wire over a TCP socket is very miniscule. We thus conclude that the throughput achieved Child Nodes Group is unaffected by heartbeat frequency and replication sync intervals down to every 100 ms.

### 4.2.2 Window Latency

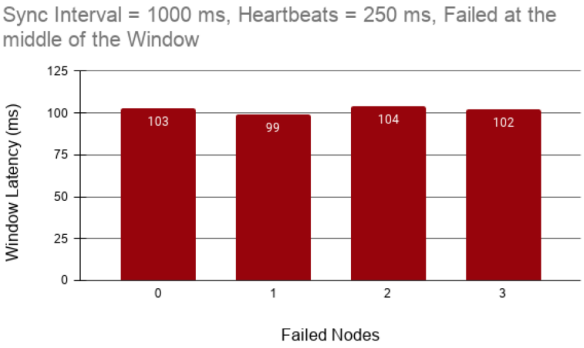
In order to measure Window latency, we first derive a baseline and then conduct our experiment by crashing multiple nodes at the during various Cuts within the Window. As our baseline, we compute the Window latency at the Root with eight Child Nodes consuming events from six Sensors. In our baseline, we disable heartbeats and replication altogether.

We first briefly discuss how and when we fail Child Nodes. In Fig 4.3, we have a Tumbling Window of ten seconds that we break into ten *Cuts*. We define three fail points during the Window where we manually fail our Child Nodes. The first fail point is at the end of the third Cut, the mid fail point at the end of the sixth Cut, and the final fail point is at the end of the final Cut. The first fail point corresponds to an early state in the Tumbling Window, the mid fail point at the middle of the Window and the final fail point just before the Window finishes.

In Fig 4.4, we have a replication sync interval of one second and heartbeat rate of




(a) Failing up to 3 Child Nodes in the beginning




(b) Failing up to 3 Child Nodes in the middle

Figure 4.4: Failing Child Nodes at First and Mid Fail Point . Replication Sync Interval = 1 second, Heartbeat Rate = 250 ms

250 ms. This means that each second, every Child Node will receive up to 7 * 4 = 28 heartbeats and it will send 7 * 4 = 28 heartbeats to seven other Child Nodes. Thus, for all Child Nodes, each second we have (28 + 28) * 8 = 448 heartbeats. The total heartbeat count flowing through $\mathcal{CN}$ for the duration of the Window is thus, 448 * 10 = 4480. Furthermore, we have a replication sync interval of one second. This means that each Child Node sends 7 * 10 = 70 Fargo Payloads to other Child Nodes and receives 7 * 10 = 70 Fargo Payloads from other Child Nodes. Thus, there are a total of 140 * 8 = 1120 Fargo Payloads flowing through $\mathcal{CN}$ for the duration of the Window.

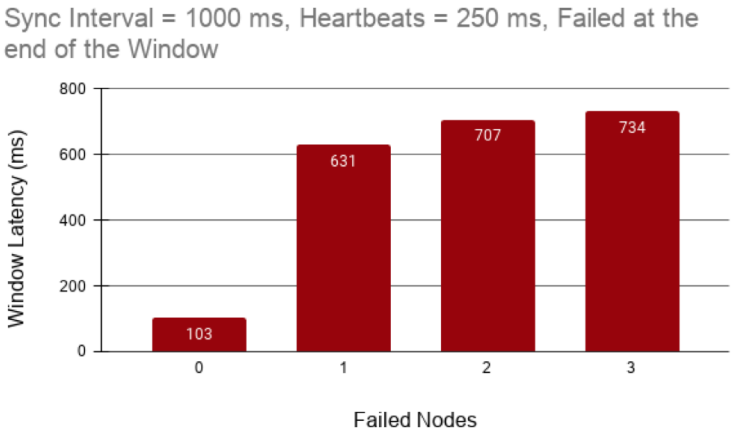
In Fig 4.4(a), our Window reaches the first fail point and we manually fail one, two and three Child Nodes. Our baseline tells us that without failing any Child Node, the Window latency is 103 ms. At the first fail point, when we fail one Child Node, we see that the latency is 105 ms, which is more or less the same as our baseline. When we fail two Child Nodes, the latency is 104 ms and it is 108 ms when we fail three Child Nodes. The fact that we have no to miniscule rise in Window latency is due to two reasons. First, recall that each Sensor emits up to 1625 events per second to every Child

