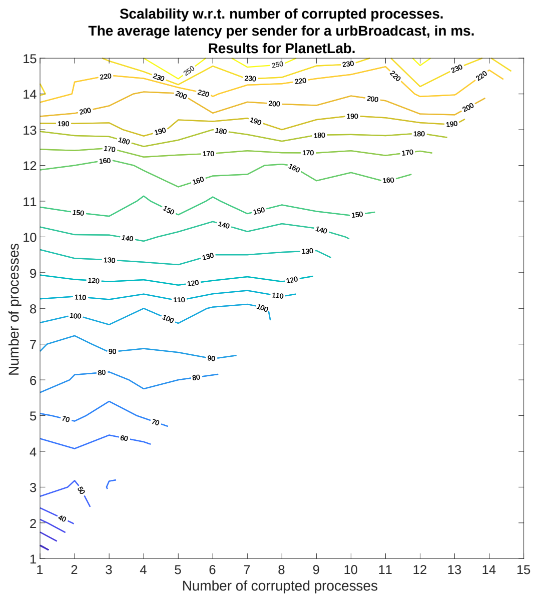


Self-stabilizing Communication Abstractions for Replicated Systems

Evaluating a stacking approach implementation of self-stabilizing distributed communication abstractions

Master's thesis in Computer science and engineering

Chaiyapruek Muangsiri
Oskar Jedvert

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

MASTER'S THESIS 2020

Self-stabilizing Communication Abstractions for Replicated Systems

Evaluating a stacking approach implementation of self-stabilizing distributed communication abstractions

Oskar Jedvert
Chaiyapruek Muangsiri

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Self-stabilizing Communication Abstractions for Replicated Systems
Evaluating a stacking approach implementation of self-stabilizing distributed communication abstractions
Oskar Jedvert
Chaiyapruek Muangsiri

Supervisor: Elad Michael Schiller, Department of Computer Science and Engineering
Advisor: Robert Gustafsson, CombiTech
Examiner: Tomas Olovsson, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Contour graph of the experiment scalability of number of processes with respect to number of corrupted processes. For more information please see Section 5.1.4

Typeset in LaTeX
Gothenburg, Sweden 2020

Self-stabilizing Communication Abstractions for Replicated Systems
Evaluating a stacking approach implementation of self-stabilizing distributed communication abstractions
Oskar Jedvert
Chaiyapruek Muangsiri
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

A self-stabilizing application can in the presence of transient faults resume its regular execution in a finite amount of time. The benefits of self-stabilizing replicated applications are becoming increasingly important as the traffic on the internet grows and the need for globally accessibly storage increases as we become more connected. There are two studied protocols, uniform reliable broadcast and set-constrained delivery broadcast. These can be used in combination to implement more powerful applications, one such example is the atomic snapshot application which is implemented and verified together with the studied protocols. By combining multiple protocols together, the implementation become more general and provide more reusability for the applications.
This report validates the correctness and evaluates the performance of two self-stabilizing communication protocols as well as the applications. The main focus of the report is the correctness and performance of the studied protocols under different system settings and environments. In order to achieve this, two different environment is used for experimental evaluation and the system settings are varied across the experiments.
The authors believe that the current implementation is correct, since the system is able to recover after the last occurrence of a transient fault. The recovery period is fairly short and has negligible impact on performance.

Keywords: science, computer science, engineering, thesis, self-stabilization, fault-tolerance, replicated systems, distributed systems, computer network.

Acknowledgements

We would like to thank Elad Michael Schiller, our supervisor, for his patience and great support throughout the project. We would also like to thank CombiTech and Robert Gustafsson, our company advisor, for their support and for providing us with a place to work. Finally we would like to thank our peers Oskar Lundström and Chibin Kou for their suggestions which helped us progress faster.

Oskar Jedvert, Gothenburg, October 2020
Chaiyapruek Muangsiri, Gothenburg, October 2020

Contents

List of Figures

1
Introduction

The rapid adoption of the internet is causing traffic to increase and the need for robust, scalable solutions is higher than ever before. One way to create scalable services is through distributed systems.

As the name suggest a distributed system refers to several machines connected either physically or through the internet, trying to perform tasks to accomplish a shared goal. Distributed systems can come in different forms, modern distributed systems leverage replication scheme to increase the capacity of the service at a moments notice. Transient faults are a temporary violation of the assumptions under which the system is assumed to operate. This can for instance be a memory corruption leading to some control variable changing unexpectedly, transitioning the system to an arbitrary state. To recover from such faults the *self-stabilization* paradigm can be used.

The purpose of the Master's thesis is to implement, validate, and evaluate multiple self-stabilizing *communication abstractions* for recently designed self-stabilizing communication abstractions by Lundström, Raynal, and Schiller [43, 42]. A communication abstraction is a communication layer which abstracts the communication details from the application while providing interfaces for sending and requesting data. We are using a stacking approach for the implementation. This means that we have multiple communication layers in which the lower layers provides interfaces to the upper layers in order to achieve the required properties of the higher layer. These self-stabilizing communication abstractions provides multiple advantages such as redundancy and availability to both small and large distributed systems. In case a minority of replicas fail, the system can recover in a bounded time and resume its legal execution given that no replicas fail during the recovery period. After implementing a communication abstraction, we validate the implementation with regards to its correctness and performance using an experimental evaluation.

1.1 Context and Motivation

Cloud solutions are becoming increasingly popular in many business areas since they provide fast scaling, helping with peak traffic, and redundancy giving more uptime. This is made possible due to the distributed system that the cloud is based of. Many cloud solutions, such as Kubernetes, employs what is called *state machine replication* to provide fast scaling and redundancy. While there is a mechanism to detect failing processes in most state machine replication based systems, the common response is to terminate the failing process and spawn a new one or restart it, such is the case for Kubernetes [32]. This means the information that the process held can be lost, imposing restrictions on how information is stored. Meaning that the developers need to ensure that the state of the system is stored or replicated outside the processes. While state machine replication is sufficient to create powerful highly scalable applications, solving the storage issue is not trivial.

The storage issue can be solved by using external storage or by replicating the data or events across the entire distributed system with some ordering constraints. The ordering constraint will depend on the applications needs, for instance a consistent counter application using only increments works regardless of delivery order as long as all messages delivered are the same across the system. While for a banking system ordering is very important since by reordering events can make, for instance, the deposit of some amount followed by a withdrawal for a lower amount might fail if the events are reordered.

Finally, all systems work under certain assumptions. Most distributed systems are made to work with the presence of crashes, but for some tasks this is not possible without enriching the system with failure detectors. For instance, consensus requires failure detectors, in a crash-prone system the entire system can halt and wait for a single processor which could be crashed or just progressing slowly. However, in the real world more things can go wrong other than the process crashing. For example, we can have a memory corruption in the form of a bit flip causing some essential variable of an algorithm to change completely. This is what is called a *transient fault*, which is discussed further in the Section 1.3. Distributed systems that can recover from transient faults are called *self-stabilizing* distributed systems.

1.2 Task descriptions

The studied algorithms are self-stabilizing versions of Uniform Reliable Broadcast and Set-Constrained Delivery.

Uniform Reliable Broadcast, or URB, informally achieves consistent delivery of messages across the distributed system. In the algorithm the URB protocol is enriched with FIFO delivery ordering.

Set-Constrained delivery broadcast, or SCD, allows messages to be delivered as a set with some constraints. This communication abstraction was first defined in 2017 [36].

The SCD algorithm interfaces with the URB algorithm to achieve reliable broadcast and FIFO ordering. We call this a “stacking approach” to the implementation of the algorithms. The architecture in Figure 1.1 provides an overview of this interfacing. The lower layers provide interfaces to the upper layers which they can leverage to achieve the desired behaviour. For example, in Figure 1.1, SCD interfaces with URB to provide set-constrained delivery to the snapshot application.

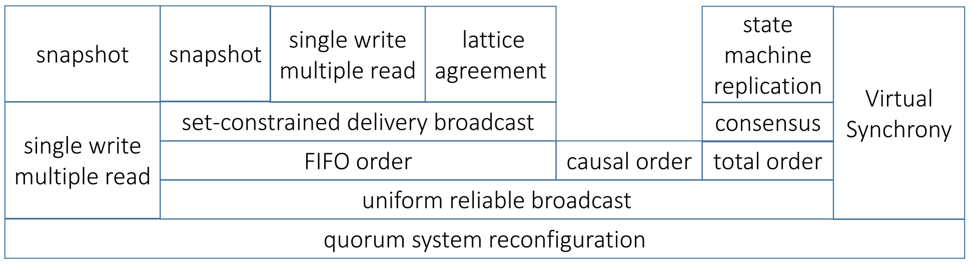

snapshot
snapshot
single write multiple read
lattice agreement
state machine replication
Virtual Synchrony
set-constrained delivery broadcast
consensus
single write multiple read
FIFO order
causal order
total order
uniform reliable broadcast
quorum system reconfiguration

Figure 1.1: The system architecture

1.3 Fault Model

We consider an asynchronous message-passing system with no shared memory between processes. Furthermore, there is no notion of a universal clock and the processes can not access the local clock either. The fault model includes node crashes, transient faults and communication failures such as, packet omission, duplication and reordering. A transient fault is any temporary violation of assumptions under which the system is designed to operate, e.g. the corruption of a control variable such as the current message number or program counter. To handle such faults, the studied communication abstractions use the property called self-stabilization which was proposed by Dijkstra in 1973 [7].

If a node crashes, it is assumed that it will never rejoin the system, allowing the system to ignore crashed nodes which could otherwise lead to slower performance.

1.4 Related Work

Now days, there are many self-stabilizing algorithm for distributed systems, For example, there are self-stabilizing algorithms for the Internet [21, 3, 22, 4], mobile ad hoc networks [5, 47, 38, 11, 45, 39, 41, 37, 40, 49, 25, 17, 17], replication systems [28, 31, 30, 29, 13, 23, 16, 19, 29, 15, 14, 10]. The design criteria of self-stabilization was extended to also consider selfishness [24, 18, 20] and Byzantine faults [34, 12, 33].

This work implements a self-stabilizing version of Set-Constrained Delivery Broadcast [35]. We note the existence of a Byzantine-Tolerant version as well [2]. This algorithm considers Byzantine faults, which is when a process is exhibiting Byzantine behaviour. When a process exhibits Byzantine behaviour this means that it can behave arbitrarily: it can crash, fail to receive or send messages, send arbitrary messages, start in an arbitrary state and make any arbitrary state transition. The algorithm can handle up to t byzantine processes where $t < n/4$ and n is the number of processes in the system. This algorithm also assumes these nodes to be faulty meaning that they are not concerned with the byzantine process when considering the termination, validity, integrity and order of the algorithm with respect to correct processes, since a byzantine process is not a correct process.

1.5 Our Contribution

We are the first, to the best of our knowledge, to evaluate two important components of self-stabilization uniform reliable broadcast and set-constrained delivery broadcast by Lundström, Raynal, and Schiller [43, 42].

We found that for the experiments where computational power is important, such as for the *bufferUnitSize* experiment, the PlanetLab environment outperformed the local network environment. While for experiments where the link latency is more important, such as the number of senders experiment, the local environment outperformed the PlanetLab environment thanks to its extremely low link latency. We also found that the recovery period after a transient fault has negligible impact on performance, both in terms of throughput and latency of delivered messages in both environments.

For the sake of supporting scientific progress in the area, the code of the developed protocols as well as the data used for presenting the experimental results can be found at `www.self-stabilizing-cloud.net`.

1.6 Document Structure

We outline our system setting in Chapter 2 as well as introduce the different paradigms and algorithms used in the project. We describe the system and its implementation in Chapter 3. The evaluation and experiment plan can be found in Chapter 4 together with the evaluation environments. We present our results in Chapter 5 with graphs from the experiments. Finally we discuss our results in Chapter 6.

2

Scientific Background

This chapter introduces the system settings and other important factors to the success of the algorithms.

2.1 System Settings

The system is an asynchronous message-passing system assuming fair communication. Fair communication means that if a process p_i receives a message from p_j, then said message was previously sent by p_j where p_i refers to the i'th process. Also, for any message m, if p_i receives m infinitely often from p_j then p_j has sent m infinitely often. The system consists of a set, P, of n failure-prone processes with unique identifiers. The failures include crashes and transient faults. In the case of crashes, the process is assumed to never restart and is excluded for the rest of the execution. For transient faults the self-stabilization paradigm is leveraged to transition back into a *legitimate state*, more on self-stabilization can be found in Section 2.2. There is no notion of universal clocks instead the processes use a logical local clock to measure the local progression.

The execution model can be described using the *interleaving model* [9]. This model describes the execution as a series of interleaving configurations and steps. Each step consists of an internal computation and a *single* communication operation, receiving or sending a message. A state s_i contains all p_i's variables and communication channels. The term *system state*, *global state* or *configuration* refers to the collection of all states in the system, $c = \{s_1, s_2, s_3, .., s_n\}$. Finally, an execution can then be described as multiple configurations, each followed by a single step, $E = c[0], a[0], c[1], a[1], ..$ where $c[0]$ is the initial configuration and each following configuration $c[k+1]$ is a result of the step $a[k]$ on the configuration $c[k]$.

2.2 Self-Stabilization

The self-stabilization paradigm was first introduced by Edsger W. Dijkstra in 1973[7]. He calls a system self-stabilizing when, “regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps”. A *legitimate state* can be defined as a system state where the system holds the requirements for the current algorithm. For instance, the requirements for a leader election algorithm is that at one point at most one leader may exist in the system. Self-stabilization was later formally defined by M. Schneider in 1993[48] as:

> “For a system S with respect to a predicate P, over its set of global states, where P is intended to identify its correct execution. S is self-stabilizing with respect to predicate P if it satisfies the following two properties:
>
> 1. **Closure** - P is closed under the execution of S. That is, once P is established in S, it cannot be falsified.
>
> 2. **Convergence** - Starting from an arbitrary global state, S is guaranteed to reach a global state satisfying P within a finite number of state transitions.”

The model of failure used by Schneider to discuss fault-tolerance is transient fault. Transient failures can change the local state of a process which in turn will change the global state of the system. A self-stabilizing system can then recover from such transient failures given that they do not continuously occur.

Furthermore, since transient failures can’t always be detected, a process in a self-stabilizing system must keep checking if its local state is legitimate. Meaning if a transient failure occurs, the process will eventually detect that it is no longer in a legitimate state and then take some action. This requires what is referred to as the “do forever loop” or “forever loop”.

2.3 Failure Detectors

Asynchronous consensus protocols can fail to reach consensus in the presence of even a single process crash [8][26]. This stems from the impossibility for a asynchronous protocol to tell the difference between a slow and a crashed process. In the case of SCD and URB the system can continue but the performance may be impacted. Since URB needs to wait for all processes to receive and acknowledge a message, the performance can be affected or the entire system’s progression can be halted entirely in the presence of crashes. To prevent this, distributed failure detectors can be used.

Each local process has access to one or more failure detector modules, each failure detector module monitors the processes in the system and maintains a list of currently *trusted* processes. If a process is erroneously removed from the *trusted* list by the failure detector it can later be included again. An asynchronous distributed system can then use this *trusted* process list to continue its progression without having to wait for slow or crashed processes.

However, if a failure detector makes a mistake it should not affect the correctness of the algorithm/implementation running on the processes. This means that if a process is erroneously marked as crashed and later included in the *trusted* list again, said process should still behave according to the specification of the algorithm/application. For example, if a process running a broadcast algorithm such as URB is erroneously marked as crashed it should after some time be marked as trusted again. Once this happens the messages that were broadcasted during the period in which the process was marked as crashed should arrive at the this process at some point. Meaning once all processes finishes, they should all have the same delivered messages regardless if they were erroneously marked as crashed at some point and later unmarked. This can also be solved by not allowing processes to reenter the *trusted* list once they have been removed from it. This solution is used in the studied protocols, in this way the system can ignore very slow processes which may lead to better performance.

