5-9-2014

Resource Discovery and Cooperation in Decentralized Systems

Seda Davtyan

University of Connecticut, seda@engr.uconn.edu

Follow this and additional works at: http://digitalcommons.uconn.edu/dissertations

Recommended Citation
http://digitalcommons.uconn.edu/dissertations/425
Resource Discovery and Cooperation in Decentralized Systems

Seda Davtyan, Ph.D.
University of Connecticut, 2014

Massive distributed cooperative computing in networks involves marshaling large collection of network nodes possessing the necessary computational resources. The computing power of the resources is used in solving partitionable, computation intensive problems. This is referred to as Internet, or network, supercomputing. Traditional approaches to Internet supercomputing employ a master processor and many worker processors that execute a collection of tasks on behalf of the master. Despite the simplicity and advantages of centralized schemes, the master processor is a performance bottleneck and a single point of failure. Additionally, a phenomenon of increasing concern is that workers may return incorrect results.

In this thesis, we present algorithms for the problem of network supercomputing that eliminate the master and instead use decentralized approach, where workers cooperate in performing tasks. The problem is studied under a variety of failure models, and all algorithms are designed to deal with undependable and crash-prone workers. Additionally, we present an algorithm that estimates the reliability of workers.

In order for the willing nodes to act in a concerted way in decentralized systems they must first discover one another. This is the general setting of the Resource Discovery Problem (RDP), and it serves as a building block for any kind of decentralized collaborative computing.

For the resource discovery problem this thesis explores solutions that can cope with intermittent failures, and, in particular we design self-stabilizing algorithms that solve the resource discovery problem in a deterministic synchronous setting.
Resource Discovery and Cooperation in Decentralized Systems

Seda Davtyan

B.S., Applied Mathematics, Yerevan State University 2001
M.S., Informatics and Applied Mathematics, Yerevan State University 2003

A Dissertation
Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy
at the
University of Connecticut
2014
APPROVAL PAGE

Doctor of Philosophy Dissertation

Resource Discovery and Cooperation in Decentralized Systems

Presented by

Seda Davtyan, B.S., M.S.

Major Advisor

Alexander A. Shvartsman

Associate Advisor

Alexander Russell

Associate Advisor

Laurent Michel

University of Connecticut

2014
DEDICATION

I dedicate my dissertation work to my late father, Ashot Davtyan, who taught me not to be afraid of taking a harder path in life. You did not live long enough to see me graduate, but you will always be in my heart. Your love for books inspired me to start reading at a very early age. You were one of the most intelligent people that I have ever met. It is thanks to you and my mother, Anush Durukhyan, that I pursued my doctoral degree.
ACKNOWLEDGEMENTS

Approaching the end of a long journey I would like to express my gratitude to all the people that made this journey a pleasant one.

First, I could not have done it without the help and constant guidance of my major advisor, Dr. Alexander Shvartsman. He accepted me as a student and offered me an assistantship without which I could have never pursued my doctoral degree in the United States. His insightful criticism aided me in becoming a better researcher. Perhaps more importantly, he has been my inspiration and my mentor. It has been an honor knowing you and your wonderful family, especially, your lovely wife Sana. I thank you both for always making me feel welcome in your house and for making me feel as if I have a family so far from home.

I am also grateful to my associate advisors, Dr. Alexander Russell and Dr. Laurent Michel. Despite your busy schedules you have always made yourself available and assisted me in overcoming the obstacles I encountered while working on my thesis. I am privileged to have had the opportunity to work with you.

I am grateful to the National Science Foundation for supporting my research as a part of the NSF Award 1017232.

A special thank you goes to Dr. Kishori Konwar who has been a great collaborator and a friend. You encouraged me to push my boundaries and to pursue directions that I would not have otherwise. You never let me give up on any idea we have had. I thank you for your
understanding, your patience, continuous feedback, and all the fruitful discussions that we have had over these past years.

I thank all my lab-mates and friends in the United States. Since the day I arrived you made a foreign country feel like home. A special feeling of gratitude to my former lab-mate and my dear friend Dr. Nicolas Nicolau and his lovely wife, Dr. Andri Ioannou, who stand by me through thick and thin.

I also would like to thank my dear parents, brother, extended family and all my friends in Armenia. Your support and encouragement helped me immensely through this tough process. I would not be writing this thesis if my family would not have persuaded me to take this opportunity.

Most of all I would like to thank my best friend and my husband Tigran for always being by my side, supporting me, believing in me, cheering me up, and most importantly, for loving me. Your exuberant optimism overshadowed my moments of apathy, helping me to stay concentrated on the task at hand. You have more confidence in me than I will ever have.
CREDITS

This dissertation incorporates research results appearing in the following publications:

[24] This paper, corresponding to Section 3.3 of Chapter 3, is a joint work with Kishori M. Konwar and Alexander A. Shvartsman. It was given as a brief announcement at PODC’13, the 32nd ACM Symposium on Principles of Distributed Computing.

[26] This paper, corresponding to Section 3.4 of Chapter 3, is a joint work with Kishori M. Konwar and Alexander A. Shvartsman. It was presented at OPODIS’13, the 17th International Conference On Principles Of Distributed Systems.

[22] This paper, corresponding to Section 4.2 of Chapter 4, is a joint work with Kishori M. Konwar and Alexander A. Shvartsman. It was presented at OPODIS’11.

[21] This paper, corresponding to Section 4.3 of Chapter 4, is a joint work with Kishori M. Konwar, Alexander Russell, and Alexander A. Shvartsman. It was presented at ICDCN’13, the 14th International Conference On Distributed Computing and Networking. The paper was invited for the TCS Special issue on Distributed Computing.

[25] This paper, corresponding to Section 4.4 of Chapter 4, is a joint work with Kishori M. Konwar and Alexander A. Shvartsman. It was presented at ISPDC’13, the 12th International Symposium on Parallel and Distributed Computing.

[23] This paper, corresponding to parts of Section 4.4, is a joint work with Kishori M. Konwar and Alexander A. Shvartsman. It was given as a brief announcement at PODC’12.
TABLE OF CONTENTS

Chapter 1: Introduction 1

1.1 Motivation .................................................. 1
1.2 Background .................................................. 4
1.3 Thesis Contributions ......................................... 6
   1.3.1 Resource Discovery Problem ............................. 7
   1.3.2 Decentralized Network Supercomputing .................. 9
   1.3.3 Thesis Organization .................................... 10

Chapter 2: Related Work 11

2.1 Resource Discovery ........................................... 11
   2.1.1 Static Settings ......................................... 12
   2.1.2 Dynamic Systems ...................................... 15
   2.1.3 Summary ................................................ 20
2.2 Network Supercomputing ...................................... 20
   2.2.1 Internet Supercomputing ................................ 21
   2.2.2 Related Work ......................................... 25
   2.2.3 Summary ................................................ 29

Chapter 3: Resource Discovery 30

3.1 System Model and Definitions ............................... 31
   3.1.1 Measures of efficiency ................................ 37
3.2 Self-Stabilizing Resource Discovery Algorithms ............ 38
3.3 Resource Discovery with Constrained Multicast ................................................. 43
  3.3.1 Algorithm Analysis ....................................................................................... 47
3.4 Resource Discovery with Aggressive Broadcast .................................................. 63
  3.4.1 Algorithm Analysis ....................................................................................... 67
3.5 Closing Remarks ................................................................................................. 80

Chapter 4: Network Supercomputing ........................................................................... 82
  4.1 System Model and Definitions ........................................................................... 83
  4.2 Network Supercomputing Without Centralized Control ..................................... 88
    4.2.1 Description of Algorithm NS ..................................................................... 89
    4.2.2 Analysis of Algorithm NS ......................................................................... 92
    4.2.3 Tolerating Crash Failures .......................................................................... 98
  4.3 Decentralized NSP Under Failure Models \(\langle F^{fp}, E_1 \rangle\) and \(\langle F^{p\ell}, E_2 \rangle\) ................................................................. 103
    4.3.1 Description of Algorithm DAKS ............................................................... 104
    4.3.2 Analysis of Algorithm DAKS ................................................................... 108
    4.3.3 Finalizing Algorithm Parameterization ...................................................... 136
  4.4 Estimating Reliability of Workers ..................................................................... 136
    4.4.1 Description of Algorithm \(A_{est}\) ................................................................. 138
    4.4.2 Estimation of Processor Reliability ........................................................... 143
    4.4.3 Analysis of Algorithm \(A_{est}\) .................................................................... 147
  4.5 Closing Remarks ............................................................................................... 162

Chapter 5: Summary and Future Directions ............................................................... 163
  5.1 Summary ........................................................................................................... 163
5.2 Future Directions ................................................... 165

5.2.1 Resource Discovery Problem ................................. 165

5.2.2 Network Supercomputing .................................... 167

5.2.3 Other Future Directions ...................................... 170

Bibliography .......................................................... 180
# LIST OF FIGURES

1. Example of a Connectivity Graph .................................................. 33
2. Instance of a Legitimate Configuration for RDP .............................. 35
3. Data types, signature, state, and transitions of \( Channel_{i,j} \) at node \( i \in V \) .......................... 37
4. Data types, signature, and state of \( RD - M_i \) at node \( i \in V \) .................. 44
5. Transitions of \( RD - M_i \) at node \( i \in V \) ........................................ 45
6. Data types, signature, and state of \( RD - B_i \) at node \( i \in V \) .................. 64
7. Transitions of \( RD - B_i \) at node \( i \in V \) ........................................ 65
8. Algorithm \( NS \) for \( t = n \); code at processor \( i \in P \) ......................... 90
9. Algorithm \( DAKS \) for \( t = n \); code at processor \( i \in P \) ......................... 105
10. Algorithm \( A_{est} \) at processor \( i \in P \) ........................................ 139
11. Estimation of the probabilities for each \( j \in P \) ............................... 140
12. The Stopping Rule Algorithm (SRA) for estimating \( \mu_Z \) ................ 145
Chapter 1

Introduction

This thesis studies the problem of collaborative computing in decentralized systems given that the processors are undependable and susceptible to crashes. As a building block for collaborative computing we also study the problem of discovering the resources that are willing to collaborate with each other on solving computationally intensive problems. In this chapter we first present the motivation for this work. We then present a brief review of the prior research done in this area. And finally, we identify the open problems derived from the prior work and present the research contributions of this dissertation.

1.1 Motivation

A large collection of networked computers may need to cooperate in implementing a distributed system, for example, to provide a shared data service, or to perform a set of tasks. Cooperative network computing is becoming increasingly popular for harnessing the power of the global Internet computing platform, e.g., [3, 5, 1, 2]. The necessary first step in such settings is to discover the relevant resources in the network. This step can be formulated as the
Resource Discovery Problem (RDP), where each willing resource must find all other available
resources.

Similar problems appear in peer-to-peer Internet systems where a large number of users
share files without having to rely on centralized servers. Such systems are highly dynamic, with
nodes constantly joining and leaving the network, making it desirable to efficiently discover
the nodes that are willing to cooperate.

Since the problem is employed over the Internet it is important that the algorithms solving
RDP are fault-tolerant and are able to deal with situations where nodes may join and leave the
computation. In particular it is important to develop self-stabilizing solutions that are able to
deal with intermittent failures. Here the algorithm must automatically bring a system into a
legitimate state in spite of transient failures.

After discovering the computational resources, the resources can be used in solving par-
titionable, computation-intensive problems. A large number of computers connected over the
Internet cooperatively working on computationally intensive tasks is referred to as Internet
Supercomputing, or network supercomputing. A typical Internet supercomputer consists of a
master computer and a large number of computers called workers. Applications submit the
tasks to be performed to the master that in turn directs the workers to perform the tasks and
then collects the results. Several Internet supercomputers are in existence today. For instance,
Internet PrimeNet Server encompasses about 30,000 computers, achieving throughput of over
1 teraflop [3], and even higher throughput is reported by the SETI@home project [5].

A major concern in network supercomputing is the correctness of the results returned by
the workers. While most workers may be reliable, workers have been known to return incorrect
results. This may be due to unintended failures caused (e.g., by over-clocked processors), or
the workers claiming to have performed assigned work so as to obtain incentives, such as getting a higher rank on the SETI@home list of contributed units of work. Prior research developed models and algorithms for network supercomputing, e.g., [36, 37, 59]. Using a variety of probabilistic failure models, the goal is to design algorithms that correctly perform all tasks with high probability (whp). One drawback of the existing approach is the assumption of the existence of a reliable master processor. Despite the simplicity and advantages of this approach, the master is a single point of failure. The master is further assumed to be able to keep up with the large number of results returned by the workers, making such systems poorly scalable. In any message passing system, during some short time interval, a network node can maintain only a limited number of connections. Centralized schemes are not suitable for big data processing that often involves a large amount of input data and also produces a large amount of output data. As an example, consider the applications in molecular biology that require large reference databases of gene models or annotated protein sequences, and large sets of unknown protein sequences [46]. Dealing with voluminous input and output data requires a large scale platform and a distributed file system providing the necessary computational power and storage. Therefore, a more scalable approach is to use a decentralized system, where the input is distributed and, once the processing is complete, the output is distributed across multiple nodes. Thus scalable distributed (i.e., not centralized) solutions are desirable. In other words, it is important to remove the assumption of an infallible and bandwidth-unlimited master processor and consider fully decentralized solutions using just the cooperating workers.
1.2 Background

Resource discovery problem was first introduced by Harchol-Balter, Leighton, and Lewin [47] in the context of an application at Akamai Technologies with the motivation to build an Internet-wide content-distribution system that would speed up the access to web pages of major content providers. Before the computing nodes start cooperating in implementing the service, they need to find each other. Resource discovery algorithms need to be efficient in terms of time and network communications. The authors presented several algorithms for the resource discovery problem in synchronous message passing systems that work for weakly connected initial graphs. The most efficient algorithm, called Name-Dropper, is a randomized algorithm with time complexity $O(\log^2 n)$, message complexity $O(n \log n)$, and communication complexity $O(n^2 \log^3 n)$, all whp. Here $n$ is the number of participating nodes. Name-Dropper algorithm is very simple, in each round, each node choses a random neighbor and shares its entire knowledge of the network with it, e.g., the IP addresses of the nodes that it knows. The time required for the entire graph to become complete is nearly optimal, since, as they observe, the lower bound on the time complexity of any algorithm solving the resource discovery problem is $\Omega(\log n)$.

Kutten, Peleg, and Vishkin [61] provided a very efficient deterministic algorithm for the resource discovery problem in the same model as [47], with improved time, message, and communication complexities. The time complexity of the algorithm is $O(\log n)$, the message complexity is $O(n \log n)$, and the communication complexity is $O(|E_0| \log^2 n)$, where $E_0$ is the set of edges in the initial graph.
Edsger W. Dijkstra introduced *self-stabilization* in 1973 [27]. He considered a ring, where each node corresponds to a machine. For each machine one or more so-called “privileges” are defined, i.e., boolean functions of its own state and the states of its neighbors; when such a boolean function is true, then the privilege is “present”. More than one privilege can be present at a time. Dijkstra assumed an existence of a central daemon that can select one of the privileges present. The machine enjoying the selected privilege will then make its “move”, i.e., it is brought into a new state that is a function of its old state and the states of its neighbors. If for such a machine more than one privilege is present, the new state may also depend on the selected privilege. After completion of the move, the daemon will select a new privilege. There is a global criterion, telling whether the system as a whole is in a “legitimate” state. Based on Dijkstra’s definition it is required that: (1) in each legitimate state one or more privileges are present; (2) in each legitimate state each possible move brings the system to a legitimate state; (3) each privilege must be present in at least one legitimate state; and (4) for any pair of legitimate states there exists a sequence of moves transferring the system from one state to the other. He then proceeded to define a system to be *self-stabilizing* if and only if, regardless of the initial state and regardless of the privilege selected each time for the next move, at least one privilege is always present and the system is guaranteed to find itself in a legitimate state after a finite number of moves.

Fernández, López, Santos, and Georgiou [36] abstracted the problem of massive distributed computation in the form of a distributed system consisting of a fail-free *master* processor and a collection of *n* processors, called *workers*, that can execute tasks. They explored ways of improving the quality of the results obtained from untrusted workers in the settings where a bandwidth-unlimited and infallible master is coordinating the workers. They assume that the
service provided by the workers is not free. The master is charged a work-unit for each task assigned to a worker. The goal is for the master computer to accept the correct values of the tasks \( \text{whp} \), with the smallest amount of work (number of workers the master assigns a task). They consider two ways of bounding the number of faulty processors: (1) the maximum number of workers that may fail is bounded by \( f < \frac{n}{2} \), and (2) all processors are faulty with probability \( p < \frac{1}{2} \), independently of each other.

Konwar, Rajasekaran, and Shvartsman [59] extended [36] by removing the assumption that the number of faulty processors or the probability of a processor acting maliciously is known to the master. They consider a synchronous model of computation where processors communicate by exchanging authenticated messages. For \( n \) processors and \( n \) tasks they consider two failure models: (1) model \( \mathcal{F}_a \), where \( 0 < f < \frac{1}{2} \) fraction of the workers provide faulty results with probability \( 0 < p < \frac{1}{2} \), given that the master has no a priori knowledge of the values of \( p \) and \( f \), and (2) model \( \mathcal{F}_b \), where at most \( 0 < f < \frac{1}{2} \) fraction of the workers can reply with arbitrary results and the rest reply with incorrect results with probability \( p \), \( 0 < p < \frac{1}{2} \), where the master knows the values of \( p \) and \( f \). To estimate the values of \( p \) and \( f \), in the model where the master is not aware of those values, they use the stopping rule algorithm by Dagum et al. [19] and provide an \((\varepsilon, \delta)\)-approximation of \( p \) and \( f \), for any \( 0 < \delta < 1 \) and \( \varepsilon > 0 \).

### 1.3 Thesis Contributions

This thesis aims to study the problem of network supercomputing in decentralized systems under a variety of failure models. Resource discovery problem is a necessary first step for any kind of network supercomputing, and in particular, it is most necessary in the context of decentralized systems. The latter is because there is no notion of a unique server, that can be notified
about a join by a newly arriving node. Instead a new node can join any participating node in the system and all nodes should learn the identity of a newly added node. The first problem area of the thesis is to develop fault-tolerant solutions for the resource discovery problem in dynamic systems, with nodes constantly joining and leaving the network. Such a solution should be able to deal with intermittent failures of processors and should gracefully handle topological changes. In particular, we explore self-stabilizing solutions for the resource discovery problem.

The second problem area focuses on utilization of the discovered resources, and in particular, the problem of Network Supercomputing. We concentrate on addressing the drawbacks of the previously studied master-worker approach, under different probabilistic models. Below we give a more detailed summary of thesis contributions.

1.3.1 Resource Discovery Problem

Given that solutions for the resource discovery problem [47, 61] are deployed over the Internet, we are interested in solutions that are fault-tolerant and can handle changes without any outside intervention. In particular, in this thesis, we investigate self-stabilizing solutions for the resource discovery problem. A self-stabilizing solution does not require any initialization of the local state variables. A node can join and leave the computation, without any global disruption.

We pursue a rigorous approach to presenting and analyzing self-stabilizing algorithms. We are interested in presenting our algorithms using Timed Input/Output Automata (TIOA) formalism [55]. We formalize the properties of self-stabilizing solutions (in terms of closure and convergence conditions [10]). Our formalization enables one to reason rigorously about
algorithms solving the problem. We believe that our approach can provide valuable tools for methodical study of self-stabilizing algorithms.

In Chapter 3 we present self-stabilizing algorithms that solve the Resource Discovery Problem (RDP) in deterministic synchronous settings. Following [47], the algorithmic approach is formulated in terms of evolving knowledge graphs, where vertices represent the participating network nodes, and edges represent one node’s knowledge about another. We assume that the nodes are subject to arbitrary perturbations to their local (volatile) states; this includes arbitrary patterns of crash and restart events that occur in matched pairs, with the associated corruption of local states. We further assume that a corrupted variable may contain a value that is syntactically indistinguishable from a valid value. This is in contrast with some works in self-stabilization, where failures cause erasures of variable values, making such failures easily detectable, cf. [31]. Other works, e.g., [69], assume that any node identifier must represent an actual node in the system. The assumption that a corrupted value may be syntactically indistinguishable from a valid value raises additional challenges. This is because a node has no way of detecting an occurrence of a transient failure by verifying its own local state, or the local states of its neighbors. To ensure the convergence of the algorithm we employ a verification mechanism in every round. The latter is done by communicating with a subset of known nodes in every round and employing an acknowledgement mechanism that allows us to confirm the identities of all nodes in question, where the size of the subset depends on the particular algorithm. Notice that at no point a node is aware of the cessation of transient failures, and hence, it cannot rely on its local knowledge, neither it can rely on the information transmitted to it through the messages received. Hence, if the number of nodes to be verified in every round is
large, e.g., proportional to the local knowledge, then this may result in a high message complexity. We develop two algorithms for the problem that show the tradeoffs between time and communication complexities, the faster the algorithm terminates the more information needs to be communicated within a round.

To summarize, we develop fault-tolerant and self-stabilizing algorithms for the resource discovery problem, where the convergence time of one of the algorithms is asymptotically optimal. In fact, the convergence time of our algorithm is worse that the optimal time complexity by only a factor of 2. Here the convergence time refers to the time required for the algorithm to solve the problem after the cessation of transient failures. In addition, we propose a formal treatment of both the problem and the algorithms, that allows us to rigorously reason about the properties of the developed algorithms. We believe that our approach can provide valuable tools for methodical study of self-stabilizing algorithms in general.

1.3.2 Decentralized Network Supercomputing

As we already mentioned traditional centralized approaches to internet supercomputing, e.g., [59, 36], assume the existence of an infallible and bandwidth unlimited master processor. The master is a performance bottleneck and a single point of failure, thus it is important to develop decentralized algorithms for the problem. We develop an original approach that eliminates the master and instead uses a decentralized algorithm, where workers cooperate in performing tasks. We rigorously study this problem under a variety of failure models. In Chapter 4 we present randomized synchronous algorithms for \( n \) processors and \( t \) tasks \((t \geq n)\), and assess time, work and message complexities of the algorithms in each respective model and for each respective algorithm. Our algorithm can terminate efficiently in the presence of any
allowable number of crashes and it allows each processor to locally determine the termination time.

We assume that for the set of non-crashed processors the average probability of a processor returning a bogus result is inferior to $\frac{1}{2}$. In order to diverge from the latter assumption, and allow the average probability of non-crashed processors returning bogus results to become greater than $\frac{1}{2}$, the algorithm should be able to determine the probabilities of workers returning incorrect result. To enable a solution in the setting where the probabilities are not known, and moreover, the average probability is not bounded, we develop an algorithm that estimates the reliability of workers without any prior knowledge of the probabilities $\{p_i\}, \ i \in P$, where $P$ is the set of participating processors.

1.3.3 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2 we present related work. In Chapter 3 we present self-stabilizing algorithms for the resource discovery problem. In Chapter 4 we present decentralized algorithms for solving the problem of network supercomputing under various failure models. Additionally, we present an algorithm for estimating the probabilities of processors returning wrong results. We conclude in Chapter 5, where we provide a summary and discuss our future research goals.
Chapter 2

Related Work

This chapter presents the current research in distributed systems regarding the cooperative computing in networks, which involves resource discovery, followed by the cooperation of the discovered resources to perform computation-intensive problems.

In Section 2.1 we give an overview of the current research related to resource discovery problem, both in static and dynamic settings. In Section 2.2 we provide an overview of the related research with respect to network supercomputing, and discuss related research for efficiently estimating the probability of a process to return a correct result with arbitrary accuracy.

2.1 Resource Discovery

The algorithms for the resource discovery problem are most useful if they are designed to work in dynamic, rather than static, systems. In this section we give an overview of how the research evolved from designing algorithms for the RDP in static settings to designing algorithms in dynamic settings. Additionally, we provide an overview of the research done with
respect to self-stabilizing algorithms for the RDP and designing self-stabilizing algorithms that maintain topological structures, e.g., a spanning tree.

2.1.1 Static Settings

Harchol-Balter, Leighton, and Lewin [47] presented several algorithms for the resource discovery problem in synchronous message passing systems. The authors assumed that the communication network initially forms a weakly connected graph. Nodes $u$ and $v$ can communicate with each other only if there is a directed edge from $u$ to $v$, or in other words if $u$ “knows” $v$. The underlying communication network is modeled as a complete undirected graph over the set of participating nodes. They developed a randomized algorithm, called Name-Dropper, where all machines learn about each other within $O(\log^2 n)$ rounds with high probability (whp), where $n$ is the number of participants. The message complexity of the algorithm is $O(n \log^2 n)$, and the communication complexity of the algorithm is $O(n^2 \log^3 n)$, both whp. The Name-Dropper algorithm is very simple, in every round each node $u$ sends the set of its neighbors, the nodes that it knows about, to one randomly selected neighbor $v$. The Name-Dropper algorithm has been implemented at the Laboratory for Computer Science at MIT as a part of a project to build a large-scale distributed cache. This algorithm has been licensed to Akamai Technologies, provider of an Internet-wide content-distribution system. The authors observed that any algorithm solving the resource discovery problem takes at least $\log n$ rounds. This is because the diameter of the initial graph can be as high as $n$, and in every round the diameter of the graph can at most halve.

Law and Siu [65] gave a randomized algorithm for strongly connected initial graphs in synchronous message passing systems. The algorithm, called Absorption, has time complexity
$O(\log n)$, message complexity $O(n \log n)$, and pointer complexity (number of machine addresses passed) $O(n^2)$, all whp. The authors note that on weakly connected graphs, so-called Double-Link algorithm can be executed before Absorption, where in one time step each node sends a message about itself to each node in its known set of nodes. However, the overall message complexity would degrade to $O(n^2)$ whp. They also describe a variant of the Absorption on strongly connected graphs, called Absorption-M, that has $O(\log^2 n)$ expected time and $O(n)$ expected message complexity. The Absorption algorithm starts by partitioning a graph of $n$ nodes into clusters, where each cluster has one leader. At the beginning of the algorithm every node is the leader of its single-node cluster. All members of each cluster know their leader. The algorithm proceeds by merging the clusters until only one cluster is left, such that the leader knows about all other nodes in the network with probability 1. The ultimate leader then broadcasts the pointers to the entire network in one time step. As authors note, the major weakness of the Absorption algorithm is its reliance on the leader to distribute the knowledge.

Kutten, Peleg, and Vishkin [61] gave a very efficient deterministic algorithm for RDP in synchronous message passing systems. As in [47] they assume that the machines are logically connected via a directed graph $G(V, E)$ and a vertex $v$ can send a message to vertex $u$ only if $v$ “knows” $u$, or in other words edge $(v, u) \in E$. The time complexity of the algorithm is $O(\log n)$, the message complexity is $O(n \log n)$, and the communication complexity is $O(|E_0| \log^2 n)$, where $E_0$ is the set of edges in the initial graph. The authors assume that a deletion of even a single logical arc is not allowed. The algorithm proceeds as follows: each node starts as an active root that points to itself. The tree root can either be in passive or active state. A root $u$ becomes passive if its tree stops changing. The algorithm ensures that at any point during the execution a directed pointer graph $\mathcal{G} = (V, P)$, where $P = \{\text{ptr}_v, v \in V\}$,
is a directed forest plus self-loops at the roots. As the algorithm progresses join invitations are sent by active roots. After receiving a join invitation a star-root (all vertices in the tree point to the root) may choose to join another tree. Number of join invitations sent by an active root \( r \) in every phase depends on the phase number and it increases geometrically until all the outgoing edges of \( r \) have been dealt with, this includes the outgoing edges forwarded to \( r \) by its children. Each active star-root tries to form a larger tree by either joining another tree or by helping others to join to its tree. In addition, in every phase an active tree reduces its height by means of a pointer shortcut operation.

Kutten and Peleg [60] extended [61] to asynchronous networks and gave an algorithm with time \( \Delta T + O(\log n) \), where \( \Delta T \) is the difference between the wake-up times of the last and first vertices to be awakened, the message complexity is \( O(n \log n) \), and the communication complexity is \( O(|E_0| \log^2 n) \). As opposed to synchronous systems, here algorithm cannot first transform every directed edge to a bidirectional edge, and then apply an algorithm for a bidirectional graph. To handle the asynchrony and still provide an efficient algorithm, the authors assume that the vertices are periodically awakened by an external signal. The justification for the latter assumption is that if not all vertices are awakened by the external signal, then, given that the initial graph is a line, it may take \( \Omega(n) \) time until the chain of messages arrives at the last vertex.

Abraham and Dolev [7] provide upper and lower bounds for the asynchronous RDP. They proved that \( \Omega(n \log n) \) messages are required if the size of the network is unknown. When each node knows the size of the connected component they provide an algorithm with message complexity \( O(n \alpha(n, n)) \), where \( \alpha(n, n) \) is the inverse of the Ackermann’s function.
Konwar, Kowalski, and Shvartsman [57] considered RDP in a static synchronous setting and studied it under different assumptions about the ability of the nodes to communicate. In particular they considered the ability of the nodes to multicast messages and the size of the messages. They showed lower and upper bounds on the number of rounds needed to solve RDP in each of the considered models. In the model where each node multicasts messages to all known nodes, and the message size is correspondingly $O(n \log n)$, the nodes can decide when to stop within some constant number of local steps. Note, that the termination does not require any a priori knowledge of the number of participating nodes. In addition, the algorithms proposed in [57] move away from the notion of electing a unique root or a leader that first learns the identities of the nodes itself and then broadcasts the collected knowledge to the rest of the nodes, e.g., [65, 61]. Thus, [57] does not require the leader to detect the discovery of all nodes.

2.1.2 Dynamic Systems

Many real-world networks such as peer-to-peer (P2P) networks, Internet and, social networks are highly dynamic, with nodes constantly joining and leaving the network. The basic building block in such a dynamic system is to efficiently discover the nodes that are willing to cooperate. The algorithms designed for those systems should be able to tolerate continuously changing topologies of the network. For example, Gnutella [72, 52] was the first P2P network of its kind, mainly used for file sharing. It is a fully distributed alternative to semi-centralized systems such as Napster, MP3 file sharing environment [13]. Instead of having clients and servers, Gnutella has a two-tier topology, where a small subset of participating peers become ultrapeers, forming the top-level overlay, while the rest are called leaf peers, that connect to
the top-level through one or multiple ultrapeers. Each ultrapeer tries to maintain thirty leaf children, and each leaf tries to maintain three ultrapeer parents.

As already mentioned, Kutten et al. [61] provided a very efficient deterministic algorithm for the resource discovery problem. However it does not provide strong fault-tolerance properties and does not deal with dynamic situations, and so its correctness and performance cannot be guaranteed in the presence of failures. The authors suggested that in order for their algorithm to handle dynamic networks their algorithm could be re-run from time to time. However, this is not easy because of the associated problem of detecting termination without any a priori knowledge of the number of the participating nodes; this is referred to as Lipton’s question (per [47]).

Konwar, Kowalski, and Shvartsman [58] considered dynamic settings where the set of participants changes over time. They studied the number of communication rounds needed to solve the problem under a variety of assumptions about joins and failures. The authors considered an asynchronous message passing system and raised the following question: For any two participants that join the dynamic system, how long does it take for them to discover each other? For complexity analysis the authors restrict the asynchrony and impose some constraints to guarantee that the universe is connected. The problem is considered in a variety of models that take into account joining at a single node versus multiple nodes, coupled with the presence or absence of failures.

Haeupler et al. [45] considered gossip based approach in dynamic networks motivated by resource discovery in large scale distributed networks, such as P2P and social networks. They proposed and analyzed two discovery algorithms, one of them is called push discovery or triangulation, and the second one is called pull discovery or two-hop walk. In the push
**discovery** algorithm in every round each node chooses two neighbors and connects them by “pushing” their mutual information to each other. In other words in each round each node adds an undirected edge between its two randomly selected neighbors. If the edge already exists, then the graph is not modified. In the **pull discovery** algorithm, in each round each node connects itself to a random neighbor of a neighbor by “pulling” a random neighboring ID from a randomly selected neighbor. They concentrate on limiting the size of the message, and in particular, each node is allowed to send at most $O(\log n)$ bits in every round. They show that for undirected graphs both the push and pull discovery algorithms converge in $O(n \log^2 n)$ time whp. They prove that $\Omega(n \log n)$ is a lower bound on the number of rounds. Additionally, the authors show that their results also apply when they require only a subset of $k \leq n$ nodes to converge, and this takes $O(k \log^2 k)$ rounds whp. For the directed graphs they show that the pull process takes $O(n^2 \log n)$ to converge whp. They further show a matching lower bound for weakly connected graphs, and $\Omega(n^2)$ lower bound for strongly connected directed graphs.

The authors also note that the algorithms that send more than $O(\log n)$ bits per message can be made bandwidth-limited by spreading the transfer of long messages across possibly a linear number of rounds. However, this will require the nodes to coordinate and maintain the state, while their algorithm is stateless.

**Self-Stabilizing Algorithms.** In dynamic networks it is important to design algorithms that are able to deal with intermittent failures, and in particular to consider **self-stabilizing** solutions. Here the algorithm must automatically bring the system into a legitimate state in spite of transient failures. The self-stabilization requirement is that a legitimate state is reached from an arbitrary state in a finite time.
Self-stabilization was introduced by Edsger W. Dijkstra in 1973 [27]. In [28] Dijkstra provides proofs for the problems introduced in [27]. Shlomi Dolev wrote in his book [29], that Dijkstra’s work was not widely noticed until Leslie Lamport’s invited talk at the ACM Symposium on Principles of Distributed Computing in 1983. In his talk Lamport said that he considered [27] as Dijkstra’s most brilliant published paper and that he regarded it as a milestone in work on fault tolerance.

Self-stabilizing systems are designed to start in an arbitrary state and still converge to a desired behavior. As a result, such a system can automatically recover after experiencing a fault.

Dolev and Herman [31] pursued a super-stabilizing approach to designing algorithms that maintain topological structures (e.g., a spanning tree) in the presence of perturbations. They introduced a notion of a passage predicate, which is a safety property that should hold while the protocol tries to reestablish the legitimacy predicate following a topology change. The protocol is then called superstabilizing if it is self-stabilizing and it guarantees to satisfy the passage predicate while the system undergoes a topology change starting from a legitimate configuration. The authors assume that each processor is equipped with an interrupt statement that indicates the occurrence of a topology change event. Following the topology change event the state variables of each processor that store information about the neighborhood are updated. In addition, they assume that transient failures cause erasures of variable values, making such failures easily detectable.

Nor, Nesterenko, and Scheideler [69] developed a self-stabilizing algorithm, called Corona, for skip-list construction in asynchronous networks. They prove that in asynchronous networks one must constrain the states from which self-stabilizing solutions can be constructed: the state
information has to form a weakly connected graph and it must only contain identifiers that are present in the system. They also assume that messages in transit cannot be corrupted, or in other words, a message can only carry information about existing nodes in the network.

Recently, Kniesburges, Koutsopoulos, and Scheideler [56] proposed a self-stabilizing algorithm for the resource discovery problem in synchronous message passing setting. Motivated by [45] they also concentrate on limiting the number of messages and bits sent per round by every processor. They introduce a new efficiency measure for the resource discovery problem, called work. They measure the work of a node based on the number of IDs each node receives or sends while executing the algorithm. They also assess the maximum number of messages sent or received by a single node. Their algorithm converges in $O(n)$ rounds, where each node sends and receives $O(n)$ messages in total, and each message contains $O(1)$ IDs, while the total work is $O(n^2)$. To minimize the number of messages sent by a node in every round, the authors assume that each node only shares information with its immediate successor and predecessor, where the predecessor of a node is defined to be a node with the next larger identifier. In particular, in each round a node forwards one of the nodes it knows about in a round-robin manner to its predecessor. As they describe, the intuition behind their strategy is that if a node repeats this process a sufficient number of times, then eventually the root will learn about all IDs in the system. The root then will forward the IDs in a round-robin manner to its successor, who will then forward it to its successor, and so on. Their strategy suffers from not being stateless, as opposed to [45]. In addition, the authors also considered the super-stabilization approach proposed in [31], and they showed that after system reaches a legitimate configuration a single join will take $O(n)$ rounds and messages, while it takes $O(1)$ rounds and messages to recover the clique after a node leaves the network.
2.1.3 Summary

Section 2.1 discusses models under which the resource discovery problem was previously considered. It is worth noting that all known asynchronous algorithms for solving the RDP put some restrictions on asynchrony assumptions to be able to perform complexity analysis, e.g., [60, 58]. While considering self-stabilizing algorithms for the problem it is important that the algorithm handles any kind of corruption of node’s local variables, while still remaining efficient, in terms of time, message, and communication complexities. Prior research shows that it is impossible to develop algorithms that are efficient in all considered complexity measures. In this thesis we present two algorithms for the resource discovery problem that handle arbitrary corruptions of node’s local variables. We show that the stabilization time of the algorithm can be substantially reduced by increasing the communication complexity.

2.2 Network Supercomputing

Since 1996 a global community of users has emerged dedicated to performing real research on a massive scale using ordinary home and office computers connected through the Internet. Thousands of individuals, businesses, teams (such as Ars Technica’s Team Prime Rib), schools, universities, and other agencies offer their spare CPU time to this math research. In this section we provide a few examples of existing Internet Supercomputers and give an overview of the current research that abstracts the problem of massive distributed computation in the form of a distributed system consisting of a central server and a collection of worker processors. Additionally, we give a brief overview of some problems in Distributed Computing, e.g. Do-All, that are related to the problem of network supercomputing.
2.2.1 Internet Supercomputing

Internet PrimeNet Server [3] is a grid system for the Great Internet Mersenne Prime Search (GIMPS). PrimeNet comprises tens of thousands of home and office PCs, servers, and laptops with either LAN or dialup Internet connections. So far, the largest Mersenne Prime was found on January 25, 2013 and it is $2^{57,885,161} - 1$.

Another example of cooperative network supercomputing is SETI@home [5], where SETI stands for Search for Extra Terrestrial Intelligence. SETI@home is an Internet-based public volunteer computing project employing the BOINC (Berkeley Open Infrastructure for Network Computing) software platform, hosted by the Space Sciences Laboratory, at the University of California, Berkeley. The purpose of SETI is to analyze radio signals, searching for signs of extra terrestrial intelligence. SETI@home was released to the public on May 17, 1999, making it the second large-scale use of distributed computing over the Internet for research purposes, as Distributed.net [1] was launched in 1997. The latter was founded in response to the $RC5 - 32/12/7$ (56 bit) Secret Key Challenge, a contest testing RAS LAB’s 56 bit encryption algorithm technology. After 212 days of work $RC5 - 56$ challenge was solved. At the end of the contest, 4000 active teams of volunteers managed to evaluate 46% of the possible solutions. Other examples of cooperative network supercomputing include, MilkyWay@home [4] and Einstein@home [2].

Fernández, López, Santos, and Georgiou [36] abstracted the problem of massive distributed computation in the form of a distributed system consisting of an infallible master processor and a collection of $n$ processors, called workers, that can execute tasks. The worker processors
are untrusted and might act maliciously. In addition, they assume that there is a known non-decreasing probability $d$ of master receiving a reply from a given worker on time. They further assume that each task returns a binary value, and the master should accept only correct values \textit{whp}. They assume that the service provided by the workers is not free. The master is charged a work-unit for each task assigned to a worker. The goal is for the master computer to accept the correct value of the task \textit{whp}, with the smallest amount of work (number of workers the master assigns the task). They consider two ways of bounding the number of faulty processors: (1) the maximum number of workers that may fail is bounded by $f < \frac{n}{2}$, and (2) all processors are faulty with probability $p < \frac{1}{2}$, independently of each other. For each model they establish a minimum number of workers that need to perform the same task in order for the master to decide with probability of success at least $1 - \varepsilon$, where $\varepsilon \ll n$ (e.g., $1/n$). They propose two very simple algorithms: (1) the \textit{Majority Based Algorithm} where the master decides on the majority of received responses, and (2) the \textit{Threshold Based Algorithm} in which if the master receives a certain number of responses with equal value then it makes a decision, otherwise it decides on the majority of the received responses. The latter one is an early-terminating algorithm.

Konwar, Rajasekaran, and Shvartsman [59] extended [36] by removing the assumption that the number of faulty processors or the probability of a processor acting maliciously is known to the master. They consider a synchronous model of computation where processors communicate by exchanging authenticated messages. For $n$ processors and $n$ tasks they consider two failure models: (1) model $\mathcal{F}_n$, where $0 < f < \frac{1}{2}$ fraction of the workers provide faulty results with probability $0 < p < \frac{1}{2}$, given that the \textit{master} has no a priori knowledge of the values of $p$ and $f$, and (2) model $\mathcal{F}_b$, where at most $0 < f < \frac{1}{2}$ fraction of the workers can reply with arbitrary
results and the rest reply with incorrect results with probability $p$, $0 < p < \frac{1}{2}$, where the master knows the values of $f$ and $p$. In the model where the master does not know the values of $f$ and $p$, they use a stopping rule algorithm by Dagum et all [19] to estimate these values. They provide an $(\varepsilon, \delta)$-approximation of $f$ and $p$, for any $0 < \delta < 1$ and $\varepsilon > 0$. While analyzing their algorithms, they consider two definitions of work. The first is termed task-oriented work ($\text{work}$), and it does not account for idling or waiting processors [32]. The second is termed total work ($\overline{\text{work}}$), including idling and waiting [50]. In model $F_a$ they provide an algorithm that performs all $n$ tasks in $O(\log n)$ time, $\text{work} = O(n)$, and $\overline{\text{work}} = O(n \log n)$, all whp. In model $F_b$ they provide an algorithm that performs all $n$ tasks correctly in $O(\log n)$ time and has $\text{work}$ and $\overline{\text{work}}$ complexities $O(n \log n)$, both whp, for $p > 0$, $f < \frac{1}{2}$, and $(1-f)(1-p) > \frac{1}{2}$.

Fernández, Georgiou, and Mosteiro [37] pursued a game-theoretic approach. They assume that there are three types of workers: malicious, always return an incorrect result, altruistic, always return a correct result, and rational, that act based on their self interest. They assume a stochastic distribution of the workers over the three types. Under this approach, the master provides either a reward or a penalty, should a worker be honest or cheat respectively. The design objective is for the master to force a desired unique Nash Equilibrium [68], i.e. to design a strategy for the workers such that a worker has no incentives to gain by changing only its own strategy. They assume that neither master, nor workers, know the type of other workers, however the probability distribution is known to all of them. Note that only rational workers play the game looking for Nash Equilibrium, since malicious and altruistic workers always cheat and are honest, respectively.

Christoforou, Fernández, Georgiou, and Mosterio [18] extended [37] by adding an unreliable communication. Here the results computed by a worker may be delivered with a delay,
or not delivered at all. The latter is modeled by means of a positive probability, determining the delivery of the result. If the reliability of the network is low, then it is possible, that the result calculated by a rational worker is not delivered to the master, even if the worker choses to calculate the result correctly. In this case, the worker is not rewarded, since the master has no way of knowing the result, however the worker incurs cost for performing the task. For this reason, the authors also allow the rational workers to refrain from performing the task. Hence, in this model the master has a more challenging task, it needs to provide necessary incentives for a rational worker to reply honestly, even if the network reliability is low. The authors develop and analyze two game-theoretic mechanisms in this model, and apply them to two settings: SETI-like volunteer computing applications and contractor-based applications such as Amazon’s mechanical turk.

Paquette and Pelc [70] considered a model of a fault-prone system in which a decision has to be made on the basis of unreliable information and designed a deterministic strategy that leads to a correct decision \(whp\). The decision is computed in time linear in the number of processors. They consider the decision strategy in both synchronous and asynchronous systems. In the latter case the decision is also complicated by the fact that some processors may respond late even though they are fault free. They also considered the decision optimality from a local point of view. Instead of maximizing the overall probability of correctness, they consider every set of values conveyed by the processors, and choose the conditionally most probable original value that could yield this set. They construct a locally optimal decision strategy, where the decision value is again calculated in the time linear in the number of processors. Such a strategy is called locally optimal, since it does not consider the impact of such a choice on the overall probability of correctness. They also show, that if all the processors have the
same failure probability smaller than \( \frac{1}{2} \), then local probability maximization strategy may lead to a decision strategy with the highest probability of correctness. However, surprisingly, the latter is not true if the probabilities of failure are either not similar, or greater than \( \frac{1}{2} \).