Node. The failed Child Node had already computed the aggregation for up to 2300 * 2 = 4600 events for each Sensor at the end of the Second Cut. Using the *Child Node Fault Tolerance Protocol*, at the end of the second Cut, the failed Child Node replicates a Fargo Payload to all other Child Nodes where this Fargo Payload contains the latest processed offsets of each Sensor along with the CRDT Δ-group. By the end of the third Cut, each Sensor has sent up to 2300 * 3 = 6900 events to each Child Node. This means that by the end of the third Cut, when a distinct Child Node detects the failed Child Node, using the *Event Re-streaming protocol*, it only has to re-stream up to 6900 - 4600 = 2300 events from each Sensor. In total, the new leader of the partitions of the failed Child Node re-streams up to 2300 * 8 = 18,300 events. Had we disabled replication altogether, our new leader would have to re-stream 18,300 * 3 = 54,900 events. Note that from the Sensor's point of view, re-sending events is not guaranteed to be quick. On our machine, in this case, it took approximately 1 to 1.5 seconds for each Sensor to re-stream all lost events. This is due to the fact that the Sensor queries the Event Buffer to return all the events after a specific offset. Then, one by one, it re-streams these events to the new leader. The second reason we did not see a rise in Window latency is because of the size of the Window itself. As it took each Sensor approximately 1 to 1.5 seconds to re-stream all events, this time was adjusted due to the size of the Window itself. This can be understood as follows : as the Window failed near to the beginning of the third Cut, we are still left with approximately seven seconds till the Window finishes. In the meantime, our Sensors use a separate thread to re-stream all lost events asynchronous to the new leader Child Node.

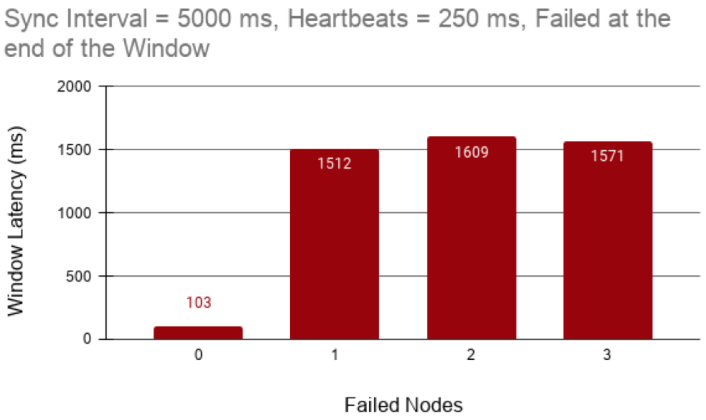
So far, we have explained the reasoning behind no increase in latency when we fail one Child Node. However, note that the same reasoning also applies if we fail two and three Child Nodes. This is because our Sensors use a different thread to re-stream all the events that are requested by the new leader. As the re-streaming is happening concurrently with the Window processing, we see no increase in latency.

In Fig 4.4(b), when our Window reaches the mid fail point, we fail one, two and three Child Nodes. We see that there is virtually no increase in latency with respect to our baseline. The reason behind why this is so is the same as we reasoned above for Fig 4.3.

In Fig 4.5, we fail our Child Nodes at the end fail point. In both cases, we see a stark rise in Window latency. In Fig 4.5(a), we still maintain the sync interval of one second, but this time, we fail our Child Nodes at the very last Cut. When a single Child Node fails, we see a latency of 631 ms, for two Child Node failures its 707 ms and for three its 734. As we mentioned before, it takes approximately 1-1.5 seconds for a Sensor to re-stream the events to the new Child Node. In this case, since we failed the Child Nodes at the very end of the Window, the time it takes for the Sensors to re-stream to the data exceeds the end time of the Window itself. In Fig 4.5(b), we have a replication sync interval of 5 seconds. This means that during the entire duration of the Window, a Fargo Payload is replicated only once. We see an even greater rise in Window latency of 1.5 seconds for a single Child Node failure. This is because in this case, the Child Node that became the leader of all partitions of the failed Child Node had to re-stream approximately 2300 * 4 = 9,200 events for each Sensor. In total, across all six Sensors,




(a) Failing up to 3 Child Nodes in the end. Replication Sync Interval = 1000 ms




(b) Failing up to 3 Child Nodes in the end. Replication Sync Interval = 5000 ms

Figure 4.5: Failing Child Nodes at End Fail Point. Heartbeat Rate = 250 ms

the new leader re-streamed about 9,200 * 6 = 55,200 events. This explains why we see a latency of 1.5 seconds for a single Child Node failure. The new Child Node leader had to re-stream about 55,200 - 18,300 = 36,900 more events than when we had a sync interval of 1 second in Fig 4.5(a). In summary, with an increase in number of events being re-streamed growing by 2x, the latency also grew 2x.

