# Message Dissemination with Epidemic Algorithms in Onyx

**Sondre Coldevin Madsen Basma**

Master of Science in Computer Science
Submission date: September 2017
Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer Science

# Message Dissemination with Epidemic Algorithms in Onyx

Sondre Basma

*August 24, 2017*

Version: First Version

NTNU

IDI

Distributed Systems

# Message Dissemination with Epidemic Algorithms in Onyx

Sondre Basma

*Supervisor* Svein Erik Bratsberg

August 24, 2017

# Abstract

The Onyx Platform is a data processing framework, that utilizes a masterless coordination design and a centralized log through the ZooKeeper system. At large cluster sizes, the centralized log experiences performance issues due to the large amount of read and write requests. This thesis utilizes epidemic techniques for sharing log events in order to reduce read requests to the primary log nodes. An implementation of these techniques will be presented, together with an analysis of the results. Problems with actually applying the received log events to the local state made realistic performance testing impossible, but the actual epidemic message dissemination show some promising results with a high degree of connectivity and small average shortest path between nodes.

# Sammendrag

Onyx er et rammeverk som tilbyr dataprosessering i distribuerte systemer. Onyx bruker en design som ikke er avhengig av en sentralisert maskin som bestemmer kordinasjonen mellom nodene. Dette gjør imidlertid rammeverket avhengig av en tjeneste som heter ZooKeeper. ZooKeeper tilbyr en sentralisert log som holder en total orden på hendelser. I store samlinger av distribuerte systemer opplever maskinen som drifter ZooKeeper et stort antall henvendelser, noe som utgjør et problem for ytelsen til systemet. Denne oppgaven bruker epidemiske teknikker for å redusere antallet lesehenvendelser til den sentraliserte loggen. En implementasjon av disse teknikkene vil bli presentert, sammen med en analyse av resultatene. Problemer med å få de epidemiske loghendelsene til å endre den lokale tilstanden til maskinene gjorde at en realistic ytelsestest ble umulig, men selve delingen av de epidemiske beskjedene viste noen lovende resultater som kan brukes videre.

# Contents

# 1 Introduction

The Onyx Platform is a data processing framework similar to engines such as Storm [30] and Apache Spark [27]. Their similarities lie in that the goal of these frameworks is to provide fast and efficient data processing in a distributed manner. The Onyx Platform differ from these mainstream data processing engines in some important ways. It is written in Clojure, and utilizes the functional data structures of Clojure to provide transparency for developers working with Onyx. In addition, it is designed to be masterless, implying that a single node is not designated as a coordinator or master in a cluster. However, Onyx is dependent on a centralized immutable log structure. This log structure is implemented by ZooKeeper and in clusters with a large number of nodes, the centralized log faces a considerable amount of strain through read and write requests.

The objective of this thesis is to look at one possible implementation of epidemic techniques in order to reduce the load on these primaries. The basis for this implementation is to be formed through knowledge of the structures underlying the Onyx framework. The thesis will start by investigating these factors and their theoretical underpinnings. The ZooKeeper framework constitutes the coordination backbone of the original implementation, and a good understanding of its principles is therefore crucial. How the Onyx platform utilizes these principles will also be explored, together with a more thorough investigation of its high-level design principles, as well as how its underlying message system functions. In addition a discussion of the platforms current problems will take place, justifying the motivation for this thesis. Attention will then turn to varieties of epidemic algorithms and how these algorithms might be utilized in the new implementation. A possible implementation will then be presented, as well as its current results.

Note that this thesis is a continuation of a project report delivered at IDI, NTNU in December 2016 by the same author [3]. The theoretical subjects and the motivation are the same. Chapter 2, 3, and Chapter 5, concerning ZooKeeper, epidemic algorithms and an overview of the Onyx Platform, is the same as presented in that project report. The introduction of Chapter 6 as well as Section 6.1 is the same as in the project report, while the rest of that chapter is new additions/heavily modified.

## 1.1 Motivation

A McKinsey report stated that the amount of data available has been exploding over the last few years [24], with a 40 percent projected growth as well as a 60 percent potential increase in retailers operating margins through the analytic use of big data. This has led to a data revolution in many fields, including research and the financial sector. From a developers perspective, the challenges are often described by the three V's: variety, velocity and volume [18]. Variety represent the heterogeneity and often inconsistent features of the data received, while velocity denotes the increasing speed at which data arrives, but also the speed at which an computation is expected to arrive at an answer. Volume refers to the increasing size of the data available for analysis and modeling. One must at the same time give the ability to express succinct queries and relevant presentation of the finished analysis to the end user.

While these are the characteristics of the current development of challenges related to data analysis, progress in the performance space of single computing resources has not been keeping up at a satisfying pace, limiting the ability to scale sufficiently. This has led to an increased focus in research and industry on the coordination and collaboration of computing instances to increase the scalability of applications. Big data ecosystems like Haadop [14] consists of a pipeline of components designed to deal with the challenges introduced by the big data revolution. These components include implementations to deal with storage, data ingestion, data processing and a number of other features. Although these components have different characteristics they all have the ability to utilize distributed computing resources to allow scalability.

In this context, the Onyx platform is in the same space as data processing engines like Spark [27] and Storm [30] and thus must be able to offer scale of processing as well as the other features expected by such processing platforms.

# 2 ZooKeeper

ZooKeeper is a service for coordinating processes of distributed applications. Its aim is to provide a simple coordination server that makes it possible to implement client specific coordination primitives. This is done by providing an API to manipulate wait-free data objects, as well as guarantees of FIFO client ordering of requests and linearizable writes. These properties makes it possible for clients to easily make their own coordination primitives, in fact the system can provide implementation of consensus for an arbitrary number of processes. It offers a high degree of availability and performance; this is done through the implementation of an atomic broadcast protocol called ZAB [19]. The FIFO client ordering guarantee enables clients to have hundreds of thousands of requests outstanding. This asynchronous property is useful in a number of ways as clients often benefit from being able to multitask. In such coordination systems, read operations usually dominates. ZooKeeper handles this by letting reads be handled at each server locally, though under relaxed consistency guarantees.

## 2.1 Terminology

- **Znodes** : data nodes that can be manipulated through the API.
    - **Regular** : persistent nodes that can be created or deleted only through calls by the API.
    - **Ephemeral** : These are znodes associated with a session, meaning they get automatically removed at end of a session (see below).

- **Data tree** : Hierarchical structure of the znodes.

- **Session** : A client connects and initiates a session. Requests are sent through a session handle. A session can be explicitly terminated, or be ended by the expiration of a timeout.

## 2.2 Data Model and API

All znodes are structured according to a hierarchy. This structure closely resembles that of a file system, and UNIX notation is used as paths. See Figure 2.1.

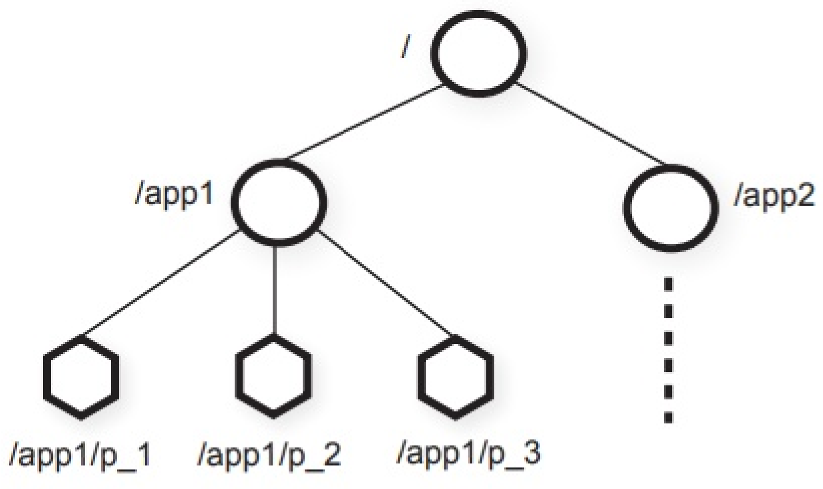

Fig. 2.1: Data tree of znodes. Source [17].

In addition to the two different types of znodes, regular and ephemeral, one can also create znodes that append a strictly increasing counter to its name. This is done by having the value of the counter increase in a subsequent manner. If parent node $a$ has child $b$, and a new child $c$ is created, then the sequence value of $c$ is always larger than $b$ or any other preceding descendant of $a$. If one uses the *sequential* flag to create a new node, this behavior will be used.

The system also provides a service called watches. Watches look for updates at a specific data object (znode), notifying the client if anything changes. Watches are single-use, meaning that after returning the notification, the client will have to reattach the watch on the object. Watches are also associated with a session, meaning that all watches are discontinued on session close.

ZooKeeper automatically stores some metadata on the znodes, such as time stamps and version number. This allows clients to only conditionally update a znode depending on which version it is. ZooKeeper also allows the client itself to store some configuration data, this could be practical in several cases where the client has application specific data that is needed by all znodes. One such case could be the location of the leader in a primary-backup protocol. The maximum allowable size for data storage at a znode is set to 1MB [32]. The API used to manipulate the data tree is presented in Table 8.9.

**Tab. 2.1:** API Calls and explanations

| Call | Explanation |
|---|---|
| **create(path, data, flags)** | Creates znode with path *path* and stores *data* in it. *flags* represent the type of znode (ephemeral, regular) and whether sequential mode is enabled. |
| **delete(path, version)** | Delete znode at *path* if *version* is correct version. |
| **exists(path, watch)** | Returns true if znode with pathname *path* exists. Watch flag enables a client to set a watch on the znode. |
| **getData(path, watch)** | Returns data and metadata belonging to znode at *path*. Watch flag enables a client to set a watch on the znode. |
| **setData(path, data, version)** | Writes *data* to znode at *path* if version number is correct. |
| **getChildren(path, watch)** | Returns the set of children belonging to znode at *path*. Can set watch to get notified if znode adds child. |
| **sync(path)** | Waits for all updates pending at the start of the operation to propagate to the server that the client is connected to. |

The next section will take a look at how ZooKeeper is able to guarantee the total ordering of the log that Onyx depends on. It will focus mainly on the agreement protocol, ZAB.

## 2.3 Implementation

In order to guarantee linearizable writes and FIFO client ordering of requests, ZooKeeper employs a pipeline of three components. The first component is called the request processor, and this component is responsible for making client requests into idempotent transactions. The fact that transactions are idempotent makes it easier to implement the agreement protocol and is also the cause of at-least-once message delivery semantics being sufficient. The request processor does this by taking a request and generating the new state of the application after applying that request. This new state is then made into a transaction and fed to the agreement protocol. Finally, the result of the transaction is delivered to the replicas. The

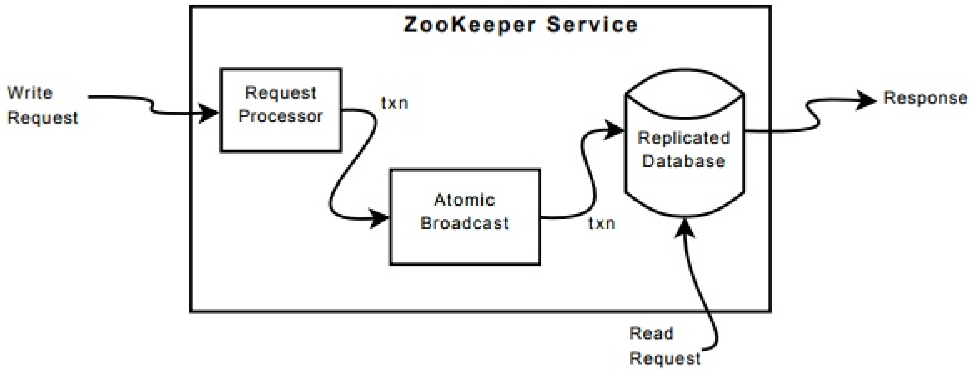

Fig. 2.2: ZooKeeper pipeline. Source [17].

agreement protocol, or ZooKeeper Atomic Broadcast (ZAB), is responsible for the total order of update transactions; total order is the ability of a system to agree on a sequence of events. It replicates state changes to achieve fault tolerance and availability while still maintaining consistency. This protocol is based on the same properties as Paxos [21] and Viewstamped Replication [23]. In other words, it depends on the *quorum intersection property*, namely that the group of nodes that is involved in one currently executing step of the protocol must share at least one node with the group of nodes involved in the next step of the protocol. This ensures that at at least one node has knowledge of what happened in the previous step. To enforce this property, it is required that a majority of nodes participates in an operation. This implies that with a group of $2f+1$ nodes, one can only tolerate failure of $f$ nodes. ZAB is a primary-backup protocol, using one primary/leader to create idempotent transactions and propose those changes to the rest of the group, the backups/followers are responsible for voting on those transactions.

