UiT THE ARCTIC UNIVERSITY OF NORWAY

Faculty of Science and Technology
Department of Computer Science

Verification of the Chord protocol with TLA$^+$

—

Jørgen Aarmo Lund

INF-3990 Master's Thesis in Computer Science

To my friends and family.

“On two occasions I have been asked,
’Pray, Mr. Babbage, if you put into the machine wrong figures,
will the right answers come out?’”
–Charles Babbage

Abstract

In traditional software engineering methodologies, software correctness is established through testing and progressive fault mitigation. Safety properties are established by demonstrating that a sufficiently large number of test cases fail to violate them.

In contrast, *formal verification methods* permit a systems design process where desired safety properties are stated outright in the system specification, and enforced by automated analysis tools. This is of particular interest in designing distributed systems, where safety properties may be easy to formally define and specify, yet hard to implement in practice.

Despite this promise, the use of formal methods has largely been confined to academia and certain classes of safety-critical systems. Recently, however, companies like Amazon and Microsoft have adopted formal verification tools to verify distributed system designs.

In this thesis, we present a formal specification of the Chord distributed hash table protocol, using the TLA$^+$ specification language. We specify the protocol at a coarse level with a relaxed failure model, and then increase the granularity and introduce fail-stop failures, yielding a formal specification of Chord with asynchronous messaging and fault-tolerance mechanisms.

We first model-check the specification under the constraint that no failures occur, and show that it satisfies critical safety properties. We then show that the introduction of failures leads the specification to admit several behaviors which break the safety properties Chord promises, potentially leading to permanent partitions in the network and performance degradation.

As part of this work, we provide an overview of formal verification methods; we discuss certain formalisms and logics involved in modelling and proving algorithms, show potential advantages of applying formal methods to distributed systems design, and identify barriers keeping formal methods from widespread use.

Acknowledgements

I am very grateful to my advisor Håvard D. Johansen for his patience, insight, and guidance. Without his continued support and motivation, this thesis would not have been possible.

I would like to thank my family for their support and patience, and for keeping me motivated throughout the writing process.

I would also like to thank my colleagues at DIPS — Arild, Håkon, Hans-Erik, Joakim, Kari, Marius, Ole-Martin, and Sverre — for their support and guidance. I would especially like to thank my boss Daniel for being so accommodating in the final month of thesis work.

My officemates in the Corpore Sano lab — Andreas, Nikolai, Natalie, Thomas and Tor-Arne — provided valuable feedback for early drafts, and endured more complaints about formal verification than anyone should be expected to. For this I am grateful.

Contents

List of Figures

List of Tables

List of Abbreviations

CTL	computation tree logic
FPGA	field-programmable gate array
GPU	graphics processing unit
IDE	integrated development environment
JVM	Java virtual machine
LTL	linear time logic
OOM	out of memory
PODC	Principles of Distributed Computing
RPC	Remote Procedure Call
SSD	solid-state drive
TLA	Lamport's Temporal Logic of Actions
TLC	the TLC model checker
TON	Transactions on Networking
ZF	Zermelo–Fraenkel

Glossary

appendage A chain of one or more Chord nodes that haven't been inducted into a larger ring

bisimilar See *bisimulation*

bisimulation Demonstrating that two transition systems are equivalent by showing that they can stepwise match each other's actions [1], so that each system *simulates* the other. See also [2]

Byzantine failure A failure model where failing nodes have completely arbitrary behavior [3]

enabled action An action that can be performed from the current state.

eventual consistency A guarantee that replicas of data will eventually converge to the same value if no updates occur [3]

fail-stop A failure model where nodes only fail by halting [3]

inductive invariant An invariant of a specification that is also an invariant of its next-state action [4].

liveness property An assertion that a behavior eventually satisfies a proposition (i.e. "something good eventually happens" [5])

loopy A Chord ring without ordered identifiers, where more than one node has a node with a smaller identifier as its successor

pure-join The Chord protocol without fault-tolerance mechanisms

safety property An assertion that a proposition is true for any state in a behavior (i.e. "something bad does not happen" [5])

state predicate An action with no primed variables, which can be evaluated to true or false for any given state

state space The set of all possible states a system can take on

strong fairness A formula stating that an action eventually occurs if it is infinitely *often* enabled after some point. See [4]

strongly connected component A subcomponent of a graph where every vertex is reachable from every other vertex [6]

stuttering An action which leaves all variables unchanged.

transitive closure A relation R^* over another relation R such that $(r, s) \in R^*$ if there is a path from r to s in R [7, 8], i.e.

$$(r, s) \in R^* \Leftrightarrow (r, a_1) \in R \wedge (a_1, a_2) \in R \wedge \cdots \wedge (a_n, s) \in R$$

weak fairness A formula stating that an action eventually occurs if it is perpetually enabled after some point. See [4]

List of Symbols

$\Box$	Always
$\Box\Diamond$	Always eventually
$\Diamond$	Eventually
$\Diamond\Box$	Eventually always
$\exists$	There exists
$\forall$	For all
$\mathbb{N}^*$	The positive integers $\{1, 2, 3, \cdots\}$
$\vee$	Boolean OR
$\wedge$	Boolean AND

1 Introduction

Diagnosing errors in distributed systems presents unique difficulties [9]. They have causes of errors that are not immediately obvious: from race conditions, and deadlocks, to unreliability in the underlying network. Since the state of the system is distributed across multiple autonomous components, reconstructing the state leading to a particular error can be difficult. For instance, logging across distributed components may not be synchronous, and typically does not capture enough information to fully reconstruct the order of events between different hosts [10].

The difficulty in diagnosing errors in distributed systems at runtime dovetails with a larger trend of new software development tools and methods which verify systems' correctness during development. These include unit testing frameworks [11], programming languages with stronger compile-time constraints and guarantees (e.g. Mozilla's *Rust* [12]) and static-analysis tools [13, 14].

Part of this trend is a resurgence of *formal verification* methods. Formal verification methods model systems in a formal mathematical and logical framework, and attempt to provide a complete description of a system in an unambiguous, formal *specification*. Formal specifications can be thoroughly verified with automated tools, providing mathematically rigorous and reproducible analysis of the system.

Formal verification systems consist of three components [15, 16]:

- A framework to describe and model systems, typically featuring a *description language* for machine-interpretable system descriptions.

- A *specification language*, which similarly allows formal expression of the properties and requirements that the system model should satisfy.

- A *verification method* that can determine whether a system model satisfies a set of requirements.

In turn, verification methods can be divided into:

- *Proof-based* methods, which argue that the system is correct by deduction: by applying transformations and proof rules to derive the requirements from the model, they lead to deductive proofs showing that the properties must follow from the definition of the system. *Proof assistants* [17] allow the resulting proofs to be verified automatically, alerting the designer to any logical inconsistencies.

- *Model-based* methods, which demonstrate system properties by induction: *model checkers* [16] enumerate possible states that the system can be in, and verifies that none of these states leads to a violation of the requirements. If the system is inconsistent with the specification, the checker can provide a detailed counterexample showing the steps leading to the inconsistency.

The idea of applying formal reasoning to verify software is far from a new one. McCarthy argued for a mathematical theory of computation as early as 1961 [18]. During the 1980s, a significant amount of research effort was devoted to introducing formal verification methods into software development processes.

Despite these efforts, practical usage of formal verification remained sparse. In 1996, Hoare [19] argued that it had fallen by the wayside as a way of building reliable systems, as better software development and testing processes ended up providing just as much practical confidence in systems' correctness. In his 2004 book, Gutmann outlines [20] the main challenges projects faced in applying formal verification methods to software projects:

- Formal methods scale poorly to larger systems. In particular, model-based methods are vulnerable to *state space explosion*, where the specification's complexity leads to so many possible states that checking them becomes intractable. Proof-based methods suffer from a similar issue with proof length, where proofs become vastly longer than the specification they're verifying.

- While formal verification methods are occasionally claimed to guarantee correct and flaw-free systems [21], they do not provide such guarantees in practice. Formal specifications must make certain assumptions and abstractions to be practically verifiable, potentially obscuring errors, and model checkers and proof assistants can themselves have bugs which lead to erroneous specifications being accepted.

- Formal verification is hard to fit into agile methodologies. Every feature added to the system alters the design of the system, meaning the formal specification of the design must be updated as well, in turn requiring the specification to be verified from scratch.

- There is no practical link between the model and the actual implementation. The system is essentially implemented twice — as a formal specification, then an implementation — and since there is no way to guarantee that the implementation satisfies the specification, it is possible to introduce errors in the implementation that are not captured by the specification, and vice versa.

With time, the scalability of formal verification methods have improved, with increased computing power, the general availability of cloud computing [22] services, and theoretical advances making verification of larger specifications tractable. This has lead to certain companies adopting formal verification for non-safety critical applications. In verifying formal specifications as part of their workflow, Amazon engineers [23] reported discovering errors which would otherwise have been extremely difficult to uncover and reproduce. In one instance, they uncovered a bug in their DynamoDB [24] database service with a 35-step error trace.

One the primary motivating factors Amazon cited for adopting formal verification tools was Zave's analysis [25] of the Chord [26, 27, 28] distributed hash table. Advertised for "its simplicity, provable correctness, and provable performance" [26], Chord received the SIGCOMM Test of Time award in 2011.

Questioning the "informal reasoning" [25] in the paper, Zave created a formal specification of Chord's network maintenance and fault-tolerance algorithms. In doing so, Zave found several instances of the protocol being underspecified, to the point of violating its claimed provable correctness. She outlined examples of the protocol breaking its promised safety invariants, at worst leading to unrecoverable splits in the network. Zave would later present a version [29] of Chord with new fault-tolerance algorithms correcting these flaws, and show its correctness with an Alloy proof.

One particularly interesting formal verification framework for distributed systems design is the TLA$^+$ specification language [30], which has been adopted by Intel [31], Microsoft [32], and Amazon [23]. Designed by Leslie Lamport to formally describe distributed and concurrent systems, the TLA$^+$ language is based on Lamport's TLA [33] logic, additionally incorporating Zermelo–Fraenkel (ZF) set theory and first-order logic [4]. TLA$^+$ supports both proof- and model-based verification, with the TLC [34] model checker and TLAPS [35] proof assistant both operating on TLA$^+$ specifications.

1.1 Thesis statement

This thesis focuses on the argument that the validity of formal verification hinges on the perceived correctness of the model-checker or proof assistant. We argue that this criticism might be addressed by showing that results from one formal verification framework can be reproduced in other frameworks.

To this end, this thesis will specify the Chord protocol in the specification language TLA$^+$ [4], and show that the resulting specification admits the counterexamples presented by Zave in her 2012 paper [25].

Our hypothesis is that *the Chord protocol can be formally specified in TLA$^+$, and the resulting specification can be model-checked to prove that it does not preserve the safety invariants promised by Liben-Nowell et al. [36]*.

1.2 Scope and limitations

In reproducing Zave's work, we adopt a similar scope and failure model for our specification. Therefore, this thesis will only specify the algorithms necessary to establish and maintain the Chord network, and ignore the storage and retrieval of key-value pairs. It also assumes the same fail-stop failure model Stoica et al. assume in their presentation of the protocol.

In particular, we will not consider Byzantine failure, denial-of-service attacks [37], or deliberate attempts to disrupt the protocol [38]. While there are attempts to make Chord resistant to Byzantine faults (such as the work of Fiat et al. [39]), neither Stoica et al. nor Zave include them in their failure models.

1.3 Context

This thesis is written within the context of the Corpore Sano[1] center, which conducts interdisciplinary research in the intersection between computer science, sports science and medical research.

As part of this work, Corpore Sano has conducted several research projects involving distributed systems, including the Fireflies [38] Byzantine fault-tolerant overlay network protocol and the Diggi [40] framework for secure distributed agent-based computation.

Corpore Sano also researches privacy and security measures for systems handling sensitive medical data, and have proposed approaches for enforcing privacy policies by associating programmable *meta-code* [41] with files, and using trusted SGX enclaves to implement use-based privacy [42].

1.4 Methodology

In 1989, as part of an ACM task force on the Core of Computer Science, Denning et al. [43] outlined their model of computer science as a composition of three paradigms:

1. **Theory**, encompassing the mathematical underpinnings of computer science: hypothesizing about theoretical objects and their relationships, constructing formal theorems about them, and proving these theorems through logical and mathematical reasoning.

2. **Abstraction**, which covers the scientific method — forming hypotheses, constructing models and experiments — and concerns itself with creating verifiable hypotheses about systems.

3. **Design**, which accounts for the engineering aspect — systems specification, design, implementation, and testing — and emphasizes the ability to build practical implementations of systems.

Denning et al. emphasize the overlapping nature of these paradigms. System models build on theoretical foundations, the engineering process builds on hypotheses verified with these models, and theoretical structures are ultimately based in the realities of implementing systems.

1. `https://corporesano.no/`

This overlap becomes apparent when examining formal verification methods. Formal verification is tightly bound to the theoretical domain, with underpinnings in formal mathematics and logic. Simultaneously, it requires explicit abstraction — we hypothesize that a formal description of a system is consistent with certain properties — and allows formal, unambiguous specifications of a system's design.

Bearing this in mind, this thesis employs a systems research methodology spanning all three paradigms. We construct theoretical models of the protocol, form hypotheses about them, and verify these hypotheses. While we do not implement the protocol itself, we construct formal specifications of the protocol, and perform experiments using these specifications to evaluate formal verification tools.

1.5 Outline

The remainder of this thesis is structured as follows:

Chapter 2 gives a brief history of formal verification methods, using Lamport's Temporal Logic of Actions (TLA) [33] as a focal point. This chapter will also summarize the Floyd-Hoare [44] logic and Pnueli's [45] temporal logic for concurrent program analysis, in order to provide a theoretical foundation and points of comparison when we finally describe the TLA logic.

Chapter 3 provides an outline of the Chord protocol, based on the papers and technical reports published by Stoica et al. [26, 27, 28] and the analysis of the protocol by Liben-Nowell et al. [36]. It also provides a summary of the protocol analysis by Zave [25].

Chapter 4 describes the process of formally specifying the Chord protocol, starting from a simplified specification under relaxed assumptions, which is then gradually refined until we have a full specification of the Chord protocol which models both multiple concurrent network events and fail-stop failures.

Chapter 5 gives the results of model-checking our specification, listing both the properties demonstrated, and the time spent model-checking the specification for a range of network sizes. It also lists Chord's claimed safety properties, and whether they hold in our specification. Finally, it discusses the results and their implications.

Chapter 6 lists related work in formal verification of distributed systems, and other attempts to formally verify the Chord protocol.

Chapter 7 concludes this thesis, and proposes future work.

2 Background

In this chapter, we provide a brief history of formal verification methods, and summarize the logics presented by Hoare [44] and Pnueli [45], with the goal of providing a historical and theoretical background for Lamport's TLA logic [33].

The idea of making formal arguments about programs' correctness was entertained by several early computer scientists: even as early as the 1800s, Charles Babbage [46] wrote about the "Verification of the Formulae Placed on the [Operation] Cards" of his analytical engine.

One especially important early result is Alan Turing's examination of David Hilbert's *Entscheidungsproblem*. The Entscheidungsproblem ("decision problem") [47] asks for a general procedure that determines whether a formula in a first-order logic is also possible to prove within the axioms of that logic.

In 1936, Turing demonstrated [48] that solving this problem also requires a solution to the *halting problem*, which asks for a general procedure to determine whether a program runs forever or eventually halts. Turing showed that the existence of such an algorithm would lead to a logical contradiction, proving that there is no general solution to the halting problem. In his proof, Turing introduced the idea of a *Turing machine*, a simple hypothetical machine that was nonetheless capable of executing arbitrarily complex algorithms.

In proving the halting problem had no solution, Turing also showed that the

Entscheidungsproblem did not have a general solution. This result implies that there is no general verification method for programs [49] either: for every verification method, there are certain propositions it cannot prove or refute. This means that every logical framework for program verification has to decide which kinds of systems and properties it can practically model and verify.

Turing would go on to do further work in proving programs' correctness, submitting a paper [50] in 1949 on verifying a routine for addition of numbers, which outlined a general proof method for algorithms similar to the method later presented by Floyd [51]. In 1947, Goldstine and von Neumann [46] would also write a paper explaining how "assertion boxes" could be used to reason around "operation boxes" in order to justify the results of a set of operations.

However, it is unclear if Turing or von Neumann knew of each other's work, and neither paper seems to have been known to the later proponents of formal verification. Formal verification of software would only become a topic of discussion again in 1961, when McCarthy called for a mathematical theory of computation [18], hoping to inspire formal semantics for describing programming languages.

2.1 Floyd-Hoare logic

In 1967, Floyd [51] laid the groundwork for applying formal deduction to computer programs by framing algorithms as *flowcharts* of interconnected commands.

Floyd defines a flowchart as a directed graph of *commands* — such as variable assignments, control flow statements, and the algorithm's beginning and end — connected by edges which define the allowed control flow.

Figure 2.1 shows an example of a flowchart over an algorithm to sum a series of numbers together, which takes the n numbers $a_1, a_2, \cdots a_n$, and stores their sum in the variable S. The initial tag $n \in \mathbb{N}^*$ constrains the parameter n to the set of positive numbers $\mathbb{N}^* = \{1, 2, 3, \cdots\}$, and forms the basis for proving that the program ends with S taking on the sum of the numbers a (so long as we disregard integer overflow).

Each edge of control is then additionally tied to a logical proposition, called the edge's *tag*, which constrains the preconditions or results from a command. For instance, a tag may demand that only numbers are assigned to a variable, or that a variable is non-negative before entering a loop.

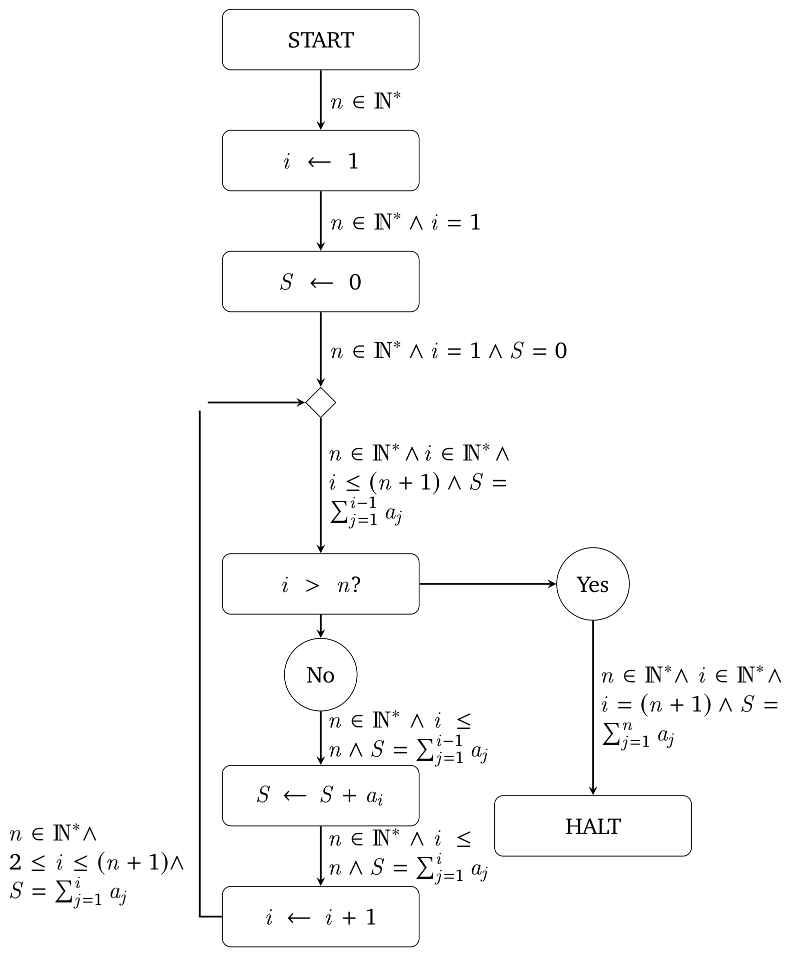

START
$n \in \mathbb{N}^*$
$i \leftarrow 1$
$n \in \mathbb{N}^* \wedge i = 1$
$S \leftarrow 0$
$n \in \mathbb{N}^* \wedge i = 1 \wedge S = 0$
$n \in \mathbb{N}^* \wedge i \in \mathbb{N}^* \wedge i \leq (n+1) \wedge S = \sum_{j=1}^{i-1} a_j$
$i > n?$
Yes
No
$n \in \mathbb{N}^* \wedge i = (n+1) \wedge S = \sum_{j=1}^{n} a_j$
$n \in \mathbb{N}^* \wedge i \leq n \wedge S = \sum_{j=1}^{i-1} a_j$
$S \leftarrow S + a_i$
HALT
$n \in \mathbb{N}^* \wedge i \leq n \wedge S = \sum_{j=1}^{i} a_j$
$n \in \mathbb{N}^* \wedge 2 \leq i \leq (n+1) \wedge S = \sum_{j=1}^{i} a_j$
$i \leftarrow i + 1$

Figure 2.1: An example of a Floyd logic flowchart, adapted from Floyd's original paper [51].

Floyd then constructs a simplified flowchart programming language consisting of single variable assignment, branching and joining commands. Then, it is shown that the logical validity of each command is transitive; by letting the *consequent* propositions constraining commands' results also act as *antecedent* constraints on the proceeding commands' input, it is possible to compose larger proofs spanning multiple commands, and by extension entire algorithms.

In this logic, verification takes place by proving that if each command's antecedents are satisfied, every command also satisfies its consequents. Then, by transitivity, the entire algorithm can be shown to return correct answers if its preconditions are satisfied. Hoare would later refer to this property as *partial correctness* [44].

The logic also made it possible to demonstrate *termination*, showing that an algorithm will eventually halt. This was accomplished by making every loop contingent on a steadily decreasing function, and showing that each loop must eventually terminate.

Two years later, Hoare [44] built on Floyd's logic to construct a set of axioms for applying formal deduction to algorithms. Hoare dispensed with the algorithm flowcharts, instead opting for a notation with *triples*: theorems on the form

$$P\{Q\}R$$

where Q is a (segment of a) program, and P and R are logical propositions, respectively known as the *precondition* and *postcondition*. When the precondition P is satisfied and Q is executed, the theorem states that R should be satisfied when Q successfully terminates.

For instance, the summation algorithm shown earlier in Figure 2.1 can be expressed by the triple

$$(n \in \mathbb{N}^*)\{i := 1; S := 0; \textbf{while } (i \leq n) \textbf{ do } (S := S + a_i; i := i + 1)\}(S = \sum_{i=1}^{n} a_i) \tag{2.1}$$

Hoare then outlines four axioms for analyzing algorithms:

The axiom of assignment The first axiom states that assignment of variables does not invalidate prior propositions: any proposition $P(f)$ about the value of the variable f remains valid if we assign f to a new variable x, and substitute all f in $P(f)$ with x to get the new proposition $P(x)$:

$$\vdash P(f)\{x := f\}P(x) \tag{2.2}$$

The consequence rule The second axiom provides a way to reason about implications with Hoare triples, showing that *modus ponens* can be applied to both pre- and post-conditions:

$$\vdash P\{Q\}R \wedge (R \rightarrow S) \Rightarrow \vdash P\{Q\}S \tag{2.3}$$

$$\vdash P\{Q\}R \wedge (S \rightarrow P) \Rightarrow \vdash S\{Q\}R \tag{2.4}$$

The composition rule The third axiom allows program segments to be composed into larger programs, akin to how Floyd's flowcharts allowed proofs for multiple commands:

$$\vdash P\{Q_1\}R_1 \wedge \vdash R_1\{Q_2\}R \Rightarrow \vdash P\{Q_1; Q_2\}R \tag{2.5}$$

The iteration rule The final axiom allows reasoning about looping algorithms by examining loops on the form "while the proposition B is true, repeatedly execute the segment S", written as "**while** B **do** S".

In essence, the iteration rule states that if the completion of the program S implies that the proposition P is true, and P is true before S is executed, then S can be repeated for any number of iterations without falsifying P. Additionally, since the loop only completes while its associated proposition is true, the postcondition can assume the proposition is false:

$$\vdash P \wedge B\{S\}P \Rightarrow \vdash P\{\textbf{while } B \textbf{ do } S\}\neg B \wedge P \tag{2.6}$$

Together, these four axioms form an axiomatic basis for inferring and proving properties of sequential algorithms.

While Floyd-Hoare logic proved to be an important foundation for formal verification, it suffered from several weaknesses:

- Constructing proofs in the logic turned out to be "tedious, difficult and [a process which] required human ingenuity" [16]. Even short programs could require long proofs to demonstrate partial correctness, limiting the scalability of the approach.
- The definition of partial correctness hinged on the program eventually halting, rendering it unsuitable for programs meant to run indefinitely [45], such as operating systems.

- It did not account for programs running in parallel: concurrency failures, such as deadlocks and livelocks, were not accounted for in the logic. Owicki and Gries [52] proposed solving this by adding new axioms allowing the addition of auxiliary variables to aid in the analysis, but noted that the process of proving programs could become "much longer" depending on the concurrency primitives used. A recent approach by Kojima and Igarashi [53] adapts Hoare logic for parallel analysis of graphics processing unit (GPU) kernels by augmenting triples with additional sets that represent the active threads of execution.

2.2 Temporal logic

The key idea to resolving these issues turned out to be a move from propositional to *temporal* logic: instead of working with propositions that were either true or false, temporal propositions could be true *some* of the time.

In 1977, Pnueli [45] proposed a method for concurrent program analysis building on temporal logic. Instead of treating computation as a set of discrete steps, the new logic viewed algorithms as sequences of atomic *events* separated by continuous spans of time.

In Pnueli's logic, a system $< S, R, s_0 >$ consists of a set S of states the system can take on, a relation R describing the allowed transitions between states, and the initial system state s_0.

