Correct by Design with TLA+
Early Preview

Richard Tang

February 14, 2025

Acknowledgement

A big thanks to Anthony Giardina for reviewing the content of this book.

Contents

Part I

Introduction

Chapter 1

Motivation

1.1 Catching Problems Early

Years ago, I worked on a proprietary low-power processor in an embedded system. The processor ran a microcode featuring a custom instruction set. To enter a low-power state, a set (possibly hundreds) of instructions were executed. These instructions progressively put the system in a lower power state. For example: Turn off IP A, then turn off IP B, then turn off the power island to the IPs. To save cost and power, the low-power processor had very limited debuggability support.

An experienced reader may start to notice some red flags.

If the microcode attempts to access the memory interface when the power island has been shut off, the processor will hang. Since the debug power island has been shut off, the physical hardware debug port is also unavailable, leaving the developer with *no way* of live debugging problems. At this point, the developer needs to search through numerous instructions to catch system constraint violations (invariants) *manually*.

As one can imagine, maintaining the microcode was very expensive. Fortunately, the proprietary low-power processor only had a handful of instructions, so I created a simulator for this proprietary processor to verify the microcode before deploying it on-target. The simulator models the processor states as a state graph, with executed instruction, transitions the state machine to the next state. At every state, all the invariants are verified. Example invariants include:

- Accessing memory interface after power off leads to a hang
- Accessing certain register in certain chip revision leads to a hang
- Verify IPs are shut off in the allowed order

The verification algorithm was implemented using a *depth-first-search*, providing 100% microcode coverage before deployment on target.

To generalize, we can model an arbitrary system as a set of states and a set of invariants that must be upheld at all times. The complexity of such an arbitrary system generally grows quadratically as the number of states grows linearly (eg. In an N-state system, adding state N+1 may introduce N transitions into the new state). There are many engineering problems with a large number of states, such as lockless or wait-free data structures, distributed algorithms, OS schedulers, consensus protocols, and more. As the number of states grows, the problem becomes more challenging for designers to reason about.

So, how do we produce a system that is *correct by design?*

1.2 The Generalized Problem

Fast forward to now: I stumbled across TLA+, a formalized solution of what I was looking for.

We are at a point in the technology curve where vertical scaling is no longer practical, with CPU speed plateaued in the past decade or so. The industry is exploring horizontal scaling solutions, such as hardware vendors focusing on adding more CPU cores, or software vendors buying more low-end hardware instead of a less high-end hardware. This shifts the technology complexity from vertical to horizontal, demanding solutions to maximize concurrent resource utilization. There is one slight problem though:

Humans are not good at concurrent reasoning.

Our cognitive system is optimized for sequential reasoning. Enumerating all scenarios in one's mind to ensure an arbitrary design accommodates all the corner cases is challenging.

Consider a distributed system. The system is a cluster of independently operating entities, which need to somehow collectively offer the correct system behavior, while any one of the machines may receive instructions out of order, crash, recover, etc.

Consider a single producer multiple consumer lockless queue. The consumers may reserve an index in the queue in a certain order but may release it in a different order. What if one reader is slow, and another reader is superfast and possibly lapses the slow reader?

Consider an OS scheduler with locks. Assume all the processes have the same priority. Can a process starve the other processes by repeatedly acquiring and releasing the lock? How do we ensure scheduling is fair?

The *anti-pattern* is to keep band-aiding the design until the user stops filing bug reports. This is never ideal. Per Murphy's law, anything that can go wrong *will go wrong*, and a hard-to-reproduce bug will come in at the most inconvenient time. How do we make sure the solution is *correct by design*? To solve this problem, we must rely on tools to do the reasoning *for us.*

1.3 What is TLA+?

TLA+ is a *system specification language* to describe a system without implementation details. TLA+ allows a designer to describe a system as a set of states with transitions from one state to the next. Designers can describe invariants that must hold in every state and liveness properties a sequence of states must satisfy. One of TLA+'s keys is once the system is modeled as a finite set of states, the states can be *exhaustively* explored (via breath-first-search) to ensure properties are upheld throughout the entire state space (either per state or a sequence of states).

1.4 About This Book

To my surprise, there is not as much material on TLA+ as I assumed for such a critical tool in a designer's toolbox. This book was initially a set of notes I took while learning TLA+. I decided to formalize these notes into this short book, which I hope the readers will find helpful in their TLA+ journey.

The book intends to teach the reader how to write TLA+ spec for their design to provide confidence in *design correctness*. This book is targeted at software designers, hardware designers, system architects, and in general anyone interested in designing correct systems.

To get the most out of the book, the reader should have general computer science knowledge. The reader doesn't need to be an expert in a particular language to understand this book; TLA+ is effectively its language. This book is example-driven and will go through designs such as lockless queues, simple task schedulers, consensus algorithms, etc. Readers will likely enjoy a deeper insight if there is familiarity with these topics.

1.5 How to Use This Book

This book was designed to be used as a reference, providing examples and references using TLA+.

Examples are split into two categories: Examples written using TLA+ and examples written using PlusCal (the C-like syntax that transpiles down to TLA+). I believe they are useful for different use cases. The differences will be highlighted in their respective sections. All examples will follow a similar layout, covering the problem statement, design, spec, and safety properties.

All examples in this book will be presented using TLA+ *mathematical notation*. Converting between Mathematical and ASCII notation is trivial due to the one-to-one mapping. Readers are encouraged to consult Table 8 in [1] as needed.

The last part of the book provides language references and some focused topics. Readers can use the last part of the book as a general reference.

Chapter 2

TLA+ Primer

2.1 Purpose

The key insight to TLA+ is modeling a system as a state machine. A simple digital clock can be represented by two variables: hour and minute. The number of possible states in a digital clock is $24 * 60 = 1440$. For example, a clock in state 10:00 will transition to state 10:01. Assume an arbitrary system described by N variables, each variable having K possible values. Such an arbitrary system can have up to N^K states.

For every specification, the designer can specify *safety* properties (or invariants) that must be true in *every* state. For example, in any state of the digital clock, hour *must* be between 0 to 23, or formally described as $hour \in 0..23$. Similarly, the minute must have a value between 0 to 59, or $minute \in 0..59$. Examples of invariants of a system include: Only one thread has exclusive access to a critical region, all variables in the system are within allowable value, and the resource allocation manager never allocates more than available resources.

The designer can also specify *liveness* properties. These are properties to be satisfied by a *sequence of states*. One liveness property of the digital clock could be when the clock is 10:00, it will eventually become 11:00 (10:00 *leads* to 11:00). Example liveness property include:

a distributed system eventually converges, the scheduler eventually schedules every task in the task queue, and the resource allocation manager fairly allocates resources.

A TLA+ Spec can be checked by TLC, the model checker. TLC uses *breadth-first search* algorithm to explore *all* states in the state machine and ensure safety and liveness properties are upheld.

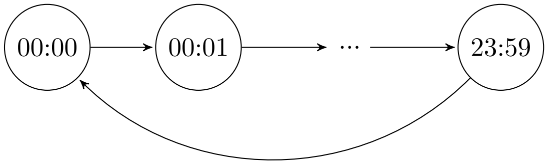
A TLA+ Spec describes the system using *temporal logic*. The syntax may appear unfamiliar, but like any other programming language an un-initiated reader should become familiarized quickly.

2.2 Design

In this example, we will specify a *digital clock*. The digital clock has a few simple requirements:

- Two variables to represent state: hour and minute
- The clock increments one minute at a time
- Hour is between 0 to 23, inclusive
- Minute is between 0 to 59, inclusive
- Clock wraps around at midnight (ie. 23:59 transitions to 00:00)

The state graph for the clock looks like this:



Note that in this particular example each state only has one entry and one exit. TLA+ doesn't preclude states with multiple entries and multiple exists.

2.3 Spec

The *Init* state of such a system can be described as:

$$
\begin{aligned}
&\mathit{Init} \triangleq \\
&\quad \land\ \mathit{hour} = 0 \\
&\quad \land\ \mathit{minute} = 0
\end{aligned}
$$

$\triangleq$ is the *defines equal* symbol and $\land$ is the *logical and* symbol. The above TLA+ syntax can be read as *Init* state is defined as both hour and minute are 0.

The spec also always includes a *Next* definition, an *action formula* describing how the system transitions from one state to another. Action formula contains *primed* variables, representing values of variables in their next state. The *Next* action for the digital clock can be defined as:

$$
\begin{aligned}
&\mathit{NextHour} \triangleq \\
&\quad \land\ \mathit{minute} = 59 \\
&\quad \land\ \mathit{hour}' = (\mathit{hour} + 1)\%24 \\
&\quad \land\ \mathit{minute}' = 0 \\
&\mathit{NextMinute} \triangleq \\
&\quad \land\ \mathit{minute} \neq 59 \\
&\quad \land\ \mathit{hour}' = \mathit{hour} \\
&\quad \land\ \mathit{minute}' = \mathit{minute} + 1 \\
&\mathit{Next} \triangleq \\
&\quad \lor\ \mathit{NextMinute} \\
&\quad \lor\ \mathit{NextHour}
\end{aligned}
$$

Here's a breakdown of what the spec does:

- *Next* can take either *NextHour* or *NextMinute*. $\lor$ is the *logical or* operator.
- *Next* takes *NextMinute* when *minute* is not 59. When *NextMinute* takes: *hour* in the next state equals to *hour* in the current state, but *minute* in the next state is *minute* in current state plus one.

- *Next* takes *NextHour* when *minute* is 59. When *NextMinute* takes: *hour* in the next state equals to *hour* in the current state plus one and modulus by number of hours in a day and *minute* in the next state equals to zero.

Note that the formulas are *state descriptions*, not *assignment*. *minute = 59* describes the state transition takes when minute *equals* 59. Since this is an equality description, *minute = 59* and *59 = minute* are equivalent in TLA+.

Finally, the Spec itself is formally defined as:

$$
\begin{aligned}
&vars \triangleq \langle hour,\ minute \rangle \\
&Spec \triangleq \\
&\qquad \land\ Init \\
&\qquad \land\ \Box[Next]_{vars}
\end{aligned}
$$

Note this forms the basis for **all** TLA+ spec. Every example in this book will include a *Spec* definition similar to this.

$\Box[Next]_{vars}$ deserves some special attention:

- *vars* is defined earlier to be *all* variables in the spec, in this case, hour and minute. A combination of these variables at different values constitutes the states of the system (eg. 23:59 and 00:00 are different states in the system).

- $\Box[Next]_{vars}$ is a box-action formula, where *Next* is an action and *vars* is a state function.

- $\Box$ (necessity operator) asserts the formula is always true for every step in the behavior.

- Steps in the behaviour are defined as $[Next]_{vars}$, where *Next* describes the action and *vars* capturing all variables representing the state.

This can be roughly translated to: the system is valid for every step *Next* can take.

2.4 Safety

A safety property describes an invariant that must hold in *every* state of the system. A common invariant is *type safety* checks. In a digital clock, an hour can only be in value between 0 to 23, and a minute can only be the value of 0 to 59:

$$
\begin{aligned}
&\mathit{Type\_OK} \triangleq \\
&\quad \land\ \mathit{hour} \in 0 \,..\, 23 \\
&\quad \land\ \mathit{minute} \in 0 \,..\, 59
\end{aligned}
$$

When an hour or minute falls outside of the specified range, the model checker reports violation.

2.5 Liveness and Fairness

Liveness property verifies certain behaviors across a sequence of states. One liveness property is to confirm the clock wraps around at midnight, a property that can only be verified after checking at least two states:

$$
\begin{aligned}
&\mathit{Liveness} \triangleq \\
&\quad \land\ \mathit{hour} = 23 \land \mathit{minute} = 59 \leadsto \mathit{hour} = 0 \land \mathit{minute} = 0
\end{aligned}
$$

$\leadsto$ is the *leads to* operator, suggesting something is eventually true. TLA+ provides a set of operators to describe the liveness property.

To verify liveness, we need to modify the spec slightly to enable *fairness* to prevent *stuttering.* In plain terms, fairness ensures a state always transitions to *some other state.* Without fairness, the spec is allowed to *stutter*, or *not transition* to any state. This fails the liveness property check as the model checker cannot verify the behavior across a sequence of states. To get a more comprehensive description of fairness, refer to the last part of the book.

$$
\begin{aligned}
&\mathit{Spec} \triangleq \\
&\quad \land\ \mathit{Init}
\end{aligned}
$$

$$\land \Box[Next]_{vars}$$
$$\land \mathrm{WF}_{vars}(Next)$$

$WF_{vars}(Next)$ is the fairness qualifier.

2.6 Model Checker

A TLA+ spec can be verified using a model checker. The model checker runs the spec and verifies all specified safety and liveness properties are fulfilled. The model checker is a library written in Java and can be invoked from the command line. For instructions on installing the model checker and related tools, please see [3].

After installing the model checker, we need two things to verify the spec:

- clock.tla: spec
- clock.cfg: config file

For reference, clock.tla is listed below:

MODULE *clock*

EXTENDS *Naturals*
VARIABLES *hour*, *minute*
$vars \triangleq \langle hour, minute \rangle$
$Type\_OK \triangleq$
$\quad \land hour \in 0 \,..\, 23$
$\quad \land minute \in 0 \,..\, 59$
$Liveness \triangleq$
$\quad \land hour = 23 \land minute = 59 \leadsto hour = 0 \land minute = 0$
$Init \triangleq$
$\quad \land hour = 0$
$\quad \land minute = 0$
$NextMinute \triangleq$
$\quad \land minute = 59$
$\quad \land hour' = (hour + 1)\%24$
$\quad \land minute' = 0$

$NextHour \triangleq$
$\quad \land\ minute \neq 59$
$\quad \land\ hour' = hour$
$\quad \land\ minute' = minute + 1$
$Next \triangleq$
$\quad \lor\ NextMinute$
$\quad \lor\ NextHour$
$Spec \triangleq$
$\quad \land\ Init$
$\quad \land\ \Box[Next]_{vars}$
$\quad \land\ \mathrm{WF}_{vars}(Next)$

The corresponding clock.cfg is listed below:

```
SPECIFICATION Spec
INVARIANTS Type_OK
PROPERTIES Liveness
```

After putting both clock.cfg and clock.tla in the same directory, one can now run the model checker. In this book I'll assume a command line interface for the model checker:

```
java -cp /usr/local/lib/tla2tools.jar tlc2.TLC clock
...
Model checking completed. No error has been found.
...
The depth of the complete state graph search is 1440.
```

The 1440 states in the state graph represent the total number of minutes in a day.

