Formal verification of fault-tolerant peer-to-peer networks

June 1, 2022

Submitted by:

Viktor Strate Kløvedal
vistr19@student.sdu.dk

Supervisor:

Marco Peressotti
peressotti@imada.sdu.dk

Department of Mathematics and Computer Science

Resumé

I dette projekt undersøger vi protokollen Chord som modellerer en distribueret hashtabel i et peer-to-peer netværk. Vi går i dybden med hvordan protokollen kan stabilisere sig selv efter knuder i netværket svigter. I anden del kigger vi på hvordan Chord protokollen med svigtene knuder kan opstilles som en formel specifikation med værktøjet TLA$^+$. Vores specifikation opbygges som en udvidelse af de specifikationer, af den basale Chord protokol uden knudesvigt, som er udarbejdet i [1]. Til sidst bruger vi modeltjekker-værktøjet TLC til at verificere at vores specifikation er korrekt, og undersøger resultaterne herfra.

Contents

Chapter 1

Introduction

Distributed systems play a critical role in modern server infrastructure, where services are expected to handle thousands of requests per second, to be highly available and respond quickly to requests from all over the world.

But making distributed systems robust and error-free is notoriously hard as the number of possible states and interactions between just a few actors quickly expand exponentially to the thousands or even millions, which makes the system very hard to reason about and debug. Traditional software testing methods, such as unit and integration testing, do not apply well to distributed systems, as they are not well suited for testing these many possible interactions.

Formal specifications try to solve this problem by separating the design of the system from the implementation. A formal specification describes the design of a system in a higher level than code would, but it is precise enough that it can be verified by a model checker, a computer program that explores all possible states in the model and checks that the specified properties and invariants always hold. This not only helps to test the validity of the system, but it also forms as documentation of the system which can be used by engineers to implement or validate existing implementations.

In the thesis by [1] they introduce a formal specification for a distributed hash table (DHT) using the modeling language TLA$^+$ [3]. They model the fundamental routing properties of the Chord algorithm as well as the finger table used for scalable lookups. These models assume that there are no failures in the system, that is nodes will always be available and will respond to all requests.

In practice distributed systems must be able to handle node failures as it is inevitable for nodes to fail, especially as the network scales. This is often done by introducing redundancy into the system, such that when a node fails the rest of the system has the proper knowledge needed to continue. But this is also one of the strengths of a distributed system as it is what makes it highly available.

In this project, we expand upon the specification made by [1] to model failures in the system. The Chord paper [4] proposes a method to account for failures, by storing a list of successor nodes in such a way that as long as one of the nodes in the list is available, the system can always recover.

Chapter 2

The Chord protocol

The Chord protocol, proposed in [4], models a distributed hash table (DHT). Each node in the network forms a ring where nodes only know about their immediate successor and predecessor. Each node is associated with a key in the keyspace of the hash table, typically a hash of its IP address. The node is then responsible for all the keys that lie between the key of its predecessor and its own key.

2.1 Simple lookup

At its core, the Chord protocol only provides a single functionality to the user, the i.LOOKUP(key) function, which finds the data object associated with the given key in the network.

Since each node knows its successor and predecessor, it can determine whether itself is responsible for the key by checking if key lies between the key of its predecessor and itself. If that is the case, the node has the data locally and can return it directly. Otherwise, it instead forwards the request to its successor, where the operation is performed recursiveley until the responsible node is found.

Figure 2.1 shows an example of performing lookup in a Chord ring with 7 nodes. The lookup action is initiated at node N28, looking for key K8. Since the key does not lie between the predecessor of the node (N23) and the node itself, it forwards the request to its successor, N63. The request is propagated through the ring until it reaches N18, who is responsible for the keyspace $(5, 18]$ where the key lies within and the key is now located.

Internally Chord uses the similar i.LOCATESUCCESSOR(key) function, which is shown in alg. 1. It does the same as LOOKUP, but it instead returns the successor of the node responsible for the key. This function is used when new nodes join the network and when stabilizing.

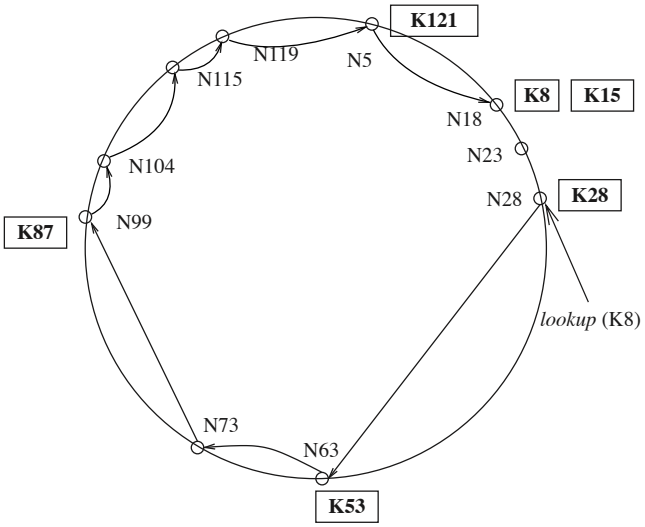


Figure 2.1: An example of simple lookup in a Chord ring. From fig. 18.2 in [2].

