Zuse Institut Berlin
Department Distributed Algorithms
and Supercomputing

Institute of Computer Science

Bachelor thesis

# A Gossiping Framework for Scalaris

Author: Jens V. Fischer
Matriculation No.: 3923671
uni@jens-fischer.eu

Primary Examiner: Prof. Dr. Katinka Wolter
Secondary Examiner: Dr. Florian Schintke

Berlin, 28th February 2014

**Abstract**

Gossiping is a technique for solving communication and computation tasks in peer-to-peer systems. It refers to the periodic, pairwise, message exchange between random peers in a network. Gossiping provides probabilistic guarantees on convergence speed and accuracy and features very good scalability, robustness and uniform load distribution. The two most important classes of gossiping algorithms are gossip based information dissemination and gossip based aggregation.

Scalaris is a distributed key-value store with guarantees of atomicity, concurrency, isolation and durability on transactions (ACID properties). It is completely based on peer-to-peer techniques and uses gossiping for several purposes.

The contribution of this work is is twofold: First, to conceptualise and implement a gossiping framework for Scalaris. An appropriate abstraction of different gossiping algorithms is established and used for the implementation. Second, the gossip based aggregation of load information in Scalaris is to be improved. The symmetric push-sum protocol is selected as aggregation algorithm and extended to allow histogram computation and periodic restarting. It is then implemented on the basis of the framework.

*Jens V. Fischer*

# Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

28th February 2014

Jens Fischer

## Zusammenfassung

Gossiping ist eine Technik zum Lösen von Kommunikations- und Berechnungsaufgaben in Peer-to-Peer Systemen. Es lässt sich als periodischer, paarweiser Nachrichtenaustausch zwischen zufällig gewählten Nachbarknoten verstehen. Gossiping gibt probabilistische Garantien bezüglich Konvergenzgeschwindigkeit und Genauigkeit und zeichnet sich durch Skalierbarkeit, Robustheit und gleichmäßige Lastverteilung aus. Informationsverbreitung und Aggregation durch Gossiping sind die zwei wichtigsten Arten von Gossiping Algorithmen.

Scalaris ist eine verteilte Key-Value-Datenbank welche Atomarität, Konsistenz, Isolation und Dauerhaftigkeit (ACID Eigenschaften) für Transaktionen garantiert. Scalaris basiert vollständig auf Peer-to-Peer Techniken und nutzt Gossiping zu verschiedenen Zwecken.

Der Beitrag dieser Arbeit ist ein zweifacher: Erstens, der Entwurf und die Implementation eines Gossiping Frameworks für Scalaris. Eine angemessene Abstraktion verschiedener Gossiping Algorithmen wird entworfen und als Basis für die Implementation genutzt. Zweitens wird das Monitoring von Lastinformation verbessert. Das "Symmetric Push-Sum Protocol wird als Algorithmus für die Aggregation ausgewählt und erweitert, um die Berechnung von Histogrammen und periodisches Neustarten zu unterstützen. Das Load Monitoring wird auf Basis des Frameworks implementiert.

## Abstract

Gossiping is a technique for solving communication and computation tasks in peer-to-peer systems. It refers to the periodic, pairwise, message exchange between random peers in a network. Gossiping provides probabilistic guarantees on convergence speed and accuracy and features very good scalability, robustness and uniform load distribution. The two most important classes of gossiping algorithms are gossip based information dissemination and gossip based aggregation.

Scalaris is a distributed key-value store with guarantees of atomicity, concurrency, isolation and durability on transactions (ACID properties). It is completely based on peer-to-peer techniques and uses gossiping for several purposes.

The contribution of this work is is twofold: First, to conceptualise and implement a gossiping framework for Scalaris. An appropriate abstraction of different gossiping algorithms is established and used for the implementation. Second, the gossip based aggregation of load information in Scalaris is to be improved. The symmetric push-sum protocol is selected as aggregation algorithm and extended to allow histogram computation and periodic restarting. It is then implemented on the basis of the framework.

# Contents

**List of Figures** **viii**

**List of Tables** **viii**

**List of Algorithms** **viii**

**List of Code Fragments** **viii**

**1 Introduction** **1**

**2 Fundamentals of Gossiping** **4**
2.1 Gossip Based Information Dissemination . . . . . 5
2.1.1 Basic Dissemination Algorithm . . . . . 5
2.1.2 Related Work . . . . . 7
2.2 Membership Management . . . . . 8
2.2.1 Example Membership Protocol: Cyclon . . . . . 9
2.2.2 Related Work . . . . . 10
2.3 Cycles . . . . . 11
2.4 Robustness in Practical Settings . . . . . 11

**3 Gossip Based Aggregation** **13**
3.1 Related Work . . . . . 13
3.2 Computing Averages . . . . . 15
3.3 Histograms . . . . . 19
3.4 Evaluation of symmetric push-sum protocol . . . . . 20
3.5 Periodic Restarting . . . . . 22

**4 The General Structure of a Gossiping Operation** **24**

**5 A Gossiping Framework for Scalaris** **26**
5.1 Communication Model . . . . . 27
5.2 Interaction with the Callback Modules . . . . . 28
5.3 Atomicity and Coupling . . . . . 31
5.4 Rounds . . . . . 32
5.5 State of Behaviour Module . . . . . 33
5.6 Usage of the Framework . . . . . 33
5.7 Evaluation of the Framework . . . . . 34

**6 Gossiping Load Information in Scalaris** **37**
6.1 Interaction with the Behaviour Module . . . . . 37
6.2 Periodic Restarting . . . . . 39
6.3 State of gossip_load . . . . . 40
6.4 Usage of `gossip_load` . . . . . 40

6.5 Evaluation of `gossip_load` . . . . . 40

**7 Conclusion** **43**

**Appendices** **45**

**A Overview of Callback Functions** **45**
A.1 Type Definitions . . . . . 45
A.2 Startup and Shutdown . . . . . 45
A.3 Gossip Operation . . . . . 45
A.4 Config Functions . . . . . 47
A.5 Notifications . . . . . 48

**B Source Code** **49**
B.1 Google Code . . . . . 49
B.2 Github . . . . . 49

**Bibliography** **50**

# List of Figures

1 Example Mass Conservation Violation in Push-Pull Gossiping 17
2 Phases of a gossiping operation . . . . . 24
3 Prepare-Request Phase . . . . . 28
4 Prepare-Reply Phase . . . . . 29
5 Integrate-Reply Phase . . . . . 30
6 Load per Key Range . . . . . 38
7 Convergence Speed of Average Function . . . . . 42

# List of Tables

1 State of `gossip.erl` . . . . . 33
2 State of `gossip_load` . . . . . 40

# List of Algorithms

1 Basic dissemination algorithm, push-pull scheme . . . . . 5
2 Cyclon: Extended Shuffling . . . . . 10
3 Push-Pull Gossiping . . . . . 16
4 Symmetric Push Sum Protocol . . . . . 18
5 Symmetric Push Sum Protocol for Histogram . . . . . 20

# List of Code Fragments

1 Histogram: Merge procedure . . . . . 21

# 1 Introduction

The rise of ever growing computer networks poses new problems for the design of distributed systems. In large networks, the failure, disconnection and reconnection of nodes becomes the norm rather than the exception. In these highly dynamic conditions, traditional hierarchical client-server architectures reach their limits with respect to scaleability and robustness. Decentralised architectures like peer-to-peer networks offer a different framework for developing solutions.

One of the proposed solutions is called gossiping. Gossiping is a technique used to solve communication and computation tasks in peer-to-peer systems. It refers to the periodic, pairwise, message exchange between random peers in a network. Every node randomly chooses a neighbour to exchange information with. The information exchange is repeated periodically after a given time interval.

The communication problem gossiping tries to solve is that of information dissemination in large-scale distributed systems, i.e. the distribution of information which originates at one node to all the nodes of a network. In recent years, gossiping has also been expanded to solve computation problems, namely the aggregation of values spread across the nodes of a distributed system. Gossiping promises to solve this problems in a scaleable and robust way.

Scalaris [38], an ongoing research project at the Zuse Institut Berlin, is a distributed key-value store with ACID properties on transactions. ACID stands for atomicity, concurrency, isolation and durability and is a set of properties of transactions in a database system. While SQL based database system always guarantees these properties, most distributed databases have to relax the ACID properties in order to achieve full decentralisation (cf. [14, 29]). Guaranteeing ACID properties in a fully decentralised database is one of the major contributions of Scalaris.

Scalaris already uses gossiping for several purposes: For the construction and maintenance of the structured overlay a modified version of T-MAN [22, 40] is used. For the monitoring and passiv load balancing, gossip based aggregation of load informaiton is used. Based on Vivaldi Coordinates[11], a synthetic network coordinates system representing the latency between nodes, an agglomerative clustering algorithm is used for data center detection [39]. Cyclon [48], a gossip based peer sampling service, is used to provide the other gossiping protocols with references to random peers.

Scalaris is implemented using the programming language Erlang. Erlang is a natural fit for implementing distributed algorithms. It provides language level support for processes and message passing and can be seen as an implementation of the *actor model* [17]. The actor model is a popular model of concurrent computation and is often used to describe distributed algorithms. The basic primitives in this model are stateful "actors" which communicate

exclusively through messages. The implementations provided in this work will use Erlang as well.

**Contribution.** The contribution of this thesis work is twofold: First, to conceptualise and implement a gossiping framework for Scalaris. Based on the analysis of different classes of gossiping algorithms, an appropriate abstraction of different tasks is established. This is then used to implement a framework that provides the infrastructure necessary to implement miscellaneous gossiping protocols in Scalaris. The framework implements the parts generic to all gossiping protocols and defines an interface. This interface needs to be implemented by a concrete gossiping protocol to provide the parts specific to the gossiping protocol in question.

As mentioned above, gossip based load monitoring is already deployed in Scalaris, using the algorithm from [20]. The second contribution of this work is to improve the load monitoring by choosing an algorithm superior to the one currently used. The algorithm will then be extended to allow for the distributed computation of simple histograms. It will finally be implemented using the newly developed framework, thus also serving as a demonstration of the functionality of the framework.

The selection of the algorithm is based on a survey of the relevant literature, an experimental evaluation of the chosen algorithm is out of the scope of this work. References to works with experimental comparisons of the different algorithms are provided in the corresponding section. Although wireless networks, especially wireless sensor networks, are a common environment for gossiping algorithms, this is out of the scope of this work as well. As Scalaris is deployed in wired networks there is no need to discuss the effects of wireless environments on the use of gossiping algorithms.

**Outline.** Following the introduction, Section 2 will present an overview of the fundamentals of gossiping. This includes a discussion of gossip based information dissemination and membership management protocols. The concept of cycles is introduced and the robustness criteria used for analysing the algorithms are established.

Section 3 provides an extensive discussion of gossip based aggregation. Several algorithms for computing averages are introduced and the symmetric push-sum protocol is chosen as the algorithm to be used in the implementation of the gossiping of load information. The protocol is then extended for the computation of basic histograms. Next the symmetric push-sum protocol is evaluated with regard to the established robustness criteria. Finally, the concept of periodic restarting is applied to the protocol to improve the robustness with regard to node failures.

The general structure of a gossiping operation is developed in Section 4. The breakdown into different phases is designed to capture the gossiping algorithms from the previous sections and provide a basis for the implementation of the framework.

Section 5 gives an overview of the general structure and the generic parts of the framework and discusses the most important implementation decisions. The implementation is based on the analysis provided in the first part and will be evaluated against the robustness criteria from Section 2.4.

The design and implementation of load information aggregation will be discussed and evaluated in Section 6.

Finally, Section 7 will present the conclusion and an outlook on future research.

# 2 Fundamentals of Gossiping

Gossiping is a technique used to solve communication and computation tasks in peer-to-peer systems. Peer-to-peer systems are, in contrast to centralised client-server architectures, characterised by the lack of any hierarchical organisation or centralised control. All the nodes constituting a peer-to-peer system are in principle equal and can carry out the same functions. Peer-to-peer networks are usually built as overlay networks. Overlay networks are built on top of an existing network, where the participating nodes cannot communicate directly with arbitrary other nodes and thus have to use the communication channels provided by the overlay network. (cf. [44]).

Peer-to-peer networks can again be subdivided into structured and unstructured overlays. In both, every node only knows a subset of all nodes, called its *partial view*. Its known peers are called its *neighbours*. In a structured overlay the partial view is filled with references to peers complying to specific criteria. The most common form are overlays implementing a *distributed hash table* (DHT). By contrast, in an unstructured overlay the partial view is filled with a random sample of peers from the whole network.

The general idea of gossiping is as follows: A node randomly chooses a peer from its partial view and exchanges data with this peer. This is done by every participating node and repeated in a periodic fashion. The most basic example for the exchanged data would be a bit of information originating at one node. Through the periodic pairwise exchanges, the information is eventually distributed to all nodes. This kind of gossiping is usually referred to as *gossip based information dissemination*.

**Classes of Gossiping Algorithms.** Different classes of gossiping algorithms can be distinguished. The main classes are information dissemination, anti-entropy and aggregate computation (cf. [5, 13, 31]).[1] Dissemination will be discussed in the next section and computing aggregates will be discussed in Section 3. Membership management (see Section 2.2) is somewhat at odds with this classification, as it is a necessary part of nearly all gossiping algorithms, but it works like information dissemination and can be seen as a special case of it.

This leaves only anti-entropy gossiping. The idea is to resolve differences of replicated datasets by comparing the complete datasets in each gossiping operation. Hashes can be used to avoid comparing identical datasets. While this is a very reliable mechanism for distributing updates in a database, it is also very costly in terms of bandwidth. Seldom even referenced [5, 44], the single use case of anti-entropy gossiping seems to be for replica reparation in distributed databases [10, 13, 19]. Anti-entropy gossiping will therefore

[1] Kermarrec and Steen [31] additionally list topology construction and resource management, but describe them as special cases of information dissemination. Birman [5], on the other hand, subsumes topology construction under aggregate computation.

not be discussed further in this work.

## 2.1 Gossip Based Information Dissemination

Gossip based information dissemination, sometimes also referred to as rumor-mongering [13], is the oldest and most widely used form of gossiping. The main goal is to rapidly propagate information in a large network.