2.4 FIFO Uniform Reliable Broadcast

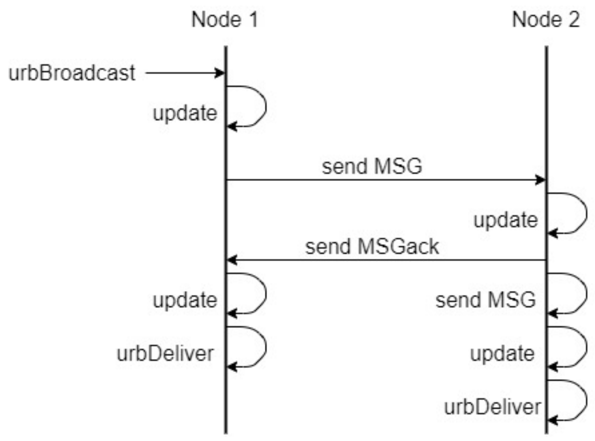


Figure 2.1: The flow of a single *urbBroadcast* operation in a system of two processes

Uniform Reliable Broadcast, or *URB*, achieves in informal terms a consistent delivery of messages across the processes in the system. Figure 2.2 shows the flow of a single *urbBroadcast* operation in a system of two processes. For simplicity's sake, the MSG send event from process 1 to itself after the first update call and the MSGack send event after node 2 sends MSG to itself are omitted. This section presents an outline of the URB algorithm starting with its protocol messages and then a high-level outline of the algorithm.

Before describing the URB algorithm, the concept of *obsolete* needs to be explained. An obsolete record informally means that the record is not of interest for any node in the system. Specifically, all processes in the system has acknowledged the record and the record has been URB-delivered by all processes.

The URB algorithm uses three types of protocol messages: *MSG*, *MSGack* and *GOSSIP*. A *MSG* protocol message contains (m, j, s) which are: the message *m*, the sender's unique identifier *j* and the sequence number for the message *s*. A *MSGack* protocol message contains (j, s) which are: the sender of the acknowledged message's unique identifier *j* and the sequence number for the acknowledged message *s*. The *GOSSIP* from node *j* to node *i* message contains (*maximumSeq*, *receivedObsSJ*, *sentObsSJ*) which are:

- The maximum sequence number present in the senders buffer for records from node *i*.
- The highest received obsolete sequence number from node *i* for node *j*.
- The highest sent obsolete sequence number from node *j* for node *i*.

Algorithm 1: High-level outline of FIFO-Uniform Reliable Broadcast for p_i

```
bufferUnitSize: Maximum number of messages in the buffer per sender
sequenceNumber: Message index number
receivedObsSeq[1..n]: highest received obsolete sequence number from each node
sentObsSeq[1..n]: highest sent obsolete sequence number to each node
nextToDeliver[1..n]: next sequence number to deliver for each sender, ensuring
  FIFO delivery order
buffer: a set of records containing message and extra metadata

fn obsolete(r): begin
    True if all nodes have recieved r, r has been delivered and
      $receivedObsSeq[r.sender] = r.sequenceNumber + 1$.
end
fn urbBroadcast(m): begin
    wait for room in the buffer then increment sequenceNumber and call update;
end

fn update(m,j,s,k): begin
    if $s \leq receivedObsSeq[j]$ then message sequence number s is too old
        discard the message and return
    end
    if message not present in buffer and m is not $\perp$ then
        Create new record with:
          • Message m
          • Sender j
          • Sequence number s
          • Delivery flag False
          • Received by set: {j,k}
        Insert the new record into the buffer
    else
        Insert {j,k} into the receivedBy set for the record with sender j and
          sequence number s
    end
end
```

```
25 While true do
26     if The buffer contains duplicate records or records with an $\perp$ message then
27         Empty the buffer
28     if sequenceNumber is out-of-bounds or messages are missing from the buffer then
29         set all indices in sentObsSeq to the value of sequenceNumber
30 foreach $p_k \in P$ do
31     ensure the gap between receivedObsS[k] and the largest sequence number in
       the buffer for sender k is not larger than bufferUnitSize
32 foreach record in buffer do
33     if all trusted nodes have received the record and next[record sender] = record
       sequence number then
34         urbDeliver the record and set record delivered flag to true
35         Increment next[record sender]
36     foreach $p_k \in P$ : k has not received the record and i is the sender of the
       record do
37         send MSG(record) to $p_k$
38 While There is an obsolete(record) in the buffer do
39     Increment receivedObsS[record sender]
40 Remove all records in the buffer that are not needed by any node by using
   receivedObsS and sentObsS.
41 foreach $p_k \in P$ do
42     Send gossip message to $p_k$ with:
         • The maximum sequence number in the buffer for sender k
         • receivedObsS[k]
         • sentObsS[k]
43 upon MSG(m,j,s) arrival from $p_k$: call update(m,j,s,k) and send MSGack(j,s)
   to $p_k$
44 upon MSGack(j,s) arrival from $p_k$: call update($\perp$,j,s,k)
45 upon GOSSIP(maximumSeq, receivedObsSJ, sentObsSJ arrival from $p_j$:
         • Set sequenceNumber to maximum of itself and maximumSeq
         • Set receivedObsS[j] to maximum of itself and sentObsSJ
         • Set sentObsS[j] to maximum of itself and receivedObsSJ
```

2.5 Set-Constrained Delivery Broadcast

The key values of *Set-Constrained Delivery Broadcast*, or *SCD* are the sequence number (sn) which is a logical clock representing the progress of the process and the buffer which contains a list of messages that have been FIFO-delivered by the lower layer. This algorithm allows multiple messages to be delivered in a set simultaneously, while still ensuring an ordering constraint. The ordering constraint can be described as: if a message delivers a set of messages with message m and later a

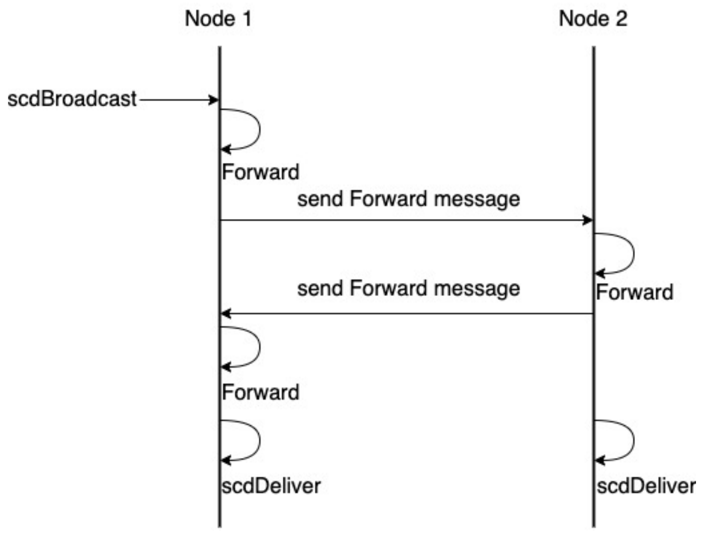


Figure 2.2: The flow of a single *urbBroadcast* operation in a system of two nodes

set of messages with message *m'*, no process delivers a set of messages with message *m'* before delivering a set of messages with message *m*.

The SCD algorithm uses *FORWARD* and *GOSSIP* messages to progress. A *FORWARD* message contains (m, j, sJ , f, sF) which are: the message, the sender's identifier, the sender's logical clock, the forwarder's identifier and the forwarder's logical clock. The *GOSSIP* message from process *j* to process *i* contains, (maxClockJ, receivedObsClockJ, sentObsClockJ, receivedSpace, sentSpace) which are:

- The maximum clock value for process *i* in process *j*'s buffer.
- The highest received obsolete clock value from process *i* for process *j*.
- The highest sent obsolete clock value from process *j* for process *i*.
- The lowest remotely stored clock value in process *j* for process *i*.
- The lowest locally stored clock value for process *j*.

Algorithm 2: High-level outline of the self-stabilizing Set-Constrained Delivery Broadcast algorithm for p_i

```
1  bufferUnitSize: Maximum number of messages in the buffer per sender
2  buffer: stores message records that has been delivered by the FIFO-URB layer
     but not yet SCD-delivered
3  logicalClock: local logical clock used to measure $p_i$'s local progress and to
     identify $p_i$'s SCD-broadcast messages
4  receivedObsClock[1..n]: highest received obsolete clock value from each process.
5  sentObsClock[1..n]: highest sent obsolete clock value to each process.
6  receivedSpace[1..n]: lowest clock value stored in the buffer from each process.
7  sentSpace[1..n]: lowest stored clock value for each process.
8
9  fn obsolete(record, k): begin
10     Returns True if the record has been broadcasted, delivered and
         receivedObsClock[k] +1 = the record's clock value for k
11 fn saved(j): begin
12     Returns a set of all clock values stored in the buffer sent by $p_j$
13 fn mS(j): begin
14     Returns one of the following
         • Returns 0 if $p_j$ is not trusted
         • Returns receivedObsClock[j] if j equals to $p_j$'s unique identifier and $p_j$ is
           trusted
         • Returns the minimum value of sentObsClock[k] where k is all the unique
           identifiers that $p_i$ trusts
15 fn scdBroadcast(m): begin
16     Waits until there is room in the buffer by ensuring that the length of the
         set returned by calling saved(i) is less than $n \cdot$ bufferUnitSize forward.
17 fn forward(m, j, sJ, f, sF): begin
18     if Record with sender j and clock[j] = sJ exists then
19         Update clock[f] for the record to sF
20     else if Clock value sJ is not too old, i.e. sJ > receivedObsClock[j] then
21         Create new record with:
             • Message m
             • Sender j
             • Clock, which is a list of clock values for all nodes. Initialized to $[\infty, ...., \infty]$
               and then updated with: (clock[i],clock[j],clock[f]) ← (logicalClock, sJ, sF).
             • Delivery flag False
22     else if receivedObsClock[f] +1 = sF then
23         Increment receivedObsClock[f]
24 fn tryDeliver() begin
25     Create a toDeliver set containing all records that has been broadcasted but
         not yet delivered such that a majority of the indices in the record's clock
         list are less than $\infty$.
```

```
26
27    while $\exists r \in$ toDeliver, $\exists r' \in$ buffer $\setminus$ toDeliver: r' has not been delivered
        and a majority of nodes has received r' before r do
28        Remove r from toDeliver
29    if toDeliver is not empty then
30        foreach record in toDeliver do
31            Set record delivered flag to True
32        scdDeliver all messages in toDeliver
33 while True do
34    if There are duplicate records in the buffer, records with clock value for $p_i$
        equals to $\infty$ or the length of the set returned by saved(i) is greater than $n \cdot$
        bufferUnitSize then
35        Empty the buffer
36    if If the logicalClock is out-of-bounds or message are missing from the buffer
        then
37        Set all indices in sentObsClock to logicalClock and set
            receivedObsClock[i] to logicalClock
38    foreach Record in the buffer that has not yet been broadcasted or the
        broadcast operation has terminated do
39        FIFO-URB-Broadcast the record and mark the record as broadcasted.
40    tryDeliver();
41    while There exists an obsolete(record, k) in the buffer for $p_k \in P$ do
42        Increment receivedObsClock[k].
43    Remove all records in the buffer that are no longer needed by observing the
        values in receivedObsClock, sentObsClock, receivedSpace, sentSpace.
44    foreach $p_k \in P$ do
45        if saved(k) returns an empty set then
46            set receivedSpace[k] to $\perp$
47        else
48            set receivedSpace to the minimum value between receivedObsClock[k]
                and saved(k)
49        Send GOSSIP message to $p_k$ with: (Maximum clock value for $p_k$ in the
            buffer, receivedObsClock[k], sentObsClock[k])
50 upon FORWARD(m, j, sJ, f, sF) FIFO-URB-delivered from $p_j$: call
    forward(m, j, sJ, f, sF)
51 upon GOSSIP(maxClockJ,receivedObsClockJ, sentObsClockJ, receivedSpaceJ,
    sentObsSpaceJ ) from $p_j$:
      • Set logicalClock to the maximum of itself and maxClockJ.
      • Set receivedObsClock[j] to the maximum of itself and sentObsClockJ.
      • Set sentObsClock[j] to the maximum of itself and receivedObsClockJ.
      • if receivedSpaceJ is not equal to $\perp$ then set sentObsSpaceJ[j] to the
        maximum of txSpace[j and itself otherwise if sentObsSpace[j] is not equal to
        $\perp$ and receivedSpaceJ is the next expected value then set txSpace[j] to
        receivedSpaceJ
      • if sentObsSpaceJ is not equal to $\perp$ then set receivedSpace[j] to the maximum
        of sentObsSpaceJ and itself otherwise if sentObsSpaceJ is the next expected
        value then set receivedSpace[j] $\leftarrow$ sentObsSpaceJ
```

2.6 Applications

With the properties provided by *SCD* and its underlying reliable FIFO-broadcast mechanism, we can implement applications on top of *SCD* easily. The application of interest is *Atomic snapshot*. The following section presents a high-level description of the *Atomic snapshot*.

2.6.1 Atomic snapshot

Atomic snapshot allows multiple readers and writers to access shared memory. In this case, it is a shared register of integers.

Algorithm 3: High-level outline of the self-stabilizing Atomic snapshot object for p_i

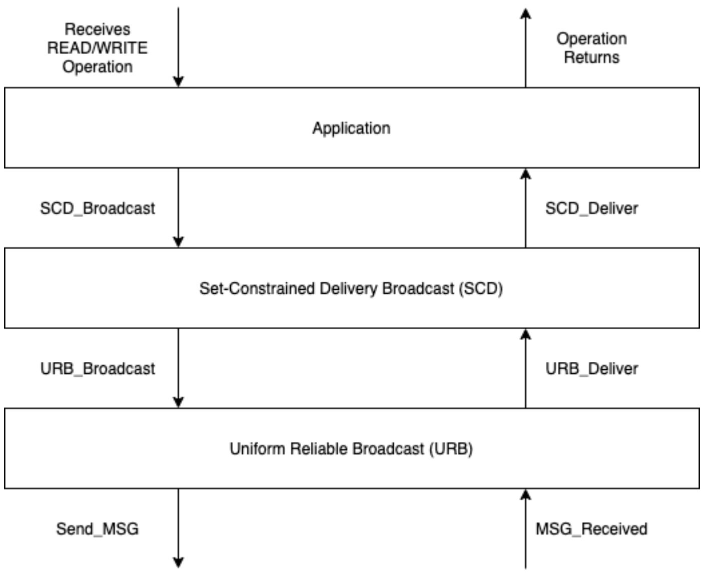
```
1 reg[1..m]: stores register values
2 tsa[1..m]: stores a timestamp associated with each register value
3 fn snapshot(): begin
      1. Call scdBroadcast with a message SYNC(i) as an argument.
      2. Wait until the unique identifier(txDes) returned from SCD is ⊥ or the
         SCD-layer hasTerminated(txDes) returns true, return reg.
4 fn write(r, v): begin
      1. Call scdBroadcast with a message SYNC(i) as an argument.
      2. Wait until the unique identifier(txDes) returned from SCD is ⊥ or the
         SCD-layer hasTerminated(txDes) returns true, return reg.
      3. Call scdBroadcast with a message WRITE(r, v, <tsa[r].date + 1, i>).
      4. Wait until the unique identifier(txDes) returned from SCD is ⊥ or the
         SCD-layer hasTerminated(txDes) returns true, return reg.
5 upon a set (WRITE[1..k], SYNC[1..l]) is SCD-delivered:
6 foreach r ∈ {1,...,k} do
7     if tsa[r] is less than the maximum timestamp contained in
        WRITE[1..k].timestamp then
8         reg[r] ← WRITE[r].v
9         tsa[r] ← the maximum timestamp contained in WRITE[1..k].timestamp
```

3 The System

The following section describes the architecture of the communication abstractions and how they cooperate. The next section continues with the implementation of the communication abstractions and the applications.

3.1 Architecture

The three main layers in the implementation are Uniform Reliable Broadcast (URB), Set-Constrained Delivery Broadcast (SCD) as well as the application layer implemented on top of the SCD layer as depicted in Figure 1.1.



Receives READ/WRITE Operation
Operation Returns
Application
SCD_Broadcast
SCD_Deliver
Set-Constrained Delivery Broadcast (SCD)
URB_Broadcast
URB_Deliver
Uniform Reliable Broadcast (URB)
Send_MSG
MSG_Received

Figure 3.1: The URB, SCD and Application layers and how they interact.