Then, an execution of the system can be framed as a sequence of states

$$G' = s_0, s_1, s_2, \cdots$$

where each pair of states (s_i, s_{i+1}) corresponds to a transition allowed by R. To allow algorithm analysis in this framework, each state s is further broken down into two components:

- The *control* component π, analogous to the processor's instruction pointer, which holds the processor's current program location

- The *data* component u, which similarly holds the processor's current state (e.g. variables, memory)

This decomposition allows the transitions R to be expressed as a composition of a *next-state* function $N(\pi, u)$, which designates the next program location π' given the previous location and data, and a *transformation* function $T(\pi, u)$

which similarly describes which values u take on, given the algorithm step at location π. To simplify the notation and the proof techniques, these functions are assumed to be deterministic.

The same decomposition is key to extending the analysis to concurrent algorithms; to simulate a system which multiple concurrent processors, we can allow multiple control components in each state

$$s = <\pi_1, \pi_2, \cdots, \pi_n; u>$$

where each π_i is the control for processor i, and the state u is shared among all processors.

Then, for each step of the system, an arbitrary processor i is selected, and the statement at the location of π_i is executed. The transition functions N and T are still assumed to be deterministic and dependent on a single control π_i, making each step of the system an atomic action.

This means that the designer must decide on the appropriate granularity for the model. For a distributed system, it may be sufficient to partition the states by communication between processes, with message sending and receipt as separate steps, while a cache-coherence protocol may require steps at the processor instruction level.

With this framework in place, the verification problem is reduced to showing which system states a property holds in. By taking the transitive closure R^* of R over the initial state s_0, it is possible to find the set X of all reachable states

$$X = \{s | R^*(s_0, s)\}$$

An *invariant* $i(s)$ of the system is a proposition that is true in every accessible state:

$$i(s) \text{ is an invariant} \Leftrightarrow \forall s \in X : i(s)$$

Many important algorithm safety properties can be phrased as invariants. For instance, partial correctness can be phrased as an invariant property nearly directly from its definition, by asserting that the preconditions P holding for the initial state implies that the postconditions R hold for the final state once the algorithm has terminated:

$$\forall s \in X : \pi = s_{\text{exit}} \Rightarrow (P(s_0) \Rightarrow R(s_{\text{exit}}))$$

Similarly, mutual exclusion can be captured by the assertion that one processor executing the critical section C implies that no other processor can be in the same section:

$$\forall s \in X : (\pi_a = C) \Rightarrow \neg\exists\pi_b : \pi_b = C \wedge a \neq b$$

However, invariants cannot express termination and liveness properties, and do not lead to a definition of correctness that's applicable to programs that run indefinitely.

To solve both of these problems, Pnueli introduces the idea of temporal implication, which he refers to as *eventuality*. These implications take the form $P \leadsto R$, and assert that the predicate P being true in one state will lead to R becoming true in a later state:

$$P \leadsto R \Leftrightarrow \forall t_1 \exists t_2 : (t_2 \geq t_1) P(t_1, \phi) \Rightarrow R(t_2, \psi)$$

For instance, demonstrating that an algorithm terminates is equivalent to proving that every processor eventually reaches an exit from its initial state:

$$(\pi_i = s_0) \leadsto (\pi_i = s_{\text{exit}})$$

With the logical framework for specifications in place, Pnueli examines possible verification methods for the temporal logic.

To prove safety properties, Pnueli uses Burstall's approach of *state induction* [54]: by showing that a property ϕ holds for the initial state s_0, and then showing that every transition from a state where ϕ is true leads to another state where ϕ holds, it is possible to show by induction that ϕ holds for every possible state of the system.

To prove liveness properties, Floyd's method of guaranteeing loops' termination by making them dependent on monotonically decreasing functions is shown to be applicable for temporal logic as well. It is also noted that the approach can be extended to work with any kind of well-founded set [55].

However, the main body of the paper is devoted to a third proof method: constructing a set of axioms for reasoning around eventuality, and using them to construct deductive proofs. Pnueli argues that this approach leads to more intuitive proofs.

$$\begin{aligned} &\textbf{Axiom A1} \\ \forall s, s^1 \quad p(s) \wedge R(s, s^1) \rightarrow q(s^1) &\Rightarrow p \leadsto q \\ &\textbf{Axiom A2} \\ p \rightarrow q &\Rightarrow p \leadsto q \end{aligned} \tag{2.7}$$

Figure 2.2: The axioms of Pnueli's temporal logic

$$\begin{aligned} &\textbf{Inference rule R1} \\ p \leadsto q, \forall s, s^1 \quad r(s) \wedge R(s, s^1) \rightarrow r(s^1) &\Rightarrow (p \wedge r) \leadsto (q \wedge r) \\ &\textbf{Inference rule R2} \\ p \leadsto q, q \leadsto r &\Rightarrow p \leadsto r \\ &\textbf{Inference rule R3} \\ p_1 \leadsto q, p_2 \leadsto q &\Rightarrow (p_1 \vee p_2) \leadsto q \\ &\textbf{Inference rule R4} \\ p \leadsto q &\Rightarrow (\exists u p) \leadsto q \end{aligned} \tag{2.8}$$

Figure 2.3: The inference rules of Pnueli's temporal logic

Pnueli suggests a minimal set of axioms and proof rules to this end, shown in Figure 2.2 and 2.3 respectively.

The axioms establish the link between logical and temporal implication: the first axiom states that $p \leadsto q$ can be shown by proving that every transition from a state where p holds leads to one where q is true, while the second axiom establishes that all logical implications are also temporal implications.

The rules are then used to derive more complex eventualities from simpler ones: the first proof rule extends the state induction approach to eventualities, showing that an inductive invariant will also hold across an eventuality. The second rule captures Hoare's composition rule, allowing a larger proof to be composed from multiple smaller eventualities. In a similar fashion, the third rule allows two eventualities with the same consequent to be joined with a disjunct.

Pnueli found that this axiomatic system was sufficient to decide the validity of any eventuality in a finite-state system [56]. In other words, any liveness property that could be expressed as a temporal implication could be verified within the logic.

However, Pnueli found the system “weak”: while it was technically possible to express safety and liveness properties through temporal formulas, doing so turned out be challenging in practice.

Subsequent temporal logics attempted [57] to solve this problem by introducing additional *temporal operators* to allow more complex propositions about sequences of states. In 1981, Pnueli presented [58] a linear time logic (LTL) for program analysis featuring these operators, allowing propositions $\Box P$ over *every* state, $\Diamond P$ for *eventual* states, and $\circ P$ for the *next* state of a behavior.

In 1982, a paper by Clarke and Emerson [59] and another by Queille and Sifakis [60] independently proposed formal program analysis based on another class of temporal logic, known as computation tree logic (CTL). Unlike LTLs, which view systems as the set of all of their possible executions [30], CTL logics consider the branching structure of executions, allowing propositions about possible states, e.g. “along some future [execution], P eventually holds” [16].

In their paper, Clarke and Emerson proposed *model-checking* specifications. By expressing the system behavior itself as a temporal formula, it was possible to generate a finite-state *model* of the system. Instead of carrying out deductive proofs to determine whether a property is satisfied, this approach allowed mechanical verification of properties by checking them against each state in the model.

2.3 Temporal logic of actions

In 1994, Lamport published a summary of his *temporal logic of actions* [33] (TLA), an LTL logic oriented around the concept of *actions*.

Actions are logical propositions over pairs of states, which represent changes in the system state. Actions include *primed* and unprimed variables, which respectively represent the state after and before the action. For instance, the action

$$\mathcal{A} \triangleq x' = x + 2 \tag{2.9}$$

states that performing the action $\mathcal{A}$ should lead to a state where x is equal to its value in the prior state plus 2.

A pair of states (s, t) is an $\mathcal{A}$ step if state t can be reached by performing the action $\mathcal{A}$ in state s. If there exists a state t such that (s, t) is an $\mathcal{A}$ step, $\mathcal{A}$ is said to be an enabled action in state s. Actions serve a similar role to triples in Floyd-Hoare logic, but do not include any operations or program segments, as they are fully specified by their pre- and postconditions.

Similar to Pnueli's temporal logic, system executions are described as sequences of states, known as *behaviors*. However, unlike Pnueli's logic, every TLA behavior is infinitely long: the termination of an algorithm is represented as an infinite sequence of *stuttering* states, where no variables change. As an LTL logic, TLA views systems as the set of all of their possible behaviors.

Unlike Floyd-Hoare logic, TLA represents both programs and properties as temporal formulas. TLA conventionally represents programs as a conjunction

$$Init \land \Box[Next]_{Vars} \land F \tag{2.10}$$

where

- $Init$ is the *initial state predicate*, a proposition which holds for every state that is a valid initial state for the system.
- $\Box[Next]_{Vars}$ is the *next-state relation*, which states that every state following the initial state must either satisfy the next-state action $Next$ or be a stuttering state:

$$\Box[Next]_{Vars} \triangleq \Box(Next \lor (Vars' = Vars)) \tag{2.11}$$

- $Vars$ is the set of variables required to specify the system. Variables in TLA have no types, and can assume any value. Type safety is a safety property of a specification, not a syntactic requirement.
- F is the *fairness condition* of the specification.

To verify that a program satisfies a particular property, we verify that it satisfies a particular temporal formula. Safety properties are verified by asserting that the program satisfies a state predicate P for every state, using the *always* operator $\Box$:

$$P \text{ is always true } \Leftrightarrow \Box P \tag{2.12}$$

Liveness properties are expressed through the *eventually* operator $\Diamond$, which is defined in terms of the always operator, as the assertion that the negation of a property does not hold forever [61]:

$$\Diamond F = \neg\Box\neg F \tag{2.13}$$

For instance, Pnueli's temporal implication is represented by *leads-to* $\leadsto$ formulas, which assert that one proposition being true always eventually lead to a state where another is true:

$$F \leadsto G = \Box(F \Rightarrow \Diamond G) \tag{2.14}$$

It is possible to assert that a proposition eventually holds and is true in *every* subsequent state, through *eventually-always* $\Diamond\Box$ formulas

$$\Diamond\Box F = \Diamond(\Box F) \tag{2.15}$$

It is also possible to state that propositions are true infinitely often with *always-eventually* formulas

$$\Box\Diamond F = \Box(\Diamond F) \tag{2.16}$$

To prove liveness properties, specifications must guarantee progress: by itself, a next-state relation $\Box[Next]_{Vars}$ can be satisfied by an infinite series of stuttering states. While Pnueli's logic guarantees progress by assuming some form of fair scheduling for processes, TLA specifies liveness through *weak* and *strong* fairness conditions.

Weak fairness conditions state that actions that are perpetually enabled must eventually happen:

$$WF_f(\mathcal{A}) \triangleq (\Box\Diamond\langle\mathcal{A}\rangle_f) \lor (\Box\Diamond\neg Enabled\langle\mathcal{A}\rangle_f) \tag{2.17}$$

Strong fairness conditions require actions that are enabled infinitely often to happen infinitely often as well:

$$SF_f(\mathcal{A}) \triangleq (\Box\Diamond\langle\mathcal{A}\rangle_f) \lor (\Diamond\Box\neg Enabled\langle\mathcal{A}\rangle_f) \tag{2.18}$$

Both weak and strong fairness conditions allow an arbitrary number of stuttering steps. There is no way to write a formula hinging on whether a stuttering step has occurred, ensuring that TLA$^+$ specifications are *stuttering invariant*: safety and liveness properties will hold for a specification regardless how many stuttering steps are inserted. Stuttering invariance makes it possible to represent concurrent programs as conjunctions of multiple specifications:

$$\begin{aligned} P_1 &\triangleq \mathit{Init}_1 \land \Box[\mathit{Next}_1]_{\mathit{Vars}_1} \land F_1 \\ P_2 &\triangleq \mathit{Init}_2 \land \Box[\mathit{Next}_2]_{\mathit{Vars}_2} \land F_2 \\ P_1 \land P_2 &\triangleq \mathit{Init}_1 \land \mathit{Init}_2 \land [\mathit{Next}_1 \lor \mathit{Next}_2]_{\langle \mathit{Vars}_1, \mathit{Vars}_2 \rangle} \land F_1 \land F_2 \end{aligned} \tag{2.19}$$

The composition $P_1 \land P_2$ of the programs P_1 and P_2 is a new system where each step obeys the next-state action of P_1 or P_2, and both programs' initial states and fairness conditions are preserved.

2.4 Summary

The idea of using formal logical reasoning to verify programs' correctness is nearly as old as computer science itself, with early computing pioneers like Turing and von Neumann outlining their own procedures for proving that programs had the desired result.

Floyd and Hoare proposed using propositional logic to reason about pre- and post-conditions of operations, and composing smaller proofs about individual operations into larger proofs over programs.

Later approaches adopted temporal logic to reason about concurrent programs, and representing program executions as sequences of states. Pnueli [45] proposed using LTL logic to construct deductive proofs over programs involving temporal implication, while Clarke and Emerson [59] proposed model-checking based on CTL logic.

Lamport's TLA is an untyped LTL temporal logic which attempts to minimize the amount of temporal reasoning, in favor of reasoning about actions. In the logic, programs and properties are both expressed as temporal formulas. TLA specifications are stuttering invariant, allowing composition of specifications.

3 The Chord protocol

This chapter provides a summary of the Chord distributed hash table protocol, which we will later formally specify and verify. We also provide an outline of Zave's [25] analysis of the protocol, which identified inconsistencies and flaws in the Chord protocol, and forms the basis for our formal analysis.

We base this description of Chord on the 2001 paper by Stoica et al. [26], which appeared in SIGCOMM, their 2001 MIT technical report [27], and their revised 2003 paper [28] in *Transactions on Networking (TON)*, as well as the 2002 analysis by Liben-Nowell et al. [36] in *Principles of Distributed Computing (PODC)*. Unless otherwise noted, we use the 2003 TON paper as the primary source for algorithm pseudocode.

The Chord protocol implements a distributed hash table, using consistent hashing [62] to distribute key-value pairs across the members of the network, which organize themselves in a ring structure.

The stated design goals [26, 28] of the protocol are

- *Load balancing*: responsibility for keys is distributed evenly across nodes to prevent undue load on a few servers.
- *Decentralization*: the protocol is fully peer-to-peer, with no central authorities or special responsibilities conferred to certain nodes.

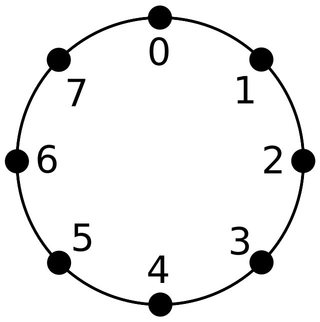


Figure 3.1: An example of the identifier space with $m = 3$, yielding $2^3 = 8$ potential key and node identifiers.

- *Scalability*: the time to perform a lookup scales logarithmically with the number of nodes in the network, and each node only needs to keep track of a subset of nodes.
- *Availability*: lookups should function in spite of nodes joining and leaving the network.
- *Flexibility*: there are no limits or constraints on which values can be used as keys.

3.1 The identifier space

The key to achieving these properties is mapping keys to a flat *identifier space*. An identifier space (alternatively *identifier circle*) I is a finite group of the natural numbers modulo 2^m, with a partial ordering [63] over the identifiers. Figure 3.1 shows an example of an identifier space with $m = 3$.

The same identifier space is employed for both key-value pairs and members of the network: key-value pairs are assigned *key identifiers* by hashing the key, and nodes are assigned *node identifiers* by hashing their network address. To hash keys and addresses, Chord implements a *consistent hash* function by hashing the key with the SHA-1 [64] hash function, and interpreting the first m bits of the hash as an unsigned integer, yielding numeric identifiers in the range $[0, 2^m - 1]$.

Chord then assigns any given key k to the first node whose identifier is greater than or equal to the key's identifier, referring to it as the *successor node* $successor(k)$ of the key k. To illustrate the relationship, Stoica et al. use the analogy of finding the key on the identifier circle, and tracing the circle

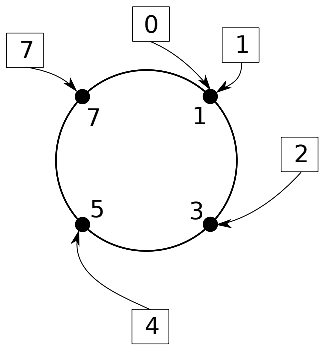


Figure 3.2: Keys (square) are assigned to their successor nodes (circles), which have identifiers following them in the identifier order.

clockwise until the first node is found.

This relationship is continuously reevaluated as nodes join and leave the network. A new member may have a node identifier closer to a key's identifier than its previous successor, in which case responsibility for the key must be transferred to the new node. Similarly, a node leaving the network must first delegate its keys to another node.

The consistent hashing limits the extent of these changes. After joining, a node only needs to contact a single node to receive the keys it is responsible for. Conversely, a leaving node only has to assign its keys to one other node. This allows nodes to enter and leave the network with minimal disruption.

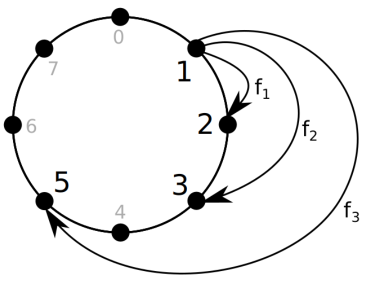
The main links between the nodes themselves are the *successor pointers* maintained by each node, which point to the nodes following them in the identifier order, leading to the characteristic ring structure. Then, lookups can be implemented in a recursive fashion, as shown in Figure 3.3, such that nodes only need to maintain the key-value pairs they are the successors of.

However, under this algorithm, the lookup time scales linearly with the size of the network: at worst, a query might lead to a successor node *preceding* the initial n, meaning it has to traverse every node in the network. To achieve logarithmic lookup time, Chord introduces a *finger table* for each node. The finger tables essentially act as a skip list [65] over the ring, allowing lookups to skip intermediate nodes which cannot be the successor of the key.

For each node n, the finger table consists of a set of *finger pointers* f_i given by $f_i = successor(n + 2^{i-1})$, as in Figure 3.4. By offsetting each pointer with powers of 2, the lookup algorithm in Figure 3.5 can halve [27] the distance to the target identifier, leading to a logarithmic lookup time.

```
// Queries node $n$ for the successor of the key $id$.
function N.FIND_SUCCESSOR(id)
    if $id \in (n, successor]$ then
        return $successor$
    else
        // Forward the query to the successor of $n$.
        return $successor.find\_successor(id)$
    end if
end function
```

Figure 3.3: Simplified pseudocode for retrieving the successor node of a given identifier (from Figure 3a of [28])



Finger table for node 1

1	f_1=successor($1+2^0$)=2
2	f_2=successor($1+2^1$)=3
3	f_3=successor($1+2^2$)=5

Figure 3.4: An example of a finger table.

```
// Queries node $n$ for the successor of the key $id$, using the finger table.
function N.FIND_SUCCESSOR(id)
    if $id \in (n, successor]$ then
        return $successor$
    else
        n' = $closest\_preceding\_node(id)$
        return $n'.find\_successor(id)$
    end if
end function

// Attempts to find the closest predecessor for $id$ in node $n$'s finger table.
function N.CLOSEST_PRECEDING_NODE(id)
    for $i = m$ downto $1$ do
        if $finger[i] \in (n, id)$ then
            return $finger[i]$
        end if
    end for
end function
```

Figure 3.5: Optimized algorithm for retrieving successor nodes with finger tables (from Figure 5 of [28])

The papers differ on how the tables should be initialized and maintained. The original technical report and the SIGCOMM paper [27, 26] suggest initializing every finger pointer after joining the network, and updating finger tables mainly as part of other network events. Nodes periodically verify that finger pointers are valid by issuing *find-successor* queries.

The 2003 TON paper [28] proposes not looking up fingers when joining the network, but instead initializing finger tables with empty[1] pointers. Nodes then gradually fill in the tables by scheduling their own *find-successor* queries. In effect, this scheme leads to a form of eventual consistency for the finger tables.

3.2 Establishing and maintaining the network

To establish and maintain the network, Chord requires nodes to maintain *predecessor* pointers, which point to the nodes immediately preceding them in

1. If a finger pointer is missing during a lookup, the table falls back to the closest available finger.

```
// Establishes a new Chord network in node $n$.
function N.CREATE($n'$)
    $predecessor = nil$
    $successor = n$
end function

// Has node $n$ attempt to join the network node $n'$ is part of.
function N.JOIN($n'$)
    $predecessor = nil$
    $successor = n'.find\_successor(n)$
end function
```

Figure 3.6: The pseudocode for joining or establishing a Chord network (Figure 6 of [28])

the identifier order. The network maintenance algorithms ensure that every node's successor pointer is eventually correct, so that every node is considered the successor of exactly one other node [26].

Initially, the network is established by a single node setting itself as its successor, as in Figure 3.6, creating a single-node ring. Then, subsequent nodes join the network by calculating their node identifiers, polling a known member of the network for the successor node of their identifier, and setting it as their initial successor.

Once the node has found its initial successor, it starts integrating itself into the network, and gradually brings the network closer to an ideal state. This is achieved through the *stabilize* and *notify* algorithms, shown in Figure 3.7. Nodes run *stabilize* periodically, querying their successor for its predecessor. If the successor's predecessor lies closer in identifier order, it is adopted as the node's new successor. After stabilizing, the node *notifies* its new successor of its existence. Upon receiving the notification, the successor adopts the node as its predecessor if it lies closer than the current predecessor.

3.3 Fault-tolerance

Chord assumes a *fail-stop* failure model [36], where nodes function correctly until some point in time, where they stop servicing requests and are not heard from again. Communication is assumed reliable [25], with no messages dropped or altered in transit. The network availability is also assumed to be

```
// Notifies $n$'s successor of $n$'s existence, and determines whether
// its predecessor is a better candidate for successor.
function N.STABILIZE()
    $x = n.successor.predecessor$
    if $x \in (n, n.successor)$ then
        $n.successor = x$
    end if
    $n.successor.notify(n)$
end function

// Signals to node $n$ that $n'$ may be its predecessor.
function N.NOTIFY($n'$)
    if $n.predecessor = nil \vee n' \in (n.predecessor, n)$ then
        $n.predecessor = n'$
    end if
end function
```

Figure 3.7: The pseudocode for stabilization and notification protocols, which gradually improve the network topology.

symmetric [28] — if node 1 can send messages to node 2, then node 2 can message node 1 in return — and *transitive*: if node 1 can message 2 and node 2 can message 3, then node 1 can message node 3.

An important part of this failure model is the assumption that nodes can reliably detect whether another node has failed. In their simulation of the protocol, Stoica et al. [28] implement failure detection through a timeout mechanism, marking a node as failed if it does not respond to any queries within a set time.

The key assumption Stoica et al. make in their fault-tolerance design is that only the nodes' successor pointers have to be correct for a lookup to make progress [26]. The predecessor pointer is only used for network maintenance, and errors in the finger table can be tolerated by falling back to the successor pointers.

Thus, the main responsibility of the fault-tolerance algorithms is to handle the case where a node's immediate successor fails. Chord solves this by replacing nodes' successor pointers with *successor lists*, so that each node knows about the r next nodes in the identifier order, not just the first.

Then, once a node discovers that its successor has failed, it discards it from

```
// Replace a failed immediate successor (assumed to be $n.successors[1]$)
// with the nearest live node from the list.
function N.FIX_SUCCESSOR()
    if $n.successors[1]$ has failed then
        $u$ = the smallest $u$ such that $n.successors[u]$ is live
        $n.successors = \langle s_u, s_{u+1}, \cdots, s_r \rangle$
    end if
end function

// Remove a failed predecessor from node $n$.
function N.FIX_PREDECESSOR()
    if $n.predecessor$ has failed then
        $n.predecessor = nil$
    end if
end function

// Adopt the successor list from node $n$'s immediate successor.
function N.FIX_SUCCESSOR_LIST()
    $\langle s_1, s_2, \cdots, s_r \rangle = n.successors[1].successors$
    $n.successors = \langle n.successors[1], s_1, s_2, \cdots, s_{r-1} \rangle$
end function
```

Figure 3.8: Pseudocode for the Chord fault-tolerance algorithms presented by Stoica et al., from Figure 4 of the PODC analysis [36].

its successor list, and adopts the next node in the list as its successor. Nodes also periodically poll their predecessor to determine whether it is alive: if it has failed, the node marks itself as no longer having a predecessor.

In order to construct the successor lists, each node periodically polls its successor for its successor list. On receiving it, the node prepends its successor to the list, and adopts it as its new successor list.

Neither the technical report nor the SIGCOMM paper provide pseudocode for these algorithms, instead opting for informal outlines. Oddly, the TON paper provides pseudocode only for the predecessor checking routine, and gives a similar outline for the other two algorithms. Thus, we have to look to the PODC analysis for the fault-tolerance algorithm pseudocode in Figure 3.8.

The lookup, network maintenance and fault-tolerance algorithms collectively make up the Chord protocol.

3.4 Zave's analysis

Having described the Chord protocol, we now give a brief outline of its analysis by Zave [25]. "Inconsistencies and ambiguities" in the claimed invariants and a "mix [of] correctness with performance analysis" in the proofs lead her to question the protocol's correctness, prompting a formal analysis of the protocol. To formally specify the protocol, Zave uses the Alloy [66] specification language, which models object structures and relationships with first-order relational logic [67] extended with transitive closures.

Zave notes that the goal of the protocol is essentially a form of eventual consistency for the pointers between nodes, or "eventual reachability": eventually, all errors in the ring structure is repaired, and every node can contact every other node through lookups. Errors in the protocol could lead to certain nodes becoming unreachable, leading to permanent partitioning in the network.