## 4.3 Discussion

In this Chapter, we evaluated Fargo's performance by measuring the throughput that is achieved by Child Nodes Group and Window latency. We observed that there is virtually

no drop in throughput when increasing the heartbeat and replication sync frequencies down to every 100 milliseconds. This can be explained by the fact that heartbeats are mere pings where the message exchanged is a single bit, and the size of the Fargo Payload is very small as compared to the stream of events that a Child Node receives from Sensors. This gives Fargo the ability to be more aggressive when it comes to heartbeat and replication sync rates. When it comes to node failures, we observed that in most cases, there is virtually no increase in Window latency for Child Nodes failure up to three nodes. We postulate that this latency will not go up for however many Child Nodes we fail, as long as there exists one Child Node that can re-stream events and is computationally capable of performing incremental aggregations on these re-streamed events. However, we also observed two cases under which fault tolerance is achieved albeit at the cost of increased Window latency. The first scenario under which Window latency might increase is if a Child Node or multiple Child Node fail at the very end of the Window. This is because no matter how fast the Sensors are in querying the Event Buffers for events and re-streaming them to the new leader Child Node, the time it would take for the Sensors to re-stream events will always exceed the end time of the Window. Second, if the replication is disabled or set to a high sync interval and a Child Node crashes at a considerable time after the last Fargo Payload replication took place, the Sensors would have to re-stream more events. This causes a stress not only on the Sensor's computational capabilities itself, but also puts a strain on the network bandwidth required to re-stream events quickly and reliably. Fault tolerance does not come for free, however, we see that in most cases with Fargo fault tolerance can be achieved cheaply. As a general guideline, we recommend that the heartbeat frequency and replication sync intervals to be set to 1/10th of the size of the Window. Setting it higher might entail an increase in Window latency as we observed in Fig 4.5.

As a last word, we would like to also mention that if there are multiple Window queries being run, the increase in latency that will be incurred by Fargo will only be for a small number of Window instances. For instance, if we have a Tumbling Window with size 10 seconds, and the Child Node fails at the end of this Window, the latency incurred will only be for this particular Window instance. As the next instance is created, the Child Nodes will already have removed the failed Child Node from their *ChildNodeRegistry* and therefore the CRDT created for the new Window instance will not contain this replica.

# 5 Related Work

Fargo is build upon the research conducted in three key areas, namely (1)Streams Processing, (2) In-Network Aggregations and (3) Replicated Data Types. In this section, we present related work in all three areas. In Section 5.1, we discuss modern Stream Processing Engines and how they provide support for Windowed Aggregations and fault tolerance. In Section 5.2 we discuss the work done in the field of In-Network Aggregations with a focus on Streams Processing. Finally, in Section 5.3, we discuss alternative approaches to Full Mesh Topology that is used in Fargo and other Replicated Data Types that may be used as an alternative to CRDT's.

## 5.1 Stream Processing Engines

In this section, we present similar cloud and fog SPE's. We do not provide an exhaustive list, we present modern state-of-the-art SPE's only.

The two of the most widely adapted SPE's in the industry are Apache Flink[CKE+15] and Apache Spark[ZCD+12]. Apache Flink is an open-sourced distributed modern SPE that allows a user to write streaming and batch queries in a data-parallel and pipelined manner[CKE+15]. It provides multiple Window Types in the form of Tumbling Windows, Sliding Windows and Session Windows, amongst others. Apache Flink provides fault tolerance with exactly-once semantics using a check-pointing algorithm which is based on Chandy-Lamport Algorithm[CL85]. Apache Spark is an open-source distributed general-purpose cluster-computing framework for batch processing[ZCD+12]. Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams[Spa]. In contrast to Apache Flink and Apache Spark, Fargo provides support for only Sliding Windows and Algebraic Aggregates and provides fault tolerance with at-least-once-delivery semantics using CRDT's and the Events Re-streaming Protocol to achieve fault tolerance with at-least-once-delivery semantics.