Algorithm 1 Based on alg. 18.1 in [2].
Algorithm executed at node i, that locates the successor to the *key*.

```
Variables: successor, predecessor

function i.LocateSuccessor(key)
    if key ∈ (i, successor] then
        return successor
    else
        return successor.LocateSuccessor(key)
    end if
end function
```

Scalable lookup Besides the simple lookup algorithm which requires $O(n)$ network hops, there exists a scalable lookup function that using a finger table reduces the required hops to $O(\log n)$. This is however not the focus of this project and will not be expanded further upon.

2.2 Churn

Churn relates to nodes joining and leaving the network dynamically. Since the system is distributed, all nodes will not always have a complete picture of all the nodes in the network. Therefore it is important that the ring is kept in a safe state even when new nodes are in the process of joining and leaving.

Definition 1 (Safe ring[1]) *A ring is safe when:*

1. *Each node's successor is correctly maintained.*
2. *For every key k, node $successor(k)$ is responsible for k.*

This is made possible by utilizing the fact that lookup in the ring always goes around in the same direction following each nodes successor.

This means that a new node can join the ring without its immediate nodes knowing about it, and the ring will still be safe, that is the i.LOOKUP(key) function still works for all $i \in Nodes$.

A node i joins the ring by asking some node j that is already a part of the ring, to locate the successor of the joining node i. Node i will set its successor variable to the returned node and thus add itself as an appendage to the ring. At this point the appending node has not fully joined the ring but is still able to perform the i.LOCATESUCCESSOR(key) correctly.

Algorithm 2 Based on Alg 18.3 in [2].
Function for node i to join the ring, given any node j already in the ring.

Variables: *successor*, *predecessor*

function i.JOINRING(j) ▷ where j is any node on the ring to be joined
  predecessor $\leftarrow \perp$
  successor $\leftarrow j$.LOCATESUCCESSOR(i)
end function

2.3 Stabilization

Each node periodically executes the stabilize function (alg. 3). This function first asks its successor for its predecessor, ideally this should be the same node as the one performing the stabilization. But in cases where the ring is not ideal, the successing node does not necessarily know about this node. For example when the node has just joined the network by executing i.JOINRING(j).

If x lies exclusiveley between i and the node which i believes to be its successor, then x is closer to i than its current successor and i makes x its new successor. Lastly i calls the NOTIFY function on its successor node, notifying its successor that it thinks it is its predecessor.

The NOTIFY function on the other node then checks if the calling node is a better predecessor than its current one and transfers the keys that the new predecessor should be responsible for, as well as updates its predecessor variable.

[1] The two invariants are proposed in [4], chapter 4.4

Algorithm 3 Based on Alg 18.3 in [2].
Stabilize function that is executed periodically.

```
Variables: successor, predecessor

function i.STABILIZE
    x ← successor.predecessor
    if x ∈ (i, successor) then
        successor ← x
    end if
    successor.NOTIFY(i)
end function

function i.NOTIFY(j)                    ▷ j believes it is predecessor of i
    if predecessor = ⊥ or j ∈ (predecessor, i) then
        transfer keys in range [j, i) to node j
        predecessor ← j
    end if
end function
```

The stabilize function ensures that existing nodes eventually learn about new nodes, and that new nodes become a part of the ring, while keeping the ring safe at all times.

2.4 Failures

The algorithms described earlier in this chapter has all assumed no failures in the network. To make the protocol fault-tolerant, the paper [4] proposes that each node, instead of maintaining only its direct successor, instead maintains a list of its first k successors. This way, if a node suddenly cannot reach its successor it can try the next in its list until a reachable replacement is located. This means that LOCATESUCCESSOR becomes LOCATESUCCESSORS (plural) and now returns a successor list rather than a single one (see appendix A).

The new FAULTSTABILIZE will now have to account for the possibility of nodes being unreachable. This is done by trying to reach each successor in the *successors* list from the beginning. If the node does not respond within a predetermined time, the request will timeout and the next successor in the list will be used instead. When the first successor responds, the function proceeds in much of the same way as the STABILIZE function proposed earlier in alg. 3. Except that it updates the whole successor list rather than a single value.

The i.CHECKPREDECESSOR function is also introduced, which is also executed periodically and makes sure that node i's predecessor is still active. It simply checks if the predecessor is still reachable and removes it if that is not the case.

Algorithm 4 Fault-tolerant stabilize.

```
Variables: successors[1..k], predecessor

▷ returns predecessor and successors in a single request
function i.PredecessorAndSuccessors
    return (predecessor, successors)
end function

function i.FaultStabilize                                   ▷ executed periodically
    for count ← 1 ... k do
        try
            s ← successors[count]
            (p, ss) ← s.PredecessorAndSuccessors
            if p ∈ (i, s) then
                successors ← [ p ]
            else
                successors ← [ s | ss[1..(k − 1)] ]
            end if
            successors[1].Notify(i)
            return
        after timeout continue to next loop iteration
    end for
    raise "no reachable successors"
end function

function i.CheckPredecessor                                 ▷ executed periodically
    if predecessor has failed then
        predecessor ← ⊥
    end if
end function
```

2.4.1 Node failure example

Fig. 2.2 shows a diagram consisting of 6 states that demonstrates an example of a node failing and the steps the system takes to recover from it.

(a) Initially the system is in an ideal state. Each node's successor list correctly points to its immediate successor and the next successor after that.