In order to achieve FIFO client request ordering even during failures, ZAB had to introduce a new property that satisfies the following requirements:

- **Local primary order** : If a primary broadcasts *message1* before *message2*, then any process delivering *message2* must deliver *message1* first.

- **Global primary order** : If a primary broadcasts *message1*, and then the primary changes and the new primary broadcasts *message2*, then any process that delivers both m2 and m1 must deliver m1 first.

If a protocol satisfy both of these properties, then the protocol is said to have *primary order*. The algorithm ZAB employs to follow these requirements, is divided into three separate steps; discovery, synchronization and broadcast (normal mode). The discovery phase is associated with a new leader and tries to detect the most up to date sequence of transactions in the quorum. Because each transaction must be agreed upon by a majority of the nodes, one is able to guarantee that transactions agreed upon in the last cycle is transferred to the new leader. The synchronization phase consists of the leader proposing changes to the followers, and when a majority of followers agree to a change, the leader asks them to commit the transaction. After synchronization, the nodes go to the broadcast phase. The broadcast phase consists of servicing write requests from the client. If the leader fails, a new leader is elected and the protocol is re-initiated.

Through this algorithm ZooKeeper is able to guarantee that messages delivered have an agreed upon sequence, the total order property that the Onyx platform depends on.

# 3 The Onyx Platform

*As far as building distributed systems, as an industry, we're not very good at it at all.*

— **Michael Drogalis**
(Creator Onyx. Clojure/Conj 2015.)

The Onyx platform was created by Michael Drogalis, and is a cloud-scale, distributed, fault tolerant, high performance data processing platform that uses a hybrid interface to be able to handle both batch and streaming workloads. It is written purely in the programming language Clojure. Clojure is a general-purpose dynamic programming language, a dialect of Lisp and a mainly functional language [15], it is hosted on the Java Virtual Machine (JVM) and thus can exploit the rich ecosystem of Java libraries. M. Drogalis created Onyx on the concept that current distributed processing frameworks were based on an idea that led to diminishing developer ability to reason about the data flow of programs on these platforms [11]. The model in question consists of the tight coupling between behavior, data flow and side effects together with the use of abstract language constructs. This model is used in processing frameworks such as Storm [30] and [27]. In using these frameworks allows the developer to specify behavior (functions), data flow and side effects (like connecting to a database or logging) in one program using abstract language constructs like Resilient Distributed Datasets [31] through an API. This tight coupling between data and language constructs results in a loss of ability to understand the different parts of the program in isolation, and sets the object of computation, data, in the background.

The idea of separation of entities to enhance informal reasoning about a program is not a new one and has been discussed in discourses like [26]. In addition, with replacing abstract language constructs with simple data structures like vectors and maps one minimizes complexity and increase the informal reasoning abilities. The increased ability to reason about the program has been the main motivation behind Onyx. The distribution model of Onyx is based on a *masterless* protocol, with no peers taking coordination responsibility. The coordination layer is implemented in ZooKeeper. This chapter will start by taking a look at the Data Model and API of Onyx, and then investigate the underlying low-level architecture and algorithms.

## 3.1 Data Model and API

Units of data in Onyx are represented by *segments*. Segments go through a series of transformations; these transformations are just plain clojure functions. The transformations receive segments and return segments. Relations and flow between transformations are defined in a directed acyclic graph (DAG) contained in an idiomatic clojure vector of vectors. The inner vectors contain identifiers for each node of the graph, representing input, processing functions and outputs. Each node in the graph represent a *task*, the smallest unit of work in Onyx.

**Listing 3.1:** Pseducode: Vector of vectors to represent a DAG

```
[ [input_1, processing_1],
  [input_2, processing_2],
  [processing_1, processing_3],
  [processing_2, processing_3],
  [processing_3, output] ]
```

This structural specification is called the *workflow*. The fact that the workflow is created by using idiomatic clojure constructs makes code and tool reuse easy; any tool that the developer uses to work with DAGs would be easily employable to work with this structure. The workflow is defined separately. This separation is not only spatial but temporal and this brings flexibility; the workflow can be created independently of runtime and deployment, and then compiled to a representation that Onyx understands.

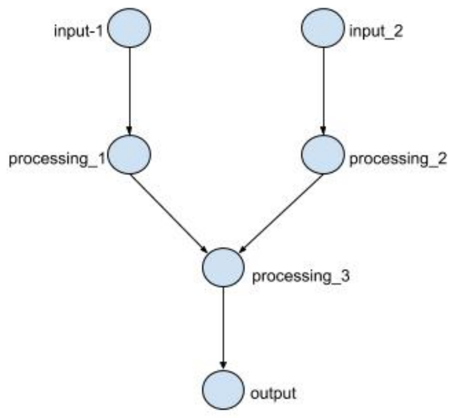

**Fig. 3.1:** Representation of workflow as a DAG.

In many situations a developer might want to determine the direction of data based on state. Onyx uses a construct called *flow conditions* to handle these scenarios. Flow conditions directs segments of data based on where the data is coming from, where the data is going and a specified predicate function. Flow conditions are represented by a clojure map for each of these entries.

In order to bind workflow definitions with actual implementations of these definitions, Onyx uses a construct called the *catalog*, the catalog contains vectors of maps. The catalog connects functions with positions in the workflow and provides these tasks with parameters, configuration and documentation; the catalog is a sequence of task specifications. This structure allows the parametrization of functions separate from other parts of the program, allowing developers to provide parameters in isolation.

The modification of state during run time is needed in order to deal with necessary side effects like logging or database connections. The facilitation of such modifications is provided by a feature called *lifecycles*. Lifecycles allow the developer to inject arbitrary code at different stages of processing on the nodes in the cluster. At each such stage a context map containing a number of useful entries is made available, the developer can then create functions to read or modify these entries to execute wanted side effects. The stages made available are before and after setup/execution of each task, in addition to stages before and after each batch of segments. These lifecycles are specific to one peer and task, and is not a coordinated effort. However, one can specify that a lifecycle is run for every task on a peer. .

In addition to the structures already described one would also need a way to analyze streams of data. Capturing and analyzing data streams presents a challenge, as streams are unbounded. This introduces problems like straggling/disordered data and difficulty of guaranteeing idempotent operations. A common technique used to process unbounded data streams is called *windowing*. Windowing allows the application to divide the unbounded data set into finite pieces and do computation on these pieces. Onyx features several different windowing techniques; fixed, sliding, global and session. The different features and implementations of these windowing techniques is out of scope for this discourse but can be reviewed at [12]. These windows support aggregation functions like sum, min, max, average and by-key. Each window is associated with a specific task over which the window operates. Each window is also associated with a *trigger*, the trigger specifies when the data that are currently in the window is to be moved to a place specified by the user.

The combination of the workflow, catalog, flow conditions, windows and triggers is known as a *job*. Each task is associated with one job, and a node only executes one job at a time. The data model and API of Onyx presented here are completely driven by idiomatic data structures and the transfer of plain clojure functions that makes it possible to separate these entities from deployment and runtime environments, and enables better understanding of the data flow. Section 3.2 will investigate how this is possible to do in a distributed masterless system by taking a look at the underlying architecture and algorithms.

### 3.1.1 At-least-once delivery

The messaging algorithm in Onyx is based on the concepts employed in the Storm [30] messaging algorithm. One can represent the processing of segments as a DAG, like the workflow described in Fig 3.1. Each segment in the DAG is associated with a UUID that tracks it through its lifetime, all segments are also given an acknowledgment value, which is a random 64-bit integer. The processing of these segments are distributed throughout the cluster so in order to maintain state, each segment also has an assigned peer that maintains state through an *acking daemon* [12]. The algorithm utilizes the canceling properties of the XOR operation when it is applied to two equal sequences.

When a segment is generated it transmits the generated acknowledgment value to the acking daemon, the XOR operation is then applied to the acknowledgment value together with any previous value in the state. When a segment is completed it transmits the acknowledgment value one more time and the acking daemon employs the XOR operation once again on the shared state, effectively canceling out the initial value sent from that segment. The result is that for a given segment tree, when the shared state of that tree is zero, one knows that it has finished executing. Failure of a peer processing a segment is handled by keeping the root segment until the tree has finished. A timeout detection mechanism will notice a segment processing failure and cause the tree to be replayed from the root segment [10]. This together with idempotent segment transformations enables Onyx to deliver an at-least-once processing guarantee.

## 3.2 Low-level Architecture

The Onyx architecture is a masterless peer-to-peer structure. This is motivated by the fact that the traditional master-slave model has some inherent limitations, these limitations stem from the fact that one gives one node the coordination responsibilities resulting in possible overload and a single point of failure. Although these limitations are mitigated by many different strategies that are thoroughly reviewed in literature, the perceived stagnation in the distributed coordination research space inspired the creator to seek out different architectures as an experiment [8]. This resulted in an architectural structure that have peers centered around an immutable append-only log distributed by ZooKeeper [17], as such the architecture is not completely masterless. Onyx leverages the strict order properties that ZooKeeper guarantees to let peers write and read from this log. Each peer has a local replica that maintains cluster state, including information about every other peer in the cluster. This organization is reflected in Fig. 3.2.

The messaging layer between peers is implemented by using Aeron [25], this is a messaging library that is reliable, handles package loss and out-of-order arrivals. It also handles many different transmission media like TCP, UDP and PCI-e.

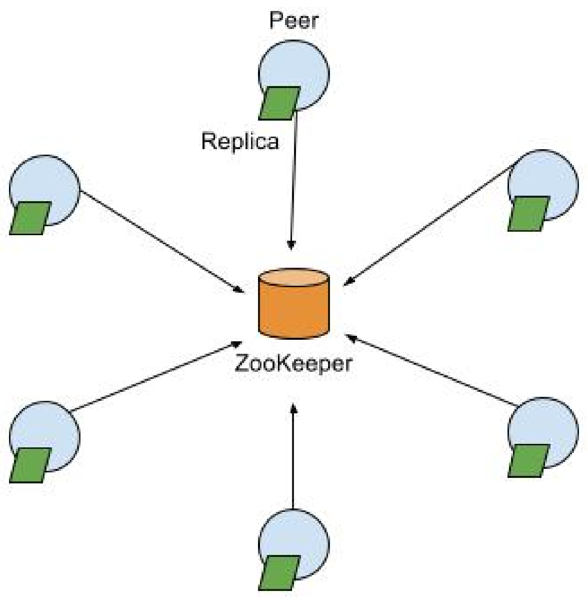

Fig. 3.2: Representation of ring structure and centralized ZooKeeper coordination layer.

### 3.2.1 The Log

The log implementation is based on a directory of persistent sequential znodes and their monotonically increasing counters. Each log entry is represented by such a node, and peers are notified about changes to the log by setting a watch on the directory. Changes to the log are proposed by each peer individually, and each change represent a change to the cluster state. Such a change is called an *event*, and events is typically submitting a job, deleting a job, a peer joining the cluster, a peer leaving the cluster etc. Each peer is notified of changes through its *inbox*, as events are appended to the log, each peers inbox receives a message containing the event, this done through a thread that is listening for notifications from the zookeeper directory. When a peer wants to write to the log, it composes a message describing the event and puts that message in its *outbox*. The first event of the log is contained in a znode at a fixed address, called the *origin*.