Gao and Malewicz [39] considered the problem of maximizing the expected number of correct results when the tasks have dependencies. Their distributed system is composed of a reliable server that coordinates unreliable workers that compute correctly with probability \( p < 1 \), and where any incorrectly performed task corrupts all dependent tasks. The goal is to produce a schedule for task execution by the participants, reliable server and unreliable workers, that maximizes the expected number of correct results under a constraint on the computation time. They consider \( k \times k \) mesh, and develop different scheduling strategies based on the bound on \( p \). They establish that if \( p \) is close to 1 then the best strategy for maximizing the expected number of correct results is for the reliable server to execute the diagonal tasks of the mesh, while if \( p \) is closer to 0, the reliable server should execute the border tasks.

### 2.2.2 Related Work

**Do-All Problem.** A related problem, called Do-All, deals with the setting where a set of processors must perform a collection of tasks in the presence of adversity [43, 51]. Here the termination condition is that all tasks must be performed, but the processors need not learn the results of the computation. Chlebus and Kowalski [17] gave a lower bound \( \Omega(t + n \frac{\log n}{\log \log n}) \) on work of any algorithm solving Do-All, including randomized, against an adaptive linearly bounded adversary.

Georgiou, Russell, and Shvartsman [42] considered a variation of the Do-All problem, called Omni-Do in partitionable networks. In an Omni-Do problem a processor stops executing
tasks only if it knows the results of all tasks. In partitionable networks a group of processors may become disconnected during the computation, with a possibility of a reconnection during the course of the algorithm. Disconnected groups cannot communicate with each other, while they assume that processors in the same group communicate cost free. They, further assume that processors do not crash. If a new group is created during a reconfiguration, a transition from one network partition to another, then it is assumed that the group possesses the combined knowledge of all its members. They propose a randomized algorithm, called RANDOM SELECT (RS), where a processor (or a group), randomly selects an undone task, based on the combined knowledge of members of the group about the performed tasks. Given that a lower bound for work is $\Omega(t \cdot g)$ for any task-performing algorithm, where $t$ is the number of tasks, and $g$ is the number of disconnected components, the authors pursue competitive analysis [73]. They show that RS obtains the competitive ratio $(1 + cw/e)$, where $cw$ is the computation width and $e$ is the base of the natural logarithm. The notion of computation width associates a natural number with a history of changes in the communication medium.

Georgiou and Kowalski [40] studied dynamic cooperative computing in synchronous message passing systems, where $n$ processors can crash/restart and tasks can be continuously and dynamically injected to the system. They view this problem as an online problem and pursue competitive analysis [73]. They measure performance in terms of the number of pending tasks and compare it with the optimum number obtained by the best offline algorithm that knows a priori the crash-restart-injection pattern. They study the problem with respect to two properties about the task performance guarantees. The first one, correctness property, requires that information about the tasks is not lost. The second one, fairness property, requires that all tasks
injected into the system are eventually performed. The problem is studied under a variety of assumptions about the task injector authority. First, they assume that there is a central scheduler, that at the beginning of each round informs all operational processors about all pending and newly injected tasks. Second, they relax the latter model, and consider a weaker centralized authority, called central injector, which at the beginning of each round informs processors about all tasks injected in the current round and all tasks performed in the previous round. Lastly, they consider a local injector, that injects tasks to processors without giving them any global information. The authors derive a lower bound of $OPT + \frac{n}{3}$ on the pending task competitiveness of any deterministic algorithm, where $OPT$ denotes the optimal off-line algorithm. They then develop an algorithm that achieves competitiveness of $OPT + 2n$ for the models of central scheduler and central injector. Finally they develop an algorithm for the local injector model that achieves $OPT + 3n$ competitiveness given that reliable multicast [15] is available.

**Rumor-Spreading.** In master-worker setting all workers report to the master, while in the decentralized setting the results of the task should be distributed among all workers. It is therefore important to utilize efficient gossiping strategies. A simple rumor-spreading algorithm for the random phone-call model is the push algorithm that operates as follows: starting from the round in which rumor $r$ is generated, every participant $v$ that knows $r$ randomly selects a participant $u$ and makes a call, informing $u$ of the rumor $r$. After $\Theta(\log n)$ rounds all participants know $r$ whp [71]. The runtime of the push algorithm is asymptotically optimal, the message complexity of the algorithm is $\Theta(n \log n)$.

Karp, Schindelhauer, Shenker, and Vocking [54] proposed a push-pull rumor spreading algorithm, which has asymptotically optimal runtime, with a smaller message complexity compared to the push algorithm. In the push-pull algorithm each participant $v$ not only pushes
rumor $r$ to a randomly selected participant $u$, but also $v$ tells about $r$ to any participant that contacts $v$ in the current round. In a sense everyone who calls $v$ pulls $r$ from $v$. The algorithm terminates after $\log_3 n + \Theta(\log \log n)$ rounds whp, and has $\Theta(n \log \log n)$ message complexity when a rumor has only one source. They also propose a variant of this algorithm, for the case when a rumor has more than one source, that terminates in $O(\log n)$ rounds and has $\Theta(n \log \log n)$ message complexity. Additionally, the authors show that in the random phone-call model no decentralized algorithm that takes $O(\log n)$ rounds and uses $O(n)$ messages per rumor can guarantee that a rumor is spread to all participants whp. Moreover, they show that regardless of the number of rounds $\Omega(n \log \log n)$ messages are required for any address-oblivious algorithm such as push and push-pull algorithms.

Fraigniaud and Giakkoupos [38] studied the bit communication complexity of a push-pull algorithm in a random phone-call model. They proposed an address-oblivious algorithm that uses $O(\log n)$ rounds and $O(n(b + \log \log n \log b))$ bits of communication for spreading a $b$ bit rumor among all the participants whp. During the push phase of the algorithm, they require the recipient of the rumor to inform the sender whether it already knew the rumor. A participant stops spreading the rumor $r$ if it pushed the rumor to a certain number of participants that were already aware of $r$. While the pull transmission is executed only once in every $\log n / \log \log n$ rounds.

$(\varepsilon, \delta)$-Approximation. Depending on the considered model of computation it might be important to estimate the probabilities of workers faithfully performing the tasks, in both master-worker and decentralized settings. Dagum, Karp, Luby, and Ross [19] describe an approximation algorithm $AA$, that for a random variable $Z$ distributed in $[0, 1]$ with $\mathbb{E}[Z] = \mu_Z$, and for input parameters $\varepsilon > 0$ and $\delta > 0$, produces an $(\varepsilon, \delta)$-approximation $\hat{\mu}_Z$ of $\mu_Z$. They consider
\( \hat{\mu}_Z \) to be an \((\varepsilon, \delta)\)-approximation of \( \mu_Z \) if \( \Pr[\mu_Z(1-\varepsilon) \leq \hat{\mu}_Z \leq \mu_Z(1+\varepsilon)] \geq 1-\delta \). An \((\varepsilon, \delta)\)-approximation of \( \mu_Z \) is desired for problems where the exact computation of \( \mu_Z \) is NP-hard.

The \((\varepsilon, \delta)\)-approximation has been applied to a wide range of difficult scientific problems. For example, it has been successfully applied for approximation of probabilistic inference in Bayesian networks [20], solving Ising model problems in statistical mechanics [49], estimation of convex bodies [34], and estimating the number of solutions to a DNF formula [53].

### 2.2.3 Summary

Section 2.2 presented an overview of different models under which the network supercomputing problem has been considered and discussed a related problem, called Do-All. A lower bound on work complexity of Do-All problem is also a lower bound for the network supercomputing problem considered in this thesis. Prior research concentrated on centralized models of computation, where an infallible master processor assigns tasks to a collection of processors, called workers. In this thesis we develop algorithms that remove the assumption of an infallible and bandwidth-unlimited master processor and consider a fully decentralized solution using just the cooperating workers.
Chapter 3

Resource Discovery

The ability to discover relevant resources is important in dynamic distributed settings where the available resources (e.g., computers) need to cooperate in solving common problems. One of the key requirements for solutions for the Resource Discovery Problem (RDP) is that they must be able to cope with intermittent failures. In this chapter we present self-stabilizing algorithms for the RDP. Here an algorithm must automatically bring the system into a legitimate state in spite of transient failures. We first describe models of computation and failures and formalize the closure and convergence properties of self-stabilizing algorithms [10], with the goal of presenting our algorithms using the Timed Input/Output Automata [55] formalism. In the sections that follow we present our self-stabilizing algorithms for the RDP that work for dynamic systems and handle transient failures that result in corruptions of state variables that are indistinguishable from valid states. Finally, we prove the correctness and self-stabilization of the presented algorithms.
3.1 System Model and Definitions

**Model of computation.** We consider a universe of processors, with unique identifiers from a well-ordered set $U$. Let $V \subseteq U$ be the subset of processors that participate in the computation; this set is chosen by the environment. We let $v_0$ stand for $\min \{v : v \in V\}$. The set $V$, its cardinality, and $v_0$ are unknown to the processors, but each processor in $V$ is aware of one other processor in $V$.

The processors communicate over a fully connected synchronous network. There is a known upper bound $d$ on message delays. If a node expects a message from another node and the message is sent, then it is delivered within $d$ time units. Nodes have access to local timers that can be used to implement message time-outs. Local computation takes negligible time relative to $d$.

We define a *round* to be some constant period of time sufficient for a processor to send or multicast messages, to perform some local computation, and to accept any incoming messages. Let $t$ be a time duration sufficient for implementing a round; $t$ is established at compile time with the knowledge of the delay upper bound $d$. For our purposes it suffices to set $t$ to $2d$.

The round structure provides only a coarse notion of synchrony. Distinct processors may execute different sequences of instructions during rounds, and the algorithm *cannot* assume that the individual instructions at different processors within concurrent rounds are synchronized. Lastly, we do not assume that all processors begin participating in the computation simultaneously; instead we allow the processors to join the computation at arbitrary times.

**Failure model.** The processors are subject to transient failures. A transient failure is an event that corrupts the state of the system, but it does not change the algorithmic behavior of the
system: the static code of each processor and any constants are incorruptible. A failure may arbitrarily perturb state variables, including the program counter. A corrupted variable may contain a value that is syntactically indistinguishable from a valid value. Thus, in our model it is possible for a state variable to contain incorrect information even though it appears to hold a valid value of a correct type. Messages in transit can also be arbitrarily corrupted.

Local states, configurations, and transitions. The local state of a processor consists of the values of its variables and its program counter. We denote by $s_v$ the state of node $v$. A configuration is a cross product of the local states.

Definition 3.1.1 A system $S$ is a triple $(C, A, \tau)$, where $C$ is a set of configurations, $A$ is a set of actions, and $\tau$ is a transition function: $\tau : C \times A \rightarrow C$. An execution of $S$ is a sequence $c_0, a_0, c_1, a_1, c_2, \ldots$ such that for all $i \geq 0$, $\tau(c_i, a_i) = c_{i+1}$.

We denote a transition from configuration $c_i$ to $c_{i+1}$ by $c_i \xrightarrow{\tau} c_{i+1}$ and we let $c \xrightarrow{\tau} c'$ stand for the fact that $c'$ can be reached from $c$ by zero or more transitions.

When a state variable $X$ of node $v$ appears outside of the scope of its definition we use a notation $X_v$. We denote a state variable $X$ of node $v$ in configuration $c$ by $c.X_v$.

Self-stabilization. Self-stabilization is the ability of a system to recover from transient failures following their cessation. The impact of a failure is that the transition from configuration $c$ to configuration $c'$ may not obey the transition function $\tau$, that is, a failure may cause $c' \neq \tau(c)$.

In addition to the local state corruption we assume that a system can start in any configuration. In designing solutions resilient to transient failures we use self-stabilization techniques, formalizing self-stabilization in terms of closure and convergence properties (cf. [10]).
Definition 3.1.2 (Self-stabilization) Let problem $P$ be to establish and maintain an invariant $\psi()$, given as a boolean predicate over configurations. System $S = (C, A, \tau)$ is a self-stabilizing solution for problem $P$, if the following two conditions hold:

**Closure:** $\forall c \in C, \forall a \in A : \psi(c) \implies \psi(\tau(c,a))$, i.e., $\tau$ maintains the invariant.

**Convergence:** $\forall c \in C : \exists c' \in C : c \xrightarrow{\tau} c' \land \psi(c')$, i.e., $\psi()$ can be established in the absence of failures.

**Problem Statement.** We let each processor $v$ have a constant $nb_v \in V$, where $v \neq nb_v$, representing the knowledge of node $v$ of one other node (a neighbor). This induces a directed graph.

**Definition 3.1.3 (Connectivity Graph)** Given the set $V$ and $nb_v$ for all $v \in V$, we define the *connectivity graph* as a directed graph $G = (V, E)$, where $E = \{(u, v) : nb_u = v\}$.

Figure 1 is an example of an eight node connectivity graph.

![Figure 1: Example of a Connectivity Graph](image-url)
Following the original formulation in [47] we assume that the connectivity graph is weakly-connected. Notice, that weak connectivity is necessary in order for the network to evolve into one connected component. In practice this means that a joining node is given an address of at least one machine in the network. Each processor $v$ has three main local variables, $\text{parent}_v \in V$, $\text{Children}_v \in 2^V$, and $\text{world}_v$, where $\text{parent}_v = u$ means that $v$ considers $u$ to be its parent, $u \in \text{Children}_v$ means that $v$ considers $u$ to be its child, where $\text{Children}_v$ is the set of all children of $v$, and finally, if $v$ knows $u$ then $u \in \text{world}_v$. Boolean variable $\text{active}_v$ indicates whether the node $v$ is active or not. Depending on the algorithm, each processor may also have other auxiliary, algorithm specific, variables that are introduced when the algorithm is presented.

For convenience we let $G^u = (V, E^u)$ be the undirected graph induced by $G = (V, E)$, called the initial knowledge graph. Let $D$ be the diameter of $G^u$ and $\text{dist}(u, v)$ be the length of the shortest path from node $u$ to $v$ in $G^u$.

**Definition 3.1.4 (Knowledge Graph)** Let $c$ be some configuration of the system. The knowledge graph at $c$ is a derived state variable $c.\mathcal{G} = (c.\mathcal{V}, c.\mathcal{E})$, where

1. $\mathcal{V} = \{v \in V : c.\text{active}_v = \text{true}\}$ is the set of nodes that joined the computation.

2. $c.\mathcal{E} = \{(u, v) : u, v \in c.\mathcal{V} \land v \in c.\text{world}_u\}$ is the set of edges.

In this definition of $c.\mathcal{E}$ an edge $(u, v)$ models the fact the $u$ knows $v$, however, this does not imply that $v$ knows $u$. Where the configuration $c$ is implied by the context, we use the simplified notation $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. We denote by $G^u$ the undirected graph induced by $\mathcal{G}$. 

34
Definition 3.1.5 (Resource Discovery Problem) Let $G$ be a weakly connected graph. The Resource Discovery Problem (RDP) is to establish and maintain the following invariant on configurations: $(\exists v \in V : (\text{Children}_v = V) \land (\forall u \in V : \text{parent}_u = v)) \land (\forall u \in V : \text{world}_u = V)$, that is,

1. There exists a node $v \in V$ such that $\text{Children}_v = V$.
2. For every node $u \in V$ we have $\text{parent}_u = v$.
3. For every node $u \in V$ we have $\text{world}_u = V$.

For example, given the weakly connected graph depicted in Figure 1, we say that the RDP is solved if an algorithm converges to the graph depicted in Figure 2. Where all nodes in the graph know each other, thus forming a complete graph, every node recognizes node ‘1’ as its parent, and finally, node ‘1’ recognizes all nodes in the graph as its children.

![Figure 2: Instance of a Legitimate Configuration for RDP](image)

Programming notation. We use Timed Input/Output Automata (TIOA) [55] formalism to specify and reason about our algorithm. A timed automaton is a labeled state transition system.
The state of the automaton is defined by its state variables. The discrete transitions of the automaton are defined in terms of actions, where each action is of the type input, output, or internal. The state of the timed automaton may change in two ways: by discrete transitions that change the state atomically, and by trajectories that describe the evolution of the state over intervals of time. The overall system is composed of the automata for all processors and its state is composed of the states of all automata.

The automata must be input-enabled and must not block time passage. A timed automaton executes by performing a sequence of alternating trajectories and discrete transitions, in which the states match up properly. We consider only executions where during any finite time period no infinite number of actions occur. We also consider only fair executions, where during each (algorithm-specific) round every locally-controlled enabled action (i.e., internal and output actions) occurs by the end of that round, and for every message sending action that is enabled at the beginning of the round the corresponding message receiving action occurs before the end of the round.

Communication primitives. Nodes communicate via multicast, where each multicast results in a point-to-point message from the source to each destination. A multicast is invoked using the msend\((m, I)_i\) action, where \(m\) is the message, \(I\) is the set of destinations, and \(i\) is the node invoking the multicast. Multicast messages are received through the mrecv\((m, u)_i\) action, where \(m\) is the message, \(u\) is the source node, and \(i\) is the node receiving the message. Actions msend\((m, I)_i\) and mrecv\((m, u)_i\) are implemented in a straightforward way using point-to-point send/receive. We denote by Channel\(_{i,j}\) the conventional synchronous channel from node \(i\) to node \(j\). In Figure 3 we present the data types, signature, state, and transitions of Channel\(_{i,j}\).
Data-types:
- $U$, the set of node identifiers
- $M$, the set of messages

Signature:
- Input: $send(m, j), m \in M, j \in U$
- Output: $receive(m, j), m \in M, j \in U$

State:
- $queue$, a FIFO queue of elements of $M$

Transitions:
- Input $send(m, j)$
  - Effect: add $m$ to $queue$
- Output $receive(m, j)$
  - Precondition: $m$ is first on $queue$
  - Effect: remove first element of $queue$

Figure 3: Data types, signature, state, and transitions of $Channel_{i,j}$ at node $i \in V$ at node $i$ for $i \in V$. We assume that each $Channel_{i,j}$ contains state variable $S_{i,j}$ storing messages $m$ in transit from node $i$ to node $j$.

**Modeling state and message corruption.** The adversary can arbitrarily perturb the state of any node and corrupt any message in transit. We model state corruption at node $i$ by means of the action $perturb_i$ that is always enabled and whose effects contain the $HAVOC$ statement (borrowed from Lampson’s SPEC language) that arbitrarily changes the state of node $i$ due to a failure. We model the corruption of a message in transit using action $corrupt(m, m')_{i,j}$ defined as:

```
“corrupt(m, m')_{i,j} : Precondition: m \in S_{i,j}; Effects: S_{i,j} \leftarrow (S_{i,j} - \{m\}) \cup \{m'\},”
```

that is, whenever message $m$ is in transit (i.e., in $S_{i,j}$), it can be replaced by some message $m'$.

### 3.1.1 Measures of efficiency.

We assess the efficiency of the algorithm in terms of *stabilization time* and *stabilization message complexity*. The stabilization time is measured in terms of the worst case number of rounds following the cessation of perturbations needed to establish the resource discovery invariant.
Message complexity deals with the number of point-to-point messages, where in the case of multicast, each instance of multicast is assessed as the number of the resulting point-to-point messages. The stabilization message complexity is measured in terms of the worst case number of point-to-point messages sent among the participants to establish the resource discovery invariant following the cessation of perturbations. Note that local state corruptions may cause messages to be sent to an arbitrary subset of processors in $U$. In assessing stabilization message complexity we charge to the environment any messages sent by an algorithm prior to the cessation of perturbations, and any messages sent to bogus destinations as a result of state corruptions. This is because the adversary may cause an arbitrary number of messages sent in each round; in particular, in the case when $|V| = o|U|$, bogus messages may dominate message complexity, rendering any algorithm inefficient. (In the analysis we show that, following the cessation of perturbations, after at most three complete iterations no messages are sent to bogus destinations.)

3.2 Self-Stabilizing Resource Discovery Algorithms

In this section we present our approach to designing self-stabilizing algorithms for solving the resource discovery problem in deterministic synchronous message passing systems using TIOA. Our algorithms tolerate arbitrary perturbations to the node’s local state and are guaranteed to solve the problem once such failures subside. The idea behind the algorithms is similar. The main difference is in the way the information is propagated. Here, using a more aggressive propagation mechanism allows an algorithm to converge to a legitimate configuration faster,
asymptotically meeting the lower bound [47]. On the other hand, the algorithm that uses a con-
strained multicast instead of an aggressive broadcast, has a lower message complexity. This
shows a trade-off between time and communication complexities.

Any self-stabilizing algorithm that uses TIOA is equipped with a clock. Our algorithms
are designed for synchronous message passing systems, and here the variable clock represents
the time of the synchronous system. Algorithm specifications include the transitions join and
perturb. The environment may activate node $i$ by using input action join$_i$, and it may disable
and/or corrupt the state of node $i$ by means of input action perturb$_i$, where HAVOC assigns
arbitrary values to the state variables, modeling a transient failure. If HAVOC sets active to
false, the action models a crash of a node.

Local operations of a node are structured in terms of rounds. Recall that failures cannot
change the synchronous nature of the system, and thus clock is the only variable that is not
affected by transient failures. The variable records the passage of time consistently at all nodes:
the trajectory evolves clock at the same rate as real time ($d(\text{clock}) = 1$). We establish the
compile-time constant $t$ to be sufficient for a node to multicast outgoing messages, to perform
the needed local computation, and to accept any incoming messages (this constant is readily
obtained from the structure of the algorithm and from the knowledge of the worst case message
delivery delay $d$). The constant $t$ is used to control the duration of a round.

Our algorithms have an iterative structure consisting of two synchronous rounds. We refer
to the first round as the gossip phase and to the second round as the confirm phase. The value
of clock is used to determine whether an active node is in the gossip or confirm phase. The
communication takes the form of a constrained “gossip,” where in the gossip phase a node mul-
ticasts to a certain subset of nodes that it knows about (the destinations may change depending
on the algorithm). While in the confirm phase each node responds to the messages received in the gossip phase and to its neighbors. Thus, a node receiving such responses validates the identities of the nodes it contacted in the gossip phase.

In the algorithms the unique root is ultimately discovered as the node with the smallest identifier. Each node computes a local minimum based on its “verified” knowledge and it considers the node with the smallest identifier to be its parent. Because failures can corrupt the state of any node, if a node does not hear from its parent during an iteration it decides that something is wrong and resets its state. Any node includes among its children every node that considers it to be the parent. Each node also maintains knowledge about neighbors based on the initial graph $G$.

While explicit modeling of message corruption is straightforward, doing so would complicate the reasoning about the properties of our algorithms. Therefore we next show how to model arbitrary message corruption using only the perturbation of local states.

**Modeling message corruption through local state corruption.** Let $\alpha$ be an execution (untimed executions are sufficient for our purposes) where node $i$ sends a message $m$ to node $j$ by invoking action $\text{msend}(m, I)_i$, where $j \in I$. While in transit, message $m$ is corrupted by means of action $\text{corrupt}(m, m')_{i,j}$. Later in $\alpha$, action $\text{mrecv}(m', i)_j$ results in the delivery of message $m'$ to node $j$. Thus $\alpha$ can be given as:

```
..., s, \text{msend}(m, I)_i, s_1, \ldots, s_2, \text{corrupt}(m, m', j)_i, s_3, \ldots, s_4, \text{mrecv}(m', i)_j, s', \ldots
```

where $s$, $s_k$, and $s'$ are the states. Note that the delivery of the corrupted message to node $j$ affects only the state component of $j$ in $s'$ and the state of the channel. Additionally, the state components of nodes in the states starting with $s_3$ and ending with $s_4$ are not affected by the corruption.
Let us break $\alpha$ into two execution fragments $\alpha_1$ and $\alpha_2$, where $\alpha_1$ is the prefix of $\alpha$ up to state $s'$ and $\alpha_2$ is the suffix of $\alpha$ starting with state $s'$. Consider now the family of the execution fragments, where each fragment $\beta$ is obtained from $\alpha_1$ by removing action $\text{corrupt}(m, m', j)_i$ after state $s_2$, updating the states that are affected by this removal, appending action $\text{perturb}_j$ as the last action, and adding a suitable state $s''$ that records the result of the perturbation:

$$\ldots, s, \text{msend}(m, I)_i, s_1, \ldots, s_2 = s'_3, \ldots, s'_4, \text{mrecv}(m, i)_j, \bar{s}, \text{perturb}_j, s''$$

Let us examine the states in $\beta$ from $s'_3$ to $s'_4$: they are exactly the same as the corresponding states in $\alpha_1$ from $s_3$ to $s_4$, except that $S_{i,j}$ in $\text{Channel}_{i,j}$ contains message $m$ instead of the corrupted message $m'$. Thus, action $\text{mrecv}(m, i)_j$ results in a state $\bar{s}$ that is exactly the same as state $s'$ in $\alpha_1$, except that the state component of $j$ is possibly different in $\bar{s}$ because, instead of message $m'$, message $m$ was delivered. Then action $\text{perturb}_j$ arbitrarily perturbs only the state component of node $j$ from state $\bar{s}$, producing state $s''$. Again, $s''$ is exactly as $s'$ in $\alpha_1$, except possibly differing in the state components of node $j$. Recall that any perturbation of the state of $j$ is allowed by $\text{perturb}_j$. Thus, at least one such resulting $\beta$ has action $\text{perturb}_j$ that alters the state component of $j$ so that $s''$ in $\beta$ becomes identical to $s'$ in $\alpha$.

Now, since this particular $\beta$ ends in state $s'' = s'$ and since $\alpha_2$ begins with $s'$, $\beta$ and $\alpha_2$ can be concatenated to yield a new execution $\alpha' = \beta \circ \alpha_2$, where the traces of each node are exactly the same as in $\alpha$, except that node $j$ delivers $m$, followed immediately by the perturbation of its state. Moreover, $\alpha$ and $\alpha'$ are indistinguishable to any node (since only node $j$ is affected by the transformation), while node $j$ has no way of distinguishing the two executions because the state $\bar{s}$ cannot be examined by $j$ since it has no control over action $\text{perturb}_j$. 

41
For any execution $\alpha$ that contains corrupt actions, this transformation can be iteratively applied until no actions corrupt remain. The result is an execution that models message corruption through local state perturbations. This construction allows us to assume that messages are not corrupted in transit.

**Resource Discovery Invariant.** Here we state our algorithms’ invariant $\psi()$ that directly implies the RDP invariant in Definition 3.1.5. Notice, that invariant $\psi()$ may change depending on the algorithm. This being said, for the algorithms presented in this chapter, it is sufficient to have a unique invariant that is stated below.

Configuration $c \in C$ is a legitimate configuration if invariant $\psi(c)$ holds, where the invariant is defined as follows.

1. For every node $v \in V$ we have $active_v = true$ and $parent_v = \min\{u : u \in V\}$.

2. There exists a node $v_0 \in V$ such that $v_0 = \min\{u : u \in V\}$ and $Children_{v_0} = V$, while for every other node $w \neq v_0$, with $w \in V$, we have $Children_w = \emptyset$.

3. For every node $v \in V$ we have $world_v = V$.

**Notation Used in Algorithm Analysis.** In the analysis of our algorithms we use the following additional notations. We use $\tilde{\tau}$ to denote the transition function of the algorithm, with $\tilde{A}$ denoting the corresponding set of actions. We use $\tau$ to denote the transition function that excludes transitions corresponding to actions $join$ and $perturb$ that are caused exclusively by the environment; we use $A$ to denote the corresponding set of actions. In reasoning about the self-stabilization properties of executions we will consider only those executions where $join$ and $perturb$ occur only in some finite execution prefix.
3.3 Resource Discovery with Constrained Multicast

We specify the behavior of each node \(i \in V\) as a timed I/O automaton, called \(RD-M_i\) (for Resource Discovery with Multicast). The complete specification of \(RD-M_i\) is given in Figure 4 that contains definitions of data types, constants, signature, and state variables, and Figure 5 that defines the transitions and the trajectory. The full system, called \(RD-MS\), is the composition of automata \(RD-M_i\) for \(i \in V\), the multicast implementation, and the \(Channel_{i,j}\) automata for \(i, j \in V\).

Every node computes a local minimum based on its knowledge about other nodes, and it considers the node with the smallest identifier to be its parent. We note, that a node \(v\) considers \(u\) to be its parent only if \(v\) receives a message from \(u\) in the current iteration and \(u\) is the node with the smallest identifier among all nodes that contacted \(v\). Additionally, every node considers “prospective” parents, the nodes whose identifiers are no greater than its parent’s, but that may have not directly communicated with our node. In the gossip phase a node multicasts to its parent, prospective parent, children, and neighbors in \(G\), while in the confirm phase the node responds to the messages received in the gossip phase and to its neighbors in \(G^u\).

We now discuss the state and operation of \(RD-M_i\) in more detail. The main state variables of \(RD-M_i\) are \(active_i\), \(Children_i\), \(world_i\), \(Nbrs_i\), and \(parent_i\). The boolean \(active_i\) indicates whether node \(i\) is active or not, set \(Children_i\) contains the children of node \(i\), set \(world_i\) contains its siblings in the evolving knowledge graph, set \(Nbrs_i\) contains the identifiers of the nodes that \(i\) considers to be neighbors in \(G^u\), lastly \(parent_i\) is the identifier of the node that node \(i\) considers to be its parent. The remaining variables are used for control: \(phase_i\), \(pro\_parent_i\), \(New\_Chld_i\), \(do\_msend_i\), \(R_i\), and \(Dest_i\). Here \(phase_i\) controls whether
Data-types:
- \( U \), the set of node identifiers
- \( M \), the set of messages

Constants:
- \( nb : U \) outgoing neighbor of \( i \)
- \( t : \mathbb{R} > 0 \)

Derived Constants:
- \( \tilde{N} = \{ i \} \cup \{ nb \} \)

Signature:
- Input:
  - \( mrecv(m, u)_i, m \in M, u \in U \)
  - \( \text{join}_i \)
  - \( \text{perturb}_i \)
- Output:
  - \( \text{msend}(m, I)_i, m \in M, I \subseteq U \)

State:
- \( \text{active} : \text{bool} \)
- \( \text{phase} : \{ \text{gossip, confirm} \} \)
- \( \text{clock} : \mathbb{R} \)
- \( \text{do_msend} : \text{bool} \)
- \( \text{parent} : U \)
- \( \text{pro_parent} : U \)
- \( R : \mathbb{2}^U \)
- \( \text{Dest} : \mathbb{2}^U \)
- \( \text{world} : \mathbb{2}^U \)
- \( \text{Children} : \mathbb{2}^U \) set of children of \( i \)
- \( \text{New_Child} : \mathbb{2}^U \)
- \( \text{Nbrs} : \mathbb{2}^U \) set of all neighbors of \( i \)
- \( \text{ProP} : \mathbb{2}^U \)

Figure 4: Data types, signature, and state of RD-M\( i \) at node \( i \in V \)

the node is in the gossip or the confirm phase. \( \text{pro_parent}_i \) is the identifier of the node that node \( i \) considers to be its prospective parent. The set \( \text{New_Child}_i \) is used to keep \( \text{Children}_i \) up to date. The boolean \( \text{do_msend}_i \) is used to enable gossip multicast exactly once in each round. The set \( R_i \) contains the identifiers of all nodes from which node \( i \) received a message in the current iteration. And finally set \( \text{Dest}_i \) contains the identifiers of the target nodes for multicast in the current phase.

We next describe the transitions of the algorithm. The environment may activate node \( i \) by using input action \( \text{join}_i \), and it may disable and/or corrupt the state of node \( i \) by means of input action \( \text{perturb}_i \), where \( \text{HAVOC} \) assigns arbitrary values to the state variables, modeling a transient failure. If \( \text{HAVOC} \) sets \( \text{active} \) to \( \text{false} \), the action models a crash of a node. Internal action \( \text{restart}_i \) is always enabled, modeling the assumption that each node \( i \in V \) is eventually
### Transitions:

<table>
<thead>
<tr>
<th>Action</th>
<th>Precondition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input join&lt;sub&gt;i&lt;/sub&gt;</td>
<td>active</td>
<td>active ← true</td>
</tr>
<tr>
<td>Input perturb&lt;sub&gt;i&lt;/sub&gt;</td>
<td>active</td>
<td>HAVOC</td>
</tr>
<tr>
<td>Output msend((N, p, ch), I)&lt;sub&gt;i&lt;/sub&gt;</td>
<td>active</td>
<td>do_msend ← false</td>
</tr>
<tr>
<td>Input mrecv((N, p, ch), s)&lt;sub&gt;i&lt;/sub&gt;</td>
<td>active</td>
<td>do_msend ← true</td>
</tr>
<tr>
<td>Internal restart&lt;sub&gt;i&lt;/sub&gt;</td>
<td>active</td>
<td>active ← true</td>
</tr>
<tr>
<td>Internal end-round&lt;sub&gt;i&lt;/sub&gt;</td>
<td>active</td>
<td></td>
</tr>
</tbody>
</table>

**Precondition:**
- active
- clock \( \% t = 0 \)
- parent ∈ \( R \)

**Effect:**
- if clock \( \% 2t = 0 \) then /* start gossip phase */
  - parent ← min\( \{ u : u ∈ R ∪ \{i\} \} \)
  - R ← \( \emptyset \)
  - pro_parent ← min\( \{ u : u ∈ ProP ∪ \{i\} \} \)
  - ProP ← \( \emptyset \)
  - New_Chld ← \( \emptyset \)
  - Nbrs ← \( \hat{N} \)
  - Dest ← \{ parent \} ∪ \{ pro_parent \}
  - world ← ch

**Evolve:**
- clock ← clock + \( \epsilon \)

**Trajectories**

- stop when
  - active ∧ clock \( \% t = 0 \)
- evolve
  - \( d(\text{clock}) = 1 \)

---

**Figure 5:** Transitions of \( RD-M_i \) at node \( i ∈ V \)

Active. Nodes gossip by sending and receiving messages through actions msend<sub>i</sub> and mrecv<sub>i</sub> (detailed later).

The trajectory specification says that time “stops” when clock \( \% t = 0 \) for an active node.

The value of clock is used to determine whether an active node is in the gossip or confirm...
phase. When $\text{clock} \% 2t = 0$ the node enters the gossip phase, and when $\text{clock} \% 2t = t$ the node enters the confirm phase.

A round ends with either action end-round$_i$ or action reset$_i$. Action end-round$_i$ is enabled every $t$ time units when $\text{clock} \% t = 0$ at the conclusion of each round if the node’s state suggests that its parent is active (this does not mean that perturbations did not occur). Action reset$_i$ is enabled every $2t$ time units when $\text{clock} \% 2t = 0$ and the parent does not respond during the iteration. In this case the node gives up, resets its state and starts anew.

We now discuss action end-round$_i$ in detail. When $\text{clock} \% 2t = 0$ action end-round$_i$ concludes the current phase and starts a new iteration with the gossip phase. For this phase the variables are updated as follows: parent$_i$ is set to the smallest identifier among all nodes that sent a message to $i$ in the previous iteration and itself, and pro-parent$_i$ is set to the smallest identifier among itself and the parents of all nodes that sent a message to $i$, provided it is no greater than parent$_i$. Note that while updating parent$_i$ and pro-parent$_i$ we also consider the node itself just in case a failure causes $R_i$, or correspondingly ProP$_i$, to be empty. Sets $R_i$, ProP$_i$, and New_Chld$_i$ are set to $\emptyset$, since those sets reflect the corresponding knowledge of node $i$ in the current iteration. Set Nbrs$_i$ is set to the neighbors in the connectivity graph $G^u$, and finally Dest$_i$ is set to the destinations for the multicast in this new phase. Essentially, here the node establishes a parent and cleans up its state so as to not rely on variable values that may have been corrupted.

When $\text{clock} \% 2t = t$, action end-round$_i$ concludes the current phase and commences the confirm phase. In the confirm phase node $i$ propagates its knowledge to all nodes from which it received a message in the previous gossip phase and to the neighbors in $G^u$ that were
discovered in that phase. Additionally, the node \( i \) sets \( Children_i \) to the set of nodes, if any, that considered it the parent in the previous phase.

Note that the preconditions of actions end-round\(_i\) and reset\(_i\) are mutually exclusive. Each of these actions also cause \( clock \) to advance by \( \epsilon \) \((\epsilon \ll d)\), modeling the passage of time after it was “stopped” by the trajectory, and enable \( msend \) by setting \( do_msend \) to true.

We now detail actions \( msend_i \) and \( mrecv_i \) and explain certain subtle aspects of our algorithm and some of its control variables. Action \( msend_i \) is enabled at the beginning of every round and once invoked multicasts a message to the destination nodes in \( Dest_i \). The message contains the set of neighbors in the connectivity graph \( G \) (i.e., self and its fixed neighbor), the parent, and the set of children. Recall that destinations are established at the end of the previous phase. The action sets \( do_msend \) to false to prevent multiple invocations of \( msend_i \) in a round.

Action \( mrecv_i \) updates the state variables based on the message received from node \( s \). First, \( s \) is added to the set \( R_i \) that accumulates the identifiers of the nodes from whom messages are received. If node \( s \) considers node \( i \) to be a neighbor in the gossip phase, \( s \) is added to \( Nbrs_i \). Additionally, if node \( s \) considers node \( i \) to be its parent, it is added to \( New_Chld_i \). In the confirm phase node \( i \) updates the set of prospective parents. If the identifier of the parent of node \( s \) is no greater than the parent of \( i \), it is added to \( ProP_i \). Lastly, if node \( s \) is the parent of \( i \), the set \( world_i \) records the siblings of \( i \).

3.3.1 Algorithm Analysis

We now prove the self-stabilization properties of algorithm \( RD-MS \) and analyze its performance. We start our analysis with technical lemmas.
Lemma 3.3.1 Consider an execution prefix $\alpha$ of $RD-MS$ that ends with configuration $c$. Any fair extension of the execution of a sufficient length using only the actions from $A$ reaches a configuration $c^*$ in at most one complete iteration where in $c^*$: (1a) for every node $v \in V$ we have $active_v = true$, and (1b) for any two distinct nodes $u, v \in V$, such that $u \neq v$, we have $phase_u = phase_v = gossip$, and (2) in any further execution extension following $c^*$, any invocation of action $msend_v$ in a state with $phase_v = confirm$, results in a message sent to all neighbors of $v$ in $G^u$.

Proof. First we show that $\alpha$ can be finitely extended by a complete round so that configuration $c_1$ is reached, where clause (1a) holds. Let $\alpha_1$ be any such fair extended execution. Since restart is always enabled, then by fairness restart is invoked by the end of the round for every node $v \in V$. Therefore, in configuration $c_1$ we have $active_v = true$ for every node $v \in V$. Any fair execution extension includes an evolution of clock, enabling either end-round or reset action to be invoked at the end of the round. The evolution of clock results in either $clock \% 2t = t$ or $clock \% 2t = 0$ holding, and since by the end of the round for every node $v \in V$ we have $active_v = true$, it results in the invocation of either reset or end-round action for every node $v \in V$. Let $\alpha'_1$ be any such fair extended execution, and let $c'_1$ be the resulting configuration. There are two cases to consider.

Case (i): $clock \% 2t = 0$ in configuration $c_1$. Hence, configuration $c'_1$ is reached by invoking either reset or end-round, where in the latter case, the if clause of action end-round is executed. It follows that in configuration $c'_1$ we have $phase_v = gossip$ for every node $v \in V$. Hence, clause (1b) holds in configuration $c'_1$. In the later case we assume that $c'_1 = c^*$ and
we show that following configuration $c^*$ clause (2) also holds. Clearly in configuration $c^*$ for every node $v \in V$ we have: $do_{\text{msend}_v} = true$ and $\hat{N}_v \subseteq \text{Dest}_v$.

**Case (ii):** $\text{clock} \% 2t = t$ in configuration $c_1$. In this case configuration $c'_1$ is reached by invoking action end-round$_v$ for every node $v \in V$ following configuration $c_1$. From the specification of action end-round$_v$ it is also clear that the else clause of action end-round is executed. Hence, in configuration $c'_1$ for every node $v \in V$ we have $phase_v = \text{confirm}$. We fairly extend $\alpha'_1$ by a complete round reaching configuration $c^*$ by invoking either end-round$_v$ or reset$_v$ action at the end of the round for every node $v \in V$. Let $\alpha''_1$ be any such fair extended execution. From the specification of those actions it is clear that in configuration $c^*$ the following holds for every node $v \in V$: $do_{\text{msend}_v} = true$, $phase_v = \text{gossip}$ and $\hat{N}_v \subseteq \text{Dest}_v$. Hence, in configuration $c^*$ clause (1b) holds. It remains to show that following configuration $c^*$ clause (2) also holds.

Depending whether case(i) or case(ii) holds in configuration $c_1$, we fairly extend execution $\alpha'_1$ or $\alpha''_1$ respectively, by a complete round, reaching configuration $c_2$. We let $\alpha_2$ be the resulting execution. Since in configuration $c^*$ for every node $v \in V$ we have $do_{\text{msend}_v} = true$, action msend$_v$ is invoked by every node $v \in V$ at the beginning of the round and it results in a message being sent to every node $u \in \hat{N}_v$. This is because $\hat{N}_v \subseteq \text{Dest}_v$. Observe, that action mrecv is an input action and is always enabled. As we already showed prior to any invocation of mrecv$_u$ by a node $u \in V$ we have $\text{Nbrs}_u = \hat{N}_u$ and $phase_u = \text{gossip}$. By the end of the round every node $u \in V$ invokes mrecv$_u$, and since $phase_u = \text{gossip}$, this results in the inclusion in $\text{Nbrs}_u$ of all nodes $v$, such that $u \in \hat{N}_v$ for every node $v \in V$. Let $\alpha'_2$ be a fair extension of $\alpha_2$ such that configuration $c'_2$ is reached after the invocation of end-round$_v$ for every node $v \in V$. Note that action reset cannot be invoked, since $\text{clock} \% 2t = t$ in configuration $c_2$. Note also, that the else clause of the end-round action is executed, and hence, $\text{Nbrs}_v$ is
not reset to $\tilde{N}_v$. Therefore, in configuration $c'_2$ for every node $v \in V$ set $Nbrs_v$ contains the identifiers of all neighbors of $v$ in graph $G^u$. Additionally, in configuration $c'_2$ for every node $v \in V$ we have $phase_v = confirm$, $do_{msend_v} = true$, and $Nbrs_v \subseteq Dest_v$.

Finally we consider any further execution extension that includes action $msend_v$, and we examine the first occurrence of such action at a node $v \in V$ following $c'_2$. Since in configuration $c'_2$ for every node $v \in V$ set $Nbrs_v$ contains the identifiers of all neighbors of $v$ in graph $G^u$, and since $Nbrs_v \subseteq Dest_v$, this action results in a message being sent to all neighbors of $v$. Note also, that following configuration $c^*$ this was the first invocation of $msend_v$ such that $phase_v = confirm$ for any node $v \in V$, and hence clause (2) holds. This concludes the proof.

Lemma 3.3.2 Consider an execution prefix of $RD-MS$ that ends with configuration $c$. Any fair extension of the execution of a sufficient length that uses only the actions from $A$ reaches a configuration $c^*$ in at most two complete iterations, where for every node $v \in V$ the following holds: (a) $Children_v \subseteq V$, and (b) $parent_v \in V$, and (c) action $reset_v$ is not invoked following configuration $c^*$.

Proof. Let $\alpha$ be the execution prefix. Let execution $\alpha_1$ be an extension of $\alpha$, reaching configuration $c_1$ exactly as in Lemma 3.3.1.

Given that in configuration $c_1$ for every node $v \in V$ we have $phase_v = gossip$, and from the specification of actions reset and end-round, it follows that in $c_1$ we have $R_v = \emptyset$ and $New.Child_v = \emptyset$. Note that it is possible that in configuration $c_1$ there exists a node $w \in V$ such that $parent_w \notin V$. We want show that $\alpha_1$ can be finitely extended by a complete iteration so that a configuration $c^*$ is reached where clauses (a), (b) and (c) hold for every node $v \in V$. 
By fairness, after configuration $c_1$, actions $msend_v$ and $mrecv_v$ are invoked for every node $v \in V$. Let $c_2$ be the resulting configuration and let $\alpha_2$ be such an extended execution.