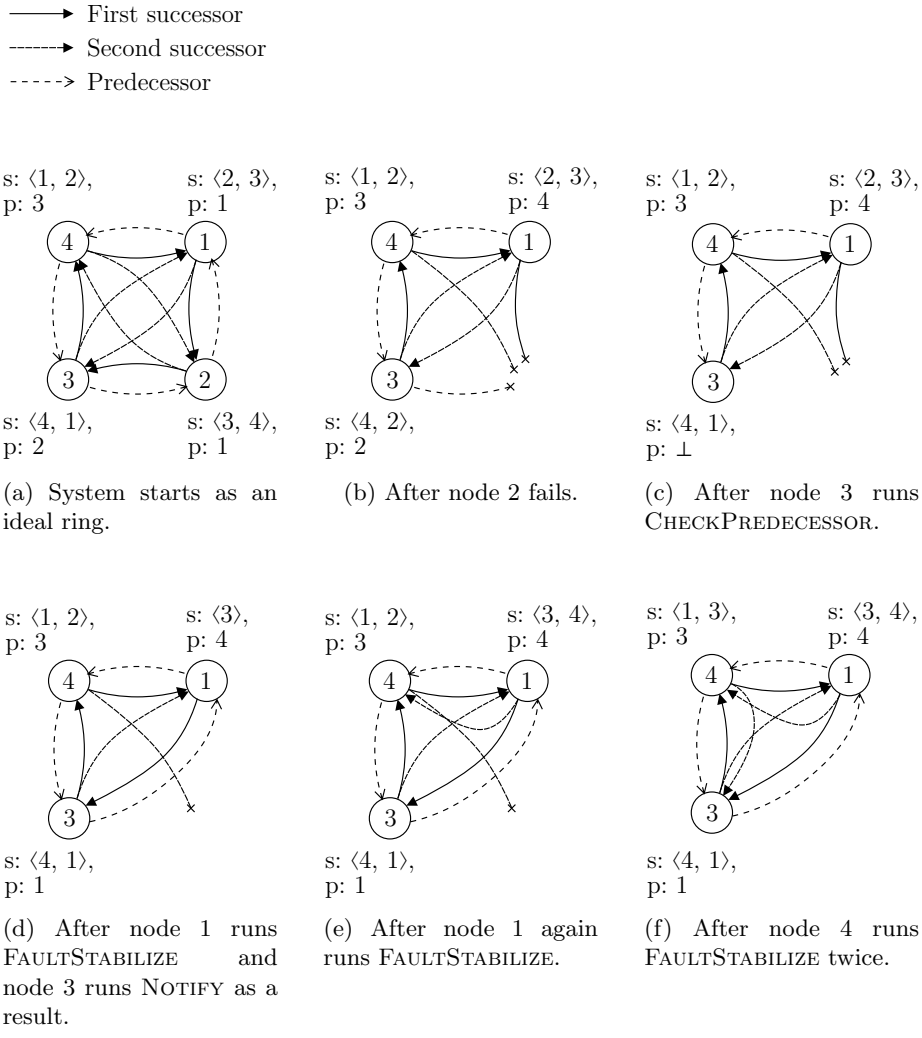
(b) Node 2 fails and the rest of the network does not know about this yet. The remaining nodes will eventually recover and become ideal again after enough executions of FaultStabilize and CheckPredecessor.

(c) Node 3 performs CheckPredecessor and it notices that its predecessor has failed and updates its *predecessor* value to $\perp$.

(d) Node 1 performs FaultStabilize, it notices that its successor has failed and finds the first node that is alive from its *successors* list, in this case node 3. Node 1 then updates its *successors* to $\langle 3 \rangle$, and calls Notify on node 3, passing itself as the argument. Node 3 will now update its *predecessor* value to 1 since its current value is $\perp$.

(e) Node 1 will then perform FaultStabilize again, this will update its *successors* list to contain the successors of its new successor (node 3).

(f) Node 4 executes FaultStabilize twice, first time it notices that its first successor has failed and updates its *successors* list to $\langle 1 \rangle$. The second execution will fetch the successors of its new successor, and its final successors list becomes $\langle 1, 3 \rangle$.



(a) System starts as an ideal ring.

(b) After node 2 fails.

(c) After node 3 runs CheckPredecessor.

(d) After node 1 runs FaultStabilize and node 3 runs Notify as a result.

(e) After node 1 again runs FaultStabilize.

(f) After node 4 runs FaultStabilize twice.

Figure 2.2: Example of a node failure in a ring with 4 nodes.

Chapter 3

Modeling the specification with TLA$^+$

The Chord TLA$^+$ specification builds on top of the work by [1]. With the major difference that the *successor* variable, a map from $Node \rightarrow Node$ has been replaced by the *successors* variable, which instead maps nodes to a list of succeeding nodes.

A *failures* variable has also been added which will be expanded on in section 3.5, up until that section, this variable will simply hold the value of the empty set.

The specification is made along with three configuration models, that each build on top of the previous. The first one is `ChordBase.cfg`, this model checks that the static structure of Chord is correct. The second one, `ChordChurn.cfg`, checks that when churn is introduced, the structure will always be safe and it will converge towards an ideal ring. And lastly, `ChordFailures.cfg` introduces the possibility for nodes to fail and checks that when that happens, the structure will eventually become safe again.