Each log entry is represented by a map with keys that bind to a function and its arguments. The function is a plain clojure function that is idempotent and deterministic with no side effects, only returning a new replica with the updated state. This new replica is then stored as the peers local replica. The combination of these idempotent, deterministic functions and the totally ordered log causes peers to be able to independently utilize log entries at their own pace. In order to actually perform the changes represented by the log entry the peer takes the new and old replica value and calculates their difference. This difference is then used to decide on two things:

1. Whether the peer should react or not. In some cases the difference should cause the peer to append an event to the log.

2. What side effects the peer should execute. This could be writing to a communication channel or talk to ZooKeeper.

Each peer has an ID that is reflected in the state change if this peer is to perform any exclusive side-effects.

### 3.2.2 Garbage Collection

The process described here, with peers adding entries to an append-only log requires infinite space for log and local replica storage. Onyx solves this by invoking a garbage collector at a periodical basis. The garbage collection process consists of a peer adding an entry to the log, the entry contains that peers ID. The idempotent, deterministic function that is part of the log entry instructs each peer reading into to compress the local replica state. The ID contained in the entry permit the calling peer to perform the side-effects consisting of compression of the log without any

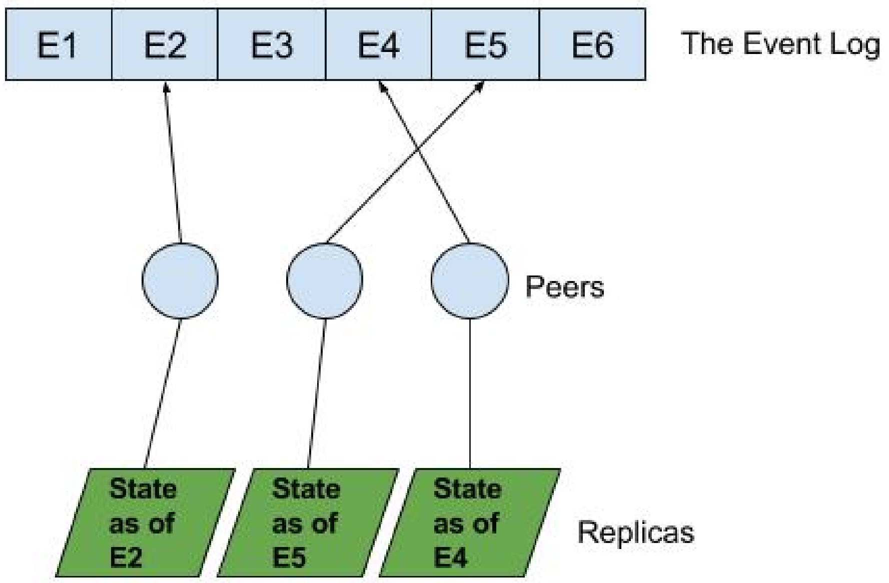

Fig. 3.3: Graphic presentation of peers reading from the event log independently with state in local replicas.

other peer attempting to do the same. The compression of the log is done by the garbage collector caller taking each log entry from the origin and up to the last one, creating a new replica state and store it in the origin znode. Then it deletes the redundant events in the log.

### 3.2.3 Cluster Join

The process of a new node joining the cluster is executed by the asynchronous interaction of three events. It utilizes a ZooKeeper directory of ephemeral znodes for every peer in the cluster. The coordination of this process is done through setting watches on these znodes in a ring structure, with each peer being responsible for monitoring one other peer. Fig. 3.4 visualizes this responsible/dependent relationship.

The peer that is looking to join the cluster starts by playing the log, creating its own local replica. Reactive responses are buffered in the outbox of this new peer, as these reactions might lead to bad state if the new peer later decides to abort

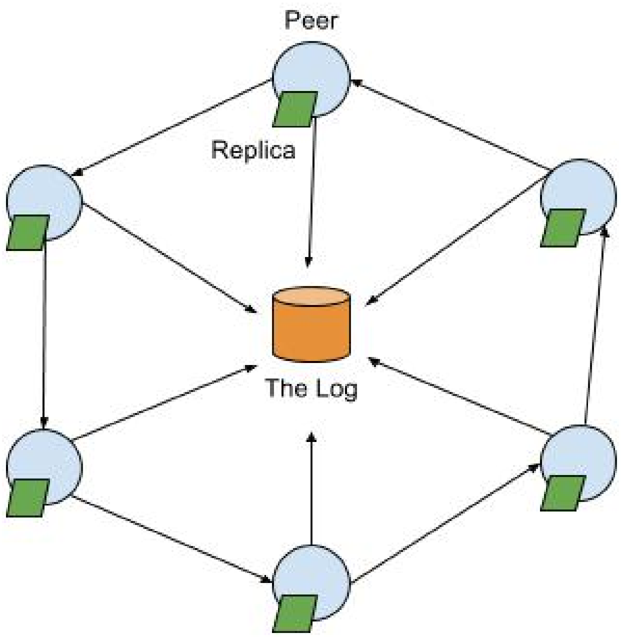

Fig. 3.4: Representation of ring structure and responsible nodes.

its attempts at joining the cluster. The new peer also emits an event to the log, signaling that it wants to join the cluster. When the nodes already fully joined in the cluster encounters this message they elect one of them to be responsible for the new node that is joining. This election is done through using the modulo function on the message id and a sorted list of peer ID's, uniformly electing the same peer to be responsible. The peer that is chosen as the responsible one sets a watch on the znode representing the joining node, and sends an event to the log notifying that it has taken responsibility. When the new peer encounters this event in the log it sets a watch on the node that the responsible node has been previously responsible for, and sends an event signaling that it has been accepted into the cluster. This stage is reflected in Fig. 3.5.

When the cluster encounters this third event, they add the new node to the cluster state, and the responsible node removes the watch on the node it was previously responsible for, concluding the join operation. Now the new peer can start to flush

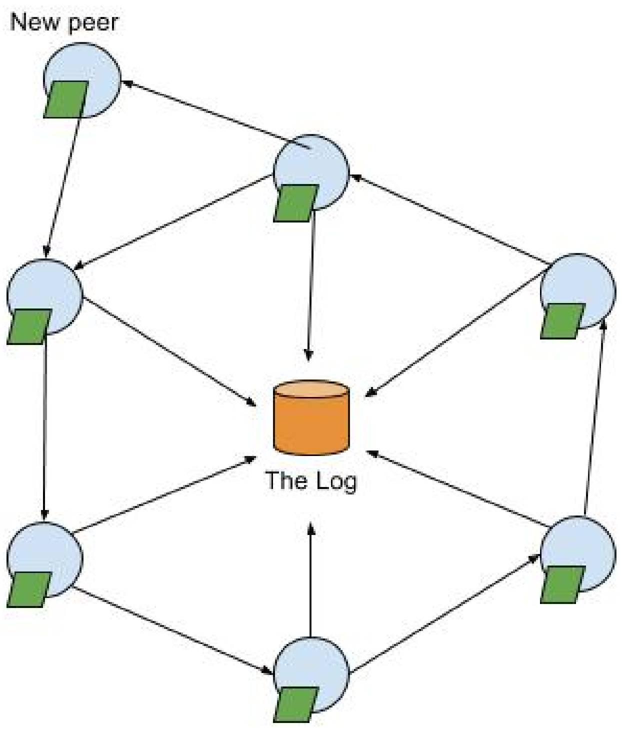

Fig. 3.5: Visualization of new node joining the cluster, with arrows between peers representing watches on znodes.

the buffered reactive messages and participate in cluster activity. The pseudocode for for a new peer and already fully joined peers are presented in listing 3.2 and 3.3

Listing 3.2: Pseudocode: Executions of peer that are joining.

```
% E1 : prepare join cluster event
% E2 : notify join cluster event
% E3 : accept join cluster event
send E1 to the log
play log and buffer reactive messages
if cluster-size == 0:
        set this node to cluster state
when encountering E2 in log:
        add watch to peer with ID contained in E2
    send E3 to the log
    flush buffered reactive messages from outbox
```

Listing 3.3: Pseudocode: Executions of an existing node in the cluster

```
% E1 : prepare join cluster event
% E2 : notify join cluster event
% E3 : accept join cluster event
when encountering E1 in the log:
    let ID = ID of peer responsible for new peer
    if this peers ID == ID:
        add watch to new peer
        send E2 to the log
 when encountering E3 in the log:
    add new node to cluster state
    if this peer is responsible:
        remove watch on previously watched peer.
```

### 3.2.4 Dead peer removal

The detection of dead peers in a ring structure as depicted in section 3.2.3 is quite straightforward. Given a dead peer, the node responsible for that peer will detect its failure with the watch on the given znode. When the failure is detected, the responsible peer will write an event to the log that signals the failure. Because the cluster state is reflected in the replica, each peer is able to compute which node should be watched by the peer that was previously responsible for the now failed node. In the case of serial simultaneous failures, this process will just repeat itself and thus tighten the ring.

### 3.2.5 Scheduling

Scheduling is concerned with resource allocation depending on the user needs. Onyx provides several different schedulers [12] for both jobs and task. One of the consequences of the architecture chosen is that each peer needs to do the scheduling of jobs and tasks individually. Each peer has to read from the event log and suggest itself as the executor for a job. This is done by the *job scheduler* and *task scheduler* in Onyx. The job scheduler has the overall responsibility for cluster communication like volunteering for tasks, and each instance runs such a scheduler. The task scheduler has responsibility for coordinating tasks and communicates with the job scheduler when it is needed. Each peer reads from the event log and by doing a transformation on the local replica state is able to detect where it is most needed. This requires knowledge of every other peer in the cluster.

## 3.3 Challenges

In the document located at [9] the developers outline a number of possible challenges related to the scalability of the coordination layer in Onyx. With large cluster sizes containing 1000 to 10000 nodes, the developers envision scalability problems especially centered around ZooKeeper contention. With ZooKeeper acting as reader/writer of the log and maintaining watches on all nodes, the load on the ZooKeeper layer might be too much, leading to slow execution. It is not simply a matter of adding new nodes to the ZooKeeper server ensemble as this will also affect the performance of modifying coordination state.

Related to this is the join process described in section 3.2.3. With three phases for each peer joining, and with each phase requiring read and writes to the log this is quite an expensive procedure that is envisioned to scale poorly at large cluster sizes. The join phase also requires a joining peer to replay the whole log to create its local replica, depending on how often the log is trimmed by the garbage collection process, this might be a lengthy sequence. The currently implemented coordination algorithms also require that complete cluster information is contained in each peers local replica that is preserved in main memory, with several thousand nodes this might become a storage problem.

The nature of the chosen architecture requires that expensive operations are executed at every node, while this is a good feature contributing to availability, it has an impact on performance. This especially regards the scheduling process as described in section 3.2.5.

It is the challenges described in the first paragraph that will be addressed in this thesis.

# 4 Aeron

Aeron is a messaging system with focus on performance, it is open source and utilizes Java as its underlying programming language. Aeron is an attempt to adapt to the evolving computer industry with multi-core, multi-socket, changing transmission mediums and increasingly large amounts of communication necessary. The endpoints have to scale as never before [29]. Aeron does this by making some compromises, the library do not support topics, like Kafka [20], but can instead be looked at as an improvement on TCP, following the ISO layer 4 transport standard and supporting multi cast delivery and some other features that TCP does not. Aeron does not support guaranteed delivery, meaning that if an endpoint dies, the message delivery would not be retried. Aeron constitutes the underlying messaging platform of Onyx.

## 4.1 Design

From a top-to-bottom perspective Aeron is designed around the concept of channels, and streams within those channels. Each stream is individually identifiable and independent within a channel. The components that write and read from a stream, are called publishers and subscribers. This would mean that each Aeron client needs a publisher to write to a stream, as well as a subscriber to read from that stream. Aeron is not designed to have a larger number of channels or streams, number of streams should not exceed into the hundreds, or else one will get a performance hit [28].

