A dissertation submitted to
Kent State University in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy

by

Rizal Mohd Nor

August, 2012
Dissertation written by

Rizal Mohd Nor

B.Sc., Johns Hopkins University, 2000

M.B.A., International Islamic University Malaysia, 2004

M.S., Kent State University, 2009

Ph.D., Kent State University, 2012

Approved by

Dr. Mikhail Nesterenko, Chair, Doctoral Dissertation Committee

Dr. Hassan Peyravi, Members, Doctoral Dissertation Committee

Dr. Arden Ruttan

Dr. Dmitry Ryabogin

Dr. Volodymyr Andriyevskyy

Accepted by

Dr. Javed Khan, Chair, Department of Computer Science

Dr. Timothy Moerland, Dean, College of Arts and Sciences
# TABLE OF CONTENTS

LIST OF FIGURES ............................................................... v

Acknowledgement ............................................................... vii

Dedication ............................................................................. ix

1 Introduction ................................................................. 1

1.1 Peer-to-Peer Topology .................................................. 2

1.1.1 Unstructured Peer-to-Peer Networks ....................... 4

1.1.2 Structured Peer-to-Peer Networks ......................... 4

1.1.3 Skip-Graphs ........................................................... 7

1.2 Self-Stabilization ......................................................... 12

1.3 Computation Models ................................................. 15

1.4 Communication Primitives ......................................... 16

1.5 Our Contribution ....................................................... 17

2 Literature Survey ............................................................ 19

2.1 Unstructured Peer-to-Peer Systems ............................ 19

2.2 Structured Peer-to-Peer Systems ................................. 23

2.2.1 Randomized Structured Peer-to-Peer Systems ........... 23

2.2.2 Deterministic Structured Peer-to-Peer Systems ......... 28

2.3 Fault Tolerance in Peer-to-Peer Systems ..................... 32
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overlay network exist on top of a physical network.</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>An unstructured peer-to-peer system topology example.</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Ordered chain topology.</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>Ordered ring.</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>Structured peer-to-peer system in a Chord ring topology.</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>An example of a randomized skip-list. Each node has a string of random bits</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>and a key. Every node that shares the same prefix of length $i$ would exist</td>
<td></td>
</tr>
<tr>
<td></td>
<td>on level $i$.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>An example of a degenerate case where shortcut links are arbitrarily high.</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>An adversary can insert or delete nodes to force the structure to concentrate</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>on a particular section and thereby making it inefficient.</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>A deterministic 1-2 skip-list structure.</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>A deterministic skip-graph structure. Each skip-list graph that does not</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>have a shortcut link on a level $i$ forms another skip-list graph.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>The Gnutella peer-to-peer network with ultra-peers.</td>
<td>21</td>
</tr>
<tr>
<td>12</td>
<td>An example of a Hyperring that has two bridges and creates 0 and 1 sub-rings.</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>Notice bridge 1 with subsequence 1001 and bridge 2 with 0110. The bridges</td>
<td></td>
</tr>
<tr>
<td></td>
<td>have a distance of 5 from each other [1].</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>2-dimensional Pagoda network structure [2].</td>
<td>31</td>
</tr>
</tbody>
</table>
14 Linearization component of Corona (l-Corona).

15 Example computation of l-Corona. To simplify the picture each process is represented by two nodes. The primed nodes are the process’ incoming channel. Solid lines denote identifiers stored in $l$ and $r$ of each process. Dashed lines are identifiers in the incoming channel.

16 Status decision component of skip-list part of Corona (sd-Corona).

17 Example computation of s-Corona. For simplicity, neighbor links are always assumed bidirectional.
Acknowledgement

I would like to thank all of my family, friends, and colleagues who have been a part of my life. Your encouragement and support have made it possible for me to complete graduate school and my dissertation. My advisor, Mikhail Nesterenko, has dedicated enormous energy and numerous hours in helping me through every step of my graduate school career. His invaluable advice, constant encouragement and guidance have helped me define the direction of my research. His attention to detail has helped me sharpen my ideas and improve the quality of my research papers and my dissertation. I thank Christian Scheideler, for his advice to help improve our research papers. In advance, I thank Arden Ruttan, Hassan Peyravi, Dmitry Ryabogin and Volodymyr Andriyevskyy for serving as examiners for my dissertation defense. I am grateful for the invaluable experience as a graduate student in the Computer Science department at Kent State University. The daily support from friends and colleagues has kept me focused and always drove me forward. I remember my first day as a graduate student, wandering the halls to look for Hassan Peyravi’s office, the graduate coordinator, to seek for advice. It is he who expanded my interest, pointed me to the right direction and suggested me to join Nesterenko’s research team in distributed systems. Marcy Curtiss and Roy Heath have contributed their time and effort to help dealing with numerous departmental problems. Md. Amiruzzaman has been a supportive friend and a helpful study partner in preparations of my preliminary exams, I couldn’t have made it without him. My friends in the Kent State rock climbing wall gym have
contributed towards a colorful life here in Kent. I enjoyed our rock climbing adventures as it was a form of releasing stress after a hard day of work in the lab. I would like to thank my sponsors, the Ministry of Higher Education Malaysia for sponsoring my studies here in Kent State University, as well as International Islamic University Malaysia for giving me the chance to apply for such a scholarship. Finally, I would like to end this acknowledgement to the most important people in my life. My father, who trained me to be a fighter and never to give up. My mother, who taught me the meaning of hard work and the reward it holds. My wife, who has sacrificed her time and career to be by my side. My son, for giving company during the long nights reading research material. My sisters, for the courage and love to endure hard and difficult times.

Rizal Mohd Nor,

June 22, 2012, Kent, Ohio
To my parents, who have supported me throughout my education. To my wife and son for making this journey with me.
CHAPTER 1

Introduction

In a peer-to-peer system, each computer uses its resources, such as disk storage or processing power, in the task of locating and storing information. Nowadays, peer-to-peer systems are frequently used for sharing a wide range of digital content such as files, videos, music and software applications. Unlike client-server architecture, in a peer-to-peer system there are no functional differences between computers; there are no centralized servers or a central authority to help organize peers, manage peer memberships or maintain topology. Therefore, peer-to-peer systems do not have a single point of failure or performance bottleneck. Neither do they overly depend on server administration policies or control. This distributed nature of peer-to-peer systems makes them singularly attractive means for large scale sharing and distribution of digital content.

In a peer-to-peer system, peers can potentially join and leave the network frequently. Moreover, in large network, these unexpected topology updates may leave the system in an unpredictable state. Self-stabilization is an elegant approach to optimistic failure recovery where the system is designed to arrive at the correct state regardless of the initial state. Hence, the system will recover from an arbitrary fault as soon as the influence of the fault stops. Self-stabilization is, therefore, an attractive
method of recovery from such arbitrary system states.

Asynchronous system is a classic model for studying the behavior of distributed algorithms [3]. It places no timing restrictions on process speed. Therefore, it is applicable to a wide range of problems in distributed systems. However, due to asynchrony, it is impossible to distinguish a crashed process from an extremely slow process. Message passing is an attractive computation model due to its low atomicity. Therefore, this model is applicable to the peer-to-peer architecture.

In this dissertation, we describe self-stabilizing and other fault tolerant peer-to-peer systems topology construction in the asynchronous message passing systems model.

1.1 Peer-to-Peer Topology

A network is a collection of computers, routers, switches and links for data transmission. It can be represented as a graph of nodes and edges. This graph is the topology of the network. In this dissertation, we use the term node, peer and process interchangeably. Also we use the term link and edge interchangeably. A directed graph can be strongly or weakly connected. A directed graph is strongly connected if for every node, there is a path to every other node. A directed graph is weakly connected if the undirected graph, produced by ignoring the direction of the edges of the original directed graph, is connected. In a physical network, a link between communicating entities represent the ability to communicate through direct connection to route a message from one node to another. Such connections may be a wire, an optical cable
or a wireless signal. An overlay network is a network that exists on top of another network. In an overlay network, such as a peer-to-peer network, routing and communication is handled by the underlying network such as the Internet. Hence, as long as one peer has the identifier of the other peer, the two peers can communicate through the underlying physical network. A peer-to-peer network can therefore construct the topology of the overlay network suitable for the specific application task. Figure 1 illustrates a peer-to-peer system as an overlay network.

A peer-to-peer system can be unstructured where the peers form arbitrary ad hoc topology or structured where the topology is maintained to satisfy certain desirable properties such as system efficiency and robustness.
1.1.1 Unstructured Peer-to-Peer Networks

Gnutella [4] and Freenet [5] are examples of popular unstructured peer-to-peer systems that are used for file sharing. Figure 2 illustrates the network topology of an unstructured peer-to-peer system. In unstructured peer-to-peer systems, there are no restrictions as to what identifier each node can hold. Therefore, no extensive actions are required to maintain the overlay topology.

Joining such peer-to-peer network is straightforward. To join the network, the new node needs to obtain the identifier of a peer who is already participating in the network. That is, the joining peer needs to create a single link to one of the existing peers. However, searching for a content in the peer-to-peer network can be inefficient. Since the network does not have regular structure, there are few options to limit the extend of the search. A query may have to be sent to every single node in the system. This requires the query to flood the peer-to-peer network with search query messages. This limits the scalability of unstructured peer-to-peer systems.

1.1.2 Structured Peer-to-Peer Networks

A structured overlay network orders the peers so as to enable efficient content search. This improves the scalability properties of peer-to-peer systems. Designing efficient structured peer-to-peer systems has been an active area of research which led to a number of publications [1, 2, 6–12].

To describe structured topology, let us assume that a peer-to-peer system consists of a set $N$ of $n$ processes with unique identifiers. The identifiers are related to the
Figure 2: An unstructured peer-to-peer system topology example.
content stored at each peer. Therefore, the search for content is abstracted as searching for a particular identifier. An elementary example of a structured peer-to-peer system is a chain of peers arranged in the order of their identifiers. See Figure 3 for an illustration of an ordered chain topology. Each peer with identifier \( v \) has two links. One link is to the process with identifier \( u \) such that \( (\forall x \in \mathbb{N} : x < v : x \leq u) \).

In other words, \( u \) is the highest identifier among the identifiers that are less than \( v \). Such process is called the predecessor of \( v \) or the left neighbor of \( v \). Similarly, the process \( w \) with the lowest identifier among the identifiers that is greater than \( v \) is the successor or right neighbor of \( v \). The algorithm to construct such ordered chain is \textit{topological sort} or \textit{linearization}.

After peers are topologically sorted, searching for a particular content in the network can be made more efficient. It is possible to search for an identifier by comparing the identifier of a query with the identifier of a peer. Since content is a mapping to peer identifiers, each peer will know exactly which direction the query messages should be routed. Each participating peer only need to relay query messages to its predecessor or successor.