**Epidemics.** Gossiping is sometimes also referred to as "epidemic dissemination" or "epidemic algorithms" [13, 15, 47]. As the name suggests, this kind of information dissemination can be described analogously to the spread of epidemics, where a disease is spread by already infected members of a group which infect other individuals with whom they come into contact. In a network, the disease to be spread is the information to be distributed and the people of a group are the nodes of the overlay network. Just as in epidemics, the distribution of the information does not follow a predetermined path, but is random and can encounter already "contaminated" nodes. The dreaded characteristics of epidemics such as rapid spreading and high resilience are desired properties in a distributed system.

### 2.1.1 Basic Dissemination Algorithm

**Algorithm 1** Basic dissemination algorithm, push-pull scheme

- information dissemination can be started in any round by any player
- each node $P$ periodically choses $f$ nodes $Q_1, ..., Q_f$ uniformly at random from the entirety of the nodes
- if $P$ already is informed (already has the information), $P$ sends the information to $Q_1, ..., Q_f$ (push)
- if $P$ is uninformed, it requests the information from every $Q_1, ..., Q_f$
- A message is only forwarded max. $t$ times

Algorithm 1 shows a procedure for gossip based information dissemination. This procedure is repeated periodically at a predefined period of time, called a *cycle* (see Section 2.3). The important parameters of the algorithm are $f$ (called fanout) and $t$ (sometimes called age). The fanout $f$ regulates how many peers are contacted during each cycle and is the determining factor of the robustness of the algorithm. Kermarrec, Massoulié and Ganesh [30] give a detailed analysis of the effect of different fanout values. They conclude, that the fanout should be set to $f = \log(n) + \gamma$ (where $\gamma$ is a system configuration parameter). The age parameter determines the termination of the algorithm, in the simplest form it is just a counter and the information is no longer spread after a predefined number of send operations. The timely termination is important for keeping the number of messages sent as low as possible.

More advanced termination mechanisms are feasible. Karp et al. [26] for example propose a more robust mechanism which takes into account how often an already informed node receives the information again.

**Push, Pull and Push-Pull.** There are three principal schemes for how to exchange the information between peers: push, pull and push-pull. In a *push scheme*, an already informed node $P$ contacts a node $Q$ and sends it the information. The information can only be spread by nodes which already have the information. In the beginning, there are a lot of uninformed nodes so the probability of a informed node selecting an uniformed one is high and the information spreads exponentially, until about half of the nodes are informed. After that, the dissemination speed slows down significantly. Karp et al. [26] call this the shrinking phase.

In a *pull scheme*, a node $P$ would contact a node $Q$ and ask for information. Pure push schemes are not used in practise, because a node which wants to inject information needs to wait until it is contacted. But the pull scheme is faster in the shrinking phase, because the more nodes are informed, the higher the probability of contacting an informed node. According to Karp et al. [26], this point is reached when about half of the nodes are informed.

In a *push-pull scheme* the information is exchanged bidirectionally, which combines the advantages of both approaches. In practice, this is the most commonly used approach and regarded as having the best dissemination speed (cf. [6, 13, 21, 26, 44]).

**Time Complexity.** Gossip based information dissemination is considered to be exponentially fast, i.e. reaching exponentially more nodes every cycle. Obviously, the exact dissemination speed depends on the concrete algorithm used, especially on the fanout value and whether a push-, pull or push-pull scheme is used. Karp et al. [26] give a detailed analysis of the time and message complexity of different dissemination algorithms. They show, for example, that their push-pull scheme reaches all nodes in $\log_3 n + \mathcal{O}(n \ln \ln n)$ cycles with very high probability. In general, the convergence speed is expected to be logarithmic in the network size, i.e. in $\mathcal{O}(\log n)$, where $n$ is the size of the network. This makes gossiping a fast and scalable information dissemination technique.

**Guarantees of Delivery.** Due to the probabilistic nature of gossiping, there is no guarantee that all nodes receive a certain message. But gossiping can usually guarantee a very high probability of reaching all nodes. Kermarrec, Massoulié and Ganesh [30] show, that if the fanout value is set to $f = \log(n) + \gamma$ (where $\gamma$ is a configuration parameter) the probability of reaching all nodes is $exp\left(-exp\left(-\gamma\right)\right)$.

### 2.1.2 Related Work

Gossiping as information dissemination can be seen as a form of application level multicasting. The most common alternative approach is to build multicast trees on top of a structured overlay. Tanenbaum and Steen [44] describe a simple scheme to build a tree on top of Chord [43], which can be used for multicasting. Scribe [8] is a more elaborate system built on top of Pastry [37].

**Robustness.** The main advantage of gossiping algorithms is seen in the greater resistance to communication disruptions. Gossiping has an inherent redundancy, as every message is delivered multiple times through multiple paths. This, combined with the random peer selection is enough to ensure a very high robustness, i.e. even in the presence of node failures and message loss the information will still reach all nodes with very high probability. Kermarrec, Massoulié and Ganesh [30] give a detailed theoretical and experimental analysis in which they show, that this reliability is strongly related to the fanout value and and that by tuning the fanout (and other system parameters) success probabilities arbitrarily close to 1 can be achieved. Gossiping in this sense is a proactive algorithm, it does not require a mechanism to detect and reconfigure from failures. Structured approaches are usually reactive and can not give comparable guarantees for robustness. It follows that a major application domain for gossiping are highly dynamic networks with, for example, a lot of churn and very unreliable communication.

**Message Overhead.** The redundancy in the message delivery also causes one of the main disadvantages of gossiping: the higher message overhead. In gossiping, the number of messages needed to reach all nodes is in $\mathcal{O}(n \log n)$, whereas deterministic schemes can achieve message complexities of $\mathcal{O}(n-1)$ [26].

**Load Distribution and Latency.** Another advantage of gossiping is the very predictable load. The exchange data is usually assumed to be of fixed size and the gossip procedure is repeated periodically at a predefined time interval. Consequently, every node in the network produces the same amount of load per time interval, i.e. the distribution of the load over time and over all nodes is uniform. As the time interval is usually set to a value well above the transmission time of a message, gossiping has in practise a higher latency then other multicast solutions. These characteristics make gossiping especially suitable for applications that require continuous data dissemination in the background. The prime example for this is system monitoring.

**Scaleability.** Algorithms for structured peer-to-peer overlays need to maintain routing information, like the finger tables in Chord, to be able to distribute messages. Additionally, when considering multicast trees, the trees need to be constructed and maintained. The only "routing information" in gossiping is the partial view. Even if one considers extending the size of the

partial view in relation to the network size, the management overhead in gossiping is much smaller. Gossiping is therefore considered to have even better scalability than most algorithms for structured peer-to-peer networks.

## 2.2 Membership Management

So far, the existence of an overlay or the possibility of a node to choose a peer uniformly at random from the entirety of nodes has been assumed as given. But the construction of a random overlay and the filling of the partial views of the nodes needs to be facilitated somehow. This can be done by gossiping itself and most gossiping systems include some mechanism to do so. This aspect of gossiping is referred to as membership management.

**Centralised Overlay Construction.** There are other ways to construct the overlay, for example a server based protocol. All nodes who want to participate in the overlay have to register with the server. The server maintains a list of all nodes and provides joining nodes with a random sample of this list (the reference to the new node would also need to be distributed to the existing nodes). Such an approach comes with all the disadvantages of client-server architectures such as a single point of failure, scalability issues, reliability issues etc. and is in general undesirable in a peer-to-peer environment. Furthermore, membership management is one of the main strengths of gossiping.

**Random Graphs.** Unstructured overlays produced by gossip based membership protocols are modelled after and measured against random graphs. A random graph is a simple, connected graph $G$ in which pairs of vertices are connected by some probability. For a collection of $n$ vertices and for each of the $\binom{n}{2}$ possible edges, an edge $\langle h, u \rangle$ is added with a probability $p_{uv}$. For modelling random overlays in gossiping, usually only simple random graphs with the same $p_{uv}$ for every pair of distinct vertices $u$ and $v$ are of interest (cf [47]).

Random graphs exhibit some properties which are very interesting for gossiping, namely a short average path length, a low average clustering coefficient and an even degree distribution.[2]

The *average path length* is a metric for the number of hops to reach any given node from a given source. A short average path length is therefore an essential property of every overlay used for information dissemination.

The *clustering coefficient*, defined as the ratio of the existing links among the node's neighbours over the total number of possible links among them, shows to what percentage the neighbours of a node are also neighbours of each other. The clustering is important with regard to the connectivity of the overlay. A lot of clustering would increase the chances of network partitions.

[2] For the following remarks, cf. [48].

Higher clustering also causes more redundancy in the message delivery, which is bad for the dissemination speed.

The *degree distribution* shows how evenly connections are distributed between the nodes. The out-degree specifies how many other nodes one node knows and is relatively fixed by the size of the partial view. More interesting is the in-degree, which specifies by how many other nodes a node is known. It is desirable to have a low standard deviation in the node degree. It indicates a good connectivity and uniform load distribution.

### 2.2.1 Example Membership Protocol: Cyclon

A good representative of the class of gossip based membership management protocols and probably the most widely used is Cyclon [48]. The basic idea follows the same structure as in information dissemination: A peer is selected from the partial view and some information is exchanged with the peer. But in Cyclon the data to be exchanged is a random subset of the node's partial view. The receiving peer uses the subset to update its own partial view. Algorithm 2 gives the exact procedure for Cyclon. Cyclon uses so called extended shuffling, which extends the basic shuffling introduced in [42] with an ageing mechanism, which is then used to always chose the peer with the oldest entry. This dramatically improves the removing of references to dead nodes. By imposing a predictable lifetime on every reference it also facilitates a more uniform distribution of pointers. The main system parameters are the cache size $c$, the shuffle length $1 \leq l \leq c$ and the cycle length $\Delta T$. The cache is the partial view of the node, the shuffle length is the number of references exchanged between nodes and the cycle length is the amount of time when a node initiates a new shuffle operation. Based on their experimental analysis, Voulgaris, Gavidia and Steen [48] suggest values of $c = 20, 50$ or $100$, $l = 3 - 8$ and $\Delta T = 10s$ as a good compromise between convergence speed and message overhead.

**Properties.** Their empirical analysis also shows, that Cyclon converges with speed logarithmic to the network size to a network strongly resembling a random graph with the discussed properties such as a short average path length, a low average clustering coefficient and an even degree distribution.

Besides the desirable properties of the overlay, the main advantage of gossip based membership protocols in general and Cyclon in particular is the remarkable robustness. Removing up to 70% of the nodes of a network *at once* does not threaten the connectivity. Furthermore, thanks to the ageing mechanism the references to dead nodes are removed very quickly. Fewer than a number of cycles equal to the cache size are needed to "forget" dead links. So not only is the connectivity preserved, but the remaining nodes also quickly restrengthen the connectivity by replacing invalid references with valid ones. The latter is referred to as *self-healing behaviour*.

**Algorithm 2** Cyclon: Extended Shuffling

Every fixed period $\Delta T$, $P$ initiates a shuffle:

- Increase by one the age of all neighbours
- Select neighbour $Q$ with the highest age among all neighbours, and $l-1$ other random neighbours
- Replace $Q$'s entry with a new entry of age $0$ and with $P$'s address
- Send the updated subset to peer $Q$
- Receive from $Q$ a subset of no more than $i$ of its own entries
- Discard entries pointing at $P$ and entries already contained in $P$'s cache
- Update $P$'s cache to include *all* remaining entries, by *firstly* using empty cache slots (if any), and *secondly* replacing entries among the ones sent to Q.

Upon receiving a set of neighbours $S$ from $P$:

- Select a random subset of $l$ neighbours from $Q$'s cache
- Send the selected subset to peer $P$
- Discard entries in $S$ pointing at $Q$ and entries already contained in $Q$'s cache.
- Update $Q$'s cache to include *all* remaining entries in $S$, by *firstly* using empty cache slots (if any), and *secondly* replacing entries among the ones sent to P.

**Application Domains.** The main application domain for membership protocols is gossiping itself. The membership protocol can be integrated with a gossiping application or run as a separate peer sampling service. When integrating the membership protocol with the gossiping application, the messages can be combined, which reduces the overall number of messages. Furthermore, the peer sampling can be tailored to the specific task. Having a separate peer sampling service has the advantage that multiple gossiping applications or other components can use the same service.

The latter gives rise to a whole new application domain for gossiping, usually referred to as topology construction. Here, the random network constructed by a membership protocol is used as a substrate for building a structured overlay through gossiping. The basic idea is similar to the described shuffling mechanism, but instead of a random selection of nodes for the cache (or view) the nodes are selected with a ranking or optimisation function. Two examples for gossip based topology construction algorithms working like this are T-Man [22] and X-Bot [32].

### 2.2.2 Related Work

Cyclon can be classified as a protocol using a cyclic strategy, meaning the partial view is updated continuously and independently of overlay related events such as joining, leaving or failing nodes. An alternative, reactive

strategy is followed by Scamp [35]. In Scamp, the partial view is only updated when nodes join or leave, failing nodes are detected with heartbeat messages. Scamp compares favourably in relatively static and error free environments due to the smaller message overhead but lacks the robustness necessary for highly dynamic networks. HyParView [33], on the other hand, is optimised for higher robustness even than Cyclon. It uses two partial views combined with a node failure detection mechanism, but comes at the cost of higher consumption of resources, especially TCP connections (which are used for failure detection).

## 2.3 Cycles

Even though most gossiping algorithms are not synchronous, it is useful to refer to the execution of the protocol in terms of cycles.[3] Assuming a push-pull scheme, a cycle is defined to be the time period during which $f \cdot 2n$ exchange operations have been made, where $n$ is the number of nodes and $f$ the fanout (see section 2.1.1). During a cycle, each node *initiates* exactly $f$ data exchanges. How often a node is *contacted* for a data exchange depends on the randomness of the peer selection, but on average every node is contacted $f$ times per cycle. It should be obvious that assigning exchange operations to a specific global time interval only works in a synchronous environment. In asynchronous environments this can only be an approximation. Due to uncertainties of different clock speeds and message delivery delays, there might never be a point in time where all nodes finished the same cycle, i.e. cycles can interleave. As most environments in which gossiping is deployed cannot guarantee synchronicity, this should be carefully considered when selecting suitable gossiping algorithms.

## 2.4 Robustness in Practical Settings

When implementing a gossiping algorithm, practical application settings of distributed networks need to be considered. The practical settings often diverge from the simplified assumptions made by the designers of the algorithm. This can affect the correctness of the algorithm, especially in the case of gossip based aggregation (see Section 3). In this work, when discussing the algorithms used and the implementation of the framework, the following three aspects will be considered (cf. [23]):

First, the *communication model*. Here one can distinguish between synchronous and asynchronous operation modes. In a *synchronous model*, restrictions are placed on when message delivery and computations can occur. The execution proceeds in lock-step rounds, i.e. there is no uncertainty of timing.