The publishers and subscribers need to interact with the Aeron media driver, which runs in its own thread. The significance of this is that one only need one driver per

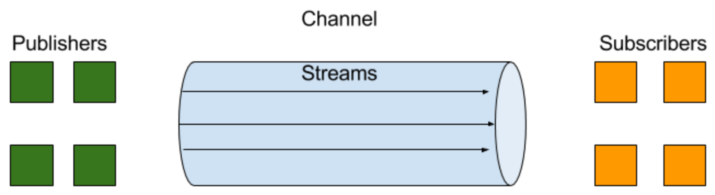

**Fig. 4.1:** Representation of how streams and channels work in Aeron.

machine, so if one is to have several Aeron clients running on different cores of the machine, only one media driver is needed. The media driver consists of senders and receivers. The sender component is responsible for writing to the assigned transmission media, be it TCP, UDP or others. While the receiver is responsible for reading from the assigned transmission media. In addition, both the client and driver have a component that is called the conductor, this component is responsible for everything that is not related to the actual sending of messages, mainly relaying system coordination events.

Aeron keeps track of messages, channels and streams through a centralized distributed log. This log is maintained in a directory which the media driver creates at startup.

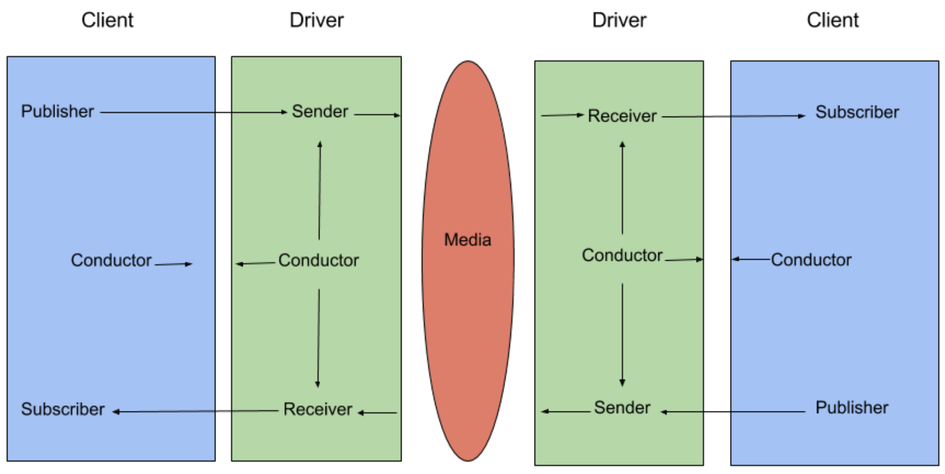

**Fig. 4.2:** Representation of Aeron architecture

## 4.2 Usage

In order to create a functioning Aeron ensemble one needs three parts; the media driver, a subscriber and a publisher. The media driver can be stand alone, or embedded into the application. The media driver creates the directory that is used for log storage. When the subscriber and publisher are initiated they need to be given a context that contains the location of the media driver directory. In addition they have to be given the URL of the channel they are to use, as well as the identification of the stream within that channel. The channel URL consists of the transmission protocol, host name and port:

$$aeron : udp?endpoint = 192.168.0.1 : 40456$$

The embedded media driver is useful for local testing, as it eliminates the need for starting a separate driver from the application. In other situations however, there is the possibility of several peers running on one machine, and as such the embedded driver is not suited for production purposes.

# 5 Epidemic Algorithms

Epidemic algorithms arose from the mathematical modeling of infectious disease spreading in populations [1], not with the goal of reducing infection spread, but to encourage it. The application of these concepts is also known as gossip protocols and such concepts have been utilized in the discrete math field [2] and the computer science field [7] for quite some years. It has been using a probabilistic model to make sure that information spreads trough a system and as such guarantees eventual consistency. Eventual consistency has become a popular model in distributed systems [4], as the model offers a good trade off between availability, the ability to handle network partitions and the aforementioned consistency. Scalability is intrinsic to the epidemic model as the model actually gets more robust with a higher number of participating nodes and work is naturally distributed, part of this is because such protocols give individual nodes the ability to make decisions without requiring coordination except from direct feedback. The main characteristic of these protocols is that participants are gossiping to other participants via some random distribution.

## 5.1 Concepts

An example of a simple epidemic model is is one which a node, when it decides to broadcast a message, selects a random set of nodes from the global ensemble of nodes and compares state with that node. If there are any differences, the difference will be resolved by transferring state information. This overly simplistic model is referred to as anti-entropy by [7] and has some apparent limitations which will be explored in the subsequent sections.

Borrowing terminology from the epidemic literature [7] a node that has received a certain message is said to be *infected* with respect to that message, while a node that has yet to receive the message is *susceptible*. A node that has received the message and is no longer interested in transferring it to other nodes is said to be *removed*. In the simple epidemic algorithm presented above, participants only have two of these states, either infected or susceptible. The mechanism that initiates message transfer can be done in two ways; the node can try to discover other nodes that have more recent messages and request them to transfer, or the node can try to discover nodes that has less recent messages and transfer the updated message to them. The latter is called *push* and the former *pull*, there is also a combination of the two that is referred to as *push-pull*.

One of the drawbacks of the algorithm described in the start of this section is that a participant will continue to attempt the spread of messages that are outdated, i.e participants have no way of knowing how relevant a message is to the other nodes in the system. These unnecessary attempts have a negative effect on performance. Such deficiencies could be mitigated by introducing the third state property denoted as removed, a removed participant is not interested in spreading a message any further. A participant should set itself to removed with respect to a message based on the perception of how widespread that message is in the cluster. Such a perception could be implemented in a number of ways, one method that mimics such information is to simply set the probability of setting a node to removed for each message transmission proportional to $1/k$ where $k$ is an integer that is configurable. One drawback of introducing probabilistic variables is that one runs the chance of an update not spreading to all participants, all infected nodes active in spreading the message might set themselves to removed before all susceptible nodes in the cluster have received the message. However, through varying the $k$ parameter one can make the probability of such an event occurring arbitrarily low, in effect ensuring that all nodes receive the update but at the cost of traffic as a result of a higher number of redundant messages. In [7] a more thorough mathematical modeling and analysis is done on the variation of this parameter.A more accurate model of cluster state with regards to message relevance might be implemented by introducing *feedback* between two nodes. This feedback mechanism gives direct information about whether a participant has received the message previously, and the node receiving feedback might stop transmitting a message after $k$ occurrences of such feedback.

The methods described up until now all assume that information pertaining to the global ensemble of participants is available to all nodes, this assumption is not scalable as the storage space associated with such information scales poorly, in addition one has to be able to maintain this information. Research to deal with this problem centers around *membership protocols* and will be the topic of discussion in the next section.

## 5.2 Membership Protocols

Membership protocols dealing with epidemic protocols attempts to connect participants in a decentralized fashion, while still maintaining the resiliency inherent to epidemic protocols and keeping traffic between nodes to a minimum. Partial views have been introduced as a means to cope with these difficulties [6, 5, 22], and such mechanisms aim to maintain a random partial subset of all participants available for communication from one node. One especially important feature of such partial views is that they manage to connect the whole ensemble of nodes, not leaving any participants isolated. In [22] the authors introduce a number of metrics useful for measuring the degree of quality associated with a partial view:

- **Connectivity** : The ability of the partial view to connect all nodes, not leaving any nodes in isolation.

- **Degree distribution** : In asymmetric partial views this is defined by two terms; *in-degree* is the number of nodes that has *this* node in its partial view, while *out-degree* is the number of nodes that *this* node has in its partial view. These terms measure the influence and reachability of a node, and should be equally distributed in order to maintain connectivity.

- **Average path length** : Defined as the average of all shortest path between pairs of nodes. This metric measure distance, and is important to keep low for performance reasons.

- **Clustering Coefficient** : Each node has a cluster coefficient that measure how densely connected that node and its neighbors are. This is done by taking the number of connections between that node and its neighbors and dividing it by the maximum connections possible. The average of these cluster coefficients gives a clue to how many redundant messages will be sent as well as whether parts of the system might become isolated.

- **Accuracy** : The number of a nodes neighbors that has failed, divided by the total number of neighbors. The average accuracy of each node is an important measure in order to determine the size of the partial views.

The number of participants that one want to relay a message to is called the *fanout*, and the size of the fanout is crucial to maintain good measures of the properties mentioned above. A high fanout gives good resiliency in the face of failures, but also introduces more redundant messages. Several fanout strategies have been explored in the past, with examples like [5] using fixed static fanouts and [6] using dynamic scalable partial views. The next section will explore a membership protocol called HyParView (Hybrid Partial Views) [22].

### 5.2.1 HyParView Membership Protocol

This protocol attempts to maintain a low fanout value, while still maintaining good information dissemination and failure resiliency. The problems of using a low fanout value is related to the in-degree measure of a node, a node with low in-degree will be subjected to more isolation and thus affect the global reliability of the system. The HyParView protocol solves this by introducing two distinct views, one *active view* and one *passive view*. The active view is small and constitutes the nodes in which one will directly communicate with, the connection is reciprocal and thus symmetric ensuring that every node has an in-degree of at least one. Each node in the active view will be contacted in the case of an update, and this together with an in-degree larger than zero ensures that every node will receive the update given enough time. Given that the active view is small, it does not give much resiliency in the case of failures. To remedy this, a larger passive view is utilized, this is solely used as a backup in case of failures.

Because each node in the active view is contacted at every update, failures are quickly recognized and a node in the passive view is used as a substitute. When a participant notices that a node is not responding it will choose a random substitute from its passive view and send a request to connect, if the sending node has no remaining nodes in its active view the request will be sent with a high priority, otherwise it will be sent with a low priority. If the request has high priority, the receiving node will immediately add it to its active view, if need be kicking out an already existing member. When the sending node is added to an active view by an accepting node, it will remove the accepting node from its passive view and add it to its active view. If the request received has low priority, the receiving node will only allow it into its active view if there is a free slot. A node that experiences being removed from an active view will be notified and remove the offending node from its active view.

The passive view is maintained by periodic *shuffling* with other participants, this maintenance procedure makes sure to include participants from the active view of nodes, as this increases the probability of having active nodes in the passive views and ensures that nodes that are no longer active will eventually be removed from circulation. Each such procedure is associated with a *time to live* integer value that is decreased for each round of sharing. A node that starts the shuffle procedure will request one node at random from its active view to participate, sending nodes from both its active view and passive view. The receiving node will then accept to shuffle only if two properties are satisfied; the time to live of the request has reached zero or it has only one node remaining in its active view. If the shuffle request is accepted the two nodes will exchange information and shuffle their partial views. If the shuffle request is rejected, the receiving node will forward the shuffle request to one random node from its active view, with the decreased time to live value.

When a node wants to join the cluster, it stars by contacting one of the already joined nodes. The contact node then adds this node to its active view, if necessary removing a node from its active view. It then forwards the join request it has received with some additional parameters; its own identification, identification of the new node and a time to live value. This forwarding is sent to each node in its active view (excluding the newly joined node). Each node in the system maintains two variables related to the join procedure, a random walk length associated with the active view and a random walk length associated with the passive view. The time to live value sent initially is set to the random walk length associated with the active view, and when a node encounters a time to live value that is zero, it adds this node to its active view unconditionally, this ensures that the new node will fill its active view eventually. A node will also accept the join request if it has only one node in its active view. The joining node will be added to another nodes passive view if the time to live value is equal to the already configured random walk length associated with the passive view. Together the two random walk length values ensure that a newly joined node will be distributed in both the passive and active views of already participating members.

One important aspect of this protocol is the interplay between the active view and the passive view. When a node determines to remove another node from its active view, the removed node is also placed in the passive view, and is thus still a part of the shuffling procedure that continues to spread its presence in the cluster.

# 6 Epidemic Techniques in Onyx

This chapters focus is on applying the epidemic concepts to the Onyx platform in order to reduce some of the challenges the platform is facing. The challenges faced by the platform center around ZooKeeper contention, costly join and scheduling procedures, and replica size. ZooKeeper is very actively used in the current implementation as a coordination layer, this includes reading/writing to the log and detection of failure. The coordination is centered around an append-only log that is trimmed by garbage collection processes. Any modifications to the coordination layer must maintain this totally ordered append-only log or risk bad state in the local replicas. As outlined in [9] one solution to this problem can be to introduce epidemic sharing of log entries, with periodical polling of the centralized append-only log when experiencing idle behavior. This will reduce the number of reads from the centralized log while still keeping it as a source of truth. In this chapter, two different gossip protocols are presented that is motivated by this proposal.