An overview of how the layers interact can be found in Figure 3.1. The left-hand side is the flow of an operation invocation until *Send_MSG* which invokes the kernel function call *send()* which sends the message to a socket. The right-hand side is the flow of of an incoming broadcast operation starting from the *MSG_Received* which consumes an incoming messages on a socket by using the *recv()* kernel function call and ending with a response containing the operation result.

The following describes the interactions between the layers for a sender of a message. When a new application operation is invoked, the node generates a new application message with contents specified by the application algorithm. This application message is then passed down to the SCD layer. The SCD layer creates a new forward message with the node's identifier and a logical timestamp. The forward message is then handed to the underlying URB broadcast layer by calling the *urbBroadcast* operation having the forward message as an argument. The URB layer then broadcast this message according to the URB algorithm specification.

Once a message arrives at a node, the behaviour is the same regardless if it is the sender of the message or not. When the URB message arrives, it is included in the URB buffer. Thereafter, the receiving node must wait for acknowledgements from all of the system nodes before URB-delivering this message. Once the message has been URB-delivered, the URB layer passes the delivered message to the upper SCD layer and the message can then be treated as a SCD-layer record. The SCD layer iterates over all records and use the lower layer's *urbBroadcast* to broadcast the records to the rest of the system if needed. In order for a SCD-layer record to be delivered, it must be received and acknowledged by the majority of the nodes in the system. After that, the message is consumed by the application layer and the application layer can perform tasks depending on the contents of the message and which application algorithm is being executed.

3.1.1 Rust Programming Language

The system is implemented in the Rust programming language[44]. Rust is a statically typed compiled language which is designed to fully support concurrent programming. The borrow checker in Rust guarantees memory safety and isolation which prevents data races, stack and buffer overflow and access to uninitialized or unallocated memory. However, this presented some implementation challenges, which are discussed in Section 3.2.1.

To exchange messages between modules the implementation uses the Rust synchronization primitive *channel*. A channel is used to send data between two threads in a safe manner allowing for coordination without using shared memory or locks. The channel provides two ends, a send end and a receiver end. Data can be sent on the send end and then consumed by continuously receiving from the receive end until there is no more pending data.

The implementation also uses another flavor of channel called *ring channel*. Unlike a normal channel ring channel is bounded and overwrites older messages when full. Ring channel favors throughput over lossless message passing, making it ideal when produced messages needs to be sent on the channel as soon as possible. In the implementation this is used to keep the Linux socket buffer clear by consuming messages and sending them on the ring channel.

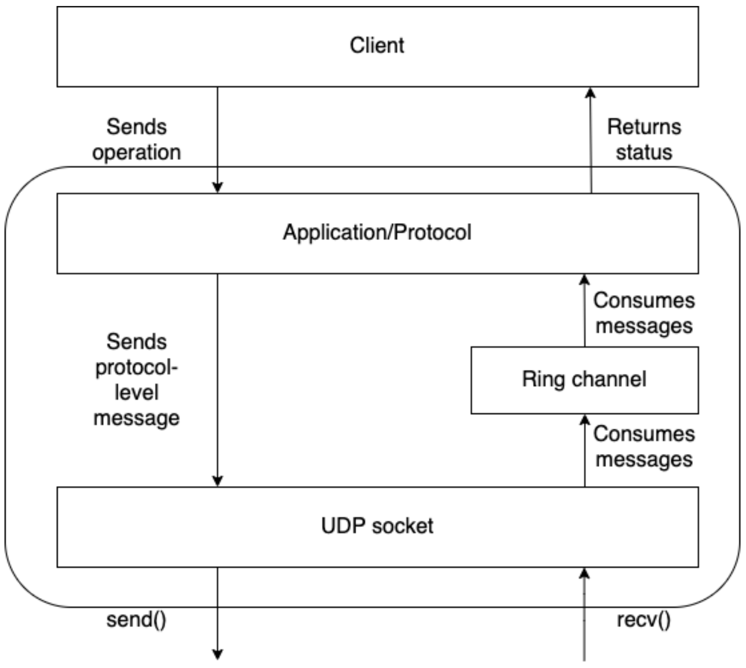


Figure 3.2: Overview of the interactions between the UDP socket, application and client.

3.2 Implementation

This section provides an overview of the implementation and outline how the algorithms have been implemented. The implementation contains the following modules

- A client, an external module which initiates operations and sends them to the node operations to the node via a communication channel.
- A node, a module which contains the logic of the implemented algorithms. It uses a single thread that receives operations from the client, receives messages from other nodes and processes the messages based on the algorithms.
  - The node contains a UDP socket to send and receive messages.
  - The node has a ring channel buffer which continuously consumes messages from the UDP buffer and provides these consumed messages to the node when needed in order to prevent the UDP buffer from getting full.

The illustration of these modules and their interactions can be found in Figure 3.2.

The node can be initialized without starting the ’do forever loop’, the client can then access the node to modify or access internal variables. Once the ’do forever loop’ has started the ownership of the node belongs to the node itself and can therefore not be accessed by the client anymore. The method which starts the ’do forever loop' provides three channel ends, which purpose is outlined later in this section. The 'do forever loop's flow of execution can be described as follows:

1. Receive messages
2. Receive operations
3. URB 'do forever loop' iteration
4. SCD 'do forever loop' iteration*

Starting with receiving pending messages, the node iterates over the ring channel which contains messages received from the UDP socket owned by the node. For each type of message, the corresponding method for handling such an incoming message is invoked and then the message is removed from the ring channel.

The node then receives pending operations, this is done similarly by looping through operations pending in the operation channel's receive end. For each operation, the node ensures there is enough room left in the buffer as specified by the algorithm. If there is room the corresponding broadcast operation is invoked and *OK* is sent on the status channel, if there was no room *ErrNoSpace* is sent on the status channel.

After the node has received all pending operations and all pending messages, it can then start the main message processing logic. First, the URB processing logic is executed as described by the URB algorithm, of which an overview can be found in Section 2.4. Lastly if the application or abstraction running is based on the SCD component, the SCD processing logic is executed as described by the SCD algorithm, of which an overview can be found in Section 2.5.

3.2.1 Client communication

Due to Rust's strict memory safety, the client has to use channels in order to communicate with the node. An overview of the client and node interactions can be found in Figure 3.3. The client can initialize the node and modify public variables until the client starts the node 'forever loop'. When the client starts the 'forever loop' ownership of the node is passed to the node itself and the node can at this point no longer be modified by the client. Starting the 'forever loop' provides provides the client with three channel ends, two send ends and one receive end. These channel ends are: the operation channel send end, the terminate channel send end and the status channel receive end. The client can use the operation channel's send end to send messages to the node which are then broadcasted according to the algorithm. Finally, the receive end can be used by the client to learn about the status of the last operation. To ensure an operation has been accepted by the node, the client therefore first sends on the operation channel and then after waits for the status from the status channel receive end. If the status channel receive end contains an

*When running the SCD component or applications built on top of the SCD component

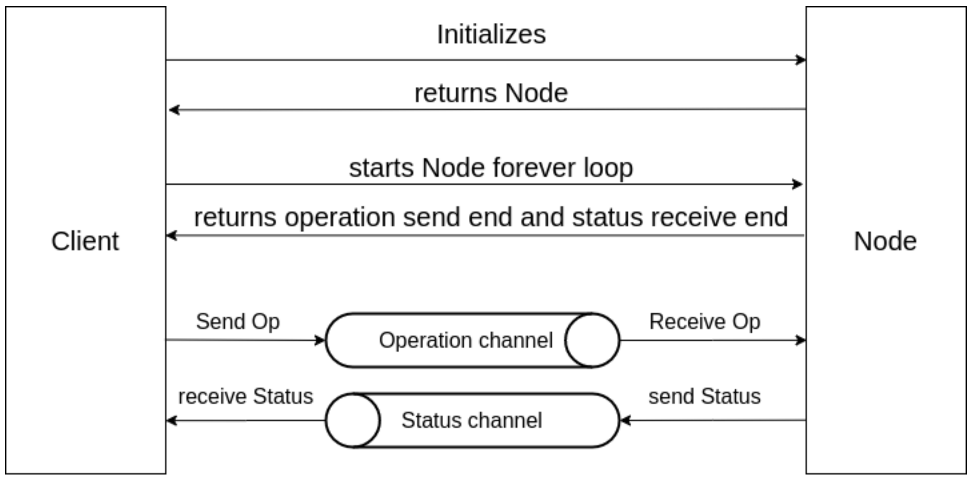


Figure 3.3: The client and node interactions.

OK, the operation was processed and is going to be broadcasted. However, if the status receiver returns *ErrNoSpace*, the operation was not accepted as there was no space left for the resulting message.

3.2.2 Optimization

A number of optimizations can be implemented in order to improve the overall performance of the URB and SCD. In this section, we briefly discuss the optimizations done to *URB* and *SCD*. First, we alter the data type of *recBy*, a set of nodes that have received and acknowledged the record stored together with each record in the buffer, which initially was a *HashSet* into a compact form of vector of bits called *BitVec*. BitVec stores a specified size of continuous bits in which each bit can be conveniently used as a Boolean variable, a bit implies true if it is set to 1, or false otherwise.

The *recBy BitVec* associated with each record is sent to other nodes along with the actual message when the node execute line 38-39 of the URB algorithm. When other nodes receive this message, they perform a set union operation between the current *recBy* coupled with the record stored in the buffer and the *recBy* contained in the received message. This means that pi can infer that pk has received the message when pi receives a message from pj that contains the unique identifier of pk in the *recBy*. This information is further used to infer if all the nodes have received and acknowledged this message. Doing this can speed up the acknowledgement process since the node does not have to wait for all the acknowledgement messages from all other nodes in the system, but instead it can be inferred by a chain of acknowledgements contained in the message.

The delivery mechanism of *URB* is also optimized. Normally, a node would wait for all acknowledgement messages from all other nodes in the system before URB-delivering a message. Instead, a message can be URB-delivered if it has been acknowledged by a majority of the current trusted nodes in the system.

Both URB and SCD contain self-stabilization statements at the beginning of the do forever loop. This can be expensive if executed every iteration. Instead of executing these statements every iteration, the *delta* parameter is used to specify how often the self-stabilization statements are executed, i.e., the self-stabilization statements are executed in every *delta* iterations. The same *delta* parameter is also applied to determine the frequency of gossip sending. This means that a gossip message is also sent every *delta* rounds in order to reduce the message sending and processing overhead.

4 Evaluation Environment and Plan

The evaluation of our implementations is done through experiments. Our experiments are constructed to answer one or more research questions. The experiments are measured against a set of evaluation criteria. Some of the evaluation criteria are parameters which are modified to observe the effect on the other evaluation criteria. This chapter outlines our evaluation criteria, research questions, evaluation environments as well as the experiment plan.

4.1 Evaluation Criteria and Research Questions

The criteria measures the performance of the system by observing the number of completed operations and the run length of the experiments. The operations considered depends on which implementation is being evaluated. For the SCD/URB implementations we are concerned with the *scdBroadcast*/*urbBroadcast* operations, i.e. the messages broadcasted by SCD/URB. For the applications we are interested in the read/write operations. Our evaluation criteria focus on the following aspects:

1. **Throughput**. The number of operations the system can complete over a period of time.
2. **Latency**. The time it takes from the invocation of an operation to its completion.
3. **Recovery time**. The time during which the system recovers after the occurrence of a transient fault.
4. **Fault-tolerance**. Given that fair communication is guaranteed the system should be able to operate normally in the presence of packet omission, duplication and reordering, as long as at most a minority of nodes are crashed.
5. **Scalability**. How the performance of the system is affected by increasing the number of servers or other scalable factors, such as senders, readers, writers, etc.

Following this criteria we have selected a few research questions. The evaluation focuses on the following research questions:

- How does the evaluation criteria for the system change with respect to the number of nodes as well as the number of senders?
- How does the *bufferUnitSize* parameter affect the evaluation criteria?
- How is the evaluation criteria affected by the trade-off parameter δ? The parameter δ describes how often we execute the self-stabilization code, i.e. the self-stabilization statements are executed every δ iterations.
- How does the the number of application-level writers and readers affect the evaluation criteria?

4.2 Experiment description

Some of our experiments are conducted on multiple implementations since their behaviours are similar and the properties under test are the same. For the communication implementations, URB and SCD, we are interested in measuring the evaluation criteria with respect to the broadcast operations and the delivery events, while for the application implementations the evaluation criteria is measured against the read and write operations. The experiments 1 to 5 are used to evaluate the communication abstraction implementations SCD and URB, while experiments 6 and 7 are used to evaluate the applications built on top of these communication layers. Our list of experiments includes:

1. **Scalability of number of servers with respect to throughput and latency**. As the number of servers in the system increases the number of acknowledgement protocol-level messages required to deliver a single message increases as well. The experiment is conducted by considering a varying number of servers in the system, *i.e.* 1 to 15 servers.
2. **Scalability of number of senders with respect to throughput and latency**. The senders are the servers that generate new messages. Increasing the number of senders increases the rate at which we generate new messages. The experiment is conducted by selecting a number of servers between 1 and 15 and then using a varying number of senders between 1 and the chosen number of nodes.
3. **Scalability of bufferUnitSize size with respect to throughput and latency**. The parameter bufferUnitSize determines how many messages that can be stored for a single sender. Increasing this allows the node to store more messages simultaneously. This is a trade-off to consider since the more messages in the buffer the more processing time is required to iterate over the buffer. However using a very small *bufferUnitSize* would cause the client

operations to be blocked and the client will have to wait for the buffer to be consumed more often. The experiment is conducted using a varying number of nodes between 1 and 15 nodes and the buffer size increases logarithmically from 1 to 10 and so on.

4. **Overhead of system recovery**. When the system recovers from an invalid configuration, the performance of the system could be affected as some processing time and messages are needed to recover. The aim is to observe how the evaluation criteria is affected by the number of nodes that transitions to an invalid configuration at a single point in time. The experiment is conducted using a varying number of nodes between 3 and 15. The number of "failing" nodes varies depending on the number of nodes but is at least 1.

5. **Overhead of the self-stabilization property with respect to δ**. Since the node must keep iterating over the buffer every δ iterations in order to achieve self-stabilization, this can reduce the performance of the algorithm. The experiment is conducted with a set number of servers of 8 and with a logarithmically increasing δ argument, i.e., 1 to 10 until a very large number such as 2^{64}.

6. **Scalability of read operations with respect to write operations**. Since the writers and readers compete for resources when running concurrently, the number of possible read operations to perform depends on the number of write operations that is performed. The experiment is conducted using 10 readers and a varying number of writers, between 0 and 10.

7. **Scalability of write operations with respect to read operations**. The aim is to observe how the number of read operations is affected by the number of concurrent write operations. The experiment is conducted using 10 writers and a varying number of readers, between 0 and 10.

4.3 Evaluation Environment

In order to validate the communication abstractions in an efficient manner and at the same time be able to test them on a global scale we consider three evaluation environments:

- **Local Machine**: At this local test environment the implementation runs on a single machine for the sake of preliminary functionality testing and debugging during development. The nodes are represented as processes and the communication is done using the UDP stack in order to mimic communication between remote nodes.

- **Local Network**: A local network evaluation environment where the implementation runs on a set of local machines. We are using three Raspberry Pis, one model 3 and two model 2 connected together by a HP Procurve V1410-24

Ethernet Switch, capable of 100 Mbit/s transmission. This environment also serves as a demonstration platform for showing the system's ability to work in local networks that, for example, Cloud system often use. These machines are running Raspbian without any desktop environment. In such environments, the nodes are limited in capacity so we might reach the bounds of the Linux socket buffer. Such corner cases of the studied algorithms are validated.

- **PlanetLab**: PlanetLab provides a number of geographically distributed connected machines, in various places in Europe [6]. PlanetLab was chosen to evaluate our implementation as it can reflect a real-world scenario, where the nodes provided have varying performances and network conditions. This introduces real-world challenges such as latency, packet loss and congestion. The machines in PlanetLab are running different versions of Linux.

4.4 Evaluation Plan

We run the experiments on two environments: geographically distributed computers by using PlanetLab [6] as well as the local network environment described in Section 4.3. Next we describe our experiment setup and the algorithmic test cases.

4.4.1 Experiment setup

Each physical PlanetLab and Raspberry Pi machine runs a single node which in turn runs the program being tested. Each node is assigned a unique identifier at the start of the execution. Senders and writers are assigned from the lowest node identifier while readers are assigned from the highest node identifier. The number of servers in the system is dependant on which experiment is being conducted.