However, it is unclear which safety properties must be satisfied for eventual reachability to be possible, due to the proofs in the technical report [27] resting on probabilistic arguments. To determine which safety properties are critical to the network's correctness, Zave examines Definition 5.6 of the analysis by Liben-Nowell et al. [36], and excludes probabilistic and quantitative statements about the protocol, leaving five claimed safety properties:

- **Part (1)**: "the network is connected", which Liben-Nowell et al. define as "there is a path using successor lists and finger tables connecting any two nodes".
- **Part (4a)**: "the cycle is non-loopy" (i.e. ordered over the identifiers)
- **Part (4b)**: "for every node v in the appendage $\mathcal{A}_u$ [rooted in a node u that is part of the cycle], the path of successors from v to u is increasing."
- **Part (5c)**: "if node v is in the appendage $\mathcal{A}_u$, then u is the first live cycle node following v."
- **Part (5d)**: "if the successor list of [the successor of node u] skips over a live node v, then v is not in the [successor list of node u]".

The original statement of the first property is quickly ruled out, since it only holds for networks that are already in an ideal state: any network with an appendage — which isn't reachable from any member of the main ring — would be disqualified. Instead, the specification expresses network connectivity as a combination of three smaller properties:

- **Part (1a)**: There is at least one ring.
- **Part (1b)**: There is at most one ring.
- **Part (1c)**: Every node is either part of the ring or an appendage of it.

Each of these properties are introduced into the protocol specification, as well as fail-stop errors like those assumed by Stoica et al. [28]. After formally verifying the specification, Zave finds that the protocol satisfies none of the claimed properties. For every proposed safety property[2], the protocol allows a corresponding sequence of network events which violates it.

3.5 Summary

The Chord protocol implements a distributed hash table by assigning identifiers to key-value pairs and network members from a flat identifier space. To look up a key-value pair, nodes recursively query each other to determine which node is responsible for a given key's identifier. Chord uses consistent hashing to assign identifiers, which lets a minimal number of keys change hands when a node enters or leaves the network.

Chord attempts to establish eventual reachability, so that every node in the network eventually becomes part of one ring. To do so, Chord nodes attempt to improve the network topology gradually, by periodically querying their neighbors for nodes that might be better neighbor candidates.

The protocol assumes a fail-stop failure model with reliable communication between nodes. To implement fault-tolerance, Chord nodes maintain lists of the nodes immediately following them in the identifier space, and periodically request updates to the list from their neighbors. This allows nodes to recover from the failure of their immediate neighbors.

In analyzing the Chord protocol, Zave finds inconsistencies in its definition, and questions its correctness. In formally verifying the protocol, she discovers that it does not satisfy any of its claimed safety properties.

2. (4a) could be satisfied by the "strong stabilization" algorithm outlined by Liben-Nowell et al. [36], which claims to restore identifier ordering. However, this scheme is "far slower" than the original protocol, and ultimately dismissed as implementations using it "might not perform well enough to be useful".

4 Specification

This chapter outlines the process of formally specifying the Chord protocol, as an example of how formal methods can be used to uncover flaws in existing systems. Our method is based on Zave's [25] work in uncovering protocol flaws in the Chord [26] peer-to-peer distributed lookup protocol.

While Zave employed the Alloy specification language [66], we will specify the protocol in TLA$^+$ — using the TLC model checker (TLC) [34] to verify our specification. Since Alloy builds on a relational rather than a temporal logic, it does not have a built-in notion of time, which means that Zave's specification has to explicitly express how events are correlated in time [68]. In contrast, TLA$^+$ makes time implicit, potentially making it easier to describe systems that are dynamic over time [69].

We will begin by formulating a synchronous version of the Chord specification, where no failures occur and only one event takes place at a time. This simpler statement of the protocol allows us to determine appropriate abstractions, and lets us model important safety invariants. Next, we revise the specification to introduce asynchronous messaging, letting the specification capture multiple concurrent network events. Finally, we introduce fail-stop failures and the Chord fault-tolerance mechanisms into the specification, completing the specification. We will review Zave's 2012 analysis [25], and show that our specification admits her counterexamples to the claimed safety invariants presented in [36].

4.1 Goals

Before we start the work of formally specifying the protocol, we must decide which properties we want to demonstrate through the formal analysis.

As TLA$^+$ is designed for checking safety and liveness properties, we disregard probabilistic and quantitative claims about the protocol, like the upper bounds on time complexity promised by the SIGCOMM Chord paper [26]. As outlined in Section 1.2, we adopt the same scope for our specification as Zave does. Zave omits finger tables because they are "not relevant to correctness" [25], and does not model the storage and retrieval of keys. This limits the scope of our specification to the core events of joining, network maintenance, and fault-tolerance.

This leaves us with four main requirements:

- *Type safety*: at no point should the protocol permit nodes to have successor or predecessor pointers linking to invalid destinations, or nodes outside the identifier space.
- *Deadlock freedom*: until the network is in the ideal state, there should always be at least one node with an enabled action allowing the protocol to proceed.
- *Network safety*: the protocol should not allow actions that partition the network, or leave it in a state where it cannot become ideal.
- *Network liveness*: after all nodes have finished joining, the protocol should eventually have arranged the network into an ideal state.

With these requirements in mind, we can decide on the appropriate abstractions for our specification.

4.2 Assumptions and abstractions

To model the protocol, we must first decide on the specification's level of atomicity. For our initial version of the specification, we make joining, stabilizing and notifying completely atomic, allowing only one event to proceed at any given time.

Enforcing this assumption in practice would seriously limit the protocol's ability to deal with *churn* [70, 71] from multiple nodes independently arriving and leaving. It would also require a central authority to govern progress, contrary to the initial design goal of a peer-to-peer system with no nodes being assigned any special authority. However, this initial abstraction allows us to write a specification close to the proposed pseudocode, which can be progressively verified while iterating towards more granular actions. In our final specification, we only allow actions that change the state of at most one node at a time, and model asynchronous message passing between nodes.

Once we have demonstrated that the protocol functions in the *pure-join* case with no failures, we introduce the failure model used by Stoica et al. and Zave (see Section 1.2) with fail-stop failures as part of the specification.

To make type safety and communication easier to model, we will treat the process identifier, the node's identifier and its network address as one and the same, as Zave does. In the specification, each of the N nodes is identified by an integer in the range $1 \ldots N$. We assume that nodes learn each others' network addresses as part of exchanging identifiers.

This simplification is reasonable so long as we are certain no two nodes will receive the same identifier, which demands that the consistent hashing scheme used:

- Has a large enough identifier space that a large number of participants do not make identifier collisions significantly more likely.
- Distributes identifiers sufficiently uniformly that nodes are unlikely to get similar identifiers, even if they are geographically close.

The SIGCOMM Chord paper justifies both of these assumptions with a probabilistic argument, arguing that SHA-1 is a cryptographic hash with good distributional properties that satisfy the latter requirement. While the SHA-1 hash function is no longer recommended [72] due to its vulnerability to fixed-prefix collision attacks, Chord only uses it to provide unique identifiers, and does not rely on any of its cryptographic properties.

One notable consequence of treating the node identifier as its network address is that each node is only allowed to run one instance of the stabilization protocol, since we specify network actions as operations on a single identifier. This is not important for our particular analysis, but precludes modelling any schemes with multiple virtual participants per node, such as the load balancing scheme described in Section 6.2 of [26].

Next, to determine the necessary conditions for maintaining the network safety invariants, we start by examining only the initial network configurations where

```
──────────────── MODULE SynchronousChord ────────────────
 Models synchronous Chord with no node failures and synchronous communication.

EXTENDS Integers, FiniteSets, Sequences, TLC
 The model uses a single constant: the number of nodes N.
CONSTANTS N
ASSUME FiniteNodesAssumption ≜ N ∈ Nat \ {0}

 Use ascending natural numbers for the node identifiers.
ProcSet ≜ 1 . . N

VARIABLES Successor,
    Predecessor,
    HasJoined,
    HasPredecessor
vars ≜ ⟨Successor, Predecessor, HasJoined, HasPredecessor⟩

├─────────────────────────────────────────────────────────┤

Init ≜
    ∧ Successor = [self ∈ ProcSet ↦ 1]
    ∧ Predecessor = [self ∈ ProcSet ↦ 1]
    ∧ HasJoined = [self ∈ ProcSet ↦ self = 1]
    ∧ HasPredecessor = [self ∈ ProcSet ↦ self = 1]
└─────────────────────────────────────────────────────────┘
```

Figure 4.1: The variables and initial state for the synchronous Chord specification.

nodes join an existing single-node network, and then gradually loosen the constraints to determine which invariants the initial state must satisfy.

Finally, since we are checking a liveness property — whether the stabilization algorithm leads to an ideal network configuration — our specification must guarantee that at least one node is making progress at any given time. We encode this in the specification through a weak fairness progress condition.

4.3 Specifying synchronous Chord

With these abstractions in place, we can outline an initial specification of *synchronous* Chord, where nodes do not fail, and only one network event — joining, stabilizing or notifying — takes place at a time.

The necessary variables and initial state for this specification are shown in Figure 4.1. The only constant that needs to be supplied at model-checking time is the number of nodes N.

Since nodes cannot fail, the only variables needed to specify the protocol are

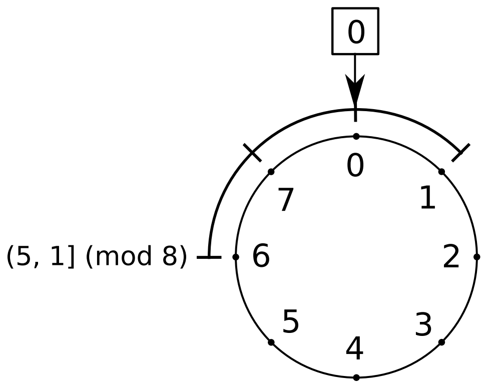


Figure 4.2: Intervals of Chord identifiers can straddle zero, requiring special checks.

a single successor and predecessor pointer for each node, which we model as functions from the node identifier to the pointer destination. While the Chord papers use *nil* to represent uninitialized successor/predecessor pointers, our specification uses separate Boolean variables. A type invariant verifies whether the successor and predecessor point to valid identifiers (see *Safety invariants*).

In our first version of the synchronous specification, the initial state *Init* has node 1 forming a single-node ring, with every other node attempting to join its network. Once we have verified the specification for this case, we will expand the set of permissible initial states.

4.3.1 Dealing with identifier order

The first obstacle to a formal specification of the protocol is the identifier space. Recall that the identifier space spans the integers modulo 2^m, forming a finite cyclic group. The protocol depends heavily on an intuitive understanding of intervals over this space. For instance, the *find-successor* algorithms in Figure 3.3 and 3.5 both require determining whether the identifier id lies inside the interval $(n, successor]$.

For the natural numbers, this test is trivially implemented as

$$id \in (n, successor] \Leftrightarrow id > n \land id \leq successor \tag{4.1}$$

but this fails when applied to the identifier space, since the identifier ordering is only partial.

Using our earlier circle with $m = 3$, we imagine an interval from node 5 to node 1, with a single key 0 assigned to node 1. From the protocol definition, it is clear that key 0 belongs to node 1, since it lies in the interval $(5, 1]$, as shown in Figure 4.2. However, since $0 \geq 5$, we find that $id \geq n$, and the test in Equation 4.1 fails.

Since TLA$^+$ does not provide any standard modules for reasoning about intervals in finite cyclic groups, we use the method described by Giesen [73], whose key insight is that interval membership is essentially symmetric. If a point lies inside an interval in a cyclic group, the point and interval endpoints can be offset by any amount, with the point remaining inside:

$$\begin{aligned} x \in [a, b](\text{mod } N) \Rightarrow \\ (x + i) \in [a + i, b + i]\ (\text{mod } N)\ \forall i \in [0, N - 1] \end{aligned} \tag{4.2}$$

This allows the problem to be restated as

$$\begin{aligned} x \in [a, b](\text{mod } N) = \\ (x - a) \in [a - a, b - a](\text{mod } N) = \\ (x - a) \in [0, b - a](\text{mod } N) \Rightarrow \\ (x - a)\ (\text{mod } N) \leq (b - a)\ (\text{mod } N) \end{aligned} \tag{4.3}$$

which can be expressed in TLA$^+$. This solution does assume that the result of $x\ (\text{mod } N)$ is positive, even for negative x; thankfully, this definition of modulo is both the one outlined in *Specifying Systems* [4] and the one implemented by the TLC model checker.

A subtle pitfall with this solution arises from the interpretation of intervals of the form $(a, a)\ (\text{mod } N)$. From the definition of an open interval $(a, b) = \{x : a < x < b\}$, we expect (a, a) to be the empty set, since no x satisfies $a < x < a$. If we extend the earlier test for closed intervals from Equation 4.3 to open ones by excluding the interval's endpoints, e.g.

$$\begin{aligned} x \in (a, b)(\text{mod } N) = \\ (x - a)\ (\text{mod } N) \leq (b - a)\ (\text{mod } N) \land x \neq a \land x \neq b \end{aligned} \tag{4.4}$$

we see that no x can lie inside (a, a), as it is either the case that $(x - a) > 0$, in which case it is greater than $(b - a) = 0$, or $(x - a) = 0$, which implies that $x = a$.

However, this interpretation leads to trouble when specifying Chord actions for small networks. For instance, consider a Chord ring consisting of a single node,

```
──────────────── MODULE SynchronousChord ────────────────
BetweenInclusive is true if x ∈ [a, b] (mod N).
BetweenInclusive(x, a, b) ≜ ((x − a)%N) ≤ ((b − a)%N)

BetweenHalfOpen is true if x ∈ (a, b] (mod N).
BetweenHalfOpen(x, a, b) ≜
    ∧ ((x − a)%N) ≤ ((b − a)%N) ∧ x ≠ a

BetweenExclusive is true if x ∈ (a, b) (mod N)
BetweenExclusive(x, a, b) ≜
    ∨ ∧ ((x − a)%N) ≤ ((b − a)%N) ∧ x ≠ a ∧ x ≠ b
     Caveat: in the case where a = b, as in a one-node ring,
    we want to interpret (a,b) as spanning the entire interval:
    ∨ ∧ a = b
──────────────────────────────────────────────────────────
```

Figure 4.3: TLA$^+$ operators for determining whether an identifier lies inside an identifier interval.

such that its successor and predecessor pointers both point at itself. To query it for the successor of a key, using the algorithm from Figure 3.3, we must first determine whether the key id lies in the interval $(n, successor]$.

Since the node is its own successor, the resulting interval is $(n, n]$, which we established was empty. Therefore, no key can be inside it, and the node must forward the query to itself, leading to an infinite loop.

We could resolve this by introducing special cases for single-node networks in the specification, at the cost of a longer and more complex specification. Instead, we choose to treat open intervals (a, a) as encompassing every identifier, leading to the TLA$^+$ operators in Figure 4.3. This solves the problem outlined above, as every key lying in the interval $(n, n]$ means that a single-node ring will correctly point to itself as the successor node of all keys.

4.3.2 Defining connectivity

Reachability is key to the Chord protocol. Since each node only has knowledge of a subset of the network, the stabilization protocol must ensure that there is a path to any node in the ring using only finger tables and successor lists.

Thus, we would like to formally specify which nodes are reachable from any given node, both in order to help us define network safety invariants, and to ensure we do not "cheat" and permit actions that rely on state the node cannot know about.

```
MODULE SynchronousChord
ReachableSuccessors(x) ≜
    LET RECURSIVE Reachable(_, _)
        Reachable(n, i) ≜
            IF $i = 0 \lor \neg HasJoined[n]$
             THEN {}
             ELSE Reachable(Successor[n], i − 1) ∪ {Successor[n]}
    IN  Reachable(x, N)
```

Figure 4.4: A TLA+ operator determining which nodes are reachable by following successor pointers from a given node x.

Since we initially only use a single successor pointer, specified as a relation from the node's identifier to the destination node's identifier, we define the set of nodes *reachable* from node n to be the set

$$Reachable(n) = \{n, successor(n), successor(successor(n)), \cdots\} \qquad (4.5)$$

We specify this in TLA$^+$ through a recursive function, seen in Figure 4.4. We use the LET RECURSIVE-IN pattern to define the recursive function, as TLA$^+$ does not allow operators to be directly defined recursively [4]. Additionally, to avoid evaluating recursive operators for potentially infinite sets, TLC requires a finite bound on the recursion. Since the number of nodes N is known, we know that at most N nodes can be reached, and we limit the function to N iterations.

4.3.3 Actions

With these fundamental definitions in place, we can now begin our first attempt to specify the actions making up the protocol. By omitting communication from the initial specification, we can specify actions purely in terms of their pre- and postconditions:

- Joining the network leaves a node with the closest possible neighbor from the candidates currently in the ring as its initial successor.

- Stabilizations require that a node's successor's predecessor lies closer than its successor, and has the node adopt it as its new successor.

- Similarly, notifications require that the predecessor's successor lies closer than the current predecessor, and makes it the new predecessor.

```
MODULE SynchronousChord
$Join(self) \triangleq$
    The only precondition for joining is that the node has not joined
    a network before:
    $\land \neg HasJoined[self]$
    After joining, the node adopts the closest successor from the
    nodes that are currently in the ring:
    $\land HasJoined' = [HasJoined \text{ EXCEPT } ![self] = \text{TRUE}]$
    $\land Successor' = [Successor \text{ EXCEPT } ![self] =$
        $\text{CHOOSE } succ \in \{Successor[self]\}$
            $\cup ReachableSuccessors(Successor[self]) :$
            $\land \neg\exists\, candidate \in ReachableSuccessors(self) :$
                $\land BetweenExclusive(self, candidate, succ)$
                $\land candidate \neq succ]$
    $\land \text{UNCHANGED } \langle Predecessor, HasPredecessor \rangle$

$Stabilize(self) \triangleq$
    $\land HasJoined[self]$
    $\land HasPredecessor[Successor[self]]$
    Stabilizing requires that the successor's predecessor lies closer,
    $\land BetweenExclusive(Predecessor[Successor[self]], self, Successor[self])$
    and ends with it as the node's next successor:
    $\land Successor' = [Successor \text{ EXCEPT } ![self] = Predecessor[Successor[self]]]$
    $\land \text{UNCHANGED } \langle HasJoined, HasPredecessor, Predecessor \rangle$

$Notify(self) \triangleq$
    $\land HasJoined[self]$
    Notification likewise requires that the successor doesn't have a
    predecessor, or that this node lies closer:
    $\land \lor \neg HasPredecessor[Successor[self]]$
      $\lor BetweenExclusive(self, Predecessor[Successor[self]], Successor[self])$
    Notification ends with the successor adopting the node as its predecessor:
    $\land Predecessor' = [Predecessor \text{ EXCEPT } ![Successor[self]] = self]$
    $\land HasPredecessor' = [HasPredecessor \text{ EXCEPT } ![Successor[self]] = \text{TRUE}]$
    $\land \text{UNCHANGED } \langle Successor, HasJoined \rangle$
```

Figure 4.5: Initial TLA$^+$ specification of joining, stabilizing and notifying

These three actions, shown in Figure 4.5, turn out to be sufficient to specify the simplified stabilization protocol.

4.3.4 Safety invariants

With the specification of the protocol in place, we specify the properties we would like it to satisfy, beginning with the safety properties.

First, we verify the type safety of the specification. Since TLA$^+$ is based on an untyped logic, the conventional way of establishing type safety is by expressing it as an invariant and passing it to the model checker. The invariant *TypeOK* (see Appendix A) serves this purpose in our specification, verifying that successor and predecessor pointers only point to valid node identifiers.

Next, we would like to verify that the protocol is deadlock free. We do not need an explicit invariant to verify this, as TLC searches for deadlocks by default. With the action definitions of Figure 4.5, TLC reports that the protocol deadlocks once the network is in the ideal state. Since the network is ideal, there are no possible improvements to make, so none of the actions are enabled. This is purely a flaw in our specification, not in the protocol, since Chord nodes should still perform network maintenance when the network is ideal. To model this additional maintenance, we add a disjunct to the next-state action which allows stuttering once the network has reached an ideal state.

Finally, we would like to verify that the protocol satisfies some network safety properties. From the protocol description, we have a reasonable idea of what constitutes an *ideal* network structure: every successor pointer is correct and every node is the successor of exactly one node.

How to guarantee that a network *eventually* enters an ideal state is less obvious. As part of her analysis, Zave argues [25] that four of the invariants in Section 3.4 are absolutely necessary for the network to maintain eventual reachability:

1. **Part (1a):** There must be at least one ring. If there are no rings, there is at least one node without a successor, which means it cannot run *stabilize* or perform any lookups.

2. **Part (1b):** There must be at most one ring. If there is more than one ring, the rings become disjoint cycles, with no means of contacting nodes in other rings.

3. **Part (1c):** At any time, every node must be either part of a ring or an

appendage connected to one. If a member is neither, it has no way of contacting the remaining members of the network, leading to a partition.

4. **Part (4a):** The members of a ring must be ordered. While the network can function in a loopy state, the loss of identifier ordering breaks the core assumption behind finger tables, potentially leading to linear lookup times.

We phrase these properties in terms of our *ReachableSuccessors* operator defined in Figure 4.4. For brevity, we say that node n *can reach* node x if it is possible to reach x by following successor pointers starting from node n:

1. **Part (1a):** If there is at least one ring, there exists at least one node n such that n can reach itself.

2. **Part (1b):** If there is at most one ring, the set of nodes that can be reached is the same for every node which forms a ring.

3. **Part (1c):** If node n is part of or an appendage of the ring, it can either reach itself, or a node s such that s can reach itself.

4. **Part (4a):** If the ring is ordered, all but one node that is part of a ring has a node with a larger identifier as its successor.

The conjunction of these four conditions forms our network safety invariant (*ValidRing* in Appendix A).

4.3.5 Liveness requirements

With the network safety invariants listed above, we can begin defining network *liveness* properties. Core to the Chord protocol is the assumption that the network maintains eventual reachability: it eventually converges to the ideal state, so long as no node leaves the network and no new nodes enter.

This is formalized as Theorem IV.3 of [28]:

> *"If any sequence of join operations is executed interleaved with stabilizations, then at some time after the last join the successor pointers will form a cycle on all the nodes in the network."*

However, a Chord network can form a cycle and still be incorrect. It is possible to have *loopy* [36] cycles that violate the assumption of ordered identifiers, as in Figure 4.6. Thus, a stronger definition is needed.

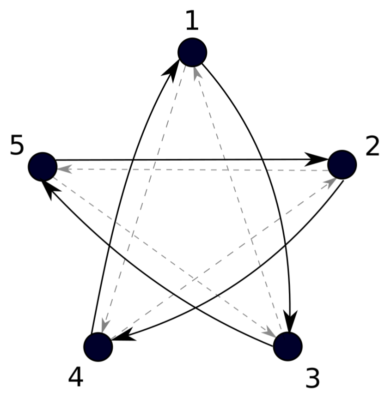


Figure 4.6: An example of a loopy ring. It is *weakly ideal* in that each node is the predecessor of its successor, but it violates the assumption that the identifiers are ordered over the ring.

For our liveness property, we use the definition of a *strongly ideal* ring [36] as our ideal state:

- Every node is its successor's predecessor. (A network satisfying this condition only is called *weakly ideal*.)
- In addition, there doesn't exist any node v such that $v \in (u, u.successor)$ while u and v are in the same ring.

This definition of an ideal network is the one employed in Zave's specification [74]. Zave confirms that Theorem IV.3 is correct in the case where no failures occur, and further strengthens it by stating that

> *"In any execution state, if there are no subsequent joins, then eventually the network will become ideal* ***and remain*** *ideal."*

We express this theorem as the temporal formula

$$\square(\mathit{Valid} \Rightarrow \Diamond\square \mathit{Ideal}) \tag{4.6}$$

which appears as *RingEventuallyBecomesStronglyIdeal* in the final specification of synchronous Chord (see Appendix A).

With the protocol, safety and liveness properties specified, we now have a complete, formally verifiable specification of pure-join Chord at the event level.

4.4 Modelling asynchronous messaging

With pure-join Chord formally specified at the event level, we can formulate a specification with atomicity at the message level, allowing multiple concurrent network events. In this specification, nodes can no longer learn about the global state of the network through shared state, but must explicitly request information through message passing.

To simulate asynchronous message passing between nodes, we specify our message-passing layer in terms of adding and drawing records from unordered sets, allowing the model-checker to thoroughly explore behaviors with out-of-order message delivery. We maintain the assumption that communication is reliable: messages may be delayed indefinitely, but they are never truly lost or altered in transit.

4.4.1 Actions with asynchronous messaging

We begin by formally specifying the *find-successor* algorithm. Since we do not include finger tables in our specification, we opt for the simplified algorithm shown in Figure 3.3. We specify it in the *recursive* style suggested in the technical report, where each "intermediate node recursively calls the lookup procedure to resolve the query" [27].

This means that responsibility for a find-successor query is delegated across multiple nodes, and there is no way for a node to determine the progress of a lookup until a response arrives. If a node fails in the process of handling a query, the query fails along with it. In our specification, we handle this by letting nodes repeat lookups until they receive a response. Figure 4.7 shows our TLA$^+$ statement of the find-successor algorithm.

Then, we can restate the join algorithm in a fashion closer to the pseudocode in Figure 3.6, with one notable difference: while the original pseudocode is written in a Remote Procedure Call (RPC) style, where other nodes' state is retrieved implicitly, we specify actions so they explicitly pass messages between nodes.

For instance, the *join* algorithm is specified as two separate actions: *Join* and *FinishJoin*, shown in Figure 4.8. *Join* sends a find-successor request to a known member of the network, while *FinishJoin* sets the successor pointer once a response arrives. By specifying actions in this manner, we can confirm that each action doesn't change the state of more than one node.