In the effort to make search algorithms more regular, the ordered chain can be modified to create a link between the peer with the highest identifier and the peer
with lowest identifier. This will wrap the ordered chain into an ordered ring. Figure 4 illustrates the ordered ring topology. Ordered ring allows searches to always proceed in one direction, i.e. can only go to the right, which simplifies searching algorithms.

The diameter of ordered chain or ordered ring is proportional to the size of the network. This means that the number of search steps is also proportional to the network size and it grows linearly as the network size increases. This query performance may not be acceptable for large scale peer-to-peer systems. The diameter of the network and the search time can be shortened with shortcut links. A shortcut link or a finger link is a link connecting two peers that are not predecessor and successor. Proper arrangement of shortcut links can dramatically shorten the search time. In Chord [13] for example, each peer has a link to peers 2^i hops away, where i is an integer from 0 to log n. This structure reduces the network diameter and the search time to the logarithm of the number of processes in the system. Figure 5 illustrates the Chord network following a ring topology with shortcut links.

1.1.3 Skip-Graphs

A skip-list [14] is built on the basis of an ordered chain of peers with nodes maintaining shortcut links to other nodes. The shortcut links are build hierarchically by
Figure 5: Structured peer-to-peer system in a Chord ring topology.
levels. This shortcuts can be created randomly, in this case the skip-list is randomized, or deterministically, in which case the skip-list is established deterministic.

In a randomized skip-list each node holds two fields: an identifier which is called a key and a string of random bits chosen independently by each node. Nodes are ordered lexicographically according to their keys. A linked-list is constructed by taking all nodes whose prefix of length $i$ at level $i$ is the same and linking adjacent nodes in the lexicographic key order. Every node that has a shortcut link in level $i - 1$ and shares the same prefix of length $i$ would have a shortcut link on level $i$. The shortcut links of this node connects to its nearest node that are successor or predecessor at this level. Shortcut links connect the nodes progressively further away with high probability. Therefore, the diameter of the graph is logarithmic with high probability. This construction allows simple topological updates: adding or removing of a node. Indeed, the node determines the highest level to which it needs to find the successor or predecessor neighbors and connect to them. Refer to Figure 6 for an illustration of the randomized skip-list.

However, the disadvantage of a randomized skip-list is that efficiency guarantee is only probabilistic. In degenerate cases, the nodes may have shortcut links in arbitrary high levels of the hierarchy (see Figure 7) or may not have shortcut links at all. An adversary can insert or delete nodes to force the structure of the randomized skip-list to deteriorate (see Figure 8).

A deterministic skip-list is defined recursively by levels. In each level $i$ of a $k$-l
Figure 6: An example of a randomized skip-list. Each node has a string of random bits and a key. Every node that shares the same prefix of length\ i would exist on level \ i. 

Figure 7: An example of a degenerate case where shortcut links are arbitrarily high.
Figure 8: An adversary can insert or delete nodes to force the structure to concentrate on a particular section and thereby making it inefficient.

Figure 9: A deterministic 1-2 skip-list structure.

skip-list, a node $u$ has a link to node $v$ if and only if $u$ and $v$ are between $k$ and $l$ hops away at level $i - 1$. Topology updates require re-balancing of the hierarchy and creating links that would satisfy the $k$-$l$ skip-list requirements. A deterministic skip-list provides firm search guarantees and it is impervious to adversarial attacks. Refer to Figure 9 for an illustration of the deterministic skip-list.

A skip-list is not robust against crashes. A failure of a single node may disconnect the network. Similarly, a skip-list is not well suited for concurrent searches. The nodes that are high in the shortcut links hierarchy tend to be used disproportionately often, therefore they become bottlenecks. The standard measures of robustness and concurrency are expansion and congestion. Expansion measures the connectivity for
a set of connected nodes. For a given graph \( G = (V, E) \) representing a network, \( V \) is the set of nodes and \( E \) is the set of links among the nodes. The expansion of a set of nodes \( P \subset V \) is a function of the number of nodes in \( V - P \) to which \( P \) is connected. In other words, if \( N(P) \) is the set of nodes to which \( P \) is connected, then the expansion of \( P \) is \( |N(P)|/|P| \). Congestion measures the maximum number of search operations traversing a node in \( G \), such that every node has a search request to a random destination. The expansion and congestion of a skip-list are \( O(\log \frac{1}{n}) \) and \( \Omega(n) \) respectively.

A skip-graph \([15–17]\) is an extension of a skip-list with greater robustness with better expansion and congestion properties. In a skip-graph at each level the nodes at level \( i - 1 \) that do not have shortcut links at level \( i \) form an alternative level \( i \). Nodes that do not participate at that level form another alternative list at that level. This way most nodes has shortcut links at every level of the hierarchy which assures sufficient redundancy. Since a skip-graph is just a collection of \( n \) skip-lists sharing some of their lower levels, the skip-graph supports search, insert and delete operations analogous to the skip-lists. Refer to Figure 10 for an illustration of the deterministic skip-graph.

1.2 Self-Stabilization

Due to their ad hoc nature, faults and inconsistencies in peer-to-peer systems are common. A fault can be addressed by explicitly detecting it and correcting the system state. However, in large systems, the fault interaction may lead the system
Figure 10: A deterministic skip-graph structure. Each skip-list graph that does not have a shortcut link on a level $i$ forms another skip-list graph.
to an incorrect state for which the system designers did not anticipate and did not create a handler. Due to such complexities, explicit fault handling approach for large scale fault-rich systems tends to be error prone. Moreover, this approach may lead to code that is bulky, difficult to reason about and maintain. Once the fault is detected, usually the system is reset to a correct state. However, the large scale of peer-to-peer systems makes achieving a global reset infeasible.

Self-stabilization is a promising approach to failure recovery that does not explicitly require fault handlers or global resets. Starting from an arbitrary state a self-stabilizing program eventually arrives at a correct state and remains in a correct state afterward. By this definition a self-stabilizing algorithm is capable of recovery from any arbitrary fault regardless of its nature as soon as the influence of the fault stops. Moreover, a self-stabilizing program does not require initialization since it converges into a correct state regardless of its initial state. A self-stabilizing approach to fault tolerance tends to yield elegant code that is easy to implement and reason about. In this dissertation, we use the term program and algorithm interchangibly.

These attractive properties have led to extensive research in the area of self-stabilization [17–42]. Most research being done in self-stabilization concentrates on fixed topologies which algorithms cannot control. Therefore, classic self-stabilizing algorithms cannot be directly applied to peer-to-peer systems. Thus, research of self-stabilizing algorithms specifically designed for peer-to-peer systems is needed.
1.3 Computation Models

The execution speeds of processes in distributed systems may differ. This speed difference is reflected in three classical models which are used to study distributed algorithms: synchronous, partially synchronous and asynchronous.

The *synchronous model* assumes that processes execute their actions in lock step. This way the computation proceeds in synchronous rounds. In each round, processes makes local computation and exchange information with other processes. Due to this synchrony assumptions, the programs written for synchronous model are easier to design and reason about. However, this model may not adequately describe some practical distributed systems. Forcing processes to operate synchronously may be impractical if not impossible. For example, running a large scale peer-to-peer system in synchronous manner may be at best inefficient.

A *partially synchronous model* allows processes to execute at different speeds but assumes that there is a fixed maximum difference between process execution speeds. Even though this model may represent a number of realistic systems, it proves to be difficult to reason about because the timing assumption has to factor in correctness arguments and other discussions. Moreover, if the correctness of an algorithm depends on this timing assumption, the operation of the algorithm may be compromised, if the system violates this assumption, e.g. if one process slows down beyond the assumed bounds.

The *asynchronous model* imposes no timing or speed limitation on the process
execution. Processes may execute in arbitrary order and at arbitrary speed. Due to the lack of timing assumptions, the algorithms designed in this model behave correctly in any arbitrary distributed systems. These attractive properties make the asynchronous model widely used in distributed systems research.

1.4 Communication Primitives

The communication primitives used to exchange messages between processes in distributed systems vary by their degree of atomicity. An often used communication primitive is shared registers. In a shared register communication model, two neighbor processes are linked by a read-and-write register. Each process can write data to a register or read from it. While this model is convenient to reason about, the atomicity of the shared register operations is rather high: the data written by one process is instantaneously available to the other. Implementing such primitives in practical distributed systems, such as the peer-to-peer systems, is a complex task in itself. This is especially so if the system implements a fault tolerant algorithm, such as the self-stabilizing algorithm, because the communication register itself has to support the fault tolerance model of the algorithm. In particular, to support a self-stabilizing algorithm, the communication register has to be self-stabilizing.

An alternative communication model is message passing. In this model, two processes share a variable called a channel whose state is a queue of messages. One process, the sender, can write to the channel by appending a message to the tail of the channel queue. The other process, the receiver, can remove the message from
the head of the channel queue and read its content. The message passing model is significantly lower atomicity. It closely resembles a number of practical distributed systems. In particular, it is quite suitable for peer-to-peer system design.

Both shared register and message passing communication model can be used with synchronous and asynchronous execution model. Note that in case of asynchronous model, the size of the channel queue is unbounded and may grow large if the execution speed of the sender process is greater than that of the receiver.

1.5 Our Contribution

In this dissertation, we present a self-stabilizing skip-list construction algorithm in the asynchronous message passing model. We call our algorithm Corona. We demonstrate how Corona stabilizes a $1$-$2$ skip-list starting from an arbitrary state that is weakly connected. We rigorously prove Corona to be correct.

We explain topology updates in Corona. We describe the extension of Corona to a skip-graph. We extend Corona to stabilize an arbitrary $k$-$l$ skip-list. We explore the boundaries of possibilities with the help of oracles. The preliminary version of the results are stated in the following publications [43,44].

The outline of the rest of this dissertation is as follows. In Chapter 2, we survey the relevant literature in self-stabilizing algorithms, peer-to-peer systems and failure detectors. In Chapter 3, we describe Corona and present impossibility results. In Chapter 4, we describe extensions to Corona for topology updates, skip-graph and $k$-$l$ skip-list. We explore the use of oracles for linearization problem. In Chapter 5,
we conclude our dissertation with research impact and future research directions.
CHAPTER 2

Literature Survey

This dissertation combines research areas in peer-to-peer networking and self-stabilization. In this chapter, we survey the appropriate literature in each area to give the reader a perspective of our research.