One fog-aware SPE that also provides fault tolerance is Frontier[OSP18]. Frontier is a distributed and resilient SPE for IoT. To our knowledge, Frontier is the only SPE for Edge Computing that provides fault tolerance. It does so by deploying multiple instances of replicated operators, called a Replicated Dataflow Graph. Its algorithm for fault tolerance is dependent on buffering data on the device itself and re-sending it to a replicated instance of the operator in case of operator failure. In the event of a node failure, Frontier replays all the events for the lost node. In contrast, Fargo replicates partial aggregates with Sensors offsets periodically to other Child Nodes. In the event of a node failure, only the events from the last processed Sensors offsets of the crashed

node are re-streamed to the new leader of the failed Child Node's Sensor partitions.

We then have NebulahStreams, a general purpose, end-to-end data management system under active development by DIMA group at Technische Universität Berlin for the IoT[ZCM+19]. NebulahStreams is a unified SPE for that provides data management from sensor in IoT, to the fog layer onwards to the cloud with functionality similar to that of Apache Flink and Apache Spark. NebulahStreams handles the heterogeneity, unreliability and elasticity of the unified fog-sensor-cloud environment. However, at the time of writing, NebulahStreams does not provide fault tolerance.

## 5.2 In-Network-Aggregations

In this section, we discuss some frameworks, both new and old that provide the capabilities to do In-Network-Aggregations.

One of the first in-network aggregations framework is The Tiny Aggregations Service(TAG). TAG follows a classical tree-based in-network aggregations approach for low-power, distributed and wireless networks[MFHH03]. TAG employs the use of pipelining aggregates by emitting an aggregate periodically. It implements basic database aggregation functions, such as : count, maximum, minimum, sum and average. TAG does not provide fault tolerance. A state-of-the-art SPE that is fog-aware and performs in-network aggregations is Disco[Dis]. In fact, our terminologies of Child Node, Root Node, Intermediate Node, Window Creation and Merging were inspired from Disco. Disco is a distributed SPE that provides support for complex windowing aggregations. Disco provides time-based and count-based Tumbling, Sliding and Session Windows. Data is streamed from Sensor nodes, aggregations are performed on independent nodes in parallel, and aggregations are pipelined upstream in a hierarchical tree-based topology where they are merged. Disco compares the performance of Holistic Aggregates and Algebraic Aggregates and experimentally shows that Algebraic Aggregates benefit greatly from distributed the aggregations across multiple independent nodes, while Holistic Aggregates do not perform well as compared to Algebraic Aggregates. This is because, for Holistic Aggregations, the computation always requires all events to be centralized on one node. Disco, however, does not provide fault tolerance. In contrast to TAG and Disco, Fargo is an SPE for in-network aggregations with its topology based on k-partite graphs and provides lightweight fault tolerance using Phi-Accraul Failure Detector and CRDT's.

## 5.3 Replicated Data Types

In this section, we present state-of-the-art approaches in replication topologies as well as Replicated Data Types.

Mergeable Replicated Data Types(MRDT) is an important breakthrough in the field of Replicated Data Types. They use invertible relational specifications of an inductively-defined data type as a mechanism to capture salient aspects of the data type relevant to how its different instances can be safely merged in a replicated environment[KPSJ19]. MRDTs allow non-commutative operations for certain data structures. Moreover, MRDT's

enable composition of merge specifications in a way that is key to deriving a sensible merge semantics for any composition of data structures[KPSJ19].

As an alternative to the Full Mesh Topology used in Fargo to perform replication, Plumtree[LPR07] provides a Tree based approach that is tolerant of large number of faults while maintaining a high reliability. Using a Tree based approach to perform replication is an interesting prospect as the number of messages that need to be replicated are substantially decreased when compared to our Full Mesh Toplogy approach. Another alternative is Thicket[FLR10], that addresses the problem of faults in a Tree based topology by constructing multiple Trees rooted at different processes. Thicket can tolerate catastrophic failure scenarios where a large fraction of the nodes in the system fail simultaneously[FLR10].

# 6 Conclusion

In this thesis, we presented Fargo, a Stream Processing Engine for fault tolerant in-network window aggregations. Our model focuses on Sliding Windows and Algebraic Aggregations. The topology is a directed acyclic k-partite graph that uses data parallelism to distribute Algebraic Aggregations across multiple nodes. Fargo provides a new approach towards fault tolerant streams processing by using Phi-Accrual Failure Detector to detect failed nodes and Conflict-Free Replicated Data Types(CRDT) to achieve Strong Eventual Consistency. Using CRDT's, we were able to replicate partial aggregates instead of raw events, thereby reducing the bandwidth and computational capacities that are required to achieve fault tolerance. Moreover, we showed that every layer in our topology is tolerant of nodes failure and our approach achieves fault tolerance with atleast-once-delivery semantics.