Each combination of experiment, number of servers or other varying parameter is run for 60 seconds during which the program generates new operations. After this period no new operations are generated and we allow the system to process the remaining messages before terminating. To determine if a node should terminate we observe the buffer of messages. If the length of the buffer is 0 or it remains unchanged over a fixed number of iterations we assume that the node is unable to progress further and can therefore terminate. In addition, we also require a node to run for a minimum number of iterations since a node can have no messages in its buffer when it starts its final iterations but there might be incoming messages.

The operation latency is measured by the sender attaching a timer to each invoked operation. For every delivered operation from itself the sender can then store the value of the timer for the newly delivered operation in a list. The average operation latency for a sender is then calculated by the average of this list. Finally the average operation latency per sender in the system is then calculated by average of each process latency list.

4.4.2 Evaluation utilities

The evaluation can be divided into three steps: the installation of the source code on the computers in the current evaluation environment, running the experiments and aggregating the results. We use a host file to store the available hosts which is shuffled randomly before each experiment. A number of hosts present in PlanetLab are inserted into the hosts file. The hosts were chosen based on plcli [46], which provided us with a list of healthy hosts in the slice.

To run the experiments without requiring too much user input we use a scenario file. This is a text file in which each line represents a scenario to run. A scenario has the following options:

- Number of servers: how many nodes should run this scenario.
- Variant: the communication abstraction or application that should be executed.
- Number of writers/senders: how many writers should be present in the system.
- Number of readers: how many readers should be present in the system.
- Number of "corrupted" nodes: how many nodes that should, at one point in time, corrupt an internal variable.

In order to simulate a transition into an invalid configuration, we flag some nodes as "corrupted" nodes, how many such nodes exist is defined by the experiment. A corrupted node will, after a fixed number of iterations, corrupt an internal variable. An internal flag is then flipped to ensure this only occurs once. Otherwise the node behaves the same way as all other nodes.

As described by the scenario, a number of nodes are executed on the hosts in the hosts file. Each node is executed on a separate host. The scenarios are executed sequentially. When a scenario finishes its execution the evaluation files are downloaded to the local machine which executed the evaluation. After downloading these evaluation files a merged result file is created, or if the result file is already present, the evaluation files are merged into the result file.

Once all scenarios have finished their execution, the result file containing the results of all executed scenarios is present locally. The result file contains a list of JSON objects. Each JSON object in this list represents the results of a single scenario. This result object contains the name of the scenario and a map with node identifiers used as keys and JSON objects as values. The JSON objects used as values in this map contains:

- The set of broadcasted messages.

- The set of URB delivered messages.
- The set of SCD delivered messages.
- The run length of the scenario.

This result file is used to aggregate the results from the scenarios and output the experiment graph.

4.4.3 Experiments

Experiments 1 to 5 are used to evaluate the communication abstraction implementations URB and SCD, while the experiments 6 and 7 are used to evaluate the application built on top of these communication layers. Each combination of the parameters for every experiment is run 10 times. The highest and the lowest values of the run results are removed in order to avoid having outliers in the results. Then, the average is used to calculate the result of the experiment.

5

Result

In this chapter we present the results of our experiments which are conducted on two environments, more information about the environments can be found in Section 4.3. Each parameter combination on every experiment have been executed ten times with the highest and lowest result removed to mitigate the effect of outliers. More on how the experiments are conducted can be found in Chapter 4.

We found that for the experiments where computational power is important, such as for the *bufferUnitSize* experiment, the PlanetLab environment outperformed the local network environment. While for experiments where the link latency is more important, such as the number of senders experiment, the local environment outperformed the PlanetLab environment thanks to its extremely low link latency. We also found that the recovery period after a transient fault has negligible impact on performance, both in terms of throughput and latency of delivered messages in both environments.

The chapter first presents the results from the experiments on Uniform Reliable Broadcast followed by the results from the Set-Constrained Delivery Broadcast. Finally we present the results from the experiments on a snapshot application implemented with our communication abstractions.

5.1 Uniform Reliable Broadcast

5.1.1 Scalability of number of servers with respect to throughput and latency

Latency is the focal evaluation criterion, Experiment 1 vary the number of processes and measure the average latency. We expect the latency to grow with the number of processes. Indeed, this is the observed result from Figure 5.1 both for the local network and PlanetLab.

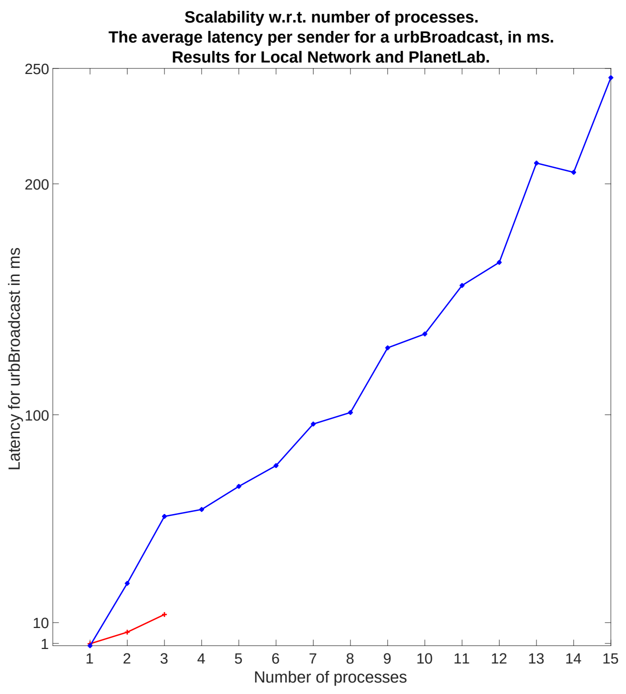


Figure 5.1: Scalability of number of processes w.r.t. urbBroadcast.

5.1.2 Scalability of number of senders with respect to latency

Since the latency is expected to grow as the number of processes is growing, there is a need to try to point out the key contributors for such growth. In Experiment 2, we aim at investigating whether the number of senders can incur a greater impact on the latency than the number of servers. From Figure 5.2 we observe that the growth in the number of servers has a slightly higher impact than the growth in the number of senders for the case of local network. Broadly speaking, similar observations can be made for the experiment on PlanetLab, see Figure 5.3.

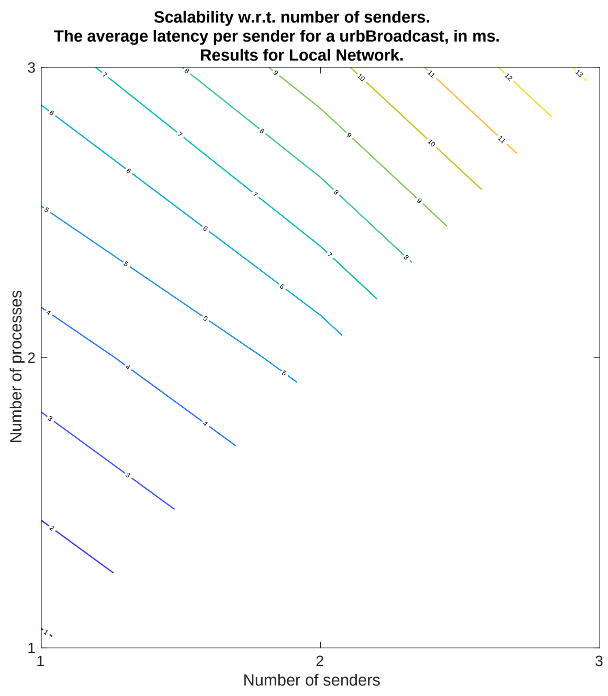


Figure 5.2: Scalability of number of senders w.r.t. urbBroadcast.

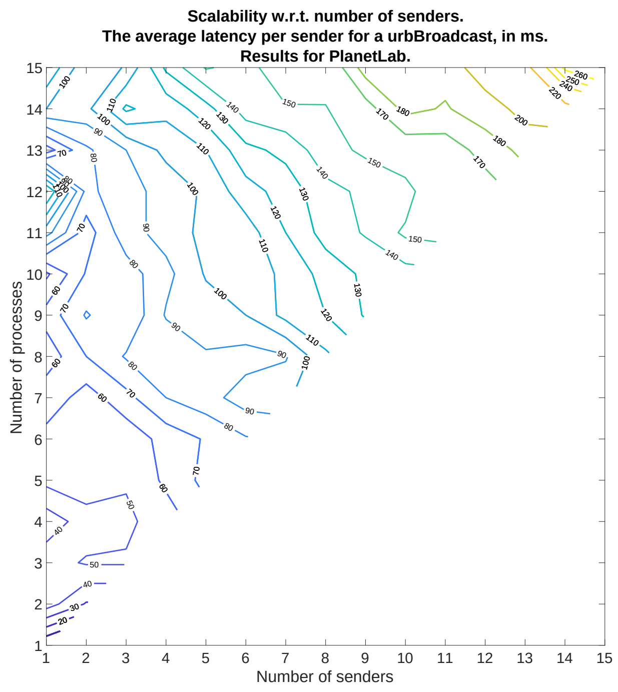


Figure 5.3: Scalability of number of senders w.r.t. urbBroadcast.

5.1.3 Scalability of bufferUnitSize with respect to latency

Since the number of messages per sender allowed in the buffer is limited by the *bufferUnitSize*, increasing it would also increase the number of records in the buffer. The program needs to iterate over the entirety of the buffer to process its contents. This means more resources are required to iterate over the buffer with a larger *buffer-UnitSize*. Because of this, we would expect to see a performance loss in latency with an increasing *bufferUnitSize* but we should see an increase in throughput. However we expect the throughput gain to diminish as the *bufferUnitSize* grows past a certain point. In Figure 5.4 we observe that as the *bufferUnitSize* increases the latency decreases, as we expected. However in Figure 5.5 we see a smaller performance penalty until *bufferUnitSize*=100 when the latency increases significantly. The difference between the environments can be explained by the fact that the Local Network uses lower computational power devices, increasing the performance impact of a large *bufferUnitSize*.

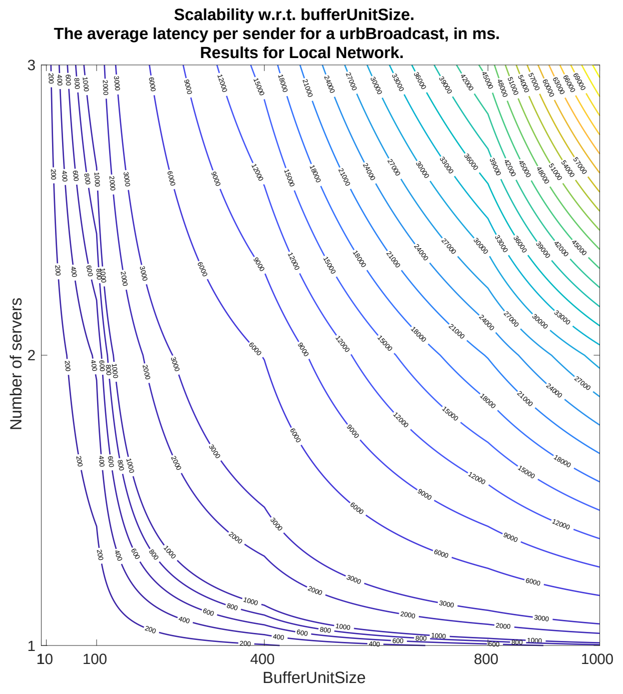


Figure 5.4: Scalability of bufferUnitSize w.r.t. urbBroadcast.

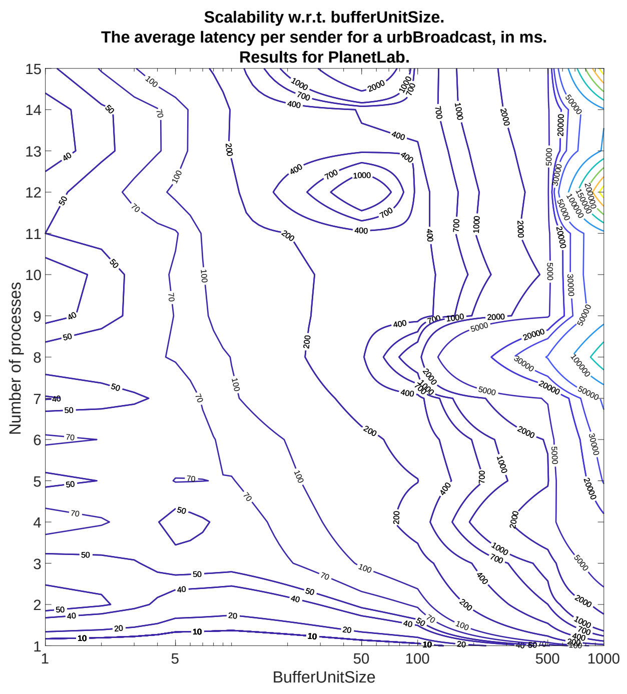


Figure 5.5: Scalability of bufferUnitSize w.r.t. urbBroadcast.

5.1.4 Overhead of system recovery

In Experiment 4, we aim to observe whether the performance of the system is affected when corrupted processes are introduced. A corrupted process executes the program normally until a set point in time when it randomly changes its local sequence number and thereafter continue normal execution. It is important to point out that processes are corrupted at different times during the execution. Therefore, the performance should not greatly increase or decrease as the number of corrupted processes grows as long as the recovery periods do not overlap. In Figure 5.6 we observe that the latency increases slightly when the number of corrupted processes grows from 0 to 1 but afterwards there is a clear linear trend. For Figure 5.7 we can observe a similar trend, increasing the number of corrupted processes has no clear impact on the latency. There is some noise in the PlanetLab environment, this can be explained by the fact that the machines in PlanetLab have varying processing power and the network condition can vary as well which could cause these fluctuations. As expected, in both environments there was not a clear correlation between the number of corrupted processes and latency.

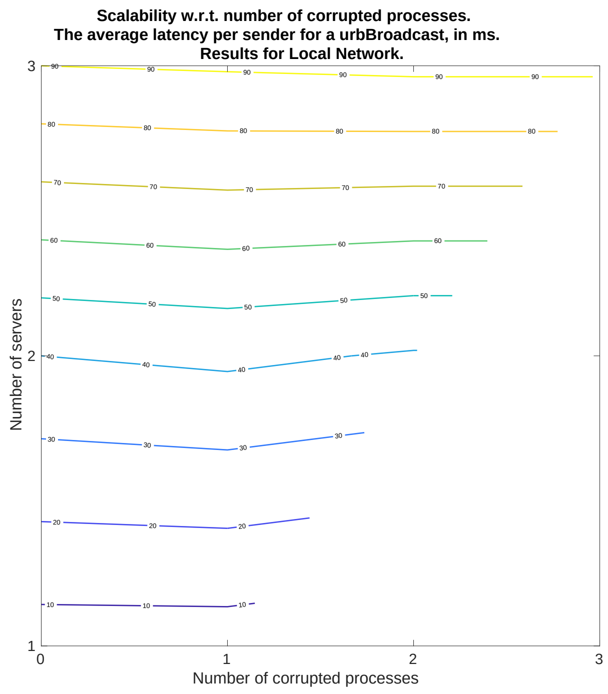


Figure 5.6: Overhead of system recovery w.r.t. urbBroadcast.

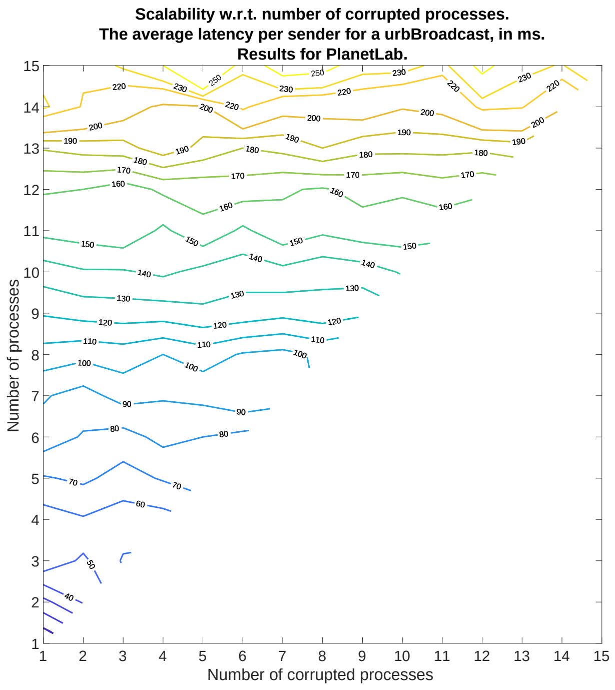


Figure 5.7: Overhead of system recovery w.r.t. urbBroadcast.