2.1 Unstructured Peer-to-Peer Systems

Gnutella [4] is a unstructured peer-to-peer system where each peer is independent. Gnutella does not require a central authority to provide peers with information about other peers. A peer can join a Gnutella network to publish a file that can be retrieved by other peers within the network. A peer joins the network by a process that requires the joining peer to find one peer that is already connected to the system. The joining peer may create additional links by asking the peer that it knows for a list of other identifiers. A peer receiving this request returns a list of identifiers to which it is currently connected. Once the joining peer receive this list of identifiers, it tries to connect to each identifier in the list by sending a connection request. If this connection request is successful, the joining peer updates its list of identifiers until it reaches a fixed set of connections. Each peer maintains this list of identifiers by periodically sending a request for a list of identifiers and connection requests. If a connection request is unsuccessful, the peer may delete the failed identifier from its list. To
search for a particular data in the network, a peer sends search query messages to it neighbors. In Gnutella prior to version 0.4, each search query message is forwarded to peers connected to the network for a fixed number of hops. Due to the ad hoc nature of the network, peers join and leave the Gnutella network at will. Once a peer leaves, its neighbors’ connection list entries become incorrect. In practice, queries often fail due to these disconnections. To improve query success rate, starting from version 0.6, Gnutella separates peers into two functional groups: ultra-peers and leaves. Ultra-peers are peers with reliable connections to other ultra-peers while leaves are peers connected to one or more ultra-peers. Ultra-peers act as a backbone of the network and facilitate search propagation. See Figure 11 for an illustration of the ultra-peer backbone. In Gnutella, the peer needs to be present in the network for its data to be retrieved. Therefore, even though a search may be successful the retrieval may still fail.

Another example of an unstructured peer-to-peer system is Freenet [5]. Similar to Gnutella, a peer in Freenet forwards query messages to its neighbors. However, after publishing a file, a peer owning this file does not need to be in the network for other peers to retrieve it. A user can publish a file by inserting it into the network. Freenet stores the file by splitting the file into several parts of data and distributing it among several peers as encrypted blocks. The blocks of data are then replicated among several more peers. As long as all parts of the file are available somewhere in the Freenet network, the file can be retrieved by any peer requesting it. Since no
Figure 11: The Gnutella peer-to-peer network with ultra-peers.
single peer is responsible for the data and blocks of data are replicated among many peers, Freenet provides a service that is both reliable and anonymous. Freenet’s ability to replicate data among peers may help improve searching and retrieving data on the network, but it requires that each peer participates in network by uploading and replicating data from other peers. Since each peer is independent, it may behave selfishly: it may limit bandwidth or only download but not replicate or upload data. If a large number of peers chooses to behave selfishly, the performance of the system deteriorates.

Alternatively, in BitTorrent [45], to attain higher download throughput, a peer has to share the data it downloads. This mechanism accelerates data distribution and data retrieval. A data provider initially divides the file it wishes to share into multiple pieces of data. The provider then generates a metadata file containing the number of pieces and the digital signature for every piece. BitTorrent has three main components: a tracker, seeds and leechers. A tracker is a centralized server that is not involved in data exchanges between peers but keeps track of peers participating in a download. Seeds are peers who have downloaded the entire file and do not need to download any more pieces of data from other peers. Leechers are peers who are still in the process of downloading the pieces of data. In order to join a torrent which is a collection of peers participating in a download process, a peer usually retrieves a metadata file from a known website and extracts the IP address of a tracker. A tracker then responds by selecting a random set of seeds and leechers. A peer joining the torrent then contacts
these peers requesting different pieces of the data. Connected peers exchange bitfield messages explaining which pieces of the data they do not have. Bittorrent relies on an algorithm which gives download priority to those peers who upload data. A new leecher without any pieces to share is given a chance to connect to Bittorrent by downloading the first piece of data. Seeds give priority to leechers with the higher download rate. Thus, Bittorrent requires the peers to provide network and storage resources in exchange for data. This improves the performance of the peer-to-peer network.

Due to their unstructured topology, Gnutella, Freenet and Bittorrent require a data search algorithm that flood large portions of the network with query messages. This leads to extensive resource use that limits the efficiency of data search and diminishes the scalability of these peer-to-peer systems.

2.2 Structured Peer-to-Peer Systems

A structured peer-to-peer system allows designers to exploit the relationship that exist between peers in a particular structure. The structure can be either randomized or deterministic.

2.2.1 Randomized Structured Peer-to-Peer Systems

Chord, Pastry, Tapestry and CAN. A popular design for structured peer-to-peer systems is based on a ring topology [6,11,12]. Introduced by Stoica et at. [12,13], Chord is a well known example of such topology. Two main concepts in Chord are
keys and node identifiers. A *key* signifies the content to be searched for in the network while identifiers differentiate the nodes. Keys and node identifiers are *m*-bit strings, i.e., the keys and identifiers range from 0 to $2^m$. A key is a hash of a keyword such as a file or filename, and node identifier is a hash of the node’s IP address. If the hashing method are consistent, the keys and node identifiers are assumed to be uniformly random. Chord relies on a distributed hash table that assigns the keys to nodes. The identifiers are arranged in a sorted ring. Each node has a link to the successor and the predecessor. Each key $k$ is assigned to the node whose identifier is equal to or greater than the key, i.e., the node identifier is the successor of the key. If there are $K$ keys and $N$ nodes, then each node is responsible for approximately $K/N$ keys. A na"ive approach for key search in a ring is to sequentially follow the successor or predecessor link by comparing the key value with each node identifier. However, Chord enables a faster search algorithm by maintaining finger or shortcut links to nodes further away in the ring. Each node maintains a *finger* table containing up to $m$ entries of shortcut links where each $i^{th}$ entry in the table contains the node identifier of successor $(n + 2^{i-1}) \mod 2^m$. This allows a peer to search for a particular key in the ring network in logarithmic time with high probability.

Since Chord assigns keys randomly and assumes keys are uniformly distributed, the assignment of keys does not take geographical proximity or locality into account. This may lead to unnecessary forwarding of a search message query across geographically distributed nodes. Pastry [11] tries to mitigate this by using proximity metrics
in its routing table. Similar to Chord, Pastry is based on a distributed hash table where the keys and node identifiers are arranged to represent positions in the circular key-space. The keys are 128-bit unsigned integers and node identifiers are based on a hashing function to provide uniformly random distribution. Keys and node identifiers are ordered lexicographically. Pastry’s shortcut link structure is more sophisticated than Chord’s. Pastry’s routing table consists of a routing table proper, a neighborhood set and a leaf set. The routing table contains the node identifiers of one or potentially many nodes whose node identifier have the appropriate prefix. The neighborhood set contains node identifiers that are close to the peer on the basis of some proximity metric such as the number of IP hops, ping delays or response times. The leaf set is the set of numerically close nodes that are larger and smaller to the node’s identifier. That is, the leaf set is equivalent to the finger table in Chord. Pastry routes a message by comparing the key in the message to a peer whose node identifier is the closest to the key. If the peer is not responsible for the key, the peer examines its leaf set and routes the message closest to the key. When the key does not match a node’s identifier in the routing table, the message is sent to the closest node in the circular space through the leaf table.

Tapestry [46, 47] is based on a distributed hash table that provides decentralized object location and routing focusing on efficiency and minimizing message latency. Additionally, it allows content distribution determination according to the needs of a given application. In Tapestry, each node and each application is given an identifier by
a hashing function to assure random and uniform distribution in the overlay network. Each node identifier is responsible for one or more application identifiers and has a map to other identifiers in the network. These identifiers are used to locally construct *neighbor maps* which are optimal routing tables similar to Pastry. Each neighbor map is organized into routing levels where each level contains entries that point to a set of nodes that are close in the network. When routing a message, each application may include an optional application-specific metric in addition to a distance metric. This allows Tapestry to maintain routing tables that are optimized to a particular application.

The Content Addressable Network (CAN) [10] is a distributed and decentralized peer-to-peer system that provides a hash table functionality similar to Chord, Pastry and Tapestry. Instead of a ring structure used in Chord or Pastry, it uses a multi-torus coordinate space system to map its overlay network. The coordinate space is dynamically partitioned among all nodes in the system such that each node is responsible for a particular virtual unique zone in the overall space. A node maintains a routing table of its neighbors in this zone. In CAN, a node routes a message to a peer, by determining which neighboring zone is the closest to the peer. The node then refers to its routing table to determine the IP address of the peer that owns this zone. The message is relayed to the node with this IP address. To join the CAN network, a new node needs to find a peer that is connected to the network. This is accomplished by a bootstrapping process that returns the IP address of an existing
peer in the network. The joining peer then identifies the zone of the IP address given to it. The peer then requests for the zone to be split by sending a message to the node owning this zone. The node owning this zone splits this zone and updates its routing table. The new node then determines its neighboring zones and updates its own routing table.

**Randomized skip-graph.** An important aspect of a peer-to-peer system is its connectivity. A system with low connectivity, such as an ordered chain, can be disconnected by a single node failure. The connectivity of such systems as Chord, Pastry, Tapestry and CAN is limited. Therefore, in case of several node failures, the system may become disconnected. Moreover, in low connectivity systems, the bottleneck nodes decrease the performance of concurrent searches. To improve the efficiency of searches the shortcut links tend to span progressively larger number of nodes. However, at higher levels, fewer nodes tend to have these links. Hence, the higher levels tend to be sparser, which adversely affects the overall connectivity of the system.

Skip-graphs introduced by Aspnes and Gauri [48] mitigate these problems. They have good congestion and expansion properties. Therefore, skip-graphs are robust and amenable to concurrent searches. In a skip-graph, nodes are ordered according to their keys which are identifiers generated uniformly random. Each node in the skip-graph is a member of multiple linked-list. The lowest level, level 0 consists of all
nodes in the system. On each level \( i \), there could be many linked-lists and every node on level \( i \) participates in one of these lists. This way each node participates in every level and the overall connectivity of this system is substantial.

Harvey et al. [8] introduce a structure similar to skip-graph which they call skipnet. In skipnet, the node identifiers are also generated uniformly random. It is arranged as multiple skip-lists. Harvey et al. focus on the content and path locality properties of the skip-graph topology, and describe several extensions in building practical system.

Chord, Pastry, Tapestry, skip-graphs and skipnet depend on randomized keys that are assumed to be distributed uniformly at random. However, the ad hoc nature of peer-to-peer systems cannot guarantee that the distribution remains uniform. For example, an adversary can maliciously insert and delete keys to make the distribution non-uniform and thus cause a search to be inefficient. While CAN does not depend on a uniformly distributed keys, it does depend on a randomized algorithm that splits a virtual coordinate system into random zones. Similarly, it cannot guarantee a uniformly random distribution of zones in the virtual coordinate system. Alternatively, a deterministic approach can provide firm guarantees and deterministic bounds on content search and topology update.