[3] Although the term "cycle" is widely used [6, 13, 20, 22, 48], "round" is also commonly used [5, 15, 26, 31, 47] to refer to the same idea. The term "round" however is also used to refer to another concept (see Section 3.5), hence "cycle" is used in this work.

In an asynchronous model, no such restrictions are placed on the timing. Message delivery is assumed to take a finite but unknown amount of time. The communication is, however, assumed to be in FIFO order, i.e. no message already sent can be surpassed by a message from the same source.

Second, the *reliability of message delivery*. Message loss refers to the loss of communication data due to temporary link or node failure. Note that message loss can occur even if reliable point-to-point connections are used, e.g. in systems where messages are routed on the application layer the failure of a node can cause message loss.

Third, *node failures*. The node failures considered here are only crash failures, i.e. node failures where the nodes stops working at an arbitrary time. Byzantine failures are out of the scope of this work.

# 3 Gossip Based Aggregation

Traditionally, gossiping was mainly associated with information dissemination. More recently, gossiping techniques have been adopted to solve aggregation problems in distributed systems. Today, this is the most active area of research (cf. [23]).

In gossip based information dissemination the basic idea is to spread some bit of information originating at one node to all the other nodes. Gossip based aggregation, on the other hand, starts from a set of multiple input values spread across all nodes of the system. Examples for interesting values to aggregate include load information, sensor readings and application layer information like the number of files stored on a file server. These local values are iteratively combined through gossiping with neighbouring peers, using an aggregation function such as sums, averages, and extremal values. The result of each iteration is called the current (local) estimate. In the end, the local estimates aggregated at each peer converge towards the same global aggregate. Gossip based aggregation thus is a way of providing local access to information about global properties. (cf. [6, 20, 23]).

It might not always be possible to converge to the true global aggregate, sometimes only an approximation is possible. This can be due to algorithmic limitations as some more complex aggregation functions are intrinsically approximative in nature, e.g. q-digest based quantile computation [18, 41]. In other cases system factors such as failing nodes and links can cause an otherwise exact algorithm to produce erroneous results, i.e. the achieved estimated aggregate is less accurate.

The complexity of gossip based aggregation is generally considered to be in $\mathcal{O}(\log n)$, resulting in a message complexity of $\mathcal{O}(n \log n)$. More details will be given when discussing concrete algorithms below.

## 3.1 Related Work

The centralised approach to aggregation is a single node which retrieves all the values from the other nodes and performs the aggregation locally at the collecting node. This can work effectively in small and static networks, but the centralised approach comes with well known disadvantages: The concentration of traffic and processing load at the central node leads to problems with scalability. As the central node is the single point of failure, reliability is also an issue (cf. [12]).

Trying to decentralise the aggregation, several distributed aggregation protocols have emerged in recent years. They can be divided into two main classes: tree based and gossip based aggregation protocols.

**Tree Based Aggregation.** The main idea of tree based aggregation protocols is to organise all nodes into an overlay tree on which the aggregation is performed. The aggregation is then performed from the leaf nodes to the

root node of the tree. A representative example for a tree based aggregation protocol is GAP (Generic Aggregation Protocol) [12]. GAP constructs and maintains a breadth first search (BFS) tree on an existing overlay. The BFS tree is then used to compute the aggregates, every node computes partial aggregates of its subtree and the aggregates are then sent along the tree towards the root. For maintaining the BFS tree, the partial view of each node also contains entries of non-tree nodes with which messages are exchanged as well. Assuming the availability of a failure detection service, GAP is able to deal with node failures.

GAP has since been extended for different purposes. Wuhib et al. [49] use GAP as basis for detecting threshold crossing alerts (TCAs) for network-level variables. A-GAP[36] is an extension to GAP focussing on configurable accuracy, allowing for trade-offs between accuracy and protocol overhead. Astrolabe [46] is an earlier attempt to a general aggregation solution, but does not provide the same robustness as GAP and its successors.

**Qualitative Comparison.** There are qualitative and quantitative differences between gossip and tree based aggregation algorithms (cf. [50]). First, gossip based protocols are usually simpler, as they do not need to construct and maintain an additional overlay for the aggregation. Second, in gossiping the result of the aggregation is available at every node of the system while in tree based approaches only the root node knows the aggregation result. Third, node failures effect both approaches in very different ways. In gossip based aggregation, the main concern with node failures is the resulting mass loss (see Section 3.4), while a tree based aggregation protocol needs to reconstruct the aggregation tree. Forth, using an aggregation tree leads to a higher load on the root node and its children, as the aggregation occurs along the tree towards the root (cf. [45]). This can become a scalability issue in very large networks, although this is still far superior to centralised approaches. Gossip based aggregation, on the other hand, is characterised by a uniform load distribution (see also Section 2.1.2).

**Quantitative Comparison.** In the quantitative comparison, the published simulation results indicate advantages for tree based aggregation. Wuhib et al. [50] compare GAP and G-GAP[4], a gossip based aggregation protocol. They compare the estimation accuracy of the aggregation in function of the round rates, the network size and the failure rate and conclude, that the GAP consistently outperforms the gossip-based protocol. Using their own extension of G-GAP, Tesfaye [45] provides an even broader experimental comparison with GAP. The author finds some cases where gossiping compares favourable, but in general the findings are in accordance with [50].

[4] G-GAP [50] stands for Gossip-based Generic Aggregation Protocol and is an extension of the push-synopses protocol from [28]. The focus of G-GAP is on continuous monitoring and robustness.

## 3.2 Computing Averages

This chapter analyses gossip based aggregation algorithms further, namely algorithms for averaging. In addition to computing the average, all the algorithms discussed are capable of computing count, sum and extremal values as well.

**Continuous Aggregation vs Periodic Restarting.** The values to be aggregated might change over time, e.g. when trying to aggregate sensor readings the sensor may periodically or continuously produce new readings. There are two ways of dealing with dynamic values: First, periodic restarting of the protocol and second, protocols that inherently allow for continuous aggregation of changing values. The basic idea of periodic restarting is the following: The aggregation is started with a set of local values which are aggregated. After the current set of values has converged, the local values are read again and the aggregation process is restarted. This amounts to periodically computing the aggregate of a snapshot of all values at a certain point in time. Assuming no node failures or message loss, this approach converges to the true global aggregate value.

In protocols which allow for continuous aggregation, changing values are constantly fed into the aggregation process and the protocol deploys some mechanism to deal with the dynamics, i.e. some mechanism to ensure that the protocol still converges to a meaningful global aggregate. Assuming no changes in the values aggregated, these protocols would also converge to the true average. In the presence of changing values, however, only an approximation is achievable at any point in time. Examples for protocols which allow for continuous aggregation are G-GAP [50], flow updating [2, 24, 25] and the evaporative approach presented in [4].

Whether the precise aggregation of snapshots or the continuous approximation is preferable depends not least on the application scenario. Additionally, the ability to deal with continuously changing values usually comes at the cost of higher message overhead. On the other hand, the ability to handle dynamic values can be extended to network dynamics, that is to say these algorithms have interesting properties with regard to robustness of node failures (periodic restarting is also a means of dealing with failing nodes, see Section 3.5 for details). Extensive analytical or empirical comparisons of the two approaches still have to be done.

Periodic restarting presents a simple and effective way of dealing with dynamics in networks and values. Additionally, G-GAP and flow control rely on a more static overlay than Cyclon provides, hence it is not clear how well they would work within Scalaris. This work will therefore concentrate on the analysis of algorithms with periodic restarting. The continuous approach offers interesting opportunities for further research, especially a detailed comparison with the algorithms presented here would be of interest.

Another approach is provided by Kashyap et al. [27]. They present an

algorithm optimised for message complexity rather than time complexity. By allowing a slightly worse time complexity of $\mathcal{O}(\log n \log\log n)$ cycles they can provide a message complexity of $\mathcal{O}(n \log\log\log n)$. As this work is more interested in convergence speed than the message overhead, this algorithm is not discussed further.

**Mass conservation.** Before discussing the different averaging algorithms, an important property of these protocols needs to be explained. Mass conservation [28] (also cf. [6, 23, 50]) states, that at any time $t$

$$\sum\nolimits_i x_i = \sum\nolimits_i v_i$$

where $x_i$ refers to the original local value at node $i$ and $v_i$ to the current local estimate at node $i$. Put into words, mass conservation expresses, that the sum $\sum_i v_i$ of all local estimates in a network is invariant at any time. When mass conservation is given, the protocol is guaranteed to converge to the true average. In this context, $\sum_i v_i$ is often referred to as the global mass. In the algorithms using a sum $s_i$ and and a weight $w_i$ (see below), $v_i$ is calculated as $v_i = {}^{s_i}/_{v_i}$.

**Algorithm 3** Push-Pull Gossiping

```
x0: initial local value

initialize
  v = x0

every interval time units
  j = getNode()
  j ! { v, true }

upon event received {v', r} from j
  if r is true then
    j ! {v , false}
  end if
  v = (v + v')/2
```

**Push-Pull Gossiping.** In the current implementation, Scalaris uses the push-pull gossiping protocol from [20][5], as shown in Algorithm 3. Periodically, a node $P$ starts a gossiping operation by sending its current estimate $v_p$ to the selected peer $Q$. $Q$ replies by sending back its current estimate $v_q$. Both $P$ and $Q$ update their local estimate by computing $v = {}^{(v_p + v_q)}/_2$. When $v$ is initialised with the local values at all nodes the average is calculated, initialising all but one node with $v = 0$ and one node with $v = 1$ will return ${}^1/_n$, i.e. it can be used to calculate the size of the network. The sum of all local values can be calculated as average times size. The theoretical and

[5] Boyd et al. [7] provide a nearly identical algorithm.

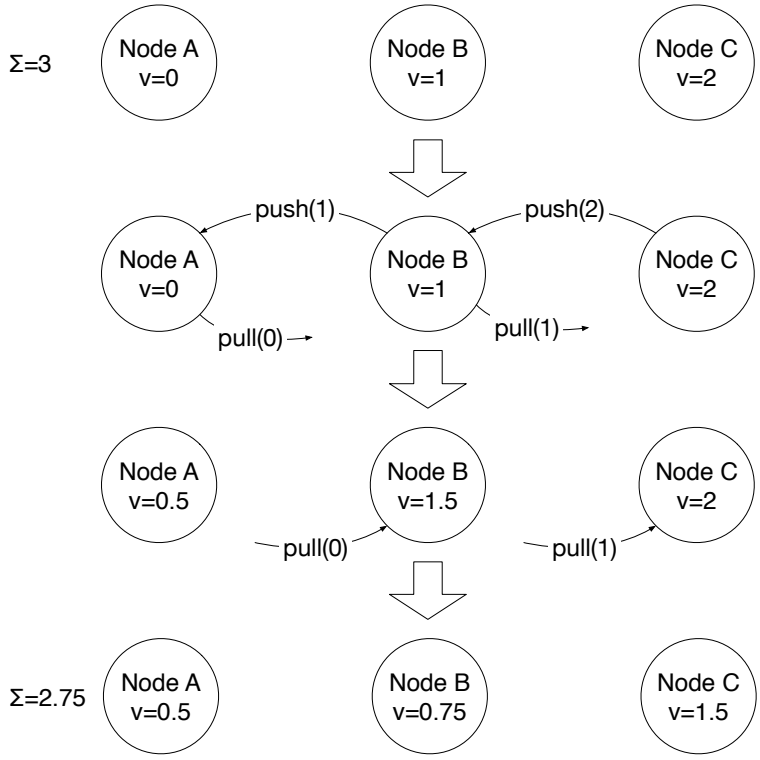


Figure 1: Example Mass Conservation Violation in Push-Pull Gossiping

experimental analysis provided in [20] show that the push-pull gossiping protocol is within the time and message complexity bounds given above (see page 13).

The main disadvantage of this algorithm is, that it violates the mass conservation invariant. As shown by Jesus, Baquero and Almeida [23], the push-pull protocol needs atomicity for the operation $v = {}^{(v_p + v_q)}/_2$ to not violate the mass conservation. This is true even in synchronous settings, i.e. without interleaving between different cycles. Figure 1 (cf. [23]) shows one cycle of a gossiping operation with three nodes involved. Node B sends a pull request to Node A while also receiving a pull request from Node C. Node B then updates its current value and uses this value when the message from Node A arrives. The mass of all nodes has changed now, $\sum_i x_i \neq \sum_i v_i$, i.e. mass conservation is violated.

Message interleaving can also result from asynchronous communication where messages from different cycles interleave. It should be evident from the above, that this would also result in mass conservation violations.

**Push-Sum Protocol.** Another approach is the push-sum protocol presented by Kempe, Dobra and Gehrke [28], which guaranties mass conservation in the case of asynchronous communication. The algorithm is the same as

Algorithm 4, just without lines 13-16 (which also supersedes the use of `r`). The basic idea is that every node $i$ maintains the local estimate as a sum $s_i$ and a weight $w_i$. The sum is initialised with the local value: $s_i = x_i$, and the weight is initialised to $w_i = 1$. The weight $w_i$ expresses the contribution of the sum $s_i$ to the current estimate $v_i = {}^{s_i}/_{w_i}$.

At the beginning of every cycle, a node $P$ sends the tuple $\{{}^{s_p}/_2, {}^{w_p}/_2\}$ to a randomly chosen peer $Q$ and stores ${}^{s_p}/_2$ and ${}^{w_p}/_2$ as its new sum and weight. When $Q$ receives the message, the local sum and weight are updated with the received sum and weight as $v_q = v_p + v_q$ and $w_q = v_p + w_q$. The whole exchange can be interpreted as dividing the weight of the estimate between the $P$ and $Q$.

Kempe, Dobra and Gehrke [28] provide detailed theoretical analysis of the time complexity of the push-sum protocol. They show that with probability at least $1 - \delta$, the protocol converges within $\mathcal{O}\left(\log n + log{}^1/_\varepsilon + {}^1/_\delta\right)$ cycles to an approximation of the average within $\varepsilon$, $\delta$ and $\varepsilon$ being two arbitrary small positive constants.

The primary disadvantage of the push-sum protocol is that it is a pure push-based gossip scheme. As detailed in Section 2.1.1, push-pull schemes are considered to have a better convergence speed than pure push based designs.