Part II

Examples with TLA+

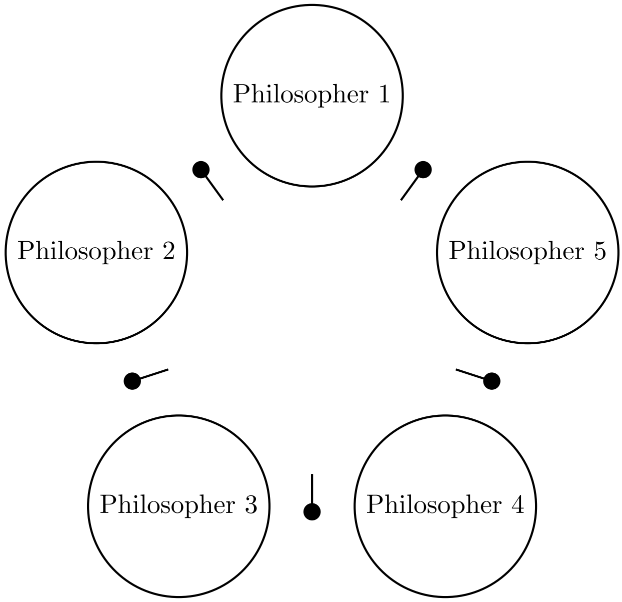
TLA+ notation is rooted in temporal logic and doesn't share the usual programming language *look and feel*. Despite the possibly foreign look, the core language semantics for TLA+ is reasonably constrained. This allows anyone with some programming experience to pick up relatively quickly.

This section provides a few example TLA+ spec with increasing complexity, easing the readers into this wonderful tool.

Chapter 3

Dining Philosophers

Dining philosophers is a famous problem used to illustrate concurrent algorithm design [11]. The problem states there are N philosophers sitting in a circle, with a fork placed between each philosopher. This is illustrated below:



Each philosopher is thinking or eating, but the philosopher needs to take *both* forks to eat. The problem is to design a solution that ensures one or more philosophers can eat when they want to.

A possible failing scenario is when *all* philosophers take the fork to their left. Now every philosopher is stuck waiting for the fork to their right, and every philosopher starve.

3.1 Design

Every philosopher behaves similarly:

- Take one fork
- Take another fork
- Eat
- Put away one fork
- Put away another fork

3.2 Spec

The core part of *Spec* looks like this:

$$
\begin{aligned}
&Next \triangleq \\
&\quad \lor \exists k \in 0 \,..\, P-1 : \\
&\quad\;\; TakeFirst(k) \\
&\quad \lor \exists k \in 0 \,..\, P-1 : \\
&\quad\;\; TakeSecond(k) \\
&\quad \lor \exists k \in 0 \,..\, P-1 : \\
&\quad\;\; Eat(k) \\
&\quad \lor \exists k \in 0 \,..\, P-1 : \\
&\quad\;\; PutFirst(k) \\
&\quad \lor \exists k \in 0 \,..\, P-1 : \\
&\quad\;\; PutSecond(k)
\end{aligned}
$$

This reflects the behavior described earlier. Note that there's a sequential dependency to these actions. The philosopher can only take the second fork after taking the first fork, eat after having both forks and put away the forks after eating.

$$
\begin{aligned}
&First(k) \triangleq k \\
&Second(k) \triangleq (k+1)\%P
\end{aligned}
$$

$$
\begin{aligned}
&TakeFirst(k) \triangleq \\
&\quad \land\, eaten[k] = 0 \\
&\quad \land\, forks[First(k)] = UNUSED \\
&\quad \land\, \text{UNCHANGED } eaten
\end{aligned}
$$

$$
\begin{aligned}
&TakeSecond(k) \triangleq \\
&\quad \land\, eaten[k] = 0 \\
&\quad \land\, forks[First(k)] = k \\
&\quad \land\, forks[Second(k)] = UNUSED \\
&\quad \land\, forks' = [forks \text{ EXCEPT } ![Second(k)] = k] \\
&\quad \land\, \text{UNCHANGED } eaten
\end{aligned}
$$

The philosopher greedily takes the first fork when possible. After the philosopher has the first fork, she greedily takes the second fork when possible.

$$
\begin{array}{l}
Eat(k) \triangleq \\
\text{LET} \\
\quad left \triangleq k \\
\quad right \triangleq (k+1)\%P \\
\text{IN} \\
\quad \land\, forks[left] = k \\
\quad \land\, forks[right] = k \\
\quad \land\, eaten' = [eaten \text{ EXCEPT } ![k] = 1] \\
\quad \land\, \text{UNCHANGED } forks
\end{array}
$$

Once the philosopher has both forks, she can eat.

$$
\begin{array}{l}
PutFirst(k) \triangleq \\
\quad \land\, eaten[k] = 1 \\
\quad \land\, forks[First(k)] = k \\
\quad \land\, forks' = [forks \text{ EXCEPT } ![First(k)] = UNUSED] \\
\quad \land\, \text{UNCHANGED } eaten
\end{array}
$$

$$
\begin{array}{l}
PutSecond(k) \triangleq \\
\quad \land\, eaten[k] = 1 \\
\quad \land\, forks[First(k)] \neq k \\
\quad \land\, forks[Second(k)] = k \\
\quad \land\, forks' = [forks \text{ EXCEPT } ![Second(k)] = UNUSED] \\
\quad \land\, eaten' = [eaten \text{ EXCEPT } ![k] = 0]
\end{array}
$$

After eating, the philosopher puts away the forks.

3.3 Safety

Omitted for this chapter.

3.4 Liveness

One liveness property is to ensure that at least one philosopher can eat when she wants to under all circumstances:

$$
\begin{array}{l}
\mathit{Liveness} \triangleq \\
\quad \exists\, k \in 0 \,..\, P-1 : \\
\quad\quad \land\ \mathit{eaten}[k] = 0 \leadsto \mathit{eaten}[k] = 1 \\
\quad\quad \land\ \mathit{eaten}[k] = 1 \leadsto \mathit{eaten}[k] = 0
\end{array}
$$

However, *Spec* defined as is doesn't implement any deadlock mitigation. Running it against the model the checker results in the following violations:

```
State 2: <TakeFirst line 19, col 5 to line 23,
    col 22 of module dining>
/\ eaten = (0 :> 0 @@ 1 :> 0 @@ 2 :> 0)
/\ forks = (0 :> 0 @@ 1 :> 100 @@ 2 :> 100)

State 3: <TakeFirst line 19, col 5 to line 23,
    col 22 of module dining>
/\ eaten = (0 :> 0 @@ 1 :> 0 @@ 2 :> 0)
/\ forks = (0 :> 0 @@ 1 :> 1 @@ 2 :> 100)

State 4: <TakeFirst line 19, col 5 to line 23,
    col 22 of module dining>
/\ eaten = (0 :> 0 @@ 1 :> 0 @@ 2 :> 0)
/\ forks = (0 :> 0 @@ 1 :> 1 @@ 2 :> 2)
```

When all philosopher takes their left fork, no one can eat.

A simple fix to the problem is for every philosopher to take the fork with the smaller index first:

$$
\begin{array}{l}
\mathit{First}(k) \triangleq \textsc{if}\ k \neq P-1\ \textsc{then}\ k\ \textsc{else}\ 0 \\
\mathit{Second}(k) \triangleq \textsc{if}\ k \neq P-1\ \textsc{then}\ k+1\ \textsc{else}\ k
\end{array}
$$

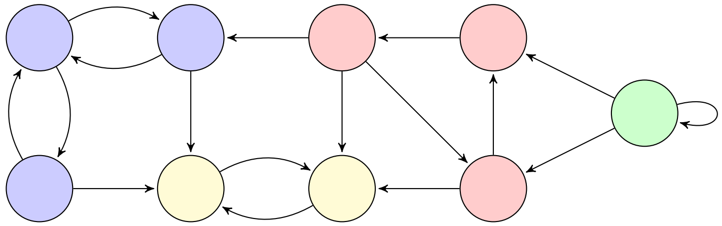
When the philosopher with the highest index wants to eat, she will need to take fork 0 first. In the case where all other philosophers have already taken their first fork, the philosopher with the highest index will fail to take her first fork (because it has already been taken by the first philosopher). This allows the philosopher with the second-highest index to make progress, thus preventing a deadlock.

The model checker will pass the updated *Spec*.

Chapter 4

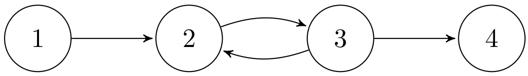
Strongly Connected Components

In a graph, a strongly connected component (SCC) is a subset of vertices where every pair of vertices are reachable from each other. An example graph with four SCCs is illustrated below, where the vertices that share the same color and belong in the same SCC:



A node is an SCC by itself. There are many applications of SCCs, including: social network analysis, web crawling, and software modularity analysis.

TLA+ also uses SCCs to verify the liveness properties of a spec. Consider the following graph from a hypothetical spec:



States 2 and 3 form an SCC. Assume the system has a liveness property that defines state 1 *leads to* state 4. The model checker can fail the spec because execution may be trapped inside the SSC (unless a fairness description is provided). The model checker must first identify all SCCs in the graph to verify liveness. This explains why verifying liveness non-trivially increases model checker runtime because the SCC identifying algorithm implemented in the model checker runs linearly.

In this chapter, we will implement a horizontally scalable SCC detection algorithm described in *A GPU Algorithm for Detecting Strongly Connected Components* [12].

4.1 Design

The following provides the pseudocode description of the parallel SCC detection algorithm as described in the paper:

$converged \leftarrow false$
while not $converged$ **do**
▷ Initialize vertex
    for all vertices $v \in V$ **do**
        $v_{in} \leftarrow v_{id}$
        $v_{out} \leftarrow v_{id}$
    end for
    ▷ Propagate max value
    $updated \leftarrow false$
    while $updated$ **do**
        for all edges $(u- > v) \in E$ **do**
            $v_{out} \leftarrow max(u_{out}, v_{out})$
            $v_{in} \leftarrow max(u_{in}, v_{in})$
        end for
        $updated \leftarrow$ at least one v_{in} or v_{out} value changed
    end while

▷ Remove edges that span SCC
for all edges $(u \rightarrow v) \in E$ **do**
  if $u_{in} \neq v_{in}$ or $u_{out} \neq v_{out}$ **then**
    $E \leftarrow E\ (u \rightarrow v)$
  end if
end for
end while

The algorithm is split into three parts: initialization, max value propagation and edge trimming.

4.2 Spec

Spec is defined into three phases: *Init*, *Update*, *Trim*. Phase *Init* is defined as follow:

$PhaseInit \triangleq$

$\land\ phase = \text{"Init"}$

$\land\ phase' = \text{"Update"}$

$\land\ edges' = new\_edges$

$\land\ new\_edges' = new\_edges$

$\land\ in' = [k \in Vertex \mapsto k]$

$\land\ out' = [k \in Vertex \mapsto k]$

$\land\ updated' = 0$

$\land\ converged' = 0$

in and *out* are defined as lookup table using *functions*.

The *Update* phase implements max value propagation:

$$
\begin{array}{l}
PhaseUpdate \triangleq \\
\quad \land\ phase = \text{"Update"} \\
\quad \land\ \lor\ \land\ \exists\, e \in edges : \\
\quad\ \ \text{LET} \\
\quad\quad\ \ src \triangleq e[1] \\
\quad\quad\ \ dst \triangleq e[2] \\
\quad\ \ \text{IN} \\
\quad\quad\ \ \land\ in' = [in \text{ EXCEPT } ![dst] = Max(in[src], in[dst])] \\
\quad\quad\ \ \land\ out' = [out \text{ EXCEPT } ![src] = Max(out[src], out[dst])] \\
\quad\quad\ \ \land\ edges' = edges \setminus \{e\} \\
\quad\quad\ \ \land\ \lor\ \land\ in' \neq in \lor out' \neq out \\
\quad\quad\ \ \land\ updated' = 1 \\
\quad\quad\ \ \lor\ \land\ in' = in \land out' = out \\
\quad\quad\ \ \land\ updated' = 0 \\
\quad \land\ \text{UNCHANGED } \langle new\_edges, phase, converged \rangle \\
\quad \lor\ \land\ edges = \{\} \\
\quad \land\ updated = 0 \\
\quad \land\ phase' = \text{"Trim"} \\
\quad \land\ \text{UNCHANGED } \langle edges, new\_edges, in, out, updated, converged \rangle \\
\quad \lor\ \land\ edges = \{\} \\
\quad \land\ updated \neq 0 \\
\quad \land\ edges' = new\_edges \\
\quad \land\ \text{UNCHANGED } \langle phase, new\_edges, in, out, updated, converged \rangle
\end{array}
$$

The edge iteration loop is implemented with an existential qualifier over edges. After processing an edge, the edge is removed from the set edges to simulate so it is not chosen again in the next iteration. Once set edges become empty, the algorithm determines if we need to repeat the propagation process.

The following implements phase *trim*:

$$
\begin{array}{l}
PhaseTrim \triangleq \\
\quad \land\ phase = \text{"Trim"} \\
\quad \land\ \lor\ \land\ edges = \{\} \\
\quad \land\ in = out \\
\quad \land\ converged' = 1 \\
\quad \land\ \text{UNCHANGED}\ \langle phase, new\_edges, edges, in, out, updated \rangle \\
\quad \lor\ \land\ edges = \{\} \\
\quad \land\ in \neq out \\
\quad \land\ phase' = \text{"Init"} \\
\quad \land\ \text{UNCHANGED}\ \langle in, new\_edges, edges, out, updated, converged \rangle \\
\quad \lor\ \land\ edges \neq \{\} \\
\quad \land\ \exists\, e \in edges : \\
\qquad \text{LET} \\
\qquad\quad src \triangleq e[1] \\
\qquad\quad dst \triangleq e[2] \\
\qquad \text{IN} \\
\qquad\quad \land\ \lor\ \land\ out[src] \neq out[dst] \lor in[src] \neq in[dst] \\
\qquad\quad \land\ new\_edges' = new\_edges \setminus \{e\} \\
\qquad\quad \lor\ \land\ out[src] = out[dst] \land in[src] = in[dst] \\
\qquad\quad \land\ \text{UNCHANGED}\ new\_edges \\
\qquad\quad \land\ edges' = edges \setminus \{e\} \\
\quad \land\ \text{UNCHANGED}\ \langle phase, in, out, updated, converged \rangle
\end{array}
$$

Similar to the previous phase, we use existential qualifiers and set subtraction to simulate iterating through all edges. Another variable *new_edges* is used to track edges to be used in the next iteration if required.