2.2.2 Deterministic Structured Peer-to-Peer Systems

Deterministic data structures have firm guarantees on the efficiency of search and topology update operations. Therefore, deterministic structures do not deteriorate
through faulty or adversarial topology updates as well as the lack of high quality ran-
dom number generator. One of the first of such structures is Hyperring. Introduced
by Awerbuch and Scheideler [1], Hyperring is a concurrent data structure that can
give exact guarantees on search and update operations. It has a maximum degree
of $O(\log n)$, and require $O(\log^3 n)$ step for update operations. It can handle concur-
rent search requests to random destinations, one request per node, with congestion
of $O(\log n)$ with high probability. In Hyperring, each node has a unique identifier
such as an IP address and they are arranged in the increasing order. The nodes are
organized as a hierarchical structure of rings. There are approximately $\log n$ levels of
rings, starting with level 0. For every ring at level $i$, two rings of level $i + 1$ share its
nodes in an intertwined fashion. See Figure 12 for an illustration.

Hyperring provides exact guarantees on search and update operations, however,
randomized peer-to-peer systems still outperform the Hyperring. To achieve bet-
ter performance, Bhargava et al. [2] introduce Pagoda, a deterministic peer-to-peer
system that has a constant degree, a logarithmic diameter and a $1/\log$arithmic ex-
pansion property. Pagoda’s network is based on a complete binary tree and a family
of leveled graphs that are similar to the Omega network [49]. Figure 13 illustrates a
2-dimensional Pagoda network. It consists of the following types of edges: column
dges, tree edges, shortcut edges, and deBruijn [50] edges. The column edges connect
peers while tree edges connect levels in the deBruijn exchange network. This two
types of edges allow Pagoda to be simple and efficient. Shortcut edges are used to
Figure 12: An example of a Hyperring that has two bridges and creates 0 and 1 sub-rings. Notice bridge 1 with subsequence 1001 and bridge 2 with 0110. The bridges have a distance of 5 from each other [1].
Figure 13: 2-dimensional Pagoda network structure [2].
keep the diameter and congestion low while the deBruijn edges assist in performing efficient routing and deterministic level balancing.

Hyperring and Pagoda both provide exact guarantees on search and update operations in a deterministic peer-to-peer system. Furthermore, Pagoda is able to match the performance of the best known randomized peer-to-peer systems. Eventhough both systems are robust, they do not provide protection against systemic faults such as a large system state corruption or an extensive network failure. A more systematic approach to fault tolerance is therefore required.

2.3 Fault Tolerance in Peer-to-Peer Systems

There is extensive literature on fault tolerance in peer-to-peer systems [12,51–56]. A common approach is to use specific fault handlers for individual faults such as node misconfigurations, node failures or message losses.

Chord original designers [12] present a protocol to overcome such faults which they call Chord stabilization protocol. The main objective of this protocol is to always maintain an up-to-date pointer to successor nodes for each node in the system. The successor pointers are then used to verify and correct finger tables to allow fast and correct searches. If joining and leaving nodes affect the Chord ring before these faults can be corrected, Chord protocol tries to rectify the problem by handling each fault case individually.

Kuhn et al. [55] introduce a peer-to-peer system that could rebalance its nodes to adapt to changes in topology due to faults. Their system is based on a hypercube.
Each peer is a part of a distinct hypercube node and each hypercube node consists of \( \Theta(\log n) \) peers. Peers have connections to other peers of their hypercube node and to other peers of the neighboring hypercube nodes. A peer moves to another hypercube node to achieve equal number of peers in each hypercube at all times. The dimensions of the hypercube changes as the number of peers increases above a certain threshold. The balancing of peers among the hypercube nodes can be seen as a dynamic token distribution problem [57] on the hypercube. Each node of a hypercube distributes tokens along the edge of the graph to maintain an equal number of tokens with other nodes. The algorithm is built on two basic components: an algorithm that performs the dynamic token distribution and an information aggregation algorithm to estimate the number of peers in the system and to adapt the hypercube dimension accordingly.

Similarly, Chen and Chen [56] present a comprehensive algorithm to overcome a variety of faults. They concentrate on providing a precise specification for maintaining the pointers to other successor nodes. They call these successor nodes leafsets. Chen and Chen then provide a protocol that satisfies this specification.

Reconstructing the system topology after an extensive fault may not be possible. Therefore, another approach to fault tolerance is to use probabilistic methods that rely on message dissemination algorithms [51–54]. It is assumed that nodes in peer-to-peer systems are randomly distributed and contain a sufficient number of redundant links. Hence, sending messages to random selected nodes can help reconstruct the network topology with high probability.
Montresor et al. [53] introduce T-MAN, a gossip protocol that could build a wide range of overlay networks from scratch. The protocol is fast, robust, and simple. In T-MAN, the desired topology is a parameter in the form of a ranking method that selects neighbors and orders nodes according to a base node. T-MAN is used to show how Chord can be constructed on demand [54]. It is shown that starting from a unstructured overlay network, Chord can be constructed efficiently with logarithmic number of steps.

Similarly, Aspnes et al. [51] introduce an algorithm for fast construction of overlay networks. The algorithm has low contention and a lower bound of \( \Omega(d + \log n) \) running time in a synchronous model. The algorithm depends on three steps. First, peers are paired by a randomized pairing algorithm that joins a fraction of the nodes. Then, these pairs are merged into balanced trees. Finally, each tree acts as a supernode and they is merge to form a tree with a single root. The algorithm repeats until the topology is fully constructed.

2.4 Self-Stabilization

A self-stabilization approach to maintaining topology for peer-to-peer systems is desirable since it recovers the system regardless of the nature of the fault. Due to its ability to recover from any arbitrary initial state as soon as the fault ends. An extensive amount of research literature have been published in the subject [36,38,40,58–65].
**Linearization.** Linearization and ring construction are fundamental tasks necessary to achieve stabilization of the whole peer-to-peer system. There is a number of papers focusing on stabilizing algorithms for this task [36, 59, 60].

Shaker and Reeves [59] introduce a method to form a directed ring topology. In the ring network protocol, each node $v$ periodically initiates a search for a closer successor by sending a search message request to a random neighbor in the network. Then node $w$ receiving this request forwards the message to a neighbor that is closer to $v$. This process continues until there is no other neighbor that is closer to $v$.

Cramer and Fuhrmann [60] introduce the Iterative Successor Pointer Rewiring Protocol (ISPRP) which efficiently constructs a ring. The protocol is self-stabilizing. It is efficient that it requires only four message exchanges and message exchanges are local to the ring network.

Onus et al. [36] present a high-level atomicity solution to linearizing an overlay network that performs in polylogarithmic time. The algorithm presented by Onus et al. assumes high atomicity where each peer know the state of its neighbors.

**Self-stabilizing peer-to-peer systems of advanced topology.** Other papers study self-stabilization for tree or hypercube network structures. Hérault et al. [38] describe a self-stabilizing spanning tree algorithm. The self-stabilizing algorithm has an invariant where the identifier of a process must be lower than its parents' and greater than its children's.
Caron et al. [61] show a snap-stabilizing prefix tree for peer-to-peer systems. It is a distributed algorithm to build a Proper Greatest Proper Common Prefix (GPCP) Tree starting from any labeled rooted tree. The algorithm ensures that the system always maintain a desirable behavior and is optimal in stabilization time. The algorithm is for shared-register system model.

Banchi et al. [62] present stabilizing peer-to-peer spatial filters. They propose and prove correct a distributed stabilizing implementation of an overlay network, called DR-tree, optimized for efficient selective dissemination of information. DR-tree copes with topology updates, system faults due to memory corruption and network faults due to disconnections. The maintenance of the structure is local and requires no additional memory to guarantee its stabilization. The structure is balanced and is of height $O(\log_m N)$, which makes it suitable for performing efficient data storage and search.

Clouser et al. [17] describe a self-stabilizing skip-list called Tiara. Tiara is designed for shared register message passing model. Tiara stabilizes a special type of a skip-list called sparse 0-1 skip-list. Tiara can construct such a structures without any knowledge of global network parameters such as the number of nodes in the system. Each node utilizes only the information available to its immediate neighbors. It preserves network connectivity so long as the initial network is connected. Tiara reconstructs the connectivity of the base sorted list on the basis of skip-list links. Clouser et al. also show how the 0-1 skip-list can be extended to a skip-graph.
Jacob et al. [40] show a skip-list algorithm for a synchronous message passing model. The authors presented a distributed and self-stabilizing algorithm that constructs a skip-graph in polylogarithmic time from any initial state in which the overlay network is weakly connected. In addition, the authors show that individual joins and leaves are handled locally and require little amount of work.

Several randomized overlay network algorithms are proposed. Dolev and Kat [63] present HyperTree for peer-to-peer systems. HyperTree is a distributed structure which supports topology updates while ensuring that the out-degree and in-degree of a peer are \( b \log_b N \) where \( N \) in the maximal number of nodes and \( b \) is an integer parameter greater than 1. In addition, the HyperTree ensures that the maximal number of hops involved in each procedure is bounded by \( \log_b N \).

Jacob et. al. [64] present a variant of synchronous message-passing model for the structured skip-graph peer-to-peer system called SKIP\(^+\). The solution presented by Jacob et. al show improvements over all previous results on the number of communication rounds needed to arrive at a scalable overlay network. They also present an algorithm that provide good polylogarithmic work bound on join and leave events.
3.1 Model, Notation and Definitions

**Peer-to-peer networks.** A peer-to-peer overlay network program consists of a set $N$ of $n$ processes with unique identifiers. A process can communicate with any other peer process as long as it has a record of the peer’s identifier. The communication is by passing messages through channels.

Peer-to-peer networks often require ordering the processes in a sequence according to their identifiers. Two processes $a$ and $b$ are *consequent*, denoted $\text{cnsq}(a, b)$, if $(\forall c : c \in N : (c < a) \lor (b < c))$. That is, two consequent processes do not have an identifier between them. For the sake of completeness, we assume that $-\infty$ is consequent with the smallest id process in the system. Similarly, the largest id process is consequent with $+\infty$.

Graph terminology helps us in reasoning about peer-to-peer networks. A *link* is a pair of identifiers $(a, b)$ defined as follows: either message $\text{message}(b)$ carrying identifier $b$ is in the incoming channel of process $a$, or process $a$ stores identifier $b$ in its local memory. See Figure 15 for illustration of this. Note that a thus defined link is directed. In referring to such a directed link $(a, b)$, we always state the predecessor process $a$ first and the successor process $b$ second. The *length* of a link $(a, b)$ is the
number of processes $c$ such that $a < c < b$. Note that the length of $(a, b)$ is zero if $\text{cnsq}(a, b)$ is true. The length of $(-\infty, a)$ is zero if $a$ is the smallest id in the system, it is some integer $n$ otherwise. Similarly, the length of $(b, +\infty)$ is zero if $b$ is maximum and $n$ otherwise. The process connectivity graph $CP$ is the graph formed by the links of the identifiers stored by the processes. A channel connectivity multigraph $CC$ includes both locally stored and message-based links. Self-loop links are not considered. By this definition, $CP$ is a subgraph of $CC$. Note that besides the processes, $CC$ and $CP$ may contain two nodes $+\infty$ and $-\infty$ and the corresponding links to them. Graph $CP$ captures current network connectivity information that all the processes keep. $CC$ reflects the connectivity data that is stored implicitly in the messages in communication channels. Again, refer to Figure 15 for an example of both graph types.