We formulate similar statements of the *stabilize* and *notify* algorithms from

```
MODULE PureJoinChord
Recursively find the successor for a given identifier.
$FindSuccessor(self) \triangleq$
    $\land\ HasJoined[self]$
    $\land$ UNCHANGED $\langle Successor, Predecessor, HasJoined,$
      $HasPredecessor, PredecessorAnswers, PredecessorRequests,$
      $Notifications\rangle$
    $\land\ \exists\ Request \in SuccessorRequests[self]:$
      If the key lies in the interval $(n, successor(n)]$, the key belongs to node $n$'s successor:
      IF  $\lor\ BetweenHalfOpen(Request.id, self, Successor[self])$
          $\lor\ self = Successor[self]$
       THEN
          Remove this query from the set of requests.
          $\land\ SuccessorRequests' = [SuccessorRequests$
              EXCEPT $![self] = @ \setminus \{Request\}]$
          Return the answer to the original sender.
          $\land\ SuccessorAnswers' = [SuccessorAnswers$
              EXCEPT $![Request.origin] =$
            $@ \cup \{[id \mapsto Request.id, successor \mapsto Successor[self]]\}]$
       ELSE
          Otherwise, pass this request to the node's successor:
          $\land\ SuccessorRequests' = [SuccessorRequests$
              EXCEPT $![Successor[self]] =$
            $@ \cup \{Request\}, ![self] = @ \setminus \{Request\}]$
          $\land\ SuccessorAnswers' = SuccessorAnswers$
```

Figure 4.7: TLA$^+$ specification of the find-successor algorithm in Figure 3.3.

```
──────────────────── MODULE PureJoinChord ────────────────────
  Join the network by requesting the successor of this node's identifier
from a known member of the network:
Join(self) ≜
        The only preconditions are that the node has not joined a network,
        ∧ ¬HasJoined[self]
        and is not in the process of joining:
        ∧ Cardinality(SuccessorAnswers[self]) = 0
        Send a request to a known member of the network:
        ∧ SuccessorRequests′ =
            [SuccessorRequests EXCEPT ![Successor[self]] =
            @ ∪ {[origin ↦ self, id ↦ self]}]
        ∧ UNCHANGED ⟨Successor, Predecessor, HasJoined,
            HasPredecessor, SuccessorAnswers, PredecessorAnswers,
            PredecessorRequests, Notifications⟩

  On receiving the successor for this node's identifier,
set it as the successor:
FinishJoin(self) ≜
       We ensure that this node has not joined a network already,
       ∧ ¬HasJoined[self]
       ∧ UNCHANGED ⟨Predecessor, HasPredecessor,
           SuccessorRequests,
           PredecessorAnswers, PredecessorRequests,
           Notifications⟩
       and that another node has sent it a reply with its initial successor:
       ∧ ∃ Answer ∈ SuccessorAnswers[self] :
            If it has, we find this node's successor from the reply,
            ∧ Successor′ = [Successor EXCEPT ![self] = Answer.successor]
            and signal that it is part of the network:
            ∧ HasJoined′ = [HasJoined EXCEPT ![self] = TRUE]
            ∧ SuccessorAnswers′ = [SuccessorAnswers EXCEPT ![self] = @ \ {Answer}]
──────────────────────────────────────────────────────────────
```

Figure 4.8: TLA$^+$ specification of the join algorithm from Figure 3.6 with asynchronous messaging.

Figure 3.7. While our synchronous specification allows nodes to send notifications to each other freely, the asynchronous version obeys the pseudocode, and demands that notifications only happen after a stabilization.

4.4.2 Expanding the set of initial states

We previously assumed that the protocol started with a single initial state, where the first node forms a single-node ring, and every other node tries to join it. We now extend the set of initial states to encompass all valid network configurations, and verify that none of them can lead to safety or liveness properties being violated.

First, we must determine which network structures are valid initial configurations. If we allowed any configuration of successor and predecessor pointers as initial states, we could have initial states with successor pointers forming disjoint cycles, violating invariant (1b) from Section 3.4.

Liben-Nowell et al. [36] point out that the graph formed by successor pointers in a valid tree is equivalent to a *pseudotree* — a graph where every component is a directed tree pointing towards a root cycle. Theoretically, we could specify an algorithm to generate the set of all pseudotrees with N vertices, and use it to construct the initial successor relation between nodes. Our literature search did not uncover any way to accomplish this in TLA$^+$.

A more straightforward approach is to consider every possible configuration of successor and predecessor pointers while enforcing the safety properties, simply discarding any initial states that do not satisfy them. This requires an additional safety property to govern the predecessor pointers in the initial state. Otherwise, the specification permits initial states where every node has its predecessor pointer unset — preventing any progress — or states where nodes' predecessor pointers lead to nodes that aren't part of the ring, leading to disruption once nodes poll them during stabilization. In our specification, an invariant requires that predecessor pointers are initially either empty, or point to a node whose successor pointer leads back to the current node.

Another issue with this approach becomes apparent when model-checking. TLC determines the set of possible initial states by enumerating every state and then evaluating the safety properties. For our proposed scheme, which considers every possible configuration of successor pointers, whether they are set, and whether the predecessor pointer is set, this means evaluating $N^N \cdot 2^N \cdot 2^N$ possible states. When testing the approach for our specification, TLC fails for $N > 5$, reporting that there are too many initial states to evaluate.

To allow us to verify the specification for $N > 5$, we introduce an option into the specification to only analyze *single-node ring* configurations as initial states. In these configurations, any of the N nodes can act as the initial single-node ring, with the rest of the nodes joining it. This compromise allows us to model-check the protocol for $N > 5$, at the cost of only examining part of the state space.

4.4.3 Verifying liveness

Model-checking the new specification shows that the previous safety invariants are maintained. There is no interleaving of messages that leads to a violation of type safety, a deadlock, or a partitioning in the network.

However, the model checker fails when verifying that the ring eventually becomes ideal, even for networks with $N = 2$ nodes. For $N = 2$, the model checker reveals that our specification permits behaviors of the form

$$\begin{aligned}
\text{Node 1 stabilizes and requests node 2's predecessor} &\longrightarrow \\
\text{Node 2 responds with its predecessor, node 1} &\longrightarrow \\
\text{Node 1 finishes stabilization, maintains node 2 as successor} &\longrightarrow \\
\text{Node 1 stabilizes and requests node 2's predecessor} &\longrightarrow \\
&\dots
\end{aligned}$$

which leave the network in a valid state, but never progress to an ideal state: node 2 is left without a predecessor, as it continuously responds to node 1 stabilizing instead of accepting its notification.

The underlying issue is that the protocol only specifies that stabilization occurs “periodically”, and the resulting specification of *Stabilize* is underconstrained. Once node 2 joins, *Stabilize* is always enabled for node 1, which means it must happen by weak fairness. Changing the specification’s fairness condition to strong fairness does not help, as node 2 accepting a notification and node 1 stabilizing are both continuously enabled actions.

We could solve this issue by introducing our own scheduling constraints through auxiliary variables. For instance, we could track how many messages each node has sent, in order to determine when a node should be pause its own activity in favor of resolving queries from other nodes. This allows us to guarantee progress so long as nodes are “reasonably active”, at the cost of increasing the complexity of the specification, introducing new assumptions, and potentially ignoring errors which arise from irregular scheduling.

Another option is to add a constraint on stabilization which depends on the global state of the network, so that a node only runs *stabilize* if it is guaranteed to improve the network. While this approach is less complex than introducing scheduling constraints, it carries a similar risk of ignoring potential errors by eliminating too many behaviors, and requires careful reasoning about which constraints to apply.

Requiring that stabilization can only occur if there are no outstanding requests or responses guarantees that the network eventually becomes ideal for 2 nodes, but leads to a potential behavior with a deadlock for 3 nodes. Requiring that the node only stabilizes when it has a non-ideal successor fails in a similar manner, leading to deadlocking behaviors where a node cannot learn its predecessor without an “unnecessary” stabilization.

Ultimately, we opt not to check liveness properties for the specifications with asynchronous messaging, as we find it hard to guarantee progress without simultaneously introducing unrealistic assumptions or needless complexity into the specification. As a sanity-check for the specification, we instead include an assertion that the ring never becomes ideal. When verifying the specification, the model-checker shows that this assertion is violated, confirming that at least one behavior leads to an ideal ring.

After restating the actions in terms of asynchronous message passing, leading to the specification in Appendix B, we can now formally verify the protocol for multiple concurrent network events.

4.5 Modelling fail-stop failures

Having introduced asynchronous messaging into our specification, we finally turn our attention to modelling fail-stop failures.

We begin by replacing our earlier definition of successor pointers with successor lists. Zave accomplishes this by introducing a second successor pointer, encoding the successor list length directly into the specification. In our specification, we use tuples to represent the successor lists, allowing their length to be set at model-checking time.

Intuitively, it would seem like the simplest way to model fail-stop failures would be to change the specification’s fairness condition to require weak fairness only for certain nodes, leading to behaviors where nodes fail by stuttering and not performing any actions. However, this approach makes it impossible to specify failure detection, which is integral to Chord's fault-tolerance algorithms. By design, TLA$^+$ specifications cannot detect stuttering steps — otherwise, it would be possible to write specifications which aren't stuttering invariant.

Instead, we add a separate variable to the specification to track whether a given node has failed, and introduce a separate action to mark a certain node as failed. Once a node has failed, it can no longer perform any actions, as we expect. This approach is also necessary to set up Chord's failure model. Stoica et al. make a probabilistic argument about the successor list length r, leaving it up to the user to set a large enough r so that r consecutive nodes failing is sufficiently unlikely. However, if our specification doesn't place any constraints on failure, the model checker simply returns a trace where r consecutive nodes indeed do fail.

Zave solves this by turning the probabilistic guarantee into a deterministic one[74], and only allowing nodes to fail if it should be possible for the network to recover from the failure. A failure is not allowed to happen if it "would leave another member with no live successor in its successor list" [25]. We adopt this constraint for our specification as well.

We grant nodes perfect failure detection, and allow them to determine whether any node has failed at any given moment. This violates our prior assumption that nodes only learn about the network's state through message passing, but is necessary to let the safety properties remain verifiable for individual states.

With successor lists and failure detection in place, we can finally specify the fault-tolerance algorithms from Figure 3.8. The TLA$^+$ specification of the algorithms can be found as *FixPredecessor*, *FixSuccessor* and *FixSuccessorList* in Appendix C.

After introducing fail-stop failures, failure detection and fault-tolerance mechanisms, we finally have a complete specification of the Chord protocol with asynchronous message passing and fault-tolerance. This specification is included in Appendix C.

4.6 Verifying counterexamples to safety properties

To verify Zave's counterexamples to the claimed safety properties, we take a two-pronged approach. We will both attempt to find our own counterexamples through model-checking our specification, and attempt to recreate the scenarios outlined in her 2012 paper [25].

The latter approach requires showing that our specification allows specific sequences of steps ending in safety properties being violated. In essence, we want to demonstrate that our specification admits a single particular behavior.

To accomplish this, we introduce an auxiliary *time* variable, and use it to restrict the set of possible actions. We specify two actions *Step* and *Done*, which respectively increments the number of steps taken, and ends in a stuttering state once a certain number of steps have occurred. We then replace the next-state relation with a disjunction over multiple *Step* actions, where each *Step* is constrained by a particular action from the Chord protocol. If a step of the proposed behavior violates the specification, the corresponding *Step* is disabled, leading to a deadlock.

Figure 4.9 shows an example of this approach, which specifies a single trace that verifies that a stabilization leaves a network in a valid state.

4.7 Summary

In this chapter, we have formally specified the Chord protocol with asynchronous messaging and fault-tolerance.

We initially specified the protocol at a coarse-grained level, only permitting a single network event to occur at any time. While this constraint would be unreasonable to enforce in practice, it allowed us to formally specify key concepts and properties, including identifier ordering, reachability, and network ideality.

Next, we increased the granularity of the specification by rewriting actions in terms of message passing between nodes. While the original pseudocode implicitly retrieves state from other nodes, our specification of message passing forces us to explicitly state when communication between nodes occurs.

Finally, we introduce fail-stop failures and fault-tolerance in the specification.

```
──────────────────── MODULE SingleBehavior ────────────────────
 Verify that a stabilization leaves the network in a valid state.
Import actions and properties from the Chord specification:
EXTENDS Chord2003
ASSUME N = 4
ASSUME SuccessorsPerNode = 2
VARIABLES Time

 Define the initial state for this trace.
ExampleInit ≜
    ∧ Time = 0
    ∧ Successors = ⟨⟨2, 3⟩, ⟨3, 4⟩, ⟨4, 1⟩, ⟨1, 2⟩⟩
    ∧ Predecessor = ⟨4, 1, 2, 3⟩
    ∧ HasJoined = ⟨TRUE, TRUE, TRUE, TRUE⟩
    ∧ HasPredecessor = ⟨TRUE, TRUE, TRUE, TRUE⟩
    ∧ HasFailed = ⟨FALSE, FALSE, FALSE, FALSE⟩
    ∧ SuccessorAnswers = [self ∈ ProcSet ↦ {}]
    ∧ SuccessorRequests = [self ∈ ProcSet ↦ {}]
    ∧ PredecessorAnswers = [self ∈ ProcSet ↦ {}]
    ∧ PredecessorRequests = [self ∈ ProcSet ↦ {}]
    ∧ Notifications = [self ∈ ProcSet ↦ {}]

Step(y) ≜
     ∧ Time = y
     ∧ Time' = y + 1

 The specification can only terminate once y steps have passed.
Done(y) ≜
      ∧ Time = y
      ∧ UNCHANGED ⟨Time⟩
      ∧ UNCHANGED vars

 The next-state action then lists the sequence of actions:
ExampleNext ≜
     Verify that the network starts in a valid state:
    ∨ ∧ Step(0) ∧ ValidRing ∧ Stabilize(1)
    ∨ ∧ Step(1) ∧ GetPredecessor(2)
    ∨ ∧ Step(2) ∧ FinishStabilize(1)
     Verify that the network ends in a valid state:
    ∨ ∧ Done(3) ∧ ValidRing

Counterexample ≜ ExampleInit ∧ □[ExampleNext]_⟨Time, vars⟩
────────────────────────────────────────────────────────────────
```

Figure 4.9: By introducing an auxiliary time variable, we can verify that our protocol description admits a particular behavior.

To ensure the model-checker does not return trivial failure scenarios, we constrain failures to only occur when the network should be able to recover from them. We also give nodes perfect failure detection to allow verification of safety properties.

Additionally, we describe a method to verify that our specification admits a particular behavior, letting us recreate specific counterexamples to Chord's claimed safety invariants.

5 Evaluation

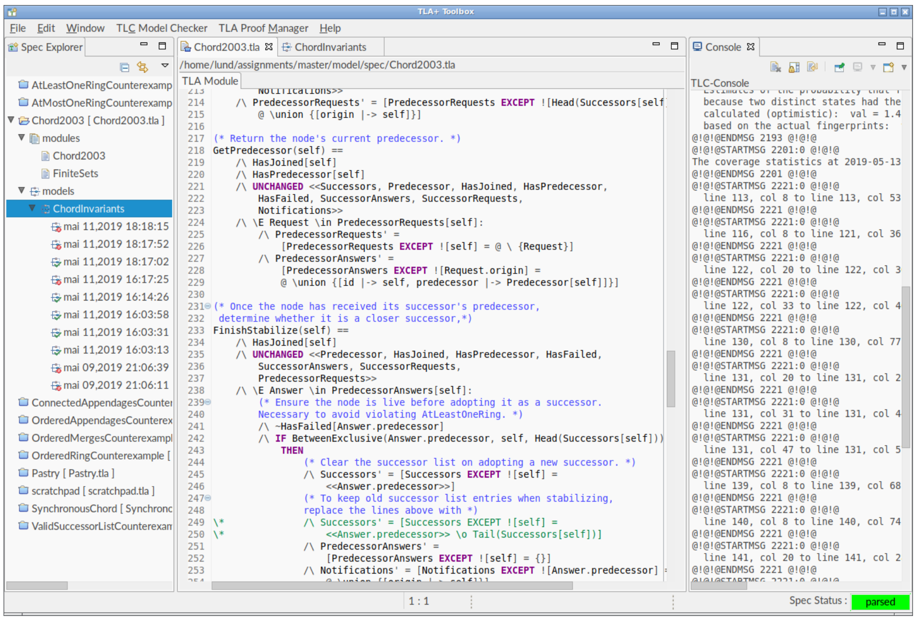
In this chapter we present the results of model-checking our specifications. We list the safety and liveness properties our specifications satisfy, and examine the computational complexity of verifying them for different model parameters. We also list Chord's claimed safety properties, and whether they hold in our specification. Finally, we discuss the implications of our findings.

5.1 Experimental setup

The specification was developed in the *TLA$^+$ Toolbox*[1] integrated development environment (IDE). It provides syntax highlighting, integrated typesetting of TLA$^+$ specifications, and an integrated model-checker. An image of the TLA$^+$ Toolbox environment is shown in Figure 5.1.

To model-check the specification, we use the TLC model-checker [34]. TLC is an *explicit-state* model-checker, which performs a breadth-first search over the state space by explicitly enumerating individual states. It keeps all data on disk and manually manages disk accesses, only using main memory as a cache. This prevents memory from being a limiting factor when verifying specifications. TLC is implemented as a Java application, allowing it to be invoked from the command line or directly from the TLA$^+$ Toolbox.

1. `http://lamport.azurewebsites.net/tla/toolbox.html`


TLA+ Toolbox
File Edit Window TLC Model Checker TLA Proof Manager Help
Spec Explorer
Chord2003.tla
ChordInvariants
/home/lund/assignments/master/model/spec/Chord2003.tla
TLA Module
Console
TLC-Console
1 : 1
Spec Status :
parsed

```
AtLeastOneRingCounterexamp
AtMostOneRingCounterexamp
Chord2003 [ Chord2003.tla ]
  modules
    Chord2003
    FiniteSets
  models
    ChordInvariants
      mai 11,2019 18:18:15
      mai 11,2019 18:17:52
      mai 11,2019 18:17:02
      mai 11,2019 16:17:25
      mai 11,2019 16:14:26
      mai 11,2019 16:03:58
      mai 11,2019 16:03:31
      mai 11,2019 16:03:13
      mai 09,2019 21:06:39
      mai 09,2019 21:06:11
ConnectedAppendagesCounter
OrderedAppendagesCounterex
OrderedMergesCounterexampl
OrderedRingCounterexample [
Pastry [ Pastry.tla ]
scratchpad [ scratchpad.tla ]
SynchronousChord [ Synchronc
ValidSuccessorListCounterexan
```

```
        Notifications>>
    /\ PredecessorRequests' = [PredecessorRequests EXCEPT ![Head(Successors[self
        @ \union {[origin |-> self]}]

(* Return the node's current predecessor. *)
GetPredecessor(self) ==
    /\ HasJoined[self]
    /\ HasPredecessor[self]
    /\ UNCHANGED <<Successors, Predecessor, HasJoined, HasPredecessor,
        HasFailed, SuccessorAnswers, SuccessorRequests,
        Notifications>>
    /\ \E Request \in PredecessorRequests[self]:
        /\ PredecessorRequests' =
            [PredecessorRequests EXCEPT ![self] = @ \ {Request}]
        /\ PredecessorAnswers' =
            [PredecessorAnswers EXCEPT ![Request.origin] =
            @ \union {[id |-> self, predecessor |-> Predecessor[self]]}]

(* Once the node has received its successor's predecessor,
 determine whether it is a closer successor,*)
FinishStabilize(self) ==
    /\ HasJoined[self]
    /\ UNCHANGED <<Predecessor, HasJoined, HasPredecessor, HasFailed,
        SuccessorAnswers, SuccessorRequests,
        PredecessorRequests>>
    /\ \E Answer \in PredecessorAnswers[self]:
        (* Ensure the node is live before adopting it as a successor.
        Necessary to avoid violating AtLeastOneRing. *)
        /\ ~HasFailed[Answer.predecessor]
        /\ IF BetweenExclusive(Answer.predecessor, self, Head(Successors[self]))
            THEN
                (* Clear the successor list on adopting a new successor. *)
                /\ Successors' = [Successors EXCEPT ![self] =
                    <<Answer.predecessor>>]
                (* To keep old successor list entries when stabilizing,
                replace the lines above with *)
\*              /\ Successors' = [Successors EXCEPT ![self] =
\*                  <<Answer.predecessor>> \o Tail(Successors[self])]
                /\ PredecessorAnswers' =
                    [PredecessorAnswers EXCEPT ![self] = {}]
                /\ Notifications' = [Notifications EXCEPT ![Answer.predecessor] =
```

```
  because two distinct states had the
  calculated (optimistic):  val = 1.4
  based on the actual fingerprints:
@!@!@ENDMSG 2193 @!@!@
@!@!@STARTMSG 2201:0 @!@!@
The coverage statistics at 2019-05-13
@!@!@ENDMSG 2201 @!@!@
@!@!@STARTMSG 2221:0 @!@!@
  line 113, col 8 to line 113, col 53
@!@!@ENDMSG 2221 @!@!@
@!@!@STARTMSG 2221:0 @!@!@
  line 116, col 8 to line 121, col 36
@!@!@ENDMSG 2221 @!@!@
@!@!@STARTMSG 2221:0 @!@!@
  line 122, col 20 to line 122, col 3
@!@!@ENDMSG 2221 @!@!@
@!@!@STARTMSG 2221:0 @!@!@
  line 122, col 33 to line 122, col 4
@!@!@ENDMSG 2221 @!@!@
@!@!@STARTMSG 2221:0 @!@!@
  line 130, col 8 to line 130, col 77
@!@!@ENDMSG 2221 @!@!@
@!@!@STARTMSG 2221:0 @!@!@
  line 131, col 20 to line 131, col 2
@!@!@ENDMSG 2221 @!@!@
@!@!@STARTMSG 2221:0 @!@!@
  line 131, col 31 to line 131, col 4
@!@!@ENDMSG 2221 @!@!@
@!@!@STARTMSG 2221:0 @!@!@
  line 131, col 47 to line 131, col 5
@!@!@ENDMSG 2221 @!@!@
@!@!@STARTMSG 2221:0 @!@!@
  line 139, col 8 to line 139, col 68
@!@!@ENDMSG 2221 @!@!@
@!@!@STARTMSG 2221:0 @!@!@
  line 140, col 8 to line 140, col 74
@!@!@ENDMSG 2221 @!@!@
@!@!@STARTMSG 2221:0 @!@!@
  line 141, col 20 to line 141, col 2
@!@!@ENDMSG 2221 @!@!@
```

Figure 5.1: Screen capture of the TLA$^+$ Toolbox IDE.

The tests were conducted with a Dell Precision T3610 workstation with an Intel Xeon E5-1620 CPU with 4 processor cores, each clocked at a base frequency of 3.6 GHz, 64 GiB of DDR3 memory with a clock frequency of 1866 MHz, and a 500 GB Samsung 840 EVO solid-state drive (SSD) with a top sequential read and write speed of 540 MB/s and 520 MB/s respectively.

The workstation runs version 19.4[2] of the *Antergos*[3] Linux distribution, based on version 5.0.9 of the Linux kernel, with all packages updated to the last versions available at the time of writing. To verify the specification, we use the version 2.13[4] of the TLC model checker, running on version `1.8.0_212-b01` of the 64-bit OpenJDK[5] Java virtual machine (JVM).

5.2 Metrics

We will measure three metrics as part of verifying our specifications:

- The **average runtime** of model-checking a specification. For the syn-

2. Approximately. Antergos operates with a rolling-release scheme similar to the Arch distribution it is based on.
3. `https://antergos.com/`
4. revision `14440ac`
5. `https://openjdk.java.net/`

chronous specification, we also list separate runtimes for checking safety properties only.

- The **number of distinct states** found by the model-checker. To avoid exploring states more than once, TLC maintains probabilistic checksums [34] of states explored. After model-checking, TLC reports the number of *distinct* states with unique checksums, giving a probabilistic estimate of the size of the state space.

- The **number of total states** explored by the model-checker, including previously explored states.

Our primary independent variable is the **number of nodes** N in the model. As Zave checks the full protocol for rings up to size 7 [74], we would ideally model-check the specifications for all $2 \leq N \leq 7$. However, time constraints force us to only check the model for $N \leq 6$. Where checking the entire state space is not possible, we also include the **maximum trace length** L of the behaviors checked.

When possible, we run 5 trials to find the average runtime of model-checking a particular model. The results list the standard deviations σ where relevant.

5.3 Verifying synchronous Chord

We begin by verifying the synchronous Chord specification, checking both safety and liveness properties. For all N from 2 to 5, we are able to check every initial state that satisfies the safety properties. For $N = 6$, TLC fails, reporting that it has too many initial states to check. To verify the protocol for $N = 6$, we use the single-node ring configuration described in Section 4.4.2.

For every configuration tested, the synchronous Chord specification satisfies the safety requirements. It is partially correct, deadlock free, and does not allow any actions that lead to partitioning in the network.

The synchronous specification also satisfies the liveness requirements, and shows that the network eventually reaches an ideal state. It also confirms Zave's pure-join correctness theorem, as the network stays in an ideal state once it has been reached.

We compare the runtime of checking safety and liveness properties to that of only checking safety properties. We use single-node ring configurations

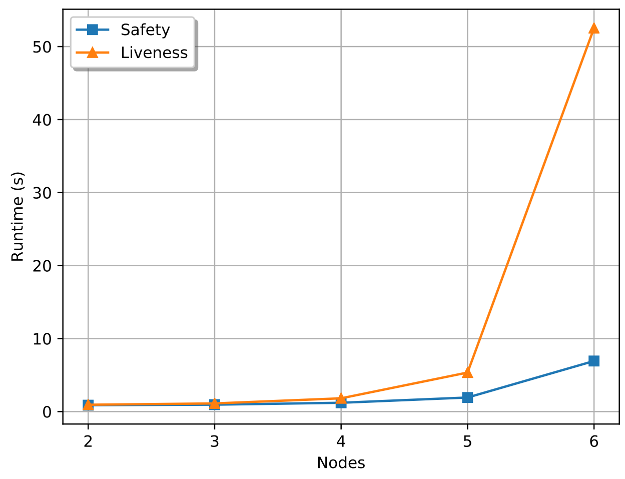


Figure 5.2: Comparison of the average runtime of model-checking the synchronous Chord specification when verifying safety and liveness, against only verifying safety properties.

as initial states for all N in this comparison, ensuring that the overhead of generating initial states is negligible. The results of this comparison are shown in Figure 5.2, and the runtimes are listed in Table 2 of Appendix D.