We first discuss the foundations of Fargo, namely Stream Processing Engines, Windowing techniques and Aggregations types, and most importantly, distributed fault tolerance concepts such as Failure Detection and CRDT's. We then provided an extensive overview of Fargo's topology that is comprised of components namely Sensors, Child Nodes, Intermediate Nodes and the Root Node. We discussed the protocols that the different layers of the topology participate in to enable (1) nodes registration and discovery, (2) Window creation and merging, (3) nodes failure detection and reaction to these failures, (4) replication of partial aggregates with the the latest sensors offsets that were processed, and lastly, (5) re-streaming only a portion of events that were not processed by failed nodes. Furthermore, we provide formal CRDT specifications for the following Algebraic Aggregations: sum, average, max, min, count and range. We gave a formal proof that showed that our data structure for sum aggregation was a state-based CRDT.

In our evaluation, we measured Window throughput and latency. We observed that setting a heartbeat frequency and replication sync interval down to every 100 milliseconds does not have any impact on Window throughput in the Child Nodes Group. Furthermore, we saw that in most cases, Fargo achieves lightweight and quick fault tolerance with no increase in Window latency in the event of Child Nodes failure. The only case where we saw an increase in Window latency was when the replication sync interval was set too high or if Child Nodes failed just before the Window finishes.

Our focus in this thesis was limited to Sliding Windows and Algebraic Aggregations. In terms of future work, we would like to integrate more Windowing types such as Session Windows into our SPE. Moreover, we would like to include more complex aggregations such as Holistic Aggregates like as median, mode, count-distinct, amongst others. As Holistic Aggregates do not lend naturally to distributed aggregations, we would like to investigate whether their corresponding Data Sketches might be a good fit for distributed aggregations using CRDT's. In this regard, interesting prospects for Data Sketches

for count-distinct query would be HyperLogLog[FFGM12] and for median would be T-Digest[DE19]. Furthermore, we would like to perform a comprehensive evaluation of Fargo's performance on a large scale. To achieve this, we aim to perform extensive bench-marking on the cloud. Lastly, our leadership election algorithm is predicated on the fact the new elected leader Child Node does not fail. Thus, we would like to evaluate other leadership election algorithms that offer better safety and liveness guarantees.

# Bibliography


[AAKO19] Abdullah, Monir, Ibrahim Al-Kohali and Mohamed Othman: *An Adaptive Bully Algorithm for Leader Elections in Distributed Systems*, pages 373–384. 07 2019.

[ABC+15] Akidau, Tyler, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt and Sam Whittle: *The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing.* Proceedings of the VLDB Endowment, 8:1792–1803, 2015.

[Akk] *Proof of Brewer's conjecture.* `https://doc.akka.io/docs/akka/current/`.

[ARC+19] Amadeo, Marica, G. Ruggeri, Claudia Campolo, Antonella Molinaro, V. Loscri and Carlos Calafate: *Fog Computing in IoT Smart Environments via Named Data Networking: A Study on Service Orchestration Mechanisms.* Future Internet, 11:222, 10 2019.

[ASB16] Almeida, Paulo Sérgio, Ali Shoker and Carlos Baquero: *Delta State Replicated Data Types.* CoRR, abs/1603.01529, 2016.

[ATAAK18] Alquraan, Ahmed, Hatem Takruri, Mohammed Alfatafta and Samer Al-Kiswany: *An Analysis of Network-Partitioning Failures in Cloud Systems.* In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 51–68, USA, 2018. USENIX Association.

[BAS17] Baquero, Carlos, Paulo Sérgio Almeida and Ali Shoker: *Pure Operation-Based Replicated Data Types.* CoRR, abs/1710.04469, 2017.

[CAP] *Proof of Brewer's conjecture.* `https://users.ece.cmu.edu/~adrian/731-sp04/readings/GL-cap.pdf`.

[CKE+15] Carbone, Paris, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi and Kostas Tzoumas: *Apache Flink™: Stream and Batch Processing in a Single Engine.* IEEE Data Eng. Bull., 38:28–38, 2015.