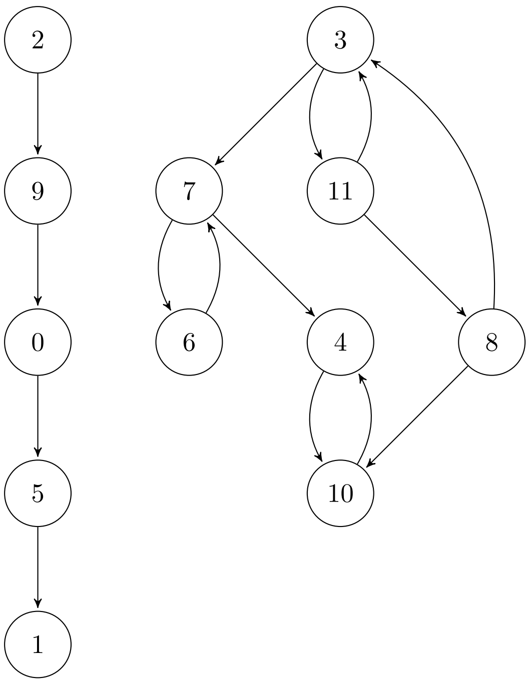
4.3 Safety

To find the solution of a given graph, let us define a safety property where converged is always 0:

$$
\begin{array}{l}
Termination \triangleq \\
\quad converged = 0
\end{array}
$$

In other words, we want the model checker to terminate when converged becomes 1.

Let us assume input similar to as defined in the paper:



Running the model checker against the spec outputs the following:

```
Error: Invariant Termination is violated.
Error: The behavior up to this point is:
...
State 15: <PhaseTrim line 76, col 5 to line 96,
    col 61 of module scc>
/\ out = ( 0 :> 0 @@
  1 :> 1 @@
  2 :> 2 @@
  3 :> 11 @@
```

```
  4 :> 10 @@
  5 :> 5 @@
  6 :> 7 @@
  7 :> 7 @@
  8 :> 11 @@
  9 :> 9 @@
  10 :> 10 @@
  11 :> 11 )
...
```

As defined by the algorithm, vertices sharing the same value are part of the same SCC. The spec correctly identifies three SCCs with more than one vertex: $\{3, 8, 11\}$, $\{6, 7\}$ and $\{4, 10\}$.

4.4 Liveness

Omitted for this chapter.

Chapter 5

Simple Scheduler

Task schedulers are ubiquitous. Every device implements *something* to manage tasks. Modern desktop or mobile device processes are non-trivial OS abstractions. Every process maintains its own virtual memory space, and the context-switching process requires the OS to "clean" the hardware before running the new process for security reasons.

For embedded devices such as hard drives or network cards, the security consideration may be relaxed as users are typically not allowed to run arbitrary code on the device. Sometimes these products don't have a full-blown operating system to save on memory and storage footprint but still need some scheduler to manage the tasks.

To solve the C10k [10] problem, languages like Rust supports asynchronous programming. Asynchronous programming enables task scheduling *within* a process to scale up system throughput. However, Rust only provides *language support* for asynchronous programming, and user must supply their runtime. The runtime must also include a scheduler to manage the tasks inside the process.

In this chapter, we will implement a very simple cooperative scheduler with tasks that share a single lock.

5.1 Design

The task scheduler has the following features:

- Supporting N execution context (ie. CPUs).
- Supporting T number of tasks.
- Tasks have identical priorities and are scheduled cooperatively.
- System shares a single locked resource.
- Any task can attempt to access the resource. Any task attempting to access the resource is guaranteed to have access at some point.
- If multiple tasks attempt to access the lock, the tasks will be scheduled in lock request order.

5.2 Spec

We will model the scheduler using the following variables:

$$
\begin{aligned}
&\mathit{Init} \triangleq \\
&\quad \land\ \mathit{cpus} = [i \in 0 \mathrel{..} N-1 \mapsto \text{``''}] \\
&\quad \land\ \mathit{ready\_q} = \mathit{SetToSeq}(\mathit{Tasks}) \\
&\quad \land\ \mathit{blocked\_q} = \langle\rangle \\
&\quad \land\ \mathit{lock\_owner} = \text{``''}
\end{aligned}
$$

A few things to note:

- The system has N executing context, represented as number of CPUs. When a task is running, *cpus[k]* is set to *taskName*. When the CPU is idle, *tcpus[k]* is set to an empty string.
- *ready_q* and *blocked_q* are initialized as *ordered tuple*, due to the cooperative scheduling requirement.
- *SetToSeq* is a macro from the community module [4] to convert a set into an ordered tuple. To use the community module, one can install required .tla files into the TLA+ project source directly.

- *lock_owner* represents the task that is currently holding the lock.

A task can be in three possible states: *Ready*, *Blocked*, and *Running*. The Spec describes required lock contention handling.

$$
\begin{aligned}
&Ready \triangleq \\
&\quad \exists\, t \in \text{DOMAIN } ready\_q : \\
&\quad\ \exists\, k \in \text{DOMAIN } cpus : \\
&\quad\quad \land\ cpus[k] = \text{``''} \\
&\quad\quad \land\ cpus' = [cpus \text{ EXCEPT } ![k] = Head(ready\_q)] \\
&\quad\quad \land\ ready\_q' = Tail(ready\_q) \\
&\quad\quad \land\ \text{UNCHANGED } \langle lock\_owner,\ blocked\_q \rangle \\
&Running \triangleq \\
&\quad \exists\, k \in \text{DOMAIN } cpus : \\
&\quad\quad \lor\ MoveToReady(k) \\
&\quad\quad \lor\ Lock(k) \\
&\quad\quad \lor\ Unlock(k) \\
&Next \triangleq \\
&\quad \lor\ Running \\
&\quad \lor\ Ready
\end{aligned}
$$

Next can update either a task that is running, or a task waiting to be scheduled.

A *Ready* task is popped off the ready queue and put onto an idle CPU. Since ready_q is implemented as an ordered tuple, fetching and popping the front is done using *Head* and *Tail*, respectively.

A *Running* task can either go back to the ready queue (done for now), acquire the global lock, or release the global lock. The sub-actions are defined below:

$$
\begin{aligned}
&MoveToReady(k) \triangleq \\
&\quad \land\ cpus[k] \neq \text{``''} \\
&\quad \land\ lock\_owner \neq cpus[k] \\
&\quad \land\ ready\_q' = Append(ready\_q,\ cpus[k]) \\
&\quad \land\ cpus' = [cpus \text{ EXCEPT } ![k] = \text{``''}]
\end{aligned}
$$

$\wedge$ UNCHANGED $\langle lock\_owner, blocked\_q, blocked \rangle$

MoveToReady defines the where task voluntarily goes back to the ready queue.

$Lock(k) \triangleq$
$\vee \wedge cpus[k] \neq$ ""
$\wedge lock\_owner =$ ""
$\wedge lock\_owner' = cpus[k]$
$\wedge$ UNCHANGED $\langle ready\_q, cpus, blocked\_q, blocked \rangle$
$\vee \wedge cpus[k] \neq$ ""
$\wedge lock\_owner \neq$ ""
$\wedge lock\_owner \neq cpus[k]$ cannot double lock
$\wedge blocked\_q' = Append(blocked\_q, cpus[k])$
$\wedge blocked' = [blocked$ EXCEPT $![cpus[k]] = 1]$
$\wedge cpus' = [cpus$ EXCEPT $![k] =$ ""$]$
$\wedge$ UNCHANGED $\langle ready\_q, lock\_owner \rangle$

Lock represents when a running task attempts to acquire the global lock. When the lock is free, the task grabs the lock and moves on. When the lock is already held, the task moves into the blocked queue to be scheduled when the lock is released. If multiple tasks attempt to acquire the lock while the lock is being held, the tasks will be inserted in the block queue in request order.

$Unlock(k) \triangleq$
$\vee \wedge cpus[k] \neq$ ""
$\wedge Len(blocked\_q) \neq 0$
$\wedge lock\_owner = cpus[k]$
$\wedge lock\_owner' = Head(blocked\_q)$
$\wedge cpus' = [cpus$ EXCEPT $![k] = Head(blocked\_q)]$
$\wedge ready\_q' = ready\_q \circ \langle cpus[k] \rangle$
$\wedge blocked\_q' = Tail(blocked\_q)$
$\wedge blocked' = [blocked$ EXCEPT $![Head(blocked\_q)] = 0]$
$\vee \wedge cpus[k] \neq$ ""
$\wedge Len(blocked\_q) = 0$
$\wedge lock\_owner' =$ ""

$\land$ UNCHANGED $\langle ready\_q,\ blocked\_q,\ blocked,\ cpus \rangle$

Unlock represents when a running task releases the lock. If there are no blocked tasks, the running task carries on as before. If there are blocked tasks, the first blocked task is scheduled to run immediately, and the running task is inserted at the end of the ready queue.

5.3 Safety

We can define safety properties to detect programmatic failures. For example: if a task is running on a CPU, this *implies* task cannot be blocked:

$$
\begin{aligned}
&Safety \triangleq \\
&\quad \forall\, t \in Tasks : \\
&\quad\ \ \forall\, k \in 0 \,..\, N - 1 : \\
&\quad\quad cpus[k] = t \Rightarrow blocked[t] = 0
\end{aligned}
$$

5.4 Liveness

Any tasks attempting to acquire the lock when the lock is already taken become blocked. A liveness property we can define is to check the scheduler guarantee any blocked task eventually acquires the lock and run. Before we describe this liveness property, we need to first make sure no task can *cannot* hold onto the lock indefinitely (which is something the model checker *will try*):

$$
\begin{aligned}
&Fairness \triangleq \\
&\quad \forall\, t \in Tasks : \\
&\quad\ \ \forall\, n \in 0 \,..\, (N - 1) : \\
&\quad\quad \mathrm{WF}_{vars}(HoldingLock(t) \land Unlock(n)) \\
&Spec \triangleq \\
&\quad \land\ Init \\
&\quad \land\ \Box[Next]_{vars} \\
&\quad \land\ \mathrm{WF}_{vars}(Next) \\
&\quad \land\ Fairness
\end{aligned}
$$

The weakness fairness description states that if the enabling condition for *HoldingLock* and *Unlock* continuously stays true (eg. a lock is being held and the task can unlock), the associated action, *Unlock*, must *always eventually* be called to satisfy the weak fairness requirement. We can now define the liveness property: a task blocked waiting for the lock *leads to* the task acquiring the lock:

$$
\begin{aligned}
&\mathit{Liveness} \triangleq \\
&\quad \forall\, t \in \mathit{Tasks} : \\
&\quad\;\; \mathit{blocked}[t] = 1 \leadsto \mathit{lock\_owner} = t
\end{aligned}
$$

Chapter 6

Simple Gossip Protocol

In a distributed system, a cluster of servers collectively provides a service. A distributed system may have 10s to 100s of servers working together to offer the service in a geo-diverse environment to maximize uptime. The servers often have requirements to know about each other. In the context of a distributed database, a server may need to know the key range of another of its peers. The cluster needs a way to communicate this information. One such mechanism is the gossip protocol.

Gossip protocol allows servers to fetch the latest cluster information in a distributed fashion. Before the gossip protocol, servers in a cluster learn about their neighbors by contacting a centralized server. This introduces a single failure point in the system. Gossip protocol relies on servers to initiate the data exchange, and the servers in the cluster periodically select a set of neighbors to gossip with.

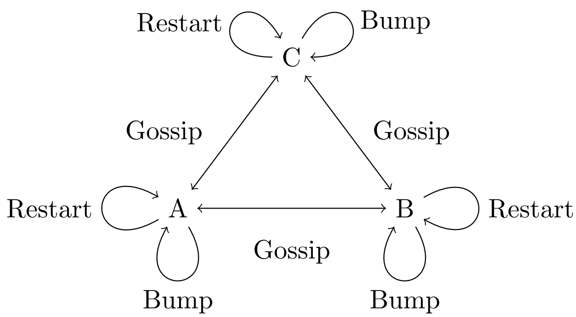
Assume an N server cluster, at some periodic interval a server selects k neighbors to gossip with. The total amount of gossip propagation time is described logarithmically below:

$$propagation\_time = \log_k N * gossip\_interval$$

With the total number of messages exchanged:

$$messages\_exchanged = \log_k N * k$$

The following represents the state graph of a three node cluster running gossip protocol:



6.1 Design

In this chapter, we will implement a simplified gossip model where:

- Each server has a version.
- Each server caches the version of all other servers.
- A pair of servers are randomly selected to gossip.
- A server can restart. Restarting a server clears the server's version cache of the other servers.
- A server can bump its version.

If the gossip protocol works correctly, every server should eventually have the latest version of all the servers.

6.2 Spec

In gossip protocol, every server needs to remember all its peer's current version:

$Init \triangleq$
$\quad \land\ version = [i \in Servers \mapsto [j \in Servers \mapsto 0]]$
$Next \triangleq$
$\quad \lor\ \exists\, i \in Servers :$
$\quad\ \ \land\ Bump(i)$
$\quad \lor\ \exists\, i, j \in Servers :$
$\quad\ \ \land\ Gossip(i, j)$
$\quad \lor\ \exists\, i \in Servers :$
$\quad\ \ \land\ Restart(i)$

The *Init* formula simply declares the version to be a two-dimensional array with all elements initialized to 0. *Next* allows either bumping the version of a server, picking a pair of servers to gossip, or restarting a server.

The following defines these steps:

$Gossip(i, j) \triangleq$
$\quad \text{LET}$
$\quad\quad Max(a, b) \triangleq \text{IF } a > b \text{ THEN } a \text{ ELSE } b$
$\quad\quad updated \triangleq [k \in Servers \mapsto Max(version[i][k], version[j][k])]$
$\quad\quad version\_a \triangleq [version \text{ EXCEPT } ![i] = updated]$
$\quad\quad version\_ab \triangleq [version\_a \text{ EXCEPT } ![j] = updated]$
$\quad \text{IN}$
$\quad\quad \land\ version' = version\_ab$

When two servers gossip, they gossip about all the servers (including themselves) and update both of their version cache with the more up-to-date entry between the two. The *LET..IN* syntax enables local macro definition. In this example, we use temporary variables defined inside *LET*, and update the primed variable inside the *IN* clause.