5.4 Verifying pure-join asynchronous Chord

We now model-check the pure-join Chord specification with asynchronous messaging. Due to the issues described in Section 4.4.3, we only verify safety properties for this specification.

For 2 and 3 nodes, TLC can explore every state of the protocol, from every possible initial configuration. For $N \geq 4$, the state space becomes large enough to make this intractable, restricting us to checking traces of finite length.

We model-check the specification for traces up to length $L = 30$. We set 30 steps as a maximum to model-check the specification in a reasonable time. We also observe that most of our counterexamples to the claimed safety properties reach a violation in less than 30 steps. On our workstation, model-checking the specification with 5 nodes and 30 steps takes on average 38 minutes. Figure 5.3 shows the relationship between the trace length and the time needed to model-check the specification, based on the runtimes listed in Table 4 of Appendix D.

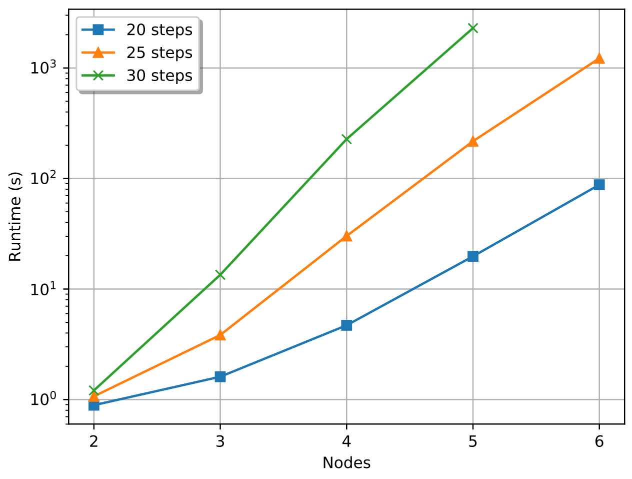


Figure 5.3: Comparison of the average runtimes of checking the pure-join Chord specification for behaviors up to 20, 25, and 30 steps. Note that the vertical axis is logarithmic — the runtime increases exponentially with the number of nodes.

Even without failures, the protocol violates one of the claimed safety properties — specifically, part (4c) of Section 3.4, which states that "if node v is in the appendage $\mathcal{A}_u$, then u is the first live cycle node following v". The specification allows the behavior from Figure 2 of [25], where two nodes concurrently attempt to join a ring of size 2, leading to a scenario where a node remains an appendage of its initial successor, even when the node following it is part of the ring.

The violation of this property leads to additional network maintenance, but doesn't lead to any critical errors. The safety properties that are critical to the network structure—connectivity and identifier ordering— are satisfied by the specification.

With no failures, we are able to verify that the protocol is partially correct and deadlock free for networks up to 5 nodes and traces up to length 30. When checking the specification for $N = 6$ and $L = 30$, TLC failed with an out of memory (OOM) error. At the time of the error, TLC explored traces up to length 28, and reported having explored 335 million distinct states.

5.5 Verifying full Chord with fault-tolerance

We finally model-check the full specification of Chord, with fail-stop failures and fault-tolerance algorithms based on the pseudocode by Liben-Nowell et al. [36]. As with the pure-join specification, we only check behaviors of up to 30 steps, and only consider the subset of possible initial states where all but one node join a single-node ring.

As originally specified, the protocol is not partially correct. Since neither *join* nor *stabilize* check whether a node is live before adopting it as a successor, the protocol allows behaviors where a node believes it is part of the network, but has no live successor. This behavior is clearly incorrect, and violates our type safety invariant. The same errors allow deadlocking behaviors.

After adding these checks, we verify that the protocol is partially correct and deadlock free for networks up to 5 nodes and behaviors up to 25 steps. When checking the full specification, TLC failed with OOM errors when checking behaviors up to 30 steps for 5 nodes, and up to 25 steps for 6 nodes.

The addition of failures and fault-tolerance algorithms increases the number of possible states significantly, as shown in Figure 5.4. Accordingly, it also takes longer to verify the protocol with fault-tolerance: model-checking the full Chord specification for 4 nodes and behaviors up to 30 steps takes 121 minutes, as opposed to less than 4 minutes for the pure-join specification. Figure 5.5 compares the time needed to verify the full Chord specification against the synchronous and the pure-join Chord specifications, based on the average runtimes listed in Appendix D, as Table 2, 4 and 6, respectively.

5.6 Claimed safety properties

Of the safety properties listed in Section 3.4,

1. **Part (1a)**: “There is at least one ring” is violated by the protocol.

    - Zave’s counterexample to this property relies on the *stabilize* algorithm (see Figure 3.7) not checking whether a node is live before adopting it as a successor. By removing this check from our specification, we find counterexamples to this property both through model-checking, and through tracing the steps outlined by Zave.

2. **Part (1b)**: “There is at most one ring” is violated by the protocol.

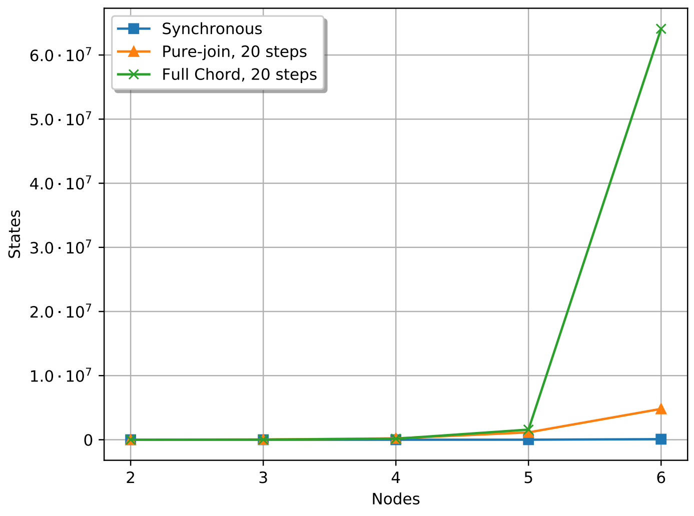


Figure 5.4: Comparison of the state space for synchronous Chord, pure-join Chord, and the full specification of Chord. For the latter two, we check behaviors up to 20 steps.

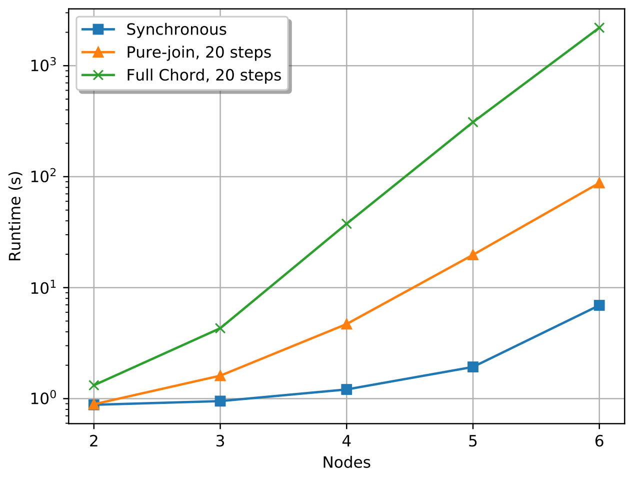


Figure 5.5: Comparison of the time needed to model-check the specifications of synchronous Chord, pure-join Chord, and full Chord with fault-tolerance. For the latter two, we check behaviors up to 20 steps. We use a logarithmic vertical axis, as in Figure 5.3.

- Here, we assume that *stabilize* in Figure 3.7 only alters the first successor in the successor list, and doesn't clear the rest of the list. By letting *FinishStabilize* retain old entries in the list, we can find violations through model-checking and through recreating Zave's counterexample.

3. **Part (1c)**: "Every node is either part of the ring or an appendage of it." is violated by the protocol.

    - Zave's counterexample to this property relies on the *join* algorithm (see Figure 3.6) not checking whether a node is live before adopting it as the initial successor. By removing this check from our specification, we find violations both through model-checking and through recreating Zave's counterexample.

4. **Part (4a)**: "The cycle is non-loopy" is violated by the protocol.

    - As in the counterexample against property (1b), we assume that *stabilize* only alters the first pointer in the successor list. We recreate Zave's counterexample, but were unable to find the counterexample by model-checking, as it requires a model with 6 nodes (see Section 5.4).

5. **Part (4b)**: "For every node v in the appendage $\mathcal{A}_u$, the path of successors from v to u is increasing" is violated by the protocol.

    - We find violations both through model-checking and through recreating Zave's counterexample.

6. **Part (5c)**: "if node v is in the appendage $\mathcal{A}_u$, then u is the first live cycle node following v" is violated by the protocol.

    - We find counterexamples to this safety property both through model-checking, and through recreating the sequence of events described by Zave.

7. **Part (5d)**: "if the successor list of [the successor of node u] skips over a live node v, then v is not in the [successor list of node u]" is violated by the protocol.

    - We again assume that *stabilize* only alters the first pointer in the successor list. Oddly, the figure depicting the counterexample (Figure 4 of [25]) shows a scenario with 5 nodes, while the corresponding counterexample in the Alloy specification only features 4 nodes. We recreate the former counterexample, but were unable to find it through model-checking.

5.7 Discussion

Figure 5.2 shows that model-checking liveness properties quickly becomes slower than only checking safety properties. This result is expected, as *Specifying Systems* [4] outright states that "checking liveness properties is a lot slower than other kinds of checking". This is due to inherent differences between safety and liveness properties: safety properties can be evaluated against individual states, whereas liveness properties only make sense for behaviors with multiple states.

This also means that the process of checking safety properties lends itself very well to parallelization. TLC verifies safety properties by having multiple worker threads fetch states from a thread-safe queue, evaluate them, and push new potential states to the back of the same queue [34].

On the other hand, evaluating liveness properties requires examining every possible behavior, not just individual states. To verify liveness properties, TLC uses a concurrent algorithm [75] based on Tarjan's strongly connected components algorithm [76]. It identifies strongly connected components in the state graph, and uses them to infer possible behaviors — if a set of states form a strongly connected component, the specification may allow behaviors which cycle through those states indefinitely. This is the case with our pure-join Chord specification, which allows behaviors where one node stabilizes an infinite number of times, preventing the other nodes from making progress.

Our naive approach to generating initial states means that we potentially only explore a subset of the state space when verifying the protocol for larger N. This is a significant weakness in our specification. Since TLC finds its initial states by "generating and checking all possible states satisfying the initial predicate" [34], solving this in the specification would mean constructing the set of possible pseudotrees directly, rather than by process of elimination. We could alternatively modify TLC to permit a larger set of initial states, or use module overriding [4] to invoke Java code from TLC, potentially letting an external utility generate the initial configurations.

From Figure 5.3, we see that modelling messaging between nodes leads to state space explosion, with the state space increasing exponentially with the number of nodes. Verifying the entire state space quickly becomes impractical, forcing us to use bounded model-checking instead. This means that we no longer prove — in the strict sense — that the safety properties hold. Instead, we make the inductive argument that we have explored enough states that a violation of safety properties is sufficiently unlikely.

TLC failing due to OOM errors is surprising, as we expected it would keep states on disk. We suspect that the state queue itself is being kept in main memory, and that the sheer number of states is causing OOMs. We use the default settings set by the TLA$^+$ Toolbox IDE when model-checking the specifications, which in our case set the heap size of the JVM to 4770 MB, and allocated 10726 MB of "offheap memory". Allocating more memory for the JVM may be a potential solution to the OOM errors, but as TLC only explored behaviors up to 27 steps before failing, doing so could simply delay the error.

Figure 5.4 and 5.5 show that modelling fault-tolerance only exacerbates these issues. Model-checking the full specification with 4 nodes takes several minutes for behaviors up to 25 steps, and over 2 hours when using 30 steps. This problem is worsened by several states being identical in all but the node identifiers, since the identifier space is rotationally symmetric. This can be seen clearly in Figure 5.6, which shows the state graph of the synchronous Chord specification for 2 nodes. Regardless of which node joins the other, the process of joining and stabilizing is identical.

To reduce the state space, TLA$^+$ allows *symmetry reduction* [77], which marks certain states as equivalent by making identical processes interchangeable. TLA$^+$ accomplishes this by letting specifications define *symmetry sets* [4], which are sets that should be equivalent under permutation. Unfortunately, due to the identifier ordering requirement, the set of identifiers cannot be permuted arbitrarily. This means that our specification cannot take advantage of symmetry reduction directly.

We were able to show that the full Chord specification admits all of Zave's counterexamples to the claimed safety properties. Due to the size of the networks involved, we were unable to find counterexamples to certain properties through model-checking. Most of the safety property violations arise from the separation of network maintenance from fault-tolerance: none of the papers specify how to integrate *stabilize* with successor lists, or account for nodes failing during a stabilization.

5.8 Summary

In this chapter, we formally verified our specifications of the Chord protocol through model-checking. We began by model-checking the synchronous speci-

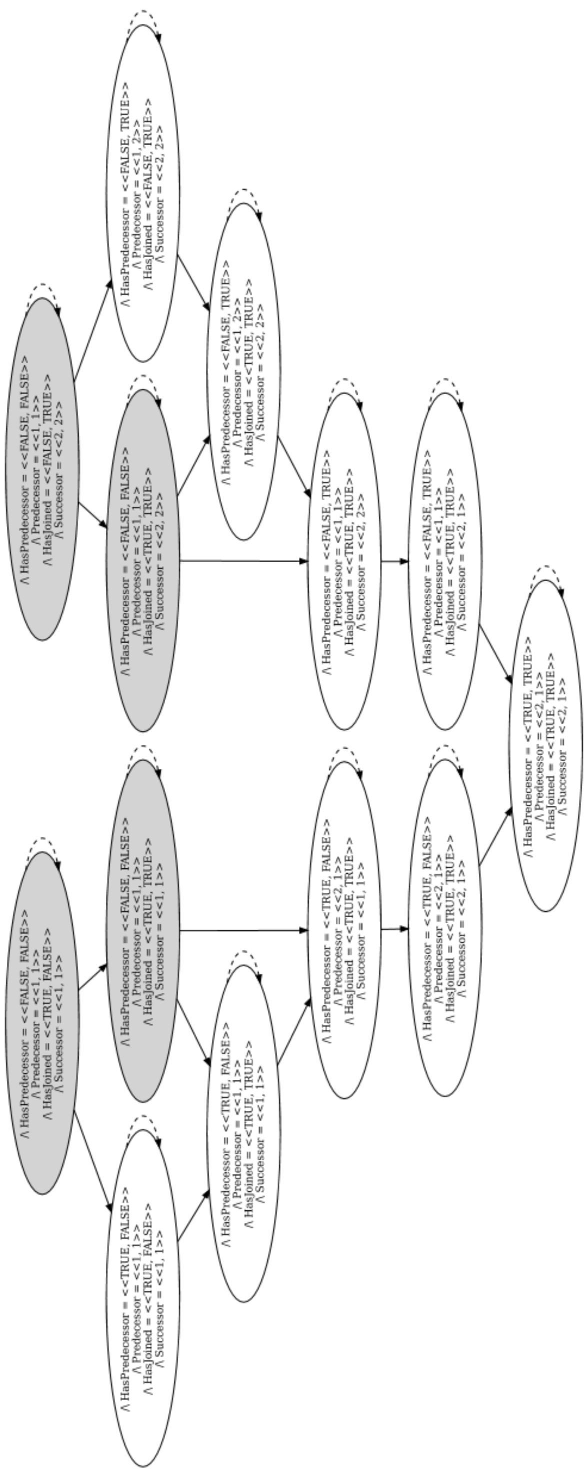

∧ HasPredecessor = <<FALSE, FALSE>> ∧ Predecessor = <<1, 1>> ∧ HasJoined = <<FALSE, TRUE>> ∧ Successor = <<2, 2>>
∧ HasPredecessor = <<FALSE, TRUE>> ∧ Predecessor = <<1, 2>> ∧ HasJoined = <<FALSE, TRUE>> ∧ Successor = <<2, 2>>
∧ HasPredecessor = <<FALSE, FALSE>> ∧ Predecessor = <<1, 1>> ∧ HasJoined = <<TRUE, TRUE>> ∧ Successor = <<2, 2>>
∧ HasPredecessor = <<FALSE, TRUE>> ∧ Predecessor = <<1, 2>> ∧ HasJoined = <<TRUE, TRUE>> ∧ Successor = <<2, 2>>
∧ HasPredecessor = <<FALSE, TRUE>> ∧ Predecessor = <<1, 1>> ∧ HasJoined = <<TRUE, TRUE>> ∧ Successor = <<2, 2>>
∧ HasPredecessor = <<FALSE, TRUE>> ∧ Predecessor = <<1, 1>> ∧ HasJoined = <<TRUE, TRUE>> ∧ Successor = <<2, 1>>
∧ HasPredecessor = <<TRUE, TRUE>> ∧ Predecessor = <<2, 1>> ∧ HasJoined = <<TRUE, TRUE>> ∧ Successor = <<2, 1>>
∧ HasPredecessor = <<FALSE, FALSE>> ∧ Predecessor = <<1, 1>> ∧ HasJoined = <<TRUE, FALSE>> ∧ Successor = <<1, 1>>
∧ HasPredecessor = <<FALSE, FALSE>> ∧ Predecessor = <<1, 1>> ∧ HasJoined = <<TRUE, TRUE>> ∧ Successor = <<1, 1>>
∧ HasPredecessor = <<TRUE, FALSE>> ∧ Predecessor = <<1, 1>> ∧ HasJoined = <<TRUE, FALSE>> ∧ Successor = <<1, 1>>
∧ HasPredecessor = <<TRUE, FALSE>> ∧ Predecessor = <<1, 1>> ∧ HasJoined = <<TRUE, TRUE>> ∧ Successor = <<1, 1>>
∧ HasPredecessor = <<TRUE, FALSE>> ∧ Predecessor = <<2, 1>> ∧ HasJoined = <<TRUE, TRUE>> ∧ Successor = <<1, 1>>
∧ HasPredecessor = <<TRUE, FALSE>> ∧ Predecessor = <<2, 1>> ∧ HasJoined = <<TRUE, TRUE>> ∧ Successor = <<2, 1>>

Figure 5.6: TLA$^+$ Toolbox visualization of the state space for synchronous Chord with 2 nodes. Regardless of whether node 1 or 2 joins the other, the process for reaching an ideal network (the bottom state) is identical.

fication, where only one network event takes place at a time, and no failures occur. Under these assumptions, the protocol is partially correct, deadlock free, and does not lead to any partitioning in the network.

We then model-checked the pure-join Chord specification with asynchronous messaging. While we were able to explore the entire state space of the protocol up to $N = 3$, larger N lead to state space explosion, restricting us to only checking traces of a certain length. For traces up to length $L = 30$, we verify that the protocol is partially correct, deadlock free, and doesn't partition the network, so long as no failures occur.

Finally, we check the full specification of Chord with fail-stop failures. While the specification maintains partial correctness, the introduction of failures leads to scenarios with deadlocks and violations of safety properties.

We were able to find violations of all of Chord's claimed safety properties. We recreated all of Zave's counterexamples, but could not demonstrate all of them through model-checking the specification ourselves, due to the scale of the networks involved.

6 Related work

One of the earliest applications of formal analysis to peer-to-peer network algorithms is the work of Borgström et al. [78], who examined the $\mathcal{DKS}$ [79] distributed hash table. To verify the $\mathcal{DKS}$ lookup protocol, they apply the CCS [80] process calculus to show weak bisimilar equivalence between the specification and their implementation. They demonstrate that their implementation satisfies the specification, by showing that it admits the same actions and system states as the specification does. However, their analysis assumes a static network configuration, and disregards the possibility of node arrivals and departures.

Zave's initial analysis of a simplified "pure-join" Chord protocol with no node failures bears similarities to the work by Bakhshi and Gurov [1], which models pure-join Chord in the π-calculus. Similar to Borgström et al., Bakhshi and Gurov establish a weak bisimilarity between Chord's stabilization protocol and its specification, and show that the protocol eventually settles the network into a ring topology after every node has joined the network.

The Chord protocol has been used previously as a testbench for formal verification methods. In 2015, Zave [68] also formally specified the Chord protocol in the Promela [81] specification language, and verified it with the Spin model checker. The Spin model simulates concurrent processes communicating through message queues, in a similar fashion to our TLA$^+$ specification. Comparing it to the Alloy specification, Zave finds that the Spin model suffers from state space explosion — checking the model for 5 nodes "aborted after using 300 gigabytes of memory". While one of the purported advantages of Spin over Alloy was its ability to check liveness properties, the vast state space made it infeasible to use Spin's liveness verification for the model.

Killian et al. [82] tested their MaceMC model checker with an implementation of Chord. However, they did not uncover the protocol issues uncovered by Zave [74]. They also enforce the property that "a node's predecessor is itself if and only if its successor is itself". This property would prevent any node from joining a single-node ring, suggesting that their implementation does not implement the protocol as originally specified.

Yabandeh et al. [83] also use a Chord implementation to test their CrystalBall model checker. CrystalBall explores the state space concurrently with the execution of the implementation to predict potential states the implementation can take on, and blocks events which could lead to a safety property violation. As such, they do not aim to find protocol errors, but instead attempt to avoid them. They also enforce the safety property Killian et al. suggest, which leads us to believe they do not implement the original Chord protocol either.

As it was principally designed to reason about distributed and concurrent systems, there exists a significant body of literature on applying TLA$^+$ to formal verification of distributed systems. Several distributed algorithms have been formally specified in TLA$^+$, such as the consensus algorithms *Disk Paxos* [84], *Egalitarian Paxos* [85] and *Raft* [86].

In our case, the most relevant work is the 2011 work by Lu et al. [87] employing TLA+ to model the Pastry [88] distributed hash table. Pastry, like Chord, operates with a ring topology and identifier space, and provides a logarithmic bound on the number of hops necessary to look up a particular key.

However, instead of associating keys with the node succeeding it in the identifier order, as Chord does, Pastry associates keys with the node whose identifier lies numerically closest. This leads to a routing algorithm where nodes maintain *routing tables* of other nodes sharing the same identifier prefix, along with *leaf sets* containing their immediate neighbors in the identifier space. To achieve better network locality, Pastry nodes additionally maintain *neighborhood sets* of geographically close nodes.

Lu et al. model the Pastry join and lookup protocols in TLA+, demonstrating both liveness and "correct key delivery", showing that only one node is responsible for any given key. They initially take a model-based approach, verifying the protocol with TLC to find counterexamples to their invariants. After finding and correcting the counterexamples, they construct a deductive proof of correct key delivery in the TLAPS proof system.

In 2018, Azmy et al. [89] found that this TLAPS proof depended on unproven axioms and occasionally erroneous assumptions. By replacing incorrect assumptions and substituting others with proven lemmas, they were able to give a full correctness proof.

6.1 State space reduction techniques

The primary issue our specification faces is state space explosion. State space explosion is a problem inherent to model-checking, as complexity-theoretic arguments show that it is unavoidable in the worst case [16]. Despite this, several techniques for state space reduction have been developed. One well-known technique is *partial order reduction* [90], which identifies events that do not affect each others' execution, and uses the information to avoid exploring certain behaviors. Another key technique is *symbolic* model-checking [91], which represents the state space symbolically instead of explicitly, allowing multiple states to be subsumed and verified simultaneously.

Recent work has been focused on applying SAT and SMT solvers [92] to perform *bounded* model-checking [93]. By using a symbolic state space representation, SMT solvers can refute safety properties by directly solving for a finite-length counterexample, speeding up model-checking substantially. Ongoing work by Konnov et al. [94] implements a bounded model-checker for TLA$^+$ specifications.

Another potential avenue for dealing with state space explosion is to move model-checking to devices with highly parallel architectures. For instance, Wijs et al. [95] implement a model-checker running on GPUs, finding that their highly parallel architecture allows significant speedups. Cho et al. [96] likewise implement a model-checker for field-programmable gate arrays (FPGAs), and report similar speedups, but opt for a probabilistic "swarm verification" solution due to the limited on-chip memory on the FPGA.

7 Conclusion

This thesis has presented a formal specification of the Chord distributed hash table in TLA$^+$. The protocol was first specified at a high level, with no concurrency or failures. The specification was then gradually refined, leading to a full specification of the protocol with asynchronous messaging and fault-tolerance.

Modelling asynchronous messaging allows the protocol to be specified in a fashion close to the pseudocode published in the 2003 TON Chord paper [28], albeit with explicit messaging between nodes instead of implicit sharing of state.

This thesis reproduces the results of Zave's [25] analysis of the Chord protocol, showing both that model-checking the TLA$^+$ specification uncovers violations of Chord's claimed safety properties [36], and that the specification admits the counterexamples presented by Zave.

7.1 Concluding remarks

Our specification is unfortunately plagued by state space explosion. The state space of the specification increases exponentially with both network size and behavior length, limiting model-checking to small networks and short traces. While TLA$^+$ offers certain state space reduction techniques, we were unable to take advantage of them.

Despite this limitation, we found formal verification useful to validate high-level designs on a small scale. This process rests on the *small-scope hypothesis* [97], which states that most system errors can be found by testing the system for all inputs within some small scope. In analyzing fault reports from distributed systems, Yuan et al. [98] found that most of the reported failures required less than 4 events to reproduce, corroborating this hypothesis. Zave [68] provides additional evidence, arguing that at most 6 nodes were necessary to model the counterexamples for Chord's claimed safety properties. This suggests that while model-checking may not lead to conclusive proofs of a system's correctness, it can still be a valuable tool for verifying system designs.