Note that $R_v$ and $New\_Chld_v$ are modified only when $mrecv_v$ is invoked. Moreover, $New\_Chld_v$ is modified only if $\text{phase}_v = \text{gossip}$. From the specification of action $mrecv_v$ it is clear that if a node $u$ is added to either $R_v$ or $New\_Chld_v$, then $u \in V$. Hence in configuration $c_2$ for every node $v \in V$ we have $New\_Chld_v \subseteq V$ and $R_v \subseteq V$. Since $\text{phase}_v = \text{gossip}$ in configuration $c_1$, it is clear that action reset cannot be invoked at the end of the round and action end-round is invoked instead. Let $\alpha'_2$ be a fair extension of $\alpha_2$ such that configuration $c'_2$ is reached after the invocation of end-round$_v$ for every node $v \in V$. From the specification of action end-round, and from the fact that for every node $v \in V$ in configuration $c_1$ we have $\text{phase}_v = \text{gossip}$, it is clear that the else clause of end-round action is executed, and hence $Children_v$ is set to $New\_Chld_v$ for every node $v \in V$. Therefore, in configuration $c'_2$ for every node $v \in V$ clause (a) holds. Additionally, in configuration $c'_2$ for every node $v \in V$ we have $R_v \subseteq V$ and $do\_msend_v = \text{true}$.

By fairness, after configuration $c'_2$, actions $msend_v$ and $mrecv_v$ are invoked for every node $v \in V$. Let us denote the resulting configuration by $c_3$ and let $\alpha_3$ be such an extended execution. From the specification of $mrecv_v$ it is clear that if a node $u$ is added to $R_v$, then $u \in V$. Hence, in configuration $c_3$ we have $R_v \subseteq V$ for every node $v \in V$. Let $\alpha^*$ be a fair extension of $\alpha_3$ such that configuration $c^*$ is reached after the invocation of either an end-round$_v$ or reset$_v$ for every node $v \in V$. Note that since in configuration $c_3$ we have $R_v \subseteq V$ and $\text{phase}_v = \text{confirm}$ for every node $v \in V$, then the existence of a node $w$ such that $\text{parent}_w \notin V$ implies that $\text{parent}_w \notin R_w$, since $R_w \subseteq V$. As a result, reset$_w$ is invoked that results in setting $\text{parent}_w$ to $w$, and hence, in configuration $c^*$ we have $\text{parent}_w \in V$ for
every node $w \in V$. This proves that there exists a configuration $c^*$, such that $c \xrightarrow{\tau} c_1 \xrightarrow{\tau} c^*$, where clauses (a) and (b) hold.

Next we show that configuration $c^*$ is such that action reset$_v$ is not invoked for any node $v \in V$ after $c^*$. Indeed, from the precondition of action reset it follows that for a node $v \in V$ action reset$_v$ is invoked when $\text{clock}\%2t = 0$ and $\text{parent}_v \notin R_v$. As we already showed in configuration $c^*$ for every node $v \in V$ we have $\text{parent}_v \in V$ and $\text{do_msend}_v = \text{true}$. By fairness, after configuration $c^*$, actions msend$_v$ and mrecv$_v$ are invoked for every node $v \in V$.

Let $c_4$ be the resulting configuration and let $\alpha_4$ be such an extended execution. Let us assume that following configuration $c^*$ there exists a node $w \in V$, such that $\text{parent}_w = w' \notin R_w$ when $\text{clock}\%2t = 0$. We want to show that such $w$ does not exist. Since in configuration $c^*$ we have $\text{parent}_w = w' \in V$ and $w' \in \text{Dest}_w$, it follows that in configuration $c_4$ we have $w \in R_{w'}$. Let $\alpha_5$ be a fair extension of $\alpha_4$ such that configuration $c_5$ is reached after the invocation of end-round$_v$ for every node $v \in V$. From the specification of action end-round it is clear that in $c_5$ we have $w \in \text{Dest}_{w'}$. By fairness, following configuration $c_5$, actions msend$_v$ and mrecv$_v$ are invoked for every node $v \in V$. Let $c_6$ be the resulting configuration and let $\alpha_6$ be such an extended execution. Since in configuration $c_5$ we have $w \in \text{Dest}_{w'}$, and from the specification of mrecv$_v$ it is clear that in $c_6$ we have $w' \in R_w$, hence even though the evolution of clock results in $\text{clock}\%2t = 0$, action reset$_w$ is not invoked since as we showed $\text{parent}_w = w' \in R_w$ in configuration $c_6$ and no node is removed from $R_w$ unless either end-round or reset action is invoked. Therefore, we showed that following configuration $c^*$ action reset$_v$ is not invoked for any node $v \in V$. This completes the proof. \qed
Corollary 3.3.3 Consider an execution prefix of RD-MS that ends with configuration $c$. Any fair extension of the execution of a sufficient length that uses only the actions from $A$ reaches a configuration $c^*$ in at most three complete iterations, where $pro\_parent_v \in V$ for every node $v \in V$.

Proof. Let $\alpha$ be the execution prefix. Let execution $\alpha_1$ be an extension of $\alpha$, reaching configuration $c_1$ exactly as in Lemma 3.3.2.

In configuration $c_1$ for every node $v \in V$ we have $parent_v \in V$. Additionally, from the specification of actions reset and end-round, it follows that in $c_1$ we have $ProP_v = \emptyset$. Note that it is possible that in configuration $c_1$ there exists a node $w \in V$ such that $pro\_parent_w \notin V$. We want to show that $\alpha_1$ can be finitely extended by a complete iteration so that a configuration $c^*$ is reached where $pro\_parent_v \in V$ for every node $v \in V$.

Let us extend execution $\alpha_1$ by a complete round, such that action end-round$_v$ is invoked for every node $v \in V$ at the end of the round. Let $c_2$ be the resulting configuration and let $\alpha_2$ be such an extended execution. Since no node is added to $ProP_v$ in the gossip phase, it follows that $ProP_v = \emptyset$ in configuration $c_2$ for every node $v \in V$.

By fairness, after configuration $c_2$, actions msend$_v$ and mrecv$_v$ are invoked for every node $v \in V$. Let $c_3$ be the resulting configuration and let $\alpha_3$ be such an extended execution. From the specification of action mrecv it follows that any node added to $ProP_v$ is a parent of some node $u \in V$. On the other hand, following configuration $c_1$, $parent_u$ is not updated for any node $u \in V$. While in configuration $c_1$ for every node $u \in V$ we have $parent_u \in V$. Hence, in configuration $c_3$ for every node $v \in V$ we have $ProP_v \subseteq V$. 

53
Any fair execution extension includes evolution of clock, enabling action end-round\(_v\) at the end of the round (action reset\(_v\) will not be invoked based on Lemma 3.3.2). Let \(\alpha_4\) be a fair extension of \(\alpha_3\) such that configuration \(c^*\) is reached after the invocation of end-round\(_v\) for every node \(v \in V\). From the specification of action end-round it follows that \(pro\_parent\_v\) is set to \(\min\{u : u \in ProP\_v \cup v\}\), for every node \(v \in V\). Therefore, \(pro\_parent\_v \in V\) for every node \(v \in V\). This is because \(ProP\_v \subseteq V\) in configuration \(c_3\) for every node \(v \in V\), and no node is added to \(ProP\_v\) following configuration \(c_3\). This completes the proof.

Next we show that after a constant number of rounds following the cessation of failures no message is sent to bogus destinations.

**Lemma 3.3.4** Consider an execution prefix of \(RD-MS\) that ends with configuration \(c\). Any fair extension of \(\alpha\) of a sufficient length that uses only the actions from \(A\) reaches a configuration \(c^*\) in at most three complete iterations, where no messages are sent to bogus destinations by any node \(v \in V\) following \(c^*\).

**Proof.** Let \(\alpha\) be the execution prefix. Let execution \(\alpha_1\) be an extension of \(\alpha\), reaching configuration \(c^*\) exactly as in Corollary 3.3.3.

The proof follows from the algorithm and from Lemmas 3.3.1 and 3.3.2, and Corollary 3.3.3. This is because according to the algorithm messages are only sent to the nodes in \(Dest\). Note that after \(c^*\) for every node \(i \in V\) we have \(parent\_i \in V\) and \(pro\_parent\_i \in V\). On the other hand, sets \(Nbrs_i\) and \(Children_i\) are reset in every iteration and any node \(v\) added to these sets after \(c^*\) is an active node that belongs to \(V\).

We now show that algorithm \(RD-MS\) satisfies the **Closure** and **Convergence** properties of Definition 3.1.2.
The proof of Theorem 3.3.5 is by induction on the length of the execution extension.

**Theorem 3.3.5 (Closure)** Consider any execution prefix of RD-MS consisting of complete iterations, where $c$ is the final configuration. If $c$ is a legitimate configuration, then any extension of the execution by up to one complete iteration using only the actions from $A$ results in $c \xrightarrow{\tau} c'$, where $c'$ is a legitimate configuration.

**Proof.** If $c$ is a legitimate configuration then the following holds for every node $v \in V$: $\text{world}_v = V$, $\text{parent}_v = \min\{u : u \in V\}$, and $\text{Children}_v = V$, while for every other node $w \neq v_0$ we have $\text{Children}_w = \emptyset$. Note that trajectory changes only the value of $\text{clock}$, and $\text{clock}$ does not appear in $\psi()$.

We proceed by induction on the length $k$ of the execution extension. For the base case, $k = 1$, let $c_1 = c$ and let us consider any transition with $c_1 \xrightarrow{\tau} c_2$. Since $\psi(c_1)$ holds, then for every node $v \in V$ we have $\text{parent}_v \in V$ in configuration $c_1$. Hence, according to Lemma 3.3.2, action reset cannot occur.

Actions mrecv and end-round cannot occur, since no messages have been sent, and the end of the round is not reached respectively. The only enabled discrete actions that can occur at the beginning of the round are msend and restart. These actions do not affect variables in $\psi$, and thus $\psi(c_2)$ holds. For the inductive step we assume that all extensions of length $k$ preserve the invariant, and we consider the discrete action $a_{k+1}$; we show that the corresponding transition with $c_{k+1} \xrightarrow{\tau} c_{k+2}$ reaches a legitimate configuration $c_{k+2}$.

We now consider each discrete action $a$ in $\tau$ at a node $v \in V$ when $\text{phase}_v = \text{gossip}$.

$a_{k+1} = \text{restart}_v$: Since $\psi(c_{k+1})$ holds, $\text{active}_v = \text{true}$, and thus $c_{k+2} = c_{k+1}$ and the invariant holds.
$a_{k+1} = \text{msend}(\langle N, p, ch \rangle, I)_v$: This action is enabled at the very beginning of a round and it does not change any variables occurring in $\psi$, thus the invariant is maintained.

We also note that since $c_{k+1}$ is a legitimate configuration, then, by fairness, every node $v \in V$ invokes action $\text{msend}(\langle N, p, ch \rangle, I)_v$ at the beginning of the round. Since $\text{phase}_v = \text{gossip}$, set $I$ consists of its parent, prospective parent, neighbors in $G$, and children, where $p = v_0$.

$a_{k+1} = \text{mrecv}(\langle N, p, ch \rangle, s)_v$: This is an input action, and it is always enabled. Given that $\text{msend}$ occurs as described above, $v$ receives the corresponding message from $s$ that is a child, a prospective child, a neighbor, or a parent. The transition may change $R_v$, but this does not affect $\psi$. Next, if $p = v$, that is, node $s$ considers node $v$ to be its parent, then $\psi(c_{k+1})$ implies that $v = v_0$, and thus $s$ is included in $\text{New Chld}_v$. If $p \neq v$, then $\text{New Chld}_v = \emptyset$ since $\text{phase}_v = \text{gossip}$. Since action $\text{mrecv}$ does not affect variables in $\psi$ when $\text{phase}_v = \text{gossip}$ the invariant $\psi(c_{k+2})$ holds.

We also note that all messages sent at the beginning of the current round are delivered before the end of the round. Since $c_{k+1}$ is a legitimate configuration, then $\psi(c_{k+1})$ implies that $\text{Children}_{v_0} = V$, and hence every node $v \in V$ receives a message from $v_0$ and adds $v_0$ to $R_v$. Additionally, since $\text{parent}_v = v_0$ for every node $v \in V$, it follows that $R_{v_0} = V$. Thus $v$ receives messages from all its children (if any) by the model assumption and since no perturbations occur. Hence, by the end of the round $\text{New Chld}_{v_0} = V$, while for every other node $u \neq v_0$ we have $\text{New Chld}_u = \emptyset$.

$a_{k+1} = \text{end-round}_v$: Since, $\text{phase}_v = \text{gossip}$ for every node $v \in V$, it follows that else clause of end-round action is executed. In configuration $c_{k+1}$ for every node $v \neq v_0$ we have $\text{Children}_v = \text{New Chld}_v = \emptyset$, while for node $v_0$ we have $\text{Children}_{v_0} = \text{New Chld}_{v_0} = V$. Hence, $\psi(c_{k+2})$ holds.
We now consider each discrete action $a$ in $\tau$ at a node $v \in V$ when $\text{phase}_v = \text{confirm}$.

$a_{k+1} = \text{restart}_v$: Since $\psi(c_{k+1})$ holds, $active_v = true$, and thus $c_{k+2} = c_{k+1}$ and the invariant holds.

$a_{k+1} = \text{msend}(\langle N, p, ch \rangle, I)_v$: This action is enabled at the very beginning of a round and it does not change any variables occurring in $\psi$, thus the invariant is maintained.

We also note that since $\psi(c_{k+1})$ holds, then, by fairness, every node $v \in V$ invokes action $\text{msend}(\langle N, p, ch \rangle, I)_v$ at the beginning of the round. Since $\text{phase}_v = \text{confirm}$, set $I$ consists of all nodes that sent a message to $v$ in the previous round and neighbors in $G^u$ (see Lemma 3.3.1).

$a_{k+1} = \text{mrecv}(\langle N, p, ch \rangle, s)_v$: This is an input action, and it is always enabled. Given that $\text{msend}$ occurs as described above, $v$ receives the corresponding message from $s$ that is a parent, a prospective parent, a child, or a neighbor. The transition may change $R_v$, but this does not affect $\psi$, since node $u$ is included in $R_v$ only if $v$ receives a message from $u$, and hence $u \in V$ and $R_v \subseteq V$. On the other hand, since the execution prefix of $RD-MS$ consists of complete iterations, it is clear that if a node $v$ invokes $\text{mrecv}_v$, where $\text{phase}_v = \text{confirm}$, then prior to that every node $v \in V$ already received all messages sent to it when $\text{phase}_v = \text{gossip}$. Hence, as we already argued $v_0 \in R_v$ for every node $v \in V$, and moreover $R_{v_0} = V$. Next, if $parent_v = s$, then $\psi(c_{k+1})$ implies that $s = v_0$, and thus $\text{world}_v = V$.

We also note that all messages sent at the beginning of the current round are delivered by the end of the round. Since action $\text{mrecv}$ does not affect $\text{Children}_v$, and since $\text{world}_v = V$, the invariant $\psi(c_{k+2})$ holds.

$a_{k+1} = \text{end-round}_v$: Since, $\text{phase}_v = \text{confirm}$ for every node $v \in V$, it follows that the if clause of end-round action is executed. Since $\psi(c_{k+1})$ holds and since for every node $v \in V$
we have \( v_0 \in R_v \subseteq V \), it follows that \( parent_v = \min\{ u : u \in R \cup v \} = v_0 \). And hence, configuration \( c_{k+2} = c' \) is reached where \( \psi(c') \) holds. This concludes the proof. \( \square \)

We next address the convergence property, starting with a preparatory lemma.

**Lemma 3.3.6** Consider an execution prefix of \( RD-MS \) that ends with configuration \( c \). Any fair extension of the execution of a sufficient length that uses only the actions from \( A \) reaches a configuration \( c^* \) in at most \( 2D + 1 \) complete iterations, such that in configuration \( c^* \) for every node \( v \in V \) we have \( parent_v = \min\{ u : u \in V \} \).

**Proof.** Let \( \alpha \) be the execution prefix. Let execution \( \alpha_1 \) be an extension of \( \alpha \), reaching configuration \( c_1 \) exactly as in Lemma 3.3.2.

Let \( \alpha_2 \) be a fair extension of \( \alpha_1 \) by a complete iteration such that a configuration \( c_2 \) is reached after the invocation of \( end-round_v \) for every node \( v \in V \). Consider a prefix of \( \alpha_2 \), ending in configuration \( c'_1 \), just before the first \( end-round_v \) action is invoked by any node \( v \in V \). By Lemma 3.3.1 every node \( v \in V \) sends a message to all neighbors in \( G^u \) when \( phase_v = \text{confirm} \). As a result in configuration \( c'_1 \) for every node \( u \in V \) such that \( \text{dist}(u, v_0) = 1 \) we have \( v_0 \in R_u \). Note that, following configuration \( c'_1 \), configuration \( c_2 \) in \( \alpha_2 \) is reached by executing the if clause of the end-round action, and as a result, in configuration \( c_2 \) for every node \( u \in V \) such that \( \text{dist}(u, v_0) = 1 \) in \( G^u \), we have \( parent_u = v_0 \).

We want to show that following configuration \( c_2 \), it takes at most two iterations for every node \( u \in V \), such that \( \text{dist}(u, v_0) = 2 \) in \( G^u \), to have \( parent_u = v_0 \). Indeed, let \( \alpha_3 \) be a fair extension of \( \alpha_2 \) by a complete iteration such that configuration \( c_3 \) is reached after the invocation of \( end-round_v \) for every node \( v \in V \). Consider a prefix of \( \alpha_3 \), ending in configuration \( c'_2 \), just before the first \( end-round_v \) action is invoked by any node \( v \in V \). We already
showed that in configuration $c_2$ for every node $u \in V$ such that $\text{dist}(u, v_0) = 1$ in $G^u$ we have $\text{parent}_u = v_0$. From the specification of actions $\text{msend}$ and $\text{mrecv}$ it is clear that in configuration $c'_2$ for every node $u \in V$ such that $\text{dist}(u, v_0) = 2$ we have $v_0 \in \text{ProP}_u$. From the specification of action end-round, and from the fact that configuration $c_3$, that follows $c'_2$ in $\alpha_3$, is reached by executing the if clause of the end-round action, it follows that in configuration $c_3$ for every node $u \in V$ such that $\text{dist}(u, v_0) = 2$, we have $\text{pro}_{\text{parent}}_u = v_0$ and $v_0 \in \text{Dest}_u$.

Let us further fairly extend execution $\alpha_3$ by a complete iteration such that configuration $c_4$ is reached after the invocation of end-round, for every node $v \in V$. Let $\alpha_4$ be such an extended execution. Consider a prefix of $\alpha_4$, ending in configuration $c'_3$, just before the first end-round action is invoked by any node $v \in V$. Since in configuration $c_3$ for every node $u \in V$ such that $\text{dist}(u, v_0) = 2$, we have $v_0 \in \text{Dest}_u$, then from the specification of action $\text{mrecv}$, it follows that $v_0 \in R_u$ in configuration $c'_3$. From the specification of action end-round, and from the fact that configuration $c_4$, that follows $c'_3$ in $\alpha_4$, is reached by executing the if clause of the end-round action, it follows that in configuration $c_4$ for every node $u \in V$ such that $\text{dist}(u, v_0) = 2$, we have $\text{parent}_u = v_0$.

Hence, each subsequent finite extension of $\alpha_4$ by two complete iterations results in $\text{parent}_v = v_0$ for every $v \in V$, such that $\text{dist}(v, v_0) \geq \text{dist}(u, v_0) + 1$ in $G^u$, provided that $\text{parent}_u = v_0$ in configuration $c_4$. It follows that by finitely extending $\alpha_4$ by at most $2(D - 2)$ complete iterations configuration $c^*$ is reached such that for every node $v \in V$ in $c^*$ we have $\text{parent}_v = v_0$. This completes the proof. $\square$

**Theorem 3.3.7 (Convergence)** Consider an execution prefix of $RD-MS$ that ends with configuration $c$. Any fair extension of the execution of a sufficient length that uses only the actions
from $A$ reaches a configuration $c_l$ in at most $2D + 2$ complete iterations, such that $c_l$ is a legitimate configuration.

**Proof.** Let $\alpha$ be the execution prefix. Let execution $\alpha_1$ be an extension of $\alpha$, reaching configuration $c_1$ exactly as in Lemma 3.3.6.

According to Lemma 3.3.6, in configuration $c_1$ for every node $v \in V$ we have $parent_v = v_0$, where $v_0 = \min\{u : u \in V\}$. Clearly in configuration $c_1$ clause (1) of invariant $\psi$ holds. It remains to show that execution $\alpha_1$ can be finitely extended so that a configuration $c_l$ is reached such that $\psi(c_l)$ holds. First we show that execution $\alpha_1$ can be fairly extended by a complete round such that a configuration $c'$ is reached where clause (2) of the invariant $\psi$ holds. By fairness, after configuration $c_1$, actions $\text{msend}_v$ and $\text{mrecv}_v$ are invoked for every node $v \in V$. Let us denote the resulting configuration by $c_2$ and let $\alpha_2$ be such an extended execution.

We want to examine the state variable $\text{New}_\text{Chld}_v$ for every node $v \in V$ in configuration $c_2$. From the specification of the end-round action it follows that in configuration $c_1$ for every node $v \in V$ we have $R_v = \emptyset$, $\text{New}_\text{Chld}_v = \emptyset$, $\text{phase}_v = \text{gossip}$, and $v_0 \in \text{Dest}_v$. On the other hand, from the specification of action $\text{mrecv}$ and from the fact that in configuration $c_1$ for every node $v \in V$ we have $parent_v = v_0$, it is clear that in configuration $c_2$ we have $\text{New}_\text{Chld}_{v_0} = V$, while for every other node $u \neq v_0$ we have $\text{New}_\text{Chld}_u = \emptyset$. Note also, that if a node $u$ is added to $R_v$, then $v$ received a message from $u$, and hence $R_v \subseteq V$. Furthermore, since $v_0$ receives a message from every node $v \in V$, it is also clear that in configuration $c_2$ we have $R_{v_0} = V$.

Let $\alpha'$ be a fair extension of $\alpha_2$ such that a configuration $c'$ is reached after the invocation of the end-round action for every node $v \in V$. Since in configuration $c'$ for every node
\( v \in V \) we have \( phase_v = \text{gossip} \), coupled with the specification of action \( \text{end-round} \), it follows that the \textbf{else} clause of the \text{end-round} is executed. Hence, in configuration \( c' \) we have \( \text{Children}_{v_0} = V \), while for every node \( u \neq v_0 \) we have \( \text{Children}_u = \emptyset \). Therefore clause (2) holds in configuration \( c' \). Note also, that in configuration \( c' \) we have \( \text{Dest}_{v_0} = V \), since \( R_{v_0} = V \) in configuration \( c_2 \).

By fairness, after configuration \( c' \), actions \( \text{msend}_v \) and \( \text{mrecv}_v \) are invoked for every node \( v \in V \). Let us denote the resulting configuration by \( c_3 \) and let \( \alpha_3 \) be such an extended execution. We want to examine the state variable \( \text{world}_v \) for every node \( v \in V \) in configuration \( c_3 \).

In configuration \( c' \) we have \( \text{Dest}_{v_0} = V \), \( \text{Children}_{v_0} = V \), while for every node \( v \in V \) we have \( \text{parent}_v = v_0 \) and \( \text{phase}_v = \text{confirm} \). Hence, from the specification of actions \( \text{msend} \) and \( \text{mrecv} \), it follows that in configuration \( c_3 \) for every node \( v \in V \) we have \( \text{world}_v = V \), and \( v_0 \in R_v \). Execution \( \alpha_3 \) can be extended by a trajectory that increases the value of \( \text{clock} \), so that ultimately \( \text{clock} \mod t = 0 \) for every node \( v \in V \), which in turn enables \( \text{end-round}_v \) for all \( v \in V \). Let us further extend \( \alpha_3 \) such that configuration \( c_l \) is reached by invoking action \( \text{end-round}_v \) for all nodes \( v \in V \). Since in configuration \( c_3 \) for every node \( v \in V \) we have \( \text{phase}_v = \text{confirm} \), it follows that the \textbf{if} clause of \text{end-round} action is executed. Note that for every node \( v \in V \), the only variable that appears in \( \psi \) and is modified in the \textbf{if} clause is \( \text{parent}_v \). However, as we showed, in configuration \( c_3 \) for every node \( v \in V \) we have \( v_0 \in R_v \), where \( R_v \subseteq V \), and hence \( \text{parent}_v = v_0 \). Therefore, \( \psi(c_l) \) holds, hence, \( c_l \) is a legitimate configuration. This completes the proof.

Finally we reason about the stabilization time and message complexity.
Theorem 3.3.8 Any execution prefix of \textit{RD-MS} ending in an arbitrary configuration can be infinitely extended to solve the resource discovery problem. The stabilization time of the algorithm is \(O(D)\), taking at most \(4D + 4\) complete rounds to stabilize. The stabilization message complexity is \(O(|V| \cdot D)\).

Proof. From the proofs of Lemma 3.3.2, Lemma 3.3.6, and Theorem 3.3.7 it follows that after the cessation of transient failures, and given that no new nodes join the computation, any fair execution extension of a sufficient length takes at most \(2D + 2\) complete iterations to reach a legitimate configuration from any configuration \(c\), and hence \(4D + 4\) complete rounds. Closure Theorem 3.3.5 guarantees that all subsequent configurations are legitimate. Thus, the algorithm establishes the resource discovery invariant (Definition 3.1.5), taking at most \(4D + 4\) complete rounds to stabilize, and then maintains the resource discovery invariant in perpetuity.

We now assess the stabilization message complexity. Consider some execution that includes an arbitrary configuration after which no actions from \(\mathcal{A} - A\) occur. Lemma 3.3.4 shows that in a constant number of rounds no message is sent to any node identifier from \(U - V\). Prior to this the algorithm may send messages to arbitrary sets of nodes due to the corruptions of local states. We charge such messages to the environment (per model assumptions) and do not include them in the message complexity. Here we only consider messages sent to the nodes in \(V\). From the specification of end-round it follows that if \(\text{phase}_v = \text{gossip}\) then every node \(v\) sends a message to its neighbors in \(G\), to its parent, to its prospective parent, and to its children, if any. Note also, that if \(u \in \text{Children}_v\), then \(u \notin \text{Children}_w\) for any node \(w \neq v\), because every node has only one parent. Thus, in the gossip phase the number of accountable messages is \(O(|V|)\). From the specification of action \texttt{mrecv} it follows that for
every node $v \in V$, we have $R_v \subseteq V$. Specifically, for a node $v \in V$ set $R_v$ contains set $N'$ of nodes, if any, that consider $v$ to be their parent, set $N''$ of nodes, if any, that consider $v$ to be their prospective parent, a neighbor in $G$, and a parent of $v$ from the previous iteration. Note also, that if for nodes $u, v, w \in V$, such that $v \neq w$, we have $u \in N'_v$ or $u \in N''_v$, then $u \notin N'_w$ or $u \notin N''_w$ respectively. This is because if node $u$ considers $v$ to be its parent, then $w$ cannot be the parent of $u$. The same goes for the prospective parent. On the other hand, from Lemma 3.3.1 it follows that $Nbrs_v$ contains all the neighbors of node $v$ in $G^u$. However, since $|\widehat{N}_v| = 2$, for every node $v \in V$, where $v \in \widehat{N}_v$, it follows that the total number of messages sent along the edges of $G^u$ cannot exceed $3|V|$; here we also count messages sent by every node to itself. Thus, in the confirm phase the number of accountable messages is $O(|V|)$. Therefore, since the stabilization time is $O(D)$, the message complexity of $RD-MS$ is $O(|V| \cdot D)$.

\[ \square \]

### 3.4 Resource Discovery with Aggressive Broadcast

In this section we present an algorithm that uses a more aggressive propagation mechanism than the one presented in the previous section, resulting in a higher message complexity. However, aggressive broadcast allows the algorithm to converge to a legitimate configuration faster, asymptotically meeting the lower bound [47].

The behavior of each node $i \in V$ is specified as a timed I/O automaton, called $RD-B_i$ (for Resource Discovery with Broadcast). The specification in Figure 6 defines data types, constants, signature, and state variables, and Figure 7 contains the definition of the transitions and the trajectory. The full system, called $RD-BS$, is the composition of automata $RD-B_i$ for $i \in V$, the multicast implementation, and the $Channel_{i,j}$ automata for $i, j \in V$. 63
Data-types:
\[ U \], the set of node identifiers
\[ M \], the set of messages

Constants:
\[ nb : U \] outgoing neighbor of \( i \)
\[ t : \text{real} > 0 \]

Derived Constants:
\[ \mathcal{N} = \{ i \} \cup \{ nb \} \]

Signature:
Input:
\[ \text{mrecv}(m, u_i), m \in M, u \in U \]
\[ \text{join}_i \]
\[ \text{perturb}_i \]
Output:
\[ \text{msend}(m, I_i), m \in M, I \subseteq U \]

State:
\[ \text{active} : \text{bool} \]
\[ \text{phase} : \{ \text{gossip}, \text{confirm} \} \]
\[ \text{clock} : \text{real} \]
\[ \text{do_msend} : \text{bool} \]
\[ \text{parent} : U \]

\[ R : 2^U \]
\[ \text{Dest} : 2^U \]
\[ \text{world} : 2^U \]
\[ \text{Children} : 2^U \text{ set of children of } i \]
\[ \text{New_Child} : 2^U \]
\[ \text{Nbrs} : 2^U \text{ set of all neighbors of } i \]

Figure 6: Data types, signature, and state of \( RD-B_i \) at node \( i \in V \)

In the gossip phase a node multicasts to its parent, children, a neighbor in \( G \), and to all other nodes it discovers, and in the confirm phase each node responds to the messages received in the gossip phase and to its neighbors. We now detail the state and operation of \( RD-B_i \).

The main variables are \( \text{active}_i \), \( \text{Children}_i \), \( \text{world}_i \), \( \text{Nbrs}_i \), and \( \text{parent}_i \). Boolean \( \text{active}_i \) indicates whether node \( i \) is active or not, set \( \text{Children}_i \) contains the children of node \( i \), set \( \text{world}_i \) contains the universe known to \( i \), set \( \text{Nbrs}_i \) contains the identifiers of the nodes that \( i \) considers to be neighbors in \( G^u \), lastly \( \text{parent}_i \) is the identifier of the node that node \( i \) views as its parent. The remaining variables are \( \text{phase}_i \), \( \text{New_Child}_i \), \( \text{do_msend}_i \), \( R_i \), and \( \text{Dest}_i \). Here \( \text{phase}_i \) controls the phase (gossip or confirm). Set \( \text{New_Child}_i \) is used to keep \( \text{Children}_i \) up to date. Boolean \( \text{do_msend}_i \) enables multicast exactly once in each round. Set \( R_i \) contains the identifiers of all nodes that contacted node \( i \) in the current iteration. And finally set \( \text{Dest}_i \) contains the identifiers of the target nodes for multicast in the current phase.
Transitions:
Input join_\textsubscript{i}
  Effect:
  \textit{active} \leftarrow \textit{true}

Input perturb_\textsubscript{i}
  Effect:
  HAVOC

Output msend(⟨N, p, W⟩, I)_\textsubscript{i}
  Precondition:
  active
  do_msend
  N = \bar{N}
  p = parent
  W = world
  I = Dest
  Effect:
  do_msend \leftarrow false

Input mrecv(⟨N, p, W⟩, s)_\textsubscript{i}
  Effect:
  if active then
    R \leftarrow R \cup \{s\}
  if phase = gossip then
    if i \in N then
      Nbrs \leftarrow Nbrs \cup \{s\}
    if p = i then
      New_Child \leftarrow New_Child \cup \{s\}
  if phase = confirm then
    world \leftarrow world \cup W \cup \{s\}

Trajectories
stop when
  active \land \text{clock} \% t = 0
  evolve
  \text{d(clock)} = 1

Internal restart_\textsubscript{i}
  Effect:
  active \leftarrow \textit{true}

Internal end-round_\textsubscript{i}
  Precondition:
  active
  \text{clock} \% t = 0
  \text{clock} \% 2t = t \lor parent \in R
  Effect:
  if \text{clock} \% 2t = 0 then /* gossip phase */
    parent \leftarrow \min \{u : u \in R \cup \{i\}\}
    R \leftarrow \emptyset
    New_Child \leftarrow \emptyset
    Nbrs \leftarrow \bar{N}
    Dest \leftarrow \{parent\} \cup Nbrs \cup Children \cup world
    phase \leftarrow gossip
  else /* confirm phase */
    world \leftarrow R \cup Nbrs
    Children \leftarrow New_Child
    Dest \leftarrow R \cup Nbrs
    phase \leftarrow confirm
    do_msend \leftarrow true
    \text{clock} \leftarrow \text{clock} + \epsilon

Internal reset_\textsubscript{i}
  Precondition:
  active \land \text{clock} \% 2t = 0 \land parent \notin R
  Effect:
  parent \leftarrow i
  world \leftarrow R \leftarrow \emptyset
  Children \leftarrow New_Child \leftarrow \emptyset
  Nbrs \leftarrow \bar{N}
  Dest \leftarrow \{parent\} \cup Nbrs \cup Children
  phase \leftarrow gossip
  do_msend \leftarrow true
  \text{clock} \leftarrow \text{clock} + \epsilon

Figure 7: Transitions of \textit{RD-B}_i at node \textit{i} \in V

We next describe the transitions. The environment may activate node \textit{i} by using the input action join_\textsubscript{i}, and it may disable and/or corrupt the state of node \textit{i} by means of the input action perturb_\textsubscript{i}, where HAVOC assigns arbitrary values to the state variables. If HAVOC sets \textit{active} to \textit{false}, the action models a crash of a node. Internal action restart_\textsubscript{i} is always enabled, modeling the assumption that each node \textit{i} \in V is eventually active. Nodes gossip by sending and receiving messages through actions msend_\textsubscript{i} and mrecv_\textsubscript{i} (detailed later).
The trajectory specification says that time “stops” when $\text{clock} \% t = 0$ for an active node. The value of clock is used to determine whether an active node is in the gossip or confirm phase. When $\text{clock} \% 2t = 0$ the node enters the gossip phase, and when $\text{clock} \% 2t = t$ the node enters the confirm phase.

A round ends with either action end-round$_i$, or action reset$_i$. Action end-round$_i$ is enabled every $t$ time units when $\text{clock} \% t = 0$ at the conclusion of each round if the node’s state suggests that its parent is active (this does not mean that perturbations did not occur). Action reset$_i$ is enabled every $2t$ time units when $\text{clock} \% 2t = 0$ and the parent does not respond during the iteration. In this case the node gives up, resets its state and starts anew.

In more detail, when $\text{clock} \% 2t = 0$ action end-round$_i$ concludes the current phase and starts a new iteration with the gossip phase. For this phase the variables are updated as follows: $\text{parent}_i$ is set to the smallest identifier among the nodes that sent a message to $i$ in the previous iteration and itself. Note that while updating $\text{parent}_i$ we also consider the node itself just in case a failure causes $R_i$ to be empty. Sets $R_i$ and $\text{New Child}_i$ are set to $\emptyset$, since those sets reflect the corresponding knowledge of node $i$ in the current iteration. Set $\text{Nbrs}_i$ is set to the neighbors in graph $G$ (i.e., self and its fixed neighbor), and finally $\text{Dest}_i$ is set to the destinations for the multicast in this new phase. Essentially, here the node establishes a parent and cleans up its state so as to not rely on variable values that may have been corrupted.

When $\text{clock} \% 2t = t$, action end-round$_i$ concludes the current phase and commences the confirm phase. In the confirm phase node $i$ propagates its knowledge to all nodes from which it received a message in the previous gossip phase and to the neighbors in $G^u$ that were discovered in that phase. Node $i$ also sets $\text{Children}_i$ to the set of nodes that considered it the
parent in the previous phase. Furthermore, node $i$ sets $\text{world}_i$ to the set of nodes from whom messages were received in the gossip phase united with the neighbors of $i$ in graph $G^u$.

Note that the preconditions of actions end-round$_i$ and reset$_i$ are mutually exclusive. Each of these actions also cause $\text{clock}$ to advance by $\epsilon$ ($\epsilon \ll d$), modeling the passage of time after it was “stopped” by the trajectory, and enable $\text{msend}$ by setting $\text{do}_i\text{msend}$ to true.

We now detail $\text{msend}_i$ and $\text{mrecv}_i$. Action $\text{msend}_i$ is enabled at the beginning of every round and its invocation multicasts a message from node $i$ to the nodes in $\text{Dest}_i$. The message contains the set of neighbors in graph $G$ (i.e., self and its fixed neighbor), the parent, and $\text{world}_i$. Recall that destinations are established at the end of the previous phase. The action sets $\text{do}_i\text{msend}$ to false to prevent multiple invocations of $\text{msend}_i$ in a round.

Action $\text{mrecv}_i$ updates the state based on the messages received. First, the set $R_i$ accumulates the identifiers of the nodes from whom messages are received. Additionally, if a message is received from node $s$ and $s$ considers node $i$ to be a neighbor in the gossip phase, then $s$ is added to $\text{Nbrs}_i$. If node $s$ considers node $i$ to be its parent, it is added to $\text{New\_Child}_i$. In the confirm phase node $i$ also updates the set $\text{world}_i$ by including node $s$ and $\text{world}_s$ received from $s$.

### 3.4.1 Algorithm Analysis

We now prove the self-stabilization properties of algorithm $\text{RD-BS}$ and analyze its performance. We start our analysis with technical lemmas.

**Lemma 3.4.1** Consider an execution prefix of $\text{RD-BS}$ that ends with configuration $c$. Any fair extension of the execution of a sufficient length using only the actions from $A$ reaches configuration $c^*$ in at most one complete iteration, where in $c^*$: (1a) for every node $v \in V$
we have \( active_v = true \), and (1b) for any two distinct nodes \( u, v \in V \) we have \( phase_u = phase_v = gossip \), and (2) in any further execution extension following \( c^* \), any invocation of action \( msend_v \) in a state with \( phase_v = confirm \), results in a message sent to all neighbors of \( v \) in \( G^n \).

**Proof.** The proof of this lemma follows the exact steps of the proof of Lemma 3.3.1. \( \square \)

**Lemma 3.4.2** Consider an execution prefix of RD-BS that ends with configuration \( c \). Any fair extension of the execution of a sufficient length using only the actions from \( A \) reaches configuration \( c^* \) in at most two complete iterations, where for every node \( v \in V \) the following holds: (a) \( Children_v \subseteq V \) and \( world_v \subseteq V \), and (b) \( parent_v \in V \), and (c) action \( reset_v \) is not invoked following configuration \( c^* \).

**Proof.** Let \( \alpha \) be the execution prefix. Let execution \( \alpha_1 \) be an extension of \( \alpha \), reaching configuration \( c_1 \) exactly as in Lemma 3.4.1.

Given that in configuration \( c_1 \) for every node \( v \in V \) we have \( phase_v = gossip \), and from the specification of actions \( reset \) and end-round, it follows that in \( c_1 \) we have \( New\_Chld_v = \emptyset \) and \( R_v = \emptyset \). Note that it is possible that in configuration \( c_1 \) there exists a node \( w \in V \) such that \( parent_w \notin V \). We want to show that \( \alpha_1 \) can be finitely extended by a complete iteration so that a configuration \( c^* \) is reached where clauses (a), (b) and (c) hold for every node \( v \in V \).

By fairness, after configuration \( c_1 \), actions \( msend_v \) and \( mrecv_v \) are invoked for every node \( v \in V \). Let \( c_2 \) be the resulting configuration and let \( \alpha_2 \) be such an extended execution.

Note that \( R_v, Nbrs_v \) and \( New\_Chld_v \) are modified only when \( mrecv_v \) is invoked. Moreover, \( Nbrs_v \) and \( New\_Chld_v \) are modified only if \( phase_v = gossip \). From the specification of action \( mrecv \) it is clear that if a node \( u \) is added to either \( R_v \), or \( Nbrs_v \) or \( New\_Chld_v \), then
$u \in V$. Hence in configuration $c_2$ for every node $v \in V$ we have $\text{New\_Chld}_v \subseteq V$, $R_v \subseteq V$, and $\text{Nbrs}_v \subseteq V$. Since $\text{phase}_v = \text{gossip}$ in configuration $c_1$, it is clear that action reset cannot be invoked at the end of the round and action end-round will be invoked instead. Let $\alpha'_2$ be a fair extension of $\alpha_2$ such that configuration $c'_2$ is reached after the invocation of end-round$_v$ for every node $v \in V$. From the specification of action end-round it is clear that the else clause is executed, and hence $\text{Children}_v$ is set to $\text{New\_Chld}_v$ for every node $v \in V$. Additionally, $\text{world}_v$ is set to $R_v \cup \text{Nbrs}_v$ for every node $v \in V$, and hence $\text{world}_v \subseteq V$. Therefore, in configuration $c'_2$ for every node $v \in V$ clause (a) holds. In addition, in configuration $c'_2$ for every node $v \in V$ we have $\text{do\_msend}_v = \text{true}$.

By fairness, after configuration $c'_2$, actions msend$_v$ and mrecv$_v$ are invoked for every node $v \in V$. Let us denote the resulting configuration by $c_3$ and let $\alpha_3$ be such an extended execution. From the specification of action mrecv$_v$ it is clear that if a node $u$ is added to $R_v$, then $u \in V$. Hence in $c_3$ we have $R_v \subseteq V$ for every node $v \in V$. Let $\alpha^*$ be a fair extension of $\alpha_3$ such that configuration $c^*$ is reached after the invocation of either an end-round$_v$ or reset$_v$ for every node $v \in V$. Note that since in $c_3$ we have $R_v \subseteq V$ and $\text{phase}_v = \text{confirm}$ for every node $v \in V$, then the existence of a node $w$ such that $\text{parent}_w \notin V$ implies that $\text{parent}_w \notin R_w$, since $R_w \subseteq V$. As a result, reset$_w$ is invoked that results in setting $\text{parent}_w$ to $w$, and hence, in configuration $c^*$ we have $\text{parent}_w \in V$ for every node $w \in V$. This proves that there exists a configuration $c^*$, such that $c \xrightarrow{\tau} c_1 \xrightarrow{\tau} c^*$, where clauses (a) and (b) hold.

It remains to show that configuration $c^*$ is such that action reset$_v$ is not invoked for any node $v \in V$ after $c^*$. The latter can be shown by following the proof of clause (c) of Lemma 3.3.2. This completes the proof.
Next we show that after a constant number of rounds following the cessation of failures no message is sent to bogus destinations.

Lemma 3.4.3 Consider an execution prefix of RD-BS that ends with configuration $c$. Any fair extension of $\alpha$ of a sufficient length that uses only the actions from $A$ reaches a configuration $c^*$ in at most two complete iterations, where no messages are sent to bogus destinations by any node $v \in V$ following $c^*$.

Proof. Let $\alpha$ be the execution prefix. Let execution $\alpha_1$ be an extension of $\alpha$, reaching configuration $c^*$ exactly as in Lemma 3.4.2.

The proof follows from the algorithm and from Lemmas 3.4.1 and 3.4.2. This is because according to the algorithm messages are only sent to the nodes in $Dest$. Note that after $c^*$ for every node $i \in V$ we have $\text{parent}_i \in V$. On the other hand, sets $\text{Nbrs}_i$, $\text{Children}_i$, and $\text{world}_i$ are reset in every iteration and any node $v$ added to these sets after $c^*$ is an active node that belongs to $V$. \hfill $\square$

We now show that algorithm RD-BS satisfies the Closure and Convergence properties of Definition 3.1.2. The proof of Theorem 3.4.4 is by induction on the length of the execution extension.

Theorem 3.4.4 (Closure) Consider any execution prefix of RD-BS consisting of complete iterations, where $c$ is the final configuration. If $c$ is a legitimate configuration, then any extension of the execution by up to one complete iteration using only the actions from $A$ results in $c \xrightarrow{\tau} c'$, where $c'$ is a legitimate configuration.

Proof. If $c$ is a legitimate configuration then the following holds for every node $v \in V$: $\text{world}_v = V$, $v_0 = \text{parent}_v = \min\{u : u \in V\}$, and $\text{Children}_{v_0} = V$, while for every other
node \( w \neq v_0 \) we have \( \text{Children}_w = \emptyset \). Note that trajectory changes only the value of \( \text{clock} \), and \( \text{clock} \) does not appear in \( \psi() \).

We proceed by induction on the length \( k \) of the execution extension. For the base case, \( k = 1 \), let \( c_1 = c \) and let us consider any transition with \( c_1 \xrightarrow{\tau} c_2 \). Since \( \psi(c_1) \) holds, then for every node \( v \in V \) we have \( \text{parent}_v \in V \) in configuration \( c_1 \). Hence, according to Lemma 3.4.2, action \( \text{reset} \) cannot occur.