$Bump(i) \triangleq$
$\land\ version[i][i] \neq MaxVersion$
$\land\ version' = [version \text{ EXCEPT } ![i] = [k \in Servers \mapsto$
$\text{IF } i \neq k \text{ THEN } version[i][k] \text{ ELSE } version[i][k] + 1]]$

Bump only increments the version if the server hasn't made it to *MaxVersion*. When the Server bumps the version, it only bumps its version and keeps all other versions in its version cache as is.

$$
\begin{array}{l}
\mathit{Restart}(i) \triangleq \\
\quad \land\ \mathit{version}' = [\mathit{version}\ \text{EXCEPT}\ ![i] = [k \in \mathit{Servers} \mapsto \\
\quad\ \ \text{IF}\ i \neq k\ \text{THEN}\ 0\ \text{ELSE}\ \ \mathit{version}[i][i]]]
\end{array}
$$

Upon *Restart*, a server reloads from its local storage (so its version persists), but the server needs to re-learn the cluster status (all other entries in its version cache are wiped).

6.3 Safety

One safety property is to confirm *all* version values are within bounds:

$$
\begin{array}{l}
\mathit{Safety} \triangleq \\
\quad \forall\, i, j \in \mathit{Servers} : \\
\quad\ \ \land\ \mathit{version}[i][j] \geq 0 \\
\quad\ \ \land\ \mathit{version}[i][j] \leq \mathit{MaxVersion}
\end{array}
$$

This can be read as: for all possible pairs of servers i and j, the value of *version[i][j]* must be within 0 and *MaxVersion*, inclusive.

6.4 Liveness

Spec defines three actions: *Bump*, *Restart*, *Gossip*. Without any fairness description, *Any* permutation of these actions are allowed by *Spec*, including:

- Restart, Restart, Restart,...
- Restart, Gossip, Restart, Gossip, ...
- Gossip, Gossip, Gossip, ...

Some of these are not of interest, for example, the cluster is stuck in a loop where everyone is constantly restarting. In the gossip protocol, we are interested in verifying the cluster over time converges towards higher version value for all servers. This means we need to make sure *Bump* is guaranteed to be called under some circumstances. This is where the fairness description comes in. To ensure *Bump* is always called:

$$
\begin{aligned}
&Spec \triangleq \\
&\quad \land Init \\
&\quad \land \Box[Next]_{vars} \\
&\quad \land \mathrm{WF}_{vars}(Next) \\
&\quad \land \forall\, i \in Servers : \\
&\qquad \mathrm{WF}_{vars}(Bump(i))
\end{aligned}
$$

The model checker explores all possible transitions permitted by *Spec*, including calling any subset of actions repeatedly. Fairness description guarantees that if the enabling condition of an action is true, the action will be taken. If the system is trapped in a *Restart* and *Gossip* loop when *Bump* can be called, specifying fairness for *Bump* ensures *Bump* is called, breaking the loop.

With *Spec* ensuring the system always migrate towards higher version number, we can now define the Liveness property:

$$
\begin{aligned}
&Liveness \triangleq \\
&\qquad \exists\, i, j \in Servers : \\
&\qquad \land\; i \neq j \\
&\qquad \land\; \Box\Diamond(\land\; version[i][i] \;= MaxVersion \\
&\qquad\quad \land\; version[i][j] = MaxVersion \\
&\qquad\quad \land\; version[j][i] = MaxVersion \\
&\qquad\quad \land\; version[j][j] = MaxVersion)
\end{aligned}
$$

The $\Box\Diamond$ represents *always eventually.* The liveness condition specifies that there exists a pair of Servers such that both of them *always eventually* make it to *MaxVersion* and have *Gossip* with each other.

Since *Spec* permits *Restart* to be called anytime, a liveness property where *all* Servers are up-to-date cannot be true. The model checker can always *Restart* one of the Servers before this property is met.

Likewise, replacing *always eventually* with *eventually always* ($\Diamond\Box$) also fails. $\Diamond\Box$ checks that once the system *eventually* enters a specified state, it *always* remains in that state. This cannot be true as the liveness condition is transient, since the model checker can always disturb any condition with a *Restart*.

For a more comprehensive discussion of fairness, please refer to Chapter 13.

Chapter 7

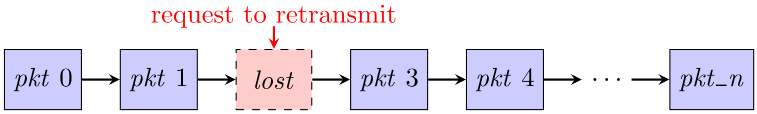
Selective Retransmit

Assume a client device that plays a video stream. Structurally, a video is composed of frames, frames are then segmented into packets to stream across a network. The client device recombines the packets into a frame and then sequences the frame to playback the video.

However, the network is not-deterministic. Depending on the route the packets take to get to the client, they may arrive out-of-order. The client may need to maintain a receive buffer for the packets and re-order the packets back into sequence before pushing the packets down to the decoding engine.

The network may also drop packets if any of the switches along the way get busy. In the case of a packet drop, the client has a few options. The client can either discard the frame and let the decoding engine downstream deal with it (which may result in visible artifacts during playback). The client can request the whole frame to be re-sent, which results in additional bandwidth consumption. The client can selectively request the missing packet to be retransmitted, which will minimize additional bandwidth consumption but increase implementation complexity.

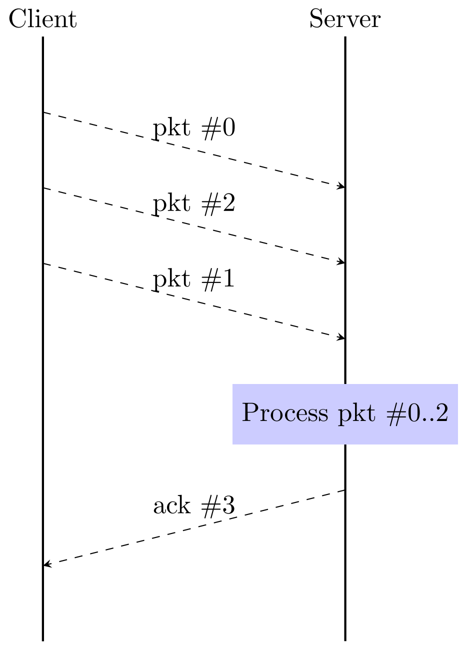
In this chapter, we will implement a simple selective retransmit algorithm.



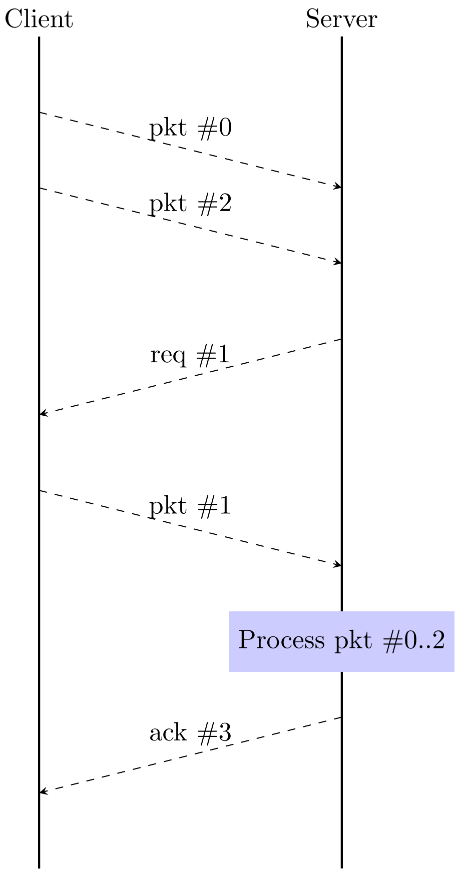
Since packets may arrive out-of-order, the server stamps the packets with sequence numbers to allow the client to order the packets as they arrive. Once the client has a set of ordered packets, it moves the packets from the receive buffer into the decoding engine to be displayed.

The video packets are often sent via unreliable channels to minimize network overhead and latency. The client sends an acknowledgement back to the server to acknowledge the received packet. This indicates to the server it can send more video data to the client. Acknowledgements are not latency-sensitive and take up a very small proportion of bandwidth, so they are transported through reliable channels.

The following illustrates packet reorder handling:



The following illustrates packet loss handling:



There are other design considerations. The server is allowed to send up to W packets before getting an acknowledgement, this reduces latency perceived by the user. The client also doesn't need to acknowledge all the packets, since the server assumes once an acknowledgement of packet N is received, then all packets before N has also been received.

7.1 Design

With the above description, we are now ready to provide a more formal description of our design:

- Client is the receiver that displays the video stream.
- Server is the sender that sends the video stream.
- Server always sends the packets in order.
- Client may receive the packets out-of-order.
- Client may never receive some packet due to loss.
- Server can send up to W packets before an acknowledgement is received
- Packet sequence number is represented by a fixed number of bytes in the network header, the sequence number will eventually wrap around once it hits the maximum representable value. The maximum sequence number is represented as N-1.
- Client puts a received packet in its receive buffer. Packets in the receive buffer may be out-of-order due to network conditions.
- Client will remove the packets from the receive buffer once the sequence number of the received packets is contiguous. The client will also send an acknowledgement back to the Server with the most recently acknowledged sequence number.
- Data packets are transported using unreliable channels due to bandwidth and latency requirements.
- Control packets are transported using reliable channels due to relaxed and latency requirements.

We are now ready to implement the *Spec*.

7.2 Spec

The following is the skeleton of the Spec:

$$
\begin{aligned}
&\mathit{Init} \triangleq \\
&\quad \land\ \mathit{network} = \{\} \\
&\quad \land\ \mathit{server\_tx} = 0 \\
&\quad \land\ \mathit{server\_tx\_limit} = W \\
&\quad \land\ \mathit{server\_tx\_ack} = 0 \\
&\quad \land\ \mathit{client\_rx} = 0 \\
&\quad \land\ \mathit{client\_buffer} = \{\} \\
&\quad \land\ \mathit{lost} = 0
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Next} \triangleq \\
&\quad \lor\ \mathit{Send} \\
&\quad \lor\ \exists\, p \in \mathit{network} : \\
&\quad\ \ \mathit{Receive}(p) \\
&\quad \lor\ \mathit{ClientRetransmitRequest} \\
&\quad \lor\ \mathit{ClientAcknowledgement} \\
&\quad \lor\ \exists\, p \in \mathit{network} : \\
&\quad\ \ \land\ p.\mathit{dst} = \text{“client”} \\
&\quad\ \ \land\ \mathit{Drop}(p)
\end{aligned}
$$

The server is represented by three variables:

- tx+1 represents the sequence number to be used in the next packet
- tx_liimit represents the highest sequence number the server can send without waiting for an acknowledgement
- tx_ack represents the most recent acknowledged sequence number

The client is represented by two variables:

- client_rx is the most recently acknowledged sequence number
- client_buffer is the receive buffer holding all the packets waiting to be re-ordered before being acknowledged

The allowed actions include packet *Receive*, which the existential quantifier also has the side effect of re-ordering. *ClientRetransmitRequest* detects and sends retransmit requests. *ClientAcknowledgement* sends acknowledgement. Finally, data packets may be dropped.

Before we start defining the actions, let us define some helper functions:

$MinS(s) \triangleq$
$\text{CHOOSE } x \in s : \forall\, y \in s : x \leq y$

$MaxS(s) \triangleq$
$\text{CHOOSE } x \in s : \forall\, y \in s : x \geq y$

$MaxIndex \triangleq$
$\quad \text{LET}$
$\quad\quad upper \triangleq \{x \in client\_buffer : x > N - W\}$
$\quad\quad lower \triangleq \{x \in client\_buffer : x < W\}$
$\quad\quad maxv \triangleq \text{IF } upper \neq \{\} \land lower \neq \{\}$
$\quad\quad\; \text{THEN}$
$\quad\quad\quad MaxS(lower)$
$\quad\quad\; \text{ELSE}$
$\quad\quad\quad MaxS(client\_buffer)$
$\quad \text{IN}$
$\quad\quad maxv$

$MinIndex \triangleq$
$\quad \text{LET}$
$\quad\quad upper \triangleq \{x \in client\_buffer : x > N - W\}$
$\quad\quad lower \triangleq \{x \in client\_buffer : x < W\}$
$\quad\quad minv \triangleq \text{IF } upper \neq \{\} \land lower \neq \{\}$
$\quad\quad\; \text{THEN}$
$\quad\quad\quad MinS(upper)$
$\quad\quad\; \text{ELSE}$
$\quad\quad\quad MinS(client\_buffer)$
$\quad \text{IN}$
$\quad\quad minv$

$Range \triangleq$

$$\begin{aligned}&\textsc{if } MaxIndex \geq MinIndex\\ &\quad\textsc{then}\\ &\qquad MaxIndex - MinIndex + 1\\ &\quad\textsc{else}\\ &\qquad MaxIndex + 1 + N - MinIndex\end{aligned}$$

At any moment the system allows a window of packets to be unacknowledged. Both the client and server are aware of the window size, represented by W. By looking at packets in its receive buffer and its most acknowledged sequence number, the client can determine which packets were lost. There's some nuisance to implement this.

Since the system does not allow more than W unacknowledged packets, the client can assume the window of the packet in its receiver buffer must have sequence number $s \in client\_rx..client\_rx + W$. Since the sequence number has a ceiling, the window of packets may wrap around the boundary. This introduces some complications around determining the minimum and maximum in the window of the packet. The functions defined above calculate the range, maximum and minimum value in the window accounting for wraparound.

Now we can look at how the client acknowledgement logic:

$$\begin{aligned}&MergeReady \triangleq\\ &\quad\land\ client\_buffer \neq \{\}\\ &\quad\land\ (client\_rx + 1)\%N = MinIndex && \text{contiguous with previous ack}\\ &\quad\land\ Range = Cardinality(client\_buffer) && \text{combined is contiguous}\end{aligned}$$

$$\begin{aligned}&ClientAcknowledgement \triangleq\\ &\quad\land\ client\_buffer \neq \{\}\\ &\quad\land\ MergeReady\\ &\quad\land\ client\_buffer' = \{\}\\ &\quad\land\ client\_rx' = MaxIndex\\ &\quad\land\ network' = AddMessage([dst \mapsto \text{“server”},\\ &\quad type \mapsto \text{“ack”},\\ &\quad ack \mapsto MaxIndex],\\ &\qquad network)\end{aligned}$$

$\land$ UNCHANGED $\langle server\_tx,\ server\_tx\_ack,\ server\_tx\_limit,\ lost\rangle$

The client only acknowledges when it has a contiguous sequence of packets that follows its most recently acknowledged packet. When MergeReady is true, the client sends the acknowledgement back to the server.