**Algorithm 4** Symmetric Push Sum Protocol

```
1  x0: initial local value
2  w0: initial local weight
3
4  initialize
5    s = x0, w = w0
6
7  every interval time units
8    j = getNode()
9    s = s/2,  w = w/2
10   j ! { {s, w}, true }
11
12 upon event received { {s', w'}, r } from j
13   if r is true then
14     s = s/2,  w = w/2
15     j ! { {s, w}, false }
16   end if
17   s = s + s', w = w + w'
```

**Symmetric Push-Sum Protocol.** The symmetric push-sum protocol [23] combines the robustness of the push-sum protocol with the better convergence speed of push-pull schemes. The symmetric push-sum protocol, as detailed in Algorithm 4, extends the push-sum protocol by adding replies

to the scheme, effectively making it a push-pull scheme.[6] Upon receiving a message, the node checks whether the message is a request (`r` is `true`), or if it is a reply. If its a request, the node sends back its own $s/2$ and $w/2$ and divides its local estimate by two. Irrespective of whether it is a request or reply, the local estimate is then updated to incorporate the foreign values.

Blasa et al. [6](also cf. [23]) provide extensive experimental comparison of push-pull gossiping, the push-sum protocol and the symmetric push-sum protocol, demonstrating the better performance of the latter with regard to accuracy and convergence speed.

The symmetric push-sum protocol can be used (like the other algorithms) for computing various aggregates including the average, the sum and the count of nodes. Which aggregates are computed depends on the initial values: To calculate the average, one initialises $s_i$ to the local value and $w_i$ to $1$ at the beginning of the protocol (or at the beginning of a new round). To compute the sum, $s_i$ is set to the local value again, but $w_i$ is set to $1$ at exactly one node and to $0$ at all other nodes. The count of all nodes can be computed the same way as the sum, but initialising $s_i$ to $1$ at every node.

Based on the discussed advantages of the symmetric push-sum protocol, namely the better robustness towards message interleaving and the superior convergence speed, this is the algorithm chosen for the implementation of load monitoring in Scalaris. The rest of this section will provide further analysis and extension of this algorithm.

## 3.3 Histograms

The symmetric push-sum protocol can be extended to compute basic histograms. To compute a histogram with this protocol it has to fulfil the following conditions: First, the number and width of the buckets needs to be known to all nodes when starting the protocol. Second, given the number and width of the buckets, every node needs to be able to locally decide upon initialisation what its contribution to one or multiple buckets of the histogram is. Third, every bucket of the histogram can be computed as an average of the local contributions. More general solutions to gossip based histogram computation exist, e.g. using q-digests [18, 41], but they are beyond the scope of this work.

Given that the above conditions are met, a histogram can be computed using the Algorithm 5 analogously to the symmetric push-sum protocol described

[6] Blasa et al. [6, p. 29] call the reply an "asynchronously perform[ed] [...] symmetric push operation", presumably to emphasize the difference to the push-pull protocol: In the push-pull protocol, the push and pull depend on each other, the updating of the local estimate can only take place when the pull was successful. In the symmetric push-sum protocol, the push and the symmetric back-push are independent of each other. In this work push-pull is used in the broader sense of bidirectional communication (see section 2.1.1), hence the symmetric push-sum protocol is considered a push-pull scheme.

**Algorithm 5** Symmetric Push Sum Protocol for Histogram

```
L₀: initial local value(s)
H₀: [{Key1, undef}, {Key2, undef}, ...]: initial empty histogram

initialize
  H = initialize(L₀, H₀)

every interval time units
  Q = getNode()
  H = divede2(H)
  Q ! { H, true }

upon event received { H', r } from I
  if r is true then
    H = divide2(H)
    I ! { H, false }
  end if
  H = merge(H, H')
```

in Section 3.2: The function `initialize(L, H)` calculates the contribution of the nodes' value(s) to the possibly multiple buckets of the histogram and populates the histogram with these initial values. As in the push-sum protocol, the local estimate for every bucket $j$ is computed as a tuple of sum $s_j$ and the weight $w_j$, resulting in buckets the form $\{Key, \{s_j, w_j\}\}$.

At the beginning of every cycle, a peer $Q$ is selected and the histogram is prepared for the data exchange. The function `divide2(H)` divides the sum $s_j$ and the weight $w_j$ of all buckets by two. The prepared histogram is then sent to $Q$. Upon receiving a message, the node checks whether the message is a request (`r` is `true`), or if it is a reply. If it is a request, the node prepares its local estimate of the histogram with `divide2(H)` and sends it to the requesting node. Irrespective of whether it is a request or reply, the local estimate is then updated with the received histogram, using the merge procedure given in Listing 1.

Note, that it is only possible to extract the global average from a histogram, if one keeps track of how many nodes contribute to each bucket.

## 3.4 Evaluation of symmetric push-sum protocol

In this section, the properties of the (symmetric) push-sum protocol towards the aspects of robustness introduced in Section 2.4 are discussed. If not stated otherwise, cf. to [6, 23, 28] for the whole section.

**Communication Model.** The push-sum protocol has been shown to be robust in terms of message interleaving in asynchronous environments. The symmetric push-sum protocol, which adds a reply mechanism to the push-sum protocol, has the same properties with regard to message interleaving.

**Listing 1** Histogram: Merge procedure

```
merge(H1, H2) ->
    zipwith(fun merge_bucket/2, Histogram1, Histogram2).

merge_bucket({Key, undef}, {Key, Value2}) ->
    {Key, Value2};

merge_bucket({Key, Value1}, {Key, undef}) ->
    {Key, Value1};

merge_bucket({Key, Value1}, {Key, Value2}) ->
    {Key, merge_avg(Value1, Value2)}.

merge_avg({Sum1, Weight1}, {Sum2, Weight2}) ->
    {Sum1+Value2, Weight1+Weight2}.
```

Message delays may reduce convergence speed, but the mass conservation property is always guaranteed as long as all messages have been received.

**Message Loss.** In principal, the loss of a message directly results in the loss of weight and therefore the violation of the mass conversation property. How much a mass loss affects the accuracy of the estimated aggregate depends on when during the gossiping the message loss occurs. The later the mass loss occurs, i.e. the more the local estimates already converges to the true global average, the smaller the effect on the accuracy of the estimated aggregate. Message loss in the earlier stages of the gossiping process, on the other hand, affects estimation of the global aggregate. It does not completely invalidate the results but reduces the accuracy of the estimates.

If nodes are able to detect message loss, the originating node will be able to reintegrate the values from the lost message into its local estimate. This way, no mass is lost and mass conversation is still ensured. In order to keep the estimation as accurate as possible, one should use every possibility to detect message loss, even if it might not be possible to detect all lost messages. The only exception would be a one-sided optimisation for low message overhead, as the detection of message loss might increase the number of messages.

**Node Failures.** Node failures affect the accuracy of the aggregation. The local value $x_i$ of a node $i$ is used to initialise its sum $s_i$. Once the gossiping has started, shares of the node $i$ are distributed to other nodes and incorporated into their local sums and weights. When node $i$ fails, its shares still remain in the system, which violates the mass conservation property.

With regard to the effect of the mass conservation violation on the accuracy, what was said about message loss holds for node failures as well: The more the local estimates have already converged, the smaller the effect on the accuracy of the estimation.

In the proposed form, neither the push-sum protocol nor the symmetric

push-sum protocol support node failure. The next section discusses, how periodic restarting can provide a mechanism to deal with node failures.

## 3.5 Periodic Restarting

As discussed in Section 3.2, the periodic restarting is a mechanism to deal with the dynamics of changing values. But the periodic restarting fulfils a dual purpose, it is also a mechanism of dealing with failed nodes.

Periodic restarting of the protocol was first proposed in [20] for the push-pull gossiping algorithm, but it can be applied to the (symmetric) push-sum protocol as well. The term "round" is used to refer to the differed repetitions of the protocol. A round is defined as the period of time until the protocol is restarted.

A round proceeds as follows: At the beginning of every round, the sum $s_i$ is initialised with the current local value $x_i$ and the weight $w_i$ is set to 1. The local estimates are then aggregated as sum and weight through Algorithm 4, until the local estimates have converged to the global aggregate. The termination of a round can be determined either by predefined criteria like minimum and maximum number of cycles or, preferably, by a convergence criterion. The criterion will be referred to as the new-round criterion. An example for a convergence criterion would be

$$\forall c - u < x \leq c : |v_x - v_{x-1}| \leq \varepsilon$$

where $c$ is the current cycle and $u$ is a system parameter. In words: Compare the change of the local estimate $v$ over the last $u$ cycles and terminate, when the change is smaller than a predefined $\varepsilon$.

In a naiv implementation, the value delivered to the application querying for the aggregation result would always be the current estimate. This would imply that every time a new round is started, the estimation error would suddenly spike, until the values of the new round have converged. This can be optimised (cf. [4]) by always keeping the properly converged aggregation result from the previous round. When queried, the previous result can be returned while a new result is computed in the current round.

This can be further optimised by deploying an additional (weaker) version of the new-round criterion, e.g. by comparing only the last $w < u$ rounds or using a larger $\varepsilon$ (this will be referred to as the best-value criterion). Upon a query for the aggregation result, the previous estimate would only be returned when the estimate of the current round does not fulfil the best-value criterion. By varying the parameters of the convergence criteria, the trade-off between accuracy and more up to date results can be fine-tuned.

**Periodic Restarting and Node Failures.** As detailed in the last section, when a node fails after the value of the node has already become part of the aggregation process, the mass conversation property is violated. The data

from a failed node keeps impacting the current round and when the round finishes, it will stay in the system as data from the last round. This means that all traces will have left the system two rounds after a node failure (upper bound). When using the discussed optimisations, the impact of the failed node will vanish even faster in practice. As soon as the current estimate has sufficiently converged according to the best-value convergence criterion, the estimates of the current round will be returned to a query. This means that a failed node's information stops impacting the result even before the next round finishes.

Applying the concept of periodic restarting is a major improvement of the (symmetric) push-sum protocol with regard to robustness against node failures.

# 4 The General Structure of a Gossiping Operation

To develop a gossiping framework it is useful to further examine the general structure of a gossiping operation. On the basis of the analysis performed in the previous sections, a breakdown into phases is provided in this section. This breakdown should be sufficiently general to capture most gossiping algorithms and sufficiently specific to provide a basis for implementation. A gossiping operation is defined as the interaction between two nodes in one cycle.

As discussed in Section 2.1.1, a push-pull scheme offers the best overall performance. It is also the more general scheme: if one would want to implement a pure pull scheme this would be possible in a push-pull framework by simply leaving the the push party empty. It wouldn't be possible however, to implement a push-pull scheme in a pure push framework. Consequently, from here on, a push-pull scheme is assumed.

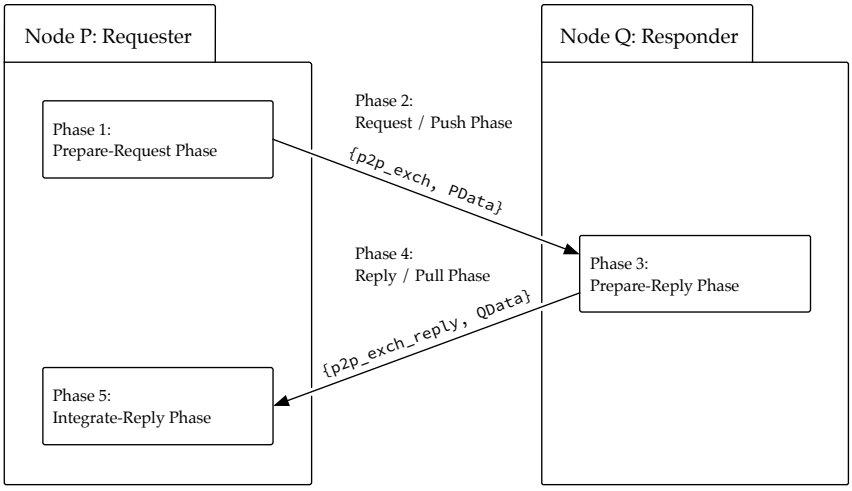


Figure 2: Phases of a gossiping operation

Figure 2 depicts the breakdown of a gossiping operation into phases. Two peers are considered, $P$ and $Q$. $P$ acts as a requester, initiating the gossiping operation. This corresponds to phases one, two and five. Kermarrec and Steen [31] call this the *active thread*. The other node, $Q$, acts in a more passive role as a responder (phase three), called the *passive thread*. In one cycle, each node is exactly once in the active requester role. How often a node is in the passive role differs, due to the random choosing of the peers. However, on average, each node is once in the passive role as well (see Section 2.3).

**Phases of a Gossiping Operation.** In the *prepare-request* phase, $P$ selects the data to be exchanged and a peer $Q$ to exchange the data with. Most gossiping algorithms assume that they can select a peer uniformly at random from the

entire set of peers, which can be achieved by means of a membership protocol (see Section 2.2). The selection of the exchange data is entirely application dependent and might involve some preprocessing at $P$.

$P$ then sends the data to $Q$ in the *request* phase or *push* phase. To distinguish the messages from phases two and four, different message tags are used. For phase two, the message tag `p2p_exch` is used.

In the third phase, the *prepare-reply* phase, $Q$ receives the request. It then integrates the data and prepares a reply with its own data. The integration of the data is again dependent on the application, but it usually includes some form of processing the data, for example merging the received data with its own data.

The data is then sent back to $P$ with the `p2p_exch_reply` message tag in the *reply* (or *pull*) phase.

In the last phase, the *integrate-reply* phase, $P$ receives the reply to the request from the first phase with the data from $Q$ and integrates the data with its own data. The data processing in this phase is not necessarily identical to the data processing in phase three.

**Logical coupling of Phases.** It is important to clarify two aspects concerning the logical and temporal coupling between the different phases of a gossiping operation. First, the logical and temporal coupling between a prepare-request and an integrate-reply phase *within one node*. Every `p2p_exch_reply` is the response to an *earlier* `p2p_exch` request. Depending on the atomicity guarantees given by the implementation, a received `p2p_exch_reply` might however not be the answer to the *last* `p2p_exch` request issued.

Second, there is no logical and temporal coupling whatsoever between the prepare-reply and the other phases *within one node*. A `p2p_exch` can be received at any point within a cycle and the gossiping implementation needs to be able to handle this.