Actions \( \text{mrecv} \) and end-round cannot occur, since no messages have been sent, and the end of the round is not reached respectively. The only enabled discrete actions that can occur at the beginning of the round are \( \text{msend} \) and \( \text{restart} \). These actions do not affect variables in \( \psi \), and thus \( \psi(c_2) \) holds. For the inductive step we assume that all extensions of length \( k \) preserve the invariant, and we consider the discrete action \( a_{k+1} \); we show that the corresponding transition with \( c_{k+1} \xrightarrow{\tau} c_{k+2} \) reaches a legitimate configuration \( c_{k+2} \).

We now consider each discrete action \( a \) in \( \tau \) at a node \( v \in V \) when \( \text{phase}_v = \text{gossip} \).

\( a_{k+1} = \text{restart}_v \): Since \( \psi(c_{k+1}) \) holds, \( \text{active}_v = \text{true} \), and thus \( c_{k+2} = c_{k+1} \) and the invariant holds.

\( a_{k+1} = \text{msend}(\langle N, p, W \rangle, I)_v \): This action is enabled at the very beginning of a round and it does not change any variables occurring in \( \psi \), thus the invariant is maintained.

We also note that since \( c_{k+1} \) is a legitimate configuration, then, by fairness, every node \( v \in V \) invokes action \( \text{msend}(\langle N, p, W \rangle, I)_v \) at the beginning of the round. Since \( \text{phase}_v = \text{gossip} \), set \( I \) consists of all nodes in \( V \) (\( \text{world}_v \subseteq \text{Dest}_v \)) that also includes its parent, neighbors in \( G \), and children, where \( p = v_0 \).

\( a_{k+1} = \text{mrecv}(\langle N, p, W \rangle, s)_v \): This is an input action, and it is always enabled. Given that \( \text{msend} \) occurs as described above, \( v \) receives the corresponding message from all nodes \( s \in V \),
including a child, a neighbor, and a parent. The transition may change \( R_v \), but this does not affect \( \psi \). Next, if \( p = v \), that is, node \( s \) considers node \( v \) to be its parent, then \( \psi(c_{k+1}) \) implies that \( v = v_0 \), and thus \( s \) is included in \( New\_Chld_v \). If \( p \neq v \), then \( New\_Chld_v = \emptyset \) since \( phase_v = \text{gossip} \). Given that action \text{mrecv} does not affect variables in \( \psi \) when \( phase_v = \text{gossip} \) the invariant \( \psi(c_{k+2}) \) holds.

We also note that all messages sent at the beginning of the current round are delivered before the end of the round. Since \( c_{k+1} \) is a legitimate configuration, then \( \psi(c_{k+1}) \) implies that \( Children_{v_0} = V \), and hence every node \( v \in V \) receives a message from \( v_0 \) and adds \( v_0 \) to \( R_v \). Additionally, since \( parent_v = v_0 \) for every node \( v \in V \), it follows that \( R_{v_0} = V \). Thus \( v \) receives messages from all its children (if any) by the model assumption and since no perturbations occur. Hence, by the end of the round \( New\_Chld_{v_0} = V \), while for every other node \( u \neq v_0 \) we have \( New\_Chld_u = \emptyset \). In addition, since for every node \( v \in V \) we have \( world_v = V \), and since \( world_v \subseteq Dest_v \), it follows that in configuration \( c_{k+2} \) we have \( R_v = V \) for every node \( v \in V \).

\( a_{k+1} = \text{end-round}_v \): Since, \( phase_v = \text{gossip} \) for every node \( v \in V \), it follows that \text{else} clause of end-round is executed. In configuration \( c_{k+1} \) for every node \( v \neq v_0 \) we have \( Children_v = New\_Chld_v = \emptyset \), while for the node \( v_0 \) we have \( Children_{v_0} = New\_Chld_{v_0} = V \). Additionally, for every node \( v \in V \) we have \( world_v = V \), since the \text{end-round}_v \) action is enabled at the end of the gossip phase, and hence, \( R_v = V \) and \( Nbrs_v \subseteq V \) for every node \( v \in V \). Therefore, \( \psi(c_{k+2}) \) holds.

We now consider each discrete action \( a \) in \( \tau \) at a node \( v \in V \) when \( phase_v = \text{confirm} \).

\( a_{k+1} = \text{restart}_v \): Since \( \psi(c_{k+1}) \) holds, \( active_v = true \), and thus \( c_{k+2} = c_{k+1} \) and the invariant holds.
\( a_{k+1} = \text{msend}(\langle N, p, W \rangle, I)_v \): This action is enabled at the very beginning of a round and it does not change any variables occurring in \( \psi \), thus the invariant is maintained.

We also note that since \( \psi(c_{k+1}) \) holds, then, by fairness, every node \( v \in V \) at the beginning of the round invokes action \( \text{msend}(\langle N, p, W \rangle, I)_v \). Since \( \text{phase}_v = \text{confirm} \), set \( I \) consists of all nodes that sent a message to \( v \) in the previous round and neighbors in \( G^u \) (see Lemma 3.4.1).

\( a_{k+1} = \text{mrecv}(\langle N, p, W \rangle, s)_v \): This is an input action, and it is always enabled. Given that \( \text{msend} \) occurs as described above, \( v \) receives the corresponding message from all nodes \( s \in V \) (\( R_v = V \)). The transition may change \( R_v \), but this does not affect \( \psi \), since node \( u \) is included in \( R_v \) only if \( v \) receives a message from \( u \), and hence \( u \in V \) and \( R_v \subseteq V \). On the other hand, since the execution prefix of \( \text{RD-BS} \) consists of complete iterations, it is clear that if a node \( v \) invokes \( \text{mrecv}_v \), where \( \text{phase}_v = \text{confirm} \), then prior to that every node \( v \in V \) already received all messages sent to it when \( \text{phase}_v = \text{gossip} \). Hence, as we already argued, \( R_v = V \), resulting in \( v_0 \in R_v \), for every node \( v \in V \). Next, since \( \text{world}_v = V \) for every node \( v \in V \) it follows that \( \text{world} \) is not modified for any node in \( V \).

We also note that all messages sent at the beginning of the current round are delivered by the end of the round. Since action \( \text{mrecv} \) does not affect \( \text{Children}_v \), and since \( \text{world}_v = V \), the invariant \( \psi(c_{k+2}) \) holds.

\( a_{k+1} = \text{end-round}_v \): Since, \( \text{phase}_v = \text{confirm} \) for every node \( v \in V \), it follows that the if clause of end-round action is executed. Since \( \psi(c_{k+1}) \) holds and since for every node \( v \in V \) we have \( v_0 \in R_v = V \), it follows that \( \text{parent}_v = \min\{u : u \in R \cup v\} = v_0 \), and hence, configuration \( c_{k+2} = c' \) is reached where \( \psi(c') \) holds. This concludes the proof. \( \square \)

We next address the convergence property, starting with preparatory lemmas.
Lemma 3.4.5 Consider an execution prefix of RD-BS that ends with configuration $c$. Any fair extension of the execution of a sufficient length using only the actions from $A$ reaches a configuration $c^*$ in at most two complete iterations, where following configuration $c^*$ for any two distinct nodes $u, v \in V$ the following holds: if $\text{phase}_u = \text{phase}_v = \text{gossip}$ and $u \in \text{world}_v$ then $v \in R_u$ and $v \in \text{world}_u$ following the invocation of end-round$_u$ action.

Proof. Let $\alpha$ be the execution prefix. Let execution $\alpha_1$ be an extension of $\alpha$, reaching configuration $c^*$ exactly as in Lemma 3.4.2.

Note that in configuration $c^*$ for every node $v \in V$ we have $\text{phase}_v = \text{gossip}$. Let us assume that there exists a node $u \in V$, such that $u \neq v$ and $u \in \text{world}_v$. By fairness, after configuration $c^*$, actions $\text{msend}_v$ and $\text{mrecv}_v$ are invoked for every node $v \in V$. Let $c_1$ be the resulting configuration and let $\alpha_2$ be such an extended execution.

Since in configuration $c^*$ we have $u \in \text{world}_v$, and since $\text{world}_v \subseteq \text{Dest}_v$, it is clear that $u \in \text{Dest}_v$. Hence, in configuration $c_1$ we have $v \in R_u$. By fairness, after configuration $c_1$ action end-round$_v$ is invoked for any node $v \in V$. Let $c_2$ be the resulting configuration and let $\alpha_3$ be such an extended execution. Since in configuration $c_1$ for every node $v \in V$ we have $\text{phase}_v = \text{gossip}$ it is clear that the else clause of the end-round is executed, and hence $v \in \text{world}_u$. This completes the proof. \hfill \Box

Lemma 3.4.6 Consider an execution prefix $\alpha$ of RD-BS that ends with configuration $c$. Let execution $\alpha_1$ be an extension of $\alpha$, reaching configuration $c_1$ exactly as in Lemma 3.4.2. Let execution $\alpha_2$ be an extension of $\alpha_1$ by 0 or more complete iterations reaching configuration $c_2$. Furthermore, let $\alpha_3$ be an extension of $\alpha_2$ by exactly one iteration reaching configuration $c_3$. Then for every node $v \in V$ we have $c_2.\text{world}_v \subseteq c_3.\text{world}_v$. 
Proof. Consider any node $u \in c_2.world_v$, we want to show that $u \in c_3.world_v$. In configuration $c_2$ we have $phase_v = gossip$ for every node $v \in V$. Let us extend $\alpha_2$ by one complete round reaching configuration $c'_2$, following the invocation of action end-round$_v$ for every node $v \in V$. Let $\alpha'_2$ be such an extended execution. From Lemma 3.4.5 it is clear that in configuration $c'_2$ we have $v \in R_u$ and $v \in world_u$. Furthermore, since the else clause of the end-round action is executed, it is also clear that $v \in Dest_u$.

By fairness, following configuration $c'_2$ actions msend$_v$ and mrecv$_v$ are invoked by every node $v \in V$. Let $c^*$ be the resulting configuration and let $\alpha^*$ be such an extended execution. From the specification of actions msend$_v$ and mrecv$_v$, it is clear that in configuration $c^*$ node $u$ is included both in $R_u$ and $world_u$. The latter is true because in configuration $c'_2$ we have $v \in Dest_u$. Next, configuration $c_3$ is reached after the invocation of action end-round$_v$ for every node $v \in V$. From above, and from the fact that the if clause of action end-round is executed, is clear that for every node $u \in c_2.world_v$ we have $u \in c_3.world_v$ in configuration $c_3$. This completes the proof. □

Lemma 3.4.7 Consider an execution prefix of RD-BS that ends with configuration $c$. Any fair extension of the execution of a sufficient length using only the actions from $A$ reaches a configuration $c^*$ in at most $\log D + 3$ complete iterations, such that in configuration $c^*$ for every node $u \in V$, we have $v_0 \in world_u$.

Proof. Let $\alpha$ be the execution prefix. Let execution $\alpha_1$ be an extension of $\alpha$, reaching configuration $c_1$ exactly as in Lemma 3.4.2.

Based on the specification of action end-round and according to Lemmas 3.4.1 and 3.4.2 it is clear that in configuration $c_1$ for every node $v \in V$ we have $Nbrs_v \subseteq world_v$, where $Nbrs_v$
is the set of neighbors of \( v \), including \( v \) itself, in the initial knowledge graph \( G^u = (V, E^u) \).

Hence, \( E^u \subseteq c_1 \cdot \mathcal{E}^u \), where \( c_1 \cdot \mathcal{E}^u \) is the set of edges of the evolving knowledge graph \( G^u \) in configuration \( c_1 \).

Let us consider any path \( u = u_0, u_1, \ldots, u_{k-1}, u_k = v_0 \) in graph \( c_1 \cdot G^u \), where \( 0 \leq k \leq D \).

Consider any three consecutive nodes \( u_{j-1}, u_j \) and \( u_{j+1} \), for \( 0 < j < k \). From configuration \( c_1 \) it follows that \( (u_j, u_{j-1}) \) and \( (u_j, u_{j+1}) \) are in \( c_1 \cdot \mathcal{E}^u \). Let us extend \( \alpha_1 \) by one complete round reaching configuration \( c_2 \), after the invocation of action end-round, for every node \( v \in V \). Let \( \alpha_2 \) be such an extended execution. From Lemma 3.4.5 it follows that for any node \( u_j \), such that \( 0 < j < k \), we have \( u_{j-1}, u_{j+1} \in c_2 \cdot \text{world}_{u_j} \), this is because in configuration \( c_1 \) we have \( u_j \in \text{world}_{u_{j-1}} \) and \( u_j \in \text{world}_{u_{j+1}} \) for all \( 0 < j < k \). Note also, that from the specification of action end-round it follows that \( \text{world}_v \subseteq \text{Dest}_v \) for every node \( v \in V \) in configuration \( c_2 \).

Let us further extend \( \alpha_2 \) by a complete round reaching configuration \( c_3 \). Let \( \alpha_3 \) be the extended execution. From the specification of actions msend and mrecv it follows that \( u_{j-2}, u_{j+2} \in \text{world}_{u_j} \). This is because, as we argued above, in configuration \( c_2 \) we have \( u_{j-2}, u_j \in \text{world}_{u_{j-1}} \) and \( u_j, u_{j+2} \in \text{world}_{u_{j+1}} \). Hence, in configuration \( c_3 \) there exists a path \( u = u_0, u_2, \ldots, u_{2i}, \ldots, u_k = v_0 \) for \( k \) even, or \( u = u_0, u_2, \ldots, u_{2i}, \ldots, u_{k-1}, u_k = v_0 \) for \( k \) odd between nodes \( u \) and \( v_0 \) in the undirected knowledge graph \( c_3 \cdot G^u \). Observe that the length of above path between the nodes \( u \) and \( v_0 \) is \( \lceil \frac{k}{2} \rceil \).

Hence, based on Lemma 3.4.6 and from above, after at most \( \log D \) subsequent finite extensions of \( \alpha_3 \) by a complete iteration configuration \( \alpha^* \) is reached in which for every node \( u \in V \) we have \( v_0 \in \text{world}_u \). This completes the proof.
Theorem 3.4.8 (Convergence) Consider an execution prefix of RD-BS that ends with configuration $c$. Any fair extension of the execution of a sufficient length using only the actions from $A$ reaches a configuration $c_l$ in at most $\log D + 5$ complete iterations, such that $c_l$ is a legitimate configuration.

Proof. Let $\alpha$ be the execution prefix. Let execution $\alpha_1$ be an extension of $\alpha$, reaching configuration $c_1$ exactly as in Lemma 3.4.7.

Following $c_1$ action $\text{msend}_v$ is enabled for every node $v \in V$. Let us extend $\alpha_1$ so that configuration $c_2$ is reached by invoking $\text{msend}(\langle N, p, W \rangle, u)_v$ for every $v \in V$, followed by an invocation of $\text{mrecv}(\langle N, p, W \rangle, v)_u$ for every node $u$. Let $\alpha_2$ be such an extended execution. Since in configuration $c_1$ for every node $v \in V$ we have $v_0 \in \text{world}_v$ and $\text{world}_v \subseteq \text{Dest}_v$, it follows that in configuration $c_2$ we have $R_{v_0} = V$. Let us further extend $\alpha_2$ so that configuration $c_3$ is reached by invoking action $\text{end-round}_v$ for all $v \in V$. Let $\alpha_3$ be such an extended execution. Based on the specification of action $\text{end-round}$, and from the fact that the else clause of the end-round is executed, it is clear that in configuration $c_3$ we have $R_{v_0} = V \subseteq \text{Dest}_{v_0}$, and subsequently, we have $\text{world}_{v_0} = V$. Following configuration $c_3$ action $\text{msend}_v$ is enabled for every node $v \in V$. Let us extend $\alpha_3$ so that configuration $c_4$ is reached by invoking $\text{msend}(\langle N, p, W \rangle, u)_v$ for every $v \in V$, followed by an invocation of $\text{mrecv}(\langle N, p, W \rangle, v)_u$ for every node $u$. Let $\alpha_4$ be such an extended execution. From the specification of action $\text{mrecv}$ it is clear that $v_0 \in R_v$ for every node $v \in V$, and since in configuration $c_3$ we have $\text{world}_{v_0} = V$, then from the specification of $\text{mrecv}$ it follows that $\text{world}_v = V$ for every node $v \in V$. Let us further extend $\alpha_4$ so that configuration $c_5$ is reached by invoking action $\text{end-round}_v$ for all $v \in V$. Let $\alpha_5$ be such an extended execution. Since configuration $c_5$ is
reached by executing the if clause of action end-round it is clear that in configuration $c_5$ for every node $v \in V$ we have $parent_v = v_0$ and $world_v = V$, hence clauses (1) and (3) of invariant $\psi$ hold in configuration $c_5$.

It remains to show that execution $\alpha_5$ can be fairly extended by a complete iteration such that a configuration $c_1$ is reached where clause (2) of invariant $\psi$ also holds. Following $c_5$ action $msend_v$ is enabled for every node $v \in V$. Let us extend $\alpha_5$ so that configuration $c_6$ is reached by invoking $msend(\langle N, p, W \rangle, v)$ for every node $v \in V$, followed by an invocation of $mrecv(\langle N, p, W \rangle, v)$ for every node $u$. Let $\alpha_6$ be such an extended execution.

We want to examine the state variable $New.Chld_v$ for every node $v \in V$ in configuration $c_6$. From the specification of end-round action it follows that in configuration $c_5$ for every $v \in V$ we have $R_v = \emptyset$, $New.Chld_v = \emptyset$, $phase_v = \text{gossip}$, and $v_0 \in Dest_v$. From the specification of action $mrecv$ and from the fact that in configuration $c_5$ for every node $v \in V$ we have $parent_v = v_0$, it is clear that in configuration $c_6$ we have $New.Chld_{v_0} = V$, while for every other node $u \neq v_0$ we have $New.Chld_u = \emptyset$. Note also, that if a node $u$ is added to $R_v$, then $v$ received a message from $u$, and hence $R_v \subseteq V$. Additionally, since in configuration $c_5$ we have $world_v = V$ for every node $v \in V$, it follows that in configuration $c_6$ for every node $v \in V$ we have $R_v = V$, this is because $world_v \subseteq Dest_v$ in configuration $c_5$.

Let us further extend $\alpha_6$ so that configuration $c_7$ is reached by invoking action end-round$_v$ for all $v \in V$. Let $\alpha_7$ be such an extended execution. Since in configuration $c_7$ for every node $v \in V$ we have $phase_v = \text{gossip}$, coupled with the specification of action end-round, it follows that the else clause of action end-round is executed. Hence, in configuration $c_7$ we have $Children_{v_0} = V$, while for every node $u \neq v_0$ we have $Children_u = \emptyset$. Therefore
clause (2) of invariant $\psi$ holds in $c_7$. Note also, that in configuration $c_7$ we have $Dest_v = V$ and $world_v = V$ for every node $v \in V$, this is because $R_v = V$ in configuration $c_6$.

Next, we extend the execution $\alpha_7$ so that configuration $c_8$ is reached by invoking action $\text{msend}(\langle N, p, W \rangle, u)_v$ for every node $v \in V$, followed by an invocation of action $\text{mrecv}(\langle N, p, W \rangle, v)_u$ for every node $u$. Let $\alpha_8$ be such an extended execution. Since in configuration $c_8$ for every node $v \in V$ we have $phase_v = \text{confirm}$, it follows that the invocation of $\text{mrecv}$ only affects variables $R_v$ and $world_v$, but since $R_v = world_v = V$ in configuration $c_7$, then from the specification of action $\text{mrecv}$ it follows that in configuration $c_8$ for every node $v \in V$ we have $R_v = world_v = V$.

We further extend $\alpha_8$ such that configuration $c_l$ is reached by invoking action $\text{end-round}_v$ for all $v \in V$. Since in configuration $c_8$ for every node $v \in V$ we have $phase_v = \text{confirm}$, it follows that the if clause of action $\text{end-round}$ is executed. Note that for every $v \in V$, the only variable that appears in $\psi$ and is modified in the if clause is $parent_v$. However, as we showed, in configuration $c_7$ for every node $v \in V$ we have $v_0 \in R_v$ and $R_v = V$, and hence $parent_v = v_0$. Therefore, $\psi(c_l)$ holds, hence, $c_l$ is a legitimate configuration.

Finally we reason about the stabilization time and message complexity.

**Theorem 3.4.9** Any execution prefix of $RD$-$BS$ ending in an arbitrary configuration can be infinitely extended to solve the resource discovery problem. The stabilization time of the algorithm is $O(\log D)$, taking at most $2 \log D + 10$ complete rounds to stabilize. The stabilization message complexity is $O(|V|^2 \cdot \log D)$.

**Proof.** From the proofs of Lemma 3.4.2, Lemma 3.4.7, and Theorem 3.4.8 it follows that after the cessation of transient failures, and given that no new nodes join the computation, any fair
execution extension of a sufficient length takes at most \( \log D + 5 \) complete iterations to reach a legitimate configuration from any configuration \( c \), and hence \( 2 \log D + 10 \) complete rounds. Theorem 3.4.4 (closure) guarantees that all subsequent configurations are legitimate. Thus the algorithm establishes the resource discovery invariant (Definition 3.1.5), taking at most \( 2 \log D + 10 \) complete rounds to stabilize, and then maintains the resource discovery invariant in perpetuity.

We now assess the stabilization message complexity. Consider some execution that includes an arbitrary configuration after which no actions from \( \tilde{A} - A \) occur. The proof of Lemma 3.4.3 reasons that in a constant number of rounds no node identifiers from \( U - V \) occur in any local state. Prior to this the algorithm may send messages to arbitrary sets of nodes due to corruptions of local states. We charge such messages to the environment (per model assumptions) and do not include them in the message complexity. Here we only consider messages sent to the nodes in \( V \). From the specification of action end-round it follows that in both gossip and confirm phases the number of accountable messages sent by each node \( v \in V \) is at most \( |V| \). Therefore, since the stabilization time is \( O(\log D) \), the message complexity of \( RD-BS \) is \( O(|V|^2 \cdot \log D) \).

3.5 Closing Remarks

Our analyses of algorithms \( RD-MS \) and \( RD-BS \) showed that by using a more aggressive information propagation we were able to decrease the convergence time complexity from \( O(D) \), for algorithm \( RD-MS \), to \( O(\log D) \), for algorithm \( RD-BS \). Notice, that the diameter of an initial connectivity graph can be as large as \( |V| \), resulting in \( O(|V|^2) \) message complexity for algorithm \( RD-MS \). In the latter case the message complexity of algorithm \( RD-MS \) is
better than the one of RD-BS by only a logarithmic factor. Moreover, if the initial connectivity graph is such that the out degree of a node is greater than one, e.g., a node knows the identifiers of more than one node, then the message complexity of algorithm RD-MS is $O(\Delta \cdot D)$, where $\Delta$ is the sum of the out degrees of all nodes in the initial connectivity graph. In the same setting the message complexity of algorithm RD-BS is not affected. It remains an open question whether we can develop an algorithm that works under our failure model and converges in logarithmic time, while keeping the message complexity low. In particular we would like our algorithm to achieve $O(n \log n)$ message complexity, similar to the algorithm presented by Kutten et al. [61].
Chapter 4

Network Supercomputing

Internet supercomputing is an approach to solving partitionable, computation-intensive problems by harnessing the power of a vast number of interconnected computers. Traditional centralized approaches to Internet supercomputing, e.g., [59, 36], employ a master processor and many worker processors that execute a collection of tasks on behalf of the master. In this chapter we investigate the means of departing from the traditional approach and develop decentralized algorithms, where workers cooperate in performing tasks. We consider a variety of failure models with increasing adversarial strength, and develop randomized algorithms for synchronous message passing systems. We assess time, work, and message complexities for each respective failure model. Additionally, we develop a randomized algorithm where depending on the number of crashes each live processor is able to terminate with the knowledge that the problem is solved with high probability. Lastly, we develop an algorithm for estimating the probability of a processor performing an incorrect computation and we analyze the algorithm.
**Chapter Structure.** In Section 4.1 we present the model of computation and definitions. In Section 4.2 we present Algorithm NS, where a collection of worker processors cooperates on a large set of independent tasks without the reliance on a centralized control. Algorithm NS is able to perform all tasks correctly with high probability (whp), while dealing with unreliable processors under an assumption that the average probability of live (non-crashed) processors returning incorrect results remains inferior to $1/2$ during the computation. Here the adversary is only allowed to crash a constant fraction of processors, and the correct termination of the $n$-processor algorithm strongly depends on the availability of $\Omega(n)$ live processors.

In Section 4.3 we present Algorithm DAKS that is able to deal with much stronger adversaries, e.g., those that can crash all but a fractional polynomial in $n$, or even a poly-log in $n$, number of processors. One of the challenges here is to enable the algorithm to terminate efficiently in the presence of any allowable number of crashes. Notice, that in contrast to algorithm NS, algorithm DAKS works correctly under any allowable number of crashes, provided that the average probability of a live processor returning a bogus result is inferior to $1/2$.

Finally, in Section 4.4 we present our decentralized Algorithm $A_{est}$ that efficiently estimates the probability of a processor performing an incorrect computation using an $(\varepsilon, \delta)$-approximation for any constant $\varepsilon > 0$ and some $\delta > 0$ chosen by the user.

### 4.1 System Model and Definitions

**System model.** There are $n$ processors, each with a unique identifier (id) from set $P = [n]$. We refer to the processor with id $i$ as processor $i$. The system is synchronous and processors communicate by exchanging reliable messages. Computation is structured in terms of synchronous rounds, where in each round a processor can send and receive messages, and perform
local polynomial computation, where the local computation time is assumed to be negligible compared to message latency. The duration of each round depends on the algorithm and need not be constant (e.g., it may depend on \( n \)). Messages sent during the send step are delivered in the next receive step of the same round.

Tasks. There are \( t \) tasks to be performed, each with a unique id from set \( T = [t] \). We refer to the task with id \( j \) as \( Task[j] \). The tasks are (a) similar, meaning that any task can be done in constant time by any processor, (b) independent, meaning that each task can be performed independently of other tasks, and (c) idempotent, meaning that each task admits at-least-once execution semantics and can be performed concurrently. For simplicity, we assume that the outcome of each task is a binary value. The problem is most interesting when there are at least as many tasks as there are processors, thus we consider \( t \geq n \).

Models of Adversity. Processors are undependable in that a processor may compute the results of tasks incorrectly and it may crash. Following a crash, a processor performs no further actions. Otherwise, each processor adheres to the protocol established by the algorithm it executes. We refer to non-crashed processors as live. We consider an oblivious adversary that decides prior to the computation what processors to crash and when to crash them.

For an execution of an algorithm, let \( F \subset P \) be the subset of processors that adversary crashes. The maximum number of processors that can crash is established by the following adversarial models.

Model \( F_{lj} \), adversary constrained by a linear fraction:

\[ |P - F| \geq hn, \text{ for a constant } h \in (0, 1). \]
**Model** $\mathcal{F}_{fp}$, adversary constrained by a fractional polynomial:

$$|P - F| = \Omega(n^a), \text{ for a constant } a \in (0, 1).$$

**Model** $\mathcal{F}_{p\ell}$, poly-log constrained adversary:

$$|P - F| = \Omega(\log^c n), \text{ for a constant } c \geq 1.$$

We denote by $\mathcal{F}_0$ a failure model where $F = \emptyset$, i.e., no processor crashes. We assume that the adversary can assign an arbitrary constant probability $p_i$, unknown to the processors, of returning an incorrect result for each processor $i \in P$. We define failure models $\mathcal{E}_2$ and $\mathcal{E}_1$ that respectively constrain the adversary’s ability to crash only processors that return correct results with probabilities greater than $1/2$ and the adversary’s ability to assign probability $p_i = 1$ to a processor $i \in P$, ensuring that processor $i$ always returns incorrect results.

**Model** $\mathcal{E}_2$: \[
\frac{1}{|P - F|} \sum_{i \in P - F} p_i < \frac{1}{2} - \zeta, \text{ for a constant } \zeta > 0.
\]

That is, the average probability of non-crashed processors returning incorrect results is inferior to $1/2$. We use the constant $\zeta$ to ensure that the average probability of incorrect computation does not become arbitrarily close to $1/2$ as $n$ grows arbitrarily large. Notice, that in model $\mathcal{E}_2$, the average probability constraint holds even if $F = \emptyset$.

**Model** $\mathcal{E}_1$: $p_i < 1$, for every $i \in P$.

That is, every processor should return correct results with probability greater than zero.

Based on the models $\mathcal{F}_0$, $\mathcal{F}_{\ell f}$, $\mathcal{F}_{fp}$, and $\mathcal{F}_{p\ell}$, that bound the size of $P - F$ from below, and the models $\mathcal{E}_2$ and $\mathcal{E}_1$, that impose constraints on the probabilities, we define failure models as pairs $(\mathcal{F}, \mathcal{E})$, where $\mathcal{F} \in \{\mathcal{F}_0, \mathcal{F}_{\ell f}, \mathcal{F}_{fp}, \mathcal{F}_{p\ell}\}$ and $\mathcal{E} \in \{\mathcal{E}_2, \mathcal{E}_1\}$. For example, failure model $(\mathcal{F}_{\ell f}, \mathcal{E}_2)$ implies that the adversary is constrained by a linear fraction in its ability to...
crash processors, and moreover, at any point of the computation, the average probability of non-crashed processors to return incorrect results is inferior to $1/2$.

**Problem Statement.** With the removal of the infallible master, a client application should be able to obtain the results from any worker. In decentralized setting, the *network supercomputing* problem is for $n$ processors to collectively perform $t \geq n$ tasks. Thus, we say that the Network Supercomputing Problem (NSP) is solved when all non-crashed processors know that all $t$ tasks have been performed and are in the possession of the results of all tasks.

**Definition 4.1.1 (Network Supercomputing Problem)** Given a set $T$ of $t$ tasks, collectively perform all tasks using $n$ processors, each non-crashed processor must learn the results of all tasks, under adversary $A$.

**Measures of efficiency.** We assess the efficiency of algorithms in terms of *time* $T$, *work* $W$, and *message* $M$ complexities. We assume that it takes a unit of time for a processor to perform a unit of work, according to its local clock, and that a single task corresponds to a unit of work. By this definition the processors are charged for idling. Our definition of work complexity is based on the *available processor steps* measure [51].

Let $A$ be an algorithm that solves a problem in the presence of adversary $A$. We denote by $execs(A,A)$ the set of all executions of algorithm $A$ for adversary $A$. For an execution $\xi \in execs(A, A)$, we denote by $P_r(\xi)$ the number of live processors at the beginning of round $r$ of the execution, and by $M_r(\xi)$ the number of point-to-point messages sent during round $r$ of the execution. For execution $\xi \in execs(A, A)$, let $\tau(\xi)$ be the number of rounds by which algorithm $A$ solves a specific problem, then the work complexity of the algorithm is:

$$W = \max_{\xi \in execs(A, A)} \{ \sum_{r \leq \tau(\xi)} P_r(\xi) \}.$$
The message complexity of the algorithm, i.e., the number of point-to-point messages sent during the execution of an algorithm, is:

\[ M = \max_{\xi \in \text{exec}(A,A)} \{ \sum_{r \leq \tau(\xi)} M_r(\xi) \}. \]

We define the time complexity of the algorithm as:

\[ T = \max_{\xi \in \text{exec}(A,A)} \{ \tau(\xi) \}. \]

Lastly, we use the common definition of an event \( \mathcal{E} \) occurring with high probability (whp) to mean that \( \Pr[\mathcal{E}] = 1 - O(n^{-\alpha}) \) for some constant \( \alpha > 0 \).

**Known Results and Definitions Used in Analyses of Algorithms.** The following lemmas and definitions are used in the analyses of our algorithms throughout this chapter. We start by stating the Chernoff bound result.

**Lemma 4.1.2 (Chernoff Bounds)** Let \( X_1, X_2, \cdots, X_n \) be \( n \) independent Bernoulli random variables with \( \Pr[X_i = 1] = p_i \) and \( \Pr[X_i = 0] = 1 - p_i \), then it holds for \( X = \sum_{i=1}^{n} X_i \) and \( \mu = \mathbb{E}[X] = \sum_{i=1}^{n} p_i \) that for all \( \delta > 0 \), (i) \( \Pr[X \geq (1 + \delta)\mu] \leq e^{-\frac{\mu\delta^2}{3}} \), and (ii) \( \Pr[X \leq (1 - \delta)\mu] \leq e^{-\frac{\mu\delta^2}{2}} \).

**Definition 4.1.3 (The Coupon Collector’s Problem (CCP) [66].)** There are \( n \) types of coupons and at each trial a coupon is chosen at random. Each random coupon is equally likely to be of any of the \( n \) types, and the random choices of the coupons are mutually independent. Let \( m \) be the number of trials. The goal is to study the relationship between \( m \) and the probability of having collected at least one copy of each of \( n \) types.

In [66] it is shown that \( \mathbb{E}[X] = n \ln n + O(n) \) and that whp the number of trials for collecting all \( n \) coupon types lies in a small interval centered about its expected value.
Fraigniaud and Glakkoupis [38] study the communication complexity of rumor-spreading in the random phone-call model. They consider $n$ players communicating in parallel rounds, where in each round every player $u$ calls a randomly selected communication partner. Player $u$ is allowed to exchange information with the partner, either by pulling or pushing information.

The following lemma, proved in [38], shows that during the push stage of the algorithm every rumor $\rho$ is disseminated to at least $\frac{3}{4}n$ players whp.

**Lemma 4.1.4** [38] With probability $1 - n^{-3+o(1)}$, at least $\frac{3}{4}$ fraction of the players knows $\rho$ at the end of round $\tau = \lg n + 3 \lg \lg n$.

### Extending the Algorithms for $t \geq n$.

For simplicity of the presentation all algorithms for the network supercomputing problem presented in this chapter are stated for $t = n$. However, it is easy to modify the algorithms so that they handle arbitrary number of tasks $t$ such that $t \geq n$.

To do that we segment the $t$ tasks into chunks of $\lceil t / n \rceil$ tasks, and construct a new array of chunk-tasks with identifiers in $T = [n]$, where each chunk-task takes $\Theta(t/n)$ time to perform by any live processor. We then use the developed algorithms, where the only difference is that each computation stage takes $\Theta(t/n)$ time to perform a chunk-task. In the sequel, if $A$ is the name of the algorithm for $t = n$, then we use $A_{t,n}$ as the name of the algorithm when $t \geq n$.

### 4.2 Network Supercomputing Without Centralized Control

In this section we present a $n$-processor, $n$-task algorithm that eliminates the master and instead uses a decentralized approach, where workers cooperate in performing tasks. We show that our algorithm works correctly for failure models $\langle F_0, E_\frac{1}{2} \rangle$ and $\langle F_f, E_\frac{1}{2} \rangle$. 
The number of rounds performed by the algorithm is an external parameter whose value is established in the analysis. Within each round each processor performs a random task (for some number of rounds), and communicates its cumulative knowledge to one randomly chosen processor. The algorithm naturally generalizes for $t$ tasks, where $t \geq n$, by having processors work on groups $\lceil t/n \rceil$ tasks instead of single tasks.

We analyze our algorithm under failure models $\langle F_0, E_2 \rangle$ and $\langle F_{\ell f}, E_2 \rangle$ and show that it is sufficient for it to iterate for $\Theta(\log n)$ rounds in order to perform all tasks whp. More specifically, we prove that after $\Theta(\log n)$ rounds every live processor holds the array of computed results that are all correct whp, and that the arrays of results are consistent among all processors whp.

With $t$ tasks ($t \geq n$), the algorithm has time complexity $\Theta(t \log n)$, work complexity $\Theta(t \log n)$, and message complexity $\Theta(n \log n)$. We show that upon termination the workers know the results of all tasks whp, and that these results are correct whp.

### 4.2.1 Description of Algorithm NS

We present our decentralized algorithm NS that uses a gossip-based approach. Each processor (worker) maintains two arrays of size linear in $n$, one used to accumulate knowledge gathered from different processors, and another to store the results. The algorithm works in synchronous rounds. The number of rounds performed by the algorithm is an external parameter whose value is established in the analysis. Within each round a processor communicates its cumulative knowledge to one randomly chosen processor and performs a random task (for some determined number of rounds). The pseudocode for the algorithm is given in Figure 8, and we now detail it.
procedure for processor $i$;
  
  external $n$ /* the number of processors and tasks */
  
  external $L$ /* $2L$ is the the number of rounds */
  
  Task[1..$n$] /* set of tasks */
  
  $R_i[1..n]$ init $\emptyset$ /* set of collected results */
  
  Results,$i[1..n]$ /* array of results */

  
  Compute:
  
  1: Randomly select $j \in T$ /* choose task id */
  2: Compute the result $v_j$ for Task$[j]$
  3: $R_i[j] \leftarrow \{ \langle v_j, i, 0 \rangle \}$ /* record result for round 0 */

  for $r = 1$ to $2L$ do

  Send:
  
  4: Randomly select a processor $q \in P$
  5: Send the array $R_i[]$ to processor $q$

  Receive:
  
  6: Let $M$ be the set of received messages
  7: for all $j \in T$
  8: $R_i[j] \leftarrow R_i[j] \cup \{ R[j] : R[ ] \in M \}$

  Compute:

  9: if $r < L$ then
  10: Randomly select $j \in T$ /* choose task id */
  11: Compute the result $v_j$ for Task$[j]$
  12: $R_i[j] \leftarrow R_i[j] \cup \{ \langle v_j, i, r \rangle \}$

  13: for each $j \in T$
  14: Results,$i[j] \leftarrow u$ such that triples $\langle u, ., . \rangle$ form a plurality in $R_i[j]$

end

Figure 8: Algorithm NS for $t = n$; code at processor $i \in P$.

**Local knowledge and state variables.** Every processor $i$ maintains the following:

- $L$, the external parameter that is used to control the number of rounds, i.e., $2L$, of the main loop; $r$ is the current round (iteration) number.

- The array of results $R_i[1..n]$, where the element $R_i[j]$, for $j \in T$, is the set of results for Task$[j]$. Each $R_i[j]$ is a set of triples $\langle v_j, i, r \rangle$ representing the result $v_j$ computed for Task$[j]$ by processor $i$ in round $r$ (here the inclusion of $r$ ensures that the results computed by processor $i$ in different rounds are preserved).
The array $\text{Results}_i[1..n]$ stores the final results.

**Control flow.** The algorithm contains the main for-loop, and we use the term *round* to refer to a single iteration of the loop. The loop contains three stages (or steps), viz., *Send*, *Receive*, and *Compute*. The algorithm starts by performing a single *Compute* stage, after which it enters the main loop. The algorithm uses an external parameter $L$ (whose value is established in the analysis of the algorithm). The main loop iterates $2L$ times, where in the first $L$ iterations all three stages are executed, and in the final $L$ iterations only the *Send* and *Receive* stages are executed. (We prove that $L$ needs to be $\Theta(\log n)$ to yield our high probability guarantee.)

We now describe the stages in more detail, starting with *Compute*. In *Compute* stage in round $r$ processor $i$ randomly selects a task $j$, computes the result $v_j$, and adds the triple $(v_j, i, r)$ to the result set $R_i[j]$. This is done in the first $L$ rounds.

In each *Send* stage, a processor chooses a target processor $q$ at random from the set of processors $P$. The array of results $R[\_\_]$ is sent to processor $q$.

During the *Receive* stage processor $i$ receives messages (if any) sent to it during the *Send* stage by other processors (including itself). Upon receiving the messages the processor updates its $R_i[j]$ (for each $j \in T$) by taking a union with the triples for task $j$ received in all messages.

When the main loop terminates after $2L$ rounds, each processor goes over the result set for every task and computes the result that corresponds to the plurality of the results (in the analysis we prove that in fact a majority exists). The results of the tasks are available locally in the array $\text{Results}_i[1..n]$. 

91
4.2.2 Analysis of Algorithm **NS**

We now analyze algorithm **NS** (for Network Supercomputing) for \( t = n \), then extend the analysis to \( t \geq n \). The following lemma shows that in any execution of algorithm **NS** within \( \Theta(\log n) \) rounds every task \( \tau \) is chosen for execution \( \Theta(\log n) \) times \textit{whp}.

**Lemma 4.2.1** In any execution of algorithm **NS** under the failure model \((\mathcal{F}_0, \mathcal{E}_\tau)\), after \( \Theta(\log n) \) rounds every task is performed \( \Theta(\log n) \) times, \textit{whp}, possibly by different processors.

**Proof.** Let us assume that after \( L = k \log n \) rounds of algorithm **NS**, where \( k \) is a sufficiently large constant, there exists a task \( \tau \) that is performed less than \((1 - \delta)L\) times among all workers, for some \( \delta > 0 \). We prove that \textit{whp} such a task does not exist.

According to our assumption at the end of round \( L \) for some task \( \tau \), we have \( |\cup_{j=1}^n R_j(\tau)| < (1 - \delta)L \). Let \( X_i \) be a Bernoulli random variable such that \( X_i = 1 \) if the task was chosen to be performed in line 10 (only once the task is chosen in line 1) of algorithm **NS** on page 90, and \( X_i = 0 \) otherwise.

Let us next define the random variable \( X = X_1 + \cdots + X_{Ln} \) to count the total number of times task \( \tau \) is performed by the end of \( L \) rounds of algorithm **NS**.

Note that according to line 10 any worker picks a task uniformly at random. To be more specific let \( x \) be an index of one of the \( Ln \) executions of line 10. Observe that for any \( x \), \( \Pr[X_x = 1] = \frac{1}{n} \) given that the workers choose task \( \tau \) uniformly at random. Let \( \mu = \mathbb{E}[X] = \sum_{x=1}^{Ln} \frac{1}{n} = L \), then by applying Chernoff bound, for the same \( \delta > 0 \) chosen as above, we have:

\[
\Pr[X \leq (1 - \delta)L] \leq e^{-\frac{L\delta^2}{2}} \leq e^{-\frac{(k \log n)\delta^2}{2}} \leq \frac{1}{n^{c_1^2}} \leq \frac{1}{n^\alpha}
\]

where \( \alpha > 1 \) for some sufficiently large \( c \). Now let us denote by \( \mathcal{E_\tau} \) the fact that \(|\cup_{i=1}^n R_i(\tau)| > (1 - \delta)L \) by round \( L \) of the algorithm and we denote by \( \mathcal{E}_\tau^c \) the complement of that event. Next
by Boole’s inequality we have \( \Pr[\cup_\tau \mathcal{E}_\tau] \leq \sum_\tau \Pr[\mathcal{E}_\tau] \leq \frac{1}{n^\beta} \), where \( \beta = \alpha - 1 > 0 \). Hence each task is performed at least \( \Theta(\log n) \) times whp, i.e., \( \Pr[\cap_\tau \mathcal{E}_\tau] = \Pr[\cup_\tau \mathcal{E}_\tau] \geq 1 - \frac{1}{n^\beta} \). □

The following lemma shows that whp after \( \Theta(\log n) \) rounds of the algorithm every worker obtains every triple generated in the system by either generating it locally or by means of gossiping.

A somewhat similar result is shown by Fraigniaud and Glakkoupis [38]. In order to avoid repetition, we anchor part of our proof to their results (Lemma 4.1.4) related to the push part of their algorithm. Lemma 4.1.4 (on page 88) shows that at the end of round \( \tau = \lg n + 3 \lg \lg n \) every triple \( \mathcal{t} = (v_j, i, r) \) (in their work a rumor \( \rho \)) is disseminated to at least \( \frac{3}{4}n \) workers (in their work players) whp.

**Lemma 4.2.2** In any execution of algorithm \( \text{NS} \) under the failure model \( \langle F_\emptyset, \mathcal{E}_\frac{1}{2} \rangle \), if every task is performed \( \Theta(\log n) \) times by round \( \rho \), then after additional \( \Theta(\log n) \) rounds each worker acquires the results for every task, whp.

**Proof.** Let us assume that in some round \( r \) task \( j \) is performed by worker \( i \); thus a triple \( \mathcal{t} = (v_j, i, r) \) is generated by worker \( i \), where \( v_j \) is the calculated value of task \( j \).

By applying Lemma 4.1.4 to our algorithm we infer that in \( \Theta(\log n) \) rounds of algorithm \( \text{NS} \) at least \( \frac{2}{3}n \) of the workers become aware of triple \( \mathcal{t} \) whp. Next consider any round \( d \) such that at least \( \frac{2}{3}n \) of the workers are aware of triple \( \mathcal{t} \) for the first time. Let us denote this subset of workers by \( S_d \) (\( |S_d| \geq \frac{3}{4}n \)).