A TLA$^+$ specification consists of three parts, the definition of a valid initial state, the allowed transitions between states and lastly a set of safety properties that the model checker will verify are not violated. The `ChordBase.cfg` configuration only models the static structure of Churn, that is, it does not allow any transitions between states. This will later be introduced in section 2.2 and 3.5. We start by modeling the valid initial states of the successors variable.

3.1 Modeling successors in TLA$^+$

The successors list is modeled in the TLA$^+$ specification by the *SuccessorTypeInvariant*. This states that the *successors* variable must be in the set of all functions that map nodes to a sequence of successor nodes.

$$[\ Node \rightarrow \{\ (s_1, s_2, \ldots, s_n) \mid s_i \in Node,\ 0 \leq n \leq K\ \}\ ],$$

where K is the size of the successor list and $(s_1, s_2, \ldots, s_n)$ is strictly ordered.

SuccessorTypeInvariant

$$
\begin{array}{l}
\mathit{StrictlyOrdered}(s) \triangleq \forall\, i \in 2 \,..\, \mathit{Len}(s) : s[i-1] < s[i] \\
\\
\mathit{NodeSucessorList} \triangleq \text{UNION } \{[1 \,..\, m \to \mathit{Nodes}] : m \in 0 \,..\, K\} \\
\\
\mathit{SuccessorListOrdered}(s) \triangleq \\
\quad \text{LET } \mathit{alive\_s} \triangleq \mathit{SelectSeq}(s, \text{LAMBDA } x : \neg \mathit{HasFailed}(x)) \\
\quad \text{IN } \quad \mathit{Len}(\mathit{alive\_s}) > 0 \implies \exists\, m \in 1 \,..\, \mathit{Len}(\mathit{alive\_s}) : \\
\qquad \mathit{StrictlyOrdered}(\\
\qquad\quad \mathit{SubSeq}(\mathit{alive\_s},\ m+1,\ \mathit{Len}(\mathit{alive\_s})) \circ \mathit{SubSeq}(\mathit{alive\_s},\ 1,\ m)) \\
\\
\mathit{IsSuccessorList}(s) \triangleq \\
\quad \land\ s \in \mathit{Seq}(\mathit{Nodes}) \\
\quad \land\ \mathit{Len}(s) \leq K \\
\quad \land\ \mathit{SuccessorListOrdered}(s)
\end{array}
$$

Spec 3.1: Modeling of successors list in TLA$^+$

One of the most fundamental things to the model is the modeling of the successors list. The chord algorithm, unlike the predecessor variable, expects the successors to always be correct in order for the ring to be safe.

The *IsSuccessorList(s)* property, in spec. 3.1, checks whether s has the structure of a valid successor list. s must be a sequence of nodes with length at most K, this is the length of the successor list defined in the model configuration. The list must also be ordered as specified in *SuccessorListOrdered(s)*. This property checks whether s without failed nodes is ordered in a ring. It does so by checking to see if there exists a way to split the list, such that the two partitions individually are strictly ordered from low to high.

3.2 Verifying the static structure of Chord

The `ChordBase.cfg` configuration, uses the following specification to verify that the successors and predecessors at the beginning always represent a valid ring, and that simple routing in the ring works.

The *SpecAnyRing* specification (spec. B.1 in the appendix) consists of two properties, *SuccessorAnyRing* and *PredecessorSafety* which make sure the *successors* and *predecessor* variables respectiveley model an ideal ring. It also specifies that all of the variables must never change. This way only the static start configurations of the protocol is considered.

Definition 2 (Ideal ring) *A ring is ideal when:*

- *Following the first successor in the successors list of each node form*

exactly one ring.

- *The ring is ordered.*
- *Either a node is a part of the ring directly or an appendage to the ring.*

When the ring is ideal the simple routing invariants (spec. 3.2) should always be maintained.

Some of the invariants only apply for reliable intervals. A key i lies in a reliable interval from a node j, when j is expected to know the correct node responsible for key i. It might be the case that an appending node wrongly believes that there are no nodes between itself and its immediate successor, and thus the in-between keys are unreliable.

- *LocateDirectSuccessor*: Any node that has joined the ring should route all keys between itself and its first successor to the first successor.
- *LocateSelf*: When node i is a part of the ring, node j has joined and i lies in a reliable interval from j, then locating the successor of i from j should return i itself.
- *LocateSameSuccessor*: For any pair of joined nodes i, j and for all keys in a reliable interval from both nodes, each node should locate the same node given the same key.
- *LocateInRing*: For all joined nodes i and all reliable keys *key*, then from i locating the successor of *key* should return a node that forms a ring.

SimpleRoutingSafety

$$
\begin{aligned}
&\mathit{LocateDirectSuccessor} \triangleq \\
&\quad \forall\, i \in \mathit{Nodes} : \mathit{HasJoined}(i) \\
&\qquad \implies \forall\, \mathit{key} \in \mathit{Addresses} : \mathit{BetweenRightClosed}(\mathit{key},\, i,\, \mathit{FirstSuccessor}(i)) \\
&\qquad\quad \implies \mathit{LocateFirstSuccessor}(\mathit{key},\, i) = \mathit{FirstSuccessor}(i)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{LocateSelf} \triangleq \\
&\quad \forall\, i,\, j \in \mathit{Nodes} : \\
&\qquad \land\, \mathit{FormsRing}(i) \\
&\qquad \land\, \mathit{HasJoined}(j) \\
&\qquad \land\, \mathit{InReliableInterval}(i,\, j) \\
&\qquad\quad \implies \mathit{LocateFirstSuccessor}(i,\, j) = i
\end{aligned}
$$