The join procedure explained in Chapter 3 requires three phases and knowledge of every other peer in the cluster, every peer is also notified of every stage of the join procedure. With epidemic membership protocols an expensive procedure is not needed, information about the new node will spread in a decentralized way to no more nodes than is needed. The HyParView protocol introduced in Chapter 4 is suited for such a task. This protocol is also suited for failure detection, as a peer that is not reachable will be quickly detected by the other peers in its active view, given that events are shared. When combining such a membership protocol with a gossip protocol for information dissemination ( like those presented in Section 6.1 ) one can give a probabilistic guarantee of information spread with each node being able to make decisions in an autonomous way, reducing local replica size and coordination.

## 6.1 Epidemics Techniques for Log Sharing in Onyx

### 6.1.1 Counter/Feedback Gossip Protocol

This algorithm is based on a push strategy centered around a *broadcast queue* with a counter/feedback implementation. The broadcast queue will be maintained in local state and will contain a vector of maps. The maps will have a key that corresponds to an unique identifier for an event, and the value belonging to that key will correspond to the counter. The algorithm will be divided into two distinct procedures, one *broadcast* procedure and one *receive* procedure. The broadcast procedures single responsibility is to broadcast events that are in the queue, while this queue is not empty. The *getRandomPeer* method utilizes the information that the local replica has about the cluster state and pulls a random peer from that pool. The *startTransmission* procedure will open up a connection through the messaging layer [25] and attempt to emit the event. In the case of more events in the queue the Broadcast method will repeat transmission. The Broadcast method is fed events from the *getEvent* method, and this method pulls events from the broadcast queue. The ordering of the broadcast queue can be implemented in different ways; one might want to serve events in a sequential order, only sending the *earliest* events until the counter threshold is reached for that event. The dissemination of earlier events will encourage straggling peers to get up to speed faster.

The receive procedures responsibility is to receive events and maintain the broadcast queue through feedback. In the case of receiving an event, the procedure will check the broadcast queue and see if it already has that event. If the broadcast queue contains the event, feedback is emitted to the peer that sent the event. If the event is not received beforehand, the event is added to the broadcast queue with counter equal to zero. The method that adds the event to the broadcast queue must also communicate with the process that handles the application of log entries. If the event received is the next event required for local replica transformation it can be applied right away. On the other hand, if the event received is ahead of the log with respect to the local replica state this event might be saved for later use. In the case of receiving feedback, the replica will increment the counter for that specific event, if this results in the counter going over the threshold, the event will be removed from the broadcast queue.

**Listing 6.1:** Pseudocode: Counter/Feedback implementation for a peer.

```
Broadcast():
    if any events in broadcast queue:
        peer = getRandomPeer()
        event = getEvent()
        startTransmission(peer, event)

Receive(message, peer):
    case message is feedback:
        createOrUpdateCounter(eventID)
        if counter for eventID > counter_treshold:
            remove event with eventID from brodcast queue.
    case message event:
        if not received before:
                add event to broadcast queue with counter=0
        if received before:
                feedback(peer, eventId)
```

Note that the messaging layer implements lock-free transmission and as such the broadcast method can fire off several transmissions without waiting for a reply. This type of flooding introduces both some advantages and disadvantages. The advantage is that event information will rapidly disseminate in the cluster as each peer receiving an event will continue spreading it eagerly until the counter threshold is reached. However, this also implies high traffic in the case of a newly discovered event. Especially in cases where a majority of peers discover a new event at around the same time with the periodical polling of the log from the ZooKeeper ensemble. By setting some sort of random offset in the timeout of the polling from event log operation, one can reduce the possibilities of this happening. Traffic can also be reduced by implementing the *getRandomPeer* method in such a way that duplicate messages for events are not sent to the same peer, in other words removing a peer for possible transmission with respect to a single event. Such an implementation could also cause peers that are temporarily disconnected to be isolated with regards to a specific event. As long as one has a source of truth like the ZooKeeper ensemble, this might not be a big problem.

### 6.1.2 Blind Counter Variation

Another way to reduce traffic is to implement a blind variation of this algorithm, a variation that does not rely on feedback. This variation could be accomplished by introducing a *time to live* (TTL) variable for each gossip cycle, a procedure much like the HyParView protocol uses. When a peer encounters an event that it has not seen before it associates that event with a TTL variable and transmits it to a random set of peers taken from the local replica. Each peer that receives an event through the gossip protocol will decrement the TTL and send it off to a random set of peers as well. The storage and application of events to local replica state is done by the *addEventToLocalState* method.

**Listing 6.2:** Pseudocode: Counter/Blind implementation for a peer.

```
Broadcast(event, TTL):
    peerList = getRandomPeerList()
    event = getEvent()
    for each peer in peerList:
        Send(peer, event, TTL)

Receive(event, TTL):
    if not received before:
        addEventToLocalState()
    if TTL > 0:
        broadcast(event, TTL−1)
```

The advantages of this method include that maintenance of a queue is no longer required, one simply sends off each event received with a time to live greater than zero.

Both of these methods assume that information of every peer in the cluster is in the local replica state. This is a challenge when one is faced with large clusters containing thousands of nodes, but reducing that information and still being able to guarantee failure detection, reliability and total order of operations is a considerable challenge. The next section hopes to discuss some concepts that might be able to overcome these obstacles.

## 6.2 Combining Blind Counter with Membership Protocols in Onyx with Aeron

Given a messaging platform that is based on the publication/subscription model, point-to-point communication is done through channels, and streams within those channels. Given the qualitative measures related to epidemic membership protocols given in Section 5.2, there are a number of points one should look at:

- Connectivity
- Degree Distribution
- Average path length
- Clustering Coefficient
- Accuracy

These qualitative measures together with the design of the messaging platform has to be taken into account when designing an epidemic algorithm for information dissemination in a publication/subscription model. The messaging platform, Aeron, also presents additional constraints, namely that it is not designed for an exceedingly large number of streams, so the number of streams should be limited. This implies that point-to-point communication between each peer is unreasonable if scalability is to be achieved, the number of streams should therefore be constrained by a function that grows logarithmically on the number of total peers in the cluster. This simplifies the qualitative analysis of the algorithm, as given by the constraint on number of streams, average path length will be kept low and degree distribution, as well as clustering coefficient, will be kept high. Redundant messages can be kept low by implementing TTL on stream crossing, and keeping track of which streams the message has traveled from, and let each peer be autonomous in deciding whether to relay the message. A high degree of connectivity can be achieved by not letting the peer subscribe and publish to the same stream. Given these constraints one can implement a membership protocol that is simple and based on a stream pool generated by the number of peers, and can be done by each peer independently:

**Listing 6.3:** Pseudocode: Stream pool implementation functional style.

```
Pick-streams (peer-count):
        number-of-streams = (floor (ln peer-count))
        //create stream-pool based on number of streams
        stream-pool = (range number-of-streams)

        publisher-stream = random pick from stream-pool.
        stream-pool = (remove publisher-stream from stream-pool)
        subscriber-stream = random pick from stream-pool.

        return (publisher-stream, subscriber-stream)
```

the code above will generate a stream pool based on number of known peers in the cluster, and pick one stream for publishing and one for subscribing. An analysis of probabilities has to be done given that one wants to be ensured a high degree of connectivity, but given the low number of streams and comparatively large number of peers, the probability of connectivity is expected to be high. This will be further explored in Chapter 8.

Combining a membership protocol such as this with an implementation of the blind counter algorithm seems like a good strategy that combines the simplicity of epidemic protocols with rapid information dissemination. In listing 6.4 such an implementation is presented. This implementation utilizes the *pick-streams* function presented in listing 6.3. It takes into account the publisher/subscriber model of Aeron and uses a TTL variable for the blind counter algorithm. The initiation phase picks two streams, and then sets the publisher stream and subscriber stream. It initializes the subscriber to listen to the stream through a new thread. On receiving a new message, it calls the receive event and if TTL is larger than zero, broadcasts the event to the publisher stream and channel provided by Aeron.

**Listing 6.4:** Pseudocode: Blind counter variation with membership operations through a stream-pool.

```
Init():
    streams = (pick-streams peer-count)
    publisher-stream = streams [0]
    subscriber-stream = streams [1]
    Start new thread:
        ListenOnStream(subscriber-stream)

ListenOnStream(subscriber-stream):
        connectChannel (subscriber-stream)
    listen-loop:
        if received:
                event = read-buffer()
            Receive (event)

Receive(event):
    if not received before:
        addEventToLocalState()
    if TTL > 0:
        event.TTL -= 1
        Broadcast(event)

Broadcast(event):
     SendToChannel(publisher-stream, event)
```

## 6.3 Conclusion

The introduction of gossip protocols to share log events through the cluster seems like a probable solution to excessive ZooKeeper load by reducing the number of reads from the centralized log store. The simplicity of the feedback/blind counter variations makes it viable for implementation and testing on the platform. The *source of truth* provided by the coordination layer in ZooKeeper makes it possible to avoid the complexity of maintaining agreement on the order of events and allows the gossip protocol to optimistically share events. If one is to allow the gossip protocol to write to the log as well, some sort of protocol that provides agreement on event ordering is needed. This introduces complexity at each peer but allows one to completely remove the centralized coordination layer. For this thesis however, total order maintenance through epidemic algorithms is out of scope.

The next chapter will introduce an implementation of a blind counter variant, based on the pseudocode presented in listing 6.4. Together with the membership protocol presented in section 6.2, the blind counter algorithm will provide the basis for implementation and analysis.

# 7 Implementation

## 7.1 Introduction

In this chapter the focus will be on the attempted implementation of the blind counter algorithm described in Chapter 6. In order to justify design decisions, the chapter will start with an exploration of the original code base, namely the two most relevant parts; the peer group manager loop and the communicator. The responsibilities and functions of these components will be explored, and as such provide a context that makes the next part of the chapter easier to understand.

The next part of the chapter will deal with the attempted implementation of epidemic dissemination in Onyx. This implementation can be broadly divided into two parts; the epidemic messenger and the communication layer. The epidemic messenger responsibilities lie in handling the messages according to the blind counter algorithm, interacting with the underlying messaging framework, and creating an epidemic membership protocol. The communication layer responsibilities is receiving log events from the epidemic messenger and actually applying them to the local state. The algorithm devised for this purpose will also be explained.

## 7.2 Overview of original implementation

We will discuss the parts of the code that has to do with dissemination of log events and applying those log events. In that regard there are some components that are especially pertinent; namely the communicator and peer group manager in the peer namespace, as well as the zookeeper component in the log namespace.

The communicator component is the layer that handles reading and writing log-events from and to the log. The zookeeper component handles the actual interaction with the zookeeper node. The exploration of the original code base will be looked at through the read and write interactions with the log. First, however, the most important components and their functionality will be covered.

### 7.2.1 Peer Group Manager Loop

The peer group manager is initialized as a thread in the peer name space. Its main responsibility is to receive log events, through an asynchronous channel provided by the core.async [16] framework. This channel is the equivalent of the inbox described in Chapter 3. In the peer-group managers thread, a loop is continuously executing and listening for messages on the inbox-ch. In the case of a received message, the message will be dispatched through a set of actions. These actions include starting the peer group, stopping the peer group and also applying log events to the local replica. The local state is recorded in an immutable map, which is continuously passed around to the actions given by the received log entry and thus transformed. The peer group manager loop also handles exceptions given to it by the inbox channel and in the case of an exception it restarts the peer. In the case of prolonged silence (given by a timeout) on the channels the loop listens to, it will also send a heartbeat to make sure that it is still in reach of other peers. In the initiation phase of the peer group manager loop, the communicator will also be initialized, which will be covered next.