5.1.5 Overhead of the self-stabilization property with respect to δ

In Experiment 5, we want to find out whether the delta parameter imposes an impact on the overall performance of the protocol. Since the self-stabilization statements are only executed every delta rounds, increasing delta would free up processing resources to be used on processing messages instead, and thus the overall performance should be improved. In order to propagate the local information to the system, gossip messages are sent every delta rounds. In addition, gossip messages are always attached with every outgoing message. Thanks to the attached gossip messages, stale records can be removed from the buffer once they are delivered by all trusted nodes due to the information inside the attached gossip message. Given this we do not expect a higher delta to increase the latency, rather we expect it to decrease it since computational resources are freed.

As observed in Figure 5.8, the latency for the case of Local Network slightly decreases as the value of delta increases as expected. Similar observations can be seen in Figure 5.9 for the case of PlanetLab with some noise. Both environments have a similar trend, the latency decreases slightly as the value of delta increases. For instance, the latency is approximately 55 for the case of 4 nodes and delta 1. The latency slightly decreases to 50 and remains the same until the delta approaches 500. When the delta is 500, the latency increases to 55 again, and then drops to 50 after that. This can be explained by the unstable network conditions and also varying computational resources of the machines we used.

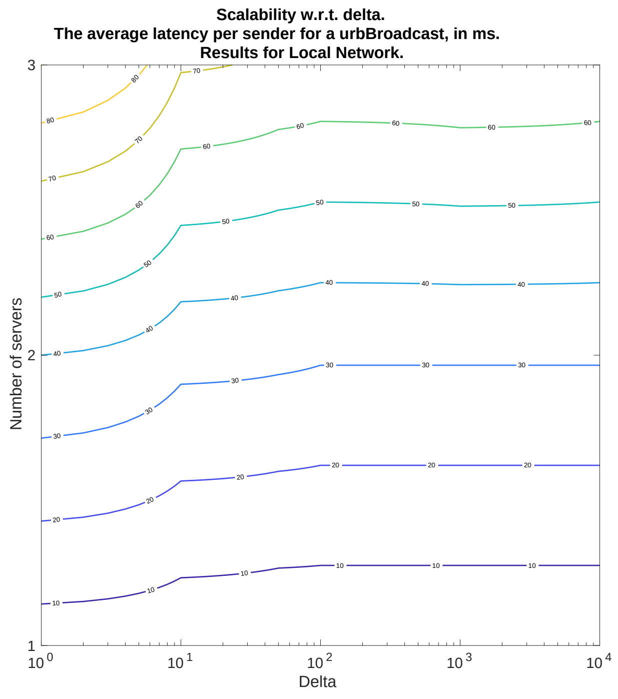


Figure 5.8: Scalability of delta w.r.t. urbBroadcast.

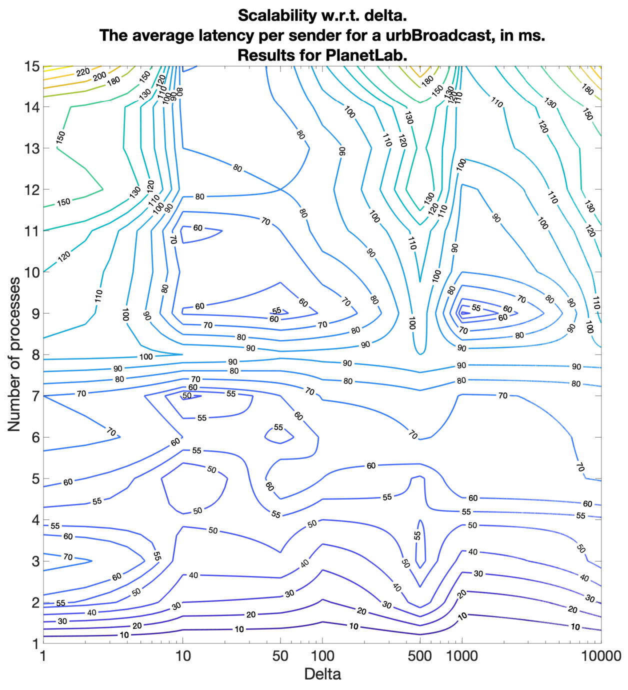


Figure 5.9: Scalability of delta w.r.t. urbBroadcast.

5.2 Set-Constrained Delivery Broadcast

5.2.1 Scalability of number of servers with respect to throughput and latency

Similarly as Experiment 1 in Section 5.1.1, we aim to find out how the number of processes affects the overall performance of both evaluation environments. Ideally, in a system with nodes that have similar network and computational power, we would expect to see a linear increase in latency as the number of processes grows. We would also expect the throughput to decrease as the number of processes increases. From Figure 5.10 we can see that the local environment's latency increases faster than for PlanetLab due to the lower computational power of the local environment. In Figure 5.10 we also observe a trend similar to a linear increase for the PlanetLab environment between 3 and 10 processes. However after that we can see steeper slope until 15 processes, this could be explained by the fact that the PlanetLab machines and networks differ a lot in performance and a few machines might be slow. Why these slow machines do not affect the lower results as much can explained through probability. Given that there is a minority of slow machines, the odds of picking a slow machine increases as the number of processes increases. This could explain the behaviour for values more than 10 processes.

In Figure 5.11 we observe a sharp decrease in throughput as the number of processes grows. This could be explained by the fact that in order to achieve high throughput we need to remove message fast to make room for new messages in the buffer. In order to remove a message from the buffer, a sender needs to know that all receivers have received, delivered and removed this message. This means a slow receiver can decrease the throughput for all senders in the system.

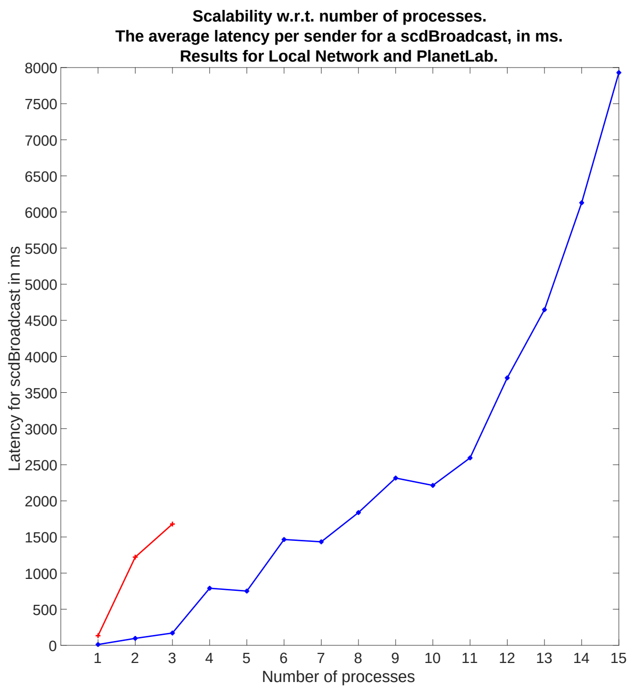


Figure 5.10: Scalability of number of processes w.r.t. scdBroadcast.

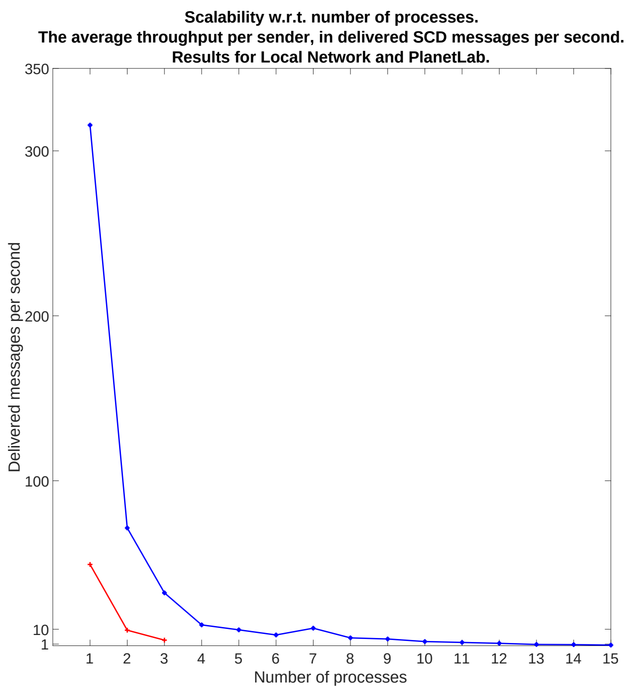


Figure 5.11: Scalability of number of processes w.r.t. scdBroadcast.

5.2.2 Scalability of number of senders with respect to throughput and latency

In addition to Experiment 2 in Section 5.1.2, this experiment measures throughput as well. We aim at investigating whether the number of senders can incur a greater impact on the latency and throughput than the number of processes.

With an increasing number of senders, more records can be stored in the buffer at the same time. Furthermore, there is a need to wait for more round trip times to deliver a message as the number of processes grows. In a system with low computational power and low link latency, we would expect the number of senders to be the dominant factor. Oppositely, in a system with high computational power and high link latency we would expect the number of processes to be the dominant factor.

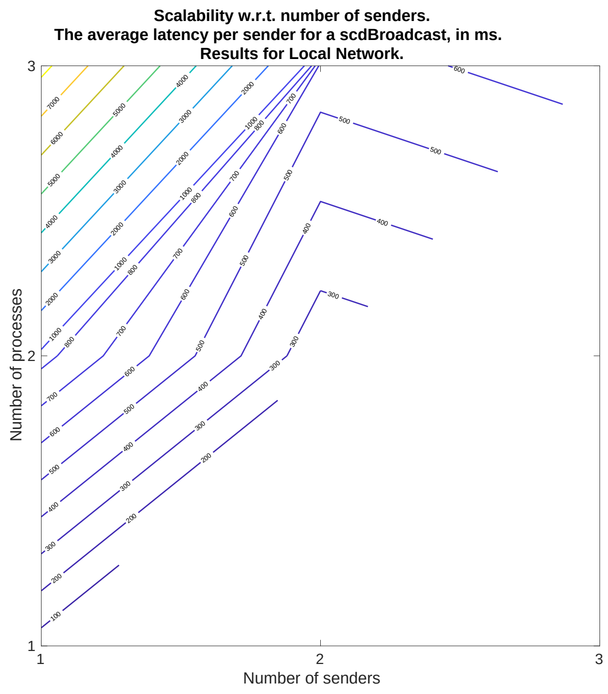


Figure 5.12: Scalability of number of senders w.r.t. scdBroadcast.

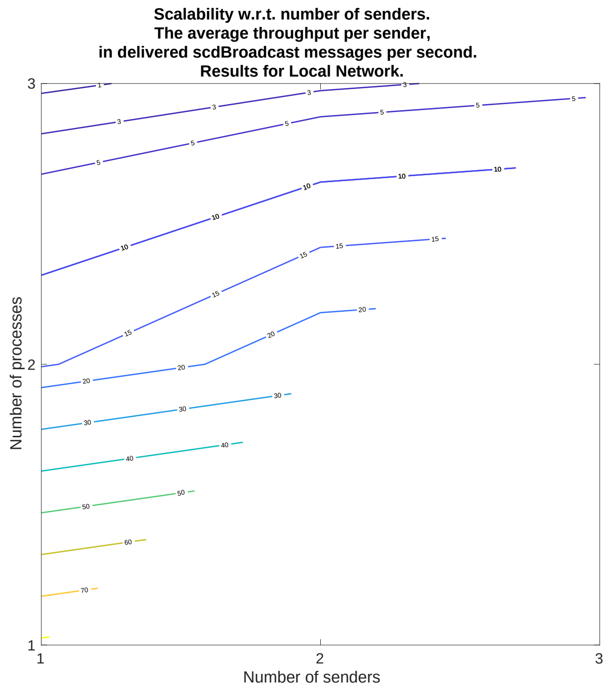


Figure 5.13: Scalability of number of senders w.r.t. scdBroadcast.

In the local environment, in Figure 5.12, we can see a decrease in latency when the number of senders grows from one to two. This could be explained by the fact that in every outgoing message, some gossip information is attached. With more senders, messages are sent more often leading to less stale gossip information in the system. However when the number of senders grows from 2 to 3 we can see an increase in a latency. There is also a large increase in latency when the number of processes grows from 2 to 3. This can be explained by the fact that we only need to wait for a majority to deliver a message. Meaning that when there is 2 processes and 1 sender, the sender can immediately deliver its own messages. On the other hand when there is 3 processes and 1 sender in the system, the sender has to wait for an acknowledgement from at least one more process.

In Figure 5.13 we can observe that the number of processes seems to be the dominant factor for throughput. This could be explained by the fact that the main problem for throughput is the removal of messages. To remove a message we need to wait for all processes to receive, deliver and notify the sender of the message that the message was delivered. However, there is also a slight increase in throughput when the number of senders grows, which could be due the same gossip piggybacking mechanism as explained earlier.

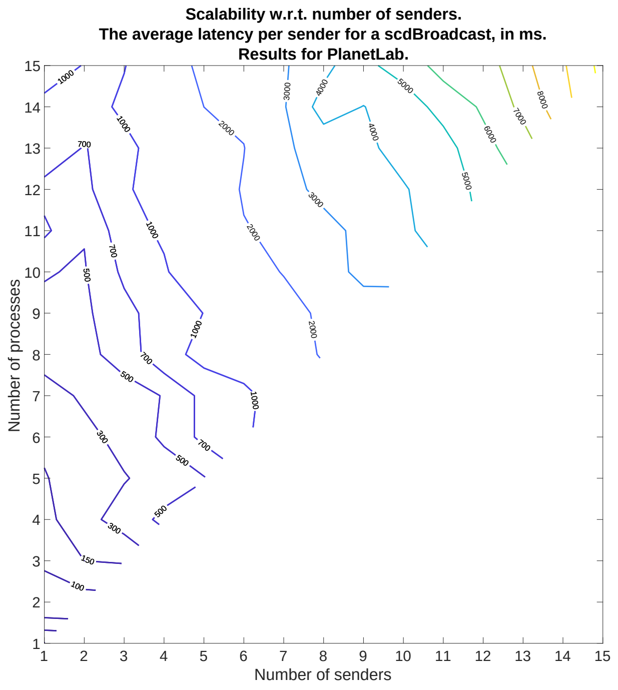


Figure 5.14: Scalability of number of senders w.r.t. scdBroadcast.

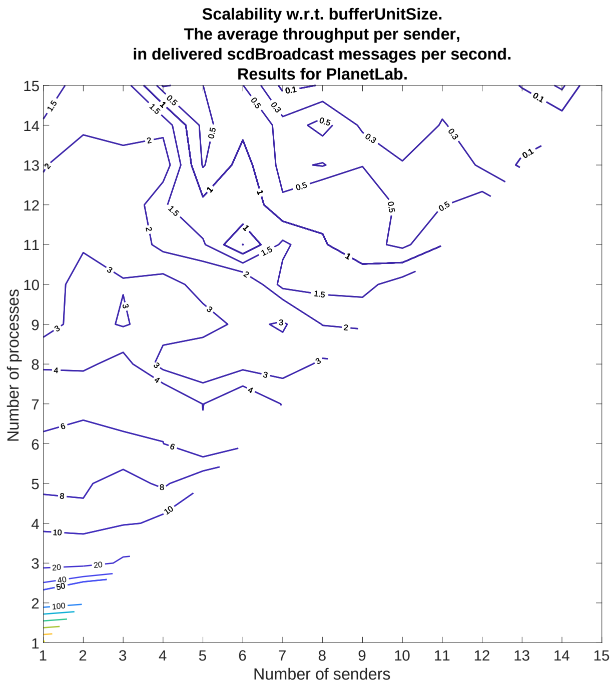


Figure 5.15: Scalability of number of senders w.r.t. scdBroadcast.