$$
\begin{array}{l}
\mathit{LocateSameSuccessor} \triangleq \\
\quad \forall\, i,\, j \in \mathit{Nodes} : (\mathit{HasJoined}(i) \land \mathit{HasJoined}(j)) \\
\quad\quad \Longrightarrow \forall\, \mathit{key} \in \mathit{Addresses} : \\
\quad\quad\quad \land\, \mathit{InReliableInterval}(\mathit{key},\, i) \\
\quad\quad\quad \land\, \mathit{InReliableInterval}(\mathit{key},\, j) \\
\quad\quad \Longrightarrow\ \mathit{LocateFirstSucessor}(\mathit{key},\, i) = \mathit{LocateFirstSuccessor}(\mathit{key},\, j)
\end{array}
$$

$$
\begin{array}{l}
\mathit{LocateInRing} \triangleq \\
\quad \forall\, i \in \mathit{Nodes} : \mathit{HasJoined}(i) \\
\quad\quad \Longrightarrow \forall\, \mathit{key} \in \mathit{Addresses} : \mathit{InReliableInterval}(\mathit{key},\, i) \\
\quad\quad \Longrightarrow\ \mathit{FormsRing}(\mathit{LocateFirstSucessor}(\mathit{key},\, i))
\end{array}
$$

$$
\begin{array}{l}
\mathit{SimpleRoutingSafety} \triangleq \\
\quad \land\, \mathit{LocateDirectSuccessor} \\
\quad \land\, \mathit{LocateSelf} \\
\quad \land\, \mathit{LocateSameSuccessor} \\
\quad \land\, \mathit{LocateInRing}
\end{array}
$$

Spec 3.2: Simple routing invariants

3.3 Modeling churn in TLA$^+$

The `ChordChurn.cfg` configuration, introduces specifications for i.JoinRing(j) and i.Stabilize(). It introduces the ChurnNext property which for each temporal step either joins a new node or stabilizes an existing one.

ChurnNext

$$
\begin{array}{l}
\mathit{ChurnNext} \triangleq \\
\quad \land\, \mathit{NoFailures} \\
\quad \land \exists\, i \in \mathit{Nodes} : \\
\quad\quad \lor\, \mathit{Join}(i) \\
\quad\quad \lor\, \mathit{Stabilize}(i)
\end{array}
$$

Join A node i can join the ring when it has not already joined and it has not failed. It also requires that there exists a node j that i can use to join the network from. Since j will use *LocateSuccessors* to guide i to its correct place in the ring, *Join* requires that j can find the successor of i which is not always the case later when failures are introduced. Node i then joins the network by updating its *successors* variable to the result of the *LocateSuccessors* call.

Below, in spec. 3.3, the third and forth conjuncture of the join procedure initially might seem to be duplicates of one another. But this is needed since the model checker will throw an exception if it tries to execute CHOOSE on an

empty set. The line before discards the join call if there exists no node that i can use to join the ring.

Join

$$
\begin{array}{l}
\mathit{CanJoin}(i,\, j) \triangleq \\
\quad \land \quad \mathit{HasJoined}(j) \\
\quad \land \quad \mathit{LocateSuccessors}(i,\, j) \neq \mathit{LocateSuccessorError} \\
\\
\mathit{Join}(i) \triangleq \\
\quad \land \neg \mathit{HasJoined}(i) \\
\quad \land \neg \mathit{HasFailed}(i) \\
\quad \land \exists\, j \in \mathit{Nodes} : \mathit{CanJoin}(i,\, j) \\
\quad \land \text{LET } j \triangleq \text{CHOOSE } j \in \mathit{Nodes} : \mathit{CanJoin}(i,\, j) \text{ IN} \\
\qquad \land \quad \mathit{successors}' = [\mathit{successors} \text{ EXCEPT } ![i] = \mathit{LocateSuccessors}(i,\, j)] \\
\quad \land \text{UNCHANGED } \langle \mathit{failures},\, \mathit{predecessor} \rangle
\end{array}
$$

Spec 3.3: TLA$^+$ implementation of i.JoinRing(j) from alg. 2.

Stabilize A node can be stabilized only if it has joined the ring. To model the outer `for`-loop of the i.FaultStabilize() function in alg. 4, the unreachable successor nodes are removed from the *successors*$[i]$ list and the result is binded to the *succs* variable. Next the s, p and ss variables from the pseudocode are calculated. If p, the successors predecessor, lies between i and s then clear the existing successor list and use $\langle p \rangle$ as the new successor list. Otherwise prepend $\langle s \rangle$ to ss and clean the list to ensure that the new successors list does not contain any duplicates.

Next the *successors*$'$ list is assigned to account for the changes and *Notify* is called on the new first successor of i. This makes sure to update the predecessor of i's successor if i is closer to it than its current predecessor is.