**Computation model.** Each process contains a set of variables and actions. A channel $C$ is a special kind of variable whose values are sets of messages. We assume that the only information a message carries is process identifiers. We further assume that a message carries exactly one identifier. The identifiers are defined. That is, a message cannot carry $\infty$. Channel message capacity is unbounded. Messages cannot be lost. The order of message receipts does not have to match the order of transmission. That is, the channels are not FIFO. Due to this, we treat all messages sent to a particular process as belonging to a single incoming channel.
An action has the form \((\text{guard}) \rightarrow (\text{command})\). \text{guard} is either a predicate over the contents of the incoming channel or \textbf{true}. In the latter case the predicate and corresponding action are \textit{timeout}. \textit{command} is a sequence of statements assigning new values to the variables of the process or sending messages to other processes.

Program state is an assignment of a value to every variable of each process and messages to each channel. A program state may be arbitrary, the messages and process variables may contain identifiers that are not present in the network. An identifier is \textit{existing} if it is present in the network. An action is \textit{enabled} in some state if its guard is \textbf{true} in this state. It is \textit{disabled} in this state otherwise. A timeout action is always enabled. We consider programs with timeout actions, hence, in every state there is at least one enabled action.

A \textit{computation} is an infinite fair sequence of states such that for each state \(s_i\), the next state \(s_{i+1}\) is obtained by executing the command of an action that is enabled in \(s_i\). This disallows the overlap of action execution. That is, action execution is \textit{atomic}. We assume two kinds of fairness of computation: weak fairness of action execution and fair message receipt. \textit{Weak fairness} of action execution means that if an action is enabled in all but finitely many states of the computation then this action is executed infinitely often. \textit{Fair message receipt} means that if the computation contains a state where there is a message in a channel, the computation also contains a later state where this message is not present in the channel.

We focus on programs that do not manipulate the internals of process identifiers.
Specifically, a program is *compare-store-send* if the only operations that it does with process identifiers is comparing them, storing them in local process memory and sending them in a message. That is, operations on identifiers such as addition, radix computation, hashing, etc. are not used. In a compare-store-send program, if a process does not store an identifier in its local memory, the process may learn this identifier only by receiving it in a message. A compare-store-send program cannot introduce new identifiers to the network, it can only operate on the ids that are already there. If a computation of a compare-store-send program starts from a state where every identifier is existing, each state of this computation contains only existing identifiers.

A state *conforms* to a predicate if this predicate is *true* in this state; otherwise the state *violates* the predicate. By this definition, every state conforms to predicate *true* and none conforms to *false*. Let $A$ and $B$ be predicates over program states. Predicate $A$ is closed with respect to the program actions if every state of the computation that starts in a state conforming to $A$ also conforms to $A$. Predicate $A$ converges to $B$ if both $A$ and $B$ are closed and any computation starting from a state conforming to $A$ contains a state conforming to $B$.

**Problems.** The *overlay network problem* maps each set of identifiers to a set of acceptable process connectivity graphs. For example, for every set of processes, the *linearization problem* specifies exactly one graph where each process is linked with its
consequent processes.

Linearized overlay networks simplify process search. When discussing a linearized network, processes with identifiers greater than $p$ are to the right of $p$, while processes with identifiers smaller than $p$ are to the left of $p$. That is, we consider processes arranged in the increased order of identifiers from left to right. See Figure 15 for an illustration.

The process search time in a simple linearized network is proportional to its size. This may not be acceptable in large-scale networks. Shortcut links are added to accelerate navigation. In a deterministic skip-list, these links are created recursively by levels. The zero (bottom) level is the linearized list of processes. In a $k$-$l$ skip-list, a node $a$ has a link to node $b$ at level $i$ if $a$ and $b$ are between $k$ and $l$ hops away at level $i - 1$. For example, in a 1-2 skip-list, $a$ and $b$ are linked at level $i$ if they are no more than three and no less than two hops away at level $i - 1$. Refer to Figure 17 for an example of a 1-2 skip-list.

In the $k$-$l$ skip-list construction problem, a set of processes is mapped to the set of possible skip-lists. Note that in a linearization problem the set of identifiers uniquely determines the connectivity graph. In case of $k$-$l$ skip-list construction, depending on which processes participate at each level, the same list of identifiers may form several possible skip-lists. Hence, the skip-list construction problem specifies multiple acceptable $CP$ graphs for a single set of processes.

We define the two problem properties below to aid us in formally stating the
necessary conditions for the existence of a solution. An overlay network problem is *single component* if it maps every set of processes to a weakly connected process connectivity graph. Intuitively, a single component network overlay problem prohibits a program from separating the network into multiple components. The linearization and skip-list construction problem are single component.

An overlay network problem $\mathcal{PG}$ is *disconnecting* if there is at least one set of processes $S$ such that for every channel connectivity graph $CP$ to which $\mathcal{PG}$ maps $S$, there is a cut set $CS$ such that $|CS| < n - 1$ which disconnects $S$. Note that such a cut set exists for any graph except for a completely connected one. Essentially, a disconnecting network overlay problem requires that in at least one case the desired channel connectivity graph is not completely connected. Naturally, both the linearization and skip-list construction problem are disconnecting.

**Problem solutions.** A program $\mathcal{PG}$ satisfies or solves a problem $\mathcal{PR}$ from a predicate $P$ if, for every set $S$, every computation of $\mathcal{PG}$ that starts in a state conforming to $P$ contains a suffix with the following property: the channel connectivity graph $CP$ is the same in every state of this suffix and this $CP$ is one of the graphs to which $\mathcal{PR}$ maps $S$. That is, starting from the initial state in $P$, the solution has to implement at least one of the required $CP$s.

Program stabilization is *graph-identical* if every computation of a stabilizing program contains a suffix where $CC$ contains the same links as $CP$. Such program
generates $CC$ links that are already present in $CP$. If a process of such program receives a message, this message carries an identifier that the recipient process already stores and the process ignores the message.

A program is *unconditionally stabilizing* (or just *stabilizing*) if it solves the problem from $P \equiv \text{true}$. That is, every computation of a stabilizing program, regardless of the initial state, contains a correct suffix. Unconditional stabilization may be too strong for a program to possess. A program is *conditionally stabilizing* if $P \not\equiv \text{true}$. That is, such program stabilizes from a limited set $P$ of states.

We define two special cases of conditionally stabilizing programs. A program is *weakly channel-connectivity stabilizing* if it stabilizes only from the initial states where the channel-connectivity graph is weakly connected. A program is *existing identifier stabilizing* if it stabilizes only from states where every identifier is existing.

3.2 Necessary Conditions

The necessary conditions stated in this chapter show that common overlay network topology specifications prohibit the existence of unconditionally stabilizing solutions. The necessary conditions are that initially the channel connectivity graphs need to be connected and non-existing identifiers are not present.

The proofs for these conditions rely on the lemma below. Intuitively, the lemma states that for the processes to form a connected topology they have to be at least weakly connected initially.

**Lemma 1.** If a computation of a compare-store-send program starts in a state where
the channel connectivity graph $CC$ is disconnected, the graph is disconnected in every state of this computation.

Proof. Let us consider, without loss of generality, a program state where the connectivity graph consists of two components $C_1$ and $C_2$. Assume the opposite: the computation starting from this state contains states where the two components of $CC$ are connected. Let us consider the first such state $s_1$. In this state there must be two process $a \in C_1$ and $b \in C_2$ that are neighbors. Assume the link is from $a$ to $b$. That is, $(a, b) \in CC$.

Since $s_i$ is the first connected state, this link does not belong to $CC$ in the preceding state $s_{i-1}$. Since the program is compare-store-send, the new link can not appear in the process memory, it must be due to a message sent to $a$ by another process $c$ in state $s_{i-1}$. A message to $a$ carrying $b$ can only be sent by a process $c$ that has links to both $a$ and $b$ in $s_{i-1}$.

Since $(c, a) \in CC$, $c$ belongs to the same component $C_1$ as $a$ in $s_{i-1}$, and since $(c, b) \in CC$, $c$ belongs to the same component $C_2$ as $b$ in $s_{i-1}$. This means that $C_1$ and $C_2$ are weakly connected in a state $s_{i-1}$ that precedes $s_i$. However, we assumed that $s_i$ is the first state where the two components are connected. This contradiction proves the lemma.

Theorem 1. If a compare-store-send self-stabilizing program is a solution to a single-component overlay network problem, this program must be weakly channel-connectivity stabilizing.
Proof. Assume the opposite. That is, there is a self-stabilizing program $\mathcal{P}G$ that solves a single-component overlay network problem $\mathcal{P}R$ and it is not necessarily weakly channel-connectivity stabilizing.

Since $\mathcal{P}G$ is a solution to $\mathcal{P}R$, for each set $S$, every computation of $\mathcal{P}G$ contains a suffix with the prescribed $CP$. Since $\mathcal{P}G$ is not necessarily weakly channel-connectivity stabilizing, this holds true for computations starting from a state where $CC$ is disconnected. Program $\mathcal{P}G$ is a compare-store-send program. According to Lemma 1, if its computation starts from a state where $CC$ is disconnected, it is disconnected in every state of this computation. Since $CP$ is a subgraph of $CC$, it has to be disconnected in every state of this computation as well. However, $\mathcal{P}R$ is single-component. Since $\mathcal{P}R$ is single component, it maps every set of processes $S$ to a weakly connected process $CP$. This means that, contrary to our initial assumption, $\mathcal{P}R$ is not a solution to $\mathcal{P}G$. Hence the theorem.

\begin{flushright}$\square$\end{flushright}

\textbf{Theorem 2.} If a graph-identical compare-store-send program is a stabilizing solution to a single-component disconnecting overlay network problem, this program must be existing identifier stabilizing.

Proof. Assume the opposite. Let $\mathcal{P}G$ be a compare-store-send program that is a graph-identical self-stabilizing solution to a single-component disconnecting overlay network problem $\mathcal{P}R$. Since $\mathcal{P}R$ is disconnecting, there is a set of processes $S$ such that for every connectivity graph, there is a cut set that disconnects this graph.