In Figure 5.14 we can see a different trend compared to the local environment. As the number of processes grows there is an increase in latency. However as the number of senders grows the impact of the number of processes decreases. For example, for 8 senders there is a smaller increase in latency as the number of processes grows compared the case when we 3 senders.

As illustrated in Figure 5.15, we can observe a similar trend compared to the local environments throughput. However, when the number of processes grows above 7 the number of senders starts to impact the throughput slightly.

The local environment did not behave as we expected with regards to the both latency and throughput. Our expectations for latency only hold for the case above 2 senders while for throughput it is in fact the opposite from what we expected, the number of processes is the dominant factor.

The latency for PlanetLab also behaved differently from what we would expect, when the number of senders grows above 8. However, below that the trend is similar to what we expected. The throughput behaviour is similar to the local environment for which the number of processes is also the dominant factor.

5.2.3 Scalability of bufferUnitSize size with respect to throughput and latency

In Experiment 3 we aim to find out how the performance of the system is affected by the *bufferUnitSize* which defines the maximum number of records per sender in the buffer. In this case this translates to a maximum of $N \cdot bufferUnitSize$ SCD records, where N is the number of processes in the system. However every SCD record requires N URB records in order to reliably deliver it. This means that the effective maximum number of records in the buffer is $N^2 \cdot bufferUnitSize$.

Therefore we expect the number of processes, N, to be the dominant factor in this experiment. We would then expect to see a large decrease in throughput as the number of processes grows. However, we would expect to see a smaller increase in latency as the number of processes grows since delivering a message requires less iterations over the buffer than removing it.

From Figure 5.16 we see that throughput peaks at bufferUnitSize 10 and decreases as the number of processes grows. In Figure 5.4 we see a smaller increase in latency when the number of processes grows and *bufferUnitSize* is the dominant factor in this case, this could be due to the low computation power of the local environment.

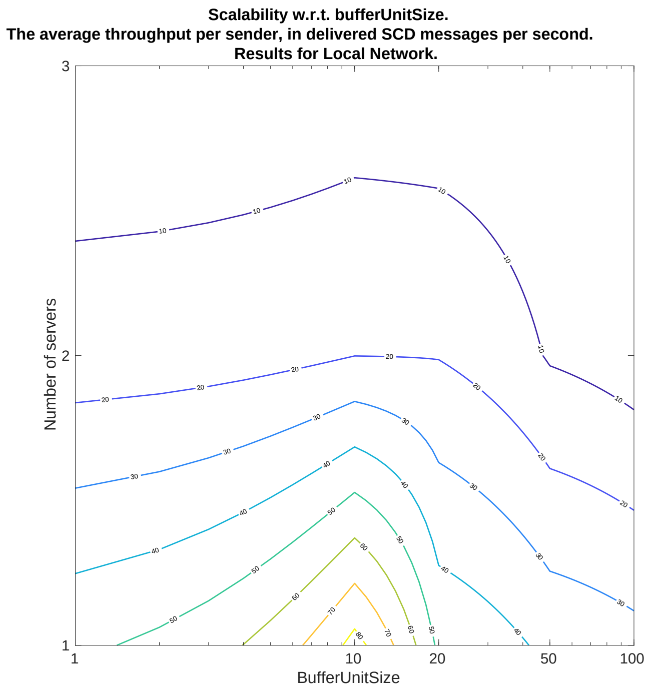


Figure 5.16: Scalability of bufferUnitSize w.r.t. scdBroadcast.

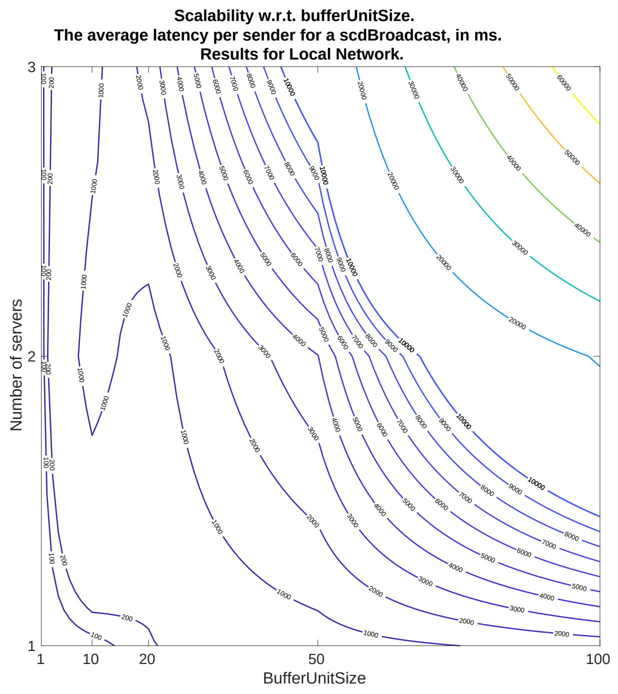


Figure 5.17: Scalability of bufferUnitSize w.r.t. scdBroadcast.

For the PlanetLab environment we can see in Figure 5.18 that throughput decreases significantly after 3 processes. For values below 3 processes *bufferUnitSize* we see a similar trend as for the local environment, we have a peak in throughput at buffer-UnitSize 10 and the number of processes is the dominant factor for the *throughput* decrease. For larger values of number of processes we can see that the *bufferUnitSize* starts to affect the throughput, making both number of processes and *bufferUnitSize* affect the throughput.

Lastly, in Figure 5.19 a small increase in latency as the number of senders grows and a larger increase in latency as the *bufferUnitSize* grows, as we would expect.

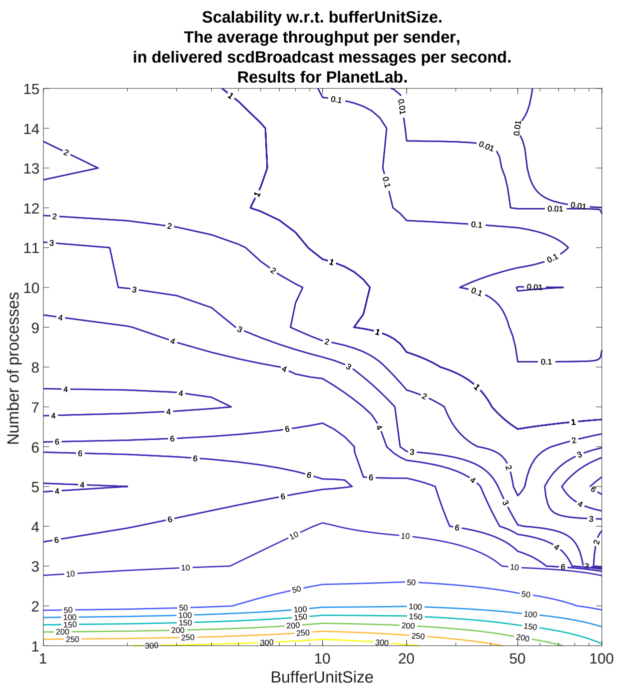


Figure 5.18: Scalability of bufferUnitSize w.r.t. scdBroadcast.

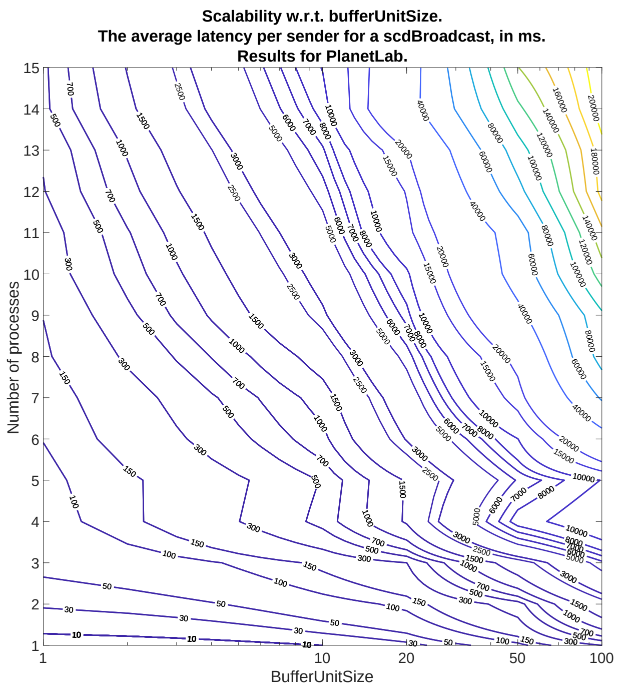


Figure 5.19: Scalability of bufferUnitSize w.r.t. scdBroadcast.

5.2.4 Overhead of system recovery

In this Experiment we aim to find out how corrupted processes can affect the performance of the system. The corrupted processes randomly changes their local sequence number variable at different times as explained in Section 4.4.2.

When a corrupted process change its sequence number, some self-stabilization statement will eventually be executed. After some local computation, the now stabilized process needs inform the rest of the system of its current local state. This means that some extra overhead is introduced in order for the system to recover after a random change in sequence number. This overhead is finite as long as the process or processes do not continuously incur this change and fair communication is guaranteed. To prevent this, in the experiment each corrupted process is only allowed to change the sequence number once and at different times. Given this we believe that there should be a small decrease in performance when the number of processes grows and a larger decrease with the growth of the number of processes.

For the local environment, which can be found in Figure5.21 for latency and Figure 5.20 for throughput, this seems to be correct as there is a small decrease in performance when the number of corrupted processes grows beyond 2. However the pattern is not as clear for lower number of corrupted processes.

As illustrated in Figure 5.23 and Figure 5.22 the PlanetLab environment has a similar trend. There is not a clear increase or decrease in performance as the number of corrupted processes grows. This could be explained by the fact that the computational overhead will not be as big of a problem for the more powerful PlanetLab machines compared to the local environment machines.

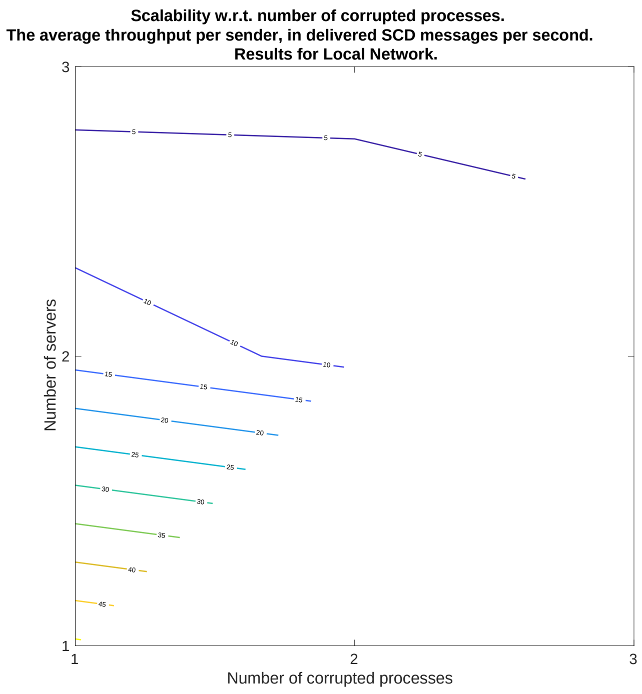


Figure 5.20: Overhead of system recovery w.r.t. scdBroadcast.

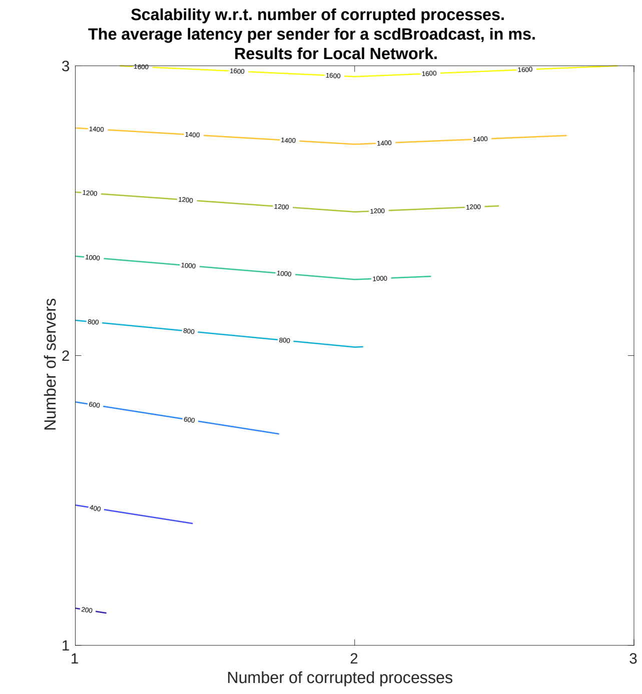


Figure 5.21: Overhead of system recovery w.r.t. scdBroadcast.

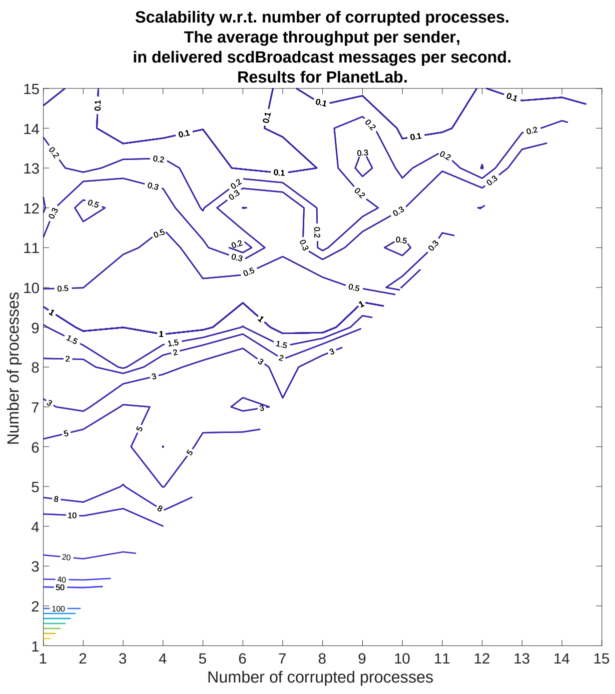


Figure 5.22: Overhead of system recovery w.r.t. scdBroadcast.

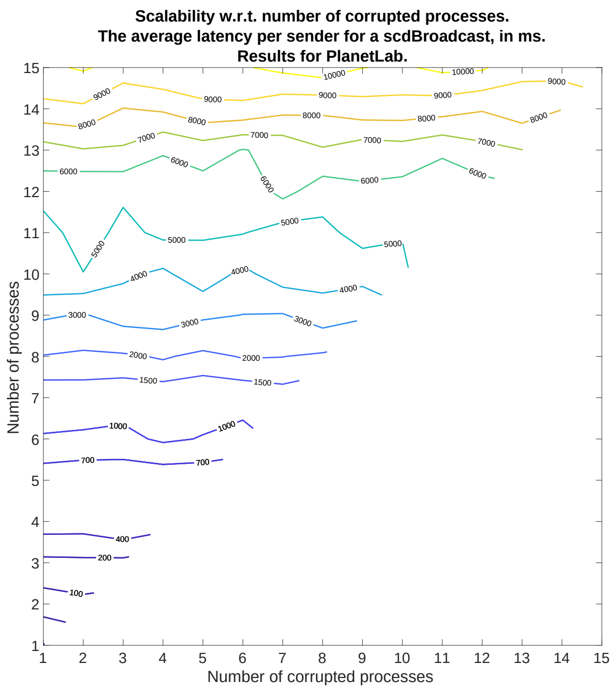


Figure 5.23: Overhead of system recovery w.r.t. scdBroadcast.

5.2.5 Overhead of the self-stabilization property with respect to δ

As described in Section 5.1.5, the aim of this experiment is to find out the impact of the *delta* parameter to the overall performance of the system. In addition to latency, this experiment measures the throughput of the system.

We expect low delta values to be more problematic on low computational power machines since self-stabilization statements are executed and gossip messages are sent more frequently. However, for high computational machines the impact of the additional self-stabilization statements and extra messages is not as problematic.

In Figure 5.25 we can see that increasing the delta decreases the latency, just as we expected from a low computational power system. However, from Figure 5.24 we observe a peak in throughput at delta 10.

From Figure 5.27 we can observe that delta does not affect the latency in the PlanetLab environment. As the number of processes grows latency increases while a growing delta does not seem to have as big of an impact on latency.