Using formal verification as a design aid, and not solely as a method of establishing a system's correctness, also forestalls the criticism that formal specification leads to duplication of work. Once a formal specification is in place, it can serve as a blueprint for the implementation, speeding up the implementation process.

The violations of Chord's claimed safety properties arise from ambiguous wording in the paper, and from treating fault-tolerance as a separate component of the protocol. By formally specifying and verifying the protocol, we were able to clear up ambiguity, uncover implicit assumptions, and identify errors arising from deficiencies in the fault-tolerance design.

7.2 Future work

Finally, we identify opportunities and potential improvements for the specification:

Other model-checkers We would like to verify our specification against other model-checkers. Two promising candidates are ProB [99], which can interpret and verify TLA$^+$ specifications through a translation layer by Hansen and Leuschel [100], and BmcMT by Konnov et al. [94], which performs bounded model checking of TLA$^+$ with an SMT solver. The specification may have to be adapted for these tools: unlike TLA$^+$, B builds on a strongly typed logic, and BmcMT does not yet support all of the features TLC does.

Initial states Our formal specification is limited by a naive approach to enumerating initial states, which leads to TLC failing due to the sheer number of possible states. It may be possible to enumerate valid states directly, or to invoke an external utility to generate initial states before letting TLC verify them.

Implementation Finally, we would like to build an implementation of Chord on top of our specification. One promising lightweight approach is *model-based trace checking*, as suggested by Howard et al. [101], which integrates logging with formal verification. By using the specification to add tracing code to significant events, it is possible to infer the behavior of the implemented system. Then, by methods similar to the one presented in Section 4.6, it is possible to show that the specification admits the behavior of the implementation.

Bibliography

[1] Rana Bakhshi and Dilian Gurov. "Verification of Peer-to-peer Algorithms: A Case Study." In: *Electronic Notes in Theoretical Computer Science* 181 (2007), pp. 35–47. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2007.01.052. URL: http://www.sciencedirect.com/science/article/pii/S1571066107003672.

[2] Robert J. van Glabbeek. "Bisimulation." In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 136–139. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_149. URL: https://doi.org/10.1007/978-0-387-09766-4_149.

[3] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems*. Pearson Education, 2013. ISBN: 1-292-02552-2 978-1-292-02552-0.

[4] Leslie Lamport, John Matthews, Mark Tuttle, and Yuan Yu. "Specifying and Verifying Systems with TLA+." In: *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*. EW 10. New York, NY, USA: ACM, 2002, pp. 45–48. DOI: 10.1145/1133373.1133382. URL: http://doi.acm.org/10.1145/1133373.1133382.

[5] Bowen Alpern and Fred B. Schneider. *Defining Liveness*. Ithaca, NY, USA: Cornell University, 1984.

[6] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. "On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-world Graphs." In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. event-place: Denver, Colorado. New York, NY, USA: ACM, 2013, 92:1–92:11. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503246. URL: http://doi.acm.org/10.1145/2503210.2503246.

[7] Yangjun Chen. "A New Algorithm for Computing Transitive Closures." In: *Proceedings of the 2004 ACM Symposium on Applied Computing*. SAC '04. event-place: Nicosia, Cyprus. New York, NY, USA: ACM, 2004, pp. 1091–1092. ISBN: 1-58113-812-1. DOI: 10.1145/967900.968121. URL: http://doi.acm.org/10.1145/967900.968121.

[8] Leslie Lamport. *The TLA+ Hyperbook*. 2015. URL: https://lamport.azurewebsites.net/tla/hyperbook.html (visited on 04/01/2019).

[9] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. "Debugging Distributed Systems." In: *Commun. ACM* 59.8 (July 2016),

pp. 32–37. ISSN: 0001-0782. DOI: 10.1145/2909480. URL: http://doi.acm.org/10.1145/2909480.

[10] Jenny Abrahamson, Ivan Beschastnikh, Yuriy Brun, and Michael D. Ernst. "Shedding Light on Distributed System Executions." In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ICSE Companion 2014. event-place: Hyderabad, India. New York, NY, USA: ACM, 2014, pp. 598–599. ISBN: 978-1-4503-2768-8. DOI: 10.1145/2591062.2591134. URL: http://doi.acm.org/10.1145/2591062.2591134.

[11] Boby George and Laurie Williams. "An Initial Investigation of Test Driven Development in Industry." In: *Proceedings of the 2003 ACM Symposium on Applied Computing*. SAC '03. New York, NY, USA: ACM, 2003, pp. 1135–1139. ISBN: 1-58113-624-2. DOI: 10.1145/952532.952753. URL: http://doi.acm.org/10.1145/952532.952753.

[12] Nicholas D. Matsakis and Felix S. Klock II. "The Rust Language." In: *Ada Lett.* 34.3 (Oct. 2014), pp. 103–104. ISSN: 1094-3641. DOI: 10.1145/2692956.2663188. URL: http://doi.acm.org/10.1145/2692956.2663188.

[13] Thomas Ball and Sriram K. Rajamani. "The SLAM Project: Debugging System Software via Static Analysis." In: *SIGPLAN Not.* 37.1 (Jan. 2002), pp. 1–3. ISSN: 0362-1340. DOI: 10.1145/565816.503274. URL: http://doi.acm.org/10.1145/565816.503274.

[14] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "A Static Analyzer for Large Safety-critical Software." In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI '03. New York, NY, USA: ACM, 2003, pp. 196–207. ISBN: 1-58113-662-5. DOI: 10.1145/781131.781153. URL: http://doi.acm.org/10.1145/781131.781153.

[15] Michael Huth. *Logic in computer science : modelling and reasoning about systems*. Cambridge U.K. New York: Cambridge University Press, 2004. ISBN: 978-0-521-54310-1.

[16] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. "Model Checking: Algorithmic Verification and Debugging." In: *Commun. ACM* 52.11 (Nov. 2009), pp. 74–84. ISSN: 0001-0782. DOI: 10.1145/1592761.1592781. URL: http://doi.acm.org/10.1145/1592761.1592781.

[17] H. Geuvers. "Proof assistants: History, ideas and future." In: *Sadhana* 34.1 (Feb. 2009), pp. 3–25. ISSN: 0973-7677. DOI: 10.1007/s12046-009-0001-5. URL: https://doi.org/10.1007/s12046-009-0001-5.

[18] John McCarthy. "A Basis for a Mathematical Theory of Computation, Preliminary Report." In: *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM '61 (Western). event-place: Los Angeles, California. New York, NY, USA: ACM, 1961,

pp. 225–238. DOI: 10.1145/1460690.1460715. URL: http://doi.acm.org/10.1145/1460690.1460715.

[19] C. A. R. Hoare. “How Did Software Get So Reliable Without Proof?” In: *Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*. FME ’96. London, UK, UK: Springer-Verlag, 1996, pp. 1–17. ISBN: 3-540-60973-3. URL: http://dl.acm.org/citation.cfm?id=647537.729681.

[20] Peter Gutmann. *Cryptographic security architecture : design and verification*. New York: Springer, 2004. ISBN: 978-0-387-21551-8.

[21] A. Hall. “Seven myths of formal methods.” In: *IEEE Software* 7.5 (Sept. 1990), pp. 11–19. ISSN: 0740-7459. DOI: 10.1109/52.57887.

[22] Qi Zhang, Lu Cheng, and Raouf Boutaba. “Cloud computing: state-of-the-art and research challenges.” In: *Journal of Internet Services and Applications* 1.1 (May 2010), pp. 7–18. ISSN: 1869-0238. DOI: 10.1007/s13174-010-0007-6. URL: https://doi.org/10.1007/s13174-010-0007-6.

[23] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. “How Amazon Web Services Uses Formal Methods.” In: *Commun. ACM* 58.4 (Mar. 2015), pp. 66–73. ISSN: 0001-0782. DOI: 10.1145/2699417. URL: http://doi.acm.org/10.1145/2699417.

[24] Swaminathan Sivasubramanian. “Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service.” In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’12. event-place: Scottsdale, Arizona, USA. New York, NY, USA: ACM, 2012, pp. 729–730. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213945. URL: http://doi.acm.org/10.1145/2213836.2213945.

[25] Pamela Zave. “Using Lightweight Modeling to Understand Chord.” In: *SIGCOMM Comput. Commun. Rev.* 42.2 (Mar. 2012), pp. 49–57. ISSN: 0146-4833. DOI: 10.1145/2185376.2185383. URL: http://doi.acm.org/10.1145/2185376.2185383.

[26] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. “Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.” In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), pp. 149–160. ISSN: 0146-4833. DOI: 10.1145/964723.383071. URL: http://doi.acm.org/10.1145/964723.383071.

[27] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications.” In: *Technical Report TR-819, MIT LCS* (Mar. 2001). URL: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-819.pdf (visited on 03/31/2019).

[28] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. “Chord: A Scalable

Peer-to-peer Lookup Protocol for Internet Applications." In: *IEEE/ACM Trans. Netw.* 11.1 (Feb. 2003), pp. 17–32. ISSN: 1063-6692. DOI: 10.1109/TNET.2002.808407. URL: http://dx.doi.org/10.1109/TNET.2002.808407.

[29] Pamela Zave. "Reasoning About Identifier Spaces: How to Make Chord Correct." In: *IEEE Trans. Softw. Eng.* 43.12 (Dec. 2017), pp. 1144–1156. ISSN: 0098-5589. DOI: 10.1109/TSE.2017.2655056. URL: https://doi.org/10.1109/TSE.2017.2655056.

[30] Stephan Merz. "The Specification Language TLA+." In: *Logics of Specification Languages*. Ed. by Dines Bjørner and Martin C. Henson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 401–451. ISBN: 978-3-540-74107-7. DOI: 10.1007/978-3-540-74107-7_8. URL: https://doi.org/10.1007/978-3-540-74107-7_8.

[31] Robert Beers. "Pre-RTL Formal Verification: An Intel Experience." In: *Proceedings of the 45th Annual Design Automation Conference*. DAC '08. New York, NY, USA: ACM, 2008, pp. 806–811. ISBN: 978-1-60558-115-6. DOI: 10.1145/1391469.1391675. URL: http://doi.acm.org/10.1145/1391469.1391675.

[32] Microsoft Azure. *High-level TLA+ specifications for the five consistency levels offered by Azure Cosmos DB*. Sept. 23, 2018. URL: https://github.com/Azure/azure-cosmos-tla (visited on 04/26/2019).

[33] Leslie Lamport. "The Temporal Logic of Actions." In: *ACM Trans. Program. Lang. Syst.* 16.3 (May 1994), pp. 872–923. ISSN: 0164-0925. DOI: 10.1145/177492.177726. URL: http://doi.acm.org/10.1145/177492.177726.

[34] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model Checking TLA+ Specifications." In: 1703 (June 1999), pp. 54–66. URL: https://www.microsoft.com/en-us/research/publication/model-checking-tla-specifications/.

[35] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. "A TLA+ Proof System." In: vol. 418. Aug. 2008, pp. 17–37. URL: https://www.microsoft.com/en-us/research/publication/tla-proof-system/.

[36] David Liben-Nowell, Hari Balakrishnan, and David Karger. "Analysis of the Evolution of Peer-to-peer Systems." In: *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*. PODC '02. event-place: Monterey, California. New York, NY, USA: ACM, 2002, pp. 233–242. ISBN: 1-58113-485-1. DOI: 10.1145/571825.571863. URL: http://doi.acm.org/10.1145/571825.571863.

[37] Håvard Johansen, Dag Johansen, and Robbert van Renesse. "FirePatch: Secure and Time-Critical Dissemination of Software Patches." In: *New Approaches for Security, Privacy and Trust in Complex Environments*. Ed. by Hein Venter, Mariki Eloff, Les Labuschagne, Jan Eloff, and Rossouw

von Solms. Boston, MA: Springer US, 2007, pp. 373–384. ISBN: 978-0-387-72367-9.

[38] Håvard D. Johansen, Robbert Van Renesse, Ymir Vigfusson, and Dag Johansen. "Fireflies: A Secure and Scalable Membership and Gossip Service." In: *ACM Trans. Comput. Syst.* 33.2 (May 2015). ISSN: 0734-2071. DOI: 10.1145/2701418. URL: http://doi.acm.org/10.1145/2701418.

[39] Amos Fiat, Jared Saia, and Maxwell Young. "Making Chord Robust to Byzantine Attacks." In: *Algorithms – ESA 2005*. Ed. by Gerth Stølting Brodal and Stefano Leonardi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 803–814. ISBN: 978-3-540-31951-1.

[40] Anders Tungeland Gjerdrum, Robert Pettersen, Håvard D. Johansen, Robbert Van Renesse, and Dag Johansen. "Trusted Computing on Privacy Sensitive Data with Diggi." In: *SOSP 2017: The ACM Symposium on Operating System Principles*. Oct. 2017.

[41] Håvard D. Johansen, Eleanor Birrell, Robbert van Renesse, Fred B. Schneider, Magnus Stenhaug, and Dag Johansen. "Enforcing Privacy Policies with Meta-Code." In: *Proceedings of the 6th Asia-Pacific Workshop on Systems*. APSys '15. New York, NY, USA: ACM, 2015, 16:1–16:7. ISBN: 978-1-4503-3554-6. DOI: 10.1145/2797022.2797040. URL: http://doi.acm.org/10.1145/2797022.2797040.

[42] Eleanor Birrell, Anders Gjerdrum, Robbert van Renesse, Håvard Johansen, Dag Johansen, and Fred B. Schneider. "SGX Enforcement of Use-Based Privacy." In: *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*. WPES'18. event-place: Toronto, Canada. New York, NY, USA: ACM, 2018, pp. 155–167. ISBN: 978-1-4503-5989-4. DOI: 10.1145/3267323.3268954. URL: http://doi.acm.org/10.1145/3267323.3268954.

[43] Peter J. Denning, D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. "Computing As a Discipline." In: *Commun. ACM* 32.1 (Jan. 1989), pp. 9–23. ISSN: 0001-0782. DOI: 10.1145/63238.63239. URL: http://doi.acm.org/10.1145/63238.63239.

[44] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

[45] Amir Pnueli. "The Temporal Logic of Programs." In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS '77. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32. URL: https://doi.org/10.1109/SFCS.1977.32.

[46] C. B. Jones. "The early search for tractable ways of reasoning about programs." In: *IEEE Annals of the History of Computing* 25.2 (Apr. 2003), pp. 26–49. ISSN: 1058-6180. DOI: 10.1109/MAHC.2003.1203057.

[47] Alfred Tarski. *Introduction to logic and to the methodology of deductive sciences*. New York: Dover Publications, 1995. ISBN: 0-486-28462-X.

[48] Alan M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem." In: *Proceedings of the London Mathematical Society* 2.42 (1936), pp. 230–265.

[49] Lawrence C. Paulson. "Computational logic: its origins and applications." In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 474.2210 (2018), p. 20170872. DOI: 10.1098/rspa.2017.0872. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.2017.0872.

[50] F. L. Morris and C. B. Jones. "An Early Program Proof by Alan Turing." In: *Annals of the History of Computing* 6.2 (Apr. 1984), pp. 139–143. ISSN: 0164-1239. DOI: 10.1109/MAHC.1984.10017.

[51] Robert W. Floyd. "Assigning Meanings to Programs." In: *Program Verification: Fundamental Issues in Computer Science*. Ed. by Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin. Dordrecht: Springer Netherlands, 1993, pp. 65–81. ISBN: 978-94-011-1793-7. DOI: 10.1007/978-94-011-1793-7_4. URL: https://doi.org/10.1007/978-94-011-1793-7_4.

[52] Susan Owicki and David Gries. "Verifying Properties of Parallel Programs: An Axiomatic Approach." In: *Commun. ACM* 19.5 (May 1976), pp. 279–285. ISSN: 0001-0782. DOI: 10.1145/360051.360224. URL: http://doi.acm.org/10.1145/360051.360224.

[53] Kensuke Kojima and Atsushi Igarashi. "A Hoare Logic for GPU Kernels." In: *ACM Trans. Comput. Logic* 18.1 (Feb. 2017), 3:1–3:43. ISSN: 1529-3785. DOI: 10.1145/3001834. URL: http://doi.acm.org/10.1145/3001834.

[54] R. M. Burstall. "Proving Properties of Programs by Structural Induction." In: *The Computer Journal* 12.1 (1969), pp. 41–48. ISSN: 0010-4620. DOI: 10.1093/comjnl/12.1.41. URL: https://dx.doi.org/10.1093/comjnl/12.1.41.

[55] Zohar Manna and Richard Waldinger. "Is "Sometime" Sometimes Better Than "Always"?: Intermittent Assertions in Proving Program Correctness." In: *Proceedings of the 2Nd International Conference on Software Engineering*. ICSE '76. event-place: San Francisco, California, USA. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 32–39. URL: http://dl.acm.org/citation.cfm?id=800253.807645.

[56] Edmund M. Clarke and E. Allen Emerson. "25 Years of Model Checking." In: ed. by Orna Grumberg and Helmut Veith. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 196–215. ISBN: 978-3-540-69849-4. DOI: 10.1007/978-3-540-69850-0_12. URL: http://dx.doi.org/10.1007/978-3-540-69850-0_12.

[57] Fred Kröger and Stephan Merz. *Temporal Logic and State Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. ISBN: 978-3-

540-68635-4. DOI: 10.1007/978-3-540-68635-4_11. URL: https://doi.org/10.1007/978-3-540-68635-4_11.

[58] Amir Pnueli. "The temporal semantics of concurrent programs." In: *Theoretical Computer Science* 13.1 (1981), pp. 45–60. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(81)90110-9. URL: http://www.sciencedirect.com/science/article/pii/0304397581901109.

[59] Edmund M. Clarke and E. Allen Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic." In: *Logics of Programs*. Ed. by Dexter Kozen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 52–71. ISBN: 978-3-540-39047-3.

[60] Jean-Pierre Queille and Joseph Sifakis. "Specification and Verification of Concurrent Systems in CESAR." In: *Proceedings of the 5th Colloquium on International Symposium on Programming*. London, UK, UK: Springer-Verlag, 1982, pp. 337–351. ISBN: 3-540-11494-7. URL: http://dl.acm.org/citation.cfm?id=647325.721668.

[61] Leslie Lamport. ""Sometime" is Sometimes "Not Never": On the Temporal Logic of Programs." In: *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '80. event-place: Las Vegas, Nevada. New York, NY, USA: ACM, 1980, pp. 174–185. ISBN: 0-89791-011-7. DOI: 10.1145/567446.567463. URL: http://doi.acm.org/10.1145/567446.567463.

[62] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web." In: *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*. STOC '97. event-place: El Paso, Texas, USA. New York, NY, USA: ACM, 1997, pp. 654–663. ISBN: 0-89791-888-6. DOI: 10.1145/258533.258660. URL: http://doi.acm.org/10.1145/258533.258660.

[63] John Fraleigh. *A first course in abstract algebra*. Boston: Addison-Wesley, 2003. ISBN: 0-201-76390-7.

[64] D. Eastlake and P. Jones. *US Secure Hash Algorithm 1 (SHA1)*. RFC 3174. Published: Internet Requests for Comments. RFC Editor, Sept. 2001. URL: http://www.rfc-editor.org/rfc/rfc3174.txt.

[65] William Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees." In: *Commun. ACM* 33.6 (June 1990), pp. 668–676. ISSN: 0001-0782. DOI: 10.1145/78973.78977. URL: http://doi.acm.org/10.1145/78973.78977.

[66] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2016. ISBN: 0-262-52890-8 978-0-262-52890-0.

[67] Chris Newcombe. "Why Amazon Chose TLA+." In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Ed. by Yamine Ait Ameur and Klaus-Dieter Schewe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 25–39. ISBN: 978-3-662-43652-3.

[68] P. Zave. “A practical comparison of Alloy and Spin.” In: *Formal Aspects of Computing* 27.2 (Mar. 2015), pp. 239–253. ISSN: 1433-299X. DOI: 10.1007/s00165-014-0302-2. URL: https://doi.org/10.1007/s00165-014-0302-2.

[69] Nuno Macedo and Alcino Cunha. “Alloy meets TLA+: An exploratory study.” In: *CoRR* abs/1603.03599 (2016). URL: http://arxiv.org/abs/1603.03599.

[70] Daniel Stutzbach and Reza Rejaie. “Understanding Churn in Peer-to-peer Networks.” In: *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*. IMC ’06. event-place: Rio de Janeriro, Brazil. New York, NY, USA: ACM, 2006, pp. 189–202. ISBN: 1-59593-561-4. DOI: 10.1145/1177080.1177105. URL: http://doi.acm.org/10.1145/1177080.1177105.

[71] Supriya Krishnamurthy, Sameh El-Ansary, Erik Aurell, and Seif Haridi. “A Statistical Theory of Chord Under Churn.” In: *Peer-to-Peer Systems IV*. Ed. by Miguel Castro and Robbert van Renesse. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 93–103. ISBN: 978-3-540-31906-1.

[72] Monique Mezher and Ahmed Ibrahim. “Introducing Practical SHA-1 Collisions to the Classroom.” In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE ’19. event-place: Minneapolis, MN, USA. New York, NY, USA: ACM, 2019, pp. 879–884. ISBN: 978-1-4503-5890-3. DOI: 10.1145/3287324.3287446. URL: http://doi.acm.org/10.1145/3287324.3287446.

[73] Fabian Giesen. *Intervals in modular arithmetic*. Sept. 24, 2015. URL: https://fgiesen.wordpress.com/2015/09/24/intervals-in-modular-arithmetic/ (visited on 04/10/2019).

[74] Pamela Zave. *Lightweight verification of network protocols: The case of Chord.* AT&T Laboratories - Research, Jan. 2010.

[75] Microsoft. *"LiveWorker.java" — source code of TLC model checker*. Mar. 27, 2019. URL: https://github.com/tlaplus/tlaplus/blob/master/tlatools/src/tlc2/tool/liveness/LiveWorker.java (visited on 05/04/2019).

[76] R. Tarjan. “Depth-First Search and Linear Graph Algorithms.” In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010. URL: https://doi.org/10.1137/0201010.

[77] A. Prasad Sistla. “Employing symmetry reductions in model checking.” In: *Computer Languages, Systems & Structures* 30.3 (2004), pp. 99–137. ISSN: 1477-8424. DOI: https://doi.org/10.1016/j.cl.2004.02.002. URL: http://www.sciencedirect.com/science/article/pii/S1477842404000156.

[78] Johannes Borgström, Uwe Nestmann, Luc Onana, and Dilian Gurov. “Verifying a Structured Peer-to-Peer Overlay Network: The Static Case.” In: *Global Computing*. Ed. by Corrado Priami and Paola Quaglia. Berlin,

Heidelberg: Springer Berlin Heidelberg, 2005, pp. 250–265. ISBN: 978-3-540-31794-4.

[79] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. “DKS (N, K, F): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications.” In: *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*. CCGRID ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 344–. ISBN: 0-7695-1919-9. URL: http://dl.acm.org/citation.cfm?id=791231.792434.

[80] Anna Ingólfsdóttir. “A semantic theory for value-passing processes based on the late approach.” In: *Information and Computation* 184.1 (2003), pp. 1–44. ISSN: 0890-5401. DOI: https://doi.org/10.1016/S0890-5401(03)00052-X. URL: http://www.sciencedirect.com/science/article/pii/S089054010300052X.

[81] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. 1st. Addison-Wesley Professional, 2011. ISBN: 0-321-77371-3 978-0-321-77371-5.

[82] Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. “Mace: Language support for building distributed systems.” In: *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2007.

[83] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. “Predicting and Preventing Inconsistencies in Deployed Distributed Systems.” In: *ACM Trans. Comput. Syst.* 28.1 (Aug. 2010), 2:1–2:49. ISSN: 0734-2071. DOI: 10.1145/1731060.1731062. URL: http://doi.acm.org/10.1145/1731060.1731062.

[84] Eli Gafni and Leslie Lamport. “Disk Paxos.” In: vol. 16. May 2003, pp. 1–20. URL: https://www.microsoft.com/en-us/research/publication/disk-paxos/.

[85] Iulian Moraru, David G. Andersen, and Michael Kaminsky. “There is More Consensus in Egalitarian Parliaments.” In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. event-place: Farminton, Pennsylvania. New York, NY, USA: ACM, 2013, pp. 358–372. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2517350. URL: http://doi.acm.org/10.1145/2517349.2517350.

[86] Diego Ongaro and John Ousterhout. “In Search of an Understandable Consensus Algorithm.” In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 305–319. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[87] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. “Towards Verification of the Pastry Protocol Using TLA+.” In: *Formal Techniques for Distributed Systems*. Ed. by Roberto Bruni and Juergen Dingel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 244–258. ISBN: 978-3-642-21461-5.

[88] Ant Rowstron and Peter Druschel. “Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems.” In: *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*. Vol. 2218. Springer Berlin Heidelberg, Nov. 2001, pp. 329–350. ISBN: 978-3-540-45518-9. URL: `https://www.microsoft.com/en-us/research/publication/pastry-scalable-distributed-object-location-and-routing-for-large-scale-peer-to-peer-systems/`.

[89] Noran Azmy, Stephan Merz, and Christoph Weidenbach. “A machine-checked correctness proof for Pastry.” In: *Science of Computer Programming* 158 (2018), pp. 64–80. ISSN: 0167-6423. DOI: `https://doi.org/10.1016/j.scico.2017.08.003`. URL: `http://www.sciencedirect.com/science/article/pii/S0167642317301612`.

[90] Cormac Flanagan and Patrice Godefroid. “Dynamic Partial-order Reduction for Model Checking Software.” In: *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’05. event-place: Long Beach, California, USA. New York, NY, USA: ACM, 2005, pp. 110–121. ISBN: 1-58113-830-X. DOI: `10.1145/1040305.1040315`. URL: `http://doi.acm.org/10.1145/1040305.1040315`.