Stabilize

```
Notify(i, j) ≜
    IF ¬HasPredecessor(i) ∨ ExBetween(j, predecessor[i], i)
     THEN predecessor' = [predecessor EXCEPT ![i] = j]
     ELSE UNCHANGED predecessor

Stabilize(i) ≜
    ∧ HasJoined(i)
    ∧ LET succs ≜ RemoveUnreachableSucessors(successors[i])
           s ≜ Head(succs)
           p ≜ predecessor[s]
           ss ≜ successors[s]
           new_succs ≜
               IF ExBetween(p, i, s) ∧ HasJoined(p) THEN ⟨p⟩
                ELSE CleanSuccessors(⟨s⟩ ∘ ss)
      IN
       ∧ successors' = [successors EXCEPT ![i] = new_succs]
       ∧ Notify(Head(new_succs), i)
    ∧ UNCHANGED failures
```

Spec 3.4: TLA$^+$ implementation of i.STABILIZE() and i.NOTIFY(j) from alg. 3.

3.4 Properties of churn

The model checker runs the churn configuration it checks for the following three liveness properties.

$$NodesNeverLeave \triangleq \Box(\forall\, i \in Nodes : FormsRing(i) \implies \Box(FormsRing(i))$$

The *NodesNeverLeave* property checks that once a node joins the ring it never leaves. Formally it states that it is always the case that for all nodes i, if i forms a ring then it will always form a ring.

$$StaysFullIdealRing \triangleq \Diamond\Box(SuccessorFullRing \wedge IdealRing)$$

The *StaysFullIdealRing* property checks that eventually the ring will always become full and ideal. This is because churn only allows for nodes to join and stabilize, and since the join and stabilize actions impose weak fairness for each node, the actions that are able to run will always do so until the ring is full and ideal.

3.4.1 Fairness

TLA$^+$ has the concept of *stuttering,* this means that by default any state can stutter, meaning that it transitions to the same state as it came from and thus does not make any progress. This concept makes it manageable to compose TLA$^+$ specifications, for example in a composition of two specifications, one can make progress while the other one stutters.

This, however, causes problems when introducing temporal properties such as *StaysFullIdealRing*, which expects the ring to eventually become ideal, but if the specification is allowed to stutter, then there is nothing stopping the specification from performing stuttering steps indefinetley and never make progress, in this case the ring will never become ideal and the property will be violated.

To solve this, TLA$^+$ has the concept of weak and strong fairness, where weak fairness means that if an actions can always be performed, then it will eventually be performed. Strong fairness is stricter and guarantees that if an action alternates between being able to be performed and not being able to, then it will eventually be performed. In our case, and in most cases, weak fairness is enough.

$$
\begin{array}{l}
\mathit{FairChurn} \triangleq \\
\quad \land \forall\, i \in \mathit{Nodes} : \\
\qquad \land \mathrm{WF}_{\mathit{vars}}(\mathit{Join}(i)) \\
\qquad \land \mathrm{WF}_{\mathit{vars}}(\mathit{Stabilize}(i))
\end{array}
$$

As for churn, the *FairChurn* property has been added, which states that the join and stabilize actions for each node impose weak fairness, this means that all configurations of join and stabilize will eventually be run and will not stutter indefinetley.

3.5 Modeling failures in TLA$^+$

When modeling failures we introduce the $\mathit{FailNode}(i)$ and $\mathit{CheckPredecessor}(i)$ functions to the *next* part of the specification.

RecoverableAfterFail We also introduce the *RecoverableAfterFail* function, which computes whether the network will be in a recoverable state after node i fails. This function would not exist in a concrete implementation of the specification, but it restricts the model checker to only calculate states that we expect to be recoverable.

A node that has not joined the network yet is always safe to fail, since no other node depend on it yet. If a node i has joined, then it is only safe for it to fail, when all nodes whose first successor is i can reach i's successor, and when i is not the only node in the ring.

RecoverableAfterFail

```
RecoverableAfterFail(i) ≜
    HasJoined(i) ⟹
        ∧ ∀ n ∈ Nodes : FirstReachableSuccessor(n) = i ⟹
              FirstReachableSuccessor(i) ∈ SeqToSet(successors[n])
        ∧ FirstReachableSuccessor(i) ≠ i
```

Spec 3.5: Function that evaluates whether the ring can recover if node i fails.

CheckPredecessor The *CheckPredecessor* property, like *Stabilize* from spec. 3.4, can be evaluated at each *next* step in the specification. And would in practice be run periodically on each node in the network. The property is only considered if node i has joined. If the predecessor of i has failed, then its predecessor value is set to $\perp$, which is represented as 0 in the TLA$^+$ specification.

CheckPredecessor

```
CheckPredecessor(i) ≜
    ∧ HasJoined(i)
    ∧ IF predecessor[i] ∈ failures
        THEN predecessor' = [predecessor EXCEPT ![i] = 0]
        ELSE predecessor' = predecessor
    ∧ UNCHANGED ⟨successors, failures⟩
```

Spec 3.6: TLA$^+$ implementation of i.CheckPredecessor from alg. 4.

FailNode At each *next* iteration of the specification, any node i that has not yet failed can fail, if the ring is recoverable after it does so. When this happens, i will be added to the set of failed nodes, and its successor list and predecessor values are reset to avoid redundant states and keep the possible states as small as possible.

FailNode

```
FailNode(i) ≜
    ∧ ¬HasFailed(i)
    ∧ RecoverableAfterFail(i)
    ∧ failures' = failures ∪ {i}
    ∧ successors' = [successors EXCEPT ![i] = ⟨⟩]
    ∧ predecessor' = [predecessor EXCEPT ![i] = 0]
```

Spec 3.7: Simulate a node failing in TLA$^+$.