Illustrated in Figure 5.26, we can see that when number of processes is less than 4 delta does not seem to have an impact when it grows beyond 10. For higher number of processes the throughput decreases as the delta grows beyond 50. For delta 1000 we can see a significant decrease in throughput for more than 3 processes.

From this we can draw the conclusion that for low computational machines a high delta is preferred if latency is the priority, if not then there is a drawback when increasing the delta beyond 10. For high computational machines delta does not seem to impact the latency at all, however throughput does decrease when delta grows and the system is larger than 4 machines.

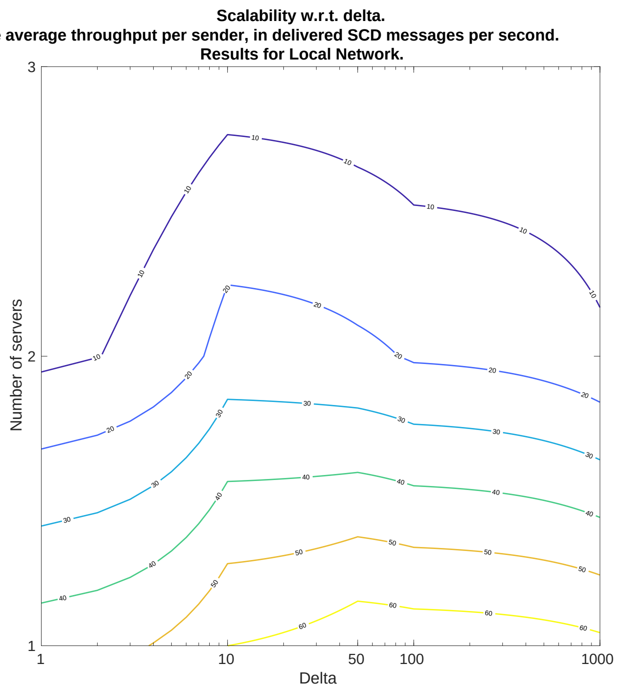


Figure 5.24: Scalability of delta w.r.t. scdBroadcast.

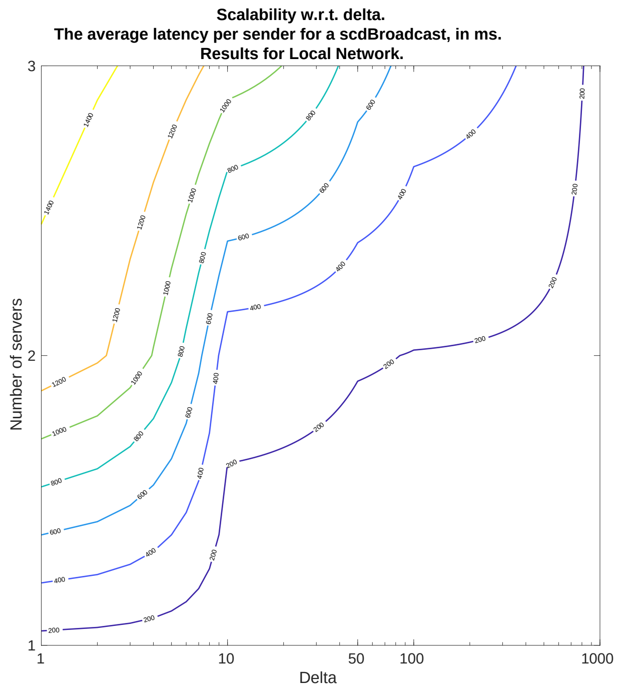


Figure 5.25: Scalability of delta w.r.t. scdBroadcast.

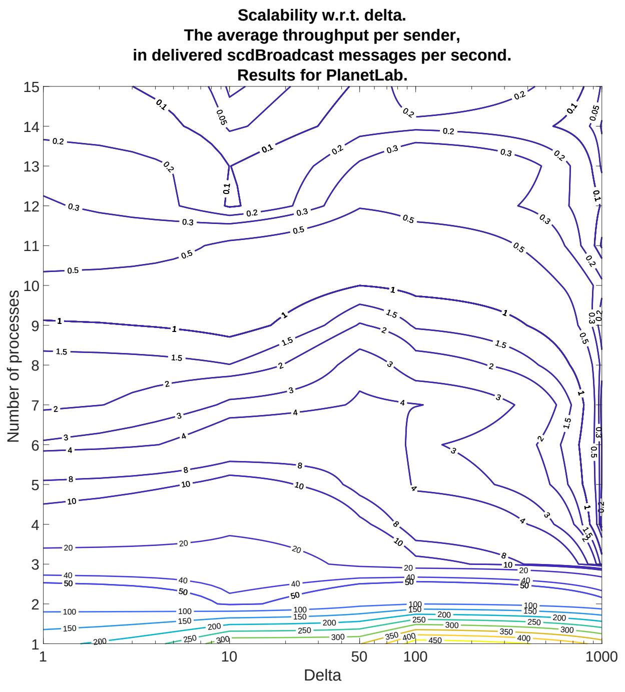


Figure 5.26: Scalability of delta w.r.t. scdBroadcast.

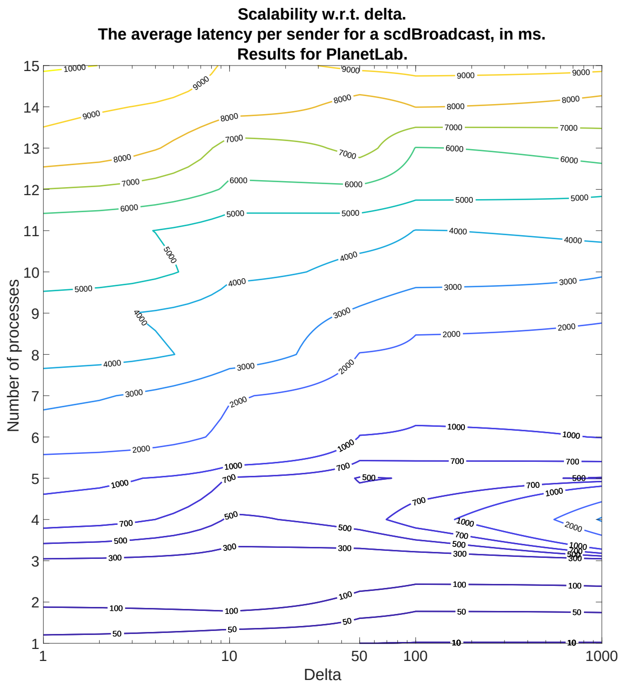


Figure 5.27: Scalability of delta w.r.t. scdBroadcast.

5.3 Applications

In this section we present our experimental results from the applications evaluated. As specified in Section 4.2 only two experiments are used to evaluate the application's performance since we are interested in the two client latencies, write and read.

5.3.1 Scalability of read operations with respect to write operations

This experiment investigates how the snapshot, or read, latency changes with respect to write operations. The experiment is conducted using a fixed number of snapshotters, which is 3 and 10 for the local and PlanetLab environment respectively. The number of writers is then varied without adding new processes, meaning that processes have more than one roles.

In an efficient implementation we would expect to see some performance penalty when increasing the number of writers, since this means more messages to process and send over the network.

As observed in Figure 5.28 we can see a slight increase for the local network for all values. However the trend is not as clear for PlanetLab, on average there is a slight increase but there are some fluctuations. For more writers than 8 the performance impact grows faster and for 2 and 6 number of writers the performance is decreased.

This could be because we randomly pick the physical PlanetLab machines from a set of 15 which vary in computational and network performance. The application needs to wait for a majority before finishing a snapshot operation, which means that if the machines used in an execution consists of a majority of slow or fast machines the performance will be impacted.

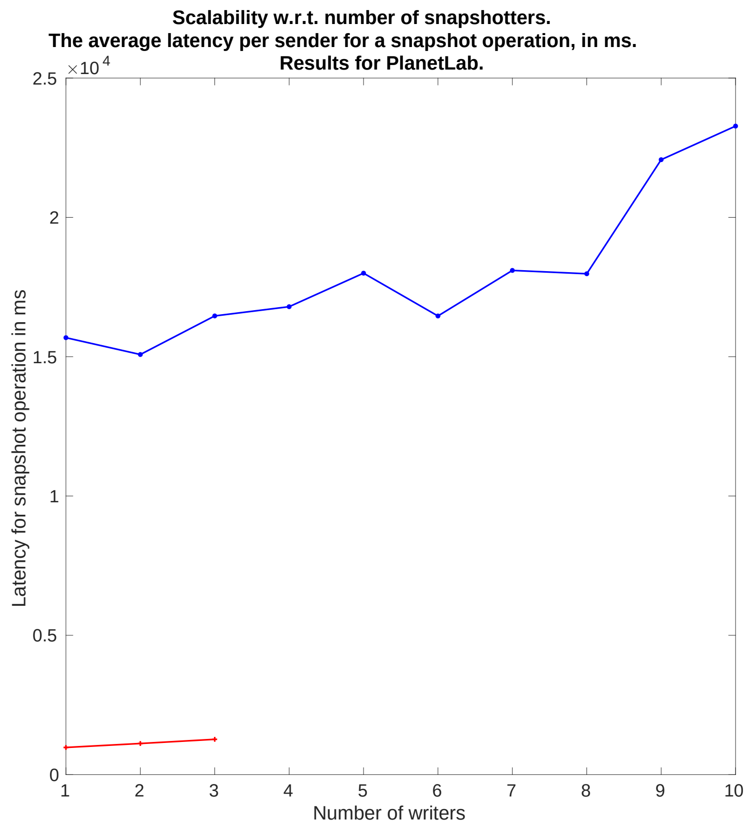


Figure 5.28: Scalability of delta w.r.t. snapshot operation.

5.3.2 Scalability of write operations with respect to read operations

This experiment investigates how the write latency changes with respect to snapshot, or read, operations. The experiment is conducted using a fixed number of writers, which is 3 and 10 for the local and PlanetLab environment respectively. The number of snapshotters is then varied without adding new processes, meaning that processes have more than one roles.

To complete a write operation, the process needs to send and deliver two messages in succession. This means that we need to wait for two round trips to a majority of the system. We expect there to be a performance penalty when increasing the number of snapshotters as this introduces more messages.

As illustrated in Figure 5.29, the latency for local network is higher than the PlanetLab environment when we have 1 and 3 snapshotters, which can be explained by the higher computational power of PlanetLab machines. The latency increases a lot when the number of snapshotters grows past 3. This could be explained by the rate of which we invoke operations, snapshot operations are invoked more frequently than write operations when a process has both roles.

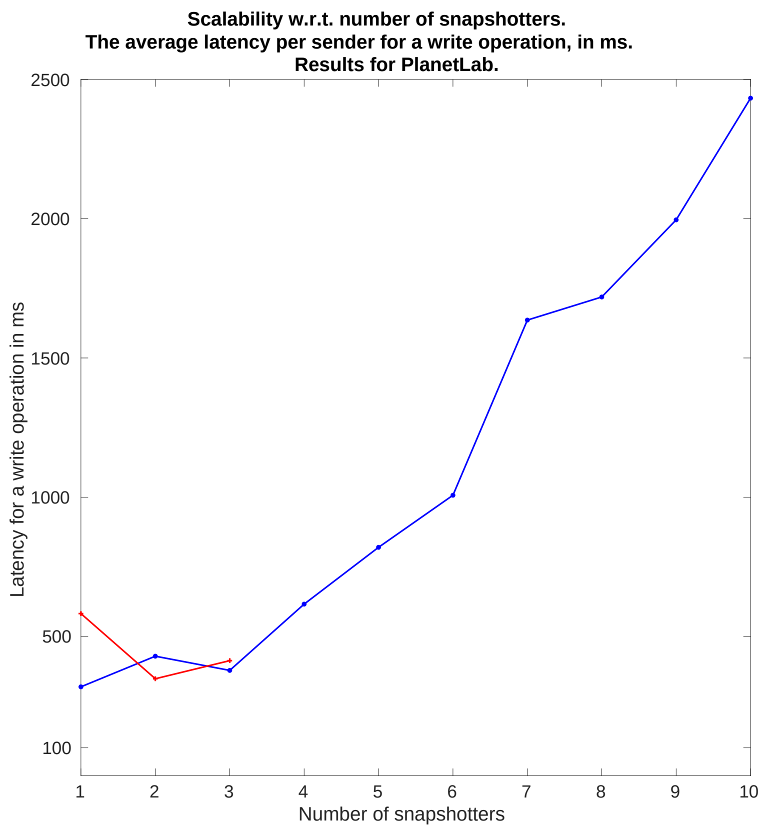


Figure 5.29: Scalability of delta w.r.t. snapshot operation.

6 Discussion

Before presenting the summary and our conclusions we would like to remind the reader that our implementation is a single threaded application since this project focuses on the correctness and validation of the studied protocols. This leads to some inefficiencies, for instance the responsibility of polling messages from the Linux socket and pushing incoming messages to a intermediary buffer could be done using a separate thread to improve performance. There are more examples of this in the implementation, therefore the implementation should not be viewed as an efficient implementation. This is especially true when the message buffer grows, since the single thread needs to execute both protocols logic in order to complete one iteration. Therefore the authors recommend to refactor the implementation into a multi-threaded application, which it used to be before it was merged in this project to verify correctness. Please refer to `www.self-stabilizing-cloud.net` for the source code of the old multi-threaded version and the projects single-threaded implementation.

6.1 Summary

In general the results looked as we would expect, the local environment outperformed PlanetLab when link latency was the dominant factor for performance while when computational power was more important PlanetLab performed better. The lower URB protocol delivered messages faster and at a faster rate than the higher SCD protocol, which of course makes sense since SCD uses multiple URB messages to deliver a single SCD message. Furthermore, the recovery period after a transient fault is fairly short and has negligible impact on performance. The snapshot application also performed as we would expect. A read/snapshot operation only requires a single SCD message to be delivered, making the latency of a read/snapshot operation very close to the SCD message latency. Similarly the write operation requires two SCD messages to be delivered in succession, the second message being broadcasted after the first delivery, which gave us a faster growth of latency in the evaluation.

6.2 Conclusion

Self-stabilization provides fault-tolerance in the presence of transient faults. This allows systems to recover in a finite amount of time if something goes wrong. However, it is hard to simulate transient faults since they can have many forms and can occur at any time during the execution. This makes it hard for developers to test their self-stabilizing implementations, creating tests to cover all possible transient faults is impossible. What the authors recommend is to set up a testing suite that automatically runs the application all day and night while randomly introducing message delays and changing local variables. This together with an automatic reporting system when the execution fails should provide the developers with more cases which in turn should help finding problems earlier.

6.3 Future work

The *evaluator* component used in the project can be extended to automatically run any group of tests repeatedly and report in case of failures. As we said in the previous section this could help developers find issues with their implementations sooner.

Both studied protocols were implemented with generality in mind, this should allow developers to use these layers as building blocks in new protocols or applications. Developers can use the URB implementation to get self-stabilizing uniform reliable broadcast, similarly, SCD provides self-stabilizing set-constrained delivery to the developer.

Bibliography

[1] Mohamed Faouzi Atig and Alexander A. Schwarzmann, eds. *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers*. Vol. 11704. Lecture Notes in Computer Science. Springer, 2019. ISBN: 978-3-030-31276-3. DOI: `10.1007/978-3-030-31277-0`. URL: `https://doi.org/10.1007/978-3-030-31277-0`.

[2] Alex Auvolat, Michel Raynal, and François Taïani. "Byzantine-Tolerant Set-Constrained Delivery Broadcast". In: *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*. Ed. by Pascal Felber et al. Vol. 153. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, 6:1–6:23. ISBN: 978-3-95977-133-7. DOI: `10.4230/LIPIcs.OPODIS.2019.6`. URL: `https://drops.dagstuhl.de/opus/volltexte/2020/11792`.

[3] Marco Canini et al. "A Self-Organizing Distributed and In-Band SDN Control Plane". In: *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*. Ed. by Kisung Lee and Ling Liu. IEEE Computer Society, 2017, pp. 2656–2657. ISBN: 978-1-5386-1792-2. DOI: `10.1109/ICDCS.2017.328`. URL: `https://doi.org/10.1109/ICDCS.2017.328`.