### 7.2.2 The Communicator

The communicators responsibilities include initializing the log, provide means of which to read the log, and also to write to the log. The initializing of the log is done through a function call to the log name space, this function call sets up the required connections with the required paths of the zookeeper server, and initializes the origin of the log. This is necessary for the client to be able to connect, read and write to the log. In order to be able to write to the log, the communicator creates

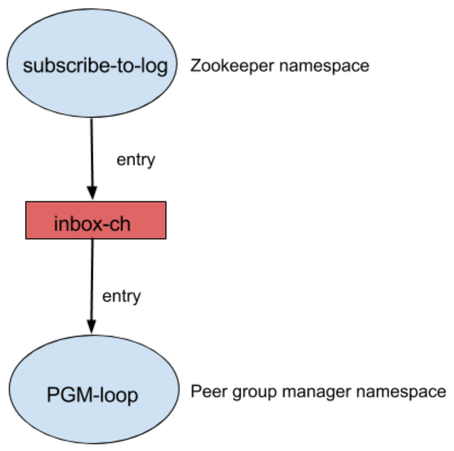

**Fig. 7.1:** Representation of the original read process in Onyx.

a component called the log writer. This component initializes an asynchronous channel that represents the outbox as explained in Chapter 3, this outbox channel is given to the peer group manager as a way to signalize when an event should be written to the log. The log writer initializes a thread that continuously listens for messages on the outbox channel, and if a message is received, the log writer will append an entry to the zookeeper log.

The component that is responsible for reading of the log is called the replica subscription. It is the replica subscription which creates the inbox channel discussed in the previous section, and passes it on to the peer group manager. The replica subscription component also subscribes to the log through a function call into the log name space, this function call creates another thread that continuously polls to see if any addition has been made to the log. If an addition has been made, this function call reads from the log and puts the new entry onto the inbox channel, which is then received in the peer group manager and handled by a dispatch.

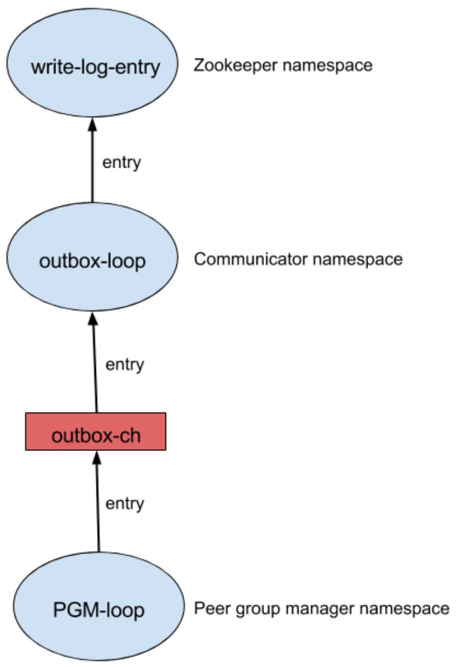

**Fig. 7.2:** Representation of the original write process in Onyx.

## 7.3 Overview of new implementation

The purpose of the implementation is to replace reads to the zookeeper log by peer-to-peer communication with the help of an epidemic algorithm. As the epidemic algorithm is unable to satisfy distributed total order of the log events, writes to the log will still be required in order to timestamp and order the log-events. This requires implementing logic to handle information dissemination between peers as well as the logic required for ordering the events, prior to application of the events to the local state. The proposed implementation divides this work into two main components:

- **The Epidemic Messenger** - Handles the messaging work and interaction with the underlying messaging framework, Aeron.

- **Communication layer** - This layer is used to communicate with the log as well as receiving updates from the underlying epidemic messenger. Divided into two parts, the log writer and replica subscription.

The remainder of the this chapter will concern itself with explaining these parts, and the logic used to accomplish the set goals. It will start with the underlying layer of the epidemic messenger and the mechanisms required for both inter-communication and intra-communication. Then it will explain the implementation of the communication layer, and how log event application is accomplished. The last part will go into how the implementation was tested.

### 7.3.1 The Epidemic Messenger

The epidemic messenger is the main component responsible for handling the messaging part of the application. This can be divided into two parts; interaction with the messaging library, Aeron, as well as relaying log events to the communication layer.

As explained in Chapter 4, Aeron follows a publisher/subscriber model. This model requires you to instantiate the aeron media driver, a subscriber, and a publisher. In this regard, the epidemic messengers responsibility is to provide the instantiation of the subscriber and publisher with the necessary resources, which amounts to the Aeron channel to be used, the stream to use within that channel, as well as a context provided by the driver.

The publisher component is the most straightforward one, as its only responsibility is to relay a message given to it by the messenger. This is done through the offer method provided by the Aeron library. The offer method takes a buffer containing the message and its length and puts that buffer on the channel, and lets the Aeron library do the work correlated with actually getting that message to the intended destination.

The subscriber component is slightly more complicated. As it is reactive, it needs to continuously listen on the provided stream, this requires the listener to be instantiated in its own thread, as to not block the whole application. This is accomplished by creating a java Consumer [1] object, that on arrival of messages calls a method that enables the subscriber to process the message. The processing of the message consists of reading the received buffer and calling a method on the epidemic messenger that enables the messenger to process the message further. The provided figure contains an overview of how messages move between the messenger, subscriber and publisher:

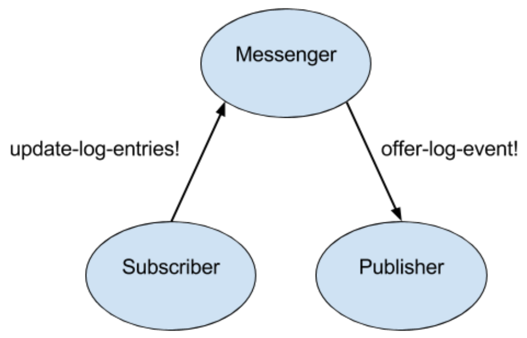

Fig. 7.3: Representation of the original write process in Onyx.

Now that the message movement has been explained, one needs to look at how the messenger processes the messages received. Besides handling message dissemination, the messenger also decides which messages to relay to the communication layer. The communication layer handles the ordering of the log events, so all the messenger has to do is to keep track of the messages already received, and only pass on those events that have not already been relayed.

The messenger receives messages in two ways; through the communication layer, these are messages that are already applied and needs to be disseminated to the other peers, and through the subscriber, these are messages that need to be applied, and conditionally relayed to other peers. The messages received through the communication layer are passed on by the log writer. These messages are already applied, but they still need to be added to the state and ordering of log events in the replica subscription. The messages received through the subscriber of the epidemic messenger, need to be handled according to the blind counter algorithm. If the log event has not been received before, the message is written to an asynchronous channel which is picked up by the replica subscription in the communication layer, which will be the next subject.

### 7.3.2 The Communication Layer

**Receving Messages**

As explained in section 7.2.2, the communication layer contains a component called the replica subscription. The replica subscription in the original code base would listen to the log and read any appendages. This is where it was decided to implement the logic for handling log events received from the epidemic messenger. In order to ensure operability of the peers it was decided to combine the epidemic messages with reads from the zookeeper log, this is done in case of messaging failure. Reading of the log is only done through a timeout, if an epidemic message is not received within some predefined timespan, a read to the centralized log will be attempted.

This mixture of log messages and epidemic messages need to be ordered so that log events are applied in the right order. The epidemic implementation has no guarantees of ordering, in fact, one might expect to receive log events that are ahead or behind the local state of the node. However, one can safely assume that once a log event has been applied to the local state, it can be discarded. These sorts of requirements fit well to a prioritized queue data structure. The prioritization will be done on the timestamp written to the log event by zookeeper, where earlier log events are prioritized.

As the replica subscription component is required to listen to an asynchronous channel for epidemic log events, a thread is created that contains a loop that continuously listens to the channel. This loop also has two asynchronous channels that are communicating with the log, one for initiating a read, and one for actually receiving the read value from the log. The loop needs to keep track of current position, namely the ordering of the last log event that is applied, in order to know whether to insert a received log event in the prioritized queue.

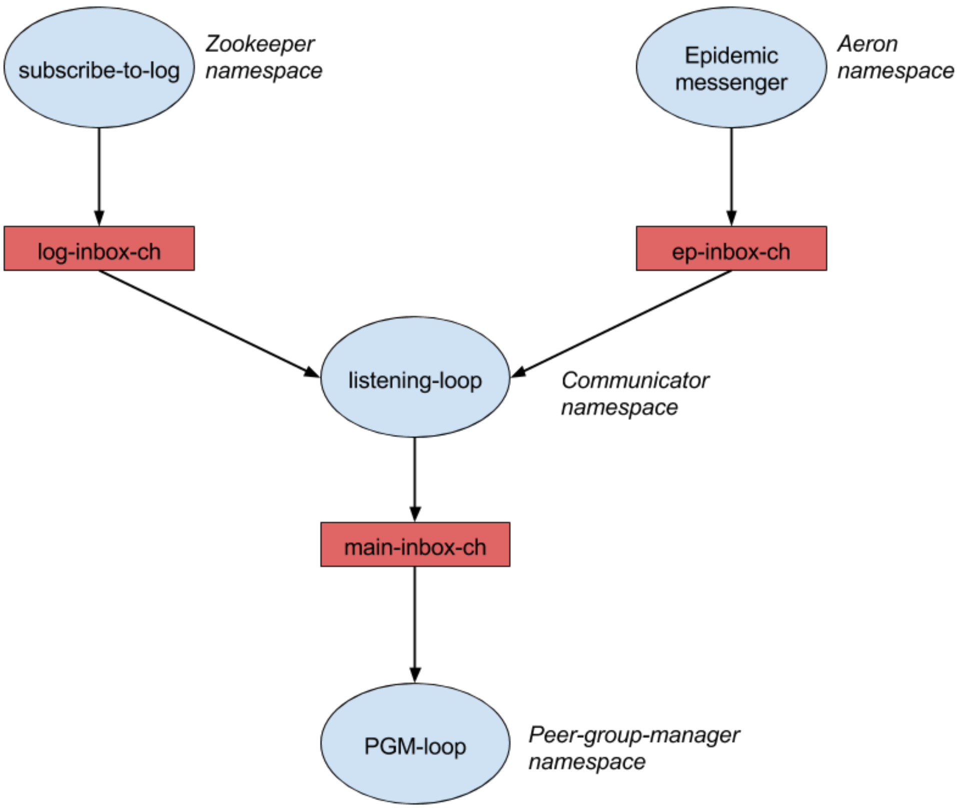

**Fig. 7.4:** Representation of the new read process in Onyx.

### Creation of Log Messages

The creation of log messages is less complicated. The peer group manager explained in section 7.2.1, does the work of deciding when to create a new log entry. In the original implementation this log entry is then written to the log by the log writer. This communication happens through asynchronous channels. The decision was made to implement the dispatch of epidemic log events at this location as well: after the entry has been written to the zookeeper log, it is returned with some log information, in that log information is the ordering given to the log event. This information, together with the log event, is then put through another asynchronous channel, going back to the peer group manager and dispatched to an action that calls a function in the epidemic messenger, triggering the message to be disseminated in an epidemic fashion.

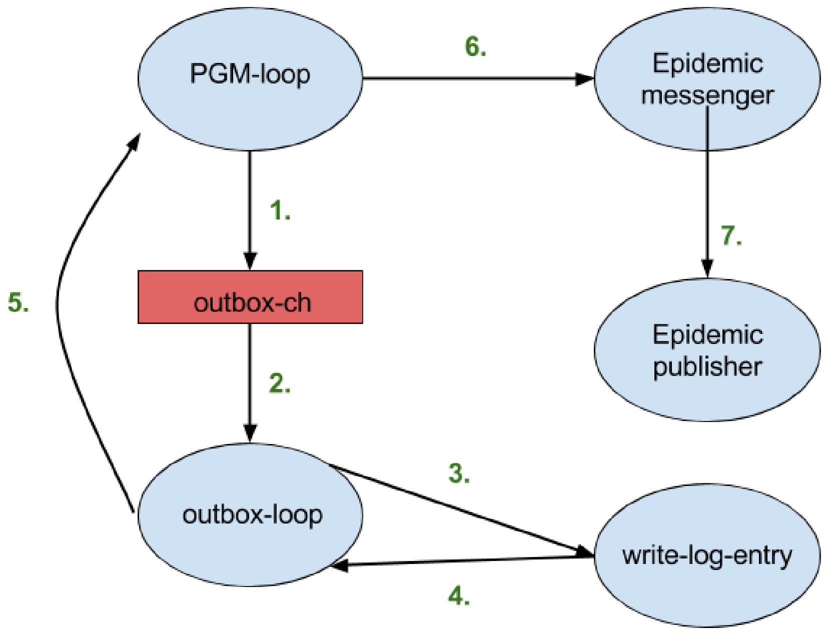

Fig. 7.5: Representation of the new write process in Onyx.