$ClientReceive(pp) \triangleq$
$\quad \land\ network' = RemoveMessage(pp,\ network)$
$\quad \land\ client\_buffer' = client\_buffer \cup \{pp.seq\}$
$\quad \land$ UNCHANGED $\langle server\_tx,\ client\_rx,$
$\quad server\_tx\_ack,\ server\_tx\_limit,\ lost\rangle$

$Missing \triangleq$
LET
$\quad full\_seq \triangleq$
$\quad\quad$ IF $MaxIndex \geq client\_rx + 1$
$\quad\quad$ THEN
$\quad\quad\quad \{x \in client\_rx + 1 \,..\, MaxIndex : \text{TRUE}\}$
$\quad\quad$ ELSE
$\quad\quad\quad \{x \in 0 \,..\, MaxIndex : \text{TRUE}\} \cup \{x \in client\_rx + 1 \,..\, N - 1 : \text{TRUE}\}$
$\quad all\_client\_msgs \triangleq \{m \in network : m.dst = \text{“client”}\}$
$\quad all\_client\_seqs \triangleq \{m.seq : m \in all\_client\_msgs\}$
$\quad network\_missing \triangleq full\_seq \setminus all\_client\_seqs$
$\quad client\_missing \triangleq full\_seq \setminus client\_buffer$
$\quad to\_request \triangleq network\_missing \cap client\_missing$
IN
$\quad to\_request$

$ClientRetransmitRequest \triangleq$
$\quad \land\ \neg MergeReady$
$\quad \land\ client\_buffer \neq \{\}$
$\quad \land\ Missing \neq \{\}$
$\quad \land\ network' = AddMessage([dst \mapsto \text{“server”},$
$\quad type \mapsto \text{“retransmit”},$
$\quad seq \mapsto \text{CHOOSE } x \in Missing : \text{TRUE}],$
$\quad network)$
$\quad \land$ UNCHANGED $\langle server\_tx,\ server\_tx\_limit,$

$client\_rx$, $client\_buffer$, $server\_tx\_ack$, $lost\rangle$

ClientReceive moves the packet from the network into the client receive buffer. The only reason why this is done as a separate step is to make debugging easier.

Missing returns a set of missing missing sequence numbers. This is done by cross-checking the client receive buffer and the outstanding network packet targeting the client. In theory, the client doesn't know if a gap in its receive buffer means the packet is lost or will arrive soon. Practically, the client will assume a packet is lost after some configurable timeout and request a retransmit.

Let us take a look at server-related definitions:

$RemoveStaleAck(ack, msgs) \triangleq$
  LET
    $acks \triangleq \{(ack - k + N)\%N : k \in 1 \,..\, W\}$
  IN
    $\{m \in msgs : \neg(\land m.dst = \text{"server"}$
      $\land m.type = \text{"ack"}$
      $\land m.ack \in acks)\}$

$ServerReceive(pp) \triangleq$
  $\lor \land pp.type = \text{"ack"}$
  $\land server\_tx\_ack' = pp.ack$
  $\land server\_tx\_limit' = (pp.ack + W)\%N$
  $\land network' = RemoveStaleAck(pp.ack, RemoveMessage(pp, network))$
  $\land$ UNCHANGED $\langle server\_tx, client\_rx, client\_buffer, lost\rangle$
  $\lor \land pp.type = \text{"retransmit"}$
  $\land network' = AddMessage([dst \mapsto \text{"client"}, seq \mapsto pp.seq],$
  $RemoveMessage(pp, network))$
  $\land lost' = lost - 1$
  $\land$ UNCHANGED $\langle server\_tx, server\_tx\_limit,$
    $client\_rx, client\_buffer, server\_tx\_ack\rangle$

When the server receives an acknowledgement for sequence number K, it assumes K-1 and prior were all received by the client. *Re-*

moveStaleAck is a model optimization to drop all acknowledgements with sequence numbers less than K. Note that sequence numbers k, k-N, and k-2*N, are all represented as k, and intheory, the client may not be able to differentiate between them. Practically, N is sized large enough to represent a few second's worth of data, so the system can safely assume a sequence number k is for the most recent N packets.

Upon receiving an acknowledgement from the client, the server bumps the server_tx_limit allowing it to send more data. The server can also receive a retransmit request and send the requested data. *lost* is configurable to determine how many packets can be dropped at the same time.

7.3 Model Reduction

7.3.1 Removing Stale Acknowlegement

When the server acknowledges K, it has already received all the packets before K. Likewise, when the client receives an acknowledgement for K, it can assume all previous packets have been received. The client may receive the acknowledgement out-of-order due to unfavourable network conditions. One reduction we can make to the model is simply *remove all stale acknowledgements*. In a practical implementation, this is also how the client can react when it receives stale acknowledgements.

There is a bit of nuisance here. Since the sequence number wraps around, the client cannot differentiate between K from K-N, K-2*N, etc. However, N is usually large enough to accommodate a few seconds of data before a sequence number is re-used. The client can be confident that any K received the current K, not K-N or earlier.

7.3.2 Retransmit Request Assumed Reliable

Retransmit requests packets can be delivered through unreliable channels. However, this makes no difference in the model. The client sends a retransmit request when it detects packets are lost. If the model drops the retransmit request, the client will just send another retransmit request because the missing packet is still missing.

While this can be modeled, it would increase the runtime without providing many benefits. For this Spec, we simply assume all control packets are reliable.

7.4 Safety

Omitted for this chapter.

7.5 Liveness

Given the system may randomly drop packets, one possible liveness condition is to verify packets of all sequence numbers are received by the client at some point. This can be described as for all possible sequence number value k, k is eventually exists in the client receive buffer.

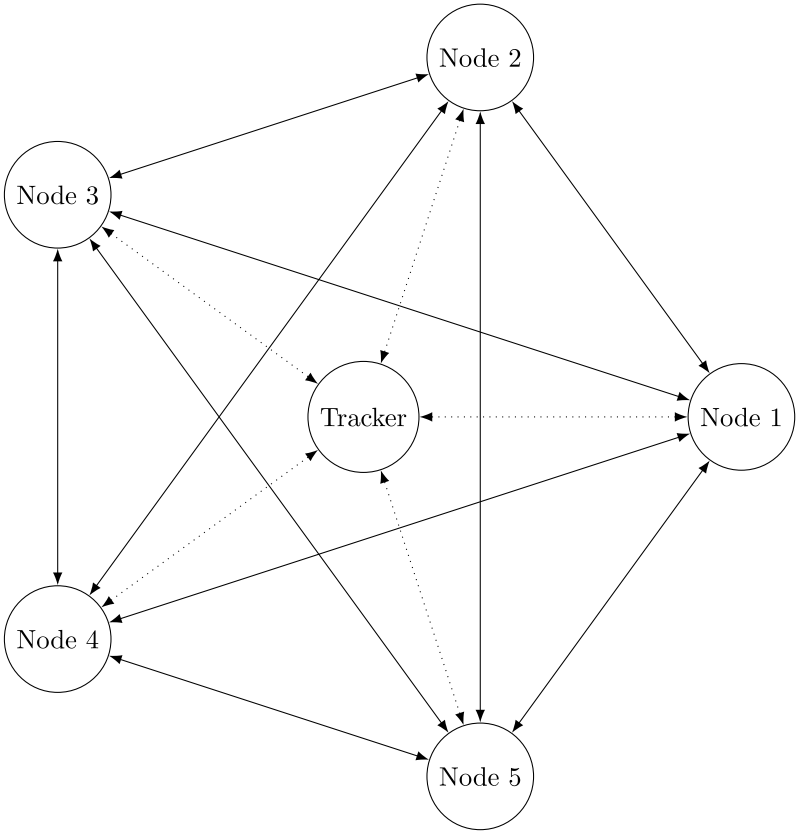
$$
\begin{aligned}
&\mathit{Liveness} \triangleq \\
&\quad \forall\, i \in 0 \,..\, N-1 : \\
&\quad\;\; \mathit{client\_buffer} = \{\} \leadsto \exists\, j \in \mathit{client\_buffer} : i = j
\end{aligned}
$$

Chapter 8

BitTorrent Protocol

BitTorrent is a peer-to-peer file-sharing protocol that allows nodes to distribute data in a decentralized fashion. The tracker is the central server that knows all the members in the cluster. A node coming online learns about its peers by talking to the tracker. Once a node learns about the peer nodes, the node can initiate data exchange with its peers directly. Files are exchanged in chunks, and a node will contact its peers to determine who has data it doesn't have.

The following is a BitTorrent cluster with five nodes:



8.1 Design

In this chapter, we will implement a simple BitTorrent cluster. The cluster will have a single tracker and N nodes. A node can join or leave the cluster, and make progress while inside the cluster.

To simplify the model, we assume a single file with N chunks is shared within the cluster. We also assume at any moment the cluster contains the entire file. This means while a node can leave the cluster at any time, it can only leave if the cluster still has the entire file after it leaves.

8.2 Spec

The cluster starts with a single *Seed* node that has the entire data set. In every step, a node can *Join*, *Progress*, or *Leave*.

$$\begin{aligned}
&\mathit{Init} \triangleq \\
&\quad \land\ \mathit{tracker} = \{\mathit{Seed}\} \\
&\quad \land\ \mathit{data} = [k \in \mathit{Client} \mapsto \text{IF } k = \mathit{Seed} \text{ THEN } \mathit{AllChunks} \text{ ELSE } \{\}]
\end{aligned}$$

$$\begin{aligned}
&\mathit{Next} \triangleq \\
&\quad \lor\ \exists\, k \in \mathit{Client} : \\
&\quad\ \ \mathit{Join}(k) \\
&\quad \lor\ \exists\, k \in \mathit{Client} : \\
&\quad\ \ \mathit{Progress}(k) \\
&\quad \lor\ \exists\, k \in \mathit{Client} : \\
&\quad\ \ \mathit{Leave}(k)
\end{aligned}$$

A node can join a cluster if it's not currently part of it:

$$\begin{aligned}
&\mathit{Join}(k) \triangleq \\
&\land\ k \notin \mathit{tracker} \\
&\land\ \mathit{tracker}' = \mathit{tracker} \cup \{k\} \\
&\land\ \text{UNCHANGED } \mathit{data}
\end{aligned}$$

A node can leave a cluster. To simplify the model, a node k can leave if and only if the cluster still has the full data set without k. A node can *Leave* even if it doesn't have the entire file:

$$\begin{aligned}
&\mathit{AllDataWithout}(k) \triangleq \\
&\quad \text{UNION } \{\mathit{data}[i] : i \in \mathit{tracker} \setminus \{k\}\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{Leave}(k) \triangleq \\
&\quad \land\ k \in \mathit{tracker} \\
&\quad \land\ \mathit{AllDataWithout}(k) = \mathit{AllChunks}
\end{aligned}$$

$\land$ $tracker' = tracker \setminus \{k\}$
$\land$ $data' = [data \text{ EXCEPT } ![k] = \{\}]$

Finally, for a node U to make progress: U must have an incomplete set of data and there's another node V that has data U doesn't have:

$Progress(u) \triangleq$
$\land\ u \in tracker$
$\land\ data[u] \neq AllChunks$ u is incomplete
$\land\ \exists\, v \in tracker :$
$\exists\, k \in AllChunks :$
$\land\ k \notin data[u]$ v has something u doesn't
$\land\ k \in data[v]$
$\land\ data' = [data \text{ EXCEPT } ![u] = data[u] \cup \{k\}]$
$\land$ UNCHANGED $tracker$

8.3 Safety

A requirement we imposed on the design is to ensure the cluster contains the entire file at all times:

$Safety \triangleq$
UNION $\{data[k] : k \in Client\} = AllChunks$

8.4 Liveness

We want to verify a newly joined node eventually gets the entire file. However, verifying *all* nodes eventually get the whole file will take too long. We will define the check liveness property for only one node:

$NodeToVerify \triangleq$ "c0"

$Liveness \triangleq$
targeted liveness condition
$data[NodeToVerify] = \{\} \leadsto data[NodeToVerify] = AllChunks$

The liveness property checks for a node with no data and eventually gets all the data.

$$
\begin{aligned}
&\mathit{Spec} \triangleq \\
&\quad \land\ \mathit{Init} \\
&\quad \land\ \Box[\mathit{Next}]_{vars} \\
&\quad \land\ \mathrm{WF}_{vars}(\mathit{Next}) \\
&\quad\ \ \text{targeted fairness description} \\
&\quad \land\ \mathrm{SF}_{vars}(\mathit{Join}(\mathit{NodeToVerify})) \\
&\quad \land\ \forall\, s \in \textsc{subset}\ \mathit{AllChunks} : \\
&\quad\ \ \mathrm{SF}_{vars}(\mathit{data}[\mathit{NodeToVerify}] = s \land \mathit{Progress}(\mathit{NodeToVerify}))
\end{aligned}
$$

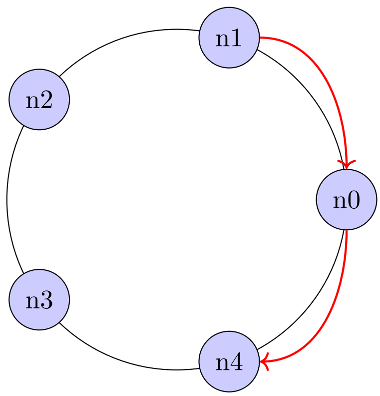
Without fairness, *Spec* permits a node to *never* make progress by having it repeatedly join and leave the cluster. To ensure *NodeToVerify* always makes progress, we need to enable strong fairness for *Progress*. However, this alone is not enough. We need to ensure *Progress* is called for *all* possible *NodeToVerify* state, this is solved using the *SUBSET* keyword.

Chapter 9

Consistent Hashing

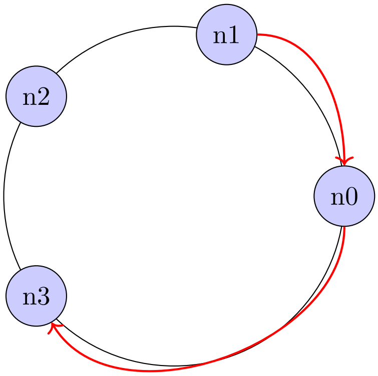
To a distributed system, traffic is hashed then distributed to different servers for load balancing. Given the intent of a distributed system is to enable horizontal scaling, the system needs to support dynamic upscaling of the cluster. With traditional hashing algorithm, changing the size of hash space requires data movement of the entire cluster. This is very undesirable. Consistent hashing was introduced to minimize data movement, data movement is only required when introducing nodes in the affected range.