# 5 A Gossiping Framework for Scalaris

This section will give an overview of the design of the gossiping framework and discuss the most important implementation decisions. The implementation is based on the analysis provided in the first part: The framework is designed to allow the implementation of gossip based dissemination (see Section 2.1) and gossip based aggregation (Section 3) protocols. Anti-entropy gossiping was not considered, because it is not relevant to Scalaris (see page 4). The communication scheme used by the framework is push-pull gossiping (Section 2.1.1), as this offers the best speed of convergence. The general structure of a gossiping operation used as basis for the framework was established in Section 4.

The membership protocol used for the peer selection is Cyclon (Section 2.2.1). The framework also allows for the periodic restarting of gossip protocols by offering round handling (Section 3.2, Section 3.5).

**Components of the Framework.** The gossiping framework comprises three kinds of components, using the *behaviour* feature of Erlang: [7]

First, the *gossiping behaviour* (interface) `gossip_beh.erl`. The behaviour defines the contract that allows the callback module to be used by the behaviour module. The behaviour defines the contract by specifying functions the callback module has to implement. An overview of all callback functions with short descriptions is given in Appendix A.

Second, the *callback modules*. A *callback module* implements a concrete gossiping protocol by implementing the `gossip_beh.erl`, i.e. by implementing the functions specified in the `gossip_beh.erl`. The callback module provides the protocol specific code. For this thesis, `gossip_load.erl` has been implemented as an example callback module (see Section 6).

Third, the *behaviour module* `gossip.erl`. The behaviour module provides the generic code of the gossiping framework. It calls the callback functions of the callback modules defined in `gossip_beh.erl`. The generic parts of the framework include the following:

- The integration into Scalaris by implementing `gen_component`, handling the startup within a node and interacting with other components of Scalaris like the ring maintenance
- The abstraction of large parts of the message handling by translating the messages to function calls to the callback module and handling the data exchange with peers
- The selection of peers
- The starting of new cycles

[7] Although the naming of the components of a behaviour is inconsistent in the literature, most common is *behaviour*, *callback module* and *behaviour module*, cf. [1, 3, 9]. Logan, Merritt and Carlsson [34] are using *behaviour interface*, *behaviour implementation* (for the callback module) and *behaviour container* (for the behaviour module) instead.

- The handling of rounds and the leader determination for the periodic restarting of the protocol
- The starting and stopping of gossiping tasks during the startup of Scalaris as well as later on
- Providing an API for value extraction

The functioning of the framework is discussed in detail in the following sections.

**Relationship Between Callback and Behaviour Module.** The relation between behaviour and callback module could be modelled as either a one-to-one or a one-to-many relation. In a one-to-one relation, a process is spawned for each callback module. The behaviour module and the callback module are both executed in the context of this process. This amounts to creating an instance of the behaviour module for each callback module. This model is used for example for Erlang's `gen_server` behaviour module.

For the gossiping framework it was decided to model it as a one-to-many relation. That is to say, the behaviour module is implemented as single process (per node) and all the callback module run in the context of this single process. This increases the complexity of the code, as the behaviour module needs to handle different callback modules, but has the advantage of reducing the number of spawned processes and allowing for a better grouping of messages.

**Startup.** The framework is started as part of the startup procedure of a Scalaris node. The framework maintains a list of callback modules in the `CBMODULES` macro which are started together with the framework. It is also possible to individually start and stop callback modules later, see Section 5.6 for details.

## 5.1 Communication Model

Scalaris is an asynchronous environment in the sense described in Section 2.4: no restrictions are placed upon the timing of message delivery and computations.

The general pattern for communication between the behaviour module and a callback module is the following: From the behaviour module to a callback module communication occurs as a call to a function of the callback module. These calls have to return quickly, no long-lasting operations, especially no receiving of messages, are allowed. Therefore, the answers to these function calls are usually realised as messages from the respective callback module to the behaviour module, not as return values of the function calls. The communication is therefore not only asynchronous in the classical sense that it does not allow blocking the caller. It is also asynchronous in the means of communication as it uses function calls as well as message sending.

The asynchronous communication yields the known advantages, such

as responsiveness and better resource utilisation. Without asynchronous communication, every call to a callback module would block the whole behaviour module, even if the callback needs to execute long running operations like requesting local information from another process. With asynchronous communication, on the other hand, multiple callback modules can issue requests concurrently. Without this, it would not be possible to implement the behaviour module as a single process.

## 5.2 Interaction with the Callback Modules

The general structure of a gossiping operation was established in Section 4 and depicted in Figure 2. This section describes how the different phases are implemented in the behaviour module.

For the gossiping framework, only the phases one, three and five are important. The sending and receiving of messages (in phases two and four) is handled by Scalaris' communication layer and `gen_component` (which the framework implements). The `gen_component` delivers the messages via message handlers to the gossiping framework, which implements two message handlers: `on_inactive()` and `on_active()`. The first handles messages during the startup of the framework and when the framework has been deactivated. The `on_active` handler handles messages during normal operation.

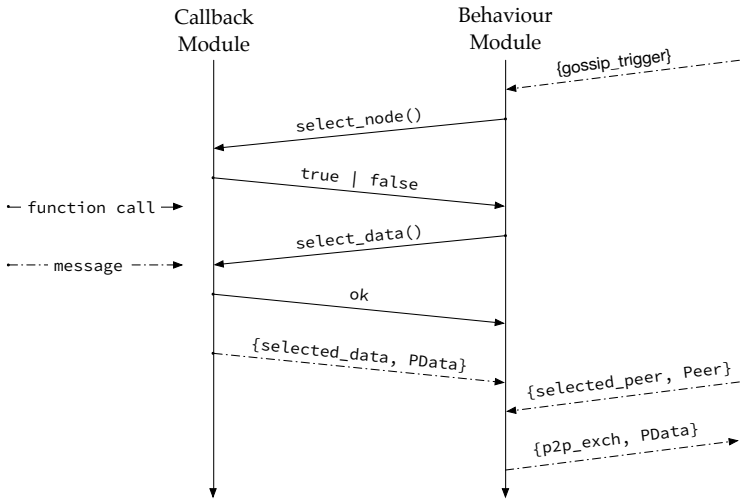


Figure 3: Prepare-Request Phase

**Prepare-Request Phase.** The *prepare-request phase* is shown in Figure 3. It consists of peer and data selection. The selection of the peer is usually managed by the framework. At the beginning of every cycle the behaviour module requests a peer from the Cyclon module of Scalaris, which is then used for the data exchange. The peer selection is governed by the `select_node()`

function[8]: returning `false` causes the behaviour module to handle the peer selection as described. Returning `true` causes the behaviour module to expect a `selected_peer` message with a peer to be used by for the exchange. How many peers are contracted for data exchanges every cycle depends on the `fanout()` config function.[9]

The selection of the exchange data is dependent on the specific gossiping task and therefore done by a callback module. It is initiated by a call to `select_data()`. When called with `select_data()`, the respective callback module has to initiate a `selected_data` message to the behaviour module, containing the selected exchange data. Both peer and data selection are initiated in immediate succession through periodical trigger messages, so they can run concurrently. When both data and peer are received by the behaviour module, a `p2p_exch` message with the exchange data is sent to the peer, that is to say to the gossip behaviour module of the peer.

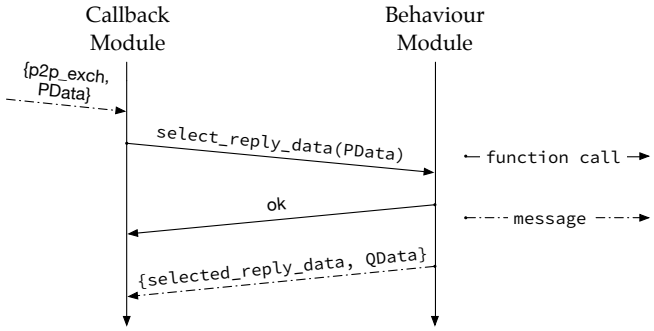


Figure 4: Prepare-Reply Phase

**Prepare-Reply Phase.** Upon receiving a `p2p_exch` message, a node enters the *prepare-reply phase* (see Figure 4) and is now in its passive role as responder. This phase is about the integration of the received data and the preparation of the reply data. Both of these tasks need to be handled by the callback module. The behaviour module passes the received data with a call to `select_reply_data(QData)` to the correspondent callback module, which merges the data with its own local data and prepares the reply data. The reply data is sent back to the behaviour module with a `selected_reply_data` message. The behaviour module then sends the reply data as a `p2p_exch_reply` message

[8] For the sake of clarity and readability, all signatures of functions given in this section may have been simplified, this is to say they have fewer arguments than the real functions in the code. The same is true for messages, which by convention in Scalaris always are n-tuples starting with a message tag, followed by an arbitrary number of arguments. Only the message tag or a simplified version of the message, lacking some arguments, is given. An overview of all callback functions is given in Appendix A.

[9] For the sake of readability, a fanout (see Section 2.1) of one is assumed for rest of the description.

back to the original requester.

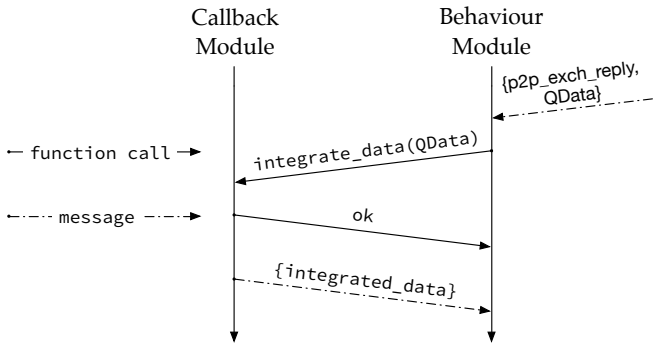


Figure 5: Integrate-Reply Phase

**Integrate-Reply Phase.** Figure 5 shows the interaction of the behaviour module and a callback module in the *integrate-reply phase*, which is triggered by a `p2p_exch_reply` message. Every `p2p_exch_reply` is the response to an *earlier* `p2p_exch` (although not necessarily to the *last* `p2p_exch` request, see Section 5.3). The `p2p_exch_reply` contains the reply data from the peer, which is passed to the correspondent callback module with a call to `integrate_data(QData)`. The callback module processes the received data and signals to the behaviour module the completion with an `integrated_data` message. On a conceptual level, a full cycle is finished at this point and the behaviour module counts cycles by counting the `integrated_data` messages. Due to the uncertainties of message delays and local clock drift it should be clear however, that this can only be an approximation. For instance, a new cycle could have been started[10] before the reply to the current request has been received (phase interleaving) and, respectively, replies from the other cycle could be "wrongly" counted as finishing the current cycle (cycle interleaving).

**Instantiation.** Many of the interactions conducted by the behaviour module are specific to a certain callback module. Therefore, all messages and function concerning a certain callback module need to identify with which callback module the message or call is associated. This is achieved by adding a tuple of the module name and an instance id to all those messages and calls. While the name would be enough to identify the module, adding the instance id allows for multiple instantiation of the same callback module by one behaviour module. This tuple of callback module and instance id is also used to store information specific to a certain callback module in the behaviour module's state (see Section 5.5 for more details).

[10] The starting of a new cycle is equivalent to receiving a `gossip_trigger` message.

## 5.3 Atomicity and Coupling

The framework deliberately does not give any atomicity guarantees. Guaranteeing atomicity (for instance for certain phases) on the level of the framework would need to be very general. For instance, one could guarantee the atomicity of the entire prepare-request phase. This would render the asynchronous communication between the behaviour module and the callback module superfluous and greatly deteriorate performance.

Furthermore, gossiping algorithms are expected to be robust and this usually implies weak atomicity requirements, e.g. if the data transformation operations are idempotent they do not need a fixed succession. If a gossiping algorithm has certain atomicity requirements they should be implemented at the level of the callback module, to keep the performance impact as minimal as possible.

The one exception is the `trigger_lock`: As long as a callback module has not answered the call to `select_data()` with a `selected_data` message, all new `gossip_trigger` messages are ignored. So the behaviour module guarantees the atomicity of the prepare-request phase of a specific callback module with regard to trigger messages for this callback module.[11] This prevents the filling of the message queue especially during startup and the starting of new rounds.

**Logical Coupling of Phases.** It is important to clarify the logical and temporal coupling introduced in Section 4. First, the temporal coupling between a prepare-request and an integrate-reply phase *within one node*. Because the framework does not give any atomicity guarantees, a new `p2p_exch` request can be sent to a peer before the current request has been answered. Therefore there can be no guarantee that a received `p2p_exch_reply` is the answer to the *last* `p2p_exch` request issued.

Second, because there is no logical and temporal coupling between the prepare-reply and the other phases *within one node*, a `p2p_exch` request can be received at any point in time and a callback module should be prepared to satisfy such a request (i.e. a call to `select_reply_data()`) independently of the other phases. If the gossip task to be implemented needs to impose restrictions on when a `select_reply_data()` call can be answered, this is to be implemented by the callback module. The callback module can, however, use the possibility to postpone calls to this end (by returning the special value `retry`). The behaviour module will then queue the request and retry later.

[11] To be perfectly clear: Saying that the atomicity is only guaranteed for trigger messages means that other messages *for the same callback module*, such as a `p2p_exch` request, can interleave with the prepare-request phase!

## 5.4 Rounds

Rounds are an important mechanism to deal with dynamic values and node failures at the level of callback modules(see Section 3.2, Section 3.5). The behaviour module offers rounds as a service to all callback modules. It is up to them to decide if they can and want to use that information.

Upon every received `p2p_exch` and `p2p_exch_reply` message, the behaviour module checks if the criteria for starting a new round are met. Three criteria are used, they need to be provided by the respective callback module (in form of "configuration" functions):

1. Convergence: Implements the new-round convergence criterion discussed in Section 3.5 and considered the actual convergence criterion.
   Function: `round_has_converged()->boolean()`.
2. Minimum number of cycles: Provides a lower bound for the duration of a round.
   Function: `min_cycles_per_round()->non_neg_integer()`
3. Maximum number of cycles: Provides a upper bound for the duration of a round.
   Function: `min_cycles_per_round()->non_neg_integer()`

**Leader Selection.** When the above criteria are a met, a new round can be started. The decision about starting a new round is made by one node, the leader, and this decision is then disseminated to the other nodes. The determination of the leader is done by using the higher layers of Scalaris, namely the key ranges of a node. The leader is defined as the node responsible for the hashed zero key. The behaviour module keeps track of the key ranges of its node by subscribing to the ring maintenance of Scalaris and uses this information to determine if a given node is the leader or not.