We denote by \( U_d \) the remaining fraction of the workers that are not aware of \( \mathcal{t} \). We are interested in the number of rounds required for every worker in \( U_d \) to learn about \( \mathcal{t} \) whp by receiving a message from one of the workers in \( S_d \) in some round following \( d \).
We show that, by the analysis very similar to the Coupon’s Collector Problem [66] (Definition 4.1.3 on page 87), in $\Theta(\log n)$ rounds triple $\vartheta$ is known to all workers whp. Every worker in $P$ has a unique id, hence we can think of those workers as of different types of coupons and we assume that the workers in $S_d$ collectively represent the coupon collector. In this case, however, we do not require that every worker in $S_d$ contacts all workers in $U_d$ whp. Instead, we require only that the workers in $S_d$ collectively contact all workers in $U_d$ whp. According to our algorithm in every round every worker in $P$ ($S_d \subset P$), selects a worker uniformly at random and sends all its data to it. Let us denote by $m$ the collective number of trials by workers in $S_d$ to contact workers in $U_d$. According to CCP if $m = O(n \ln n)$ then whp workers in $S_d$ collectively contact every worker in $P$, including those in $U_d$. Since there are at least $\frac{3}{4}n$ workers in $S_d$ then in every round the number of trials is at least $\frac{3}{4}n$, hence in $O(\ln n)$ rounds whp all workers in $U_d$ learn about $\vartheta$. Therefore, in $\Theta(\log n)$ rounds whp all workers in $U_d$ learn about $\vartheta$.

Thus we showed that if a new triple is generated in the system then whp it will be known to all workers in $\Theta(\log n)$ rounds. Now by applying Boole’s inequality we want to show that whp in $\Theta(\log n)$ rounds all generated triples are spread among all workers.

According to our algorithm every worker generates $L = \Theta(\log n)$ triples before it terminates. We have $n$ workers which means that by the end of the algorithm the number of generated triples is $\Theta(n \log n)$. Let us denote the set of all generated triples by $V$. Let $\overline{E}_\vartheta$ be the event that some triple $\vartheta$ is not spread around among all workers when the algorithm terminates. In the preceding part of the proof we have shown that $\Pr[\overline{E}_\vartheta] < \frac{1}{n^\beta}$, where $\beta > 1$. By Boole’s inequality, the probability that there exists one triple that did not get spread to all
workers, can be bounded as

\[
\Pr[\bigcup_{\varnothing \in V} \mathcal{E}_{\varnothing}] \leq \sum_{\varnothing \in V} \Pr[\mathcal{E}_{\varnothing}] = \Theta(n \log n) \frac{n}{n^\gamma} \leq \frac{1}{n^\gamma}
\]

where \( \gamma > 0 \). This implies that upon termination every worker collects all \( \Theta(n \log n) \) triples generated in the system \( \text{whp} \).

Next theorem shows that at termination the correct result for each task is obtained from the collectively computed results, whether correct or incorrect.

**Theorem 4.2.3** Algorithm NS solves the network supercomputing problem for \( t = n \) under the failure model \( (\mathcal{F}_{\emptyset}, \mathcal{E}_{\frac{1}{2}}) \) with time complexity \( \Theta(\log n) \) rounds, \( \text{whp} \).

**Proof.** We first prove that at termination the algorithm computes correctly a majority of the results for any task \( \tau \) \( \text{whp} \). Then we argue that \( \text{whp} \) at termination the result computed for each task by any processor is correct.

In order to prove the first step we estimate (with a concentration bound) the number of times the results are computed correctly. Then we estimate the bound on total number of times task \( \tau \) was computed (whether correctly or incorrectly), and we show that a majority of the results are computed correctly.

Let us consider random variables \( X_{ir} \) that denote the success or failure of correctly computing the result of some task \( \tau \) in round \( r \) by worker \( i \). Specifically, \( X_{ir} = 1 \) if in round \( r \), worker \( i \) computes the result of task \( \tau \) correctly, otherwise \( X_{ir} = 0 \). According to our algorithm we observe that \( \Pr[X_{ir} = 1] = \frac{q_i}{n} \) and \( \Pr[X_{ir} = 0] = 1 - \Pr[X_{ir} = 1] \), where \( q_i \equiv 1 - p_i \).
Let \( X_r \equiv \sum_{i=1}^{n} X_{ir} \) denote the number of correctly computed results for task \( \tau \) among all workers during round \( r \). By linearity of expected values of a sum of random variables we have

\[
E[X_r] = E[\sum_{i=1}^{n} X_{ir}] = \sum_{i=1}^{n} E[X_{ir}] = \sum_{i=1}^{n} \frac{q_i}{n}
\]

We denote by \( X \equiv \sum_{r=1}^{L} X_r \) the number of correctly computed results for some task \( \tau \) at termination. Again, using the linearity of expected values of a sum of random variables we have

\[
E[X] = E[\sum_{i=1}^{n} \sum_{r=1}^{L} X_{ir}] = \frac{L}{n} \sum_{i=1}^{n} q_i
\]

Note that since \( \frac{1}{n} \sum_{i=1}^{n} q_i > \frac{1}{2} + \zeta \), for some fixed \( \zeta > 0 \), there exists some \( \delta > 0 \), such that, \( (1 - \delta) \frac{L}{n} \sum_{i=1}^{n} q_i > (1 + \delta) \frac{L}{2} \). Also, observe that the random variables \( X_1, X_2, \cdots, X_L \) are mutually independent. Therefore, by applying Chernoff bound on \( X_1, X_2, \cdots, X_L \) we have

\[
Pr[X \leq (1 - \delta)E[X]] \equiv Pr[X \leq (1 - \delta) \frac{L}{n} \sum_{i=1}^{n} q_i] \leq e^{-\frac{\delta^2 L (1 + \delta)}{4 (1 - \delta)}} \leq \frac{1}{n^{\alpha_1}}
\]

where \( \alpha_1 > 1 \) such that \( L = k \log n \) for some sufficiently large constant \( k > 0 \).

Let us now count the total number of times task \( \tau \) is chosen to be performed during the execution of the algorithm in the course of the first \( L \) rounds. We represent the choice of task \( \tau \) by worker \( i \) during round \( r \) by a random variable \( Y_{ir} \). We assume \( Y_{ir} = 1 \) if \( \tau \) is chosen by worker \( i \) in round \( r \), otherwise \( Y_{ir} = 0 \). Since \( Y_{ir} \)'s are mutually independent we have \( E[Y_{ir}] = \frac{1}{n} \). We denote by \( Y \equiv \sum_{i=1}^{n} \sum_{r=1}^{L} Y_{ir} \) the number of times task \( \tau \) is computed at termination. By linearity of expected values we have \( E[Y] = L \). Then by applying Chernoff bound for the same \( \delta > 0 \) chosen as above we have

\[
Pr[Y \geq (1 + \delta)E[Y]] \equiv Pr[Y \geq (1 + \delta)L] \leq e^{-\frac{(1+\delta)^2 L}{4}} \leq \frac{1}{n^{\alpha_2}}
\]
for some \( \alpha_2 > 1 \). Hence, by applying Boole’s inequality to the bounds on the above two events we have

\[
\Pr[\{X \leq (1 - \delta)\frac{L}{n} \sum_{i=1}^{n} q_i\} \cup \{Y \geq (1 + \delta)L\}] \leq \frac{2}{n^\alpha}
\]

where \( \alpha = \min\{\alpha_1, \alpha_2\} > 1 \).

Therefore, from above and by using \((1 - \delta)\frac{L}{n} \sum_{i=1}^{n} q_i > (1 + \delta)\frac{L}{2}\) we have

\[
\Pr[Y/2 < X] \geq \Pr[\{Y < (1 + \delta)L\} \cap \{X > (1 - \delta)\frac{L}{n} \sum_{i=1}^{n} q_i\}]
\]

\[
= 1 - \Pr[\{Y \geq (1 + \delta)L\} \cup \{X \leq (1 - \delta)\frac{L}{n} \sum_{i=1}^{n} q_i\}]
\]

\[
\geq 1 - \frac{1}{n^\beta}
\]

for some \( \beta > 1 \). Hence, at termination of the algorithm whp the majority of calculated results for task \( \tau \) are correct. Let us denote this event by \( \mathcal{E}_\tau \).

From above we have \( \Pr[\mathcal{E}_\tau] \leq \frac{1}{n^\beta} \), for some \( \beta > 1 \). Now, by Boole’s inequality we obtain

\[
\Pr[\bigcup_{\tau \in T} \mathcal{E}_\tau] \leq \sum_{\tau \in T} \Pr[\mathcal{E}_\tau] \leq \frac{1}{n^\beta - 1} \leq \frac{1}{n^\gamma}
\]

where \( T \) is the set of all \( n \) tasks, and \( \gamma > 0 \).

By Lemma 4.2.2 whp all calculated results of every task are disseminated across all workers. Thus, the majority of the results computed for any task at any worker is the same among all workers, and moreover it is correct whp. Recall that according to our algorithm (line 14 on page 90) every processor computes the result of every task by taking the plurality of calculated results, and hence the claim of the theorem.

Next we assess work and message complexities of the algorithm.

**Theorem 4.2.4** Algorithm NS solves the network supercomputing problem under the failure model \( \langle \mathcal{F}_0, \mathcal{E}_{\frac{1}{2}} \rangle \), whp, with work and message complexity \( \Theta(n \log n) \).
Algorithm NS terminates in $\Theta(\log n)$ rounds, and no processor crashes under failure model $\langle \mathcal{F}_0, \mathcal{E}_2 \rangle$, thus its work is $\Theta(n \log n)$. In every round every worker sends one message to a randomly chosen worker (including itself). Hence, the message complexity is $\Theta(n \log n)$.

Finally, we consider the efficiency of algorithm NS$_t$, for $t$ tasks, where $t \geq n$. The following result is directly obtained from the analysis of algorithm NS for $t = n$ by multiplying the time and work complexities by the size $\Theta(t/n)$ of the chunk of tasks; the message complexity is unchanged.

**Theorem 4.2.5** Algorithm NS$_t$ solves the network supercomputing problem under the failure model $\langle \mathcal{F}_0, \mathcal{E}_2 \rangle$, whp, with time complexity $\Theta(\frac{t}{n} \log n)$, work complexity $\Theta(t \log n)$, and message complexity $\Theta(n \log n)$.

**Proof.** As with algorithm NS, algorithm NS$_t$ takes $\Theta(\log n)$ iterations to produce the results whp, except that each iteration now takes $\Theta(t/n)$ time. This yields time complexity $\Theta(\frac{t}{n} \log n)$. Work complexity is then $n \cdot \Theta(\frac{t}{n} \log n) = \Theta(t \log n)$. The message complexity remains the same at $\Theta(n \log n)$ as the number of messages does not change.

### 4.2.3 Tolerating Crash Failures

We now show that algorithm NS correctly performs $n$ tasks whp even if up to $(1 - h)n$ processors crash for a constant $h \in (0, 1)$, under failure model $\langle \mathcal{F}_{tf}, \mathcal{E}_{\frac{1}{2}} \rangle$. We prove that the asymptotics of the algorithm are unchanged under the failure model $\langle \mathcal{F}_{tf}, \mathcal{E}_{\frac{1}{2}} \rangle$. Specifically, we show that Lemmas 4.2.1 and 4.2.2, and Theorem 4.2.3 remain valid under this model. As before, we start with $t = n$. 

98
Lemma 4.2.6 In any execution of algorithm NS under the failure mode \( (F_{\ell f}, E_{\frac{1}{2}}) \), after \( \Theta(\log n) \) rounds every task is performed \( \Theta(\log n) \) times, whp, possibly by different processors.

Proof sketch. In the worst case all failure prone processors crash in the first round of the algorithm execution. Thus, it is sufficient to prove that whp every task is performed \( \Theta(\log n) \) times among the processors in \( P - F \). In order for every task to be performed \( \Theta(\log n) \) times whp by processors in \( P - F \) it is sufficient to increase the value of \( L \) by a factor \( \lambda = \frac{1}{k} \) (compared to the case without crashes). Since all processors pick a new task to be performed from the set of \( n \) tasks uniformly at random (line 10 of algorithm NS on page 90) we can prove the results by carrying out the computation using Chernoff bound as in the proof of Lemma 4.2.1.

Now we prove that whp after \( \Theta(\log n) \) rounds of an execution of algorithm NS every worker in \( P - F \) holds the same set of triples for every task.

Lemma 4.2.7 In any execution of algorithm NS under the failure mode \( (F_{\ell f}, E_{\frac{1}{2}}) \),if processors in \( P - F \) collectively hold \( \Theta(\log n) \) calculated results for every task by round \( \rho \), then after additional \( \Theta(\log n) \) rounds each processor \( i \in P - F \) acquires all \( \Theta(\log n) \) triples for every task \( j \), whp.

Proof sketch. Consider a triple \( \vartheta \) that is generated (or obtained by gossiping) by some processor in \( P - F \). The proof of Lemma 4.2.2 uses the results from Lemma 4.1.4 and CCP [66]. Both of these results rely on the fact that there are \( \Theta(n) \) participating processors, and since
there are at least \( hn \) live processors we have \( \Theta(n) \) processors left in \( P - F \). Therefore, following a similar line of analysis we can claim the lemma with respect to the processors that do not crash until the end of algorithm NS and the triples possessed by them.

The final theorem shows that whp the correct results for each task are computed in \( \Theta(\log n) \) rounds by the processors in \( P - F \).

**Theorem 4.2.8** Algorithm NS solves the network supercomputing problem for \( t = n \) under the failure model \( \langle F_{\frac{\ell}{2}}, E_1 \rangle \) in \( \Theta(\log n) \) rounds, whp.

**Proof.** According to algorithm NS (line 14 in Figure 8 on page 90) every live processor computes the result of every task \( \tau \) by taking a plurality among all the results. Remember that in model \( \langle F_{\frac{\ell}{2}}, E_1 \rangle \) we have \( L = \lambda k \log n \), where \( \lambda = \frac{1}{n} \). We want to prove that the majority of the results for any task \( \tau \) are correct at any live processor, whp.

To do that, for a task \( \tau \) we estimate (with a concentration bound) the number of times the results are computed correctly, then we estimate the bound on the total number of times task \( \tau \) is computed (whether correctly or incorrectly), and we show that a majority of the results are computed correctly.

Let us consider random variables \( X_{ir} \) that denote the success or failure of correctly computing the result of some task \( \tau \) in round \( r \) by worker \( i \). Specifically, \( X_{ir} = 1 \) if in round \( r \), worker \( i \) computes the result of the task \( \tau \) correctly, otherwise \( X_{ir} = 0 \). According to our algorithm we observe that for a live processor \( i \) we have \( \Pr[X_{ir} = 1] = \frac{q_i}{n} \) and \( \Pr[X_{ir} = 0] = 1 - \Pr[X_{ir} = 1] \), where \( q_i \equiv 1 - p_i \). We want to count the number of correct results calculated for task \( \tau \) when a processor \( i \in P - F \) terminates.
Let $F_r$ be the set of processors that crashes prior to round $r$. We denote by $X_r \equiv \sum_{i \in P - F_r} X_{ir}$ the number of correctly computed results for task $\tau$ among all live workers during round $r$. By linearity of expected values of a sum of random variables we have

$$
E[X_r] = E\left[\sum_{i \in P - F_r} X_{ir}\right] = \sum_{i \in P - F_r} E[X_{ir}] = \sum_{i \in P - F_r} \frac{q_i}{n}
$$

We further denote by $X \equiv \sum_{r=1}^L X_r$ the number of correctly computed results for task $\tau$ at termination. Again, using the linearity of expected values of a sum of random variables we have

$$
E[X] = E\left[\sum_{r=1}^L X_r\right] = \sum_{r=1}^L E[X_r] = \sum_{r=1}^L \sum_{i \in P - F_r} \frac{q_i}{n} = \frac{L}{n} \sum_{i \in P - F_r} q_i
$$

Note that, according to our adversarial model definition, for every round $r \leq L$ we have

$$
\frac{1}{|P - F_r|} \sum_{i \in P - F_r} q_i > \frac{1}{2} + \zeta', \text{ for some fixed } \zeta' > 0. \text{ Note also that } \frac{1}{|P - F_r|} \sum_{i \in P - F_r} q_i \geq \frac{1}{n} \sum_{i \in P - F_r} q_i, \text{ and hence, there exists some } \delta > 0, \text{ such that, } (1 - \delta) \frac{L}{n} \sum_{i \in P - F_r} q_i > (1 + \delta) \frac{L}{2}.
$$

Also, observe that the random variables $X_1, X_2, \ldots, X_L$ are mutually independent, since we consider an oblivious adversary and the random variables correspond to different rounds of execution of the algorithm. Therefore, by applying Chernoff bound on $X_1, X_2, \ldots, X_L$ we have:

$$
\Pr[X \leq (1 - \delta)E[X]] = \Pr[X \leq (1 - \delta)\frac{L}{n} \sum_{i \in P - F_r} q_i] \leq e^{-\delta^2 L \frac{1}{2(1 - \delta)}} \leq \frac{1}{n^{\alpha_1}},
$$

where $\alpha_1 > 1$ such that $L = \lambda k \log n$ for some sufficiently large constant $k$ and $\lambda = \frac{1}{n}$.

Let us now count the total number of times task $\tau$ is chosen to be performed during the execution of the algorithm in the course of the first $L$ rounds. We represent the choice of task $\tau$ by worker $i$ during round $r$ by a random variable $Y_{ir}$. We assume $Y_{ir} = 1$ if $\tau$ is chosen by worker $i$ in round $r$, otherwise $Y_{ir} = 0$. Let us denote by $Y \equiv \sum_{r=1}^L \sum_{i \in P - F_r} Y_{ir}$ the total
number of results computed for a task \( \tau \) at termination. Note that the outer sum terms of \( Y \) consisting of the inner sums are mutually independent because each sum pertains to a different round; this allows us to use Chernoff bounds. From above it is clear that \( E[Y] = L \). Therefore, by applying Chernoff bound for the same \( \delta > 0 \) as chosen above we have:

\[
\Pr[Y \geq (1 + \delta)E[Y]] = \Pr[Y \geq (1 + \delta)L] \leq e^{-\frac{\delta^2 L \log n}{3}} \leq \frac{1}{n^{\alpha_2}},
\]

where \( \alpha_2 > 1 \) for a sufficiently large \( n \).

By applying Boole’s inequality on the above two events, we have

\[
\Pr[\{X \leq (1 - \delta)\frac{L}{n} \sum_{i \in P - F_r} q_i\} \cup \{Y \geq (1 + \delta)L\}] \leq \frac{2}{n^\alpha}
\]

where \( \alpha = \min\{\alpha_1, \alpha_2\} > 1 \)

Therefore, from above and by using \((1 - \delta)\frac{L}{n} \sum_{i \in P - F_r} q_i > \frac{1}{2}(1 + \delta)L\) we have \( \Pr[Y/2 < X] \geq 1 - \frac{1}{n^\beta} \) for some \( \beta > 1 \). Hence, at termination, \( \text{whp} \), the majority of calculated results for task \( \tau \) are correct. Let us denote this event by \( \mathcal{E}_\tau \). It follows that \( \Pr[\mathcal{E}_\tau] \leq \frac{1}{n^\gamma} \). Now, by Boole’s inequality we obtain

\[
\Pr[\bigcup_{\tau \in \mathcal{T}} \mathcal{E}_\tau] \leq \sum_{\tau \in \mathcal{T}} \Pr[\mathcal{E}_\tau] \leq \frac{1}{n^{\beta - 1}} \leq \frac{1}{n^\gamma}
\]

where \( \mathcal{T} \) is the set of all \( n \) tasks, and \( \gamma > 0 \).

By Lemma 4.2.7, \( \text{whp} \) all calculated results of every task are disseminated across all workers. Thus, the majority of the results computed for any task at any worker is the same among all workers, and moreover it is correct \( \text{whp} \). Recall that according to our algorithm (line 14 of Figure 8) every processor computes the results of every task by taking the plurality of calculated results, and hence the claim of the theorem. \( \Box \)
Theorem 4.2.9  Algorithm NS solves the network supercomputing problem under the failure model \( \langle F_{\ell f}, E_{\frac{1}{2}} \rangle \), whp, with work and message complexity \( \Theta(n \log n) \).

**Proof.** Since algorithm NS terminates in \( \Theta(\log n) \) rounds its work cannot exceed \( \Theta(n \log n) \). Clearly the message complexity of algorithm NS remains unchanged, and it is \( \Theta(n \log n) \). □

Although the complexity results do not change in the presence of crashes, it is important to note that the overall number of rounds may increase by a constant factor of \( \lambda = \frac{1}{h} \). This is because, as we argued in Lemma 4.2.6, in the presence of at most \((1 - h)n\) crashes it is sufficient to increase the number of rounds by a constant factor \( \lambda \) for the claim to hold whp.

Finally, the algorithm is extended, as discussed in Section 4.1, to deal with \( t \) tasks when \( t \geq n \). Given Theorem 4.2.8, the complexity bounds established in Theorem 4.2.5 remain unchanged under failure model \( \langle F_{\ell f}, E_{\frac{1}{2}} \rangle \). The following corollary summarizes this result for algorithm NS\( t,n \) for \( t \) tasks, where \( t \geq n \).

Corollary 4.2.10  Algorithm NS\( t,n \) solves the network supercomputing problem under the failure model \( \langle F_{\ell f}, E_{\frac{1}{2}} \rangle \), whp, with time complexity \( \Theta(\frac{t}{n} \log n) \), work complexity \( \Theta(t \log n) \), and message complexity \( \Theta(n \log n) \).

4.3 Decentralized NSP Under Failure Models \( \langle F_{fp}, E_{\frac{1}{2}} \rangle \) and \( \langle F_{pt}, E_{\frac{1}{2}} \rangle \)

In this section we present a different algorithm for the problem of using network supercomputing to perform a large collection of independent tasks, while dealing with undependable processors. As before, the adversary may cause the processors to return bogus results for tasks with certain probabilities, and may cause a subset \( F \) of the initial set of processors \( P \) to crash. We consider the problem under failure models \( \langle F_{fp}, E_{\frac{1}{2}} \rangle \) and \( \langle F_{pt}, E_{\frac{1}{2}} \rangle \) described
in Section 4.1. Both models yield adversaries that are much stronger than those studied in Section 4.2. Our randomized synchronous algorithm is formulated for \( n \) processors and \( n \) tasks, where depending on the number of crashes each live processor is able to terminate dynamically with the knowledge that the problem is solved with high probability. The algorithm naturally generalizes for \( t \) tasks, where \( t \geq n \), by having processors work on groups \( \lceil t/n \rceil \) tasks instead of single tasks. Under failure model \( \langle \mathcal{F}_{fp}, \mathcal{E}_{\frac{1}{2}} \rangle \) the time complexity of the algorithm is \( O(\frac{t}{n^a} \log n \log \log n) \), its work is \( O(t \log n \log \log n) \) and message complexity is \( O(n \log n \log \log n) \). Under failure model \( \langle \mathcal{F}_{p\ell}, \mathcal{E}_{\frac{1}{2}} \rangle \) the time complexity is \( O(n) \), work is \( O(t \cdot n^a) \), and message complexity is \( O(n^{1+a}) \). Additionally, we show that for model \( \langle \mathcal{F}_{\ell f}, \mathcal{E}_{\frac{1}{2}} \rangle \) algorithm’s time, work, and message complexities match the respective complexities of Algorithm NS (Figure 8 on page 90), presented in Section 4.2. All bounds are shown to hold with high probability.

4.3.1 Description of Algorithm DAKS

Algorithm DAKS (for Decentralized Algorithm with Knowledge Sharing) employs no master and instead uses a gossip-based approach. We start by specifying in detail the algorithm for \( n \) processors and \( t = n \) tasks, then we generalize it for \( t \) tasks, where \( t \geq n \).

The algorithm is structured in terms of a main loop. The principal data structures at each processor are two arrays of size linear in \( n \): one accumulates knowledge gathered from the processors, and another stores the results. All processors start as workers. In each iteration, any worker performs one randomly selected task and sends its knowledge to just one other randomly selected processor. When a worker obtains “enough” knowledge about the tasks
Procedure for processor $i$:

1: \textbf{external} $n, /\!* n$ is the number of processors and tasks */
2: \textbf{c, k} /\!* positive constants */
3: \textbf{Task}[$1..n$] /* set of tasks */
4: \textbf{R}[$i$][$1..n$] \textbf{init} $\emptyset$ /* set of collected results */
5: \textbf{Results}[$i$][$1..n$] \textbf{init} $\emptyset$ /* array of results */
6: \textbf{prof ctr} \textbf{init} 0 /* number of profess messages received */
7: \textbf{r} \textbf{init} 0 /* round number */
8: \textbf{ℓ} \textbf{init} 0 /* number of profess messages to be sent per iteration */
9: \textbf{worker} \textbf{init} true /* indicates whether the processor is still a worker */

10: \textbf{while} prof ctr $< c \log n$ \textbf{do}
11: \textbf{Send:}
12: \textbf{if} worker \textbf{then}
13: \textbf{Let} $q$ be a randomly selected processor from $P$
14: \textbf{Send} $\langle$share, $R_i[]\rangle$ to processor $q$
15: \textbf{else}
16: \textbf{Let} $D$ be a set of $2^\ell \log n$ randomly selected processors from $P$
17: \textbf{Send} $\langle$profess, $R_i[]\rangle$ to processors in $D$
18: \textbf{ℓ} $\leftarrow$ \textbf{ℓ} + 1

19: \textbf{Receive:}
20: \textbf{prof ctr} $\leftarrow$ prof ctr + $\{|m : m \in M \land m.type = profess\}$
21: \textbf{for all} $j \in T$ \textbf{do}
22: \textbf{R}_i[j] $\leftarrow$ $R_i[j] \cup (\bigcup_{m \in M} m.R[j])$ /* update knowledge */

23: \textbf{Compute:}
24: \textbf{r} $\leftarrow$ \textbf{r} + 1
25: \textbf{if} worker \textbf{then}
26: \textbf{Randomly select} $j \in T$ \textbf{and compute the result} $v_j$ \textbf{for} Task$[j]$
27: \textbf{R}_i[j] $\leftarrow$ $R_i[j] \cup \{v_j, i, r\}$
28: \textbf{if} $\min_{j \in T} (|R_i[j]|) \geq K \log n$ \textbf{then} /* $i$ has enough results */
29: \textbf{for all} $j \in T$ \textbf{do}
30: $\textbf{Results}_i[j] \leftarrow u$ \textbf{such that} triples $\langle u, \_ , \_ \rangle$ \textbf{form a plurality} \textbf{in} $R_i[j]$
31: \textbf{worker} $\leftarrow$ false /* worker becomes enlightened */

\textbf{end}

Figure 9: Algorithm DAKS for $t = n$; code at processor $i \in P$.

performed in the system, it computes the final results, stops being a worker, and becomes “enlightened.” Such processors no longer perform tasks, and instead “profess” their knowledge to other processors by means of multicasts to exponentially increasing random sets of processors.

The main loop terminates when a certain number of messages is received from enlightened processors. The pseudocode for algorithm DAKS is given in Figure 9. We now give the details.
Local knowledge and state. The algorithm is parameterized by \( n \), the number of processors and tasks, and by compile-time constants \( C \) and \( K \) that are discussed later (they emerge from the analysis). Every processor \( i \) maintains the following:

- Array of results \( R_i[1..n] \), where element \( R_i[j] \), for \( j \in T \), is a set of results for \( Task[j] \). Each \( R_i[j] \) is a set of triples \( \langle v, i, r \rangle \), where \( v \) is the result computed for \( Task[j] \) by processor \( i \) in round \( r \) (here the inclusion of \( r \) ensures that the results computed by processor \( i \) in different rounds are preserved).

- The array \( \text{Results}_i[1..n] \) stores the final results.

- The \( prof \_ctr \) stores the number of messages received from enlightened processors.

- \( r \) is the round (iteration) number that is used by workers to timestamp the computed results.

- \( \ell \) is the exponent that controls the number of messages multicast by enlightened processors.

Control flow. The algorithm iterations are controlled by the main while-loop, and we use the term round to refer to a single iteration of the loop. The loop contains three stages, viz., Send, Receive, and Compute.

Processors communicate using messages \( m \) that contain pairs \( \langle \text{type}, R[] \rangle \). Here \( m.R[] \) is the sender’s array of results. When a processor is a worker, it sends messages with \( m.type = \text{share} \). When a processor becomes enlightened, it sends messages with \( m.type = \text{profess} \). The loop is controlled by the counter \( prof \_ctr \) that keeps track of the received messages of type profess. We next describe the stages in detail.
Send stage: Any worker chooses a target processor $q$ at random and sends its array of results $R[\cdot]$ to processor $q$ in a share message. Any enlightened processor chooses a set $D \subseteq P$ of processors at random and sends the array of results $R[\cdot]$ to processors in $D$ in a profess message. The size of the set $D$ is $2^\ell \log n$, where initially $\ell = 0$, and once a processor is enlightened, it increments $\ell$ by 1 in every round. (Strictly speaking, $D$ is a multiset, because the random selection is with replacement. However this is done only for the purpose of the analysis, and $D$ can be safely treated as a set for the purpose of sending profess messages.)

Receive stage: Processor $i$ receives messages (if any) sent to it in the preceding Send stage. The processor increments its $prof.\ ctr$ by the number of profess messages received. For each task $j$, the processor updates its $R_i[j]$ by including the results received in all messages.

Compute stage: Any worker $i$ randomly selects a task $j$, computes the result $v_j$, and adds the triple $\langle v_j, i, r \rangle$ for round $r$ to $R_i[j]$. For each task the worker checks whether “enough” results were collected. Once at least $\kappa \log n$ results for each task are obtained, the worker stores the final results in $Results_i[\cdot]$ by taking the plurality of results for each task, and becomes enlightened. In our analysis we establish that $\kappa \log n$ results are sufficient for our claims. In Section 4.3.3 we reason about the computation of compile-time constant $\kappa$. Enlightened processors rest on their laurels in subsequent Compute stages.

Reaching Termination. We note that a processor must become enlightened before it can terminate. Processors can become enlightened at different times and without any synchronization. Once enlightened, they profess their knowledge by multicasting it to exponentially
growing random subsets \( D \) of processors. When a processor receives sufficiently many such messages, i.e., \( C \log n \), it halts, again without any synchronization, and using only the local knowledge. We consider this protocol to be of independent interest. In Section 4.3.2 we reason about the compile-time constant \( C \), and establish that \( C \log n \) profess messages are sufficient for our claims; additionally we show that the protocol’s efficiency can be assessed independently of the number of crashes.

### 4.3.2 Analysis of Algorithm \( \text{DAKS} \)

We present the performance analysis of algorithm \( \text{DAKS} \) in adversarial failure models \( \langle F_{fp}, E_1 \rangle \) and \( \langle F_{pt}, E_\frac{1}{2} \rangle \). We first present the analysis that deals with the case when \( t = n \), then extend the results to the general case with \( t \geq n \) for algorithm \( \text{DAKS}_{t,n} \). We proceed by giving lemmas relevant to both adversarial models.

The following lemma shows that if \( (n \log n) \) profess messages are sent by the enlightened processors, then every live processor terminates \( \text{whp} \) in one round.

**Lemma 4.3.1** In any execution of algorithm \( \text{DAKS} \) if there exists a round \( r \) by which the total number of profess messages is \( \Theta(n \log n) \) then by the end of round \( r \) every live processor halts \( \text{whp} \).

**Proof.** Let \( \bar{n} = kn \log n \) be the number of profess messages sent by round \( r \), where \( k > 1 \) is a sufficiently large constant. We show that \( \text{whp} \) every live processor receives at least \((1 - \delta)k \log n \) profess messages, for some constant \( \delta \in (0, 1) \). Let us assume that there exists processor \( q \) that receives less than \((1 - \delta)k \log n \) of such messages. We prove that \( \text{whp} \) such a processor does not exist.
Since \( \bar{n} \) profess messages are sent by round \( r \), there were \( \bar{n} \) random selections of processors from set \( P \) in line 15 of algorithm DAKS on page 105, possibly by different enlightened processors. We denote by \( i \) an index of one of the random selections in line 15. Let \( X_i \) be a Bernoulli random variable such that \( X_i = 1 \) if processor \( q \) was chosen by an enlightened processor and \( X_i = 0 \) otherwise.

We define a random variable \( X = \sum_{i=1}^{\bar{n}} X_i \) to estimate the total number of times processor \( q \) is selected by round \( r \). In line 15 every enlightened processor chooses a set of destinations for the profess message uniformly at random, and hence \( \Pr[X_i = 1] = \frac{1}{n} \). Let \( \mu = \mathbb{E}[X] = \sum_{i=1}^{\bar{n}} X_i = \frac{1}{n} k n \log n = k \log n \), then by applying Chernoff bound, for the same \( \delta \) chosen as above, we have:

\[
\Pr[X \leq (1 - \delta)\mu] \leq e^{-\frac{\mu\delta^2}{2}} \leq e^{-\frac{(k \log n)\delta^2}{2n}} \leq \frac{1}{n^{\alpha}}
\]

where \( \alpha > 1 \) for some sufficiently large \( b \). We now define \( C \) to be \( C = (1 - \delta)k \). Thus, with this \( C \), we have \( \Pr[X \leq C \log n] \leq \frac{1}{n^\alpha} \) for some \( \alpha > 1 \). Now let us denote by \( E_{q} \) the fact that \( prof_{ctr_{q}} \geq C \log n \) by the end of round \( r \), and let \( \bar{E}_{q} \) be the complement of that event. By Boole’s inequality we have \( \Pr[\bigcup_{q}\bar{E}_{q}] \leq \sum_{q} \Pr[\bar{E}_{q}] \leq \frac{1}{n^\beta} \), where \( \beta = \alpha - 1 > 0 \). Hence each processor \( q \in P \) is the destination of at least \( C \log n \) profess messages whp, i.e.,

\[
\Pr[\bigcap_{q}E_{q}] = \Pr[\bigcup_{q}\bar{E}_{q}] = 1 - \Pr[\bigcup_{q}\bar{E}_{q}] \geq 1 - \frac{1}{n^\beta}
\]

and hence, it halts (line 10).

\( \square \)

We use the constant \( C \) from the proof of Lemma 4.3.1 as a compile-time constant in algorithm DAKS (Figure 9). The constant is used in the main while loop (line 10) to determine
when a sufficient number of profess messages is received from enlightened processors, causing
the loop to terminate.

**Lemma 4.3.2** In any execution of algorithm DAKS if a processor \( q \in P - F \) is enlightened in
round \( r \), then after additional \( O(\log n) \) rounds every live processor halts \( \text{whp} \).

**Proof.** According to Lemma 4.3.1 if \( \Theta(n \log n) \) profess messages are sent then every processor
halts \( \text{whp} \). Given that processor \( q \) does not crash it takes \( q \) at most \( \log n \) rounds to send \( n \log n \)
profess messages (per line 15 in Figure 9), regardless of the actions of other processors. Hence,
\( \text{whp} \) every live processor halts in \( O(\log n) \) rounds. \( \square \)

Next we establish the work and message complexities for algorithm DAKS for the case
when number of crashes is small, specifically when at least a linear number of processors do not
crash. As we mentioned at the beginning of this chapter, while \( \Omega(n) \) processors remain active
in the computation, algorithm DAKS performs tasks in exactly the same pattern as algorithm
NS in Section 4.2. This forms the basis for the next lemma.

**Lemma 4.3.3** Algorithm DAKS solves the network supercomputing problem under the failure
model \( \langle F_f, E_2 \rangle \), \( \text{whp} \), with work and message complexity \( \Theta(n \log n) \) when \( \Omega(n) \) processors
do not crash.

**Proof sketch.** Algorithm DAKS chooses tasks to perform in the same pattern as algorithm NS
in Section 4.2, however the two algorithms have very different termination strategies. Theo-
rem 4.2.9 of Section 4.2 establishes that in the presence of at most \( (1 - h) \cdot n \) crashes, for a
constant \( h \in (0, 1) \), the work and message complexities of algorithm NS are \( \Theta(n \log n) \). The
termination strategy of algorithm DAKS is completely different, however, per Lemmas 4.3.1
and 4.3.2, algorithm DAKS halts in \( \Theta(\log n) \) rounds after all live processors are enlightened, and the number of \texttt{profess} messages is \( \Theta(n \log n) \). Thus, with at least a linear number of processors remaining, the work and message complexities, relative to algorithm NS, increase by an additive \( \Theta(n \log n) \) term. The result follows.

We denote by \( L \) the number of rounds required for a processor from the set \( P - F \) to become enlightened. Recall, that in Section 4.2 parameter \( L \) determines the number of rounds required for algorithm NS to terminate. Here we prove that once a processor is enlightened, then every live processor halts in \( O(\log n) \) rounds. Hence, asymptotically \( L \) still determines the number of rounds required for a processor from the set \( P - F \) to terminate. This is because, as we show later in this section, it takes more than \( O(\log n) \) rounds for a processor from the set \( P - F \) to become enlightened in models \( \langle F_{fp}, E_{\frac{1}{2}} \rangle \) and \( \langle F_{p\ell}, E_{\frac{1}{2}} \rangle \). We next analyze the value of \( L \) for models \( \langle F_{fp}, E_{\frac{1}{2}} \rangle \) and \( \langle F_{p\ell}, E_{\frac{1}{2}} \rangle \).

### 4.3.2.1 Analysis of Algorithm DAKS for Failure Model \( \langle F_{fp}, E_{\frac{1}{2}} \rangle \)

In model \( \langle F_{fp}, E_{\frac{1}{2}} \rangle \) we have \( |F| \leq n - n^a \). Let \( F_r \) be the actual number of crashes that occur prior to round \( r \). For the purpose of analysis we divide an execution of the algorithm into two epochs: epoch\( a \) consists of all rounds \( r \) where \( |F_r| \) is at most linear in \( n \), so that when the number of live processors is at least \( c'n \) for some suitable constant \( c' \); epoch\( b \) consists of all rounds \( r \) starting with first round \( r' \) (it can be round 1) when the number of live processors drops below some \( c'n \) and becomes \( c''n^a \) for some suitable constant \( c'' \). Note that either epoch may be empty.
For the small number of crashes in epoch \(a\), Lemma 4.3.3 gives the worst case work and message complexities as \(\Theta(n \log n)\); the upper bounds apply whether or not the algorithm terminates in this epoch.

In the rest of this section we assume that the algorithm does not terminate in epoch \(a\) and we analyze the costs for epoch \(b\) next. If the algorithm terminates in round \(r'\), the first round of the epoch, the cost remains the same as given by Lemma 4.3.3. If it does not terminate, it incurs additional costs associated with the processors in \(P - F_{r'}\), where \(|P - F_{r'}| \leq c''n^a\). We analyze the costs for epoch \(b\) in the rest of this section. The final message and work complexities will be at most the worst case complexity for epoch \(a\) plus the additional costs for epoch \(b\) incurred while \(|P - F| = \Omega(n^a)\) per model \(\langle F_{fp}, E_\frac{1}{2} \rangle\).

First we show that \(whp\) it will take \(L = O(n^{1-a} \log n \log \log n)\) rounds for a worker from the set \(P - F\) to become enlightened in epoch \(b\).

**Lemma 4.3.4** In any execution of algorithm DAKS under the failure model \(\langle F_{fp}, E_\frac{1}{2} \rangle\), after \(O(n^{1-a} \log n)\) rounds of epoch \(b\) every task is performed \(\Theta(\log n)\) times by processors in \(P - F\), \(whp\).

**Proof.** If the algorithm terminates within \(O(n^{1-a} \log n)\) rounds of epoch \(b\), then each task is performed \(\Theta(\log n)\) times as reasoned earlier. Suppose the algorithm does not terminate (in this case its performance is going to be worse).

Let us assume that after \(\tilde{r} = \kappa n^{1-a} \log n\) rounds of algorithm DAKS, where \(\kappa\) is a sufficiently large constant and \(0 < a < 1\) is a constant, there exists a task \(\tau\) that is performed less than \((1 - \delta)\kappa \log n\) times among all live workers, for some \(\delta > 0\). We prove that \(whp\) such a task does not exist.
We define $k_2$ to be such that $k_2 = (1 - \delta)\kappa$ (the constant $k_2$ will play a role in establishing the value of the compile-time constant $K$ of algorithm DAKS; we come back to this in Section 4.3.3). According to the above assumption, at the end of round $\tilde{r}$ for some task $\tau$, we have $|\cup_{j=1}^n R_j[\tau]| < (1 - \delta)\kappa \log n = k_2 \log n$.

Let us consider all algorithm iterations individually performed by each processor in $P - F$ during the $\tilde{r}$ rounds. Let $\xi$ be the total number of such individual iterations. Then $\xi \geq \tilde{r}|P - F| \geq cn^\alpha$. During any such iteration, a processor from $P - F$ selects and performs task $\tau$ in line 24 independently with probability $\frac{1}{n}$. Let us arbitrarily enumerate said iterations from 1 to $\xi$. Let $X_1, \ldots, X_\xi, \ldots, X_\xi$ be Bernoulli random variables, such that $X_x$ is 1 if task $\tau$ is performed in iteration $x$, and 0 otherwise. We define $X = \sum_{x=1}^\xi X_x$, the random variable that describes the total number of times task $\tau$ is performed during the $\tilde{r}$ rounds by processors in $P - F$. We define $\mu$ to be $\mathbb{E}[X]$. Since $\Pr[X_x = 1] = \frac{1}{n}$, for $x \in \{1, \ldots, \xi\}$, where $\xi \geq \tilde{r}cn^\alpha$, by linearity of expectation, we obtain $\mu = \mathbb{E}[X] = \sum_{x=1}^\xi \frac{1}{n} = \frac{\tilde{r}cn^\alpha}{n} > \kappa \log n$. Now by applying Chernoff bound, for the same $\delta > 0$ chosen as above, we have:

$$\Pr[X \leq (1 - \delta)\mu] \leq e^{-\frac{\mu\delta^2}{2}} \leq e^{-\frac{(\kappa \log n)\delta^2}{2}} \leq \frac{1}{n^b} \leq \frac{1}{n^\alpha}$$

where $\alpha > 1$ for some sufficiently large $b$. Now let us denote by $\mathcal{E}_\tau$ the fact that $|\cup_{i=1}^n R_i(\tau)| > k_2 \log n$ by the round $\tilde{r}$ of the algorithm and we denote by $\mathcal{E}_\tau$ the complement of that event.

Next by Boole’s inequality we have $\Pr[\cup_{\tau} \mathcal{E}_\tau] \leq \sum_{\tau} \Pr[\mathcal{E}_\tau] \leq \frac{1}{n^\beta}$, where $\beta = \alpha - 1 > 0$.

Hence each task is performed at least $\Theta(\log n)$ times by workers in $P - F$ whp, i.e.,

$$\Pr[\cap_{\tau} \mathcal{E}_\tau] = \Pr[\cup_{\tau} \mathcal{E}_\tau] \geq 1 - \frac{1}{n^\beta}. \quad \square$$

We now focus only on the set of live processors $P - F$ with $|P - F| \geq cn^\alpha$. Our goal is to show that in $O(n^{1-\alpha} \log n \log \log n)$ rounds of algorithm DAKS at least one processor from
$P - F$ becomes enlightened. In reasoning about Lemmas 4.3.5, 4.3.6 and 4.3.7, that follow, we note that if the algorithm terminates within $O(n^{1-a} \log n \log \log n)$ rounds of epoch $b$, then every processor in $P - F$ is enlightened as reasoned earlier. Suppose the algorithm does not terminate (in focusing on this case we note that the algorithm’s performance is going to be worse).

We first show that any triple $z$ generated by a processor in $P - F$ is known to all processors in $P - F$ in $O(n^{1-a} \log n \log \log n)$ rounds of algorithm DAKS.

We denote by $S(r) \subseteq P - F$ the set of processors that know a certain triple $z$ by round $r$, and let $s(r) = |S(r)|$. Next lemma shows that after $r_1 = r' + \Theta(n^{1-a} \log n \log \log n)$ in epoch $b$ we have $s(r_1) = \Theta(\log^3 n)$.

**Lemma 4.3.5** In any execution of algorithm DAKS under the failure model $(\mathcal{F}_b, \mathcal{E}_1^2)$, by round $r_1 = r' + \Theta(n^{1-a} \log n \log \log n)$ of epoch $b$, $s(r_1) = \Theta(\log^3 n)$, whp.