[91] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. “Symbolic model checking: 10^{20} States and beyond.” In: *Information and Computation* 98.2 (1992), pp. 142–170. ISSN: 0890-5401. DOI: `https://doi.org/10.1016/0890-5401(92)90017-A`. URL: `http://www.sciencedirect.com/science/article/pii/089054019290017A`.

[92] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[93] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. “Bounded Model Checking Using Satisfiability Solving.” In: *Formal Methods in System Design* 19.1 (July 2001), pp. 7–34. ISSN: 1572-8102. DOI: `10.1023/A:1011276507260`. URL: `https://doi.org/10.1023/A:1011276507260`.

[94] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. *BmcMT: Bounded Model Checking of TLA+ Specifications with SMT*. Published: Presentation at the TLA+ Community Meeting, Oxford, UK, July. 2018. URL: `http://forsyte.at/wp-content/uploads/kkt18-bmcmt.pdf`.

[95] Anton Wijs, Thomas Neele, and Dragan Bošnački. “GPUexplore 2.0: Unleashing GPU Explicit-State Model Checking.” In: *FM 2016: Formal Methods*. Ed. by John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou. Cham: Springer International Publishing, 2016, pp. 694–701. ISBN: 978-3-319-48989-6.

[96] S. Cho, M. Ferdman, and P. Milder. “FPGASwarm: High Throughput Model Checking on FPGAs.” In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2018, pp. 435–4357. DOI: `10.1109/FPL.2018.00080`.

[97] Johannes Oetsch, Michael Prischink, Jörg Pührer, Martin Schwengerer, and Hans Tompits. “On the Small-scope Hypothesis for Testing Answer-set Programs.” In: *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning*. KR’12. event-place: Rome, Italy. AAAI Press, 2012, pp. 43–53. ISBN: 978-1-57735-560-1. URL: http://dl.acm.org/citation.cfm?id=3031843.3031849.

[98] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. “Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems.” In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 249–265. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan.

[99] Michael Leuschel and Michael Butler. “ProB: A Model Checker for B.” In: *FME 2003: Formal Methods*. Ed. by Keijiro Araki, Stefania Gnesi, and Dino Mandrioli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 855–874. ISBN: 978-3-540-45236-2.

[100] Dominik Hansen and Michael Leuschel. “Translating TLA+ to B for Validation with ProB.” In: *Integrated Formal Methods*. Ed. by John Derrick, Stefania Gnesi, Diego Latella, and Helen Treharne. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 24–38. ISBN: 978-3-642-30729-4.

[101] Y. Howard, S. Gruner, A. Gravell, C. Ferreira, and J. C. Augusto. “Model-Based Trace-Checking.” In: *arXiv e-prints* (Nov. 2011).

Appendix A: Specification of synchronous Chord

MODULE *SynchronousChord*

Models pure-join Chord with no node failures and synchronous communication, such that at most two nodes are communicating at any given time. This permits very straightforward definitions of actions, at the cost of introducing unrealistic assumptions.

```
EXTENDS $Integers$, $FiniteSets$, $Sequences$, $TLC$
CONSTANTS
    $N$,   The number of nodes in the model
    $UseInitialRing$   FALSE if we should examine all possible configurations of successor pointers
ASSUME $FiniteNodesAssumption \triangleq N \in Nat \setminus \{0\}$
ASSUME $UseInitialRingAssumption \triangleq UseInitialRing \in$ BOOLEAN
```

Use ascending natural numbers for the node identifiers, since the consistent hashing scheme is irrelevant to the stabilization protocol: like the original *SIGCOMM* paper, we assume that the key space is large enough that identifier collisions are unlikely.

```
$ProcSet \triangleq 1 \,..\, N$

VARIABLES $Successor$,
    $Predecessor$,
    $HasJoined$,
    $HasPredecessor$
$vars \triangleq \langle Successor, Predecessor, HasJoined, HasPredecessor \rangle$
```

We begin by defining convenience predicates to deal with identifier order.

BetweenInclusive is true if $x \in$ [a, b] (mod N).

```
$BetweenInclusive(x, a, b) \triangleq ((x - a)\%N) \leq ((b - a)\%N)$
```

BetweenExclusive is true if $x \in$ (a, b) (mod N)

```
$BetweenExclusive(x, a, b) \triangleq$
    $\lor \land ((x - a)\%N) \leq ((b - a)\%N) \land x \neq a \land x \neq b$
      Caveat: in the case where $a = b$, as in a one-node ring,
      we want to interpret (a,$b$) as spanning the entire interval:
    $\lor \land a = b$
```

Determine which nodes are reachable from node x by following successive successor pointers. Since *TLC* doesn't permit evaluating recursive operators over potentially infinite sequences, we explicitly bound the depth to the

number of reachable nodes, which is N for an N-node ring.

```
$ReachableSuccessors(x) \triangleq$
    LET RECURSIVE $Reachable(\_, \_)$
        $Reachable(n, i) \triangleq$
            IF $i = 0 \lor \neg HasJoined[n]$
             THEN $\{\}$
             ELSE $Reachable(Successor[n], i - 1) \cup \{Successor[n]\}$
    IN  $Reachable(x, N)$
```

Define the network safety invariants, as they are needed to determine
the set of initial states:

A node x forms a ring if it can reach itself through following successor pointers, starting from itself:

```
$FormsRing(x) \triangleq x \in ReachableSuccessors(x)$
```

A node is an appendage of the ring if it can reach at least one node forming
a ring by following successor pointers:

```
$IsRingAppendage(x) \triangleq$
    $\exists\, succ \in ReachableSuccessors(Successor[x]) : FormsRing(succ)$
```

The Chord network should have at least one ring:

```
$AtLeastOneRing \triangleq \exists\, x \in ProcSet : FormsRing(x)$
```

The Chord network should have at most one ring: the set of nodes reachable
through successor pointers should be equal for all nodes which form rings.

```
$AtMostOneRing \triangleq \neg\exists\, x \in ProcSet :$
    $\land\ FormsRing(x)$
    $\land\ \exists\, y \in ProcSet :$
        $\land\ FormsRing(y)$
        $\land\ ReachableSuccessors(x) \neq ReachableSuccessors(y)$
```

In an ordered ring, only one node should have a successor with a
smaller identifier than its own.

```
$OrderedRing \triangleq$
    LET $ring \triangleq$ CHOOSE $x \in ProcSet : FormsRing(x)$ IN
        $\land\ \forall\, self \in ReachableSuccessors(ring) :$
            $\lor\ self < Successor[self]$
            $\lor\ \neg\exists\, n \in ReachableSuccessors(ring) : n > Successor[n] \land self \neq n$
```

The four invariants above are necessary to maintain correctness:

```
$ValidRing \triangleq$
        $\land\ AtLeastOneRing$
        $\land\ AtMostOneRing$
        $\land\ \forall\, self \in ProcSet : IsRingAppendage(self)$
        $\land\ OrderedRing$
```

We allow any initial configuration which maintains the safety invariants.

For $N > 5$, *TLC* fails if we attempt to enumerate all possible combinations of successor and predecessor pointers to find initial states - the *UseInitialRing* setting instead uses a smaller set of initial states, where one node forms a ring and the rest join it.

```
$Init \triangleq$ $\land$ IF $UseInitialRing$
            THEN
            $\land\ \exists\, Start \in ProcSet :$
                $\land\ Successor = [self \in ProcSet \mapsto Start]$
            ELSE
```

```
        $\land$ $Successor \in [ProcSet \rightarrow ProcSet]$
    $\land$ $Predecessor = [self \in ProcSet \mapsto 1]$
    $\land$ $HasJoined \in [ProcSet \rightarrow \text{BOOLEAN}]$
    $\land$ $HasPredecessor = [self \in ProcSet \mapsto \text{FALSE}]$
    $\land$ $ValidRing$
```

Next, we define the actions making up the Chord stabilization protocol:

Nodes join the network by learning the successor node for their identifier from a member of the network, and setting it as their successor.

```
$Join(self) \triangleq$
     $\land \neg HasJoined[self]$
     $\land HasJoined' = [HasJoined \text{ EXCEPT } ![self] = \text{TRUE}]$
```

After joining, there are no nodes lying between this node's identifier and its successor's:

```
    $\land Successor' = [Successor \text{ EXCEPT } ![self] =$
        $\text{CHOOSE } succ \in \{Successor[self]\}$
            $\cup\ ReachableSuccessors(Successor[self]) :$
            $\land \neg\exists\, candidate \in ReachableSuccessors(self) :$
                $\land BetweenExclusive(self, candidate, succ)$
                $\land candidate \neq succ]$
    $\land \text{UNCHANGED } \langle Predecessor, HasPredecessor \rangle$
```

Each nodes periodically stabilize, setting the successor's predecessor as its new successor if it lies closer in identifier order.

```
$Stabilize(self) \triangleq$
    $\land HasJoined[self]$
    $\land HasPredecessor[Successor[self]]$
    $\land BetweenExclusive(Predecessor[Successor[self]], self, Successor[self])$
    $\land Successor' = [Successor \text{ EXCEPT } ![self] = Predecessor[Successor[self]]]$
    $\land \text{UNCHANGED } \langle HasJoined, HasPredecessor, Predecessor \rangle$
```

After a notification, a node similarly determines whether to adopt its predecessor's successor as its new predecessor:

```
$Notify(self) \triangleq$
    $\land HasJoined[self]$
    $\land$ $\lor \neg HasPredecessor[Successor[self]]$
      $\lor BetweenExclusive(self, Predecessor[Successor[self]], Successor[self])$
    $\land Predecessor' = [Predecessor \text{ EXCEPT } ![Successor[self]] = self]$
    $\land HasPredecessor' = [HasPredecessor \text{ EXCEPT } ![Successor[self]] = \text{TRUE}]$
    $\land \text{UNCHANGED } \langle Successor, HasJoined \rangle$
```

Joining, stabilizing and notifying are sufficient to specify the stabilization protocol:

```
$Node(self) \triangleq$
     $\lor Join(self)$
```

$$\begin{aligned}&\lor\ \mathit{Stabilize}(\mathit{self})\\&\lor\ \mathit{Notify}(\mathit{self})\end{aligned}$$

We define one safety invariant, the type invariant:

$$\begin{aligned}\mathit{TypeOK}\ \triangleq\ &\land \forall\, \mathit{self} \in \mathit{ProcSet} : \mathit{Successor}[\mathit{self}] \in \mathit{ProcSet}\\&\land \forall\, \mathit{self} \in \mathit{ProcSet} : \mathit{Predecessor}[\mathit{self}] \in \mathit{ProcSet}\\&\land \forall\, \mathit{self} \in \mathit{ProcSet} : \mathit{HasJoined}[\mathit{self}] \in \text{BOOLEAN}\\&\land \forall\, \mathit{self} \in \mathit{ProcSet} : \mathit{HasPredecessor}[\mathit{self}] \in \text{BOOLEAN}\end{aligned}$$

Next, we define liveness properties.

In a weakly ideal ring (as defined in the 2002 paper by Liben-Nowell et al.), every node is the predecessor of its successor, and the network is stable.

$$\begin{aligned}&\mathit{RingIsWeaklyIdeal}\ \triangleq\\&\quad \forall\, \mathit{self} \in \mathit{ProcSet} :\\&\qquad \land\ \mathit{HasJoined}[\mathit{self}]\\&\qquad \land\ \mathit{HasPredecessor}[\mathit{self}]\\&\qquad \land\ \mathit{Successor}[\mathit{Predecessor}[\mathit{self}]] = \mathit{self}\end{aligned}$$

$$\mathit{RingEventuallyBecomesWeaklyIdeal}\ \triangleq\ \Box(\mathit{ValidRing} \Rightarrow \Diamond\Box \mathit{RingIsWeaklyIdeal})$$

In a strongly ideal ring, there are additionally no nodes between a node and its successor:

$$\begin{aligned}&\mathit{RingIsStronglyIdeal}\ \triangleq\\&\quad \land\ \mathit{RingIsWeaklyIdeal}\\&\quad \land\ \forall\, \mathit{self} \in \mathit{ProcSet} :\\&\qquad \land\ \neg\exists\, v \in \mathit{ProcSet} : \mathit{BetweenExclusive}(v,\ \mathit{self},\ \mathit{Successor}[\mathit{self}])\end{aligned}$$

Zave's pure-join correctness theorem states that any valid network state will eventually progress to an ideal one, and remain in it:

$$\mathit{RingEventuallyBecomesStronglyIdeal}\ \triangleq\ \Box(\mathit{ValidRing} \Rightarrow \Diamond\Box \mathit{RingIsStronglyIdeal})$$

We add a disjunct to the next-state action to allow stuttering once the network is strongly ideal, avoiding a deadlock error:

$$\begin{aligned}&\mathit{Next}\ \triangleq\ (\exists\, \mathit{self} \in 1 \,..\, N : \mathit{Node}(\mathit{self}))\\&\quad \lor\ \land\ \mathit{RingIsStronglyIdeal} \land \text{UNCHANGED}\ \mathit{vars}\end{aligned}$$

We use weak fairness as the specification's fairness condition:

$$\begin{aligned}\mathit{Spec}\ \triangleq\ &\land\ \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}\\&\land\ \forall\, \mathit{self} \in 1 \,..\, N : \mathrm{WF}_{\mathit{vars}}(\mathit{Node}(\mathit{self}))\end{aligned}$$

Appendix B: Specification of pure-join Chord with asynchronous messaging

```
MODULE PureJoinChord

  Models the pure-join Chord stabilization algorithm
as specified in the 2001 SIGCOMM paper, using asynchronous communication between nodes.

EXTENDS $Integers, FiniteSets, Sequences, TLC$
CONSTANTS
    $N$,  Number of nodes in the model
    $UseInitialRing$  FALSE if we should examine all possible configurations of successor pointers
ASSUME $FiniteNodesAssumption \triangleq N \in Nat \setminus \{0\}$
ASSUME $UseInitialRingAssumption \triangleq UseInitialRing \in$ BOOLEAN

$ProcSet \triangleq 1 \,..\, N$

  We amend the previous specification to include unordered sets
representing the messages sent between nodes:

VARIABLES $Successor$,
    $Predecessor$,
    $HasJoined$,
    $HasPredecessor$,
    $SuccessorAnswers$,
    $SuccessorRequests$,
    $PredecessorAnswers$,
    $PredecessorRequests$,
    $Notifications$
$vars \triangleq \langle Successor, Predecessor, HasJoined, HasPredecessor,$
    $SuccessorAnswers, SuccessorRequests,$
    $PredecessorAnswers, PredecessorRequests,$
    $Notifications\rangle$

  BetweenInclusive is true if $x \in$ [a, $b$] (mod $N$).
$BetweenInclusive(x, a, b) \triangleq ((x - a)\%N) \leq ((b - a)\%N)$

  BetweenHalfOpen is true if $x \in$ (a, $b$] (mod $N$).
$BetweenHalfOpen(x, a, b) \triangleq$
    $\land ((x - a)\%N) \leq ((b - a)\%N) \land x \neq a$

  BetweenExclusive is true if $x \in$ (a, $b$) (mod $N$)
$BetweenExclusive(x, a, b) \triangleq$
    $\lor \land ((x - a)\%N) \leq ((b - a)\%N) \land x \neq a \land x \neq b$
    $\lor \land a = b$

  We keep the safety properties from the synchronous Chord specification
(see Appendix A):

$ReachableSuccessors(x) \triangleq$
    LET RECURSIVE $Reachable(\_, \_)$
        $Reachable(n, i) \triangleq$
            IF $i = 0 \lor \neg HasJoined[n]$
             THEN $\{\}$
            ELSE  $Reachable(Successor[n],\ i-1) \cup \{Successor[n]\}$
    IN   $Reachable(x,\ N)$

$FormsRing(x) \triangleq x \in ReachableSuccessors(x)$

$IsRingAppendage(x) \triangleq$
    $\exists\, succ \in ReachableSuccessors(Successor[x]) : FormsRing(succ)$

$ConnectedAppendages \triangleq$
    $\forall\, self \in ProcSet : IsRingAppendage(self)$

$AtLeastOneRing \triangleq \exists\, x \in ProcSet : FormsRing(x)$

$AtMostOneRing \triangleq \neg\exists\, x \in ProcSet :$
    $\land\ FormsRing(x)$
    $\land\ \exists\, y \in ProcSet :$
        $\land\ FormsRing(y)$
        $\land\ ReachableSuccessors(x) \neq ReachableSuccessors(y)$

$OrderedRing \triangleq$
    LET $ring \triangleq$ CHOOSE $x \in ProcSet : FormsRing(x)$IN
        $\land\ \forall\, self \in ReachableSuccessors(ring) :$
            $\lor\ self < Successor[self]$
            $\lor\ \neg\exists\, n \in ReachableSuccessors(ring) : n > Successor[n] \land self \neq n$

$ValidRing \triangleq$
    $\land\ AtLeastOneRing$
    $\land\ AtMostOneRing$
    $\land\ ConnectedAppendages$
    $\land\ OrderedRing$
```

To be a valid initial state, at least one node must have a predecessor pointer set. To ensure we only explore initial states which are reachable through normal operation, we also require that nodes' predecessor pointers are either unset, or lead to nodes leading back to them.

```
$ValidInitialPredecessors \triangleq$
    $\land\ \exists\, p \in ProcSet : HasPredecessor[p]$
    $\land\ \forall\, p \in ProcSet :$
        $\lor\ \neg HasPredecessor[p]$
        $\lor\ \land\ Successor[Predecessor[p]] = p$
           $\land\ HasJoined[Predecessor[p]]$
```

Since exploring every possible configuration of successor pointers quickly leads to a state space explosion, allow exploring a subset of the initial states where one node forms a single-node ring:

```
$InitialRing \triangleq$
    $\land\ \exists\, Start \in ProcSet :$
        $\land\ Successor = [self \in ProcSet \mapsto Start]$
        $\land$ $Predecessor = [self \in ProcSet \mapsto Start]$
        $\land$ $HasJoined = [self \in ProcSet \mapsto self = Start]$
        $\land$ $HasPredecessor = [self \in ProcSet \mapsto self = Start]$

$Init \triangleq$
    $\land$ IF $UseInitialRing$
          THEN $InitialRing$
          ELSE
          $\land$ $Successor \in [ProcSet \rightarrow ProcSet]$
          $\land$ $Predecessor \in [ProcSet \rightarrow ProcSet]$
          $\land$ $HasJoined \in [ProcSet \rightarrow \text{BOOLEAN}]$
          $\land$ $HasPredecessor \in [ProcSet \rightarrow \text{BOOLEAN}]$
    $\land$ $SuccessorAnswers = [self \in ProcSet \mapsto \{\}]$
    $\land$ $SuccessorRequests = [self \in ProcSet \mapsto \{\}]$
    $\land$ $PredecessorAnswers = [self \in ProcSet \mapsto \{\}]$
    $\land$ $PredecessorRequests = [self \in ProcSet \mapsto \{\}]$
    $\land$ $Notifications = [self \in ProcSet \mapsto \{\}]$
    $\land$ $ValidRing$
    $\land$ $ValidInitialPredecessors$
```

Next, we introduce new actions and amend previous ones to pass messages instead of retrieving the state directly.

Recursively find the successor for a given identifier.

```
$FindSuccessor(self) \triangleq$
    $\land$ $HasJoined[self]$
    $\land$ UNCHANGED $\langle Successor, Predecessor, HasJoined,$
       $HasPredecessor, PredecessorAnswers, PredecessorRequests,$
       $Notifications\rangle$
    $\land$ $\exists\, Request \in SuccessorRequests[self]$ :
      If the key lies in the interval $(n,\ successor(n)]$, the key belongs to node $n$'s successor:
      IF  $\lor$ $BetweenHalfOpen(Request.id,\ self,\ Successor[self])$
          $\lor$ $self = Successor[self]$
       THEN
          Remove this query from the set of requests.
          $\land$ $SuccessorRequests' = [SuccessorRequests$
              EXCEPT $![self] = @ \setminus \{Request\}]$
          Return the answer to the original sender.
          $\land$ $SuccessorAnswers' = [SuccessorAnswers$
              EXCEPT $![Request.origin] =$
             $@ \cup \{[id \mapsto Request.id,\ successor \mapsto Successor[self]]\}]$
       ELSE
          $\land$ $SuccessorRequests' = [SuccessorRequests$
              EXCEPT $![Successor[self]] =$
             $@ \cup \{Request\},\ ![self] = @ \setminus \{Request\}]$
```

```
                $\land$ $SuccessorAnswers' = SuccessorAnswers$
```

Join the network by requesting the successor of this node's identifier
from a known member of the network:

```
$Join(self) \triangleq$
    $\land \neg HasJoined[self]$
    $\land Cardinality(SuccessorAnswers[self]) = 0$
    $\land SuccessorRequests' =$
        $[SuccessorRequests$ EXCEPT $![Successor[self]] =$
        $@ \cup \{[origin \mapsto self, id \mapsto self]\}]$
    $\land$ UNCHANGED $\langle Successor, Predecessor, HasJoined,$
        $HasPredecessor, SuccessorAnswers, PredecessorAnswers,$
        $PredecessorRequests, Notifications\rangle$
```

On receiving the successor for this node's identifier,
set it as the successor:

```
$FinishJoin(self) \triangleq$
    $\land \neg HasJoined[self]$
    $\land$ UNCHANGED $\langle Predecessor, HasPredecessor,$
        $SuccessorRequests,$
        $PredecessorAnswers, PredecessorRequests,$
        $Notifications\rangle$
    $\land \exists Answer \in SuccessorAnswers[self] :$
        $\land Successor' = [Successor$ EXCEPT $![self] = Answer.successor]$
        $\land HasJoined' = [HasJoined$ EXCEPT $![self] =$ TRUE$]$
        $\land SuccessorAnswers' = [SuccessorAnswers$ EXCEPT $![self] = @ \setminus \{Answer\}]$
```

Periodically verify a node's immediate successor,
and tell the successor about the node, using the algorithm from Figure 7 of the *SIGCOMM* paper.

```
$Stabilize(self) \triangleq$
    $\land HasJoined[self]$
    $\land$ UNCHANGED $\langle Successor, Predecessor, HasJoined, HasPredecessor,$
        $SuccessorAnswers, SuccessorRequests,$
        $PredecessorAnswers,$
        $Notifications\rangle$
    $\land PredecessorRequests' = [PredecessorRequests$ EXCEPT $![Successor[self]] =$
        $@ \cup \{[origin \mapsto self]\}]$
```

Return the node's current predecessor.

```
$GetPredecessor(self) \triangleq$
    $\land HasJoined[self]$
    $\land HasPredecessor[self]$
    $\land$ UNCHANGED $\langle Successor, Predecessor, HasJoined, HasPredecessor,$
        $SuccessorAnswers, SuccessorRequests,$
        $Notifications\rangle$
```

If there is a request, return our current predecessor:

```
    $\land \exists\, Request \in PredecessorRequests[self]:$
        $\land PredecessorRequests' =$
            $[PredecessorRequests \text{ EXCEPT } ![self] = @ \setminus \{Request\}]$
        $\land PredecessorAnswers' =$
            $[PredecessorAnswers \text{ EXCEPT } ![Request.origin] =$
            $@ \cup \{[id \mapsto self,\ predecessor \mapsto Predecessor[self]]\}]$
```

Once the node has received its successor's predecessor,
determine whether it is a closer successor, adopting it as our new successor if it is.

```
$FinishStabilize(self) \triangleq$
    $\land HasJoined[self]$
    $\land \text{UNCHANGED } \langle Predecessor, HasJoined, HasPredecessor,$
        $SuccessorAnswers, SuccessorRequests,$
        $PredecessorRequests\rangle$
    $\land \exists\, Answer \in PredecessorAnswers[self]:$
        IF
            $\lor BetweenExclusive(Answer.predecessor, self, Successor[self])$
        THEN
            $\land Successor' = [Successor \text{ EXCEPT } ![self] = Answer.predecessor]$
            $\land PredecessorAnswers' =$
                $[PredecessorAnswers \text{ EXCEPT } ![self] = \{\}]$
            $\land Notifications' = [Notifications \text{ EXCEPT } ![Answer.predecessor] =$
                $@ \cup \{[origin \mapsto self]\}]$
        ELSE
            $\land Successor' = Successor$
            $\land PredecessorAnswers' =$
                $[PredecessorAnswers \text{ EXCEPT } ![self] = \{\}]$
            $\land Notifications' = [Notifications \text{ EXCEPT } ![Successor[self]] =$
                $@ \cup \{[origin \mapsto self]\}]$
```

Accept a notification from another node
which thinks it might be our predecessor.

```
$Notify(self) \triangleq$
    $\land HasJoined[self]$
    $\land \text{UNCHANGED } \langle Successor, HasJoined,$
        $SuccessorAnswers, SuccessorRequests,$
        $PredecessorAnswers, PredecessorRequests\rangle$
    $\land \exists\, Note \in Notifications[self]:$
        $\land$ IF
            $\lor \neg HasPredecessor[self]$
            $\lor BetweenExclusive(Note.origin, Predecessor[self], self)$
            THEN
            $\land HasPredecessor' = [HasPredecessor \text{ EXCEPT } ![self] = \text{TRUE}]$
            $\land Predecessor' = [Predecessor \text{ EXCEPT } ![self] = Note.origin]$
            $\land Notifications' = [Notifications \text{ EXCEPT } ![self] = @ \setminus \{Note\}]$
            ELSE
            $\land$ UNCHANGED $\langle HasPredecessor, Predecessor\rangle$
            $\land$ $Notifications' = [Notifications$ EXCEPT $![self] = @ \setminus \{Note\}]$

$Node(self) \triangleq$
      $\lor FindSuccessor(self)$
      $\lor Join(self)$
      $\lor FinishJoin(self)$
      $\lor Stabilize(self)$
      $\lor GetPredecessor(self)$
      $\lor FinishStabilize(self)$
      $\lor Notify(self)$

$Next \triangleq (\exists\, self \in 1\,..\,N : Node(self))$

$Spec \triangleq \land Init \land \Box[Next]_{vars}$
           $\land \forall\, self \in 1\,..\,N : \mathrm{WF}_{vars}(Node(self))$

$TypeOK \triangleq$
    $\land$   $\forall\, self \in ProcSet :$
           $\land Successor[self] \in ProcSet$
           $\land Predecessor[self] \in ProcSet$
           $\land HasJoined[self] \in$ BOOLEAN
           $\land HasPredecessor[self] \in$ BOOLEAN
           $\land IsFiniteSet(SuccessorAnswers[self])$
           $\land IsFiniteSet(SuccessorRequests[self])$
           $\land IsFiniteSet(PredecessorAnswers[self])$
           $\land IsFiniteSet(PredecessorRequests[self])$
           $\land IsFiniteSet(Notifications[self])$
```