[CL85] Chandy, K. Mani and Leslie Lamport: *Distributed Snapshots: Determining Global States of Distributed Systems.* ACM Trans. Comput. Syst., 3(1):63–75, February 1985.

[CT96] Chandra, Tushar Deepak and Sam Toueg: *Unreliable Failure Detectors for Reliable Distributed Systems.* J. ACM, 43(2):225–267, March 1996.

[CZC+13] California, Matei Zaharia University of, Matei Zaharia, University of California, Tathagata Das University of California, Tathagata Das, Haoyuan Li University of California, Haoyuan Li, Timothy Hunter University of California, Timothy Hunter, Scott Shenker University of California and et al.: *Discretized streams: fault-tolerant streaming computation at scale.* Discretized streams | Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, Nov 2013.

[dAVB17] Assunção, Marcos Dias de, Alexandre Da Silva Veith and Rajkumar Buyya: *Resource Elasticity for Distributed Data Stream Processing: A Survey and Future Directions.* CoRR, abs/1709.01363, 2017.

[DE19] Dunning, Ted and Otmar Ertl: *Computing Extremely Accurate Quantiles Using t-Digests*, 2019.

[Dis] *disco of Iot.*

[EABL18] Enes, Vitor, Paulo Sérgio Almeida, Carlos Baquero and João Leitão: *Efficient Synchronization of State-based CRDTs.* CoRR, abs/1803.02750, 2018.

[Fal] *fallacies of distributed computing.* `http://www.rgoarchitects.com/Files/fallacies.pdf`. Accessed: 2010-09-30.

[FFGM12] Flajolet, Philippe, Eric Fusy, Olivier Gandouet and Frédéric Meunier: *HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm.* 03 2012.

[FLR10] Ferreira, M., J. Leitão and L. Rodrigues: *Thicket: A Protocol for Building and Maintaining Multiple Trees in a P2P Overlay.* In *2010 29th IEEE Symposium on Reliable Distributed Systems*, pages 293–302, 2010.

[GLME+15] Garcia Lopez, Pedro, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber and Etienne Riviere: *GLP.* SIGCOMM Comput. Commun. Rev., 45(5):37–42, September 2015.

[GM82] Garcia-Molina, H.: *Elections in a Distributed Computing System.* IEEE Trans. Comput., 31(1):48–59, January 1982.

[HDYK04] Hayashibara, N., X. Defago, R. Yared and T. Katayama: *The /spl phi/ accrual failure detector.* In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pages 66–78, 2004.

[HSS+14] Hirzel, Martin, Robert Soulé, Scott Schneider, Buğra Gedik and Robert Grimm: *A Catalog of Stream Processing Optimizations.* ACM Comput. Surv., 46(4), March 2014.

[IAM+19] Isah, H., T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine and S. Khan: *A Survey of Distributed Data Stream Processing Frameworks.* IEEE Access, 7:154300–154316, 2019.

[JBA11] Jesus, Paulo, Carlos Baquero and Paulo Sérgio Almeida: *A Survey of Distributed Data Aggregation Algorithms.* CoRR, abs/1110.0725, 2011.

[KPSJ19] Kaki, Gowtham, Swarn Priya, KC Sivaramakrishnan and Suresh Jagannathan: *Mergeable Replicated Data Types.* Proc. ACM Program. Lang., 3(OOPSLA), October 2019.

[KRK+18a] Karimov, J., T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen and V. Markl: *Benchmarking Distributed Stream Data Processing Systems.* In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1507–1518, 2018.

[KRK+18b] Karimov, Jeyhun, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen and Volker Markl: *Benchmarking Distributed Stream Data Processing Systems.* 2018 IEEE 34th International Conference on Data Engineering (ICDE), Apr 2018.

[Lam78] Lamport, Leslie: *Time, Clocks, and the Ordering of Events in a Distributed System.* Commun. ACM, 21(7):558–565, July 1978.

[LPR07] Leitão, João, José Pereira and Luís Rodrigues: *Epidemic Broadcast Trees.* pages 301–310, 11 2007.

[MA12] Murshed, Md and Alastair Allen: *Enhanced Bully Algorithm for Leader Node Election in Synchronous Distributed Systems.* Computers, 1, 06 2012.