**Proof.** Consider a scenario when a processor $p \in P - F$ generates a triple $z$. Then the probability that processor $p$ sends triple $z$ to some processor $q \in P - F$, where $p \neq q$, in $n^{1-a} \log n$ rounds is at least

$$1 - (1 - \frac{cn^a}{n})^{n^{1-a} \log n} \geq 1 - e^{-b \log n} > 1 - \frac{1}{n^\alpha}$$

such that $\alpha > 0$ for some appropriately chosen $b$ and for a sufficiently large $n$. Similarly, it is straightforward to show that the number of live processors that learn about $z$ doubles every $n^{1-a} \log n$ rounds, hence whp after $(n^{1-a} \log n) \cdot 3 \log \log n = O(n^{1-a} \log n \log \log n)$ rounds the number of processors in $P - F$ that learn about $z$ is $\Theta(\log^3 n)$. □

In the next lemma we reason about the growth of $s(r)$ after round $r_1$.  

114
Lemma 4.3.6 In any execution of algorithm DAKS under failure model \( \langle F_p, E_\frac{1}{2} \rangle \), let \( r_2 \) be the first round after round \( r_1 \) in epoch such that \( r_2 - r_1 = \Theta(n^{1-a} \log n) \). Then \( s(r_2) \geq \frac{3}{5}|P - F| \) \( \text{whp} \).

Proof. Per model \( \langle F_p, E_\frac{1}{2} \rangle \), let constant \( c \) be such that \( |P - F| \geq cn^a \). We would like to apply the Chernoff bound to approximate the number of processors from \( (P - F) - S(r_1) \) that learn about triple \( z \) by round \( r_2 \). According to algorithm DAKS if a processor \( i \in (P - F) - S(r_1) \) learns about triple \( z \) in some round \( r_1 < r < r_2 \), then in round \( r + 1 \) processor \( i \) forwards \( z \) to some randomly chosen processor \( j \in P \) (lines 12-13 of the algorithm). Let \( Y_i \), where \( i \in (P - F) - S(r_1) \), be a random variable such that \( Y_i = 1 \) if processor \( i \) receives the triple \( z \) from some processor \( j \in S(r_1) \), in some round \( r_1 < r < r_2 \), and \( Y_i = 0 \) otherwise. It is clear that if some processor \( k \in (P - F) - S(r) \), where \( k \neq i \) receives triple \( z \) from processor \( i \) in round \( r + 1 < r_2 \), then random variables \( Y_i \) and \( Y_k \) are not independent, and hence, the Chernoff bound cannot be applied. To circumvent this, we consider the rounds between \( r_1 \) and \( r_2 \) and partition these rounds into blocks of \( \frac{1}{c}n^{1-a} \) consecutive rounds. For instance, rounds \( r_1 + 1, \ldots, r_1 + \frac{1}{c}n^{1-a} \) form the first block, rounds \( r_1 + \frac{1}{c}n^{1-a} + 1, \ldots, r_1 + \frac{2}{c}n^{1-a} \) form the second block, etc. The final block may contain less than \( \frac{1}{c}n^{1-a} \) rounds. We are interested in estimating the fraction of the processors in \( (P - F) - S(r_1) \) that learn about triple \( z \) at the end of each block.

For the purpose of the analysis we consider another algorithm, called DAKS’. The difference between algorithms DAKS and DAKS’ is that in DAKS’ a processor does not forward triple \( z \) in round \( r \) if \( z \) was first received in the round that belongs to the same block as \( r \) does. This allows us to apply Chernoff bound (with negative dependencies) to approximate the number of
processors in \((P - F) - S(r_1)\) that learn about triple \(z\) in a block. We let \(S'(r)\) be the subset of processors in \(P - F\) that are aware of triple \(z\) by round \(r\) in algorithm \(\text{DAKS}'\), and we let \(s'(r) = |S'(r)|\). Note, that since in \(\text{DAKS}'\) triple \(z\) is forwarded less often than in \(\text{DAKS}\), it follows that the number of processors from \(P - F\) that learn about \(z\) in \(\text{DAKS}\) is at least as large as the number of processors from \(P - F\) that learn about \(z\) in \(\text{DAKS}'\), and, in particular, \(S'(r) \subseteq S(r)\), for any \(r\). This allows us to consider algorithm \(\text{DAKS}'\) instead of \(\text{DAKS}\) for assessing the number of processors from \(P - F\) that learn about \(z\) by round \(r_2\), and we do this by having \(s'(r)\) serve as a lower bound for \(s(r)\).

Let \(X_i\), where \(i \in (P - F) - S'(r)\), be a random variable, s.t. \(X_i = 1\) if processor \(i\) receives the triple \(z\) from some processor \(j \in S'(r)\) in a block that starts with round \(r + 1\), e.g., for the first block \(r = r_1\), and \(X_i = 0\) otherwise. Let us next define the random variable \(X = \sum_{i \in (P - F) - S'(r)} X_i = s'(r + \frac{1}{c} n^{1-a}) - s'(r)\) to count the number of processors in \((P - F) - S'(r)\) that received triple \(z\) in the block that starts with round \(r + 1\).

Next, we calculate \(\mathbb{E}[X]\), the expected number of processors in \((P - F) - S'(r)\) that learn about triple \(z\) at the end of the block that begins with round \(r + 1\) in algorithm \(\text{DAKS}'\). There are \(s'(r)\) processors in \(S'(r)\) that are aware of triple \(z\). The length of the block is \(\frac{1}{c} n^{1-a}\). On the other hand, every processor \(p \in (P - F) - S'(r)\) has a probability of \(\frac{1}{n}\) to be selected by a processor \(q \in S'(r)\) in one round. Conversely, the probability that \(p \in (P - F) - S'(r)\) is not selected by \(q\) is \(1 - \frac{1}{n}\). The number of trials is \(\frac{s'(r)}{c} n^{1-a}\), hence the probability that processor \(p \in (P - F) - S'(r)\) is not selected is \((1 - \frac{1}{n})^{s'(r)} n^{1-a}\). On the contrary, the probability that a processor \(p \in (P - F) - S'(r)\) is selected is \(1 - (1 - \frac{1}{n})^{s'(r)} n^{1-a}\). Therefore, the expected number of processors from \((P - F) - S'(r)\) that will learn about triple \(z\) by the end of the block...
in algorithm DAKS′ is \((cn^a - s'(r))(1 - (1 - \frac{1}{n})^{\frac{r}{c}n^{1-a}})\). Next, by applying the binomial expansion, we have:

\[
(cn^a - s'(r))(1 - (1 - \frac{1}{n})^{\frac{r}{c}n^{1-a}}) = cn^a(1 - \frac{s'(r)}{cn^a}) \cdot (1 - \frac{s'(r)n^{1-a}}{2cn}) = s'(r)(1 - \frac{s'(r)n^{1-a}}{2cn})
\]

The number of processors from \((P - F) - S'(r)\) that become aware of triple \(z\) in the block of \(\frac{1}{c}n^{1-a}\) rounds that starts with round \(r + 1\) is \(s'(r + \frac{1}{c}n^{1-a}) - s'(r)\). While, as shown above, the expected number of processors that learn about triple \(z\) is \(\mu = E[X] = s'(r)(1 - \frac{s'(r)}{cn^a})(1 - \frac{s'(r)n^{1-a}}{2cn})\).

On the other hand, because in algorithm DAKS′ no processor that learns about triple \(z\) in a block forwards it in the same block, we have negative dependencies among the random variables \(X_i\). And hence, we can apply the regular Chernoff bound, with \(\delta = \frac{1}{\log n}\). Considering also that \(s'(r) \geq s(r_1)\) and that \(s(r_1) = \Theta(\log^3 n)\) by Lemma 4.3.5, we obtain:

\[
\Pr[X \leq (1 - \frac{1}{\log n})\mu] \leq e^{-s'(r)(1 - \frac{s'(r)}{cn^a})(1 - \frac{s'(r)n^{1-a}}{2cn}) \cdot \frac{1}{\log^2 n}} \leq e^{-k \log^3 n / 2 \log^2 n} = e^{-k \log n} \leq \frac{1}{n^{\alpha}}
\]

where \(\alpha > 0\) for some sufficiently large \(k > 2\).
Therefore, \( whp \) the number of processors that learn about triple \( z \) in a block that starts with round \( r + 1 \) is

\[
s'(r + \frac{1}{c} n^{1-a}) \geq s'(r) + s'(r)(1 - \frac{s'(r)}{cn^a})(1 - \frac{s'(r)}{2cn^a})(1 - \frac{1}{\log n})
\]

\[
\geq s'(r) + s'(r)(1 - \frac{3s'(r)}{2cn^a})(1 - \frac{1}{\log n})
\]

\[
\geq s'(r) + s'(r)(1 - \frac{3}{2})(1 - \frac{1}{\log n})
\]

\[
\geq s'(r) \frac{21}{20}
\]

for a sufficiently large \( n \), and given that \( s'(r) < \frac{3}{5} cn^a \) (otherwise the lemma is proved).

Hence we showed that the number of processors from \((P - F) - S'(r)\) that learnt about triple \( z \) at the end of the block that starts with round \( r + 1 \), is at least \( \frac{21}{20} s'(r) \) \( whp \). It remains to show that \( s(r_2) \geq \frac{3}{5} |P - F| \) \( whp \). Indeed, even assuming that processors that learnt about triple \( z \) following round \( r \) do not disseminate it, after repeating the process described above for some \( O(\log n) \) times, it is clear that \( whp \ s'(r_2) \geq \frac{3}{5} |P - F| \). On the other hand since the block size is \( \frac{1}{n^{1-a}} \) and \( r_2 - r_1 = O(n^{1-a} \log n) \) there are \( O(\log n) \) blocks.

Thus \( whp \) we have \( s'(r_2) \geq \frac{3}{5} |P - F| \) for \( r_2 - r_1 = O(n^{1-a} \log n) \), and since \( S'(r) \subseteq S(r) \) we have \( s(r_2) \geq \frac{3}{5} |P - F| \).

Next we calculate the number of rounds required for the remaining \( \frac{2}{5} |P - F| \) processors in \( P - F \) to learn \( z \). Let \( U_d \subset P - F \) be the set of workers that do not learn \( z \) after \( O(n^{1-a} \log n \log \log n) \) rounds of algorithm DAKS. According to Lemma 4.3.6 we have \( |U_d| \leq \frac{2}{5} |P - F| \). In the proof of the next lemma we use the Coupon Collector’s problem (Definition 4.1.3 on page 87).
Lemma 4.3.7  In any execution of algorithm DAKS under the failure model \( \langle F_{fp}, E_{1/2} \rangle \), if every task is performed \( \Theta(\log n) \) times by round \( \rho \) of epoch \( b \) by processors in \( P - F \) then, after additional \( O(n^{1-a} \log n \log \log n) \) rounds, at least one worker from \( P - F \) is enlightened \( \text{whp} \).

Proof. According to Lemmas 4.3.5 and 4.3.6 in \( O(n^{1-a} \log n \log \log n) \) rounds of algorithm DAKS at least \( \frac{3}{5}|P - F| \) of the workers are aware of triple \( z \) generated by a processor in \( P - F \). Let us denote this subset of workers by \( S_d \), where \( d \) is the first such round.

We are interested in the number of rounds required for every processor in \( U_d \) to learn about \( z \) \( \text{whp} \) by receiving a message from a processor in \( S_d \) in some round following \( d \).

We show that, by the analysis similar to CCP, in \( O(n^{1-a} \log n \log \log n) \) rounds triple \( z \) is known to all processors in \( P - F \), \( \text{whp} \). Every processor in \( P - F \) has a unique id, hence we consider these processors as different types of coupons and we assume that the processors in \( S_d \) collectively represent the coupon collector. In this case, however, we do not require that every processor in \( S_d \) contacts all processors in \( U_d \) \( \text{whp} \). Instead, we require only that the processors in \( S_d \) collectively contact all processors in \( U_d \) \( \text{whp} \). According to our algorithm in every round every processor in \( P - F \) \( (S_d \subset P - F) \), selects a processor uniformly at random and sends all its data to it. Let us denote by \( m \) the collective number of trials by processors in \( S_d \) to contact processors in \( U_d \). According to CCP if \( m = O(n \ln n) \) then \( \text{whp} \) processors in \( S_d \) collectively contact every processor in \( P - F \), including those in \( U_d \). Since there are at least \( \frac{3}{5}cn^a \) processors in \( S_d \) then in every round the number of trials is at least \( \frac{3}{5}cn^a \), hence in \( O(n^{1-a} \ln n) \) rounds \( \text{whp} \) all processors in \( U_d \) learn about \( z \). Therefore, in \( O(n^{1-a} \log n \log \log n) \) rounds \( \text{whp} \) all processors in \( U_d \), and thus in \( P - F \), learn about \( z \).
Let $V$ be the set of triples such that for every task $j \in T$ there are $\Theta(\log n)$ triples generated by processors in $P - F$, and hence $|V| = \Theta(n \log n)$. Now by applying Boole’s inequality we want to show that whp in $O(n^{1-a} \log n \log \log n)$ rounds all triples in $V$ become known to all processors in $P - F$.

Let $\overline{E}_z$ be the event that some triple $z \in V$ is not known to all processors in $P - F$. In the preceding part of the proof we have shown that $\Pr[\overline{E}_z] < \frac{1}{n^\beta}$, where $\beta > 1$. By Boole’s inequality, the probability that there exists one triple in $V$ that is not known to all processors in $P - F$ can be bounded as

$$\Pr[\bigcup_{z \in V} \overline{E}_z] \leq \sum_{z \in V} \Pr[\overline{E}_z] = \Theta(n \log n) \frac{1}{n^{\beta}} \leq \frac{1}{n^\gamma}$$

where $\gamma > 0$. This implies that every processor in $P - F$ collects all $\Theta(n \log n)$ triples generated by processors in $P - F$, whp. And hence, at least one of these processors becomes enlightened after $O(n^{1-a} \log n \log \log n)$ rounds.

**Theorem 4.3.8** Algorithm DAKS solves the network supercomputing problem for $t = n$ under the failure model $(\mathcal{F}_{fp}, \mathcal{E}_2)$ in epoch $b$ after $O(n^{1-a} \log n \log \log n)$ rounds, whp.

**Proof.** According to algorithm DAKS (line 28) every live processor computes the result of every task $\tau$ by taking a plurality among all the results. We want to prove that the majority of the results for any task $\tau$ are correct at any enlightened processor, whp.

To do that, for a task $\tau$ we estimate (with a concentration bound) the number of times the results are computed correctly, then we estimate the bound on the total number of times task $\tau$ is computed (whether correctly or incorrectly), and we show that a majority of the results are computed correctly.
Let us consider random variables $X_{ir}$ that denote the success or failure of correctly computing the result of some task $\tau$ in round $r$ by worker $i$. Specifically, $X_{ir} = 1$ if in round $r$, worker $i$ computes the result of the task $\tau$ correctly, otherwise $X_{ir} = 0$. According to our algorithm we observe that for a live processor $i$ we have $\Pr[X_{ir} = 1] = \frac{q_i}{n}$ and $\Pr[X_{ir} = 0] = 1 - \Pr[X_{ir} = 1]$, where $q_i \equiv 1 - p_i$. We want to count the number of correct results calculated for task $\tau$ when a processor $i \in P - F$ becomes enlightened. As before, we let $F_r$ be the set of processors that crashes prior to round $r$.

Let $X_r \equiv \sum_{i \in P - F_r} X_{ir}$ denote the number of correctly computed results for task $\tau$ among all live workers during round $r$. By linearity of expected values of a sum of random variables we have

$$\mathbb{E}[X_r] = \mathbb{E}\left[ \sum_{i \in P - F_r} X_{ir} \right] = \sum_{i \in P - F_r} \mathbb{E}[X_{ir}] = \sum_{i \in P - F_r} \frac{q_i}{n}$$

We denote by $L'$ the minimum number of rounds required for at least one processor from $P - F'$ to become enlightened. It follows from line 26 of algorithm DAKS that a processor becomes enlightened only when there are at least $K \log n$ results for every task $\tau \in \mathcal{T}$ (the constant $K$ is chosen later in this section). We see that $L' \geq \bar{c}n^{1-a} \log n$, where $0 < \bar{c} \leq 1$. This is because there are $t = n$ tasks to be performed, and in epoch $b$ we have $|P - F| \geq cn^a$ for a constant $c \geq 1$.

We further denote by $X \equiv \sum_{r=1}^{L'} X_r$ the number of correctly computed results for task $\tau$ when the condition in line 26 of the algorithm is satisfied. Again, using the linearity of expected values of a sum of random variables we have

$$\mathbb{E}[X] = \mathbb{E}\left[ \sum_{r=1}^{L'} X_r \right] = \sum_{r=1}^{L'} \mathbb{E}[X_r] = \sum_{r=1}^{L'} \sum_{i \in P - F_r} \frac{q_i}{n} = \frac{L'}{n} \sum_{i \in P - F_r} q_i$$
Note that, according to our adversarial model definition, for every round $r \leq L'$ we have
\[ \frac{1}{|P - F|} \sum_{i \in P - F} q_i > \frac{1}{2} + \zeta', \]
for some fixed $\zeta' > 0$. Note also that $\frac{1}{|P - F|} \sum_{i \in P - F} q_i \geq \frac{1}{n} \sum_{i \in P - F} q_i$, and hence, there exists some $\delta > 0$, such that, $(1 - \delta)\frac{L'}{n} \sum_{i \in P - F} q_i > (1 + \delta)\frac{L'}{2}$. Also, observe that the random variables $X_1, X_2, \ldots, X_{L'}$ are mutually independent, since we consider an oblivious adversary and the random variables correspond to different rounds of execution of the algorithm. Therefore, by applying Chernoff bound on $X_1, X_2, \ldots, X_{L'}$ we have:

\[ \Pr[X \leq (1 - \delta)\mathbb{E}[X]] = \Pr[X \leq (1 - \delta)\frac{L'}{n} \sum_{i \in P - F} q_i] \leq e^{-\frac{\delta^2 L'(1 + \delta)}{4(1 - \delta)}} \leq \frac{1}{n^{\alpha_1}}, \]

where $L' \geq \tilde{c}n^{1-a} \log n$ as above and $\alpha_1 > 1$ for a sufficiently large $n$.

Let us now count the total number of times task $\tau$ is chosen to be performed during the execution of the algorithm until every live processor halts. We represent the choice of task $\tau$ by worker $i$ during round $r$ by a random variable $Y_{ir}$. We assume $Y_{ir} = 1$ if $\tau$ is chosen by worker $i$ in round $r$, otherwise $Y_{ir} = 0$.

At this juncture, we address a technical point regarding the total number of results for $\tau$ used for computing plurality. Note that even after round $L'$ any processor that is still a worker continues to perform tasks, thereby adding more results for task $\tau$. According to Lemma 4.3.2 every processor is enlightened in $O(\log n)$ rounds after $L'$. Furthermore, in epoch $b$ following round $L'$, the number of processors that are still workers is $n'' < |P - F|$. Hence, the expected number of results computed for every task $\tau$ by workers is $k \frac{\log n}{n''}$, for some $k > 0$, that is, $O(\frac{1}{n''})$, for some $a' > 0$. Therefore, the number of results computed for task $\tau$, starting from round $L'$ and until the termination is negligible. Let us denote by $Y$ the total number of results computed for a task $\tau$ at termination. We express the random variable $Y$ as $Y \equiv
\[ \sum_{\tau=1}^{L} \sum_{i \in P-F_{\tau}} Y_{i\tau}, \text{ where } L \text{ is the last round prior to termination. As argued above, the total number of results computed for task } \tau \text{ between rounds } L' \text{ and } L \text{ is } O\left(\frac{1}{n^{\alpha'}}\right), \text{ for some } \alpha' > 0, \text{ and hence } \frac{L'}{L} = 1 + o(1). \text{ Note that the outer sum terms of } Y \text{ consisting of the inner sums are mutually independent because each sum pertains to a different round; this allows us to use Chernoff bounds. From above it is clear that } E[Y] = \tilde{c}n^{1-a}k \log n + \frac{1}{n^{a'(1)}}, \text{ and hence } L \leq 1 + o(1). \text{ Note that the outer sum terms of } Y \text{ consisting of the inner sums are mutually independent because each sum pertains to a different round; this allows us to use Chernoff bounds. From above it is clear that } E[Y] = \tilde{c}n^{1-a}k \log n + \frac{1}{n^{a'(1)}}. \text{ Therefore, by applying Chernoff bound for the same } \delta > 0 \text{ as chosen above we have:}

\[ \Pr[Y \geq (1 + \delta)E[Y]] = \Pr[Y \geq (1 + \delta)L'] \leq e^{-\delta^2 \tilde{c}n^{1-a}k \log n} \leq \frac{1}{n^{\alpha_2}}, \]

where } \alpha_2 > 1 \text{ for a sufficiently large } n.

Then, by applying Boole’s inequality on the above two events, we have

\[ \Pr\left[\left\{X \leq (1 - \delta)\frac{L'}{n} \sum_{i \in P-F_{\tau}} q_{i}\right\} \cup \left\{Y \geq (1 + \delta)L'\right\}\right] \leq \frac{2}{n^{\alpha}}, \]

where } \alpha = \min\{\alpha_1, \alpha_2\} > 1.

Therefore, from above, and from the fact that \( (1 - \delta)\frac{L'}{n} \sum_{i \in P-F_{\tau}} q_{i} > (1 + \delta)\frac{L'}{2} \), we have } \Pr[Y/2 < X] \geq 1 - \frac{1}{n^{\beta}} \text{ for some } \beta > 1. \text{ Hence, at termination, } whp, \text{ the majority of calculated results for task } \tau \text{ are correct. Let us denote this event by } E_\tau. \text{ It follows that}

\[ \Pr[E_\tau] \leq \frac{1}{n^{\gamma}}. \text{ Now, by Boole's inequality we obtain}

\[ \Pr\left[\bigcup_{\tau \in \mathcal{T}} E_\tau\right] \leq \sum_{\tau \in \mathcal{T}} \Pr[E_\tau] \leq \frac{1}{n^{\beta - 1}} \leq \frac{1}{n^{\gamma}}, \]

where } \mathcal{T} \text{ is the set of all } n \text{ tasks, and } \gamma > 0.

By Lemmas 4.3.2, 4.3.6 and 4.3.7, } whp, \text{ in } O(n^{1-a} \log n \log \log n) \text{ rounds of the algorithm, at least } \Theta(n \log n) \text{ triples generated by processors in } P-F \text{ are disseminated across all workers } whp. \text{ Thus, the majority of the results computed for any task at any worker is the same among all workers, and moreover these results are correct } whp. \square
According to Lemma 4.3.7, after $O(n^{1-a} \log n \log \log n)$ rounds of epoch $b$ at least one processor in $P - F$ becomes enlightened. Furthermore, once a processor in $P - F$ becomes enlightened, according to Lemma 4.3.2 after $O(\log n)$ rounds of the algorithm every live processor terminates, whp. Hence, the time complexity of algorithm DAKS in mode $\langle \mathcal{F}_{fp}, \mathcal{E}_1 \rangle$ is $O(n^{1-a} \log n \log \log n)$ whp. Next we assess work and message complexities.

**Theorem 4.3.9** Algorithm DAKS solves the network supercomputing problem under the failure model $\langle \mathcal{F}_{fp}, \mathcal{E}_1 \rangle$, whp, with work and message complexity $O(n \log n \log \log n)$.

**Proof.** To obtain the result we combine the costs associated with epoch $a$ with the costs of epoch $b$. The work and message complexity bounds for epoch $a$ are given by Lemma 4.3.3 as $\Theta(n \log n)$.

For epoch $b$ (if it is not empty), where $|P - F| = O(n^a)$, the algorithm terminates after $O(n^{1-a} \log n \log \log n)$ rounds whp and there are $\Theta(n^a)$ live processors, thus its work is $O(n \log n \log \log n)$. In every round if a processor is a worker it sends a share message to one randomly chosen processor. If a processor is enlightened then it sends profess messages to a randomly selected subset of processors. In every round $\Theta(n^a)$ share messages are sent. Since the algorithm terminates, whp, in $O(n^{1-a} \log n \log \log n)$ rounds, $\Theta(n \log n \log \log n)$ share messages are sent. On the other hand, according to Lemma 4.3.1, if during the execution of the algorithm $\Theta(n \log n)$ profess messages are sent then every processor terminates whp. Hence, the message complexity is $O(n \log n \log \log n)$.

The worst case costs of the algorithm correspond to the executions with non-empty epoch $b$, where the algorithm does not terminate early. In this case the costs from epoch $a$ are asymptotically absorbed into the worst case costs of epoch $b$ computed above. $\square$
Finally, we consider the efficiency of algorithm $\text{DAKS}_{t,n}$ for $t$ tasks, where $t \geq n$. Note that the only change in the algorithm is that, instead of one task, processors perform chunks of $t/n$ tasks. The communication pattern in the algorithm remains exactly the same. The following result is directly obtained from the analysis of algorithm $\text{DAKS}$ for $t = n$ by multiplying the time and work complexities by the size $\Theta(t/n)$ of the chunk of tasks; the message complexity is unchanged.

**Theorem 4.3.10** Algorithm $\text{DAKS}_{t,n}$, with $t \geq n$, solves the network supercomputing problem under the failure model $\langle F_p, E \rangle$, whp, with time complexity $O(\frac{t}{n} \log n \log \log n)$, work complexity $O(t \log n \log \log n)$, and message complexity $O(n \log n \log \log n)$.

**Proof.** For epoch $a$ algorithm $\text{DAKS}$ has time $\Theta(\log n)$, work $\Theta(t \log n)$, and message complexity is $\Theta(n \log n)$. The same holds for algorithm $\text{DAKS}_{t,n}$. For epoch $b$ algorithm $\text{DAKS}$ takes $O(n^{1-a} \log n \log \log n)$ iterations for at least one processor from the set $P - F$ to become enlightened whp. The same holds for $\text{DAKS}_{t,n}$, except that each iteration takes $\Theta(t/n)$ time due to the chunk size. This yields time complexity $O(\frac{t}{n} \log n \log \log n)$. Work complexity is then $O(t \log n \log \log n)$. Message complexity remains the same as for algorithm $\text{DAKS}$ at $O(n \log n \log \log n)$ as the number of messages does not change. The final assessment is obtained by combining the costs of epoch $a$ and epoch $b$. \hfill $\square$

### 4.3.2.2 Analysis of Algorithm $\text{DAKS}$ for Failure Model $\langle F_p, E_2 \rangle$

As before, we start with the analysis of algorithm $\text{DAKS}$, then extend the main result to algorithm $\text{DAKS}_{t,n}$. In the adversarial model $\langle F_p, E_2 \rangle$ we have $|P - F| = \Omega(\log^c n)$. We first note that when a large number of crashes make $|P - F| = \Theta(\text{poly} \log n)$, one may attempt a
trivial solution where all live processors perform all $t$ tasks. While this approach has efficient work, it does not guarantee that workers compute correct results; in fact, since the overall probability of live workers producing bogus results can be close to $\frac{1}{2}$, this may yield on the average just slightly more than $t/2$ correct results.

For executions in $\langle F_p, E_{\frac{1}{2}} \rangle$, let $|P - F|$ be at least $b \log^c n$, for specific constants $b$ and $c$ satisfying the model constraints. Let $F_r$ be the actual number of crashes that occur prior to round $r$. For the purpose of analysis we divide an execution of the algorithm into two epochs: epoch $b'$ consists of all rounds $r$ where $|F_r|$ remains bounded as in model $\langle F_{fp}, E_{\frac{1}{2}} \rangle$ (for reference, this epoch combines epoch $a$ and epoch $b$ from the previous section); epoch $c$ consists of all rounds $r$ starting with the first round $r''$ (it can be round 1) when the number of live processors drops below $b' n^a$, where $b'$ and $a$ are specified by the failure model $\langle F_{fp}, E_{\frac{1}{2}} \rangle$, but remains $\Omega(\log^c n)$ per model $\langle F_p, E_{\frac{1}{2}} \rangle$. Observe that since we are concerned with model $\langle F_p, E_{\frac{1}{2}} \rangle$, in the sequel we can chose any $a$, such that $0 < a < 1$. Also note that either epoch may be empty.

In epoch $b'$ the algorithm incurs costs exactly as in model $\langle F_{fp}, E_{\frac{1}{2}} \rangle$. If algorithm DAKS terminates in round $r''$, the first round of the epoch, the costs remain the same as the costs analyzed for $\langle F_{fp}, E_{\frac{1}{2}} \rangle$ in the previous section.

If it does not terminate, it incurs additional costs associated with the processors in $P - F_{r''}$, where $b \log^c n \leq |P - F_{r''}| \leq b' n^a$. We analyze the costs for epoch $c$ next. The final message and work complexities are then at most the worst case complexity for epoch $b'$ plus the additional costs for epoch $c$.  

126
In the next lemmas we use the fact that \(|P - F_{\text{wr}}| = \Omega((\log c)n)\). The first lemma shows that within some \(O(n)\) rounds in epoch \(c\) every task is chosen for execution \(\Theta(\log n)\) times by processors in \(P - F\) \(\text{whp}\).

**Lemma 4.3.11** In any execution of algorithm DAKS under the failure model \(\langle F_p, E_1 \rangle\), after \(O(n)\) rounds of epoch \(c\) every task is performed \(\Theta(\log n)\) times, \(\text{whp}\), by processors in \(P - F\).

**Proof.** If the algorithm terminates within \(O(n)\) rounds of epoch \(c\), then each task is performed \(\Theta(\log n)\) times as reasoned earlier. Suppose the algorithm does not terminate (its performance is worse in this case). Let us assume that after \(\tilde{r}\) rounds of algorithm DAKS, where \(\tilde{r} = \tilde{k}n\) (\(\tilde{k}\) is a sufficiently large constant), there exists a task \(\tau\) that is performed less than \((1 - \delta)\tilde{k}\log n\) times by the processors in \(P - F\), for some \(\delta > 0\). We prove that \(\text{whp}\) such a task does not exist.

We define \(k_3\) to be such that \(k_3 = (1 - \delta)\tilde{k}\) (the constant \(k_3\) will play a role in establishing the value of the compile-time constant \(K\) of algorithm DAKS; we come back to this in Section 4.3.3). According to the above assumption, at the end of round \(\tilde{r}\) for some task \(\tau\), we have

\[|\bigcup_{j=1}^{n} R_j[\tau]| < (1 - \delta)\tilde{k}\log n = k_3\log n.\]

Let us consider all algorithm iterations individually performed by each processor in \(P - F\) during the \(\tilde{r}\) rounds. Let \(\xi\) be the total number of such individual iterations. Then \(\xi \geq \tilde{r}|P - F| \geq \tilde{r}a\log^c n\). During any such iteration, a processor from \(P - F\) selects and performs task \(\tau\) in line 24 of algorithm DAKS on page 105 independently with probability \(\frac{1}{n}\). Let us arbitrarily enumerate said iterations from 1 to \(\xi\). Let \(X_1, \ldots, X_x, \ldots, X_\xi\) be Bernoulli random variables, such that \(X_x\) is 1 if task \(\tau\) is performed in iteration \(x\), and 0 otherwise. We define \(X = \sum_{x=1}^{\xi} X_x\), the random variable that describes the total number of times task \(\tau\) is performed
during the $\tilde{r}$ rounds by processors in $P - F$. We define $\mu$ to be $\mathbb{E}[X]$. Since $\Pr[X_x = 1] = \frac{1}{n}$, for $x \in \{1, \ldots, \xi\}$, where $\xi \geq \tilde{r}a \log^c n$, by linearity of expectation, we obtain $\mu = \mathbb{E}[X] = \sum_{x=1}^{\xi} \frac{1}{n} \geq \tilde{r}a \log^c n > k_3 \log n$. Now by applying Chernoff bound for the same $\delta > 0$ as chosen above, we have:

$$\Pr[X \leq (1 - \delta)\mu] \leq e^{-\frac{\mu \alpha^2}{2}} \leq e^{-\frac{(\tilde{r}a \log^c n) \alpha^2}{2}} \leq \frac{1}{n^{k \log^c n \alpha}} \leq \frac{1}{n^\alpha},$$

where $\alpha > 1$ for some sufficiently large $k$. Now let us denote by $\mathcal{E}_r$ the fact that $|\cup_{i=1}^r R_i(\tau)| > k_3 \log n$ by round $\tilde{r}$ of the algorithm, and we let $\bar{\mathcal{E}}_r$ be the complement of that event. Next, by Boole’s inequality we have $\Pr[\cup_r \bar{\mathcal{E}}_r] \leq \sum_r \Pr[\bar{\mathcal{E}}_r] \leq \frac{1}{n^\beta}$, where $\beta = \alpha - 1 > 0$. Hence each task is performed at least $\Theta(\log n)$ times whp, i.e., $\Pr[\cap_r \mathcal{E}_r] = \Pr[\cup_r \bar{\mathcal{E}}_r] \geq 1 - \frac{1}{n^\beta}$. \hfill $\Box$

Next we show that once each task is done a logarithmic number of times by processors in $P - F$, then at least one worker in $P - F$ acquires a sufficient collection of triples in at most a linear number of rounds to become enlightened. We note that if the algorithm terminates within $O(n)$ rounds of epoch $c$, then every processor in $P - F$ is enlightened as reasoned earlier. Suppose the algorithm does not terminate (leading to its worst case performance).

**Lemma 4.3.12** In any execution of algorithm DAKS under failure model $\langle F_{pl}, E_{\frac{1}{2}} \rangle$, if every task is performed $\Theta(\log n)$ times by round $\rho$ of epoch $c$ by processors in $P - F$ then, after additional $O(n)$ rounds, at least one worker from $P - F$ is enlightened whp.

**Proof.** Assume that after $r$ rounds of algorithm DAKS, every task $j \in \mathcal{T}$ is done $\Theta(\log n)$ times by processors in $P - F$, and let $\mathcal{V}$ be the set of corresponding triples in the system. Consider a triple $z \in \mathcal{V}$ that was generated in some round $\tilde{r}$. We want to prove that whp it takes $O(n)$ rounds for the rest of the processors in $P - F$ to learn about $z$. 

128
Let $\Lambda(n)$ be the number of processors in $P - F$, then $|P - F| = \Lambda(n) \geq a \log^c n$, by the constraint of model $\langle F_p^\ell, \mathcal{E}_2 \rangle$. While there may be more than $\Lambda(n)$ processors that start epoch $c$, we focus only on the processors in $P - F$. This is sufficient for our purpose of establishing an upper bound on the number of rounds of at least one worker becoming enlightened: in line 12 of algorithm DAKS every live processor chooses a destination for a share message uniformly at random, and hence having more processors will only cause a processor in $P - F$ becoming enlightened quicker.

Let $Z(r) \subseteq P - F$ be the set of processors that becomes aware of triple $z$, in round $r$. Beginning with round $\tilde{r}$ when the triple is generated, we have $|Z(\tilde{r})| \geq 1$ (at least one processor is aware of the triple). For any rounds $r'$ and $r''$, where $\tilde{r} \leq r' \leq r''$, we have $Z(\tilde{r}) \subseteq Z(r') \subseteq Z(r'') \subseteq P - F$ because the considered processors that become aware of $z$ do not crash; thus $|Z(r)|$ is monotonically non-decreasing with respect to $r$.

We want to estimate an upper bound on the total number of rounds $r$ required for $|Z(r)|$ to become $\Lambda(n)$. We will do this by constructing a sequence of random mutually independent variables, each corresponding to a contiguous segment of rounds $r_1, ..., r_k$, for $k \geq 1$ in an execution of the algorithm. Let $r_0$ be the round that precedes round $r_1$. Our contiguous segment of rounds has the following properties: (a) $|Z(r_x)| = |Z(r_0)|$ for $1 \leq x < k$, where during such rounds $r_x$ the set $Z(r_x)$ does not grow (the set of such rounds may be empty), and (b) $|Z(r_k)| > |Z(r_0)|$, i.e., the size of the set grows.

For the purposes of analysis, we assume that $|Z(r_k)| = |Z(r_0)| + 1$, i.e., the set grows by exactly one processor. Of course it is possible that this set grows by more than a unity in one round. Thus we consider an ‘amnesiac’ version of the algorithm where if more than one processor learns about the triple, then all but one processor ‘forget’ about that triple. The
information is propagated slower in the amnesiac algorithm, but this is sufficient for us to establish the needed upper bound on the number of rounds needed to propagate the triple in question.

Consider some round \( r \) with \( |Z(r)| = \lambda \). We define a random variable \( T_\lambda \) that represents the number of rounds required for \( |Z(r + T_\lambda)| = \lambda + 1 \), i.e., \( T_\lambda \) corresponds to the number \( k \) of rounds in the contiguous segment of rounds we defined above. The random variables \( T_\lambda \) are geometric, independent random variables. Hence, we acquire a sequence of random variables \( T_1, \ldots, T_{\Lambda(n) - 1} \), since \( |P - F| = \Lambda(n) \) and according to our amnesiac algorithm \( |Z(r)| \leq |Z(r + 1)| + 1 \) for any round \( r \geq \tilde{r} \).

Let us define the random variable \( T \) as \( T \equiv \sum_{\lambda=1}^{\Lambda(n) - 1} T_\lambda \). \( T \) is the total number of rounds required for all processors in \( P - F \) to learn about triple \( z \). By Markov’s inequality we have:

\[
\Pr(\kappa T > \kappa \eta) = \Pr(e^{\kappa T} > e^{\kappa \eta}) \leq \frac{\mathbb{E}[e^{\kappa T}]}{e^{\kappa \eta}},
\]

for some \( \kappa > 0 \) and \( \eta > 0 \) to be specified later in the proof.

We say that “a transmission in round \( r > \tilde{r} \) is successful” if processor \( j \in Z(r) \) sends a message to some processor \( l \in (P - F) - Z(r) \); otherwise we say that “the transmission is unsuccessful.” Let \( p_j \) be the probability that the transmission is successful in a round, and \( q_j = 1 - p_j \) be the probability that it is unsuccessful. Note that if a transmission is unsuccessful then this means that in that round none of the processors in \( Z(r) \), where \( |Z(r)| = \lambda \), were able to contact a processor in \( (P - F) - Z(r) \) (here \( |(P - F) - Z(r)| = \Lambda(n) - \lambda \)), and hence we have:

\[
q_j = (1 - \frac{\Lambda(n) - \lambda}{n})^\lambda
\]
By geometric distribution, we have the following:

$$\mathbb{E}[e^{\kappa T_\lambda}] = p_j e^\kappa + p_j e^{2\kappa} q_j + p_j e^{3\kappa} q_j^2 + \ldots = p_j e^\kappa (1 + q_j e^\kappa + q_j^2 e^{2\kappa} + \ldots)$$

In order to sum the infinite geometric series, we need to have $q_j e^\kappa < 1$. Assume that $q_j e^\kappa < 1$ (note that we will need to choose $\kappa$ such that the inequality is satisfied), hence using infinite geometric series we have:

$$\mathbb{E}[e^{\kappa T_\lambda}] = \frac{p_j e^\kappa}{1 - q_j e^\kappa}$$

In the remainder of the proof we focus on deriving a tight bound on the $\mathbb{E}[e^{\kappa T_\lambda}]$, and subsequently apply Boole’s inequality across all triples in $\mathcal{V}$.

Remember that we assumed that $e^\kappa q_j = e^\kappa (1 - \frac{\Lambda(n) - \lambda}{n}) < 1$, for $j \in Z(r)$, $\lambda = 1, 2, \cdots, \Lambda(n) - 1$ and $\kappa > 0$. Let $\kappa$ be such that $e^\kappa = 1 + \frac{\Lambda(n)}{2n}$, then we have the following

$$e^\kappa (1 - \frac{\Lambda(n) - \lambda}{n})^\lambda = (1 + \frac{\Lambda(n)}{2n})(1 - \frac{\Lambda(n) - \lambda}{n})^\lambda$$

$$= 1 + \frac{\Lambda(n)}{2n} - \frac{(\Lambda(n) - \lambda)\lambda}{n} - O\left(\frac{1}{n^2}\right)$$

$$= 1 - \frac{\lambda(\Lambda(n) - \lambda) - \frac{1}{2}\Lambda(n)}{n} - O\left(\frac{1}{n^2}\right)$$

In order to show that $e^\kappa (1 - \frac{\Lambda(n) - \lambda}{n})^\lambda < 1$ it remains to show that $\lambda(\Lambda(n) - \lambda) - \frac{1}{2}\Lambda(n)$ is positive. Note that $\lambda(\Lambda(n) - \lambda)$ is increasing until $\lambda \leq \frac{\Lambda(n)}{2}$, we should also note that we
consider cases for \( \lambda = 1, 2, \ldots, \Lambda(n) - 1 \). Hence, the minimal value of \( \lambda(\Lambda(n) - \lambda) \) will be when either \( \lambda = 1 \), or \( \lambda = \Lambda(n) - 1 \) and in both cases \( \lambda(\Lambda(n) - \lambda) - \frac{1}{2} \Lambda(n) \geq 0 \), for a sufficiently large \( n \).

Let us now evaluate the following expression:

\[
n(1 - e^\kappa(1 - \frac{\Lambda(n) - \lambda}{n})^\lambda)
\]

\[
= n \left( 1 - \left( 1 - \frac{\lambda(\Lambda(n) - \lambda) - \frac{1}{2} \Lambda(n)}{n} - O\left( \frac{1}{n^2} \right) \right) \right)
\]

\[
= n \left( \frac{\lambda(\Lambda(n) - \lambda)}{n} - \frac{1}{2} \Lambda(n) + O\left( \frac{1}{n} \right) \right)
\]

Then, we have

\[
\prod_{\lambda=1}^{\Lambda(n)-1} \frac{\lambda(\Lambda(n) - \lambda)}{n(1 - e^\kappa(1 - \frac{\Lambda(n) - \lambda}{n})^\lambda)}
\]

\[
\leq \prod_{\lambda=1}^{\Lambda(n)-1} \frac{\lambda(\Lambda(n) - \lambda)}{\lambda(\Lambda(n) - \lambda) - \frac{1}{2} \Lambda(n) + O\left( \frac{1}{n} \right)}
\]

\[
\leq \prod_{\lambda=1}^{\Lambda(n)-1} \left( 1 - \frac{\Lambda(n)}{2\lambda(\Lambda(n) - \lambda)} \right)^{-1}
\]

\[
\leq \prod_{\lambda=1}^{\Lambda(n)-1} \left( \frac{1}{2} \right)^{-1} \leq 2^{\Lambda(n)}
\]

The latest is true because \( \lambda(\Lambda(n) - \lambda) \) achieves its minimal value when \( \lambda = 1 \). Now, since \( \Lambda(n) \geq a \log^c n \) we have:

\[
\Pr(\kappa T > \kappa \eta) \leq \left( \frac{e^{\kappa(\log^c n - 1)}}{\kappa^\eta} \right) 2^{\log^c n}
\]

Since \( e^\kappa = 1 + \frac{a \log^c n}{2n} \) then by taking natural base logarithm of both sides and using Taylor series for \( \ln(1 + x) \), where \( |x| < 1 \), we have \( \kappa \leq \frac{a \log^c n}{2n} \). And hence, \( e^{\kappa(\log^c n - 1)} = O(1) \).

And we get,

\[
\Pr(\kappa T > \kappa \eta) \leq \left( \frac{2^{\log^c n}}{\kappa^\eta} \right)
\]
By taking \( \eta = kn \), where \( k > 2 \) is a sufficiently large constant, we get

\[
\Pr(\kappa T > \kappa \eta) \leq \frac{2^{\alpha \log c n}}{\alpha \log n} \leq \frac{1}{n^\alpha}
\]

where \( \alpha > 1 \) for some sufficiently large constant \( k > 2 \).

Thus we showed that if a new triple is generated by a worker in \( P - F \) then \( \text{whp} \) it is known to all processors in \( P - F \) in \( O(n) \) rounds. Now by applying Boole’s inequality we want to show that \( \text{whp} \) in \( O(n) \) rounds all triples in \( \mathcal{V} \) become known to all processors in \( P - F \).

Let \( \mathcal{E}_z \) be the event that some triple \( z \in \mathcal{V} \) is not spread around among all workers in \( P - F \). In the preceding part of the proof we have shown that \( \Pr[\mathcal{E}_z] \leq \frac{1}{n^\alpha} \), where \( \alpha > 1 \). By Boole’s inequality, the probability that there exists one triple that did not get spread to all workers in \( P - F \), can be bounded as

\[
\Pr[\bigcup_{z \in \mathcal{V}} \mathcal{E}_z] \leq \sum_{z \in \mathcal{V}} \Pr[\mathcal{E}_z] = \Theta(n \log c n) \frac{1}{n^\alpha} \leq \frac{1}{n^\beta}
\]

where \( \beta > 0 \). This implies that every worker in \( P - F \) collects all \( \Theta(n \log n) \) triples generated by processors in \( P - F \) \( \text{whp} \). Thus, at least one worker in \( P - F \) becomes enlightened after \( O(n) \) rounds.