[4] Marco Canini et al. "Renaissance:A Self-Stabilizing Distributed SDN Control Plane". In: *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*. IEEE Computer Society, 2018, pp. 233–243. ISBN: 978-1-5386-6871-9. DOI: `10.1109/ICDCS.2018.00032`. URL: `https://doi.org/10.1109/ICDCS.2018.00032`.

[5] António Casimiro, Emelie Ekenstedt, and Elad Michael Schiller. "Self-Stabilizing Manoeuvre Negotiation: The Case of Virtual Traffic Lights". In: *38th Symposium on Reliable Distributed Systems, SRDS 2019, Lyon, France, October 1-4, 2019*. IEEE, 2019, pp. 354–356. ISBN: 978-1-7281-4222-7. DOI: `10.1109/SRDS47363.2019.00048`. URL: `https://doi.org/10.1109/SRDS47363.2019.00048`.

[6] Brent Chun et al. "Planetlab: an overlay testbed for broad-coverage services". In: *ACM SIGCOMM Computer Communication Review* 33.3 (2003), pp. 3–12.

[7] Edsger W Dijkstra. "Self-stabilization in spite of distributed control". In: *Selected writings on computing: a personal perspective*. Originally published in 1973. Springer, 1982, pp. 41–46.

[8] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. “On the minimal synchronism needed for distributed consensus”. In: *Journal of the ACM (JACM)* 34.1 (1987), pp. 77–97.

[9] Shlomi Dolev. *Self-stabilization.* MIT press, 2000.

[10] Shlomi Dolev, Ronen I. Kat, and Elad Michael Schiller. “When consensus meets self-stabilization”. In: *J. Comput. Syst. Sci.* 76.8 (2010), pp. 884–900. DOI: `10.1016/j.jcss.2010.05.005`. URL: `https://doi.org/10.1016/j.jcss.2010.05.005`.

[11] Shlomi Dolev, Omri Liba, and Elad Michael Schiller. “Self-stabilizing Byzantine Resilient Topology Discovery and Message Delivery”. In: *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings.* Ed. by Teruo Higashino et al. Vol. 8255. Lecture Notes in Computer Science. Springer, 2013, pp. 351–353. ISBN: 978-3-319-03088-3. DOI: `10.1007/978-3-319-03089-0\_27`. URL: `https://doi.org/10.1007/978-3-319-03089-0%5C_27`.

[12] Shlomi Dolev, Omri Liba, and Elad Michael Schiller. “Self-stabilizing Byzantine Resilient Topology Discovery and Message Delivery - (Extended Abstract)”. In: *Networked Systems - First International Conference, NETYS 2013, Marrakech, Morocco, May 2-4, 2013, Revised Selected Papers.* Ed. by Vincent Gramoli and Rachid Guerraoui. Vol. 7853. Lecture Notes in Computer Science. Springer, 2013, pp. 42–57. ISBN: 978-3-642-40147-3. DOI: `10.1007/978-3-642-40148-0\_4`. URL: `https://doi.org/10.1007/978-3-642-40148-0%5C_4`.

[13] Shlomi Dolev, Thomas Petig, and Elad Michael Schiller. “Self-Stabilizing and Private Distributed Shared Atomic Memory in Seldomly Fair Message Passing Networks”. In: *CoRR* abs/1806.03498 (2018). arXiv: `1806.03498`. URL: `http://arxiv.org/abs/1806.03498`.

[14] Shlomi Dolev and Elad Schiller. “Communication Adaptive Self-Stabilizing Group Membership Service”. In: *IEEE Trans. Parallel Distrib. Syst.* 14.7 (2003), pp. 709–720. DOI: `10.1109/TPDS.2003.1214322`. URL: `https://doi.org/10.1109/TPDS.2003.1214322`.

[15] Shlomi Dolev and Elad Schiller. “Self-stabilizing group communication in directed networks”. In: *Acta Informatica* 40.9 (2004), pp. 609–636. DOI: `10.1007/s00236-004-0143-1`. URL: `https://doi.org/10.1007/s00236-004-0143-1`.

[16] Shlomi Dolev, Elad Schiller, and Jennifer L. Welch. “Random Walk for Self-Stabilizing Group Communication in Ad Hoc Networks”. In: *IEEE Trans. Mob. Comput.* 5.7 (2006), pp. 893–905. DOI: `10.1109/TMC.2006.104`. URL: `https://doi.org/10.1109/TMC.2006.104`.

[17] Shlomi Dolev et al. “Autonomous virtual mobile nodes”. In: *Proceedings of the DIALM-POMC Joint Workshop on Foundations of Mobile Computing, Cologne, Germany, September 2, 2005.* Ed. by Suman Banerjee and Samrat Ganguly. ACM, 2005, pp. 62–69. ISBN: 1-59593-092-2. DOI: `10.1145/1080810.1080821`. URL: `https://doi.org/10.1145/1080810.1080821`.

[18] Shlomi Dolev et al. “Game authority for robust and scalable distributed selfish computer systems”. In: *theor. Comput. Sci.* 411.26-28 (2010), pp. 2459–2466.

DOI: 10.1016/j.tcs.2010.02.014. URL: https://doi.org/10.1016/j.tcs.2010.02.014.

[19] Shlomi Dolev et al. "Practically-self-stabilizing virtual synchrony". In: *J. Comput. Syst. Sci.* 96 (2018), pp. 50–73. DOI: 10.1016/j.jcss.2018.04.003. URL: https://doi.org/10.1016/j.jcss.2018.04.003.

[20] Shlomi Dolev et al. "Rationality Authority for Provable Rational Behavior". In: *Algorithms, Probability, Networks, and Games - Scientific Papers and Essays Dedicated to Paul G. Spirakis on the Occasion of His 60th Birthday.* Ed. by Christos D. Zaroliagis, Grammati E. Pantziou, and Spyros C. Kontogiannis. Vol. 9295. Lecture Notes in Computer Science. Springer, 2015, pp. 33–48. ISBN: 978-3-319-24023-7. DOI: 10.1007/978-3-319-24024-4_5. URL: https://doi.org/10.1007/978-3-319-24024-4%5C_5.

[21] Shlomi Dolev et al. "Self-Stabilizing Automatic Repeat Request Algorithms for (Bounded Capacity, Omitting, Duplicating and non-FIFO) Computer Networks". In: *CoRR* abs/2006.05901 (2020). arXiv: 2006.05901. URL: https://arxiv.org/abs/2006.05901.

[22] Shlomi Dolev et al. "Self-stabilizing End-to-End Communication in (Bounded Capacity, Omitting, Duplicating and non-FIFO) Dynamic Networks - (Extended Abstract)". In: *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium, SSS 2012, Toronto, Canada, October 1-4, 2012. Proceedings.* Ed. by Andréa W. Richa and Christian Scheideler. Vol. 7596. Lecture Notes in Computer Science. Springer, 2012, pp. 133–147. ISBN: 978-3-642-33535-8. DOI: 10.1007/978-3-642-33536-5_14. URL: https://doi.org/10.1007/978-3-642-33536-5%5C_14.

[23] Shlomi Dolev et al. "Self-stabilizing Reconfiguration". In: *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings.* 2017, pp. 51–68. DOI: 10.1007/978-3-319-59647-1_5. URL: https://doi.org/10.1007/978-3-319-59647-1%5C_5.

[24] Shlomi Dolev et al. "Strategies for repeated games with subsystem takeovers implementable by deterministic and self-stabilizing automata". In: *IJAACS* 4.1 (2011), pp. 4–38. DOI: 10.1504/IJAACS.2011.037747. URL: https://doi.org/10.1504/IJAACS.2011.037747.

[25] Shlomi Dolev et al. "Virtual Mobile Nodes for Mobile Ad Hoc Networks". In: *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings.* Ed. by Rachid Guerraoui. Vol. 3274. Lecture Notes in Computer Science. Springer, 2004, pp. 230–244. ISBN: 3-540-23306-7. DOI: 10.1007/978-3-540-30186-8_17. URL: https://doi.org/10.1007/978-3-540-30186-8%5C_17.

[26] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. "Impossibility of distributed consensus with one faulty process". In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382.

[27] D. Frisk. "A Chalmers University of Technology Master's thesis template for LaTeX". 2016, Unpublished.

[28] Chryssis Georgiou, Oskar Lundström, and Elad Michael Schiller. "Self-Stabilizing Snapshot Objects for Asynchronous Failure-Prone Networked Systems". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Com-*

puting, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019. Ed. by Peter Robinson and Faith Ellen. ACM, 2019, pp. 209–211. ISBN: 978-1-4503-6217-7. DOI: 10.1145/3293611.3331584. URL: https://doi.org/10.1145/3293611.3331584.

[29] Chryssis Georgiou, Oskar Lundström, and Elad Michael Schiller. "Self-stabilizing Snapshot Objects for Asynchronous Failure-Prone Networked Systems". In: *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers.* Ed. by Mohamed Faouzi Atig and Alexander A. Schwarzmann. Vol. 11704. Lecture Notes in Computer Science. Springer, 2019, pp. 113–130. ISBN: 978-3-030-31276-3. DOI: 10.1007/978-3-030-31277-0_8. URL: https://doi.org/10.1007/978-3-030-31277-0%5C_8.

[30] Chryssis Georgiou et al. "Self-stabilization Overhead: A Case Study on Coded Atomic Storage". In: *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers.* Ed. by Mohamed Faouzi Atig and Alexander A. Schwarzmann. Vol. 11704. Lecture Notes in Computer Science. Springer, 2019, pp. 131–147. ISBN: 978-3-030-31276-3. DOI: 10.1007/978-3-030-31277-0_9. URL: https://doi.org/10.1007/978-3-030-31277-0%5C_9.

[31] Chryssis Georgiou et al. "Self-stabilization Overhead: an Experimental Case Study on Coded Atomic Storage". In: *CoRR* abs/1807.07901 (2018). arXiv: 1807.07901. URL: http://arxiv.org/abs/1807.07901.

[32] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: up and running: dive into the future of infrastructure.* " O'Reilly Media, Inc.", 2017.

[33] Jaap-Henk Hoepman et al. "Secure and Self-stabilizing Clock Synchronization in Sensor Networks". In: *Stabilization, Safety, and Security of Distributed Systems, 9th International Symposium, SSS 2007, Paris, France, November 14-16, 2007, Proceedings.* Ed. by Toshimitsu Masuzawa and Sébastien Tixeuil. Vol. 4838. Lecture Notes in Computer Science. Springer, 2007, pp. 340–356. ISBN: 978-3-540-76626-1. DOI: 10.1007/978-3-540-76627-8_26. URL: https://doi.org/10.1007/978-3-540-76627-8%5C_26.

[34] Jaap-Henk Hoepman et al. "Secure and self-stabilizing clock synchronization in sensor networks". In: *Theor. Comput. Sci.* 412.40 (2011), pp. 5631–5647. DOI: 10.1016/j.tcs.2010.04.012. URL: https://doi.org/10.1016/j.tcs.2010.04.012.

[35] Damien Imbs et al. "Set-Constrained Delivery Broadcast: Definition, Abstraction Power, and Computability Limits". In: *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018.* Ed. by Paolo Bellavista and Vijay K. Garg. ACM, 2018, 7:1–7:10. ISBN: 978-1-4503-6372-3. DOI: 10.1145/3154273.3154296. URL: https://doi.org/10.1145/3154273.3154296.

[36] Damien Imbs et al. "Set-constrained delivery broadcast: Definition, abstraction power, and computability limits". In: *Proceedings of the 19th International Conference on Distributed Computing and Networking.* 2018, pp. 1–10.

[37] Pierre Leone, Marina Papatriantafilou, and Elad Michael Schiller. "Relocation Analysis of Stabilizing MAC Algorithms for Large-Scale Mobile Ad Hoc

Networks". In: *Algorithmic Aspects of Wireless Sensor Networks, 5th International Workshop, ALGOSENSORS 2009, Rhodes, Greece, July 10-11, 2009. Revised Selected Papers.* Ed. by Shlomi Dolev. Vol. 5804. Lecture Notes in Computer Science. Springer, 2009, pp. 203–217. ISBN: 978-3-642-05433-4. DOI: 10.1007/978-3-642-05434-1_21. URL: https://doi.org/10.1007/978-3-642-05434-1%5C_21.

[38] Pierre Leone and Elad Schiller. "Self-Stabilizing TDMA Algorithms for Dynamic Wireless Ad Hoc Networks". In: *Int. J. Distributed Sens. Networks* 9 (2013). DOI: 10.1155/2013/639761. URL: https://doi.org/10.1155/2013/639761.

[39] Pierre Leone and Elad Michael Schiller. "Interacting Urns Processes for Clustering of Large-Scale Networks of Tiny Artifacts". In: *Int. J. Distributed Sens. Networks* 6.1 (2010). DOI: 10.1155/2010/936195. URL: https://doi.org/10.1155/2010/936195.

[40] Pierre Leone and Elad Michael Schiller. "Interacting urns processes: for clustering of large-scale networks of tiny artifacts". In: *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008.* Ed. by Roger L. Wainwright and Hisham Haddad. ACM, 2008, pp. 2046–2051. ISBN: 978-1-59593-753-7. DOI: 10.1145/1363686.1364182. URL: https://doi.org/10.1145/1363686.1364182.

[41] Pierre Leone et al. "Chameleon-MAC: Adaptive and Self-* Algorithms for Media Access Control in Mobile Ad Hoc Networks". In: *Stabilization, Safety, and Security of Distributed Systems - 12th International Symposium, SSS 2010, New York, NY, USA, September 20-22, 2010. Proceedings.* Ed. by Shlomi Dolev et al. Vol. 6366. Lecture Notes in Computer Science. Springer, 2010, pp. 468–488. ISBN: 978-3-642-16022-6. DOI: 10.1007/978-3-642-16023-3_37. URL: https://doi.org/10.1007/978-3-642-16023-3%5C_37.

[42] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. "Self-stabilizing Set-constraint Delivery Broadcast". In: *40th IEEE International Conference on Distributed Computing Systems, ICDCS.* 2020.

[43] Oskar Lundström, Michel Raynal, and Elad Michael Schiller. "Self-stabilizing Uniform Reliable Broadcast". In: *CoRR* abs / 2001.03244 (2020). Also appears in NETYS' 2020. URL: https://arxiv.org/abs/2001.03244.

[44] Nicholas D Matsakis and Felix S Klock. "The rust language". In: *ACM SIGAda Ada Letters* 34.3 (2014), pp. 103–104.

[45] Mohamed Mustafa et al. "Autonomous TDMA Alignment for VANETs". In: *Proceedings of the 76th IEEE Vehicular Technology Conference, VTC Fall 2012, Quebec City, QC, Canada, September 3-6, 2012.* IEEE, 2012, pp. 1–5. ISBN: 978-1-4673-1880-8. DOI: 10.1109/VTCFall.2012.6399373. URL: https://doi.org/10.1109/VTCFall.2012.6399373.

[46] Axel Niklasson. *plcli.* https://github.com/axelniklasson/plcli. 2019.

[47] Thomas Petig, Elad Schiller, and Philippas Tsigas. "Self-stabilizing TDMA algorithms for wireless ad-hoc networks without external reference". In: *13th Annual Mediterranean Ad Hoc Networking Workshop, MED-HOC-NET 2014, Piran, Slovenia, June 2-4, 2014.* IEEE, 2014, pp. 87–94. ISBN: 978-1-4799-

5258-8. DOI: 10.1109/MedHocNet.2014.6849109. URL: https://doi.org/10.1109/MedHocNet.2014.6849109.

[48] Marco Schneider. “Self-stabilization”. In: *ACM Computing Surveys (CSUR)* 25.1 (1993), pp. 45–67.

[49] Axel Wegener et al. “Hovering Data Clouds: A Decentralized and Self-organizing Information System”. In: *Self-Organizing Systems, First International Workshop, IWSOS 2006, and Third International Workshop on New Trends in Network Architectures and Services, EuroNGI 2006, Passau, Germany, September 18-20, 2006, Proceedings*. Ed. by Hermann de Meer and James P. G. Sterbenz. Vol. 4124. Lecture Notes in Computer Science. Springer, 2006, pp. 243–247. ISBN: 3-540-37658-5. DOI: 10.1007/11822035_22. URL: https://doi.org/10.1007/11822035%5C_22.