[MDCZFC+70] M. Díaz, C. Martín, H. Zhou C. Zhu, P. Deng F. Chen, J. Rao Y. Guo, S. Ghemawat J. Dean, M. Goudarzi, T. Das M. Zaharia, V. Chang IAT. Hashem, G. Fox S. Kamburugamuve, D. Pakkala P. Pääkkönen and et al.: *Evaluation of distributed stream processing frameworks for IoT applications in Smart Cities.* Journal of Big Data, Jan 1970.

[MFHH03] Madden, Samuel, Michael J. Franklin, Joseph M. Hellerstein and Wei Hong: *TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks.* SIGOPS Oper. Syst. Rev., 36(SI):131–146, December 2003.

[MIIM12] McSherry, Frank, Rebecca Isaacs, Michael Isard and Derek Murray: *Composable Incremental and Iterative Data-Parallel Computation with Naiad.* 11 2012.

[MMI+13] Murray, Derek G., Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham and Martín Abadi: *Naiad: A Timely Dataflow System.* In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[OSP18] O'Keeffe, Dan, Theodoros Salonidis and Peter Pietzuch: *Frontier: Resilient Edge Processing for the Internet of Things.* Proc. VLDB Endow., 11(10):1178–1191, June 2018.

[Per] *Proof of Brewer's conjecture.* `https://vertx.io/docs/kdoc/vertx/io.vertx.reactivex.core/-vertx/set-periodic.html`.

[ris] *Rise of Iot.*

[SBDBHGM+70] S. Babu, J. Widom, W. DuMouchel D. Barbara, K. Salem H. Garcia-Molina, IS. Mumick A. Gupta, D. Agrawal C. Mohan, RT. Snodgrass G. Ozsoyoglu, O. Diaz N. Paton, S. Ghosh TK. Sellis and AVS. Reddy C. Yang: *Aurora: a new model and architecture for data stream management.* The VLDB Journal, Jan 1970.

[Spa] *Spark Streaming.* `https://spark.apache.org/docs/latest/streaming-programming-guide.html`.

[SPBZ11] Shapiro, Marc, Nuno Preguiça, Carlos Baquero and Marek Zawirski: *Conflict-Free Replicated Data Types.* Volume 6976, pages 386–400, 07 2011.

[SPM+11] Shapiro, Marc, Nuno Preguiça, Carlos Baquero Moreno, Marek Zawirsky et al.: *A comprehensive study of convergent and commutative replicated data types.* page 47, January 2011.

[SST+14] Soundarabai, Paulsingh, Ritesh Sahai, J. Thriveni, Venugopal K R and Lalit Patnaik: *Improved Bully Election Algorithm for Distributed Systems.* 02 2014.

[TB92] Tam, Francis and Ranjib Badh: *A Generic Failure Model for Distributed Systems.* IFAC Proceedings Volumes, 25(30):51 – 56, 1992. IFAC Symposium on Safety of Computer Control Systems (SAFECOMP'92), Zürich, Switzerland, 28-30 October 1992.

[TGC+19] Traub, Jonas, Philipp M. Grulich, Alejandro Rodriguez Cuellar, Sebastian Bress, Asterios Katsifodimos, Tilmann Rabl and Volker Markl: *Efficient Window Aggregation with General Stream Slicing.* In *EDBT*, 2019.

[THSW15] Tangwongsan, Kanat, Martin Hirzel, Scott Schneider and Kun-Lung Wu: *General Incremental Sliding-Window Aggregation.* Proc. VLDB Endow., 8(7):702–713, February 2015.

[VCG+15] Vulimiri, Ashish, Carlo Curino, P. Brighten Godfrey, Thomas Jungblut, Jitu Padhye and George Varghese: *Global*

*Analytics in the Face of Bandwidth and Regulatory Constraints*. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 323–336, USA, 2015. USENIX Association.

[Ver] *Proof of Brewer's conjecture*. `https://vertx.io/docs/vertx-core/groovy/`.

[VR03] Van Renesse, Robbert: *The Importance of Aggregation*. Volume 2584, pages 87–92, 01 2003.

[VRMCL09] Vaquero, Luis, Luis Rodero-Merino, Juan Caceres and Maik Lindner: *fog2*. Computer Communication Review, 39:50–55, 01 2009.

[ZCD+12] Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker and Ion Stoica: *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing*. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.

[ZCM+19] Zeuch, Steffen, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Bress, Jonas Traub and Volker Markl: *The NebulaStream Platform: Data and Application Management for the Internet of Things*, 2019.