As a sanity check, test that we can reach a weakly ideal ring.
Checking the *NoRingBecomesIdeal* invariant should return a violation.

```
$RingIsWeaklyIdeal \triangleq$
    $\forall\, self \in ProcSet :$
        $\land HasJoined[self]$
        $\land HasPredecessor[self]$
        $\land Successor[Predecessor[self]] = self$

$NoRingBecomesIdeal \triangleq \neg RingIsWeaklyIdeal$
```

Appendix C: Full specification of Chord with fault-tolerance

MODULE *Chord*2003

Models the pure-join Chord stabilization algorithm
as specified in the 2003 Transactions in Networking paper, with fault tolerance algorithms based on pseudocode from the 2002 analysis by Liben-Nowell et al. The specification models asynchronous communication
between nodes and fail-stop failures.

```
EXTENDS Integers, FiniteSets, Sequences, TLC
CONSTANTS
    N,   Number of nodes in the model
    SuccessorsPerNode,   Length of nodes' successor lists
    UseInitialRing   FALSE if we should examine all possible configurations of successor pointers
ASSUME FiniteNodesAssumption ≜ N ∈ Nat \ {0}
 Assume that the successor lists are of finite length:
ASSUME FiniteNumberOfSuccessors ≜ SuccessorsPerNode ∈ Nat \ {0}
ASSUME LessSuccessorsThanNodes ≜ SuccessorsPerNode ≤ N
ASSUME InitialRingBoolean ≜ UseInitialRing ∈ BOOLEAN
ProcSet ≜ 1 . . N

VARIABLES Successors,
    Predecessor,
    HasJoined,
    HasPredecessor,
    HasFailed,
    SuccessorAnswers,
    SuccessorRequests,
    PredecessorAnswers,
    PredecessorRequests,
    Notifications
vars ≜ ⟨Successors, Predecessor, HasJoined, HasPredecessor, HasFailed,
    SuccessorAnswers, SuccessorRequests,
    PredecessorAnswers, PredecessorRequests,
    Notifications⟩

 BetweenInclusive is true if x ∈ [a, b] (mod N).
BetweenInclusive(x, a, b) ≜ ((x − a)%N) ≤ ((b − a)%N)

 BetweenHalfOpen is true if x ∈ (a, b] (mod N).
BetweenHalfOpen(x, a, b) ≜
    ∧ ((x − a)%N) ≤ ((b − a)%N) ∧ x ≠ a

 BetweenExclusive is true if x ∈ (a, b) (mod N)
BetweenExclusive(x, a, b) ≜
    ∨ ∧ ((x − a)%N) ≤ ((b − a)%N) ∧ x ≠ a ∧ x ≠ b
    ∨ ∧ a = b

 The safety properties must be rephrased to take failing nodes into account.
```

HasLiveSuccessor is true if there is at least one live successor in the successor list of node x.

```
$HasLiveSuccessor(x) \triangleq$
    $\exists\, succ \in 1 \,..\, Len(Successors[x]) : \neg HasFailed[Successors[x][succ]]$
```

PickFirstLiveSuccessor is the index of the first live successor in the successor list of node x.

```
$PickFirstLiveSuccessor(x) \triangleq$
    CHOOSE $succ \in 1 \,..\, Len(Successors[x])$ :
              Pick a live successor,
              $\land \neg HasFailed[Successors[x][succ]]$
              such that there are no live successors earlier in the list:
              $\land \neg\exists\, cand \in 1 \,..\, Len(Successors[x])$ :
                  $\land \neg HasFailed[Successors[x][cand]]$
                  $\land cand < succ$
```

If node x is live, *FirstLiveSuccessor* is the identifier of the first live successor in the successor list of node x. If node x has failed, *FirstLiveSuccessor* is the node's own identifier.

```
$FirstLiveSuccessor(x) \triangleq$
    IF $\neg HasFailed[x]$
     THEN $Successors[x][PickFirstLiveSuccessor(x)]$
     ELSE $x$
```

ReachableSuccessors is now the set of nodes reachable from node x when failed nodes are pruned from the successor lists. If node x has failed, *ReachableSuccessors* is the empty set.

```
$ReachableSuccessors(x) \triangleq$
    LET RECURSIVE $Reachable(\_, \_)$
        $Reachable(n, i) \triangleq$
            IF $i = 0 \lor \neg HasJoined[n] \lor HasFailed[n] \lor \neg HasLiveSuccessor(n)$
             THEN $\{\}$
             ELSE $Reachable(FirstLiveSuccessor(n), i - 1)$
                $\cup \{FirstLiveSuccessor(n)\}$
    IN  $Reachable(x, N)$

$FormsRing(x) \triangleq$
    $\land HasLiveSuccessor(x)$
    $\land x \in ReachableSuccessors(x)$

$IsRingAppendage(x) \triangleq$
    $\land HasLiveSuccessor(x)$
    $\land \exists\, succ \in ReachableSuccessors(FirstLiveSuccessor(x)) : FormsRing(succ)$
```

We disregard failed nodes when determining whether property (1c) is satisfied:

```
$ConnectedAppendages \triangleq \forall self \in ProcSet :$
    $\lor IsRingAppendage(self)$
    $\lor HasFailed[self]$

$AtLeastOneRing \triangleq \exists x \in ProcSet : FormsRing(x) \land \neg HasFailed[x]$

$AtMostOneRing \triangleq \neg\exists x \in ProcSet :$
    $\land FormsRing(x)$
    $\land \exists y \in ProcSet :$
        $\land FormsRing(y)$
        $\land ReachableSuccessors(x) \neq ReachableSuccessors(y)$
```

In an ordered ring, only one node should have a live successor with a smaller identifier than its own.

```
$OrderedRing \triangleq$
    $\land$ LET $ring \triangleq$ CHOOSE $x \in ProcSet : FormsRing(x) \land \neg HasFailed[x]$ IN
        $\land \forall self \in ReachableSuccessors(ring) :$
            $\lor self < Head(Successors[self])$
            $\lor HasFailed[Head(Successors[self])]$
            $\lor \neg\exists n \in ReachableSuccessors(ring) :$
                $\land n > Head(Successors[n])$
                $\land self \neq n$
                $\land \neg HasFailed[Head(Successors[n])]$

$ValidRing \triangleq$
    $\land AtLeastOneRing$
    $\land AtMostOneRing$
    $\land ConnectedAppendages$
    $\land OrderedRing$

$ValidInitialPredecessors \triangleq$
    $\land \exists p \in ProcSet : HasPredecessor[p]$
    $\land \forall p \in ProcSet :$
        $\lor \neg HasPredecessor[p]$
        $\lor \land Head(Successors[Predecessor[p]]) = p$
          $\land HasJoined[Predecessor[p]]$

$InitialRing \triangleq$
    $\land \exists Start \in ProcSet :$
        $\land Successors = [self \in ProcSet \mapsto \langle Start \rangle]$
        $\land Predecessor = [self \in ProcSet \mapsto Start]$
        $\land HasJoined = [self \in ProcSet \mapsto self = Start]$
        $\land HasPredecessor = [self \in ProcSet \mapsto self = Start]$

$Init \triangleq$
    $\land$ IF $UseInitialRing$
        THEN $InitialRing$
        ELSE
        $\land$ $Successors \in$
            $\{[p \in ProcSet \mapsto \langle initialSuccessor[p] \rangle] :$
            $initialSuccessor \in [ProcSet \rightarrow ProcSet]\}$
        $\land$ $Predecessor \in [ProcSet \rightarrow ProcSet]$
        $\land$ $HasJoined \in [ProcSet \rightarrow \text{BOOLEAN}]$
        $\land$ $HasPredecessor \in [ProcSet \rightarrow \text{BOOLEAN}]$
    $\land$ $HasFailed = [self \in ProcSet \mapsto \text{FALSE}]$
    $\land$ $SuccessorAnswers = [self \in ProcSet \mapsto \{\}]$
    $\land$ $SuccessorRequests = [self \in ProcSet \mapsto \{\}]$
    $\land$ $PredecessorAnswers = [self \in ProcSet \mapsto \{\}]$
    $\land$ $PredecessorRequests = [self \in ProcSet \mapsto \{\}]$
    $\land$ $Notifications = [self \in ProcSet \mapsto \{\}]$
    $\land$ $ValidRing$
    $\land$ $ValidInitialPredecessors$
```

Recursively find the successor for a given identifier.

```
$FindSuccessor(self) \triangleq$
    $\land \neg HasFailed[self]$
    $\land \neg HasFailed[Head(Successors[self])]$
    $\land HasJoined[self]$
    $\land$ UNCHANGED $\langle Successors, Predecessor, HasJoined,$
       $HasPredecessor, HasFailed, PredecessorAnswers, PredecessorRequests,$
       $Notifications\rangle$
    $\land \exists Request \in SuccessorRequests[self] :$
       IF  $\lor BetweenHalfOpen(Request.id, self, Head(Successors[self]))$
           $\lor self = Head(Successors[self])$
        THEN
           $\land SuccessorRequests' = [SuccessorRequests$
               EXCEPT $![self] = @ \setminus \{Request\}]$
           $\land SuccessorAnswers' = [SuccessorAnswers$
               EXCEPT $![Request.origin] =$
             $@ \cup \{[id \mapsto Request.id, successor \mapsto Head(Successors[self])]\}]$
        ELSE
           $\land SuccessorRequests' = [SuccessorRequests$
               EXCEPT $![Head(Successors[self])] =$
             $@ \cup \{Request\}, ![self] = @ \setminus \{Request\}]$
           $\land SuccessorAnswers' = SuccessorAnswers$

$Join(self) \triangleq$
     $\land \neg HasFailed[Head(Successors[self])]$
     $\land \neg HasJoined[self]$
     $\land Cardinality(SuccessorAnswers[self]) = 0$
     $\land SuccessorRequests' =$
        $[SuccessorRequests$ EXCEPT $![Head(Successors[self])] =$
        $@ \cup \{[origin \mapsto self,\ id \mapsto self]\}]$
    $\land$ UNCHANGED $\langle Successors, Predecessor, HasJoined, HasFailed,$
        $HasPredecessor, SuccessorAnswers, PredecessorAnswers,$
        $PredecessorRequests, Notifications\rangle$
```

On receiving the successor for this node's identifier,
set it as the successor:

```
$FinishJoin(self) \triangleq$
    $\land \neg HasJoined[self]$
    $\land$ UNCHANGED $\langle Predecessor, HasPredecessor, HasFailed,$
        $SuccessorRequests,$
        $PredecessorAnswers, PredecessorRequests,$
        $Notifications\rangle$
    $\land \exists\, Answer \in SuccessorAnswers[self] :$
```

Ensure the node is live before adopting it as a successor.
Necessary to avoid violating *ConnectedAppendages*.

```
        $\land \neg HasFailed[Answer.successor]$
        $\land Successors' = [Successors$ EXCEPT $![self] = \langle Answer.successor\rangle]$
        $\land HasJoined' = [HasJoined$ EXCEPT $![self] =$ TRUE$]$
        $\land SuccessorAnswers' = [SuccessorAnswers$ EXCEPT $![self] = @ \setminus \{Answer\}]$
```

Periodically verify a node's immediate successor,
and tell the successor about the node, using the algorithm from Figure 7 of the *SIGCOMM* paper.

```
$Stabilize(self) \triangleq$
    $\land \neg HasFailed[Head(Successors[self])]$
    $\land HasJoined[self]$
    $\land$ UNCHANGED $\langle Successors, Predecessor, HasJoined, HasPredecessor,$
        $HasFailed, SuccessorAnswers, SuccessorRequests,$
        $PredecessorAnswers,$
        $Notifications\rangle$
    $\land PredecessorRequests' = [PredecessorRequests$ EXCEPT $![Head(Successors[self])] =$
        $@ \cup \{[origin \mapsto self]\}]$
```

Return the node's current predecessor.

```
$GetPredecessor(self) \triangleq$
    $\land HasJoined[self]$
    $\land HasPredecessor[self]$
    $\land$ UNCHANGED $\langle Successors, Predecessor, HasJoined, HasPredecessor,$
        $HasFailed, SuccessorAnswers, SuccessorRequests,$
        $Notifications\rangle$
    $\land \exists\, Request \in PredecessorRequests[self] :$
        $\land PredecessorRequests' =$
            $[PredecessorRequests$ EXCEPT $![self] = @ \setminus \{Request\}]$
        $\land PredecessorAnswers' =$
            $[PredecessorAnswers$ EXCEPT $![Request.origin] =$
        $@ \cup \{[id \mapsto self, predecessor \mapsto Predecessor[self]]\}]$

Once the node has received its successor's predecessor,
determine whether it is a closer successor, adopting it as our new successor if it is.
$FinishStabilize(self) \triangleq$
    $\land HasJoined[self]$
    $\land$ UNCHANGED $\langle Predecessor, HasJoined, HasPredecessor, HasFailed,$
        $SuccessorAnswers, SuccessorRequests,$
        $PredecessorRequests\rangle$
    $\land \exists Answer \in PredecessorAnswers[self] :$
        Ensure the node is live before adopting it as a successor.
        Necessary to avoid violating AtLeastOneRing.
        $\land \neg HasFailed[Answer.predecessor]$
        $\land$ IF $BetweenExclusive(Answer.predecessor, self, Head(Successors[self]))$
            THEN
                Clear the successor list on adopting a new successor.
                $\land Successors' = [Successors$ EXCEPT $![self] =$
                    $\langle Answer.predecessor\rangle]$
            To keep old successor list entries when stabilizing,
            replace the lines above with
            $\land Successors' = [Successors$ EXCEPT $![self] =$
                $\langle Answer.predecessor\rangle \circ Tail(Successors[self])]$
                $\land PredecessorAnswers' =$
                    $[PredecessorAnswers$ EXCEPT $![self] = \{\}]$
                $\land Notifications' = [Notifications$ EXCEPT $![Answer.predecessor] =$
                    $@ \cup \{[origin \mapsto self]\}]$
            ELSE
                $\land Successors' = Successors$
                $\land PredecessorAnswers' =$
                    $[PredecessorAnswers$ EXCEPT $![self] = \{\}]$
                $\land Notifications' = [Notifications$ EXCEPT $![Head(Successors[self])] =$
                    $@ \cup \{[origin \mapsto self]\}]$

Accept a notification from another node
which thinks it might be our predecessor.
$Notify(self) \triangleq$
    $\land HasJoined[self]$
    $\land$ UNCHANGED $\langle Successors, HasJoined, HasFailed,$
        $SuccessorAnswers, SuccessorRequests,$
        $PredecessorAnswers, PredecessorRequests\rangle$
    $\land \exists Note \in Notifications[self] :$
        $\land \neg HasFailed[Note.origin]$
        $\land$ IF
            $\lor \neg HasPredecessor[self]$
            $\lor BetweenExclusive(Note.origin, Predecessor[self], self)$
        THEN
        $\land$ $HasPredecessor' = [HasPredecessor$ EXCEPT $![self] =$ TRUE$]$
        $\land$ $Predecessor' = [Predecessor$ EXCEPT $![self] = Note.origin]$
        $\land$ $Notifications' = [Notifications$ EXCEPT $![self] = @ \setminus \{Note\}]$
        ELSE
        $\land$ UNCHANGED $\langle Successors, HasPredecessor, Predecessor \rangle$
        $\land$ $Notifications' = [Notifications$ EXCEPT $![self] = @ \setminus \{Note\}]$
```

Instead of specifying fail-stop failures by permitting stuttering, we add a separate action which verifies that the failure should be possible to recover from, and marks the node as failed:

```
$Fail(self) \triangleq$
```

For a failure to be recoverable, every node except the failing one should have at least one live node in its successor list after the failure:

```
    $\land$ $\forall member \in ProcSet :$
        $\lor$ $member = self$
        $\lor$ $\exists x \in 1 \,..\, Len(Successors[member]) :$
            $\land$ $Successors[member][x] \neq self$
            $\land$ $\neg HasFailed[Successors[member][x]]$
    $\land$ $HasFailed' = [HasFailed$ EXCEPT $![self] =$ TRUE$]$
    $\land$ UNCHANGED $\langle Successors, Predecessor, HasJoined, HasPredecessor,$
        $SuccessorAnswers, SuccessorRequests,$
        $PredecessorAnswers, PredecessorRequests,$
        $Notifications \rangle$
```

FixPredecessor (*AKA* "flush") detects that a node's predecessor has failed, and marks it as no longer having a predecessor:

```
$FixPredecessor(self) \triangleq$
    $\land$ $HasFailed[Predecessor[self]]$
    $\land$ $HasJoined[self]$
    $\land$ $HasPredecessor[self]$
    $\land$ $HasPredecessor' = [HasPredecessor$ EXCEPT $![self] =$ FALSE$]$
    $\land$ UNCHANGED $\langle Successors, Predecessor, HasJoined,$
        $HasFailed, SuccessorAnswers, SuccessorRequests,$
        $PredecessorAnswers, PredecessorRequests,$
        $Notifications \rangle$
```

FixSuccessor (*AKA* "update") detects that a node's immediate successor has failed, and replaces it with the next successor in the list:

```
$FixSuccessor(self) \triangleq$
    $\land$ $HasJoined[self]$
    $\land$ $HasFailed[Head(Successors[self])]$
    $\land$ $Successors' = [Successors$ EXCEPT $![self] = Tail(Successors[self])]$
    $\land$ UNCHANGED $\langle Predecessor, HasJoined, HasPredecessor, HasFailed,$
```

```
        $SuccessorAnswers, SuccessorRequests,$
        $PredecessorAnswers, PredecessorRequests,$
        $Notifications\rangle$
```

FixSuccessorList (*AKA* "reconcile")
periodically fetches the successor list from the immediate successor, prepends the successor to it, and adopts it as the node's own successor list:

```
$FixSuccessorList(self) \triangleq$
    $\land \neg HasFailed[Head(Successors[self])]$
    $\land HasJoined[self]$
    $\land Head(Successors[self]) \neq self$
    $\land$ LET
        $SuccessorListLength \triangleq$
            IF $Len(Successors[Head(Successors[self])]) < SuccessorsPerNode$
             THEN $Len(Successors[Head(Successors[self])])$
             ELSE $SuccessorsPerNode - 1$
        $NextSuccessorList \triangleq Successors[Head(Successors[self])]$
      IN
            $\land Head(NextSuccessorList) \neq Head(Successors[self])$
            $\land Successors' = [Successors$ EXCEPT $![self] =$
                $\langle Head(Successors[self])\rangle \circ$
                $SubSeq(NextSuccessorList, 1, SuccessorListLength)]$
    $\land$ UNCHANGED $\langle Predecessor, HasJoined, HasPredecessor, HasFailed,$
        $SuccessorAnswers, SuccessorRequests,$
        $PredecessorAnswers, PredecessorRequests,$
        $Notifications\rangle$

$Node(self) \triangleq$
```

We introduce a conjunct to ensure that failed nodes cannot perform any actions:

```
    $\land \neg HasFailed[self]$
    $\land$  $\lor FindSuccessor(self)$
        $\lor Join(self)$
        $\lor FinishJoin(self)$
        $\lor Stabilize(self)$
        $\lor GetPredecessor(self)$
        $\lor FinishStabilize(self)$
        $\lor Notify(self)$
        $\lor Fail(self)$
        $\lor FixPredecessor(self)$
        $\lor FixSuccessor(self)$
        $\lor FixSuccessorList(self)$

$RingIsWeaklyIdeal \triangleq$
    $\forall self \in ProcSet :$
        $\lor HasFailed[self]$
        $\lor$ $\land$ $HasJoined[self]$
           $\land$ $HasJoined[Predecessor[self]]$
           $\land$ $Head(Successors[Predecessor[self]]) = self$

$Next \triangleq (\exists\, self \in 1 \,..\, N : Node(self))$
    $\lor$ $\land$ $RingIsWeaklyIdeal \land$ UNCHANGED $vars$

$Spec \triangleq$ $\land$ $Init \land \Box[Next]_{vars}$
         $\land$ $\forall\, self \in 1 \,..\, N : \mathrm{WF}_{vars}(Node(self))$

$TypeOK \triangleq$
    $\land$ $\forall\, self \in ProcSet :$
           $\land$ $Len(Successors[self]) > 0$
           $\land$ $Len(Successors[self]) \leq SuccessorsPerNode$
           $\land$ $\forall\, s \in$ DOMAIN $Successors[self] : Successors[self][s] \in ProcSet$
           $\land$ $Predecessor[self] \in ProcSet$
           $\land$ $HasJoined[self] \in$ BOOLEAN
           $\land$ $HasPredecessor[self] \in$ BOOLEAN
           $\land$ $IsFiniteSet(SuccessorAnswers[self])$
           $\land$ $IsFiniteSet(SuccessorRequests[self])$
           $\land$ $IsFiniteSet(PredecessorAnswers[self])$
           $\land$ $IsFiniteSet(PredecessorRequests[self])$
           $\land$ $IsFiniteSet(Notifications[self])$

$NoRingBecomesIdeal \triangleq \neg RingIsWeaklyIdeal$
```

Appendix D: State space and average runtimes of model-checking

N	Total states	Distinct states
2	33	13
3	224	84
4	2 111	682
5	26 567	7 024
6	401 572	87 407

Table 1: The number of states explored while verifying the synchronous Chord specification.

N	Avg. runtime (safety)	σ	Avg. runtime (liveness)	σ
2	$0.88s$	$0.02s$	$0.94s$	$0.01s$
3	$0.95s$	$0.01s$	$1.12s$	$0.01s$
4	$1.21s$	$0.02s$	$1.82s$	$0.04s$
5	$1.93s$	$0.04s$	$5.35s$	$0.19s$
6	$6.93s$	$0.05s$	$52.53s$	$2.57s$

Table 2: Comparison of the runtimes of model-checking the synchronous Chord specification for safety properties only, against verifying both liveness and safety properties.

N	L	Total states	Distinct states
2	20	37 517	1 312
2	25	100 684	1 872
2	30	177 574	1 936
3	20	437 266	27 282
3	25	2 506 629	125 824
3	30	12 163 540	477 690
4	20	3 034 213	216 384
4	25	26 968 628	1 622 588
4	30	209 402 748	10 592 532
5	20	16 222 521	1 146 755
5	25	187 111 209	11 904 540
5	30	1 974 722 673	107 611 235
6	20	70 692 663	4 815 750
6	25	992 188 816	63 801 288
6	30	OOM	OOM

Table 3: States explored when model-checking the asynchronous Chord specification for N nodes, and behaviors of at most L steps. In rows marked with OOM, TLC terminated with an out-of-memory error.

N	L	Avg. runtime	σ
2	20	0.89s	0.02s
2	25	1.07s	0.03s
2	30	1.21s	0.04s
3	20	1.61s	0.04s
3	25	3.84s	0.10s
3	30	13.46s	0.29s
4	20	4.70s	0.06s
4	25	30.19s	0.14s
4	30	227.21s	4.43s
5	20	19.76s	0.10s
5	25	217.30s	2.70s
5	30	2295.16s	12.39s
6	20	87.83s	0.55s
6	25	1223.78s	4.34s
6	30	OOM	OOM

Table 4: Average runtimes of model-checking the pure-join Chord specification with asynchronous messaging, using N nodes and traces of at most L steps.

N	L	Total states	Distinct states
2	20	131 371	8 205
2	25	612 782	29 519
2	30	1 690 334	53 588
3	20	2 246 573	157 524
3	25	23 455 738	1 510 831
3	30	199 674 510	12 646 103
4	20	24 236 717	1 581 652
4	25	378 960 098	21 606 228
4	30	4 921 828 001	239 886 960
5	20	191 830 736	11 142 260
5	25	3 875 024 159	205 088 440
5	30	OOM	OOM
6	20	1 246 704 792	64 090 212
6	25	OOM	OOM
6	30	OOM	OOM

Table 5: States explored when model-checking the full Chord specification with fault-tolerance, using N nodes and behaviors of at most L steps. In rows marked with OOM, TLC terminated with an out-of-memory error.

N	L	Avg. runtime	σ
2	20	$1.32s$	$0.04s$
2	25	$1.99s$	$0.06s$
2	30	$3.40s$	$0.25s$
3	20	$4.31s$	$0.07s$
3	25	$32.83s$	$0.30s$
3	30	$217.14s$	$3.20s$
4	20	$37.65s$	$0.63s$
4	25	$548.45s$	$9.07s$
4	30	$7263.81s$	$105.43s$
5	20	$310.06s$	$2.42s$
5	25	$6181.17s$	$45.69s$
5	30	OOM	OOM
6	20	$2198.87s$	$41.83s$
6	25	OOM	OOM
6	30	OOM	OOM

Table 6: Average runtimes of model-checking the full Chord specification with fault-tolerance mechanisms, using N nodes and traces of at most L steps.