The leader then starts a new round by sending `new_round` messages to every peer he encounters with a round lesser than his own round counter (every `p2p_exch` and `p2p_exch_reply` message contains the round the message belongs to). Every node receiving a round greater than its own enters the new round and starts distributing the round information to every peer with a lesser round. With this simple gossip based dissemination scheme the round information can be distributed with logarithmic speed. Resetting of the round counter is not necessary, as Erlang uses arbitrary-sized integers.[12]

When a new round is entered (or started) by the behaviour module, the correspondent callback module is notified with a call to `notify_change( new_round)` so it can act accordingly. If the callback module in question does not need any handling of rounds it can simply ignore the call.

Entering a new round is not a synchronous event across all nodes, therefore `p2p_exch` and `p2p_exch_reply` requests from old rounds will reach the

[12] If every round would take one second (in practice its considerably longer), a 64bit counter would suffice $\approx$ 292 billion years (assuming signed integers in two's complement: $2^{63} - 1 = 9223372036854775807$ sec $\approx$ 292 billion years).

framework. This will happen during the propagation phase of a new round and can also happen later due to message delay. The framework detects, whether a requests is from an old round and passes it, specially marked, to the respective callback module. A callback module might keep track of the data from previous rounds, hence it might still be able to satisfy the request.

## 5.5 State of Behaviour Module

The behaviour module uses a simple key value store to store state information. The behaviour module uses the `pdb_beh` of Scalaris, which allows to switch the process database (pdb) implementation between `process dictionary` and `ets`[13]. In production, the `process dictionary` is preferable due to its better performance characteristics, while `ets` is easier to debug (it can by accessed from different processes). The state is kept in one flat table, entries that need to be kept for every callback module use a tuple of the keyword and the callback module[14] as key.

When a gossiping task is stopped, all the entries concerning the callback module are deleted from the behaviour module's state, except for `cb_status`, where a tombstone is set to handle requests for already stopped modules. The framework provides a `remove_all_tombstones()` function for cleanup purposes.

Table 1: State of `gossip.erl`

| **Key** | **Value** |
|---|---|
| `cb_modules` | `list of started callback modules` |
| `msg_queue` | `the message queue` |
| `range` | `the key range of the dht_node` |
| `status` | `uninit \| init` |
| `{reply_peer, Ref}` | `the requester from a p2p_exch message` |
| `{trigger_group, Interval}` | `list of callback modules` |
| `{cb_state, CBModule}` | `the state of a callback module` |
| `{cb_status, CBModule}` | `unstarted \| started \| tombstone` |
| `{cycles, CBModule}` | `the current cycle of a callback module` |
| `{trigger_lock, CBModule}` | `locked \| free` |
| `{exch_data, CBModule}` | `{Peer, Data} \| {undefined, Data} \| ...` |
| `{round, CBModule}` | `the current of a callback module` |

## 5.6 Usage of the Framework

A gossiping task, i.e. a callback module, can be started with the function `start_gossip_task(CBModule, Args)` and stopped with the function `stop_gossip_task(CBModule)`. The parameter `CBModule` refers to the tuple

[13] The `process dictionary` and `ets` are both simple key value stores provided by Erlang. The `process dictionary` is local to one process, while `ets` can be shared between processes.

[14] The callback module is actually identified by a tuple of the module name and an instance id, see Section 5.2.

of module name and instance id (see Section 5.2) identifying the callback module. In case of `start_gossip_task()` it is also possible to provide only a module name, an instance id is then generated. The parameter `Args` represents a list of arguments passed to the `init()` function of the callback module.

To query the framework for values from a gossiping task, one has to send a message of the form `{get_values_best, CBModule, SourcePid}` to the framework. The reply to the query is sent back to the requester identified by `SourcePid` with the message `{gossip_get_values_best_response, BestValues}`. `BestValues` is a data structure holding the values, specified by the respective callback module.

Internally, the query is forwarded and answered by the callback module specified by `CBModule`. What "best values" means is dependent on the callback module, but in general it refers to the idea of properly converged values in a gossip based aggregation protocol (see Section 3.5).

## 5.7 Evaluation of the Framework

In the previous sections, the main design decisions have been presented. In this section, the framework is evaluated against the aspects of robustness introduced in Section 2.4.

**Asynchronous Communication.** As discussed in Section 5.1, all communication in Scalaris in general and the framework in particular is organised asynchronously. The framework places no restrictions upon the timing of message delivery and computations. With regard to asynchronicity, the framework evidently fulfils the requirements.

**Message Loss.** As described in Section 2.4, gossiping algorithms have to work even under conditions of message loss. But message loss can affect the accuracy of the given algorithm, especially in the case of aggregation, where the accuracy of the aggregation result is concerned (see Section 3.4). Consequently, it is important to avoid message loss as much as possible. The gossiping framework deploys several mechanism to this end.

The only messages considered in this discussion are `p2p_exch` and `p2p_exch_reply` messages. Within a gossiping operation, these are the only messages sent to other nodes, all other messages such as messages to the Cyclon process or from a callback module are local messages between different processes of one node (i.e. one VM), which are considered even more reliable than TCP.

The most important measure for avoiding message loss is the use of TCP in Scalaris, which achieves reliable point-to-point connections. There are still possibilities for message loss, e.g. when a message is delivered to the communication layer of Scalaris and the node then crashes or when messages are routed through other nodes that crash. As gossiping only uses point-to-

point connections, the latter does not apply to gossiping[15] and the former usually amounts to a node failure.

Failure of nodes, however, can cause message loss. The framework uses Cyclon for peer sampling, so references to crashed nodes remain in the system for a short period of time and a node might try to initiate a data exchange with a failed node. In this case, message delivery fails. Scalaris is able to detect this and notify the sender with a `send_error` message about the failed message delivery. When the framework receives such a `send_error` message it informs the correspondent callback module with a call to `notify_change(exch_failure)` along with the originally sent message. If the gossiping task has a way to deal with failed messages it can use this callback to act accordingly.

Message loss may also occur during the startup of the framework, when an already initiated gossip process tries to exchange data with a not yet initiated one. Such a message is handled by the `on_inactive()` message handler of the behaviour module, which uses the same mechanism of send errors described above to notify the sender that a data exchange is not possible at the moment.

This mechanism can also be triggered manually by a callback module by returning `send_back` to a `select_reply_data()` or `integrate_data()` call, which causes the behaviour module to produce a send error for the corresponding `p2p_exch` and `p2p_exch_reply` messages.

In summary, although complete reliability in message delivery cannot be guaranteed, the framework provides useful mechanism to reduce message loss and to reduce the impact of message loss.

**Node Failures.** For the selection of peers the framework relies on Scalaris' implementation of Cyclon (see Section 2.2.1) as peer sampling service, so it depends largely on Cyclon's properties. With regard to node failures the most important aspect is how fast a failed node disappears from the set of known nodes, so that the behaviour module stops trying to initiate data exchanges with these. As described in Section 2.2.1 it takes at most a number of cycles equal to the cache size (the number of nodes known to the Cyclon process) to detect dead nodes. Within this time frame a failed node can cause message loss or rather `send_erros` messages, as described in the last section.

Note that it is not necessary do have any special precautions for a failure of the leader. If the node responsible for the zero key fails, the key ranges are reassigned and the framework is notified about any changes by the ring maintenance.

The second important aspect to consider is whether a node failure produces any "garbage", that is to say if any information about a crashed node remains

[15] Except in the case of the bulkowner messages used in the global startup and shutdown of the behaviour module and gossip tasks. Bulkowner messages are a mechanism of DHT based group communication used in Scalaris, for details see [16], ch. 5)

in the state of the behaviour module. There are only two incidents where information about a peer is stored: First, in the prepare-request phase. When a peer is requested from Cyclon it might be stored in the state of the behaviour module until the correspondent callback module selected the exchange data. This is repeated at the beginning of every cycle, so any reference to a node only remains until the next cycle. The second time information about a peer is stored is during the integrate-reply phase, where the behaviour module stores the requester in its state before the `select_reply_data()` call. Upon the `selected_reply_data` message, it uses this information for the reply and deletes it from its state. The deletion is only dependent on process local messages, when the behaviour module tries to send the reply, the information about the peer is already deleted from the state, hence no information remains, even if the peer has failed.

Note that there is no need to consider the failure of a callback module with respect to garbage. As described above (see page 27), the behaviour module and the callback module execute in the same process, so it is assumed that they can not fail independently of each other.

With regard to node failures, it can be concluded that the framework ensures fast removal of references to dead nodes and that the failing of a node does not produce residual "garbage". Additionally the framework offers rounds as a means to periodically restart a gossiping protocol, which will be discussed in the evaluation of the `gossip_load` module (see Section 6.5).

**Testing.** Three kinds of tests have been performed on the behaviour module: First, the consistency of the type specifications has been verified through Dialyzer, the "Discrepancy Analyzer for Erlang" programs, a static analysis tool provided with Erlang.

Second, dynamic type checking has been performed. Using Scalaris' own property testing tool `tester`, the functions of the framework are tested with a wide range of automatically generated input values. The "For Testing" section at the end of the behaviour module provide several functions for `tester`, such as value creator, type checking and feeder functions. The value creator and type checking functions are used for the creation and checking of values not natively supported by `tester`. The feeder functions are used to restrict the input values to certain functions. The dynamic type checking is implemented in the `type_check_SUITE`.

Third, integration tests of the behaviour module and the `gossip_load` callback module have been conducted, see Section 6.5 for details.

# 6 Gossiping Load Information in Scalaris

The `gossip_load` module is an example implementation of a callback module. It implements the symmetric push-sum protocol with the extensions discussed in Section 3. The algorithm is used to compute aggregates of the load information, which is measured as the count of items currently in a node's key range.

The aggregation of load information is used in Scalaris for two purposes: First, for passive load balancing. When a node joins, the gossiped load information is used to decide where to place the new node. The node will be placed so that the standard deviation of the load is reduced as much as possible. Second, gossiping is used for system monitoring. The local estimates can be viewed for example in the Web Interface of every Scalaris node.

Different metrics are computed on the load information: The *average load* gives the arithmetic mean of the load information of all nodes. The *maximum* and *minimum load* are provided and the *standard deviation* gives information about the variation of values. Two *size estimates* (i.e. estimates of the number of all nodes) are given: First, based on leader election, i.e. the sum $s_i$ is set to $s_i = 1$ at all nodes and the weight $w_i$ is set to $w_l = 1$ at the leader and $w = 0$ at all other nodes (see Section 3.2). The size estimate is then calculated as $s/w$. Second, based on the key ranges of the nodes. The average of the key ranges of all nodes is aggregated and the size estimate is calculated as $\text{address space of keys}/\text{average key range}$.

The histogram computation introduced in Section 3.3 is used to compute a histogram of the load per key range. The load is measured as number of items per node. The key address space is divided into a number of predefined buckets and at every node the contribution of this node to one or multiple buckets is counted. The thus initialised histograms are then aggregated using the algorithm provided in Algorithm 5. Figure 6 shows an example histogram, calculated on the same setup as described in Section 6.5.

The `gossip_load` module is initialised during the startup of the gossiping framework, continuously aggregating load information in the background. Additionally, it is possible to start instances of the `gossip_load` module for the purpose of computing different sizes histograms (see Section 6.4).

## 6.1 Interaction with the Behaviour Module

**Prepare-Request Phase.** When called with `select_data()`[16] for the first time (or for the first time in a new round), the `gossip_load` module requests the current load information from its local node. Once initiated, it prepares the load data for the data exchange, which basically divides all average values by two. The exchange data is then sent back to the behaviour module in a

[16] See Appendix A for an overview of all callback functions.

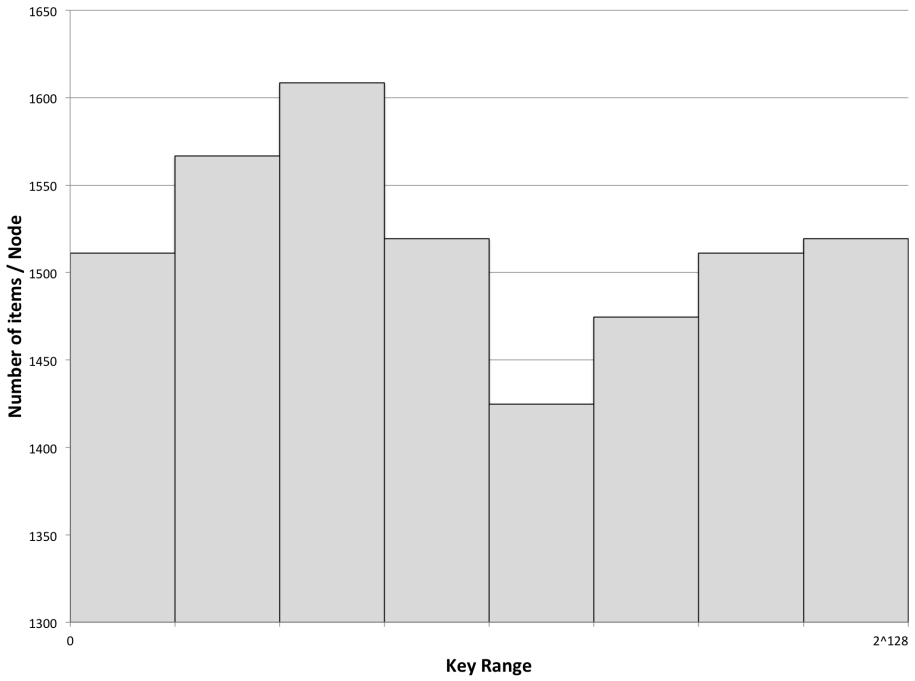


Figure 6: Load per Key Range

`selected_data` message. The `gossip_load` module relies on the behaviour module to select a peer, thus it returns `false` to the `select_node()` call.

**Integrate-Reply Phase.** Upon the `integrate_data(QData)` call, the `gossip_load` module needs to check its status. If in `uninit`, i.e. when it is requesting load information from the `dht_node` process, the call returns with `retry`, instructing the behaviour module to repeat the call later. The next thing to check is the round. The `gossip_load` module always keeps data from one earlier round (see Section 6.2 for details), which can be used to satisfy requests for old rounds. If the request is from the current round or a valid previous round[17], then the received data is merged with the own current or previous data respectively. In the case of average values, the values and weights are added up (see Algorithm 4), for the minimum and maximum the respective values are selected. The histograms are merged according to the merge procedure given in Listing 1. The `gossip_load` module then concludes the phase with an `integrated_data` message to the behaviour module.