Consider a computation $\sigma$ of $\mathcal{P}G$ with set $S$. Let $CP$ be the process connectivity
graph to which this computation converges. Let $CS$ be the cut set that separates $S$ into two subsets $S_1$ and $S_2$. Since $\mathcal{PG}$ is graph-identical, $\sigma$ contains a suffix where, in every state, $CC$ has the same links as $CP$. Let $s_1$ be the first state of this suffix.

We examine a set of processes $S_1 \cup S_2$ and construct a new state of the program for this state as follows. The state of every process in $S_1 \cup S_2$ and its incoming channel is the same as in the initial state of $\sigma$. In addition, the incoming channels of each process $a$ belonging to $S_1 \cup S_2$ in this state contain the messages that are sent to $a$ by processes of $CS$ in $\sigma$ before state $s_1$. From this new state, we execute the actions of $\mathcal{PG}$ for processes $S_1 \cup S_2$ in the same sequence as in $\sigma$. The presence of messages from processes in $CP$ allows us to do that. After this procedure we arrive at a state $s_2$. We then execute the actions of $\mathcal{PG}$ in arbitrary fair manner. Thus constructed sequence is a computation of $\mathcal{PG}$.

Note that each process of $S_1 \cup S_2$ has the same state in $s_1$ and $s_2$. Since $CS$ was a cut set of $CP$ in $s_1$, there are no links between processes of $S_1$ and $S_2$ in either $s_1$ or $s_2$. This means that $CP$ is disconnected in $s_2$. Graph $CC$ has the same links as $CP$ in $s_1$. This means that $CC$ is disconnected in $s_2$ as well. According to Lemma 1, both $CC$ and $CP$ are disconnected in every state of this computation past $s_2$.

However, $\mathcal{PG}$ is supposed to be a solution to $\mathcal{PR}$. Problem $\mathcal{PR}$ is single component. This means our constructed computation has to contain a suffix where $CP$ is weakly connected in every state. This contradiction proves the theorem. \qed
3.3 Linearization

**Problem statement.** In the linearization problem, each set of processes is mapped to the following process connectivity graph \( CP \). Each process \( p \) in \( CP \) contains exactly two outgoing links: \( p.r \) and \( p.l \). The links conform to the following predicate \( LP \):

\[
(\forall a, b \in N : a < b : \text{cnsq}(a, b) \iff ((a.r = b) \land (b.l = a)))
\]

The predicate states that two processes are neighbors if and only if they are consequent.

**l-Corona description.** Each process \( p \) maintains two variables \( r \) and \( l \) as required by the problem specification. The range of each variable are the process identifiers respectively to the left and to the right of \( p \). That is, \( r \) can only store identifiers that are greater than \( p \), while \( l \) – less than \( p \). The value of each variable may be undefined. In this case, the \( r \) variable is equal to \(+\infty\) and the \( l \) variable is equal to \(-\infty\). If non-existent identifiers are not present in the initial state of the program computation, the \( l \) variable of the smallest id process and the \( r \) variable of the largest id process are always set to \(-\infty\) and \(+\infty\) respectively.

Each process \( p \) of l-Corona contains two actions: a receive-action and a timeout action. The receive action is enabled when there is a message in the incoming channel \( p.C \). The operation of the action depends on the \( id \) carried by the message. If \( id \) is
process \(p\)

variables
\[ r, \quad \text{// right identifier, greater than } p \]
\[ l, \quad \text{// left identifier, less than } p \]

actions
\[ \text{message}(id) \in p.C \rightarrow \]
\[ \text{receive message}(id) \]
\[ \text{if } id > p \text{ then} \]
\[ \text{if } id < r \text{ then} \]
\[ \quad \text{if } r < +\infty \text{ then} \]
\[ \qquad \text{send message}(r) \text{ to } id \]
\[ \qquad r := id \]
\[ \text{else} \]
\[ \quad \text{send message}(id) \text{ to } r \]
\[ \text{if } id < p \text{ then} \]
\[ \quad \text{if } id > l \text{ then} \]
\[ \qquad \text{if } l > -\infty \text{ then} \]
\[ \qquad \text{send message}(l) \text{ to } id \]
\[ \qquad l := id \]
\[ \text{else} \]
\[ \quad \text{send message}(id) \text{ to } l \]
\[ \text{true} \rightarrow \]
\[ \text{if } r < +\infty \text{ then } \text{send message}(p) \text{ to } r \]
\[ \text{if } l > -\infty \text{ then } \text{send message}(p) \text{ to } l \]

Figure 14: Linearization component of Corona (l-Corona).

greater than \(p\), it is compared to \(r\). If \(id\) is less than \(r\), then \(p\) discovered a closer right neighbor. Process \(p\) then forwards the old right neighbor identifier to the new process and reassigns its variable \(r\). However, if the received \(id\) is no less than \(r\), then the current right neighbor of \(p\) is no further away than \(id\). In this case \(p\) sends \(id\) for process \(r\) to process. If \(r\) is not initialized, it is assigned the received \(id\). The identifier that is smaller than \(p\) is handled similarly. The timeout action sends the process identifier to its left and right neighbors. An example computation of l-Corona
Correctness proof. We prove that l-Corona is weakly-channel connected and existing identifier stabilizing to the linearization problem. Therefore, throughout this subchapter we assume that in every initial state, only existing identifiers are present and the channel connectivity graph is weakly connected.

Observe that due to the operation of the algorithm, in case \( a < b \), link \((a, b)\) can only be replaced by a link \((a, c)\) such that \( a < c < b \). Likewise, link \((b, a)\) can only be replaced by \((b, c)\) such that \( a < c < b \). That is, a link in \( CP \) can only be shortened. An example of \( CP \) link shortening is shown in Figure 15: the link \((b, d)\) is shortened to \((b, c)\) in transition from 15(a) to 15(b). Note that every process in \( CP \) contains exactly two outgoing links. One is pointing to the left, the other — to the right.

Similarly, in case \( a < b \), a link \((a, b) \in CC \setminus CP\) can be replaced only by a link \((c, b)\) such that \( a < c < b \). In the other direction, a link \((b, a) \in CC \setminus CP\) can be replaced only by a link \((c, a)\) such that \( a < c < b \). Again, the link in \( CC \) can only be shortened. For example, link \((c, a) \in CC \setminus CP\) in Figure 15 is shortened to \((b, a)\) in transition from 15(c) to 15(d). Note that unlike \( CP \), a process may contain more than two outgoing links in \( CC \setminus CP \). Furthermore, while some links are shortened, longer ones may be added by timeout actions.

Lemma 2. If a computation of l-Corona starts from a state where \( CC \) contains a path from process \( a \) to \( b \), then in every state of this computation, there is a path from
$a$ to $b$ as well.

\textit{Proof.} We show that the execution of every action of l-Corona either adds a link, retains all links, or replaces a link by a path. Therefore, none of the paths that contain these links before the action execution are disconnected by it.

Let us consider the receive-action and focus on the identifier that the message carries. The self-loops are not considered in $CC$. Therefore, the case of $id = p$ is not applicable. We will only discuss the case of $id > p$, the case of $id < p$ is similar. If $r = +\infty$, the link is retained by $p$, and $CC$ does not change.

Otherwise, this action of the program depends on the value of $r$. If $id > r$, then $p$ forwards $id$ to process $r$. That is, the link $(p, id)$ is replaced by the path $(p, r)$ and $(r, id)$ in $CC$. Now, if $id$ is between $p$ and $r$, then $p$ sends the value of $r$ to $id$ and updates the value of its right link to $id$. In other words, the link $(p, id)$ is not changed in $CC$ but link $(p, r)$ is replaced by the path $(p, id)$ and $(id, r)$. Thus, the receive-action of l-Corona does not disconnect paths in $CC$.

The case of the timeout action is straightforward as it only adds links to $CC$ and thus cannot disconnect paths in $CC$.

\begin{lemma}
If a computation of l-Corona starts in a state where for some process $a$ there are two links $(a, b) \in CP$ and $(a, c) \in CC \setminus CP$ such that $a < c < b$, then this computation contains a state where there is a link $(a, d) \in CP$ where $d \leq c$.

Similarly, if the two links $(a, b) \in CP$ and $(a, c) \in CC \setminus CP$ are such that $b < c < a$, then this computation contains a state where there is a link $(a, d) \in CP$ where
\end{lemma}
\[ d \geq c. \]

Intuitively, Lemma 3 states that if there is a link in the incoming channel of a process that is shorter than what the process already stores, then, the process’ links will eventually be shortened. The proof is by simple examination of the algorithm.

**Lemma 4.** *If a computation of l-Corona starts in a state where for some process \( a \) there is an edge \((a, b) \in CP\) and \((a, c) \in CC \setminus CP\) such that \( a < b < c \), then the computation contains a state where there is a link \((d, c) \in CP\), where \( d \leq b \).*

Similarly, if the two links \((a, b) \in CP\) and \((a, c) \in CC \setminus CP\) are such that \( c < b < a \), then this computation contains a state where there is a link \((d, c) \in CP\), where \( d \geq b \).

Intuitively, the above lemma states that if there is a longer link in the channel, it will be shortened by forwarding the id creating this link to the id’s closer successor. The proof is immediately follows from the operation of the algorithm.

**Lemma 5.** *If a computation of l-Corona starts in a state where for some processes \( a, b, \) and \( c \) such that \( a < c < b \) (or \( a > c > b \)), there are edges \((a, b) \in CP\) and \((c, a) \in CC\), then the computation contains a state where either some edge in \( CP \) is shorter than in the initial state or \((a, c) \in CP\).*

*Proof.* The timeout action in process \( c \) is always enabled. When executed, it adds \textit{message}(c) to the incoming channel of process \( a \). Then, the lemma follows from Lemma 3. \( \square \)
Lemma 6. If a computation starts in a state where there is a link \((a,b) \in CP\), then the computation contains a state where some link in \(CP\) is shorter than in the initial state or there is a link \((b,a) \in CP\).

Proof. Assume without loss of generality that \(a < b\). Once \(a\) executes its always enabled timeout action, link \((b,a)\) is added to \(CC\). We need to prove that either some link in \(CP\) is shortened or this link is added to \(CP\).

Let us consider a link \((b,c) \in CP\) such that \(c < b\). There can be three cases with respect to the relationship between \(a\) and \(c\). In case \(c < a\), the lemma follows from Lemma 3. In case \(c = a\), the claim of the lemma is already satisfied. The case of \(c > a\) is the most involved.

According to Lemma 4, if \(c > a\), the computation contains a state where a shorter link to \(a\) belongs to \(CC\). That is, there is a process \(d\) such that \(a < d \leq c\) and \((a,d) \in CC\). Let us consider link \((e,d) \in CP\) such that \(e < d\).

If \(e < a\), then, according to Lemma 3, some link in \(CP\) shortens. If \(e = a\), then some link in \(CP\) shortens according to Lemma 5. In both cases the claim of this lemma is satisfied.