In consistent hashing, the hash space is assumed to be a ring, where the largest hash value plus one wraps around back to hash of 0. Servers in a consistent hashing cluster takes up different ranges in the ring. For a given request, the client where the request lands by hashing the request first, then walk the ring clockwise until it finds a server. Assume the following example:

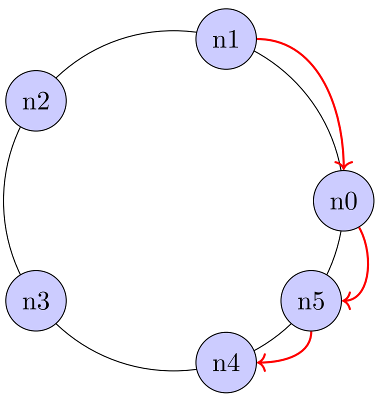


If the request lands between n1 (exclusive) and n0 (inclusive), the request will be processed by n0. Similarly, if the request lands between n0 (exclusive) and n4 (inclusive), the request is to be processed by n4.

Assume a case where n4 goes offline:



In such case requests that were processed by n4 will land on n3 instead. Similarly, if a new node n5 is added:



Part of what n4 used to service will now be serviced by n5.

9.1 Design

9.2 Spec

Chapter 10

Raft Consensus Protocol

Raft is a consensus algorithm that enables a cluster of nodes to agree on a collective state even in the presence of failures. An application of Raft is a database replication protocol. With a replication factor of 3 (eg. data is replicated across 3 nodes) and a hard drive failure rate of 0.81% per year, the possibility of total failure where the entire replication group goes down is $1 - 0.0081^3 = 99.9999\%$ uptime [6].

This chapter implements only the leader election portion of the protocol to limit the scope of the discussion. For a full description of the Raft protocol, please refer to the original paper [7].

10.1 Design

We will briefly describe Raft and its leadership election process below:

- A Raft cluster has N nodes, the cluster works collectively as a *system* to offer some service
- Each node can be in one of three possible states: Follower, Candidate, Leader
- During normal operations, a cluster of N nodes has a single leader and N-1 followers

- The leader handles all the client interactions. Requests sent to followers will be redirected to the leader

- The leader regularly sends a heartbeat to the follower, indicating its alive

- If a follower fails to receive a heartbeat from the leader after timeout, it will become a candidate, vote for itself, and campaign to be leader

- A candidate who collects the majority of the vote becomes the leader

- If multiple candidates are campaigning and a split vote happens, candidates will eventually declare an election timeout and start a new round of election

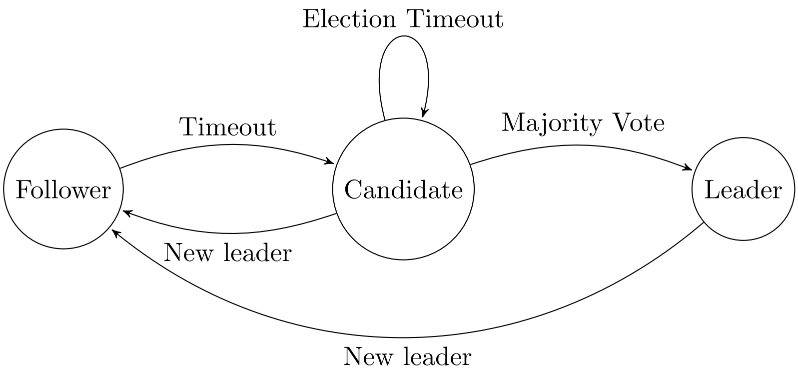
- The cluster can have multiple leaders due to unfavorable network conditions, but the leaders must be on different terms

- A newly elected leader will send a heartbeat to other nodes to establish leadership

- All requests and responses include the sender's term, allowing the receiver to react accordingly

The protocol also included a description of log synchronization, state recovery, and more. Many details are omitted in this chapter to reduce modeling costs. The N nodes in the cluster operate *independently* following the above heuristics. Hopefully, this highlights the complexity of verifying the correctness of the protocol.

The following illustrates the state diagram of one node in the cluster:



10.2 Spec

The following implements the skeleton portion of the leader election protocol:

$$
\begin{array}{l}
\mathit{Init} \triangleq \\
\quad \land\ \mathit{state} = [s \in \mathit{Servers} \mapsto \text{“Follower”}] \\
\quad \land\ \mathit{messages} = \{\} \\
\quad \land\ \mathit{voted\_for} = [s \in \mathit{Servers} \mapsto \text{“”}] \\
\quad \land\ \mathit{vote\_granted} = [s \in \mathit{Servers} \mapsto \{\}] \\
\quad \land\ \mathit{vote\_requested} = [s \in \mathit{Servers} \mapsto 0] \\
\quad \land\ \mathit{term} = [s \in \mathit{Servers} \mapsto 0]
\end{array}
$$

$$
\begin{array}{l}
\mathit{RequestVoteSet}(i) \triangleq \{ \\
\quad [\mathit{fSrc} \mapsto i,\ \mathit{fDst} \mapsto s,\ \mathit{fType} \mapsto \text{“RequestVoteReq”},\ \mathit{fTerm} \mapsto \mathit{term}[i]] \\
\qquad : s \in \mathit{Servers} \setminus \{i\} \\
\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Campaign}(i) \triangleq \\
\quad \land\ \mathit{vote\_requested}[i] = 0 \\
\quad \land\ \mathit{vote\_requested}' = [\mathit{vote\_requested}\ \text{EXCEPT}\ ![i] = 1] \\
\quad \land\ \mathit{messages}' = \mathit{messages} \cup \mathit{RequestVoteSet}(i) \\
\quad \land\ \text{UNCHANGED}\ \langle \mathit{state},\ \mathit{term},\ \mathit{vote\_granted},\ \mathit{voted\_for} \rangle
\end{array}
$$

$\mathit{KeepAliveSet}(i) \triangleq \{$

$$
\begin{array}{l}
\quad [fSrc \mapsto i,\ fDst \mapsto s,\ fType \mapsto \text{“AppendEntryReq”},\ fTerm \mapsto term[i]] \\
\qquad : s \in Servers \setminus \{i\} \\
\}
\end{array}
$$

$$
\begin{array}{l}
Leader(i) \triangleq \\
\quad \land\ state[i] = \text{“Leader”} \\
\quad \land\ messages' = messages \cup KeepAliveSet(i) \\
\quad \land\ \text{UNCHANGED } \langle state,\ voted\_for,\ term,\ vote\_granted,\ vote\_requested \rangle
\end{array}
$$

$$
\begin{array}{l}
BecomeLeader(i) \triangleq \\
\quad \land\ Cardinality(vote\_granted[i]) > Cardinality(Servers) \div 2 \\
\quad \land\ state' = [state \text{ EXCEPT } ![i] = \text{“Leader”}] \\
\quad \land\ \text{UNCHANGED } \langle messages,\ voted\_for, \\
\quad\ \ term,\ vote\_granted,\ vote\_requested \rangle
\end{array}
$$

$$
\begin{array}{l}
Candidate(i) \triangleq \\
\quad \land\ state[i] = \text{“Candidate”} \\
\quad \land\ \lor\ Campaign(i) \\
\quad \lor\ BecomeLeader(i) \\
\quad \lor\ Timeout(i)
\end{array}
$$

$$
\begin{array}{l}
Follower(i) \triangleq \\
\quad \land\ state[i] = \text{“Follower”} \\
\quad \land\ Timeout(i)
\end{array}
$$

$$
\begin{array}{l}
Receive(msg) \triangleq \\
\quad \lor\ \land\ msg.fType = \text{“AppendEntryReq”} \\
\quad \land\ AppendEntryReq(msg) \\
\quad \lor\ \land\ msg.fType = \text{“AppendEntryResp”} \\
\quad \land\ AppendEntryResp(msg) \\
\quad \lor\ \land\ msg.fType = \text{“RequestVoteReq”} \\
\quad \land\ RequestVoteReq(msg) \\
\quad \lor\ \land\ msg.fType = \text{“RequestVoteResp”} \\
\quad \land\ RequestVoteResp(msg)
\end{array}
$$

$$
\begin{array}{l}
Next \triangleq \\
\quad \lor\ \exists\, i \in Servers : \\
\quad \lor\ Leader(i) \\
\quad \lor\ Candidate(i) \\
\quad \lor\ Follower(i)
\end{array}
$$

$\lor \exists\, msg \in messages : Receive(msg)$

- *Next* either picks a server to make progress, or picks a message in the message pool to process. Message processing is done by *Receive*, handling is state agnostic
- *message* is defined to be a set that holds a collection of functions, where each function is a message with source, destination, type, and more specified
- *voted_for* tracks who a given node previously voted for. This prevents a node from voting more than once
- *vote_granted* tracks how many votes a candidate has received
- *vote_requested* tracks if a node has already issued a request vote to its peers
- *Follower* either Receive or Timeout and campaign to be a leader
- *Candidate* campaigns to be a leader, and becomes one if it has enough vote. Failing to collect enough votes, *Candidate* start a new election on a new term. It can also receive a request with a higher term and transition to be a *Follower*.
- *Leader* will establish its leadership by sending *AppepndEntryReq* to all its peers

Spec implements four messages AppendEntry request/response, RequestVote request/response. Handling for all messages is similar in structure. In this chapter, we will look at *RequestVoteReq* only. Readers are encouraged to check the remaining definition as an exercise:

$$
\begin{array}{l}
RequestVoteReq(msg) \triangleq \\
\quad \text{LET} \\
\qquad i \triangleq msg.fDst \\
\qquad j \triangleq msg.fSrc \\
\qquad type \triangleq msg.fType \\
\qquad t \triangleq msg.fTerm \\
\quad \text{IN}
\end{array}
$$

haven't voted, or whom we voted re-requested
$\lor \ \land t = term[i]$
$\land \ \lor voted\_for[i] = j$
$\lor voted\_for[i] = \text{""}$
$\land voted\_for' = [voted\_for \text{ EXCEPT } ![i] = j]$
$\land messages' = AddMessage([fSrc \mapsto i,$
$fDst \mapsto j,$
$fType \mapsto \text{"RequestVoteResp"},$
$fTerm \mapsto t,$
$fSuccess \mapsto 1],$
$RemoveMessage(msg, messages))$
$\land \text{ UNCHANGED } \langle state, term, vote\_granted,$
$vote\_requested, establish\_leadership\rangle$
already voted for someone else
$\lor \ \land t = term[i]$
$\land voted\_for[i] \neq j$
$\land voted\_for[i] \neq \text{""}$
$\land messages' = AddMessage([fSrc \mapsto i,$
$fDst \mapsto j,$
$fType \mapsto \text{"RequestVoteResp"},$
$fTerm \mapsto t,$
$fSuccess \mapsto 0],$
$RemoveMessage(msg, messages))$
$\land \text{ UNCHANGED } \langle state, voted\_for, term,$
$vote\_granted, vote\_requested, establish\_leadership\rangle$
$\lor \ \land t < term[i]$
$\land messages' = AddMessage([fSrc \mapsto i,$
$fDst \mapsto j,$
$fType \mapsto \text{"RequestVoteResp"},$
$fTerm \mapsto term[i],$
$fSuccess \mapsto 0],$
$RemoveMessage(msg, messages))$
$\land \text{ UNCHANGED } \langle state, voted\_for, term,$
$vote\_granted, vote\_requested, establish\_leadership\rangle$
revert to follower
$\lor \ \land t > term[i]$
$\land state' = [state \text{ EXCEPT } ![i] = \text{"Follower"}]$
$\land term' = [term \text{ EXCEPT } ![i] = t]$

$$
\begin{aligned}
&\land\ voted\_for' = [voted\_for\ \text{EXCEPT}\ ![i] = j] \\
&\land\ vote\_granted' = [vote\_granted\ \text{EXCEPT}\ ![i] = \{\}] \\
&\land\ vote\_requested' = [vote\_requested\ \text{EXCEPT}\ ![i] = 0] \\
&\land\ establish\_leadership' = [establish\_leadership\ \text{EXCEPT}\ ![i] = 0] \\
&\land\ messages' = AddMessage([fSrc \mapsto i, \\
&fDst \mapsto j, \\
&fType \mapsto \text{“RequestVoteResp”}, \\
&fTerm \mapsto t, \\
&fSuccess \mapsto 1], \\
&RemoveMessage(msg,\ messages))
\end{aligned}
$$

The handling is split into three cases:

- If the received request is on a higher term, the processing node grants a vote and becomes a Follower
- If the received request is on a lower term, the processing node ignores the request
- If the received request is on the same term, the processing node only grants vote if it hasn't voted, or has voted for the same requester prior

10.3 Model Reduction

The model checker will run *Spec* as defined but is unlikely to be completed in a reasonable amount of time due to exponential state growth. We need to simplify the model, and careful consideration must go into finding the right balance between maximizing model correctness and minimizing model checker runtime.

The main strategy is to *bound* the state graph. The following describes a set of optimizations implemented for this example.

10.3.1 Modeling Messages as a Set

In the original Raft TLA+ Spec [8], messages are modeled as an *unordered map* to track the count of each message. It is possible for a sender to repeatedly send the same message (eg. keepalive), and grow the message count in an unbounded fashion.

messages in this example has been implemented as a set, which effectively limits the message instance count to one. It is still possible for messages to grow unboundedly because of the monotonically increasing term value. Further changes are described below.

10.3.2 Limit Term Divergence

It is possible for a node to *never* make progress. Such a case can occur when a node is partitioned off while the rest of the cluster elects a new leader and moves onto newer terms. Many of the interesting behaviors of Raft are how it addresses these cases. In a cluster of nodes with mixed terms, the nodes with older terms will eventually converge onto newer terms when they are contacted by a new leader. This converging behavior will happen whether the stale node is either 1 or N terms away from the current leader, and the former is much less costly to simulate than the latter because of the reduced number of states.