[17] A previous round is valid, if the round counter of the received exchange data and the round counter of the previous round match. Only in this case can the `gossip_load` module satisfy the request. This is not always the case, e.g. if a previous round has not converged or during the startup of a later joining node.

**Prepare-Reply Phase.** The *prepare-reply* phase is initiated by a call of the behaviour module to `select_reply_data(PData)`, which checks the round information and merges back the received data in the same way as described for `integrate_data()`. Next, the reply data is selected analogously to the exchange data selection in `select_data()`. The phase is finished by sending the reply data to the behaviour module in a `selected_reply_data` message.

## 6.2 Periodic Restarting

As described in Section 3.5, rounds are an important mechanism to improve the robustness of the protocol through periodic restarting.

The main criterion for starting a new round is the convergence of the current local estimates of the gossiped values. On every merge, i.e. upon every call to `select_reply_data()` or `integrate_data()`, the estimates before and after the merge are compared against a convergence epsilon. If the difference is less than the convergence epsilon, a counter is incremented. When meeting a predefined target value (`convergence_count_new_round`), a call to `round_has_converged()` will return `true` and the behaviour module will be allowed to start a new round. Note that the convergence count can be incremented multiple times during each cycle, as there are on average two merges per cycle.

Additionally, the `min_cycles_per_round()` and `max_cycles_per_round()` functions provide simple numerical upper and lower cycle bounds for starting a new round.

The `gossip_load` module always keeps the data from one earlier round, implementing the optimisations discussed in Section 3.5. The data from the previous round might be needed to satisfy queries for the aggregation results, i.e. `get_best_values` requests, if the current round has not converged yet. The data from the previous round is also used to potentially satisfy `p2p_exch` and `p2p_exch_reply` requests for old rounds, as described in Section 5.4. When a new round starts, the previous load data is only replaced with the load data from the current round, if the current round has converged. Note, that in all non-leader nodes, entering a new round is not necessarily equivalent to a converged current round. This can be due to difference in the convergence speed at different nodes because of the probabilistic nature of gossiping, but also happens in nodes currently joining. To avoid that already slight differences in the convergence speed lead to not replacing the old data, the best-values criterion (`convergence_count_best_values`) is used instead of the new-round criterion (`convergence_count_new_round`) when deciding on the convergence of rounds in non-leader nodes.

## 6.3 State of gossip_load

The state of the `gossip_load` module is maintained in `ets` tables, one for the current round and one for the previous round (because multiple tables are used it is not possible to use the `process dictionary`). When a `gossip_load` instance is stopped, both tables are deleted. Table 2 gives an overview about a `gossip_load` state table.

Table 2: State of `gossip_load`

| Key | Value |
|---|---|
| convergence_count | convergence counter |
| instance | the instance id |
| leader | true \| false |
| load_data | #load_data record |
| merged | the merged counter |
| no_of_buckets | the number of histogram buckets |
| prev_state | the table id of the previous state |
| range | the key range of the dht_node |
| request | true \| false |
| requester | the pid of the requester |
| round | the current round |
| status | uninit \| init |

## 6.4 Usage of `gossip_load`

The `gossip_load` module is supposed to be used through the API of the framework, see Section 5.6. The API functions need to identify the `gossip_load` instance. The standard `gossip_load` instance, which is running continuously in the background, is identified by `{gossip_load, default}`.

Additionally, the `gossip_load` module allows for the computation of histograms on a *per request basis*. With `request_histogram(Size, SourcePid)`, a histogram with `Size` number of buckets can be requested. The histogram will be sent back to `SourcePid`, as soon as all the values have properly converged.

## 6.5 Evaluation of `gossip_load`

**Asynchronous Communication.** The symmetric push-sum protocol was, amongst other things, chosen for is robustness against message interleaving in asynchronous settings. The implementation of the `gossip_load` module relies on this property to fulfil the asynchronicity requirements.

**Message Loss.** Losing messages, especially in the early phases of a round when the values haven't converged yet, affects the correctness of the results. The `gossip_load` module uses the notifications from the behaviour module about failed data exchanges. When the module is called with `notify_change(exch_failure, Data)` it merges the data with its own data the

same way it merges data from a peer in the `select_reply_data()` and `integrate_reply()` calls. This prevents mass conservation violations and thus ensures, that the data still converges to the correct global value.

The most common case for this mechanism to take effect is during the starting of a `gossip_load` instance. As there can be no guarantees that the startup happens synchronously at all nodes, one node might already request a `p2p_exch` while the `gossip_load` instance at the contacted node is still during startup.

If the exchange failure is due to a crashing node, this mechanism prevents at least some weight loss at the other nodes, although it cannot prevent the incorrectness introduced by the node failure itself (see section 3.4).

The deployed mechanism provides a significant improvement of the bare algorithm from section 3.2 and also of the current implementation of the load monitoring in Scalaris. It can not, however, achieve complete robustness against message loss.

**Node Failure.** As discussed in Section 3.5, the information from a failed node keeps impacting the accuracy of the aggregation results for no longer than two rounds. The `gossip_load` module uses the discussed optimisations in the round handling (see Section 6.2) , returning values from the last round to a `get_values_best` request if the current round has not converged sufficiently according to the best-values criterion (`convergence_count_best_values`). This means that in practice, a failed node stops impacting the load information even before the next round finishes.

Periodic restarting presents a simple and effective way of dealing with node failures. This concept was already deployed in the current implementation of load information gossiping. The contribution of the `gossip_load` implementation with regard to rounds lies first in the improvement of the round handling, e.g. by adding the the possibility to satisfy requests from old rounds. Second, it lies in the demonstration of the practical applicability of periodic restarting to the (symmetric) push-sum protocol.

**Convergence Speed.** As stated in the introduction, an experimental evaluation is out of the scope of this work. However, one experiment has been conducted for demonstration purposes. The setup was the following: 256 Scalaris nodes have been started on one machine and 100000 items have been written. The time interval for the cycles needed to be set to an unusual high value of 60s, normally an interval of 1s is used in Scalaris. This was necessary to allow for a quasi-parallel execution on only one machine, shorter time intervals would have led to extrem asynchrony in the cycles on different nodes.[18] The local estimates have then been extracted at the end of every cycle at all nodes. Figure 7 shows convergence speed as the reduction of

[18] Note that asynchrony in the cycle count is not a problem per (see Section 3.4). The objective here was to demonstrate the convergence speed as accurately as possible, not the robustness.

standard deviation of all local estimates as a function of the cycles. The graph clearly shows that the convergence speed in the this experiment was well within expected bounds of speed logarithmic to the network size given in Section 3.2.

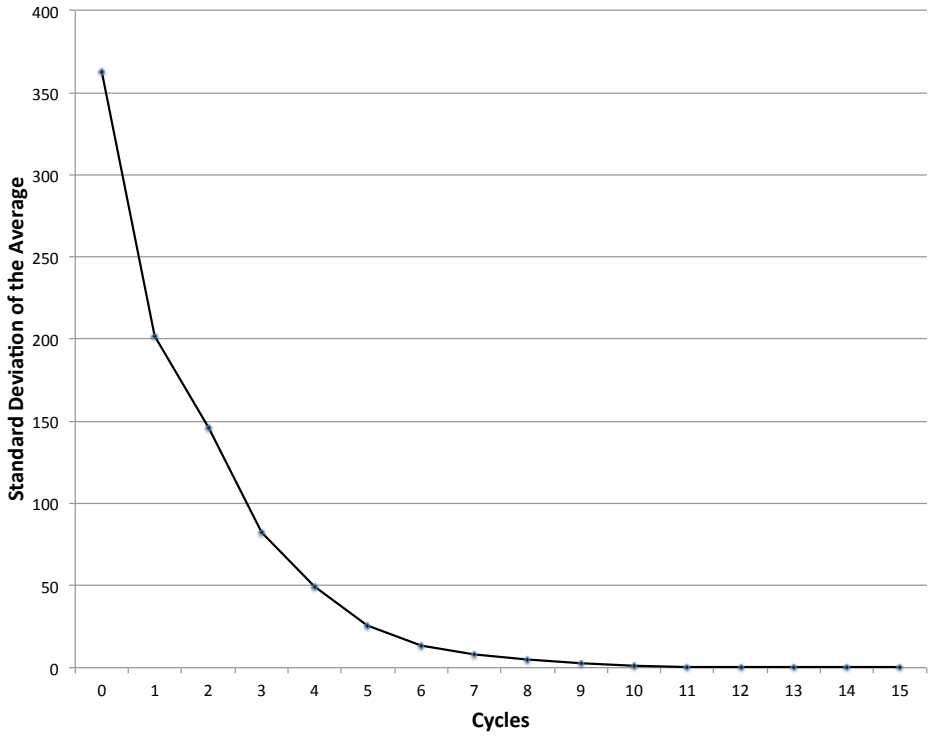


Figure 7: Convergence Speed of Average Function

**Testing.** The same static and dynamic consistency and type testing as described in Section 5.7 has been performed on the `gossip_load` module.

Additionally, `gossip_SUITE` provides integration tests of the `gossip_load` module with the behaviour module. They test the correct working of the load information aggregation as well as the histogram computation. The basic idea is to start a test setup of Scalaris, write some entries and query the local load information from all nodes. With the the knowledge of the all values the true values for the load metrics and the histogram are computed. Gossiping is then started and the results from the gossip based aggregation are compared against the true values.

# 7 Conclusion

Gossiping has been shown to be a useful answer to the problems of information dissemination and aggregate computation in peer-to-peer systems. The main advantages lie in the robustness, the scalability and the uniform load distribution while still being reasonably fast.

The symmetric push-sum protocol was chosen as an aggregation algorithm based on the analysis of different algorithms. It was extended to allow the computation of simple histograms and optimised for the use in dynamic conditions. The latter was achieved by improving the concept of periodic restarting and applying it to the symmetric push-sum protocol.

Based on the analysis of the different classes of gossiping protocols, a general structure of a gossiping operation was conceptualised. This was used to implement a generic gossiping framework. The framework facilitates the implementation of arbitrary gossip based information dissemination and aggregation algorithms. As a communication scheme for the gossiping push-pull was chosen because of the superior performance characteristics.

The evaluation of the framework showed, that it is well equipped to work in asynchronous environments, provides useful mechanisms to reduce and deal with message loss and that it removes references to dead nodes quickly.

The functionality of the framework was demonstrated by implementing load monitoring as an example callback module. The evaluation showed that the deployed mechanisms provide a significant improvement with regard to message loss and that the periodic restarting is an effective way of dealing with node failures.

Both the framework and the load monitoring implementation have already been incorporated into the production code of Scalaris.

**Future Work.** As mentioned in the introduction, Scalaris already uses several gossiping protocols. The gossiping framework provides a basis for unifying the use of gossiping protocols. To this end, the existing gossiping protocols need to be reimplemented on the basis of the framework. This includes T-Man for topology construction and maintenance, the data center detection and computation of Vivaldi Coordinates.

The framework also provides a basis for extending the use of gossiping protocols in Scalaris. In the implementation of active load balancing, which is currently in development, global aggregates of different additional load metrics such as execution time of transactions, CPU utilisation and memory usage are needed as input to the load balancing algorithms. They can be implemented using the framework.

Another useful extension would be the computation of value distribution characterisations like quantiles and histograms. This can be implemented as gossip based aggregation using q-digests as presented in [41] and experimentally evaluated in [18]. Q-digest are an approximative approach to fully

decentralised computation of quantiles, ranks or histograms, using tree-like summaries that can be computed locally and merged upon exchange.

For the aggregation of load information the use of approaches that allow for continuous aggregation of changing values should be considered. In the opinion of the author, the most interesting approach presented so far is Flow Updating [24, 25](also cf. [2]). Besides allowing for continuous aggregation, Flow Updating also promises resilience against message loss and, in the extended version[25], against node failures. Flow updating assumes a relatively static partial view, which is at odds with the dynamic partial view provided by Cyclon. This could be remedied by the use of a second partial view (this idea is inspired by [33]). This second view would be filled from the Cyclon cache in a reactive fashion, i.e. when a node fails its reference is replaced with a reference from the first partial view. This would result in a relatively static view, suitable for Flow Updating, without losing the robustness of Cyclon.

# Appendices

# A Overview of Callback Functions

## A.1 Type Definitions

`State :: any()`

The `State` of the callback module. No restrictions are based on the data type, it depends entirely on the callback modules.

`Data, QData, PData :: any()`

The data to be exchanged in the gossip operations between peers. No restrictions are based on the data type, it depends entirely on the callback modules.

`RoundStatus :: current_round | old_round`

The `RoundStatus` indicates whether a request is from the current or a previous round.

`Round :: non_neg_integer()`

The `Round` a request belongs to.

## A.2 Startup and Shutdown

```
init(Instance) -> {ok, State}.
init(Instance, Arg1) -> {ok, State}.
init(Instance, Arg1, Arg2) -> {ok, State}.
    Instance :: { CBModule::module(),
                  InstanceId:: atom() | uid:global_uid() }.
```

Called by the behaviour module upon startup.
Init function with additional arguments.
The `Instance` is the tuple of module name and instance id described in Section 5.2. Usually used to initialise the state of the callback module.

```
shutdown(State) -> {ok, state_deleted}.
```

Called by the behaviour module upon `stop_gossip_task(CBModule)`.
It should be the opposite of `init()` and do any necessary cleaning up.