3.5.1 Failure properties

It is not as easy to describe properties for failures as it is for churn, since churn converges toward an ideal ring and the ring is expected to always be safe. When failures are introduced, this is not the case anymore. Instead, when a node fails the ring can become unsafe and requests such as i.LocateSuccessor(key) might fail until the ring self-stabilizes.

The ring should be able to always recover and become safe after it has had the chance to stabilize itself. In TLA$^+$ this can be described as

$$RingAlwaysRecover \triangleq \Box\Diamond(SuccessorAnyRing),$$

which states that it is the case that the ring will *always eventually* become safe. Where a safe ring in the specification is defined as a ring where the *successors* variable represent any valid ring.

The *NodeStaysFailed* property ensures that once a node fails it will always stay failed.

$$NodeStaysFailed \triangleq$$
$$\Box(\forall\, i \in Nodes : HasFailed(i) \implies \Box(HasFailed(i)))$$

Chapter 4

Evaluation

4.1 Model checker results

The specification has been checked with the TLC model checker using the three configurations `ChordBase.cfg`, `ChordChurn.cfg`, `ChordFailures.cfg`, and the results can be seen in the respective subtables of table 4.1. The left most column describes the model parameters N, the number of nodes, and K, the length of the successor list. For each configuration the number of states and of which are unique in parenthesis, the diameter of the graph ie. the longest path found and lastly how long it took the model checker to check the specification with the given configuration.

The model checker did not find any property violations for any of the configurations of the final specification, which means that we can say with great certainty that the specification is correct.

It makes sense for the diameter for the base configuration to always be one, since that configuration does not contain any transitions and thus it only calculates and verifies initial states. The churn configuration produces a larger diameter than the failures configuration, since when a node fails, that node can not make any more progress that could contribute to the diameter increasing.

Constants	**Base** States		Diameter	Time
$N = 1$ $K = 1$	2	(1)	1	02s
$N = 2$ $K = 2$	10	(5)	1	02s
$N = 3$ $K = 3$	56	(28)	1	20s
$N = 4$ $K = 2$	422	(211)	1	3h 15m 24s
$N = 4$ $K = 3$	422	(211)	1	1d 1h 42m 29s

(a) Base configuration

Constants	**Churn** States		Diameter	Time
$N = 1$ $K = 1$	3	(2)	2	02s
$N = 2$ $K = 2$	30	(14)	6	03s
$N = 3$ $K = 3$	453	(140)	10	24s
$N = 4$ $K = 2$	7856	(1749)	15	3h 24m 06s
$N = 4$ $K = 3$	11208	(2563)	16	1d 2h 15m 29s

(b) Churn configuration

Constants	**Failures** States		Diameter	Time
$N = 1$ $K = 1$	5	(2)	2	02s
$N = 2$ $K = 2$	119	(33)	6	03s
$N = 3$ $K = 3$	3874	(708)	8	24m 48s
$N = 4$ $K = 2$	122391	(15491)	10	2d 18h 21m 29s
$N = 4$ $K = 3$	-	-	-	-

(c) Failures configuration.

Table 4.1: TLA$^+$ model checker results

4.2 Examining the state diagram

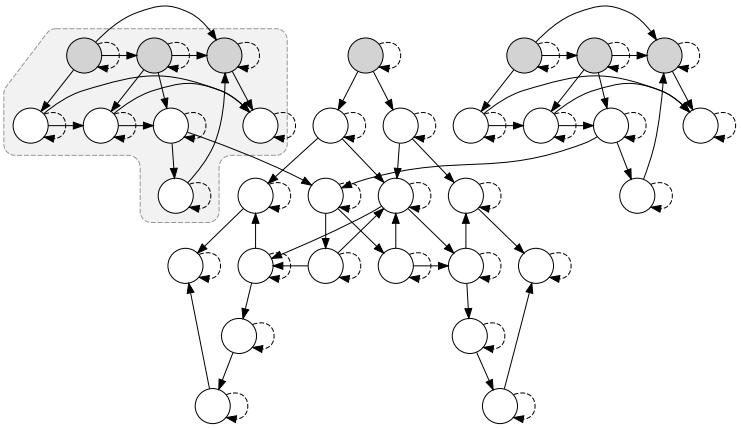
Running the model checker on very small inputs produces such few states that they can be drawn in a diagram and examined. Such a state diagram for the failures configuration with two nodes, producing 33 unique states, is shown on fig. 4.1. Here each node represents a unique state and an arc between to nodes represents either a *Join*, *FaultStabilize*, *CheckPredecessor* or *FailNode* state transition.

The first diagram on fig. 4.1a shows an overview of the produced states, but without the variables of each state. The next diagram on fig. 4.1b shows an expanded view of a subset of the states with the values of the variables at each state.