## 7.4 Summary

This chapter started by looking at the original implementation, explaining the relevant parts that had to with writing and reading to the ZooKeeper log. The interplay between these components, with the peer group manager loop as the receiver and applier of log events, with the communicator acting as a intermediary. The changes made to these components in order to realize the goal of epidemic log sharing were also explained, The addition of an epidemic messenger to the messaging namespace, which responsibilities is to interact with the underlying messaging framework, Aeron, and implement the blind counter algorithm as well as the epidemic membership protocol. The consequent changes to the communicator layer were also explained, with the need to throttle read access to the log, as well as decide which log events to relay to the peer group manager.

The next chapter will concern itself with the analysis and testing of these components.

# 8 Results

## 8.1 Introduction

In this chapter the focus will be on the results of the implementation presented in Chapter. 7. Because of time constraints the whole implementation is not ready for testing, the parts discussed in this chapter will be the part of the implementation that is covered in 7.3.1. Broadly speaking the dissemination of messages consists of two parts; the stream pool generator and the epidemic messenger. The characteristics of the stream pool generator function were discussed in Chapter 6.2, and a combination of that chapter together with the theoretical groundwork laid in Chapter. 5 will be used to define the metrics and subsequent discussion of those results.

After covering the analysis of the stream pool generator function, the focus will shift towards actual message dissemination through local instances of the messenger. Here the metrics are defined as coverage, or how many peers actually receive the message, and the fraction of redundant messages. These metrics will help us decide what TTL would be suitable for the current implemented algorithm.

## 8.2 Setup

The tests were performed locally. Time constraints and issues with the implementation made it unfeasible to do a complete test on a large cluster. The tests were done by pulling in the relevant parts of the onyx project into the test project and then utilizing test functions on those parts. A graph library called Loom [13] were used for creating graphs of the collected data, and those graphs were then subsequently used to carry out the computations. The first component tested were the stream pool generation, which set the precedent for the membership protocol. Separating the stream pool generation from the rest of the testing made it possible to do thorough and iterative tests on that component in isolation, doing thousands of iterations to get a good handle on how robust the algorithm is, and whether the predictions made in Chapter 6 seem correct. This would not be possible to do locally if it were to be tested together with the other parts of the application.

The second component tested were the epidemic messenger as a whole, how message dissemination works. This component is constrained substantially by doing it locally, as you have to create an Aeron media driver, and for each peer a subscriber and publisher. On the machine used, this limited the testing to 55 peers. The testing worked by starting a media driver and then starting up the epidemic messenger; initiating message dissemination from one of the peers.

The third component that should have been tested were the application of log events. Time constraints made this impossible to do, as the implementation is not ready.

## 8.3 Test Cases

### 8.3.1 Membership protocols

The membership protocol pertains to testing the function that generates the stream pool. This function is based on a random factor as explained in Chapter 6.2 and Chapter 7. Test cases that take this random factor into account should then be concerned about averages. Three qualitative measures have been taken into account, the degree distribution, shortest average path length and the connectivity. For each of these averages, a variable number of peers may be tested. This thesis chose to concern itself with the numbers that expose weaknesses in the algorithm, namely the exact number of peers that would increase the stream-pool count by 1. In order to increase the robustness of the results, iterative abilities have also been incorporated into the testing algorithms. This means that each test is executed a given number of times, before taking the average of all those tests.

### Average Degree Distribution

The average degree distribution is calculated by taking a number of generated peers and calculating each peers in-degree and out-degree. These values are then independently averaged over all the peers. This process is repeated a specified number of times, before calculating the total average out-degree and in-degree. In addition, we calculate the variance of the in-degree of each set of peers. This variance signifies how much the in-degree varies, which is a measure of how balanced the cluster is.

### Shortest Average Path Length

The shortest average path length is calculated by the help of a graph library called Loom [13]. First the peers are generated, and then the peer list is transformed into a graph, before utilizing the Loom library to create a directed graph with all weights equal to 1. The streams are utilized as nodes, and a direct connection occurs if one peer has a subscription stream equal to another peers publisher stream.

After creating the directed graph, a shortest path between all pairs algorithm is utilized. This algorithm is also provided by the Loom library. These shortest paths are then averaged. This process is repeated a specified number of times, i.e iterations, and then the total average is calculated.

### Connectivity

The connectivity is calculated according to the definitions of connectivity set in Chapter. 6.2. This implies that if the set of publication streams are not equal to the set of subscription streams, connectivity is not achieved. Connectivity is measured by doing this test on every set of peers that are generated a specified number of times. If one these sets of peers show to have not achieved connectivity, the connectivity variable would end up as false.

### 8.3.2 Message Dissemination

In this section of the testing we will take a look at how the messages spread through the node network. We will look at examples of flow and results pertaining to whether or not all nodes were reached and what percentage of total messages were redundant. The tests were done locally and were accomplished by creating instances of the epidemic messenger component. After the messengers have been initialized, one message was given to one of the messengers. At this point the messengers should take care of the dissemination themselves. After a specified timeout period the analysis will be done.

The analysis is structured so that two metrics are measured. The **dissemination coverage** signifies which fraction of the peers received the message. So a dissemination coverage of 1.0 implies that all nodes received the message. The dissemination coverage is calculated by receiving the stored entries from all peers, and comparing it with the original peer list. The second metric that is measured is the number of redundant messages. This metric is calculated by adding two counters to each messenger, counting total messages received, as well as redundant messages received. A redundant message is defined as a message that already has been received by the peer.

To get a more realistic look at how a cluster of messengers would interact, we repeat the process of collecting metrics on several different sets of peers. These metrics are then averaged. Because this analysis was done locally, the sets of peers are limited to 8, 21 and 55 in number. The metrics are collected 20 times for each of these sets and then averaged. In order to get a grasp of how the time-to-live (TTL) variable of each message impacts the metrics, the TTL is varied from 1 to 4 on each run. The TTL was also limited by the fact that these tests were performed locally, with 55 peers and a TTL of 5, the Aeron media driver seemed to get an overload and a subsequent segmentation fault, suggesting a large number of redundant messages.

## 8.4 Goals

### 8.4.1 Membership protocols

In Chapter 6, the probabilities of connectivity with a certain stream pool algorithm were discussed. The results showed low probability of isolated nodes. Given the probability of a high degree of connectivity some implications can be made. The shortest average path length should turn out quite low, as the shortest path between nodes is directly correlated with how densely connected that graph is. Likewise, the degree distribution, which is the qualitative measurement used for direct connections between peers should be kept relatively high.

### 8.4.2 Message Dissemination

The main goal of the message dissemination is that the dissemination coverage is close to or equal to 1. Given a right TTL, this goal should be accomplished given that connectivity in the network is achieved. With regards to the ratio of redundant messages, one would prefer a low number. This ratio is also dependent on the TTL, if the TTL is to high, one can expect a larger ratio of redundant messages.

## 8.5 Results

### 8.5.1 Membership protocols

The results of the tests done on membership is presented in Table 9.1 to 9.5. We will break the analysis down into its constituent parts.

**Average Degree Distribution**

As per the results shown in Tables 9.1 to 9.5 we can see that the degree distribution with regards to both in-degree and out-degree gets lower the more peers are added to the cluster. This is to be expected because of the stream pool function makes it so that the stream pool size is constrained by a function of $ln(n)$.

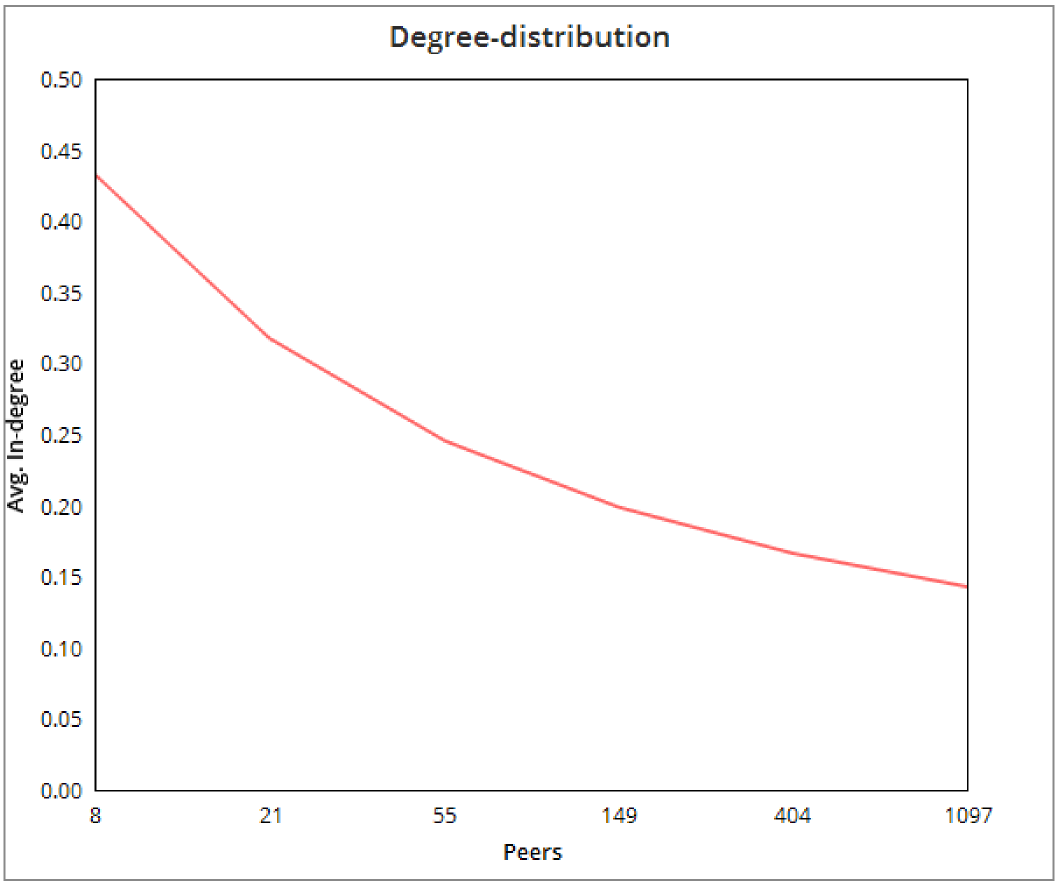

**Fig. 8.1:** Representation of the original read process in Onyx.

The in-degree seen as a whole is quite high. This is good for message dissemination purposes. The variance of the in-degree decreases as the number of peers increases. The variance is not that high to begin with, see Table 9.1, and this suggests that the clusters will be well balanced.

### Average Shortest Path

The average shortest path is a measure of how long time a message will take to reach another node. The average shortest path in the results hover around 1. The implications of this is that average node has a shortest path of 1 to any other node.

This small average shortest path is a good sign, as it means that a message will be using a short amount of time to reach the other nodes.