We can include *LimitDivergence* as a conjunction in *Timeout*:

$$
\begin{array}{l}
\mathit{LimitDivergence}(i) \triangleq \\
\quad \text{LET} \\
\qquad \mathit{values} \triangleq \{\mathit{term}[s] : s \in \mathit{Servers}\} \\
\qquad \mathit{max\_v} \triangleq \text{CHOOSE } x \in \mathit{values} : \forall\, y \in \mathit{values} : x \geq y \\
\qquad \mathit{min\_v} \triangleq \text{CHOOSE } x \in \mathit{values} : \forall\, y \in \mathit{values} : x \leq y \\
\quad \text{IN} \\
\qquad \lor\ \land\ \mathit{term}[i] \neq \mathit{max\_v} \\
\qquad \lor\ \land\ \mathit{term}[i] = \mathit{max\_v} \\
\qquad \land\ \mathit{term}[i] - \mathit{min\_v} < \mathit{MaxDiff}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{Timeout}(i) \triangleq & \\
\quad \land\ \mathit{LimitDivergence}(i) & \\
\quad \land\ \mathit{state}' = [\mathit{state} \text{ EXCEPT } ![i] = \text{“Candidate”}] & \\
\quad \land\ \mathit{voted\_for}' = [\mathit{voted\_for} \text{ EXCEPT } ![i] = i] & \text{voted for myself} \\
\quad \land\ \mathit{vote\_granted}' = [\mathit{vote\_granted} \text{ EXCEPT } ![i] = \{i\}] & \\
\quad \land\ \mathit{vote\_requested}' = [\mathit{vote\_requested} \text{ EXCEPT } ![i] = 0] &
\end{array}
$$

$\land\ term' = [term \text{ EXCEPT } ![i] = @ + 1]$ bump term
$\land\ establish\_leadership' = [establish\_leadership \text{ EXCEPT } ![i] = 0]$
$\land$ UNCHANGED $\langle messages \rangle$
/ PrintT(state')

10.3.3 Normalize Cluster Term

However, *term* can grow unbounded. A monotonically increasing counter is what many consensus protocols rely on to represent the latest reality. We want to *normalize* the range of terms in the cluster so the minimum value resets back to 0 to bound the state graph. This is a trick described in [9].

$Normalize \triangleq$
LET
$values \triangleq \{term[s] : s \in Servers\}$
$max\_v \triangleq \text{CHOOSE } x \in values : \forall\, y \in values : x \geq y$
$min\_v \triangleq \text{CHOOSE } x \in values : \forall\, y \in values : x \leq y$
IN
$\land\ max\_v = MaxTerm$
$\land\ term' = [s \in Servers \mapsto term[s] - min\_v]$
$\land\ messages' = \{\}$
$\land$ UNCHANGED $\langle state, voted\_for,$
$vote\_granted, vote\_requested, establish\_leadership \rangle$

$Next \triangleq$
$\lor\ \land \forall\, i \in Servers : term[i] \neq MaxTerm$
$\land\ \lor \exists\, i \in Servers :$
$\lor\ Leader(i)$
$\lor\ Candidate(i)$
$\lor\ Follower(i)$
$\lor\ \exists\, msg \in messages : Receive(msg)$
$\lor\ \land \exists\, i \in Servers : term[i] = MaxTerm$
$\land\ Normalize$

The implementation ensures only the state machine only moves forward when none of the nodes is on *MaxTerm*. If any of the nodes are on *MaxTerm*, the cluster terms are normalized.

Another caveat here is in the initial implementation I didn't update messages. This led to liveness property violation as the messages had terms disagreeing with the system state. To simplify *Spec* I simply cleared all messages. This indirectly verifies a portion of the packet loss handling in *Spec* as well.

10.3.4 Sending Request as a Batch

The send requests were initially implemented using the existential quantifier. This introduces many interleaving states. This was replaced with a universal quantifier so the set of messages is only sent once. The implementation no longer tracks if the responses were received since *Spec* should handle packet loss scenarios as well.

$$
\begin{aligned}
&\mathit{RequestVoteSet}(i) \triangleq \{\\
&\quad [\mathit{fSrc} \mapsto i,\ \mathit{fDst} \mapsto s,\ \mathit{fType} \mapsto \text{“}\mathsf{RequestVoteReq}\text{”},\ \mathit{fTerm} \mapsto \mathit{term}[i]]\\
&\qquad : s \in \mathit{Servers} \setminus \{i\}\\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Campaign}(i) \triangleq\\
&\quad \land\ \mathit{vote\_requested}[i] = 0\\
&\quad \land\ \mathit{vote\_requested}' = [\mathit{vote\_requested}\ \text{EXCEPT}\ ![i] = 1]\\
&\quad \land\ \mathit{messages}' = \mathit{messages} \cup \mathit{RequestVoteSet}(i)\\
&\quad \land\ \text{UNCHANGED}\ \langle \mathit{state},\ \mathit{term},\ \mathit{vote\_granted},\\
&\quad\ \ \mathit{voted\_for},\ \mathit{establish\_leadership} \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{KeepAliveSet}(i) \triangleq \{\\
&\quad [\mathit{fSrc} \mapsto i,\ \mathit{fDst} \mapsto s,\ \mathit{fType} \mapsto \text{“}\mathsf{AppendEntryReq}\text{”},\ \mathit{fTerm} \mapsto \mathit{term}[i]]\\
&\qquad : s \in \mathit{Servers} \setminus \{i\}\\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Leader}(i) \triangleq\\
&\quad \land\ \mathit{state}[i] = \text{“}\mathsf{Leader}\text{”}\\
&\quad \land\ \mathit{establish\_leadership}[i] = 0\\
&\quad \land\ \mathit{establish\_leadership}' = [\mathit{establish\_leadership}\ \text{EXCEPT}\ ![i] = 1]\\
&\quad \land\ \mathit{messages}' = \mathit{messages} \cup \mathit{KeepAliveSet}(i)\\
&\quad \land\ \text{UNCHANGED}\ \langle \mathit{state},\ \mathit{voted\_for},\ \mathit{term},\ \mathit{vote\_granted},\ \mathit{vote\_requested} \rangle
\end{aligned}
$$

10.3.5 Prune Messages with Stale Terms

When a node's term advances, all messages targeted to this node with older terms are discarded. Keeping messages with stale terms allows the model checker to verify the node correctly discards them, but can exponentially grow the state machine. To simplify the model, we can prune stale messages as we add a new message:

$$
\begin{array}{l}
AddMessage(to\_add, msgs) \triangleq \\
\quad \text{LET} \\
\qquad pruned \triangleq \{msg \in msgs : \\
\qquad\; \neg(msg.fDst = to\_add.fDst \land msg.fTerm < to\_add.fTerm)\} \\
\quad \text{IN} \\
\qquad pruned \cup \{to\_add\}
\end{array}
$$

$$RemoveMessage(to\_remove, msgs) \triangleq$$

10.3.6 Enable Symmetry

Since the behavior is symmetric between nodes, we can enable symmetry to speed up model checker runtime:

$$Perms \triangleq Permutations(Servers)$$

10.4 Safety

One of the goals of the protocol is to ensure the cluster only has one leader. The clusters can have multiple leaders due to unfavorable network connections. For example, a leader node is partitioned off and a new leader is elected. However, even when the cluster has multiple leaders, they *must* be on different terms. The leader with the highest term is effectively the *true leader*. This invariant can be implemented like so:

$$
\begin{array}{l}
LeaderUniqueTerm \triangleq \\
\quad \forall s1, s2 \in Servers : \\
\quad\; (\land state[s1] = \text{“Leader”} \\
\qquad \land state[s2] = \text{“Leader”}
\end{array}
$$

$$
\begin{array}{l}
\quad \land\, s1 \neq s2) \\
\quad \Rightarrow (term[s1] \neq term[s2])
\end{array}
$$

For every pair of nodes, they cannot both be Leaders and have the same term.

10.5 Liveness

In any failure recovery scenario, the nodes in the cluster converge to a higher term value either voluntarily or involuntarily. For example:

- A node timed out and started a new election on a new term
- A partitioned follower receives a heartbeat from a new leader on a new term
- A candidate receiving a request vote from another candidate on a higher term

In any case, a node's term number always increases. This can be described as below:

$$
\begin{array}{l}
Converge \triangleq \\
\quad \forall\, s \in Servers : \\
\quad\; term[s] = 0 \leadsto term[s] = MaxTerm - MaxDiff
\end{array}
$$

Instead of *MaxTerm*, we use *MaxTerm-MaxDiff* to ensure the liveness property is always upheld even after *Normalization*. However, running *Spec* against TLC now will encounter a set of stuttering issues. We also need to update the fairness description to ensure all possible actions are called when the enabling conditions are *eventually always* true:

$$
\begin{array}{l}
Liveness \triangleq \\
\quad \land\, \forall\, i \in Servers : \\
\quad\;\; \land\, \mathrm{WF}_{vars}(Leader(i)) \\
\quad\;\; \land\, \mathrm{WF}_{vars}(Candidate(i)) \\
\quad\;\; \land\, \mathrm{WF}_{vars}(Follower(i)) \\
\quad \land\, \mathrm{WF}_{vars}(\exists\, msg \in messages : Receive(msg))
\end{array}
$$

Part III

Examples with PlusCal

PlusCal is a C-like syntax that allows designer to describe their *Spec* in a more programming language like fashion. I'm of the opinion that these are suitable to describe concurrent algorithm, where the code execution between multiple contexts may interleave in any way imaginable. While it certainly is possible to express these in TLA+ directly, I find it to be error prone, somewhat comparable to writing in Assembly instead of C. In this section we will describe a few PlusCal example.

Chapter 11

SPSC Lockless Queue

A single producer single consumer (SPSC) lockless queue is a data exchange queue between a producer and a consumer. The SPSC lockless queue enables data exchange between producer and consumer without the use of a lock, allowing both producer and consumer to make progress in all scenarios.

An example application of an SPSC queue is a data exchange interface between the ASIC and the CPU in a driver implementation.

A real implementation needs to account for memory ordering effects specific to the architecture. For example, ARM has a weak memory ordering model where read/write may appear out-of-order between CPUs. In this chapter, we will assume *logical* execution order where each command is perceived as issued sequentially (even across CPUs) to focus the discussion on describing the system using TLA+.

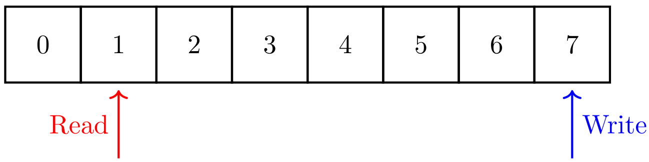
11.1 Design

The following describes the SPSC queue requirements:

- Two executing context, reader and writer
- Writer pointer advances after write

- Reader pointer advances after reading
- If read pointer equals write pointer, queue is empty
- If writer pointer + 1 equals read pointer, queue is full

The following is an example of an SPSC queue:



Since the reader and writer execute in different contexts, the instructions in a read and write can interleave in *any* way imaginable:

- Reader empty check can happen just as the writer is writing data
- Writer full check can happen just as the reader is reading data
- Reading and writing can occur concurrently

The key observation is the index held by the write pointer is reserved by the writer. Similarly, the index held by the read pointer is reserved by the reader. The only exception is when the read pointer equals to write pointer, then the queue is empty. Given the possible ways the reader and writer execution can interleave, we can use TLA+ to verify the design.

11.2 Spec

TLA+ specification can be written using its native formal specification language, or a C-like syntax called PlusCal (which transpires down to its native form). In this example, I chose to implement the specification using PlusCal, since the content to be verified is pseudo implementation. While it is possible to specify SPSC in native TLA+, I find the approach more error-prone as each line is effectively an individual state to be modeled.

The following is a snippet of the spec written in PlusCal:

```
procedure reader()
begin
r_chk_empty:
    if rptr = wptr then
    r_early_ret:
        return;
    end if ;
r_read_buf:
    assert buffer[rptr] ≠ 0;
r_cs:
    buffer[rptr] := 0;
r_upd_rtpr:
    rptr := (rptr + 1)%N ;
    return;
end procedure ;
```

The reader checks if the queue is empty by comparing the read and write pointers. If the queue is empty, reader early returns. If the queue is not empty, the reader reads the index and advances the read pointer.

```
procedure writer()
begin
w_chk_full:
    if (wptr + 1)%N = rptr then
    w_early_ret:
        return;
    end if ;
w_write_buf:
    assert buffer[wptr] = 0;
w_cs:
    buffer[wptr] := wptr + 1000;
w_upd_wptr:
    wptr := (wptr + 1)%N ;
    return;
end procedure ;
```

The writer checks if the queue is full checking if there's more space to write to. If the queue is full, the writer early exists. If the queue is not full, a writer writes to the queue and advances the write pointer.

The key insight here is the read and write pointer effectively *reserves* the index they are pointing to. The state of the indices is unknown to the other context. Assume a reader index of k, the writer cannot write to index k since the reader might be reading from it. The only time a writer can write to k is when the read index is no longer k, suggesting the reader is done with the index. Symmetrically reasoning applies with the write index.

11.3 Safety

A correctness property for a lockless algorithm is to ensure reader and writer cannot access the same index at the same time. Both reader and writer can be working inside their critical section, but they *cannot* be working on the same index. The following formula describes this safety property:

$$
\begin{aligned}
&\mathit{MutualExclusion} \triangleq \\
&\qquad \neg((pc[\mathit{WRITER}] = \text{“}\mathsf{w\_cs}\text{”}) \land (pc[\mathit{READER}] = \text{“}\mathsf{r\_cs}\text{”}) \land \mathit{rptr} = \mathit{wptr})
\end{aligned}
$$

11.4 Liveness

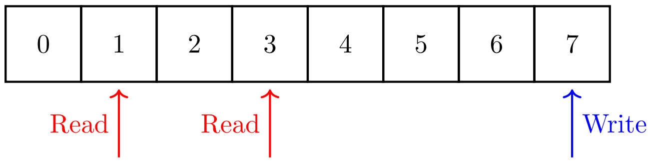
For liveness, we want to check the queue never hangs. This can be described as all indices are eventually used and unused:

$$
\begin{aligned}
&\mathit{Liveness} \triangleq \\
&\qquad \land \forall\, k \in 0 \,..\, N - 1 : \\
&\qquad \mathit{buffer}[k] \neq 0 \leadsto \mathit{buffer}[k] = 0 \\
&\qquad \land \forall\, k \in 0 \,..\, N - 1 : \\
&\qquad \mathit{buffer}[k] = 0 \leadsto \mathit{buffer}[k] \neq 0
\end{aligned}
$$

Chapter 12

SPMC Lockless Queue

As the name suggests, an SPMC lockless queue supports a single producer *multiple* consumer usage topology.



An SPMC queue can be tricky to get right. There are many things to consider:

- Readers can lapse each other.
- Readers can compete for reading indices.
- Readers can lapse writer.
- One reader can starve other readers.
- A slow reader can block the system.