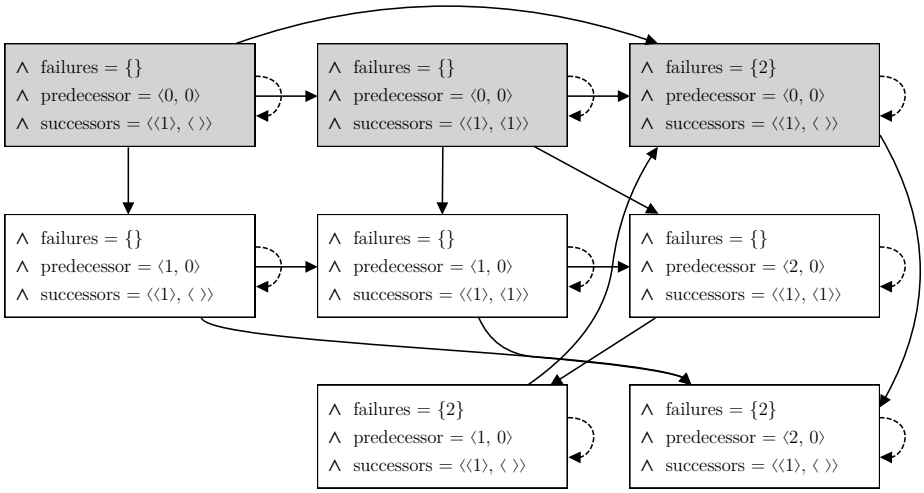
4.3 Things not modeled

In practice it would be convenient for a node to gracefully leave the system, by notifying its neighbours about its departure and simultaneously transfer its keys to its successor, rather than directly failing and potentially losing data. This was not modeled since it did not add much new and would increase the complexity and thus the evaluation time of the model substantially.

Another aspect that was not considered in this project is data replication where keys would be split out over multiple nodes in the system, such that when a node fails the other nodes in the network have the necessary information needed to reconstruct the keys of the failed node.



(a) Overview of the state diagram. Fig. 4.1b shows an expanded view of the top left highlighted subgraph.



(b) Expanded view of the top left highlighted subgraph on fig. 4.1a. Each node shows a state with the value of its variables.

Figure 4.1: TLA$^+$ state diagram for failures configuration with $N = 2$ and $K = 2$

Chapter 5

Conclusion

In this project we have studied how the Chord algorithm can be modified to account for node failures by introducing a successor list at each node. This has been implemented as a formal specification using the specification language TLA$^+$. The specification has been verified to be correct by the TLC model checker, for networks with up to 4 nodes, enough nodes to say that the specification can be assumed to be correct with a high confidence.

Throughout the project, the complexity of the specification had to be hold at a minimum, in order to run the model checker within a reasonable time. This limitation has meant that some parts of the protocol has been simplified and assumptions has been made. Extensions to the protocol like the finger table has not been considered as verifying that alone, without modeling failures, already takes a lot of time.

Also the communication model is assumed to be synchronous, which means that messages sent arrive instantly at the receiving node. The failure model of the specification is also limited to only node failures. This means that other kinds of failures, such as the links between nodes failing, or nodes omitting sending or receiving messages are not considered.

Bibliography

[1] Adam and Mads. "Formal verification of the peer-to-peer protocol Chord with TLA$^+$". In: (May 2021), p. 28.

[2] Ajay D Kshemkalyani and Mukesh Singhal. *Distributed computing*. Cambridge, England: Cambridge University Press, Mar. 2011.

[3] Leslie Lamport. *The TLA$^+$ Home Page*. 1999. URL: `http://lamport.azurewebsites.net/tla/tla.html` (visited on 03/18/2022).

[4] Ion Stoica et al. "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications". In: *SIGCOMM Comput. Commun. Rev.* 31.4 (2001). ISSN: 0146-4833. DOI: `10.1145/964723.383071`. URL: `https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf`.

Appendix A

Pseudo code for Chord with multiple successors

Algorithm 5 Locate the successor list of the first k successors to *key*

Variables: *successors*[1..k], *predecessor*

Require: $key \neq i$
function i.LocateSuccessors(key)
    $s \leftarrow successors[1]$
    if $key \in (i, s]$ **then**
        return *successors*
    else
        return s.LocateSuccessor(key)
    end if
end function

Algorithm 6 Creating a new ring and joining an existing one.

Variables: *successors*[1..k], *predecessor*

function i.CreateNewRing
    $predecessor \leftarrow \perp$
    $successors \leftarrow [i]$
end function
function i.JoinRing(j) ▷ where j is any node on the ring
    $predecessor \leftarrow \perp$
    $successors \leftarrow j$.LocateSuccessors(i)
end function

Appendix B

TLA$^+$ specifications

SpecAnyRing

$$
\begin{array}{l}
\mathit{SuccessorAnyRing} \triangleq \\
\quad \land \mathit{TypeInvariants} \\
\quad \land \mathit{ExactlyOneRing} \\
\quad \land \mathit{OrderedRing} \\
\quad \land \mathit{OnlyAppendages} \\
\quad \land \forall\, i \in \mathit{Nodes} : \mathit{HasFailed}(i) \implies \\
\qquad \land \mathit{successors}[i] = \langle\rangle \\
\qquad \land \mathit{predecessor}[i] = 0 \\
\\
\mathit{PredecessorSafety} \triangleq \\
\quad \land \mathit{PredecessorTypeInvariant} \\
\quad \land \mathit{ReachableFromPredecessor} \\
\quad \land \mathit{PredecessorInRing} \\
\\
\mathit{SpecAnyRing} \triangleq \\
\quad \land \mathit{SuccessorAnyRing} \\
\quad \land \mathit{PredecessorSafety} \\
\quad \land \Box[\text{UNCHANGED } \mathit{vars}]_{\mathit{vars}}
\end{array}
$$

Spec B.1: Specification for a static Churn configuration.