Tab. 8.1: Avg. degree distribution, avg. shortest path and connectivity for 8 peers, 2 streams averaged over an iteration of 1000 times

| | |
|---|---|
| Avg. In-degree | 0.4322 |
| Avg. Out-degree | 0.4322 |
| Avg. In-degree variance | 0.0226 |
| Avg. shortest path | 1.0 |
| All connected? | Yes |

Tab. 8.2: Avg. degree distribution, avg. shortest path and connectivity for 21 peers, 3 streams averaged over an iteration of 1000 times

| | |
|---|---|
| Avg. In-degree | 0.3172 |
| Avg. Out-degree | 0.3172 |
| Avg. In-degree variance | 0.0102 |
| Avg. shortest path | 1.0381 |
| All connected? | Yes |

Tab. 8.3: Avg. degree distribution, avg. shortest path and connectivity for 55 peers, 4 streams averaged over an iteration of 1000 times

| | |
|---|---|
| Avg. In-degree | 0.2455 |
| Avg. Out-degree | 0.2455 |
| Avg. In-degree variance | 0.0032 |
| Avg. shortest path | 1.0231 |
| All connected? | Yes |

Tab. 8.4: Avg. degree distribution, avg. shortest path and connectivity for 149 peers, 5 streams averaged over an iteration of 1000 times

| | |
|---|---|
| Avg. In-degree | 0.1987 |
| Avg. Out-degree | 0.1987 |
| Avg. In-degree variance | 0.010 |
| Avg. shortest path | 1.0014 |
| All connected? | Yes |

Tab. 8.5: Avg. degree distribution, avg. shortest path and connectivity for 404 peers, 6 streams averaged over an iteration of 500 times

| | |
|---|---|
| Avg. In-degree | 0.1662 |
| Avg. Out-degree | 0.1662 |
| Avg. In-degree variance | 0.0 |
| Avg. shortest path | 1.0 |
| All connected? | Yes |

Tab. 8.6: Avg. degree distribution, avg. shortest path and connectivity for 1097 peers, 7 streams averaged over an iteration of 500 times.

| | |
|---|---|
| Avg. In-degree | 0.1427 |
| Avg. Out-degree | 0.1427 |
| Avg. In-degree variance | 0.0 |
| Avg. shortest path | 1.0 |
| All connected? | yes |

### 8.5.2 Message Dissemination

The results of the analysis is given in Table. 9.7 to Table. 9.9. They show average coverage and avg redundancy ratio for sets of peers with size 8, 21, and 55 as well as for TTL from 1 to 4. Looking at the results we can see some trends. The average dissemination coverage increases with the TTL. This is not surprising, given that increased TTL makes it more probable for the message to cross streams and thus reaching more peers. Based on the results, one might also be able to deduce what an approximate optimal TTL would be based on peer group size and stream pool. A conservative choice, based on the results, would be to set TTL equal to stream pool size. This would guarantee message dissemination, at least in the simulations presented here. However, it would also mean an extremely high degree of redundancy.

The average received redundant messages per peer is very high. For guaranteed message dissemination coverage one must be able to tolerate at least 70 percent redundant messages. This fraction is way to high and is one of the delimiting factors of the current algorithm. One way of reducing this redundancy would be to implement some sort of stream check on the messages before deciding to pass them on. This would require each message to keep a history of visited streams, and a peer would check this set of streams against its own publisher stream, if the publisher stream is present it would imply that the message has already been at the designated stream, and message dissemination to that stream is canceled. In theory, this would greatly reduce the redundant messages.

We can also see some results that were not expected. With a set of 8 peers and 2 streams, and a TTL of 4, the average coverage does not equal 1, see Table 8.7, meaning that in some cases not all peers were reached. Given the probability of connectivity explored in Section. 8.5.1, we should expect the network to be connected, and given a large enough TTL this should result in total dissemination coverage. No concrete explanation have been found for this discrepancy, one possible reason is that in this exact case, some subscriber have been lagging and thus not been able to join the ensemble fast enough. Another possible reason is that as we are dealing with probabilities this case may just have been an isolated node. This exact analysis has been rerun and the same discrepancy have not repeated itself. This also highlights the fact that this analysis done on a real cluster would be pertinent.

Tab. 8.7: Avg.coverage and average redundancy for 8 peers and 2 streams with varying TTL.

| **TTL** | **avg-coverage** | **avg-redundant** |
|---|---|---|
| **1** | 0.6125 | 0.5 |
| **2** | 1.0 | 0.7036 |
| **3** | 1.0 | 0.8582 |
| **4** | 0.9562 | 0.8897 |

Tab. 8.8: Avg.coverage and average redundancy for 21 peers and 3 streams with varying TTL.

| TTL | avg-coverage | avg-redundant |
|---|---|---|
| 1 | 0.3642 | 0.5 |
| 2 | 0.9571 | 0.7433 |
| 3 | 1.0 | 0.9437 |
| 4 | 1.0 | 0.9870 |

Tab. 8.9: Avg.coverage and average redundancy for 55 peers and 4 streams with varying TTL.

| TTL | avg-coverage | avg-redundant |
|---|---|---|
| 1 | 0.2572 | 0.5 |
| 2 | 1.0 | 0.7797 |
| 3 | 1.0 | 0.9788 |
| 4 | 1.0 | 0.9988 |

## 8.6 Summary

In this chapter the focus has been on the implementation of analysis, as well as a discussion of the results of that analysis. In the case of membership protocols, three main metrics were analyzed; the average degree distribution, shortest average path length and connectivity. Based on theoretical knowledge from Chapter 5 and Chapter 6 these metrics were measured over an average. Because of the separation between the generation of the stream pool and the actual messenger, one was able to do thousands of iterations to create a substantial basis for the averages. The results proved to correlate with the predictions made, showing a high degree of connectivity, average degree distribution and a low average shortest path.

However, what was not predicted were the implications this would have on the next part of the analysis. The high degree of connectivity and a low number of common streams generated a high degree of redundant messages. A possible solution to this problem was presented, but time constraints make implementation and analysis of this solution impossible.

# 9 Conclusions

The main goal of this thesis was to implement functionality in order for the Onyx platform to reduce its load on the centralized log. Given the potential benefits of an epidemic protocol, and its inherent simplicity the author believed that message dissemination through epidemic means was a viable pathway for the attainment of that goal. The realization of this goal did not happen, due to time constraints and surprising complexity in the implementation of the chosen algorithms, the application of log events was never successful. The interaction between the framework and the centralized log was more complex than first believed, and still poses a problem to the implementation of applying log events through epidemic means. In addition, the implementation of more rudimentary parts, like the epidemic subscribers and publisher took longer time than expected. These delays was mostly due to ignorance of the author, with regards to an understanding of the Onyx framework as a whole as well as complexities involved with working with the underlying message layer, Aeron. However, some intermediary stepping-stones were accomplished. An epidemic messenger was implemented that functioned reasonably well. The implementation of that messenger followed a blind counter variation for message dissemination as well as implementing a function that generated membership for peers autonomously. The only real problem with the results of the protocol were the high number of redundant messages, and these redundant messages are believed to be avoidable by changing the message dissemination algorithm to take into account the pathway the messages have taken before. With the implementation of such a change to the dissemination algorithm, the epidemic messenger is deemed to be a success.

# Bibliography

[1]Norman T. J. Bailey. *The Mathematical Theory of Infectious Diseases and its Applications (second edition)*. 1975 (cit. on p. 29).

[2]Brenda Baker and Robert Shostak. „Gossip and Telephones*“. In: (1972) (cit. on p. 29).

[3]Sondre Basma. „Epidemic Algorithms for Log Sharing with Onyx“. In: (2016) (cit. on p. 2).

[4]David Bermbach and Stefan Tai. „Eventual Consistency: How soon is eventual?“ In: (2011) (cit. on p. 29).

[5]David Bermbach and Stefan Tai. „Lightweight Probabilistic Broadcast“. In: (2003) (cit. on pp. 31, 32).

[6]David Bermbach and Stefan Tai. „SCAMP: Peer-to-peer lightweight membership service for large-scale group communication“. In: (2001) (cit. on pp. 31, 32).

[7]Alan Demers and Dan Greene, Carl Hauser, Wes Irish, et al. „Epidemic algorithms for replicated database maintenance“. In: (1987) (cit. on pp. 29, 30).

[17]Patrick Hunt, Mahadev Konar, Flavio Junqueira, and Benjamin Reed. „ZooKeeper: Wait free coordination for Internet-scale systems“. In: (2012) (cit. on pp. 6, 8, 15).

[18]H. V. Jagadish and Johannes Gehrke and Alexandros Labrinidis, Yannis Papakonstantinou and Jignesh M. Patel, Raghu Ramakrishnan, and Cyrus Shahabi. „Big data and its technical challenges“. In: (2014) (cit. on p. 3).

[19]Flavio Junqueira, Benjamin Reed, and Marco Serafini. „Zab: High-performance broadcast for primary-backup systems“. In: (2011) (cit. on p. 5).

[21]Leslie Lamport. „Paxos Made Simple“. In: (2001) (cit. on p. 8).

[22]Joao Leitao, Jose Pereira, and Luıs Rodrigues. „HyParView: a membership protocol for reliable gossip-based broadcast“. In: (2007) (cit. on pp. 31, 32).

[23]Barbara Liskov and James Cowling. „Viewstamped Replication Revisited“. In: (2012) (cit. on p. 8).

[24]James Manyika, Michael Chui, Brad Brown, et al. „Big data: The next frontier for innovation, competition, and productivity“. In: (2011) (cit. on p. 3).

[26]Ben Moseley and Peter Marks. „Out of the Tar Pit“. In: (2006) (cit. on p. 11).

[30]Ankit Toshniwal. „Storm @Twitter“. In: (2014) (cit. on pp. 1, 3, 11, 14).

[31]Matei Zahari, Mosharaf Chowdhury, Tathagata Das, et al. „Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing“. In: (2012) (cit. on p. 11).

## Websites

[8]Michael Drogalis. *Clojure West Conference - Inside Onyx. Visited on 04.12.2016*. 2016. URL: `https://www.youtube.com/watch?v=KVByn_kp2fQ&t` (cit. on p. 15).

[9]Michael Drogalis. *Coordination Scalabilites. Visited on 08.12.2016*. 2015. URL: `https://github.com/onyx-platform/onyx/blob/0.9.x/doc/design/proposals/coordination_scalability.adoc` (cit. on pp. 22, 35).

[10]Michael Drogalis. *Lambda Jam 2015 - Michael Drogalis - Beginning Onyx - Functional, Distributed Computation in Clojure. Visited on 06.12.2016*. 2015. URL: `https://www.youtube.com/watch?v=6zlPmtPm7ig&t` (cit. on p. 15).

[11]Michael Drogalis. *Onyx: Distributed Computing for Clojure. Visited on 02.12.2016*. 2015. URL: `https://www.youtube.com/watch?v=YlfA8hFs2HY` (cit. on p. 11).

[12]Michael Drogalis. *Onyx User Guide 0.9x. Visited on 02.12.2016*. 2016. URL: `http://www.onyxplatform.org/docs/user-guide/0.9.x/` (cit. on pp. 14, 15, 21).

[13]Aysylu Greenberg and Justin Kramer. *Loom. Visited on 15.08.2017*. 2017. URL: `https://github.com/aysylu/loom` (cit. on pp. 55, 57).

[14]Haadop. *Haadop Ecosystem Table. Visited on 14.12.2016*. 2016. URL: `https://hadoopecosystemtable.github.io/` (cit. on p. 3).

[15]Rich Hickey. *Clojure Programming Language Homepage. Visited on 04.12.2016*. 2016. URL: `https://clojure.org/` (cit. on p. 11).

[16]Rich Hickey. *clojure.core.async. Visited on 31.08.2017*. 2017. URL: `https://clojure.github.io/core.async/` (cit. on p. 46).

[20]Kafka. *Kafka. Visited on 15.08.2017*. 2017. URL: `https://kafka.apache.org/` (cit. on p. 25).

[25]Todd Montgomery, Richard Warburton, and Martin Thompson. *Coordination Scalabilites. Visited on 09.12.2016*. 2016. URL: `https://github.com/real-logic/Aeron/wiki` (cit. on pp. 16, 36).

[27]Apache Org. *Apache Spark. Visited on 02.12.2016*. 2016. URL: `http://spark.apache.org/` (cit. on pp. 1, 3, 11).

[28]Martin Thompson. *Aeron Best Practices. Visited on 15.08.2017*. 2017. URL: `https://github.com/real-logic/Aeron/wiki/Best-Practices-Guide` (cit. on p. 25).

[29]Martin Thompson. *Aeron: Open-source high-performance messaging by Martin Thompson. Visited on 15.08.2017*. 2014. URL: `https://www.youtube.com/watch?v=tM4YskS94b0` (cit. on p. 25).

[32]*ZooKeeper Documentation. Visited on 02.12.2016*. 2016. URL: `https://zookeeper.apache.org/doc/current/api/org/apache/zookeeper/ZooKeeper.html` (cit. on p. 7).

# List of Figures

# List of Tables

## Colophon

This thesis was typeset with $\LaTeX\,2_\varepsilon$. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.