The following theorem shows that, with high probability, during epoch \( \epsilon \) the correct results for all \( n \) tasks, are available at all live processors in \( O(n) \) rounds.

**Theorem 4.3.13** Algorithm DAKS solves the network supercomputing problem for \( t = n \) under the failure model \( \langle F_\ell, E_\frac{1}{2} \rangle \) in epoch \( \epsilon \) after \( O(n) \) rounds, \( \text{whp} \).

**Proof sketch.** The proof of this theorem is similar to the proof of Theorem 4.3.8. This is because, by Lemma 4.3.11, in \( O(n) \) rounds the processors in \( P - F \) generate \( \Theta(\log c n) \) triples, where \( c \geq 1 \) is a constant. According to Lemma 4.3.12 in \( O(n) \) rounds of algorithm DAKS at
least one processor in $P - F$ becomes enlightened. While according to Lemma 4.3.2 every live worker terminates, and hence is enlightened, in additional $O(\log n)$ rounds.

According to Lemma 4.3.12, after $O(n)$ rounds of epoch $c$ at least one processor in $P - F$ becomes enlightened. Furthermore, once a processor in $P - F$ becomes enlightened, according to Lemma 4.3.2 after additional $O(\log n)$ rounds every live processor terminates whp. It follows that the time complexity of algorithm DAKS in model $\langle F_{pl}, \mathcal{E}_1 \rangle$ is $O(n)$. Next we assess work and message complexities (using the approach in the proof of Theorem 4.3.9). Recall that we may choose arbitrary $a$, such that $0 < a < 1$

**Theorem 4.3.14** Algorithm DAKS solves the network supercomputing problem under the failure model $\langle F_{pl}, \mathcal{E}_1 \rangle$ with work and message complexity $O(n^{1+a})$, for any $0 < a < 1$.

**Proof.** To obtain the result we combine the costs associated with epoch $b'$ with the costs of epoch $c$. As reasoned earlier, the worst case costs for epoch $b'$ are given in Theorem 4.3.9.

For epoch $c$ (if it is not empty), where $|P - F| = \Omega(\log^c n)$, algorithm DAKS terminates after $O(n)$ rounds whp and there are up to $O(n^a)$ live processors, thus its work is $O(n) \cdot O(n^a) = O(n^{1+a})$. In every round, if a processor is a worker it sends a share message to one randomly chosen processor. If a processor is enlightened then it sends profess messages to a randomly selected subset of processors. In every round $O(n^a)$ share messages are sent. Since whp algorithm DAKS terminates in $O(n)$ rounds, $O(n^{1+a})$ share messages are sent. On the other hand, according to Lemma 4.3.1, if during an execution $\Theta(n \log n)$ profess messages are sent then every processor terminates whp. Hence, the message complexity is $O(n^{1+a})$. 
The worst case costs of the algorithm correspond to executions with a non-empty epoch \( c \), where the algorithm does not terminate early. In this case the costs from epoch \( b' \) are asymptotically absorbed into the worst case costs of epoch \( c \) computed above. \( \square \)

Last, we extend our analysis to assess the efficiency of algorithm \( \text{DAKS}_t; n \) for \( t \) tasks, where \( t \geq n \). This is done based on the definition of algorithm \( \text{DAKS}_t; n \) using the same observations as done in discussing Theorem 4.3.10.

**Theorem 4.3.15** Algorithm \( \text{DAKS}_t; n \), with \( t \geq n \), solves the network supercomputing problem under the failure model \( \langle F_{pt}, E_2 \rangle \) whp, with time complexity \( O(t) \), work complexity \( O(t \cdot n^a) \), and message complexity \( O(n^{1+a}) \), for any \( 0 < a < 1 \).

**Proof.** The result for algorithm \( \text{DAKS}_t; n \) is obtained (as in Theorem 4.3.10) by combining the costs from epoch \( b' \) (ibid.) with the costs of epoch \( c \) derived from the analysis of algorithm \( \text{DAKS} \) for \( t = n \) (Theorem 4.3.14). This is done by multiplying the time (number of rounds) and work complexities by the size of the chunk \( \Theta(t/n) \); the message complexity is unchanged. \( \square \)

**Observation 4.3.16** We note that it should be possible to derive tighter bounds on the complexity of the algorithm. This is because we only assume for epoch \( c \) that the number of live processors is bounded by the generous range \( b \log c n \leq |P - F_r| \leq b' n^a \). In particular, if in epoch \( c \) there are \( \Theta(poly \log n) \) live processors, the work complexity becomes \( O(t \cdot poly \log n) \) and the message complexities becomes \( O(n \cdot poly \log n) \) as follows from the arguments along the lines of the proofs of Theorems 4.3.14 and 4.3.15.
4.3.3 Finalizing Algorithm Parameterization

Lastly, we discuss the compile-time constants \( C \) and \( K \) that appear in algorithm DAKS (starting with line 2). Recall that we have already given the constant \( C \) in Section 4.3.2; the constant stems from the proof of Lemma 4.3.1.

We compute \( K \) as \( \max\{k_1, k_2, k_3\} \), where \( k_2 \) and \( k_3 \) come from the proofs of Lemmas 4.3.4 and 4.3.11. The constant \( k_1 \), as we detail below, emerges from the proof of Lemma 4.2.1 of Section 4.2.2 in the same way that the constants \( k_2 \) and \( k_3 \) are established in Lemmas 4.3.4 and 4.3.11.

As we discussed in conjunction with Lemma 4.3.3, algorithm DAKS in epoch \( a \) performs tasks in the same pattern as in algorithm NS of Section 4.2.1 when \( \Omega(n) \) processors do not crash. Lemma 4.2.1 of Section 4.2.2 shows that after \( \Theta(\log n) \) rounds of algorithm NS there is no task that is performed less than \( k(1 - \delta) \log n \) times, \( \text{whp} \), for a suitably large constant \( k \) and some constant \( \delta \in (0, 1) \). Thus, we let \( k_1 \) to be \( k_1 = k(1 - \delta) \). This allows us to define \( K \) to be \( K = \max\{k_1, k_2, k_3\} \), ensuring that the constant \( K \) in algorithm DAKS (and thus in algorithm DAKS\( _{t,n} \)) is large enough to satisfy all requirements of the analysis.

4.4 Estimating Reliability of Workers

For the problem of using network supercomputing to perform a large collection of independent tasks, in Sections 4.2 and 4.3 we introduced a decentralized approach and provided randomized synchronous algorithms that perform all tasks correctly with high probability, while dealing with misbehaving and crash-prone processors. The main weaknesses of those algorithms is that they assume that the average probability of a non-crashed processor returning
incorrect results is inferior to $\frac{1}{2}$. A stronger adversarial model may allow the average probability of a live worker to return incorrect results to become greater than $\frac{1}{2}$, provided that there exists a subset $H \subseteq P - F$, such that $|H| = h|P - F|$ and $\frac{1}{|H|}\sum_{i \in H} p_i < \frac{1}{2} - \zeta$, for constants $h \in (0, 1)$ and $\zeta > 0$. Here the average probability of worker misbehavior can be greater than $\frac{1}{2}$ for processors in $P - F$, and hence, any algorithm solving the network supercomputing problem in the latter setting should be able to select a subset of processors whose results it can use for calculating the final results for the tasks in $T$. One way to do this it to assume that the probabilities $p_i, \ i \in P$ are known, however this is a strong assumption. A better approach is for each processor to estimate $p_i, \ i \in P$, where the estimation can be integrated into the algorithm solving the network supercomputing problem.

In this section we present a randomized synchronous distributed algorithm that tightly estimates the probability of each processor returning correct results. As before, we start with the set $P$ of $n$ processors, and we let $F$ be the set of processors that crash. We consider an oblivious adversary that assigns an arbitrary constant probability $p_i > 0$ of returning a correct result for each processor $i \in P$. Our algorithm estimates the probability $p_i$ of returning a correct result for each processor $i \in P - F$, making the estimates available to all these processors. The estimation is based on the $(\varepsilon, \delta)$-approximation, where each estimated probability $\tilde{p}_i$ of $p_i$ obeys the bound $\Pr[p_i(1 - \varepsilon) \leq \tilde{p}_i \leq p_i(1 + \varepsilon)] > 1 - \delta$, for some $\delta > 0$ and constant $\varepsilon > 0$ chosen by the user. The algorithm presented in this section has a termination strategy that differs from the one described in Section 4.3. However, similar to algorithm DAKS, each processor terminates without any global coordination. We assess the efficiency of the algorithm in adversarial models $\langle F_{\ell f}, E_1 \rangle$, $\langle F_{fp}, E_1 \rangle$, and $\langle F_{p\ell}, E_1 \rangle$ that bound the number of processors the adversary is allowed to crash.
We show that for model $\langle F_{\ell f}, E_1 \rangle$ our algorithm computes an $(\varepsilon, \delta)$-approximation of $p_i$, for $i \in P$ with the time complexity $T = \Theta(\log n)$, work complexity $W = \Theta(n \log n)$, and message complexity $M = \Theta(n \log^2 n)$. For model $\langle F_{fp}, E_1 \rangle$ we have $T = O(n^{1-a} \log n \log \log n)$, $W = O(n \log n \log \log n)$, and $M = O(n \log^2 n)$. Lastly, for model $\langle F_{p\ell}, E_1 \rangle$ we have $T = O(n)$, $W = O(n^{1+a})$, and $M = O(n^{1+a})$. All bounds are shown to hold with high probability.

### 4.4.1 Description of Algorithm $A_{est}$

We present our decentralized algorithm $A_{est}$ that uses a gossip-based approach to share information. The algorithm is structured in terms of the main loop that iterates through three stages: QUERY, RESPONSE, and GOSSIP. Each stage consists of three steps, Send, Receive, and Compute, that are executed synchronously by the processors. In the QUERY stage each processor sends, receives, and performs test tasks. During the RESPONSE stage the processor replies with the results for the test tasks, if any, and collects such results sent by other processors. If enough information is gathered, the processor becomes “enlightened.” In the GOSSIP stage each processor gossips the collected results to one other processor, except that enlightened processors “profess” their results to an exponentially growing random sets of processors. The processors then update their local knowledge based on the received messages, and, if sufficient information was propagated, compute the estimates for the probabilities $p_i$ and halt. The pseudocode for algorithm $A_{est}$ is given in Figure 10; the algorithm uses subroutine Estimation() to compute the probabilities, given in Figure 11. We next describe the algorithm in greater detail.

**Inputs.** Each processor $i$ receives as inputs the number of processors $n$, the estimation parameters $\varepsilon$ and $\delta$, and the set of test tasks $TT_i$ from its environment.
procedure for processor $i$:

input $n, r /* n is number of processors */
$\epsilon, \delta, \gamma /* \epsilon > 0 and \delta > 0$ are estimation parameters */
$TT, \mathcal{R} /*$ the set of test tasks for $i$ */
output Estimate([1..n]) init $/*$ array of estimates of $p_j$ for each $j \in P$ */
$R_i([1..n])$ init $/*$ set of collected result indicators (res, src, rnd) */
int $r$ init 0 /* round number */
bool $\ell$ init false /* specifies the number of profess messages to be sent per iteration */
bool $\text{enlightened}$ init false /* indicates whether the processor is "enlightened" */

while true do

QUERY STAGE
Send:

1: Let $q$ be a randomly selected processor id from $P$
2: Let $t$ be a randomly selected task from $TT_i$
3: Send $(t, i)$ to processor $q$

Receive:

4: Let $M = \{m : m = (\text{task, id})\}$ be the set of received messages
Compute:

5: if $|M| > \lceil \log n \rceil$ then
6: $M \leftarrow$ random selection of $\lceil \log n \rceil$ elements from $M$
7: Let $V = \{(\text{val, id}) : m \in M \wedge \text{val} = \text{result of } m.\text{task} \wedge \text{id} = m.\text{id}\}$

RESPONSE STAGE
Send:

8: for each $w \in V$ do
9: Send $(w.\text{val})$ to $w.\text{id}$

Receive:

10: if message $(\text{val})$ is received from $q$ chosen in QUERY STAGE then
11: if $\text{val}$ is the correct result for task $t$ chosen in QUERY STAGE then
12: $R_i[q] \leftarrow R_i[q] \cup (1, t, r)$ /* test task was computed correctly */
else
14: $R_i[q] \leftarrow R_i[q] \cup (0, i, r)$ /* test task computed incorrectly */
else /* no response from processor $q$ */
15: $R_i[q] \leftarrow R_i[q] \cup (-1, i, r)$ /* 1 is used to record a crash */
Compute:

17: if $\forall j \in P : \sum_{x \in R_i[j]} I_{\{1\}}(x.\text{res}) \geq \Gamma_1$ /* sufficient no. of correct results */
then $\ell$ /* or $j$ crashed */
enlightened $\leftarrow$ true /* processor becomes enlightened */
19:

GOSSIP STAGE
Send:

20: if enlightened then /* gossip aggressively */
21: Let $D$ be a set of $2^\ell - \log n$ processor ids randomly selected from $P$
22: Send $(\text{profess}, R_i[\ell, \ell, i])$ to processors in $D$
23: $\ell \leftarrow \ell + 1$
24: else
25: Let $q$ be a randomly selected processor id from $P$
26: Send $(\text{share}, R_i[\ell, \ell, i])$ to processor $q$

Receive:

27: Let $M = \{m : m = (\text{type, } R, \ell, i, d)\}$ be the set of received messages
28: if $\exists m \in M : m.\text{type} = \text{profess}$ then
29: enlightened $\leftarrow$ true /* processor becomes enlightened */
30: if $\exists m \in M : (\ell, i) < (m.\ell, m.\text{id})$ then
31: $\ell \leftarrow 0$
Compute:

32: for each $j \in P$ do
33: $R_i[j] \leftarrow R_i[j] \cup \bigcup_{m \in M} m.R[j]$
34: if $\exists m \in M : m.\ell \geq \lceil \log n \rceil$ then
35: Estimate($R_i[\ell]$, Estimate([1..n])) /* store estimates in Estimate([1..n]) */
36: halt
37: $r \leftarrow r + 1$

Figure 10: Algorithm $A_{est}$ at processor $i \in P$. 139
subroutine Estimation([R[1..n], Estimate[1..n]])
1: Let $\Gamma = (4\lambda \log (\frac{2}{\delta}))/\varepsilon^2$ and let $\Gamma_1 = 1 + (1 + \varepsilon)\Gamma$
2: for each $j \in P$ do
3: if $\exists (res, src, rnd) \in R[j] : res = -1$ then
4: Estimate[$j$] $\leftarrow -1$
5: else
6: Let $S$ be the list of tuples $(res, src, rnd)$ in $R[j]$, sorted by the round number $rnd$ in ascending order
7: Let $N$ be s.t. $\sum_{k=1}^{N} S[k].res < \Gamma_1 \leq \sum_{k=1}^{N+1} S[k].res$
8: Estimate[$j$] $\leftarrow \Gamma_1/N$

Figure 11: Estimation of the probabilities for each $j \in P$.

Output. Each processor $i$ outputs the estimates of probabilities $p_j$ for each $j \in P$ in array $Estimate_i[1..n]$. If a crash of processor $j$ is detected, $Estimate_i[j]$ is set to $-1$.

Local knowledge and state variables. Every processor $i$ maintains the following:

- Array $R_i[1..n]$ stores the results of test tasks, where element $R_i[j]$ is a set of results of test tasks done by processor $j$. Each $R_i[j]$ is a set of tuples $(v, s, r)$ representing the correctness of the result $v$ ($v \in \{0, 1, -1\}$) computed by processor $j$ on behalf of processor $s$, in round $r$. (This ensures that the results computed by processor $j$ in different rounds $r$ and for different processors $s$ are included.) The value $v = 0$ means that the result was computed incorrectly, $v = 1$ means that it was computed correctly, and $v = -1$ means that processor $j$ has not returned a result, hence, per our model assumption, it either crashed or received more than $\lceil \log n \rceil$ computation requests (later we argue that if processor $j$ has not returned a result then whp it crashed).

- $r$ is the round (iteration) number that is used to timestamp the computed results.

- $\ell$ controls the number of messages multicast by enlightened processors: the multicast is sent to $2^{\ell-1} \log n$ destinations. The value of $\ell$ is also used to “prioritize” processors, where higher values of $\ell$ correspond to higher priority, with ties broken by the processor
identifiers. That is, given two distinct processors $i$ and $j$ we say that processor $j$ has higher priority than $i$ if $(\ell_i, i) \prec (\ell_j, j)$, where $\prec$ is a lexicographic comparison. i.e., $(\ell_i, i) \prec (\ell_j, j)$ if and only if either (i) $\ell_i < \ell_j$, or (ii) $\ell_i = \ell_j$ and $i < j$.

- enlightened is a boolean that determines whether the processor has enough information to start “professing” its knowledge by means of aggressive gossip.

**Control flow.** We refer to each iteration of the main while-loop as the round. The loop is synchronous, but each processor exits the loop based on its local state, thus the loop may not terminate simultaneously; to model this we let the loop iterate forever and include an explicit halt for each processor $i$. Next we detail each of the three stages within a round. Recall that each stage is comprised of three steps.

**QUERY stage:**

- **Send step:** Processor $i \in P$ selects at random a target processor $q \in P$ and a task $t \in TT_i$ and sends the request containing task $t$ to $q$.

- **Receive step:** The processor receives the requested tasks sent to it in the preceding step (if any).

- **Compute step:** If the number of tasks requested is less than $\lceil \log n \rceil$, the processor computes all the tasks received. Otherwise, it randomly selects $\lceil \log n \rceil$ tasks and computes the results for the selected tasks. The results are stored in a temporary set variable $V$ where each element is a pair $(\text{val}; \text{id})$, where val is the result of the task computed by processor $i$ as requested by processor id. (We show that although the algorithm performs at most $\lceil \log n \rceil$ tasks, whp every processor receives less than $\lceil \log n \rceil$ computation requests in one step.)

**RESPONSE stage:**
Send step: Based on data in $V$, processor $i$ sends results of tasks to the respective requesters.

Receive step: Processor $i$ receives the result in a message $m$ (if any) from processor $q$ that it selected in the QUERY stage. If the result for the test task is correct then processor $i$ adds $\langle 1, i, r \rangle$ to $R_i[q]$, otherwise it adds $\langle 0, i, r \rangle$. If, however, it does not receive a message from $q$ then it adds $\langle -1, i, r \rangle$ to $R_i[q]$, where $-1$ indicates that processor $q$ crashed.

Compute step: Processor $i$ uses the values in $R_i[\ ]$ to check whether it gathered a certain number of results (in the analysis we will show that this is sufficient for computing the $(\varepsilon, \delta)$-approximation). If so, the processor becomes enlightened. This is done with the help of the function call $I_{(1)}(x.res)$ in line 17. The function $I_A : \mathbb{N} \rightarrow \{0, 1\}$ is the indicator function, such that $I_A(x)$, for any set $A \subseteq \mathbb{N}$, returns value 1 if $x \in A$ and 0 otherwise (this is also used in the analysis).

GOSSIP stage:

Send step: If processor $i$ is enlightened, it aggressively gossips its knowledge by professing it to an exponentially growing random set of processors. The size of the set is governed by the exponent $\ell$ that is incremented in each round. Otherwise the processor shares its knowledge with one randomly chosen processor.

Receive step: Processor $i$ receives messages. If it receives a profess message, it also becomes enlightened. Additionally, if a profess message is received from a processor with a higher priority (as determined by the lexicographic comparison in line 30) the processor sets $\ell$ to 0.

Compute step: Processor $i$ updates its knowledge in $R_i[\ ]$ by including the information gathered from the received messages. If processor $i$ receives a message $m$ such that $m.\ell \geq$
\[\lceil \log n \rceil\], then it calls the Estimation() procedure to compute the needed probability estimates and halts. Otherwise processor \(i\) increments \(r\) and moves to the next round.

**Estimation() subroutine:** The subroutine, given in Figure 11, calculates an estimate \(\hat{p}_j\) of probability \(p_j\) for every processor \(j \in P\) and stores the result in \(\text{Estimate}[j]\). For a processor \(j\) whose crash is detected (due to the lack of a response), we set \(\text{Estimate}[j] = -1\). In the next section we discuss the rationale behind the estimation computation and the choice of parameters \(\Gamma\) and \(\Gamma_1\). The estimate \(\hat{p}_j\) is calculated as follows. First the tuples in \(R[j]\) are sorted according to the round number, then the sum of the first \(N\) result correctness indicators (recall that 1 means correct, 0 means incorrect) is computed for the largest \(N\) such that the sum remains inferior to \(\Gamma_1\). The estimate \(\hat{p}_j\) is then computed as \(\frac{1}{N}\).

### 4.4.2 Estimation of Processor Reliability

Getting an \((\varepsilon, \delta)\)-approximation \(\hat{p}_i\) for \(p_i\), for any \(\varepsilon, \delta > 0\), where \(\Pr[p_i(1 - \varepsilon) \leq \hat{p}_i \leq p_i(1 + \varepsilon)] > 1 - \delta\), might sound like a straightforward problem solvable by collecting a sufficient number of samples and selecting the majority as the outcome. However, such a solution is programmable if we know the required number of samples \(a priori\). In fact this number will be dependent on the values of \(p_i\), \(\varepsilon\) and \(\delta\). Since the value of \(p_i\) is unknown, we want the algorithm to terminate as early as possible, once the useful computations are done, without reliance on the value of \(p_i\) as either an input or a bound. The algorithm should be able to detect if sufficient number of samples are collected on the fly to arrive at an \((\varepsilon, \delta)\)-approximation. Below we explain this with an example.

Suppose we have a random variable \(X\), where \(X \in \{0, 1\}\), such that \(\Pr[X = 0] = p\) and \(\Pr[X = 1] = 1 - p = q\). Consider the independent and identically distributed (iid)
random variables $X_1, X_2, \ldots, X_m$ whose distribution is that of $X$. Therefore, $\mathbb{E}[X] = \mathbb{E}[X_1] = \ldots = \mathbb{E}[X_m] = q$. Suppose we want to use the unbiased estimator $\frac{S_m}{m}$ of $q$, where $S_m = \sum_{i=1}^{m} X_i$. An estimator $T(X_1, X_2, \ldots, X_m)$ of a parameter $\theta$ is called unbiased estimator of $\theta$ if $\mathbb{E}_q[T(X_1, X_2, \ldots, X_m)] = \theta$ [14]. Let us choose $m = c \log n$, for some $c > 0$, in an attempt to have a reasonable number of trials.

By a simple application Chernoff bounds we can show that for $\delta > 0$

$$\Pr \left[ \frac{S_m}{m} \geq (1 + \delta)q \right] \leq e^{-\frac{mq\delta^2}{3}} \leq e^{-\frac{\delta^2 cq \log n}{3}} \leq n^{-\frac{cq\delta^2}{3}}$$

A similar relation can be shown for the case where $\Pr \left[ \frac{S_m}{m} \leq (1 - \delta)q \right] \leq n^{-\frac{cq\delta^2}{3}}$. Observe that unless we have some prior information about the value of $q$ (or $p$), other than the trivial bound $0 \leq q \leq 1$, we may not know what $c$ to choose to determine the number of repetitions for obtaining the desired accuracy for the estimation of $q$. Thus it is desirable to have an algorithm that has an online rule for stopping the computation.

Subroutine Estimation() in Figure 11 is used for calculating an $(\varepsilon, \delta)$-approximation of $p_i$ as described above. Now we elaborate on the technical aspects of $(\varepsilon, \delta)$-approximation and determine the value of $\delta$ for our analysis to hold whp. For every processor $i \in P - F$ we further bound the number of test tasks required to compute $\tilde{p}_i$.

The idea behind the subroutine Estimation() is based on the Stopping Rule Algorithm (SRA) of Dagum et al. [19]. For completeness we reproduce in Figure 12 this well-known algorithm for estimating the mean of a random variable with support in $[0, 1]$, with $(\varepsilon, \delta)$-approximation. Let $Z$ be a random variable distributed in the interval $[0, 1]$ with mean $\mu_Z$. Let $Z_1, Z_2, \ldots$ be independently and identically distributed according to $Z$ variables. We say the estimate $\tilde{\mu}_Z$ is an $(\varepsilon, \delta)$-approximation of $\mu_Z$ if $\Pr[\mu_Z(1 - \varepsilon) \leq \tilde{\mu}_Z \leq \mu_Z(1 + \varepsilon)] > 1 - \delta$. 

144
input parameters: \((\varepsilon, \delta)\) with \(0 < \varepsilon < 1, \delta > 0\)

1: Let \(\Gamma = 4\lambda \log \left( \frac{2}{3} \right)/\varepsilon^2\) \hspace{1cm} /* \(\lambda = (e - 2) \approx 0.72 \) */
2: Let \(\Gamma_1 = 1 + (1 + \varepsilon)\Gamma\)
3: initialize \(N \leftarrow 0, S \leftarrow 0\)
4: while \(S < \Gamma_1\) do \(N \leftarrow N + 1; S \leftarrow S + Z_N\)
5: output: \(\hat{\mu}_Z \leftarrow \frac{\Gamma_1}{N}\)

Figure 12: The Stopping Rule Algorithm (SRA) for estimating \(\mu_Z\).

Let us define \(\lambda = (e - 2) \approx 0.72\) and \(\Gamma = 4\lambda \log \left( \frac{2}{3} \right)/\varepsilon^2\). Now, Theorem 4.4.1 (slightly modified, from [19]) tells us that SRA provides us with an \((\varepsilon, \delta)\)-approximation with the number of trials within \(\frac{\Gamma_1}{\mu_Z}\) whp, where \(\Gamma_1 = 1 + (1 + \varepsilon)\Gamma\).

**Theorem 4.4.1 (Stopping Rule Theorem)** Let \(Z\) be a random variable in \([0, 1]\) with \(\mu_Z = \mathbb{E}[Z] > 0\). Let \(\hat{\mu}_Z\) be the estimate produced and let \(N_Z\) be the number of experiments that SRA runs with respect to \(Z\) on inputs \(\varepsilon\) and \(\delta\). Then,

\[
(i) \quad \Pr[\mu_Z(1 - \varepsilon) \leq \hat{\mu}_Z \leq \mu_Z(1 + \varepsilon)] > 1 - \delta,
\]

\[
(ii) \quad \mathbb{E}[N_Z] \leq \frac{\Gamma_1}{\mu_Z},\text{ and}
\]

\[
(iii) \quad \Pr[N_Z > (1 + \varepsilon)\frac{\Gamma_1}{\mu_Z}] \leq \frac{\delta}{2}.
\]

SRA computes an \((\varepsilon, \delta)\)-approximation with an optimal number of samplings, within a constant factor [19], thus SRA-based method provides substantial computational savings.

First, we want to show that \(\Pr[N_Z > (1 + \frac{1}{\varepsilon})c \log n] \leq \frac{1}{n^c}\) for some \(c > 0\) and \(\alpha > 0\).

Let us choose a \(\delta = \frac{2}{n^{\alpha}}\), for some \(\alpha > 0\), then for any \(\varepsilon > 0\) and \(\Gamma_1 = 1 + (1 + \varepsilon)\Gamma\) we have

\[
\Gamma = 4\lambda \log \left( \frac{2}{2^{n^\alpha}} \right)/\varepsilon^2 = 4\lambda \log (n^\alpha)/\varepsilon^2 = \frac{4\lambda \alpha \log n}{\varepsilon^2}.
\]
Also, we have \( \Gamma_1 \leq (1 + \varepsilon) \frac{4\lambda\alpha' \log n}{\varepsilon^2} \) for some \( \alpha' > \alpha \). Now, using the Stopping Rule Theorem (Theorem 4.4.1) we have

\[
\frac{1}{n^{\alpha}} \geq \mathbb{P}[N_Z > (1 + \varepsilon) \frac{\Gamma_1}{p_i}] \geq \mathbb{P}[N_Z > (1 + \varepsilon)^2 \frac{4\lambda\alpha' \log n}{p_i \varepsilon^2}] = \mathbb{P}[N_Z > (1 + \frac{1}{\varepsilon})^2 \frac{4\lambda\alpha' \log n}{p_i}] = \mathbb{P}[N_Z > (1 + \frac{1}{\varepsilon})^2 c \log n]
\]

where \( c = \frac{4\lambda\alpha'}{p_i} > 0 \), i.e. \( c = O(1) \). Since we are interested in \( \text{whp} \) guarantee, for a sufficiently large \( n \), we can suitably choose the constant \( \alpha \), such that \( \delta = \frac{1}{n^{\alpha}} \).

Our subroutine \( \text{Estimation}() \) is directly based on SRA. To estimate \( p_i \) for \( i \in P \) we need the sampling results (i.e., the results of the test tasks). We compute the \( (\varepsilon, \delta) \)-approximation by looking at the history of the results stored in the list \( S \) sorted in ascending order of the rounds to consider the results in the order they were sampled. Note that the results \( \text{Estimate}_{i}[\cdot] \) may not be the same across all processors because the samples in \( R_i[\cdot] \) may be different, however all we need is a sufficient number of results to compute an \( (\varepsilon, \delta) \)-approximation.

In our adaptation of SRA to estimate \( p_j \), in algorithm \( A_{\text{est}} \) the corresponding random variable \( Z \) takes the values \( \{0, 1\} \); 0 for incorrect results and 1 for correct results. Note that in this case we have a random variable \( Z \), where \( Z \in \{0, 1\} \), such that \( \mathbb{P}[Z=1] = p_j \) and \( \mathbb{P}[Z=0] = 1 - p_j = q_j \). Therefore, since \( \mathbb{E}[Z] = p_j \) we can estimate \( p_j \) using SRA. Based on the above derivation of a bound on \( N_Z \) from Theorem 4.4.1 we know that, for every \( p_i \), \( O(\log n) \) computations of test task results, from processor \( i \) are sufficient to compute an \( (\varepsilon, \frac{1}{n^{\alpha}}) \)-approximation of \( p_i \) by subroutine \( \text{Estimation}() \), \( \text{whp} \). The following lemma summarizes this result.
Lemma 4.4.2 In algorithm $A_{\text{est}}$, subroutine Estimation() computes an $(\varepsilon, \frac{1}{n^2})$-approximation, for some constant $\alpha > 0$, of $p_i$ for any $i \in P$, and the number of responses from each live process $i$ sufficient for the estimation is $O(\log n)$, $whp$.

4.4.3 Analysis of Algorithm $A_{\text{est}}$

In this section we analyze the performance of algorithm $A_{\text{est}}$ in failure models $\langle F_{\ell f}, E_1 \rangle$, $\langle F_{fp}, E_1 \rangle$, and $\langle F_{pl}, E_1 \rangle$. We first show that if a processor becomes enlightened then every live processor terminates quickly.

Lemma 4.4.3 In any execution of algorithm $A_{\text{est}}$, if a processor $q \in P - F$ is enlightened in round $\rho$, then after additional $\Theta(\log n)$ rounds every live processor terminates $whp$.

Proof. According to the GOSSIP stage of the algorithm if processor $q$ is enlightened then it starts sending profess messages. Without loss of generality we assume that $q$ is the processor with the highest priority among all enlightened processors. According to Compute step of GOSSIP stage (line 34 of algorithm $A_{\text{est}}$ on page 139) every processor halts once it receives a profess message $m$ from some processor such that $m.l \geq \lceil \log n \rceil$. Since processor $q$ has the highest priority, once enlightened, it does not reset its $\ell$ to 0, and hence in $\Theta(\log n)$ rounds of the algorithm processor $q$ sends $\tilde{n} = cn \log n$ profess messages, where $c \geq 1$ is a constant. Let $r$ be the round in which processor $q$ sends $\tilde{n}$ profess messages.

We want to prove that in round $r$ every processor receives a profess message from $q$ $whp$. Let us assume that there exists a processor $w$ that does not receive a profess message from processor $q$ in round $r$. We prove that $whp$ such a processor does not exist. Since $\tilde{n}$ profess messages are sent in round $r$, there were $\tilde{n}$ random selections of processors from set $P$ in
line 21 by processor \(q\); let \(i\) be the index of one such selection. Let \(X_i\) be a Bernoulli random variable such that \(X_i = 1\) if processor \(w\) was chosen by processor \(q\) and \(X_i = 0\) otherwise.

We define the random variable \(X = \sum_{i=1}^{\bar{n}} X_i\) to estimate the total number of times processor \(w\) is selected in round \(r\). In line 21 processor \(q\) chooses a destination for the \(\text{pro}f\)ess message uniformly at random, and hence \(\Pr[X_i = 1] = \frac{1}{n}\). Let \(\mu = \mathbb{E}[X] = \sum_{i=1}^{\bar{n}} X_i = \frac{1}{n}cn \log n = c \log n\), then by applying Chernoff bound, for some \(1 > \delta > 0\), we have:

\[
\Pr[X \leq (1 - \delta)\mu] \leq e^{-\frac{\mu \delta^2}{2}} \leq e^{-\frac{(c \log n) \delta^2}{2}} \leq \frac{1}{n^{\frac{\delta^2}{2}}} \leq \frac{1}{n^\beta}
\]

where \(\beta > 0\). Hence, \(\Pr[X \leq 1] \leq \frac{1}{n \log n}\). Let \(E_w\) denote the fact that processor \(w\) receives a message from processor \(q\) in round \(r\), and let \(\bar{E}_w\) be the complement of that event. By Boole’s inequality we have \(\Pr[\bigcup_w E_w] \leq \sum_w \Pr[\bar{E}_w] \leq \frac{1}{n^\gamma}\), where \(\gamma = \log n - 1 > 0\). Hence each processor \(w \in P - F\) receives at least one \(\text{pro}f\)ess message from processor \(q\) in round \(r\) whp, i.e., \(\Pr[\cap_w E_w] = \Pr[\cap_w \bar{E}_w] = 1 - \Pr[\cap_w E_w] \geq 1 - \frac{1}{n^\gamma}\). Therefore, given that in round \(r\) we have \(q.l > \lceil \log n \rceil\), every live processor terminates in \(\Theta(\log n)\) rounds of the algorithm \(\text{whp}\).

Next lemma shows that if a processor \(q \in P - F\) is enlightened, then in each subsequent round \(O(n \log n)\) \(\text{pro}f\)ess messages are sent \(\text{whp}\).

**Lemma 4.4.4** In the \(\text{Send}\) step of \(\text{GOSSIP}\) stage of algorithm \(A_{est}\) \(O(n \log n)\) \(\text{pro}f\)ess messages are sent in every round \(\text{whp}\).

**Proof.** We use induction on the round number, by showing that in every round there can be at most \(kn \log n\) messages for a sufficiently large constant such that \(k > 8\). Unless stated otherwise, hereafter by messages we mean messages of \(\text{pro}f\)ess type that are being sent in the \(\text{Send}\) step of \(\text{GOSSIP}\) stage.
The base case is the first round, say round \( t_0 \), in which some set of processors sets their enlightened variable to true. There can be at most \( n \) such processors, and according to our algorithm, after enlightened is set to true for a processor, it starts with \( \ell = 0 \) and sends \( \frac{1}{2} \log n \) profess messages, and hence, \( O(n \log n) \) profess messages are sent during round \( t_0 \). Let \( M_t \) be the set of messages sent by all processors in round \( t \). Note that in round \( t_0 \) we have \( |M_{t_0}| \equiv m_{t_0} \leq kn \log n \).

**Induction hypothesis:** In round \( t > t_0 \) we have \( m_t \leq kn \log n \).

**Induction step:** We want to show that in round \( t + 1 \) we have \( m_{t+1} \leq kn \log n \).

Consider the processors at the beginning of round \( t + 1 \). Observe that any message from a processor with a higher priority to the processor with a lower priority will reset \( \ell = 0 \) at the latter processor.

Let \( d_i \) denote the number of messages sent by the processor \( i \in P \) in Send step of GOSSIP stage of round \( t \). By the construction of the algorithm \( d_i = 2^{\ell - \left\lceil \log n \right\rceil} \) where \( \ell \) is the level of a processor and \( 0 \leq \ell \leq \left\lceil \log n \right\rceil \). Note that any two distinct processors \( i, j \in P \) can be at different levels (\( \ell_i \neq \ell_j \)). Let us assume that the processor id’s are ranked in the descending order of the \( d_i \)’s. Hereafter when we refer to the \( i \)'th processor we mean the processor with ranking \( i \), based on \( d_i \).

We define a random variable \( X^t_i \) for each processor \( i \in P \). After all messages are sent and received in round \( t \) we let \( X^t_i = 0 \) if processor \( i \) received a message from a processor \( j \) with a higher priority, and \( X^t_i = 1 \) otherwise. Let us further denote by \( p_i = \Pr(X^t_i = 1) \), note that \( p_0 = 1 \) since the processor 0 has the highest priority. Therefore, \( p_0 = 1; p_1 = (1 - \frac{1}{n})d_0; \) \( p_2 = (1 - \frac{1}{n})^{d_0+d_1}; \ldots; p_i = (1 - \frac{1}{n})^{\sum_{j=0}^{i-1} d_j}. \)
We define $X^t = \sum_{i=0}^{n-1} d_i X_i^t$ as a random variable that counts the number of messages that are sent during round $t + 1$. Clearly, $X^{t+1} \leq 2 \sum_{i=0}^{n-1} d_i X_i^t + n \log n$. The expected number of messages sent in round $t + 1$ is bounded by:

$$E[X^t] + n \log n = 2 \sum_{i=0}^{n-1} d_i E[X_i^t] + n \log n = 2 \left( d_0 + \sum_{i=1}^{n-1} d_i c^\sum_{j=0}^{i-1} d_j \right) + n \log n$$

where $c \equiv c(n) = 1 - \frac{1}{n}$. Consider the descending arrangement of $d_i$’s grouped in blocks of consecutive terms as

$$d_0, d_1 \cdots d_{k-1}, d_k \cdots d_{k-2}, \cdots d_s$$

where each group includes a maximum number of $d_i$’s such that $\sum_{i=k_j}^{k_{j+1}-1} d_i < n \log n$, with a possible exception for the last block, where $j = 0, 1, \cdots, s$, $k_0 = 0$, and $k_{s+1} = n$. We note that at the minimum the first grouping of $d_i$’s is within the constant factor of $n \log n$, otherwise the total number of messages sent is less than $kn \log n$ and the inductive step holds for round $t + 1$. Using such blocking and the fact that $c < 1$ and $d_i \geq 0$ we have

$$\sum_{i=1}^{n-1} d_i c^\sum_{j=0}^{i-1} d_j \leq$$

$$\sum_{i=1}^{k_1-1} d_i + \sum_{i=k_1}^{k_2-1} d_i c^{\sum_{j=0}^{i-1} d_j} + \cdots + \sum_{i=k_s}^{n-1} d_i c^{\sum_{j=0}^{i-1} d_j} \leq$$

$$\sum_{i=1}^{k_1-1} d_i + \sum_{i=k_1}^{k_2-1} d_i \frac{1}{n} + \cdots + \sum_{i=k_s}^{n-1} d_i \frac{1}{n^{s-2}} \leq 2n \log n$$

since $c(n)^{n \log n} \to \frac{1}{n}$ as $n \to \infty$. Therefore, we have

$$E[X^{t+1}] \leq 2E \sum_{i=0}^{n-1} d_i X_i^t + n \log n \leq 7n \log n$$
By Chernoff bound with negative dependencies for some $\delta > 0$ we have
\[
\Pr(X^{t+1} \geq (1 + \delta)\mathbb{E}[X^{t+1}]) \leq e^{-\frac{1}{2}\mathbb{E}[X^{t+1}]\delta^2} \leq e^{-\frac{1}{2}n\log n\delta^2} \leq \frac{1}{n\beta}
\]
where $\beta$ is some positive constant. \qed

To simplify the presentation we proceed by defining the *estimability* property, that tells us whether enough samples have been gathered.

**Definition 4.4.5 (Estimability)** We say that probability $p_j$ is *estimable* for $j \in P$ in round $r$ of algorithm $A_{est}$, if at the end of round $r$ we have $\sum_{x \in \bigcup_{i \in P} R_i[j]} I_{\{1\}} x.res \geq \Gamma_1$, or for some processor $i \in P - F, \exists x \in R_i[j]$ such that $x.res = -1$.

In the previous section we showed that the number of responses sufficient to estimate $p_i$ with $(\varepsilon, \frac{1}{n^\omega})$-approximation using subroutine Estimation() is $O(\log n)$. (In the sequel we let $\delta$ stand for $\frac{1}{n^\omega}$.) We next assess the number of rounds required for a processor $i \in P - F$ to become enlightened, that is the number of rounds required for $i$ either to collect sufficient responses for every processor $j \in P$ or to possess the result $-1$ from $j$, indicating that it crashed. The analysis follows along the lines of the analysis done in Sections 4.2 and 4.3; except that here we argue about random selection of processors versus tasks.

In Compute step of QUERY stage a processor does at most $\lceil \log n \rceil$ tasks. Thus, it is possible that a live processor will not respond to a request to perform a test task. In this aspect the algorithm differs from the approach in Sections 4.2 and 4.3, where if a task is selected by a live processor, then it is consequently executed. Fact 4.4.6 below (a rewording after [12]) shows that *whp* no processor receives more than $\lceil \log n \rceil$ requests in one round.

**Fact 4.4.6** If $n$ balls are uniformly randomly placed into $n$ bins with probability at least $1 - \frac{1}{n}$, for some $c > 0$, the fullest bin has $(1 + o(1)) \frac{\log n}{\log \log n}$ balls.
We now analyze our algorithm in the three adversarial models. Let $F_r$ be the set of processors crashed before round $r$.

### 4.4.3.1 Analysis of Algorithm $A_{est}$ for Failure Model $\langle F_{lf}, E_1 \rangle$

Here $|F_r|$ is bounded as in model $\langle F_{lf}, E_1 \rangle$ with at most $fn$ processor crashes for a constant $f \in (0, 1)$. Next lemma determines the number of rounds required for algorithm $A_{est}$ in model $\langle F_{lf}, E_1 \rangle$ so that whp $p_j$ is estimable for every processor $j \in P$.

**Lemma 4.4.7** In any execution of algorithm $A_{est}$ under the failure model $\langle F_{lf}, E_1 \rangle$, after $O(\log n)$ rounds $p_j$ is estimable for every processor $j \in P$, whp.

**Proof.** According to Lemma 4.4.2 the number of responses from each live processor $i$ sufficient for subroutine Estimation() to compute an $(\varepsilon, \frac{1}{\sqrt{n}})$-approximation of $p_i$ is $O(\log n)$ whp. Let $\tilde{n} = k \log n$ be the number of responses sufficient to estimate $p_i$ for any processor $i \in P$, where $k > 0$ is a sufficiently large constant. From above, and from the definition of estimability, it follows that the probability $p_j$ is estimable for a live processor $j \in P - F$ at the end of some round $r$ if processors in $P - F$ collectively possess $\tilde{n}$ results from processor $j$. On the other hand, if a processor $j$ crashes prior to the round $r$ then $p_j$ is estimable if either by round $r$ it executed at least $\tilde{n}$ tasks assigned to it by processor in $P - F$, or a processor $i \in P - F$ did not receive a response from $j$ (line 16 of algorithm $A_{est}$), after sending a task to $j$ (lines 1-3 of algorithm $A_{est}$).

We want to show that whp after $r = \kappa \tilde{n}$ rounds of algorithm $A_{est}$, where $\kappa = \frac{1}{1-j}k$ is a constant, every live processor $j \in P - F$ executes at least $\tilde{n}$ tasks assigned to it by processors in $P - F$. Conversely, based on the Fact 4.4.6, we want to show that every processor $w \in P$ is
selected by processors in $P - F$ to execute a task at least $\tilde{n}$ times by round $r$. Note that, in the latter case, by the argument provided above, it follows that $p_w$ is estimable for every processor $w \in P$, whether live or not.

Let us assume that after $r = \kappa\tilde{n}$ rounds of algorithm $A_{est}$ there exists a processor $w \in P$, such that it is selected by processors in $P - F$ to execute a task less than $(1 - \delta_1)k \log n$ times, for some $\delta_1 > 0$. We prove that whp such a processor does not exist.