## A.3 Gossip Operation

```
select_node(State) -> {boolean(), State}.
```

Called by the behaviour module at the beginning of every cycle.
Should return `true`, if the peer selection is to be done by behaviour module, `false` otherwise. If `false` is returned, the behaviour module expects a `selected_peer` message.

```
select_data(State) -> {ok, State}.
```

Called by the behaviour module at the beginning of a cycle.
The callback module has to select the exchange data to be sent to the peer.
The exchange data has to be sent back to the behaviour module as a message of the form `{selected_data, Instance, ExchangeData}`.

```
select_reply_data(PData, Ref, RoundStatus, Round, State) ->
        {discard_msg | ok | retry | send_back, State}.
    Ref :: pos_integer()
```

Called by the behaviour module upon a `p2p_exch` message.
Passes the `PData` from a `p2p_exch` request to the callback module. The callback module has to select the exchange data to be sent to the peer.
The `Ref` is used by the behaviour module to identify the request.
The `RoundStatus` and `Round` information can be used for special handling of messages from previous rounds.
The selected reply data is to be sent back to the behaviour module as a message of the form `{selected_reply_data, Instance, QData, Ref, Round}`.

```
integrate_data(QData, RoundStatus, Round, State) ->
        {discard_msg | ok | retry | send_back, State}.
    Round::non_neg_integer()
```

Called by the behaviour module upon a `p2p_exch` message.
Passes the `QData` from a `p2p_exch_reply` to the callback module.
Upon finishing the processing of the data, a message of the form `{integrated_data, Instance, RoundStatus}` is to be sent to the behaviour module.

```
handle_msg(Message, State) ->
        { discard_msg | ok | retry | send_back , State}.
    Message::comm:message()
```

Called by the behaviour module upon messages of the form `{cb_reply, CBModule, Msg}`.
Passes the message `Msg` to the callback module, used to handle messages for the callback module.

```
get_values_best(State) -> BestValues.
    BestValues :: any()
```

Called by the behaviour module upon `{get_values_best}` messages. The callback module has to return the "best" values. What the "best" values are depends on the gossip algorithm in question, but in general it refers to the idea of properly converged values in a gossip based aggregation protocol (see Section 3.5). The data type of `BaseValues` depends on the concrete callback module.

```
get_values_all(State) -> AllValues.
    AllValues :: any()
```

Called by the behaviour module upon `{get_values_all}` messages. The callback module has to return "all" values. What "all" values are depends on the gossip algorithm in question, but in general it refers to the idea that a callback module might keep values from multiple rounds (see Section 3.5). The data type of `AllValues` depends on the concrete callback module.

```
web_debug_info(State) ->
        {KeyValueList::[{Key::string(), Value::string()},...],
            State}.
```

Called by the behaviour module upon `{web_debug_info}` messages. The callback module has to return debugging infos, to be displayed in the Scalaris Web Debug Interface.

## A.4 Config Functions

```
fanout() -> pos_integer().
```

The fanout, i.e. the number of peers contacted per cycle (see Section 2.1).

```
trigger_interval() -> pos_integer().
```

The time interval in ms after which a new cycle is triggered.

```
min_cycles_per_round() -> non_neg_integer().
```

The minimum number of cycles per round.

```
max_cycles_per_round() -> pos_integer().
```

The maximum number of cycles per round.

```
round_has_converged(State) -> {boolean(), State}.
```

Implements the new-round convergence criterion (see Section 3.5). Returns `true` if the round has converged, `false` otherwise.

## A.5 Notifications

```
notify_change(new_round, NewRound, State) -> {ok, state()}.
    NewRound :: pos_integer()
```

Notifies the the callback module about the beginning of round `NewRound`.

```
notify_change(leader, {MsgTag, NewRange}, State) -> {ok, state()}.
    MsgTag :: is_leader | no_leader
    NewRange :: intervals:interval()
```

Notifies the the callback module about a change in the key range of the node. The `MsgTag` indicates whether the node is a leader or not, the `NewRange` is the new key range of the node.

```
notify_change(exch_failure, {MsgTag, Data, Round}, State) -> {ok,
    state()}.
    MsgTag :: p2p_exch | p2p_exch_reply
    Round :: pos_integer()
```

Notifies the the callback module about a failed message delivery, including exchange `Data` and `Round` from the original message.

# B Source Code

## B.1 Google Code

Scalaris is hosted at Google Code, the official project page can be found at:

`https://code.google.com/p/scalaris/`

The relevant files and the links to the final version[19]:

- `src/gossio.erl`: the behaviour module, main part of the framework:
  `https://code.google.com/p/scalaris/source/browse/trunk/src/gossip.erl?spec=svn6108&r=6108`
- `src/gossip_beh.erl`: the behaviour (interface) for the callback modules:
  `https://code.google.com/p/scalaris/source/browse/trunk/src/gossip_beh.erl?spec=svn6108&r=6108`
- `src/gossip_load`: example callback module, implementing load information aggregation:
  `https://code.google.com/p/scalaris/source/browse/trunk/src/gossip_load.erl?spec=svn6108&r=6108`
- `test/gossip_SUITE.erl`: Test suite, providing integration tests:
  `https://code.google.com/p/scalaris/source/browse/trunk/test/gossip_SUITE.erl?spec=svn6108&r=6108`

## B.2 Github

A version with nicer formatting, including syntax highlighting, can be at the authors Github repository:

`https://github.com/jvf/scalaris`

The relevant files and the links to the final version:

- `src/gossio.erl`: the behaviour module, main part of the framework:
  `https://github.com/jvf/scalaris/blob/gossip_thesis_submission/src/gossip.erl`
- `src/gossip_beh.erl`: the behaviour (interface) for the callback modules:
  `https://github.com/jvf/scalaris/blob/gossip_thesis_submission/src/gossip_beh.erl`
- `src/gossip_load`: example callback module, implementing load information aggregation:
  `https://github.com/jvf/scalaris/blob/master/src/gossip_load.erl`
- `test/gossip_SUITE.erl`: Test suite, providing integration tests:
  `https://github.com/jvf/scalaris/blob/gossip_thesis_submission/test/gossip_SUITE.erl`

[19] The final version is the latest revision within the deadline of the thesis.

# Bibliography

[1] Ericsson AB. *OTP Design Principles*. 2013. URL: `http://www.erlang.org/doc/design_principles/des_princ.html`.

[2] Paulo Sérgio Almeida et al. "Fault-Tolerant Aggregation: Flow-updating Meets Mass-distribution". In: *Proceedings of the 15th International Conference on Principles of Distributed Systems*. OPODIS'11. Toulouse, France: Springer-Verlag, 2011, pp. 513–527.

[3] Joe Armstrong. *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf, 2013.

[4] Nicola Bicocchi, Marco Mamei and Franco Zambonelli. "Handling dynamics in gossip-based aggregation schemes". In: *Computers and Communications, 2009. ISCC 2009. IEEE Symposium on*. IEEE. 2009, pp. 380–385.

[5] Ken Birman. "The Promise, and Limitations, of Gossip Protocols". In: *SIGOPS Oper. Syst. Rev.* 41.5 (Oct. 2007), pp. 8–13.

[6] Francesco Blasa et al. "Symmetric Push-Sum Protocol for decentralised aggregation". In: *Proceedings of AP2PS 2011, the Third International Conference on Advances in P2P Systems*. IARIA, 2011, pp. 27–32.

[7] Stephen Boyd et al. "Randomized gossip algorithms". In: *IEEE/ACM Trans. Netw.* 14.SI (June 2006), pp. 2508–2530.

[8] Miguel Castro et al. "SCRIBE: A large-scale and decentralized application-level multicast infrastructure". In: *IEEE Journal on Selected Areas in Communications (JSAC* 20 (2002), p. 2002.

[9] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, 2009.

[10] Cassandra Community. *Anti-entropy Overview*. 2014. URL: `http://wiki.apache.org/cassandra/ArchitectureAntiEntropy`.

[11] Frank Dabek et al. "Vivaldi: a decentralized network coordinate system". In: *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*. SIGCOMM '04. Portland, Oregon, USA: ACM, 2004, pp. 15–26.

[12] Mads Dam and Rolf Stadler. "A generic protocol for network state aggregation". In: *In Proc. Radiovetenskap och Kommunikation (RVK*. 2005, pp. 14–16.

[13] Alan Demers et al. "Epidemic algorithms for replicated database maintenance". In: *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. PODC '87. Vancouver, British Columbia, Canada: ACM, 1987, pp. 1–12.

[14] Stefan Edlich et al. "NoSQL". In: (2011).

[15] P.T. Eugster et al. "Epidemic information dissemination in distributed systems". In: *Computer* 37.5 (2004), pp. 60–67.

[16] Ali Ghodsi. "Distributed $k$-ary System: Algorithms for Distributed Hash Tables". PhD Dissertation. Stockholm, Sweden: KTH—Royal Institute of Technology, Oct. 2006.

[17] Carl Hewitt, Peter Bishop and Richard Steiger. "A universal modular actor formalism for artificial intelligence". In: *Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc. 1973, pp. 235–245.

[18] Marie Hoffmann. "Approximate Algorithms for Distributed Systems". MA thesis. Freie Universität Berlin, 2013.

[19] Basho Technologies, Inc. *Riak Glossary*. 2014. URL: `http://docs.basho.com/riak/latest/theory/concepts/glossary/#Active-Anti-Entropy-AAE-`.

[20] Márk Jelasity, Alberto Montresor and Ozalp Babaoglu. "Gossip-based aggregation in large dynamic networks". In: *ACM Trans. Comput. Syst.* 23.3 (Aug. 2005), pp. 219–252.

[21] Márk Jelasity et al. "Gossip-based Peer Sampling". In: *ACM Trans. Comput. Syst.* 25.3 (Aug. 2007).

[22] Márk Jelasity and Ozalp Babaoglu. "T-Man: Gossip-Based Overlay Topology Management". In: *Engineering Self-Organising Systems*. Ed. by SvenA. Brueckner et al. Vol. 3910. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 1–15.

[23] Paulo Jesus, Carlos Baquero and Paulo Sérgio Almeida. "Dependability in Aggregation by Averaging". In: *CoRR* abs/1011.6596 (2010).

[24] Paulo Jesus, Carlos Baquero and Paulo Sérgio Almeida. "Fault-Tolerant Aggregation by Flow Updating". In: *Proceedings of the 9th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*. DAIS '09. Lisbon, Portugal: Springer-Verlag, 2009, pp. 73–86.

[25] Paulo Jesus, Carlos Baquero and Paulo Sergio Almeida. "Fault-Tolerant Aggregation for Dynamic Networks". In: *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*. SRDS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 37–43.

[26] R. Karp et al. "Randomized rumor spreading". In: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. FOCS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 565–.

[27] Srinivas Kashyap et al. "Efficient gossip-based aggregate computation". In: *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. PODS '06. Chicago, IL, USA: ACM, 2006, pp. 308–317.

[28] David Kempe, Alin Dobra and Johannes Gehrke. "Gossip-Based Computation of Aggregate Information". In: *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*. FOCS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 482–.

[29] A. Kemper and A. Eickler. *Datenbanksysteme: Eine Einführung*. Oldenbourg Wissenschaftsverlag, 2011.

[30] Anne-Marie Kermarrec, Laurent Massoulié and Ayalvadi J. Ganesh. "Probabilistic Reliable Dissemination in Large-Scale Systems". In: *IEEE Trans. Parallel Distrib. Syst.* 14.3 (Mar. 2003), pp. 248–258.

[31] Anne-Marie Kermarrec and Maarten van Steen. "Gossiping in distributed systems". In: *SIGOPS Oper. Syst. Rev.* 41.5 (Oct. 2007), pp. 2–7.

[32] João Carlos Antunes Leitao et al. "X-BOT: A Protocol for Resilient Optimization of Unstructured Overlays". In: *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*. SRDS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 236–245.

[33] Joao Leitao, José Pereira and Luís Rodrigues. "HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast". In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 419–429.

[34] Martin Logan, Eric Merritt and Richard Carlsson. *Erlang and OTP in Action*. Manning, Nov. 2010.

[35] Laurent Massoulié et al. "Peer Counting and Sampling in Overlay Networks: Random Walk Methods". In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*. PODC '06. Denver, Colorado, USA: ACM, 2006, pp. 123–132.

[36] Alberto Gonzalez Prieto and Rolf Stadler. "A-GAP: An adaptive protocol for continuous network monitoring with accuracy objectives". In: *Network and Service Management, IEEE Transactions on* 4.1 (2007), pp. 2–12.

[37] Antony Rowstron and Peter Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems". In: *Middleware 2001*. Springer. 2001, pp. 329–350.

[38] Thorsten Schütt, Florian Schintke and Alexander Reinefeld. "Scalaris: Reliable Transactional P2P Key/Value Store". In: *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*. ERLANG '08. Victoria, BC, Canada: ACM, 2008, pp. 41–48.

[39] T. Schütt et al. "Gossip-based topology inference for efficient overlay mapping on data centers". In: *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*. 2009, pp. 147–150.

[40] T. Schütt et al. “Self-Adaptation in Large-Scale Systems: A Study on Structured Overlays across Multiple Datacenters”. In: *Self-Adaptive and Self-Organizing Systems Workshop (SASOW), 2010 Fourth IEEE International Conference on*. 2010, pp. 224–228.

[41] Nisheeth Shrivastava et al. “Medians and beyond: new aggregation techniques for sensor networks”. In: *Proceedings of the 2nd international conference on Embedded networked sensor systems*. SenSys ’04. Baltimore, MD, USA: ACM, 2004, pp. 239–249.

[42] Angelos Stavrou, Dan Rubenstein and Sambit Sahu. “A Lightweight, Robust P2P System to Handle Flash Crowds”. In: *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS (JSAC*. 2004, pp. 6–17.

[43] Ion Stoica et al. “Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications”. In: *IEEE/ACM Trans. Netw.* 11.1 (Feb. 2003), pp. 17–32.

[44] A.S. Tanenbaum and M. van Steen. *Verteilte Systeme: Grundlagen und Paradigmen*. I : Informatik. Pearson Education Deutschland GmbH, 2003.

[45] Demerew Ketsela Tesfaye. “Gossip vs. Tree-based Monitoring under Different Networking Conditions”. PhD thesis. Master’s thesis, KTH Royal Institute of Technology, 2010.

[46] Robbert Van Renesse, Kenneth P. Birman and Werner Vogels. “Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining”. In: *ACM Trans. Comput. Syst.* 21.2 (May 2003), pp. 164–206.

[47] M. Van Steen. *Graph Theory and Complex Networks: An Introduction*. Maarten Van Steen, 2010.

[48] Spyros Voulgaris, Daniela Gavidia and Maarten van Steen. “CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays”. In: *Journal of Network and Systems Management* 13.2 (2005), pp. 197–217.

[49] Fetahi Wuhib et al. “Decentralized computation of threshold crossing alerts”. In: *16 th IFIP/IEEE Distributed Systems Operations and Management (DSOM’05*. 2005, pp. 24–26.

[50] Fetahi Wuhib et al. “Robust monitoring of network-wide aggregates through gossiping”. In: *Network and Service Management, IEEE Transactions on* 6.2 (2009), pp. 95–109.