Let us now consider the case where \(e > a\). According to Lemma 4, the link to process \(a\) in \(CC\) shortens. The same argument applies to the new shorter link to \(a\) in \(CC\). That is, either some link in \(CP\) shortens or a link to \(a\) shortens. Since the length of the link to \(a\) is finite, some link in \(CP\) eventually shortens. Hence the lemma.
Lemma 7. If the computation is such that if \((a, b) \in CP\) then \((b, a) \in CP\) in every state of the computation, then this computation contains a suffix where \(((a, b) \in CP) \Rightarrow ((a, b) \in CC)\)

Lemma 7 states that if \(CP\) does not change in a computation then eventually, the links in \(CP\) contain all the links of \(CC\). The proof follows from the operation of the algorithm.

Lemma 8. Let \(CP\) be strongly connected in some state of the system. Let it also be that for every pair of processes \(a\) and \(b\) in this state, if \((a, b) \in CP\) then \((b, a) \in CP\). In this case, this state satisfies LP.

Proof. Let us prove the if part of LP first. Assume that the state in the condition of the lemma violates LP. That is, there is a pair of consequent processes \(u\) and \(v\) that are not neighbors. By condition of the lemma, \(CP\) is strongly connected. This means that there is a path from \(u\) to \(v\). Let us consider the shortest such path. Since \(u\) and \(v\) are not neighbors, the path has to include processes to the left or to the right of both \(u\) and \(v\). Assume without loss of generality \(u < v\) and the path includes processes to the right of \(u\) and \(v\). Let us consider the rightmost process in this path \(w\). Let \(x\) and \(y\) be the processes that respectively precede and follow \(w\) in this path. Since \(w\) is the rightmost, both \(x\) and \(w\) are to the left of \(w\).

Note that each process in \(CP\) can have at most one outgoing left and one outgoing right neighbor. By the condition of the lemma the outgoing neighbor of a process is also its incoming neighbor. Since \(x\) precedes \(w\) in the path from \(u\) to \(v\) and \(y\)
follows $w$, $x$ is the incoming and $y$ is the outgoing neighbors of $w$. Yet, $x$ and $y$ are both to the left of $w$. This means that $x = y$. However, this also means that $w$ can be eliminated from the path from $u$ to $v$ and can be shortened this way. However, we considered the shortest path from $u$ and $v$. It cannot be further shortened. We arrived at a contradiction which proves the if part of the lemma.

The only if part follows from the observation that each process can only have a single right and single left neighbor. That is, if a process is already a neighbor with the consequent process it cannot be a neighbor with any other process. \hfill \Box

**Theorem 3.** Program $l$-Corona is a weakly channel-connectivity existing identifier stabilizing solution to the linearization problem.

*Proof.* To prove the theorem we show that $l$-Corona stabilizes to $LP$. The closure of $LP$ follows immediately from the operation of $l$-Corona. Indeed, $LP$ states that the links in $CP$ connect consequent processes. The only change that $l$-Corona can do to links in $CP$ is shorten them. However, the length of the links to consequent processes is already zero and they cannot be further shortened.

Let us now address the convergence of $LP$. Consider a computation of $l$-Corona. According to Lemma 6, for each process $a$ if there is a link $(a, b) \in CP$, then some link is shortened in $CP$ or there is a state where $(b, a)$ also belongs to $CP$. Since links can be shortened only a finite number of times in a computation, there is a suffix of this computation where in every state if $(a, b)$ belongs to $CP$ so does $(b, a)$. Note that $CP$ does not change in this suffix of the computation, hence, according to Lemma 7,
there is also a suffix where links in CP and CC are identical.

According to Lemma 2, CC is not disconnected during a computation of l-Corona. This means that in this suffix CP is also connected. According to Lemma 6 then, CP is strongly connected. Then, according to Lemma 8, this computation contains a state where LP is satisfied. Hence the theorem. \qed

3.4 Skip-List Stabilization

**Problem statement.** The problem maps each set of processes to a set of valid 1-2 skip-lists. In each skip-list the bottom level is linearized and for each level $i > 0$, the following predicate $SL$ holds: any two processes $a$ and $b$ are neighbors at level $i$ if the distance between $a$ and $b$ at level $i - 1$ is no less than 2 and no more than 3 hops.

**s-Corona description.** Each level of s-Corona has two sub-levels: *status decision* sublevel — sd-Corona, and *neighbor linking* sublevel sn-Corona.

sd-Corona of level $i$ uses neighborhood information of level $i - 1$ to determine the status of a process at level $i$. Depending on whether the process participates at level $i$, the process status is either up or down. If a process is down at level $i$ it is down at all levels above $i$. On the basis of this information sn-Corona links $p$ with its left and right neighbor at level $i$. sn-Corona of level $i$ does not influence the operation of sd-Corona at level $i$. If process $p$ is up, sn-Corona inspects $i - 1$ neighbors three hops away from $p$ to determine the nearest up neighbor and connects it to $p$. To ensure overall CC connectivity preservation sn-Corona sends itself the link to the
previous neighbor at level 0 for l-Corona to handle. The stabilizing implementation of sn-Corona is relatively straightforward. We, therefore, do not present it and focus on sd-Corona instead.

**sd-Corona description.** sd-Corona operates similarly at each level. At every level it maintains a set of variables that belong to only this level. At level $i$, process $p$ of sd-Corona makes use of the identities $p.(i - 1).l$ and $p.(i - 1).r$ of its respective left and right neighbors at level $i - 1$. sd-Corona at level $i$ does not change these identities. Therefore, they are assumed constant for the operation of sd-Corona at this level.

At level $i$, process $p$ of sd-Corona maintains two status variables: $p.i.st$ and $p.i.str$. The values for both are **up** and **down**. Variable $p.i.st$ stores the status of $p$ itself. Variable $p.i.str$ keeps the status of the right neighbor of $p$. The status of the rightmost and leftmost process at level $i$ are fixed as **up** and **down** respectively and are considered constant.

The idea of sd-Corona is to ensure that no two consequent neighbors are **up** and no three of them are **down**. To break symmetry in deciding who of the neighbors should change status, the decision of the right neighbor is favored.

sd-Corona has three guards. The timeout guard sends the status of $p$ to its neighbors. The two receive guards process messages from the left and right neighbors of $p$. If $p$ receives a status value from its right neighbor, it updates $p.i.str$ and its own
status. If both $p$ and its right neighbor are **up** then $p$ changes its status to **down**. If $p$ receives a message from its left neighbor and discovers that its neighbors and itself are **down**, it changes its own status to **up**. The operation of s-Corona is illustrated in Figure 17.

**Correctness proof.**

**Lemma 9.** If process $a$ at level $i$ of sd-Corona changes its status $st$ only a finite number of times in the computation, then this computation contains a suffix where every message in the outgoing channel of $a$ carries the same value as $a.i.st$ and $b.i.str = a.i.st$ for the left neighbor $b$ of $a$.

**Proposition 1.** If, in some computation, none of the processes at some level $i$ change their status, then this computation also contains a suffix where for each process $a$, $a.i.r$ and $a.i.l$ point to the nearest up process at this level and do not change.

**Lemma 10.** If in some computation none of the processes at some level $i-1$ change their right and left neighbors, then this computation also contains a suffix where none of the processes at level $i$ change their status.

**Proof.** The proof is by induction on the number of processes on level $i$. The induction is carried out from the right end of the process list. To simplify the description we assume the processes are numbered 1 to $n$ from right to left. Note that the status of the first (rightmost) process is constant. Assume that there is a suffix of the computations where $j-1$ right processes do not change their status.
According to Lemma 9, this computation also contains a suffix where all messages from process $j - 1$ to process $j$, as well as $j.r$ have the same value as the status of process $j - 1$. In this case there is a suffix of the computation, where $j.i.r$ does not change. Then, in this suffix $j.i.st$ may change at most once. Specifically, if $j.i.st$ and $j.i.r$ are both down, then $j.i.st$ can be set to up if $j$ receives a message with status = down from process $j + 1$. Thus, this computation contains a suffix where $j$ does not change its status. The lemma follows by induction.

Lemma 11. In each computation of s-Corona, every process $p$ changes its status and its left and right neighbors only finitely many times.

Proof. The proof is by induction on the levels of s-Corona. At level zero, the lemma holds due to Theorem 3. Assume that there is a suffix of this computation where the status and neighbors of processes at level $i - 1$ do not change. Then, according to Lemma 10, there is a suffix of this computation where the status of processes at level $i$ does not change either. If that is the case, then, due to Proposition 1, there is also a suffix where the neighbors do not change. The lemma follows by induction.

Theorem 4. s-Corona is a weakly channel-connectivity existing identifiers stabilizing solution to the 1-2 skip-list construction problem.

Proof. To prove the theorem, we show that s-Corona converges to the 1-2 skip-list predicate $SL$. According to Lemma 11, the processes in sd-Corona change their status only finitely many times.
Due to the algorithm design, this means that the sd-Corona converges to predicate where, two consequent processes at level $i - 1$ cannot be up and three consequent ones cannot be down. That is, the process status at level $i$ is appropriate for the $1$-$2$ skip-list. Due to Proposition 1 they are correctly linked. Hence the theorem.
Figure 15: Example computation of l-Corona. To simplify the picture each process is represented by two nodes. The primed nodes are the process’ incoming channel. Solid lines denote identifiers stored in $l$ and $r$ of each process. Dashed lines are identifiers in the incoming channel.
process $p$

constants

$p.(i-1).r, p.(i-1).l$ // identifiers of right and left neighbors at
// level $i - 1$

variables

$p.i.st,$ // own status at level $i$, either up or down
// constant and set to up for process with highest id
// constant and set to down for process with lowest id
$p.i.str$ // status of right neighbor

actions

message(status) $\in p.C$ from $p.(i-1).r$ $\rightarrow$
receive message(status),
p.i.str := status,
if $(p.i.st = up) \land (p.i.str = up)$ then
  p.i.st := down

message(status) $\in p.C$ from $p.(i-1).l$ $\rightarrow$
receive message(status),
if $(status = down) \land (p.i.st = down) \land (p.i.str = down)$ then
  p.i.st := up

true $\rightarrow$
if $p.(i-1).r < +\infty$ then send message(p.i.st) to $p.(i-1).r$,
if $p.(i-1).l > -\infty$ then send message(p.i.st) to $p.(i-1).l$

Figure 16: Status decision component of skip-list part of Corona (sd-Corona).
(a) initial state

(b) at level 0, processes $d$ and $h$ receive messages that their right neighbors are up, they change their statuses to down

(c) at level 0, $e$ receives message from $f$ that its status is up and changes its own status to down; $f$ and $i$ are linked at level 1

(d) at level 0, $d$ receives messages that both $c$ and $e$ are down and changes its status to up, links with neighbors at level 1

(e) at level 1, $i$ receives message from $f$ that its status is down, updates its own status to up

(f) at level 2, $i$ links with $b$

Figure 17: Example computation of s-Corona. For simplicity, neighbor links are always assumed bidirectional.
CHAPTER 4

Extensions

4.1 Topology Updates

A topology update is a node joining or leaving the set of processes $N$. We address topology updates when the system is in correct state, i.e., we consider the simple case where a node joins or leaves a linearized set of processes. Formally, we assume that in the initial state of the computation, the program satisfies the linearization predicate $LP$. Note that the skip-list above this linearized list may be incorrect due to nodes joining or leaving. However, it turns out that a slight update per level is sufficient to handle that given that every node $v$ stores, in addition to $v.i.l$ and $v.i.r$, a flag for both $v.i.l$ and $v.i.r$ in order to remember if $v.i.l$ (resp. $v.i.r$) also has an $i$-level link back to $v$. When determining the status of level $i$ only once the flags w.r.t. level $i-1$ have been set, a joining node will only start getting integrated in level $i$ once it found its right place in level $i-1$, which implies the following lemma.