According to our assumption at the end of round $r$ for some processor $w$, we have $|\bigcup_{i \in P - F} R_i[w]| < (1 - \delta_1)k \log n$. We prove that for any processor $w \in P$ whp the latter cannot happen. This is because even if $w$ crashes prior to some round $r' < r$ and a processor $i \in P - F$ assigns a task to $w$ in round $r'$ then $\langle -1, i, r' \rangle$ is added to $R_i[w]$ according to line 16 of algorithm $A_{est}$.

Let $X_i$ be a Bernoulli random variable such that $X_i = 1$ if processor $w$ was chosen to perform a task in line 1 of the algorithm by a processor in $P - F$, and $X_i = 0$ otherwise. Based on the adversarial model $\langle \mathcal{F}_f, \mathcal{E}_1 \rangle$, we know that $|P - F| \geq (1 - f)n$, where $f \in (0, 1)$.

Let us next define the random variable $X = X_1 + ... + X_{r(1 - f)n}$ to count the total number of times processor $w$ is selected by processors in $P - F$ by the end of $r$ rounds. Note that according to line 1 any processor chooses a processor from $P$ for executing a test task uniformly at random, and hence $\Pr[X_i = 1] = \frac{1}{n}$. Let $\mu = \mathbb{E}[X] = \sum_{i=1}^{r(1 - f)n} X_i = \kappa\tilde{n}(1 - f)n \cdot \frac{1}{n} = \frac{1}{1 - f}k(1 - f)n \log n = k \log n$, then by applying the Chernoff bound, for the same $\delta_1$ chosen as above, we have:

$$\Pr[X \leq (1 - \delta_1)\mu] \leq e^{-\frac{\mu\delta_1^2}{2}} \leq e^{-\frac{(k \log n)\delta_1^2}{2}} \leq \frac{1}{n^{\alpha^2}} \leq \frac{1}{n^{\alpha^2}}$$
where \( \alpha' > 1 \) for some sufficiently large \( b \). Thus, we have \( \Pr[X \leq (1 - \delta_1)k \log n] \leq \frac{1}{n^\alpha'} \) for some \( \alpha' > 1 \). Now let us denote by \( E_w \) the fact that \( |\bigcup_{i \in P-F} R_i [w]| > (1 - \delta_1)k \log n \) by the end of round \( r \), and let \( \bar{E}_w \) be the complement of that event. By Boole’s inequality we have

\[
\Pr[\bigcup_w E_w] \leq \sum_w \Pr[E_w] \leq \frac{1}{n^\beta},
\]

where \( \beta = \alpha' - 1 > 0 \). Hence each processor \( w \in P \) is the destination of at least \( (1 - \delta_1)k \log n \) test task execution requests whp, i.e.,

\[
\Pr[\bigcap_w E_w] = \Pr[\bigcup_w \bar{E}_w] = 1 - \Pr[\bigcup_w E_w] \geq 1 - \frac{1}{n^\beta}.
\]

Hence \( p_w \) is estimable whp. This completes the proof.

\[ \square \]

The proof of the next lemma is similar to the proof of Lemma 4.2.2 of Section 4.2.

**Lemma 4.4.8** In any execution of algorithm \( A_{est} \) under failure model \( (F_{\ell f}, E_1) \), if \( p_j \) is estimable in round \( \rho \) for every processor \( j \in P \) then, after additional \( O(\log n) \) rounds, at least one processor from \( P - F \) is enlightened whp.

**Proof.** Let us assume that in some round \( r \) processor \( i \in P - F \) selects some processor \( j \in P \) and assigns a test task \( t \) to it. According to algorithm \( A_{est} \) a triple \( \vartheta \equiv \langle v_j, i, r \rangle \) is added by processor \( i \) to \( R_i [j] \), where \( v_j \) is 1 if \( t \) was computed correctly by \( j \), 0 if it was computed incorrectly, and \(-1\) if processor \( j \) did not respond to \( i \). According to Fact 4.4.6 the latter means that \( j \) crashed whp. Based on Lemma 4.4.7 in \( O(\log n) \) rounds of algorithm \( A_{est} \), \( p_j \) is estimable for every processor \( j \in P \), and hence, as we argued in the proof of Lemma 4.4.7, there are \( O(\log n) \) triples generated for every processor \( i \in P \). Let \( V \) be the corresponding set of triples in the system. We want to prove that once a triple \( \vartheta \in V \) is generated in the system by a processor in \( P - F \) then whp it takes \( O(\log n) \) rounds for the rest of the processors in \( P - F \) to learn about \( \vartheta \).
In model $\langle F_\ell f, E_1 \rangle$ at most $fn$ processors may crash, where $f \in (0, 1)$. Thus, there are $\Theta(n)$ processors left in $P - F$. Hence, we can apply Lemma 4.1.4 to algorithm $A_{est}$ and we infer that in $O(\log n)$ rounds of the algorithm at least $\frac{3}{4}n$ of processors in $P - F$ become aware of triple $\vartheta$ whp. Next consider any round $d$ such that at least $\frac{3}{4}n$ of the processors in $P - F$ are aware of triple $\vartheta$ for the first time. Let us denote this subset of processors by $S_d (|S_d| \geq \frac{3}{4}n)$.

We denote by $U_d$ the remaining fraction of the processors from $P - F$ that are not aware of $\vartheta$. We are interested in the number of rounds required for every worker in $U_d$ to learn about $\vartheta$ whp by receiving a message from one of the workers in $S_d$ in some round following $d$.

We show that, by the analysis very similar to the Coupon’s Collector Problem (Definition 4.1.3), in $O(\log n)$ rounds triple $\vartheta$ is known to all processors in $P - F$ whp. Every processor in $P - F$ has a unique id, hence we can think of those processors as of different types of coupons and we assume that the processors in $S_d$ collectively represent the coupon collector.

In this case, however, we do not require that every processor in $S_d$ contacts all processors in $U_d$ whp. Instead, we require only that the processors in $S_d$ collectively contact all processors in $U_d$ whp. According to our algorithm, if no processor from $P - F$ is enlightened, in every round every processor in $P - F$ ($S_d \subset P - F$), selects a processor uniformly at random and sends all its data to it in a share message (line 26 of the algorithm). Let us denote by $m$ the collective number of trials by processors in $S_d$ to contact processors in $U_d$. According to CCP if $m = O(n \ln n)$ then whp processors in $S_d$ collectively contact every processor in $P - F$, including those in $U_d$. Since there are at least $\frac{3}{4}(1 - f)n$ processors in $S_d$ then in every round the number of trials is at least $\frac{3}{4}(1 - f)n$, hence in $O(\log n)$ rounds whp all processors in $U_d$ learn about $\vartheta$. Note, that the number of rounds may increase by a constant factor of $\frac{1}{1 - f}$ in
comparison to the case when there are no crashes, however this does not affect our asymptotic results. Therefore, in $O(\log n)$ rounds whp all processors in $U_d$ learn about $\theta$.

Thus we showed that if a new triple is generated in the system then whp it will be known to all live processors in $O(\log n)$ rounds. Now by applying Boole’s inequality we want to show that whp in $O(\log n)$ rounds all generated triples are spread among all live processors.

Since $|P| = n$ there are $O(n \log n)$ triples in $V$ by the time every processor in $P$ is estimable. Let $\mathcal{E}_\theta$ be the event that some triple $\theta \in V$ is not spread around among all live processors in $\bar{t} = k \log n$ rounds where $k > 0$ is a sufficiently large constant. In the preceding part of the proof we have shown that $\Pr[\mathcal{E}_\theta] < \frac{1}{n^\gamma}$, where $\beta > 1$. By Boole’s inequality, the probability that there exists one triple that did not get spread to all live workers, can be bounded as

$$\Pr[\bigcup_{\theta \in V} \mathcal{E}_\theta] \leq \sum_{\theta \in V} \Pr[\mathcal{E}_\theta] = O(n \log n) \frac{1}{n^\beta} \leq \frac{1}{n^\gamma}$$

where $\gamma > 0$. This implies that upon termination every live processor collects all $O(n \log n)$ triples generated in the system whp. Thus, at least one processor in $P - F$ becomes enlightened after $O(\log n)$ rounds whp.

$$\square$$

Next we assess time complexity, work complexity, and message complexity of algorithm $A_{est}$ under the failure model $\langle F_f, E_1 \rangle$.

**Theorem 4.4.9** For every processor $i \in P - F$ algorithm $A_{est}$ computes an $(\varepsilon, \delta)$-approximation of $p_i$, for the given $\delta > 0$ and $\varepsilon > 0$, under the failure model $\langle F_f, E_1 \rangle$, with time complexity $\Theta(\log n)$, work complexity $\Theta(n \log n)$, and message complexity $\Theta(n \log^2 n)$. 

156
Proof. According to Lemma 4.4.3 once a processor $q \in P - F$ is enlightened, algorithm $A_{est}$ terminates after additional $\Theta(\log n)$ rounds whp. On the other hand, according to Lemmas 4.4.7 and 4.4.8 at least one processor from $P - F$ is enlightened in $O(\log n)$ rounds of algorithm $A_{est}$. Hence, the time complexity of the algorithm is $\Theta(\log n)$. There are $\Omega(n)$ live processors in every round, and hence, the work complexity of the algorithm is $\Theta(n \log n)$.

Lastly, according to Lemma 4.4.4, once a processor is enlightened, in the Send step of the GOSSIP stage $O(n \log n)$ profess messages are sent in every round whp. Notice, that $O(n)$ messages are sent in every round if no processor is enlightened. On the other hand, according to Lemma 4.4.3, once a processor from $P - F$ is enlightened, algorithm $A_{est}$ terminates after $\Theta(\log n)$ rounds. Hence, the message complexity of the algorithm is $\Theta(n \log^2 n)$. \qed

4.4.3.2 Analysis of Algorithm $A_{est}$ for Failure Model $\langle F_{fp}, E_1 \rangle$

In model $\langle F_{fp}, E_1 \rangle$ we have $|F| \leq n - n^a$. For the purpose of analysis we divide an execution of the algorithm into two epochs: epoch a consists of all rounds $r$ where $|F_r|$ is at most linear in $n$, so that when the number of live processors is at least $c'n$ for some suitable constant $c'$; epoch b consists of all rounds $r$ starting with first round $r'$ (it can be round 1) when the number of live processors drops below some $c'n$ and becomes $c''n^a$ for some suitable constant $c''$. Note that either epoch may be empty.

For the small number of crashes in epoch a, Theorem 4.4.9 in Section 4.4.3.1 gives the worst case work as $\Theta(n \log n)$ and message complexity as $\Theta(n \log n)$; the upper bounds apply whether or not the algorithm terminates in this epoch.

Next we consider epoch b. If the algorithm terminates in round $r'$, the first round of the epoch, the cost remains the same as given by Theorem 4.4.9. If it does not terminate, it incurs
additional costs associated with the processors in $P - F_{\nu'}$, where $|P - F_{\nu'}| \leq c''n^a$. We analyze the costs for epoch $b$ in the rest of this section. The final message and work complexities will be at most the worst case complexity for epoch $a$ plus the additional costs for epoch $b$ incurred while $|P - F| = \Omega(n^a)$ per model $\langle F_{fp}, E_1 \rangle$.

**Lemma 4.4.10** In any execution of algorithm $A_{est}$ under failure model $\langle F_{fp}, E_1 \rangle$, after $O(n^{1-a} \log n)$ rounds of epoch $b$, $p_j$ is estimable for every processor $j \in P$, whp.

**Proof sketch.** The proof of the lemma is easily obtained by arguing along the lines of Lemmas 4.4.7 and 4.3.4.

**Lemma 4.4.11** In any execution of algorithm $A_{est}$ under failure model $\langle F_{fp}, E_1 \rangle$, if $p_j$ is estimable in round $\rho$ for every processor $j \in P$, then, after additional $O(n^{1-a} \log n \log \log n)$ rounds of epoch $b$, at least one processor in $P - F$ is enlightened, whp.

**Proof sketch.** The proof of this lemma is easily obtained by arguing along the lines of Lemmas 4.4.8 and 4.3.7.

**Theorem 4.4.12** For every processor $i \in P - F$ algorithm $A_{est}$ computes an $(\varepsilon, \delta)$-approximation of $p_i$, for the given $\delta > 0$ and $\varepsilon > 0$, under the failure model $\langle F_{ef}, E_1 \rangle$, with time complexity $O(n^{1-a} \log n \log \log n)$, work complexity $O(n \log n \log \log n)$, and message complexity $O(n \log^2 n)$.

**Proof.** To obtain the result we combine the costs associated with epoch $a$ with the costs of epoch $b$. The work and message complexity bounds for epoch $a$ are given by Theorem 4.4.9 and are $\Theta(n \log n)$ and $\Theta(n \log^2 n)$ respectively.
For epoch \( b \) (if it is not empty), where \( |P - F| = O(n^a) \), per Lemmas 4.4.3, 4.4.10 and 4.4.11 the algorithm terminates after \( O(n^{1-a} \log n \log \log n) \) rounds \( \text{whp} \) and there are \( \Theta(n^a) \) live processors, thus its work is \( O(n \log n \log \log n) \).

On the other hand, according to Lemma 4.4.4, once a processor is enlightened, in the Send step of the GOSSIP stage \( O(n \log n) \) profess messages are sent in every round \( \text{whp} \). Notice, that in epoch \( b \) \( O(n^a) \) messages are sent in every round if no processor is enlightened. On the other hand, according to Lemma 4.4.3, once a processor from \( P - F \) is enlightened, algorithm \( A_{est} \) terminates after \( \Theta(\log n) \) rounds. Hence, the message complexity of the algorithm is \( O(n \log^2 n) \).

The worst case costs of the algorithm correspond to the executions with non-empty epoch \( b \), where the algorithm does not terminate early. In this case the costs from epoch \( a \) are asymptotically absorbed into the worst case costs of epoch \( b \) computed above. \( \square \)

4.4.3.3 Analysis of Algorithm \( A_{est} \) for Failure Model \( (F_{p\ell}, \mathcal{E}_1) \)

In the adversarial model \( (F_{p\ell}, \mathcal{E}_1) \) we have \( |P - F| = \Omega(\log^c n) \). For executions in \( (F_{p\ell}, \mathcal{E}_1) \), let \( |P - F| \) be at least \( b \log^c n \), for specific constants \( b \) and \( c \) satisfying the model constraints. Let \( F_r \) be the actual number of crashes that occur prior to round \( r \). For the purpose of analysis we divide an execution of the algorithm into two epochs: epoch \( b' \) consists of all rounds \( r \) where \( |F_r| \) remains bounded as in model \( (F_{fp}, \mathcal{E}_1) \) (for reference, this epoch combines epoch \( a \) and epoch \( b \) from the previous section); epoch \( \epsilon \) consists of all rounds \( r \) starting with the first round \( r'' \) (it can be round 1) when the number of live processors drops below \( b' n^a \), where \( b' \) and \( a \) are specified by the failure model \( (F_{fp}, \mathcal{E}_1) \), but remains \( \Omega(\log^c n) \) per model
\( \langle F_{p\ell}, E_1 \rangle \). Observe that since we are concerned with model \( \langle F_{p\ell}, E_1 \rangle \), in the sequel we can chose any \( a \), such that \( 0 < a < 1 \). Also note that either epoch may be empty.

In epoch \( b' \) the algorithm incurs costs exactly as in model \( \langle F_{fp}, E_1 \rangle \). If algorithm \( A_{est} \) terminates in round \( r'' \), the first round of the epoch, the costs remain the same as the costs analyzed for \( \langle F_{fp}, E_1 \rangle \) in the previous section.

If it does not terminate, it incurs additional costs associated with the processors in \( P - F_{r''} \), where \( b \log c\; n \leq |P - F_{r''}| \leq b'n^a \). We analyze the costs for epoch \( \epsilon \) next. The final message and work complexities are then at most the worst case complexity for epoch \( b' \) plus the additional costs for epoch \( \epsilon \).

In the next lemmas we use the fact that \( |P - F_{r''}| = \Omega(\log c\; n) \). The first lemma shows that within some \( O(n) \) rounds in epoch \( \epsilon \) \( p_j \) is estimable for every \( j \in P \), whp.

**Lemma 4.4.13** In any execution of algorithm \( A_{est} \) under failure model \( \langle F_{p\ell}, E_1 \rangle \), after \( O(n) \) rounds of epoch \( \epsilon \), \( p_j \) is estimable for every \( j \in P \), whp.

**Proof sketch.** The proof of the lemma is easily obtained by arguing along the lines of Lemmas 4.4.7 and 4.3.11.

**Lemma 4.4.14** In any execution of algorithm \( A_{est} \) under failure model \( \langle F_{p\ell}, E_1 \rangle \), if \( p_j \) is estimable in round \( \rho \) for every \( j \in P \), then, after \( O(n) \) rounds of epoch \( \epsilon \), at least one processor in \( P - F \) is enlightened, whp.

**Proof sketch.** The proof of this lemma is easily obtained by arguing along the lines of Lemmas 4.4.8 and 4.3.12.

Next we assess time complexity, work complexity, and message complexity of algorithm \( A_{est} \) under the failure model \( \langle F_{p\ell}, E_1 \rangle \).
Theorem 4.4.15 For every processor \( i \in P - F \) algorithm \( A_{est} \) computes an \((\varepsilon, \delta)\)-approximation of \( p_i \), for the given \( \delta > 0 \) and \( \varepsilon > 0 \), under the failure model \( \langle F_{pt}, \mathcal{E}_1 \rangle \), with time complexity \( O(n) \), work and message complexities \( O(n^{1+a}) \).

Proof. To obtain the result we combine the costs associated with epoch \( b' \) with the costs of epoch \( c \). As reasoned earlier, the worst case costs for epoch \( b' \) are given in Theorem 4.4.12.

For epoch \( c \) (if it is not empty), where \( |P - F| = \Omega(\log^c n) \), per Lemmas 4.4.3, 4.4.13 and 4.4.14, algorithm \( A_{est} \) terminates after \( O(n) \) rounds whp and there are up to \( O(n^a) \) live processors, thus its work is \( O(n^{1+a}) \).

On the other hand, according to Lemma 4.4.4, once a processor is enlightened, in the Send step of the GOSSIP stage \( O(n \log n) \) profess messages are sent in every round whp. Notice, that in epoch \( c \), when no processor is enlightened, \( O(n^a) \) messages are sent in every round. On the other hand, according to Lemma 4.4.3, once a processor from \( P - F \) is enlightened, algorithm \( A_{est} \) terminates after \( \Theta(\log n) \) rounds. Hence, the message complexity of the algorithm is \( O(n^{1+a}) \), for any \( 0 < a < 1 \).

The worst case costs of the algorithm correspond to executions with a non-empty epoch \( c \), where the algorithm does not terminate early. In this case the costs from epoch \( b' \) are asymptotically absorbed into the worst case costs of epoch \( c \) computed above.

\[ \square \]

Observation 4.4.16 We note that it should be possible to derive tighter bounds on the complexity of the algorithm. This is because we only assume for epoch \( c \) that the number of live processors is bounded by the generous range \( b \log^c n \leq |P - F_r| \leq b' n^a \). In particular, if
in epoch there are $\Theta(poly \log n)$ live processors, the work and message complexities become $O(n \cdot poly \log n)$ as follows from the arguments along the lines of the proofs of Theorem 4.4.15.

4.5 Closing Remarks

In this chapter we presented algorithm NS (figure 8 on page 90) that solves the network supercomputing problem under failure models $\langle F_{\emptyset}, E_1 \rangle$ and $\langle F_{\ell f}, E_1 \rangle$, and algorithm DAKS (figure 9 on page 105) that solves the problem under failure models $\langle F_{\emptyset}, E_1 \rangle$, $\langle F_{\ell f}, E_1 \rangle$, $\langle F_{p}, E_1 \rangle$, and $\langle F_{p\ell}, E_1 \rangle$. In algorithm DAKS every processor can decide locally when the problem is solved and terminate.

The major weakness of both algorithms is that our failure models require that the average probability of returning incorrect results for non-crashed processors is inferior to $1/2$ during the execution of the algorithm. A stronger adversarial model may allow the average probability of a live worker to return incorrect results to become greater than $1/2$, provided that there exists a subset $H \subseteq P - F$, such that $|H| = h|P - F|$ and $\frac{1}{|H|} \sum_{i \in H} p_i < \frac{1}{2} - \zeta$, for constants $h \in (0, 1)$ and $\zeta > 0$. Here the average probability of worker misbehavior can be greater than $1/2$ for processors in $P - F$, and hence, any algorithm solving the network supercomputing problem in the latter setting should be able to select a subset of processors whose results it can use for calculating the final results for the tasks in $T$.

To enable the possibility of selecting a subset of trusted processors, we presented algorithm $A_{est}$ (figure 10 on page 139) that computes an $(\varepsilon, \delta)$-approximation of $p_i$, $i \in P$, for some $\delta > 0$ and a constant $\varepsilon > 0$ chosen by the user. The approximation of worker misbehavior can be integrated into an algorithm solving the network supercomputing problem, e.g., algorithm DAKS (figure 9 on page 105).
Chapter 5

Summary and Future Directions

This dissertation studies distributed cooperative computing in decentralized systems. Resource discovery problem is a necessary step enabling network supercomputing in dynamic settings in the absence of a central server, or a master, or fixed collection of workers. In particular, we studied self-stabilizing approach to this problem. We then focused on utilizing the discovered resources, and in particular, on the problem of collaboration, that we refer to as network supercomputing. We developed two synchronous decentralized algorithms that can perform a set of tasks using a distributed system of undependable, crash-prone processors. Additionally, we presented a synchronous decentralized algorithm that assesses the reliability of processors in the context of cooperative distributed computing. We now provide a summary of the contributions of this thesis and we identify future directions in this research area.

5.1 Summary

The growing popularity of the Internet computing platforms, e.g., [3, 5, 1, 2], motivated us to study the network supercomputing problem in dynamic environments, and develop fully
decentralized solutions for the problem. In the dynamic settings for network supercomputing one cannot assume a priori that the collection of cooperating computers is known. Thus the necessary first step is to discover the relevant resources in the network. The latter was originally formulated as Resource Discovery Problem (RDP) used as one of the building blocks in the global content distribution system ultimately developed by Akamai [47]. We developed two fault-tolerant and self-stabilizing algorithms for the resource discovery problem, RD-MS and RD-BS, where the time complexity (the time required for the algorithm to converge to a legitimate state) of the latter is asymptotically optimal [47]. Additionally, we proposed a formal treatment of both the problem and the algorithm, that allowed us to present our algorithms using Timed Input/Output Automata (TIOA) formalism [55]. In turn, this allows one to use Tempo [6], a comprehensive language based on TIOA, to model distributed systems with (or without) timing constraints and to reason rigorously about the correctness of algorithms and their self-stabilization properties. We believe that this approach can provide valuable tools for design and methodical study of self-stabilizing algorithms. Noteworthy, in this thesis we considered the most severe transient failures that result in corruptions of state variables that are indistinguishable from valid states.

Significant part of my thesis investigated fully decentralized solutions for network supercomputing that do not rely on the master and instead use only the cooperating failure-prone workers. We considered a variety of failure models where the workers may crash and may return incorrect results under probabilistic assumptions. Here the network supercomputing problem is to correctly perform a set of tasks using a collection of undependable workers, such that the results are known to all live processors. We developed a randomized algorithm where, depending on the number of crashes, each live processor is able to terminate dynamically with
the knowledge that the problem is solved with high probability. An attractive aspect of algorithm DAKS is its termination strategy that involves only a constant overhead in time for all processors to terminate once all tasks are performed; thus the asymptotic time complexity, and thus the algorithm’s scalability, is not impacted. We further developed an algorithm that estimates the probability of a processor to return correct results using an \((\epsilon, \delta)\)-approximation, under the failure models considered for solving the network supercomputing problem.

5.2 Future Directions

This thesis focused on the study of network supercomputing and resource discovery problems in synchronous message passing systems. However, real systems may experience message delays, and thus it is important to extend our results to asynchronous systems. In this section we present possible research directions that can utilize the results presented in this thesis as the basis for the development of efficient algorithms that work in more hostile environments. In particular, Section 5.2.1 studies future directions for the resource discovery problem, while Section 5.2.2 studies future directions for the network supercomputing problem. Finally, Section 5.2.3 studies future research directions that we came across in the course of studying the problems discussed in this thesis.

5.2.1 Resource Discovery Problem

We propose the following directions for developing fault-tolerant algorithms for the resource discovery problem in dynamic systems.

**Improving communication complexity.** As mentioned earlier, in deterministic synchronous settings Kutten et al. [61] provided an algorithm that requires \(O(\log n)\) rounds to solve RDP,
with the message complexity $O(n \log n)$, where $n$ is the number of nodes in the network. We developed a self-stabilizing algorithm that works in dynamic systems and matches the time complexity of [61], however it is still not clear whether there exists a self-stabilizing algorithm that works under the failure model defined in Section 3.1 and matches both the time and message complexities of [61]. An interesting research topic is to investigate this further by either developing an algorithm, if one exists, or by proving a lower bound for the message complexity in our failure model.

Haeupler et al. [45] and Kniesburges et al. [56] concentrated on minimizing the number of node identifiers sent in each round. To handle transient failures in our model we insist that each node sends a substantial number of messages in a given round, thus the overall message complexity of our algorithms is high. The latter is done to clean up the state from bogus identifiers. In fact, in algorithm $RD-BS$ in every round a node sends a message to every node that it knows about. Based on the bounds proved in [45, 56] it is clear that no algorithm exists for our model that converges in nearly optimal time, and in addition is optimal in terms of both message and bit communication complexities. It is then interesting to investigate the tradeoffs between the number of messages and the number of bits sent in every round, provided that we still want the algorithm to converge quickly.

**RDP in asynchronous systems.** Nor et al. [69] showed that for a self-stabilizing algorithm to converge in an asynchronous system, it is required that no bogus identifiers are present in the state of any node. An interesting open question is whether it is possible to limit the asynchrony in the hope of being able to tolerate the kind of transient failures described in our model.

Konwar et al. [58] considered the resource discovery problem in asynchronous systems, under a variety of assumptions about joins and crash failures. For the purpose of analysis
the authors restricted asynchrony. An interesting research topic is to investigate whether it is possible to design fault-tolerant algorithms in dynamic settings that can handle not only crash failures of the processors but also byzantine failures. It seems plausible that the asynchrony should be restricted to handle byzantine failures of processors in dynamic systems.

Super stabilization. Dolev and Herman [31] introduced super stabilization for the algorithms that maintain topological structures in the presence of transient failures. We conjecture, that after converging to a legitimate configuration algorithms \( RD-MS \) and \( RD-BS \), with minor modifications and without introducing the \textit{interrupt} statement of [31], can handle single joins and departures of nodes bringing the system back into a legitimate configuration in a constant number of rounds. We plan to extend our work to handle super stabilization.

Another interesting research topic is to investigate the possibility of developing a self-stabilizing algorithm that can be run on any network and can provide some guarantees that the bogus identifiers were removed from the states of all nodes. After reaching such a state another algorithm can be run that solves a specific problem. Of course, it will be useful if such an algorithm is super stabilizing, guaranteeing that the state variables of newly joined nodes will also be freed of bogus identifiers without the disruption of the main algorithm.

5.2.2 Network Supercomputing

For the problem of network supercomputing we propose further directions for improving the analysis of the algorithms proposed in Chapter 4 and developing new algorithms that work under stronger adversarial patterns and task dependency assumptions.

Deriving stronger analytical results. First, it is important to derive lower bounds for time, work, and message complexities in various models of computations considered in Section 4.1.
Second, we note that our analysis so far were limited to only models of computation described in Section 4.1. This being said, it is important to derive analysis for time, work and message complexities that depend not only on the number of processors and tasks, but also on the actual number of crash failures. Algorithm DAKS in Figure 9 solves the problem of network supercomputing correctly, even if only one processor remains operational, given that the average probability of live processors returning wrong result is inferior to \( \frac{1}{2} \). It is important for the analysis to reflect the dynamics of the algorithm.

**System model and adversity.** We suggest strengthening our adversarial model. So far we considered an oblivious adversary that decides prior to the computation what processors to crash and when to crash them. It would be interesting to study the problem under a weakly adaptive adversary, that decides what processors to crash prior to the computation, however it decides when to crash them during the computation. We conjecture, that in our model of computation handling a strongly adaptive, or online adversary might be challenging, if not impossible. In the latter case both decisions, what processors to crash and when to crash them, are made during the computation. It might be interesting to consider trade off s among the adversarial strategies, and when/if applicable to prove the impossibility results. Another approach to strengthening the adversary is allowing the processors to restart. It would be interesting to study the limitations that we will need to impose on crash-restart patterns in order to derive efficient algorithms.

Till now we only concentrated on synchronous systems of computation, we suggest extending our results to asynchronous systems. It is interesting to study the tradeoffs between synchronous, asynchronous and partially synchronous systems [33]. Additionally, we assume
that the processors can misbehave, in that they can return wrong values with a certain probability, however, the processors were not malicious. It would be interesting to study the problem in the setting where processors act in a byzantine manner [64], or to consider a game theoretic approach of [37].

An immediate extension of our work is to use the estimation technique provided in Section 4.4 and consider the problem of network supercomputing in the model where the average probability of live workers is allowed to exceed $\frac{1}{2}$. We started this line of research and developed an algorithm, where a linearly bounded adversary can crash any subset $F$ of processors, such that $|F| \leq f \cdot n$, for a constant $f$, where $0 < f < 1$, provided that each execution has a hardened set of processors $H \subseteq P - F$, with $|H| > h \cdot n$, where $0 < h < 1 - f$ and the average probability of processors in $H$ returning incorrect results is inferior to $\frac{1}{2}$. For our algorithm to work we further assumed that the probabilities $\{p_i\}, i \in P$ of workers returning incorrect results are known. It would be interesting to remove this assumption and develop an algorithm that would work for models $F_{f}, F_{f_{p}}$, and $F_{p_{l}}$. We conjecture that the complexities derived in Sections 4.2.2 and 4.3.2 will not change, except may be for the message complexity, that may increase by a logarithmic factor.

Lastly, an interesting research topic is to study the problem of network supercomputing in the setting where the tasks have dependencies. So far we only considered performing independent tasks. Many scientific computation problems involve several computational sub-steps, where some sub-steps depend on preceding sub-steps. The dependency structure of the computational steps can be represented in the form of a dependency graph. The problem then becomes to perform all tasks correctly whp, given the task dependency graph.
5.2.3 Other Future Directions

In the course of studying the problems discussed in this thesis we came across some other interesting problems, that are, either directly or indirectly, related to my work. In this section I slightly diverge from the work presented in this thesis and present various directions that are interesting research topics.

5.2.3.1 Distributed Shared Memory Systems.

So far we only considered and developed algorithms for message passing systems. However, it is also interesting to consider algorithms that use shared memory systems, especially given the common believe that it is easier to develop algorithms that use shared memory. Implementing shared storage systems that provide consistent shared memory services in dynamic distributed systems comes with its own challenges. Here we only consider shared memory systems that provide objects supporting two access operations: read that obtains the current value of the object, and write that replaces the old value of the object with a new value. Such objects are often called registers. The easiest way to implement a storage system is by having a central server, where the server accepts requests to perform some operations on the data and returns responses. However, this simple approach suffers from two major problems. First, the server becomes a performance bottleneck. Second, this approach is not fault tolerant, since the server is a single point of failure. Fault tolerance can be achieved by replicating the data on multiple servers, preferably at geographically distributed and distinct network locations. Data consistency is one of the major problems that comes with replication, since the system should be able to find the latest value of the replicated object. The notion of consistency is formalized as atomicity [62] or, equivalently, linearizability [48].
Background. A seminal work by Attiya, Bar-Noy, and Dolev [11], commonly referred to as ABD algorithm, implements atomic shared memory in the asynchronous, crash-prone, message-passing environment. This algorithm can tolerate up to $f < \frac{n}{2}$ replica node crashes, where $n$ is the total number of replicas. The ABD algorithm serves as a building block for many atomic shared memory algorithms, including in dynamic settings. To achieve an ordering of the written values, each value is associated with a timestamp. Read and write operations are similar, each is implemented in terms of two phases, a Get phase that queries replicas for information, and a Put phase that propagates information to replicas. Each phase protocol ensures that some majority participates in the communication exchange: first the messages are sent to all replica hosts, then the replies are collected until some majority of replicas responds. Recall that since $n > 2f$, a majority of non-crashed replicas always exists. Thus, each phase terminates after a single communication round, and any operation terminates after two communication rounds. The correctness of this implementation, i.e., atomicity, follows from the fact that for any pair of operations when one follows another, at least one correct replica is involved in the Put phase of the first operation and in the Get phase of the second; this ensures that the second operation will always “see” the value that is at least as recent as that of the most recent preceding operation.

Rambo is a dynamic memory service supporting multi-reader/multi-writer objects [44]; Rambo stands for Reconfigurable Atomic Memory for Basic Objects. This algorithm uses configurations, each consisting of a set of replica hosts plus a quorum system defined over these hosts, and supports reconfiguration, by which configurations can be replaced. Notably, any quorum configuration may be installed at any time, and quorums from distinct configurations are not required to have non-empty intersections. The algorithm ensures atomicity in all
executions. When there are no reconfigurations, the algorithm operates similarly to the ABD algorithm, where each of the two phases involves interaction with one complete quorum in the current configuration. New participants join the service by means of message handshakes with at least one existing participant. Any participant may crash at any time. To enable long-term operation of the service, quorum configurations can be reconfigured. Reconfigurations are performed concurrently with any ongoing read and write operations, and do not directly affect such operations. Additionally, multiple reconfigurations may be in progress concurrently. The reconfiguration is a two step process: (1) introduction of a new configuration, and (2) upgrade to the new configuration and garbage collection of obsolete configuration(s). The new configuration is selected by using a consensus algorithm (e.g., Paxos by Lamport [63]).

GeoQuorums [30] is an approach to implementing atomic shared memory on top of a physical platform that is based on mobile nodes moving in arbitrary patterns. An ad hoc network uses no preexisting infrastructure, instead, the network is formed by the mobile nodes who cooperate to route communication from sources to destinations. GeoQuorums can be viewed as a system of two layers, where the top layer implements a dynamic replicated storage system, and the bottom layer provides object replicas in terms of stationary focal points that are implemented by the mobile nodes. The focal points are geographic areas of interest that are normally “populated” by mobile nodes. Mobile nodes in the vicinity of a focal point participate in implementing a stationary virtual object, called the focal point object. The implementation at each focal point supports a local broadcast service, LBcast, that provides reliable, totally ordered broadcast. LBcast is used to implement a type of replicated state machine, one that tolerates joins and leaves of mobile nodes. If every mobile node leaves the focal point, the focal point object fails. Next, this approach defines a collection of quorum systems over the focal points.
Each quorum system involves two sets, called *get-quorums* and *put-quorums*, with the property that every get-quorum intersects every put-quorum. The use of quorums enables the service to tolerate the failure of a limited number of focal point objects. For reasons of performance, or in response to periodic migration of mobile nodes, different quorum systems can be installed. *GeoQuorums* introduced the first general reconfiguration capability that does not rely on consensus. The algorithm reconfigures among a finite number of predetermined configurations, and instead of consensus it uses a two-phase protocol. In the first phase the invoker contacts any complete get-quorum and put-quorum of all preceding configurations (note that at most one pair of messages per focal point is needed even if the finite number of possible configurations is large), then in the second phase information is conveyed to any complete put-quorum of the next configuration. *GeoQuorums* implements a modified approach to read and write operations that allows some operations to complete in just one phase. This is accomplished for the case of writes with the help of a global positioning system (GPS) clock to generate tags for the written values, thus ordering writes. This obviates the need for the phase that in other implementations determines the highest tag, and the write protocol here performs just a single put phase that interacts with any put-quorum in the current configuration. If the write detects a concurrent reconfiguration, it also awaits response from put-quorums in every configuration (this involves at most one contact with each focal point). Once the write completes, the tag becomes *confirmed*. For the case of reads, this protocol involves one or two phases. The first, Get, phase proceeds as a typical query phase to obtain the value with the maximum tag from some complete get-quorum; if a concurrent reconfiguration is detected, then the phase also awaits responses from one get-quorum from each configuration (again, at most one message exchange per focal point). Once the Get phase completes, and it is determined that the
maximum obtained tag is confirmed, the read terminates. Otherwise, the read performs the Put phase that propagates the maximum tag-value pair to some put-quorum. The information about the confirmed tags is propagated through the system to enable single-phase read operations.

_DynaStore_ [8] is an implementation of a dynamic atomic memory service for multi-writer/multi-reader objects. It integrates the ABD algorithm and allows reconfiguration of the collection of replica hosts without the use of consensus. The participants start with a default local configuration, that is, some common set of replica hosts. The algorithm supports three kinds of operations: _read_, _write_, and _reconfig_. The read and write operations involve two phases, and in the absence of reconfigurations, the protocol is similar to ABD: it uses majorities of replicas, where each replica maintains the value of the object and the associated tag. Reconfiguration is done by means of two phases and it uses a distributed _weak snapshot_ service to announce the locally-originating changes by means of the update primitive, and obtain the changes submitted by other members of the configuration by means of the _scan_ primitive. The snapshot itself is not atomic as it does not globally order updates, and because _scan_ is not guaranteed to reflect all previously completed updates. Yet, the snapshot service is sufficient for establishing a certain directed acyclic graph (DAG) that is stored locally as part of the state of each participant. Vertices of the graph correspond to configurations that can be produced by means of _changes_ that in turn correspond to the edges. The reconfiguration involves traversals of such DAGs, representing possible sequences of changed configurations. In each traversal the dag may be revised to reflect multiple changes, submitted at different hosts, to the same configuration. The assumption that a majority of the involved hosts are not removed and do not crash ensures that there is a path through the DAG that is guaranteed to be common among all
hosts. Interestingly, the hosts themselves do not learn of this common path, however, traversing all paths ensures that the common path is also traversed. The traversal terminates when a sink node is reached. An additional assumption that there is a finite number of reconfigurations ensures termination.

More detailed description on implementing distributed shared memory can be found in a recent survey [67].

**Future Work.** In the future it is interesting to investigate performance aspects of linearizable data services. We know that in general a read/write operation is required to perform two Get and Put phases. Skipping the second phase may violate linearizability. Note, however, that once Get is invoked, the value to be returned is known. It is interesting to identify the circumstances under which the value returned by Get can be used. Next, it is interesting to define a weaker notion of linearizability, e.g., by taking a subset of executions and arguing that it is indistinguishable from a linearizable execution.

In [35, 41] it was shown that under certain constraints it is possible to obtain algorithms that require a single round-trip phase for most operations and two round trips for some operations in a broad class of executions. An operation that requires only a single round-trip to complete is called a fast operation. An interesting research topic is to investigate whether in general case fast operations are sometimes possible. GeoQuorums [30] does not examine this in a global sense. For example with GeoQuorums a write operation completes in one phase with the help of GPS, while for the case of a read the protocol may involve either one or two phases. E.g., if the Get operation returns twice the same value and a timestamp, i.e. the read is confirmed, then there is no need for the read to perform a Put phase, and read terminates. On the other hand, if the reads are infrequent and writes are frequent then it is not possible to circumvent two round
trips for a read operation. This being said, if the number of reads is greater than the number of writes then it is possible that one can achieve a better performance.

It is then very important to take into account the frequency of reads and writes. We suggest developing a parametrizable algorithm that determines whether a Put phase can be skipped depending on the fraction of reads and writes. For example, if 90% of operations are read operations and only 10% are write operations then the majority of values returned are the values that were previously returned and we will be able to do single round-trip phase reads.

Resource Discovery Problem (RDP) that we studied in this thesis is directly related to implementing distributed shared memory for dynamic networks. For example, Rambo [44] requires a reconfigurations service for participating nodes. Reconfiguration requires a discovery of the recourses that are willing to participate. One might be able to show that using RDP and certain heuristics it is possible to reconfigure from a current configuration to a future configuration, and moreover it might be the case that this service comes for free.

Another interesting direction, is to see whether it is possible to develop a self-stabilizing algorithm for implementing distributed shared memory for dynamic networks, e.g., Rambo. We can first consider a simpler algorithm that works in static settings, such as ABD [11]. Recall that, for the ABD algorithm to work, it is required that a majority of the processors do not crash. Question arises whether we need to put a restriction on the number of allowed transient failures, while considering a self-stabilizing algorithm for the problem. If we do not limit the number of processors that experience a transient failure we might end up in a situation similar to having byzantine processors. This is because following a transient failure the largest timestamp can be bogus if it is allowed to have syntactically indistinguishable values of state variables. It might be necessary to assume that the corrupted values are distinguishable. To sum
up, it is interesting to examine the dynamics of self-stabilizing algorithms for implementing distributed shared memory.

As we already mentioned, Rambo uses consensus, the problem of reaching agreement among the processors in distributed settings, in order to reconfigure. It is paradoxical that if consensus does not terminate then reads and writes are fast, if however, consensus terminates quickly, then reads and writes might not terminate.

As mentioned earlier DynaStore [8] is an implementation of a dynamic atomic memory service that integrates the ABD algorithm and allows reconfiguration of the collection of replicas hosts without the use of consensus. The authors assume that the number of reconfigurations is finite. We suspect that under the assumptions made in the paper it is possible that consensus terminates. Since the algorithm proceeds by searching a sink in the DAG, the algorithm will never terminate if the number of reconfigurations is infinite. An interesting research topic is to investigate whether consensus is really required to implement dynamic data services. If we are able to show that consensus is required then the assumptions made in DynaStore [8] are very strong.

5.2.3.2 Competitive Analysis and Derandomization

It seems clear that dynamic distributed systems, where the set of tasks and/or processors changes with time, are very desirable for almost all problems considered so far. For example in decentralized network supercomputing we assume that the set of tasks does not change and it can be downloaded from some external repository prior to starting the computation. However, as in [40] it is plausible to assume that the tasks are being injected by some external entity in every round. For the resource discovery problem we assume that nodes can join
and leave the computation at any time, however to perform the analysis, we limit joins and
departures of nodes at some point of the computation. Finally, as we already discussed, the
implementation of distributed shared memory for dynamic networks requires the system of
processors to reconfigure. Moreover, to guarantee the longevity of the service, it is required
that the system participants change over time, since all of the original server participants are
bound to fail at some point.

Given this, it is interesting to consider the problems described in this thesis in dynamic
networks. To do this we need to develop so-called online algorithms. Additionally, we need to
change our traditional approach to analyzing the algorithms that we used so far, and instead, we
want to use competitive analysis. The concept of online algorithms had become very popular
since the publication of a landmark paper by Sleator and Tarjan [73], where they introduced
the notion of competitive analysis. In an online algorithm, the strategy of each participant at
each point in time depends only on the past information. In contrary, the offline algorithm
has full information about the past and the future and can compute an optimal strategy for any
problem. Performance comparison of the online algorithm with the best offline algorithm is
called competitive analysis.

Another interesting research topic is to explore the derandomization of our randomized al-
gorithms. For the problems discussed in this thesis it is interesting to explore various directions
and techniques that would enable us to derandomize our algorithms efficiently. For example
one can consider using a set of \( n \) permutations of \([n]\) with low contention [9] to eliminate
random choice from the algorithm. In [9] the authors showed that when \( n \) permutations are
chosen randomly then \( \text{whp} \) the contention is bounded by \( O(n \log n) \). However, finding the set
of permutations deterministically and efficiently is challenging. Anderson and Woll [9] also
showed how to search for this set deterministically, however that required exponential in $n$ time. Kanellakis and Shvartsman [51] proposed a method that generates permutations online and at a constant cost per each permutation element. Individual elements of each permutation are computed by processors locally and independently, making this algorithm practical. They conjectured that the resulting set of permutations has low contention, but provided no analysis. Later, Chlebus at al. [16], presented an analysis of the contention of the set of permutations proposed in [51]. They showed that the contention of the set with respect to a certain family of adversarial permutations is $O(n \log^2 n)$, where $n + 1$ is a prime. Their analytical results cover only a subset of the possible adversarial patterns of asynchrony, however, the authors did a simulation study and conjectured that the contention of a set of $n$ permutations of $[n]$, proposed in [51], should remain low for any adversarial pattern, given that $n + 1$ is prime.

Another approach is to consider combinatorial structures, such as expander graphs, sparse graphs that are highly connected. Notice, that expander graphs with desired properties can also be hard to construct. An interesting research topic is to study derandomization using special combinatorial structures, as well as the construction aspects of the structures in question, making the developed algorithms practical.
Bibliography