**Lemma 12.** A removal or addition of a node at level $i-1$ leads to at most one process status change in $sd$-Corona at level $i$.

**Proposition 2.** The operation of $sn$-Corona at level $i$ in case of a single status change of a node in $sd$-Corona at level $i$ is equivalent to a single state transition that reconnects up neighbors at level $i$. 
Recursively applying Lemma 12 and Proposition 2 to the levels of the skip-list, we obtain the following theorem.

**Theorem 5.** The number of topological changes Corona requires to reconstruct the skip-graph after a single topology update is in $O(\log n)$

4.2 Skip-Graphs

We would like to describe the extension of Corona to skip-graphs. For that, Corona has to run two instances of sn-corona at each level $i$. The main instance operates as before, while the alternative instance constructs an alternative list out of the nodes that do not participate in the main list. Note that in the 1-2 skip-list, one alternative list can always be constructed. An instance of sd-Corona at level $i + 1$ runs each of the lists. The process of splitting into main and alternative list continues iteratively on each thus formed list. No changes are required in either l-Corona or sd-Corona.

4.3 $k$-l Skip-List

Corona can be extended to accommodate an arbitrary $k$-l skip-list in several ways. For example, each process in the extended version of Corona maintains the status of $k - 1$ right neighbors and one left neighbor. If $p$ detects that it is up and there is an up right neighbor less than $l$ hops away, then $p$ changes its status to down. If $p$ is down and there are $k + 1$ consequent down processes, it goes up.
4.4 Oracles

In their seminal paper, Fischer et al. [66] show that for elementary problems, such as consensus, are not solvable in asynchronous systems even for a single process crash. Consensus is a classic problem in solving fault tolerance. In consensus, initially each process inputs a value and the processes have to select an output a single value. Intuitively, in a purely asynchronous system, processes cannot distinguish a crashed process to an arbitrarily slow one. This result limits the usefulness of the asynchronous system model as a tool for studying fault tolerance. Chandra and Toueg [67] circumvent this impossibility by introducing a concept of failure detector. A failure detector is a distributed oracle that provides information about failed processes. Failure detector itself cannot be implemented in asynchronous systems. The failure detector encapsulates the synchrony assumptions necessary to solve consensus and other problems in asynchronous systems. Therefore, researchers are interested in finding the weakest failure detector that requires the least amount of information needed to solve a problem in the presence of a fault. The weakest failure detectors have been proposed for a number of problems such as Atomic Broadcast, Consensus, Agreement, leader election and others [67–72].

Chandra and Toueg [67] present an unreliable failure detector to solve consensus in asynchronous systems with crash failures. They show that unreliable failure detector that makes an infinite number of mistakes can be used to solve consensus despite crashes. They also determine which detectors can be used to solve Consensus despite
any number of crashes, and which ones require a majority of correct processes.

In the companion paper, Chandra et al. [68] show that their failure detector, \( \Omega \) is the weakest failure detector for solving Consensus. Chandra et al. formally define a failure detector \( D \) to be the weakest failure detector for solving a problem \( M \) if: \( D \) sufficiently solves \( M \) and that there exists an algorithm that solves \( M \) using \( D \); and \( D \) is necessary to solve \( M \) such that any failure detector that is sufficient to solve \( M \) provides at least as much information about failures as \( D \) does.

Uniform consensus is a version of consensus that requires no two processes, correct or faulty can reach different values after a decision have been made. Delporte et al. [69] prove that \( (\Omega, \Sigma) \) is the weakest failure detector for uniform consensus. In each process, a quorum failure detector \( \Sigma \) outputs a set of processes. This guarantees that at any time and any process, two sets of output intersect. Furthermore, they show that \( (\Omega, \Sigma) \), after a given time, every set output at a correct process consists only of correct processes.

A nonuniform consensus is a version of consensus that requires no two correct processes can reach different decisions. This differs from the uniform consensus since a faulty process can decide on any proposed value. Eisler et al. [70] determine the weakest failure detector for this problem. They defined a nonuniform version of \( \Sigma \), \( \Sigma'' \), and prove that \( (\Omega, \Sigma'') \) is the weakest failure detector to solve nonuniform consensus in any environment. In their solution, \( \Sigma'' \) is similar to \( \Sigma \), except that the intersection requirement is restricted to quorums output at correct processes. In other words,
any two quorums output at correct processes intersect. Quorum output at faulty processes, however, may fail to intersect with quorums output at other processes.

Guerraoui et al. [71] present Υ, a failure detector that provides very little information. In every run of the distributed system, Υ will eventually inform the processes that some set of processes in the system cannot be the set of correct processes in that run. Even though their solution might provide a solution for an arbitrarily long time and only excludes on possibility of correct sets, Υ still captures nontrivial failure information.

Zielinsky [72] presents Anti-Ω which is both sufficient and necessary for the set agreement problem. In set agreement problem, \( n \) processes have to decide on at most \( n - 1 \) proposed values. Each query to the Anti-Ω detector returns a process id. The detector guarantees that there is a correct process whose id will be returned only finitely many times. Therefore, eventually, the detector will never output this process id. However it is possible that Anti-Ω might not stabilize and that more than one process id will be returned infinitely often.

**Problem definition.** As the impossibility results indicate, certain properties of the computation model and the peer-to-peer system problems preclude purely algorithmic solutions to even elementary peer-to-peer system constructions. Oracles are able to encapsulate the impossible and highlight the actual program.
An elementary problem for peer-to-peer systems is topological sorting or linearization. This problem is often the foundation for more sophisticated peer-to-peer topologies. On the other hand, it is complex enough to serve as a case study for the application of oracles in peer-to-peer networking. To increase the generality of our results we would like to consider non-stabilizing linearization problem.

Specifically, we define the problem as follows. Initially each process is assigned a set of identifiers. The channels are empty. The problem requires the following output. Each process with identifier \( p \) has to output, not necessarily simultaneously, the identifiers of its successor and predecessor. If \( p \) has the highest or lowest identifier in the system, \( p \) has to respectively output \( +\infty \) or \( -\infty \). We can consider two variants of the problem: strict and eventual linearization. In strict linearization (SL) each process outputs the identifiers exactly once. Only correct outputs are allowed. This variant is similar to classic consensus. In a relaxed variant of the problem, eventual linearization (EL), each process has to eventually output correct identifiers. However, the process may output incorrect identifiers for a fixed prefix of the computation.

The problem statement also depends on whether non-existent identifiers may be present in the system. Non-existing identifier linearization (NID) allows these identifiers to be present in the initial state and injected by the oracles. Existing-only identifier (EID) variant prohibits such identifiers.

This classification defines four possible linearization problem variants.
**Oracles.** We assume that the oracles may do the following actions: add an identifier to a memory of a process, remove identifier from a memory of a process or change the process behavior. The oracles are therefore called id-adding, id-removing and behavior-changing respectively.

The oracles can be finite or infinite. For a finite oracle, there is a constant $K$, such that in any program computation the number of times the oracle performs its action is at most $K$. This constant may be different for different oracles.

We introduce the following oracles. *Weak connectivity oracle* (WC) is a finite id-adding oracle. If a computation contains a state where the weak-connectivity system graph is disconnected, WC adds the identifier such that the graph is connected. *Continuous weak connectivity oracle* (CWC) is the infinite variant of WC. That is, CWC may reconnect the system infinitely many times. *Participant oracle* (P) is a finite id-removing oracle. If a computation contains a state where a process has a non-existing id in its memory removes it. *Continuous participant oracle* (CP) is the infinite variant of P. *Neighbor detection oracle* (ND) is the behavior changing oracle: if it detects the presence of the left or right neighbor identifier of the process, it makes the process output this neighbor identifier.

**Solution conjectures.** We conjecture that the algorithm to solve the linearization problem is exactly the same as described in Chapter 3. However, depending on the problem variant, the necessary oracles differ.
Specifically, strict linearization with existing identifiers requires WC oracles. Eventual linearization with existing identifiers needs CWC and ND oracles. Strict linearization with non-existing identifiers requires WC and P oracles. Eventual linearization with non-existing identifiers needs CWC, ND and CP oracles. The proofs of these conjectures and further details are now in preparation for publication [44].
CHAPTER 5

Conclusion

Research impact. In this dissertation we presented an algorithm for building a robust peer-to-peer system. The presented system is described and rigorously proven correct in the computation model that allows its implementations in a realistic system.

In our work we preserve the generality of asynchronous message passing system which makes it possible to preserve the correctness of the algorithm regardless of the implementation details.

Our research provides engineers with a blueprint to build realistic implementation and provides researchers with a foundation for further exploration of robust peer-to-peer algorithm design.

Future work. In closing we would like to outline several interesting research directions that would extend our work to further its applicability and significance. Corona is able to handle topological changes within a limited locality. We would like to extend Corona to be able to handle concurrent topology changes and contain the affects of churn. In peer-to-peer systems nodes join and leave continuously. This churn [73–75] poses a challenge for system stabilization. Prior work in handling churn mainly focuses on maintaining topology through overwhelming numbers of stable, non-churn
nodes. However, it seems that due to the properties of message-channels used as additional storage containers, the effect of churn can be counteracted at significantly lower costs.

The Corona algorithm provides capabilities that can be used in a wide range of peer-to-peer applications such as filesharing, collaborative writing, distributed backup, lookup services, cloud computing and many more. We would like to develop a peer-to-peer application based on the skip-graph version of Corona and to measure its performance. It would be interesting to observe how the system would behave in a stress test of node failures, insertion and update operations on a real network.
BIBLIOGRAPHY