And under all circumstances, system *correctness* must be maintained.

12.1 Design

For the design:

- SPMC is implemented as a circular queue with the size of N
- The status of the individual index is represented as a status array of size N
- The status of each index is either *UNUSED*, *WRITTEN*, or *READING*
- Each reader maintains its own read pointer
- A *outstanding* counter is incremented by the writer when the write is complete, and decremented by the reader when it reserves a read

Whenever the write finishes a write, it increments *outstanding* to indicate some buffer is ready to read.

To read, a reader performs a two-step reservation:

- The reader decrements *outstanding*. A successful decrement means the reader is *guaranteed* a read index.
- After successful decrement of *outstanding*, the reader walks its read pointer until it successfully reserves the next available index to read. This is done by attempting to CAS update an index from *WRITTEN* to *READING*. If the update fails, then the index was already reserved by another reader.

There may be more than one approach to implementing SPMC, the above description is what we will implement in this chapter.

12.2 Spec

The following is the core reader implementation:

procedure *reader*( )
variable

```
    i = self ;
begin
r_chk_empty:
    if outstanding ≠ 0 then
     outstanding := outstanding − 1 ;
    else
    r_early_ret:
        return ;
    end if ;
r_try_lock:
    if status[rptr[i]] = WRITTEN then
     status[rptr[i]] := READING ;
    else
    r_retry:
        rptr[i] := (rptr[i] + 1)%N ;
        goto r_try_lock ;
    end if ;
r_data_chk:
    assert buffer[rptr[i]] = rptr[i] + 1000 ;
r_read_buf:
    buffer[rptr[i]] := 0 ;
r_unlock:
    status[rptr[i]] := UNUSED ;
r_done:
    return ;
end procedure ;
```

The reader performs a non-zero check on outstanding. If the outstanding is zero, the queue is empty, and the reader early returns.

If outstanding is K, then at most K readers can reserve an index to read. If the system has M readers, then M-K readers will fail to reserve a read index. The readers now compete to reserve a read. More specifically:

- Reader loads outstanding, and stores that onto the local variable counter.
- Reader early returns if the counter is zero

- Reader attempts to update outstanding with CAS using counter and counter - 1.
- If CAS fails, go back to the top and retry.

If non-success is returned, another reader has *won* the reservation. The current reader can attempt to reserve again if *outstanding* is non-zero.

If success is returned, the reader is *guaranteed* a read. However, readers may still compete during index reservation. To reserve an index, a reader issues CAS to update the index status from *WRITTEN* to *READING*. CAS failure indicates another reader has already reserved this index. The reader will bump the read pointer and try to reserve the next index.

Now let us take a look at the writer implementation:

procedure *writer*( ) **begin**
w_chk_full:
  if $outstanding = N - 1$ **then**
  w_early_ret:
    return ;
  end if ;
w_chk_st:
  if $status[wptr] \neq UNUSED$ **then**
  w_early_ret2:
    return ;
  end if ;
w_write_buf:
  $buffer[wptr] := wptr + 1000$ **;**
w_mark_written:
  $status[wptr] := WRITTEN$ **;**
w_inc_wptr:
  $wptr := (wptr + 1)\%N$ **;**
w_inc:
  $outstanding := outstanding + 1$ **;**
w_done:

return ;
end procedure ;

The writer first checks outstanding, and early returns if the queue is full. After the fullness check, the writer then checks if the current index is UNUSED. This is to account for the edge case where a reader has performed the reservation first step to decrement outstanding but hasn't done the actual read. If both checks pass, then the writer now has an *UNUSED* index it can write to.

12.3 Safety

When a reader reserves an index to read, the reader must have exclusive access. This can be described as: For any pair of readers inside the critical section, they must operate on different indices:

$$
\begin{aligned}
&\mathit{ExclusiveReservation} \triangleq \\
&\quad \forall\, x,\, y \in \mathit{READERS} : \\
&\quad (\wedge\, x \neq y \\
&\quad\ \ \wedge\, pc[x] = \text{“r\_read\_buf”} \\
&\quad\ \ \wedge\, pc[y] = \text{“r\_read\_buf”}) \\
&\quad\ \ \Rightarrow (\mathit{rptr}[x] \neq \mathit{rptr}[y])
\end{aligned}
$$

Similarly, for any reader and writer inside the critical section, must operate on different indices as well:

$$
\begin{aligned}
&\mathit{ExclusiveReadWrite} \triangleq \\
&\quad \forall\, x \in \mathit{READERS} : \\
&\quad\ \ (\wedge\, pc[x] = \text{“r\_read\_buf”} \\
&\quad\ \ \ \ \wedge\, pc[\mathit{WRITER}] = \text{“w\_write\_buf”}) \\
&\quad\ \ \ \ \Rightarrow (\mathit{rptr}[x] \neq \mathit{wptr})
\end{aligned}
$$

12.4 Liveness

All indices must be used as the system runs. The following verifies all unused indices are eventually used, and all used indices are eventually unused:

$$
\begin{array}{l}
Liveness \triangleq \\
\quad \land \forall k \in 0 \,..\, N-1 : \\
\quad\; buffer[k] = 0 \leadsto buffer[k] \neq 0 \\
\quad \land \forall k \in 0 \,..\, N-1 : \\
\quad\; buffer[k] \neq 0 \leadsto buffer[k] = 0
\end{array}
$$

The following describes a more subtle scenario. We need to ensure the system remains functional even if readers complete out-of-order. The following describes such a scenario, where two non-contiguous indices have been reserved for reading. In this case, we expect the indices to eventually be re-used. This means the the system remains functional after such scenario.

$$
\begin{array}{l}
Liveness2 \triangleq \\
\quad \forall k \in 0 \,..\, N-3 : \\
\quad \land (\land status[k] = READING \\
\quad\;\; \land status[k+1] = UNUSED \\
\quad\;\; \land status[k+2] = READING) \\
\quad\;\; \leadsto (status[k] = WRITTEN) \\
\quad \land (\land status[k] = READING \\
\quad\;\; \land status[k+1] = UNUSED \\
\quad\;\; \land status[k+2] = READING) \\
\quad\;\; \leadsto (status[k+2] = WRITTEN)
\end{array}
$$

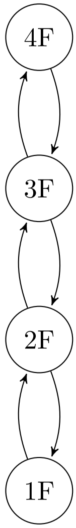
Part IV

Reference

Chapter 13

Fairness

For rigorous definition and proof, please refer to [1]. This chapter focus on the application fairness by describing an elevator that eventually makes it to the top floor:



13.1 Liveness

Consider the following elevator *Spec*:

$$
\begin{array}{l}
\text{MODULE } elevator \\
\text{EXTENDS } Integers \\
\text{VARIABLES } a \\
vars \triangleq \langle a \rangle \\
TOP \triangleq 4 \\
BOTTOM \triangleq 1 \\
Init \triangleq \\
\quad \land\ a = BOTTOM \\
Up \triangleq \\
\quad \land\ a \neq TOP \\
\quad \land\ a' = a + 1 \\
Down \triangleq \\
\quad \land\ a \neq BOTTOM \\
\quad \land\ a' = a - 1 \\
Spec \triangleq \\
\quad \land\ Init \\
\quad \land\ \Box[Up \lor Down]_a
\end{array}
$$

The building has a set of floors and the elevator can go either up or down. The elevator keeps going up until it's the top floor, or keeps going down until it's the bottom floor. TLC will pass the spec as is.

Let's introduce a liveness property. The elevator should always at least go to the second floor:

$$
\begin{array}{l}
Liveness \triangleq \\
\quad \land\ a = 1 \leadsto a = 2
\end{array}
$$

The model checker will report a violation on this property:

```
Error: Temporal properties were violated.
Error: The following behavior constitutes a counter-example:
State 1: <Initial predicate>
a = 1
```

```
State 2: Stuttering
```

Since *Spec* permits *stuttering*, the state machine is allowed to perpetually stay on 1F and *never* go to 2F. This can be fixed by introducing a fairness description.

13.2 Weak Fairness

Weak fairness is defined as:

$$\Diamond\Box(ENABLED\langle A\rangle_v) \Rightarrow \Box\Diamond\langle A\rangle_v \tag{13.1}$$

$ENABLED\langle A\rangle$ represents *conditions required* for action A. The above translates to: if conditions required for action A to occur is *eventually always* true, then action A will *always eventually* happen.

Without weak fairness defined, the elevator may *stutter* at 1F and never go to 2F. Weak fairness states that if the conditions of an action are*eventually always* true (ie. elevator decides to stay on 1F but *can* go up), the elevator *always eventually* goes up.

$Spec \triangleq$
$\quad \land\ Init$
$\quad \land\ \Box[Down \lor Up]_a$
$\quad \land\ \mathrm{WF}_a(Down)$
$\quad \land\ \mathrm{WF}_a(Up)$

Running the spec against the model checker passes again. What if we want to verify the elevator eventually always goes to the top, not just to 2F. Let's modify the Liveness property again:

$Liveness \triangleq$
$\qquad \land\ a = BOTTOM \leadsto a = TOP$

The model checker now reports the following violation:

```
Error: Temporal properties were violated.
Error: The following behavior constitutes a counter-example:
```

```
State 1: <Initial predicate>
a = 1
State 2: <Up line 10, col 5 to line 11, col 17 of
  module elevator>
a = 2
Back to state 1: <Down line 13, col 5 to line 14,
  col 17 of module elevator>
```

The model checker identified a case where the elevator is perpetually stuck going between 1F and 2F but never goes to 3F. Weak fairness is no longer enough, because the the elevator is not stuck on 2F repeatedly, but stuck going *between* 1F and 2F. This is where we need strong fairness.

13.3 Strong Fairness

Strong fairness is defined as:

$$\Box\Diamond(\mathit{ENABLED}\langle A\rangle_v) \Rightarrow \Box\Diamond\langle A\rangle_v \tag{13.2}$$

The difference between weak and strong fairness is the *eventually always* vs. *always eventually.*

In weak fairness, once the state machine is stuck in a state forever, the state machine always transitions to a possible next state permitted by the *Spec* (eg. if the elevator is stuck on 1F but can go to 2F, it will). With strong fairness, the elevator doesn't need to be stuck on 2F to go to 3F. If the elevator *always eventually* makes it to 2F, it *always eventually* go to 3F.

Intuitively we are tempted to enable strong fairness like so:

$$\begin{aligned}
&\mathit{Spec} \triangleq \\
&\quad \land \mathit{Init} \\
&\quad \land \Box[\mathit{Up} \lor \mathit{Down}]_a \\
&\quad \land \mathrm{WF}_a(\mathit{Down}) \\
&\quad \land \mathrm{SF}_a(\mathit{UP})
\end{aligned}$$

However, model checker *still* reports the same violation.

If we take a closer look at the enabling condition for *Up*, it only requires a current floor to be not the *top floor*. When the elevator is stuck in a loop going Up and Down between 1F and 2F indefinitely, strong fairness for Up is *already satisfied*. What we want is strong fairness on *Up* for *every floor*, instead of *any floor except top floor*. So if the elevator makes it to 2F once, it will *always eventaully* go to 3F. If the elevator makes it to 3F once, it will *always eventaully* go to 4F, and so on. The following is the change required:

$$
\begin{aligned}
&\mathit{Spec} \triangleq \\
&\quad \land\ \mathit{Init} \\
&\quad \land\ \Box[\mathit{Up} \lor \mathit{Down}]_a \\
&\quad \land\ \mathrm{WF}_a(\mathit{Down}) \\
&\quad \land\ \forall f \in \mathit{BOTTOM} \,..\, \mathit{TOP} - 1 : \\
&\qquad \land\ \mathrm{WF}_a(\mathit{Up} \land f = a)
\end{aligned}
$$

With this change, the model checker will pass.

Chapter 14

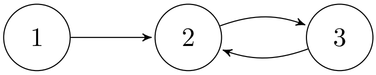
Liveness

While safety properties can catch per-state contradictions, liveness properties allow you to verify the behavior across a series of states. This is TLA+'s *superpower*. Designers are rarely interested only in the correctness of individual states in the system, but also the correctness of system *across* a set of states.

This book has provided a few examples of liveness properties so far: eg. The elevator eventually makes it to the top floor, consensus protocol eventually converges, the scheduling algorithm guarantees a lock requester eventually gets the lock, etc. I argue any system worth the reader's time to model using TLA+ must have interesting liveness properties to verify.

Unfortunately, liveness check also takes *much* longer, since the very definition of verifying property across a series of states makes the task very hard to parallelize. Care must go into refining the model to keep the model checker runtime reasonable. In this chapter, we will discuss a simple state machine example to illustrates liveness properties.

Assume a simple three-system system:



This can be described by the following spec:

MODULE *liveness*

EXTENDS *Naturals*

VARIABLES *counter*

$$vars \triangleq \langle counter \rangle$$

$$EventuallyAlways \triangleq \Diamond\Box(counter = 3)$$
$$AlwaysEventually \triangleq \Box\Diamond(counter = 3)$$

$$Init \triangleq$$
$$\land\ counter = 0$$

$$Inc \triangleq$$
$$\land\ counter' = counter + 1$$

$$Dec \triangleq$$
$$\land\ counter' = counter - 1$$

$$Next \triangleq$$
$$\lor\ \land\ counter \neq 3$$
$$\land\ Inc$$
$$\lor\ \land\ counter = 3$$
$$\land\ Dec$$

$$Spec \triangleq$$
$$\land\ Init$$
$$\land\ \Box[Next]_{vars}$$
$$\land\ \mathrm{WF}_{vars}(Next)$$

Note the fairness description in the spec. Without fairness the spec is allowed to stutter and fail any liveness property checks.

14.1 Always Eventually

We want to verify the system *always eventually* transition state 3. This can be described by the following liveness property:

$$AlwaysEventually \triangleq \Box\Diamond(counter = 3)$$

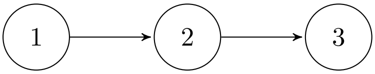
Once the system makes it to state 3, the system is stuck in a loop transitioning states 2 and 3. It doesn't *remain* in state 3, but it does *always eventually* transition to state 3. The system as described fulfills this liveness property.

14.2 Eventually Always

However, the system does not *eventually always* remain in state 3, because the system toggles between state 2 and 3. The liveness property to check the system *eventually always* transition to and remain state 3 is shown below:

$$EventuallyAlways \triangleq \Diamond\Box(counter = 3)$$

To satisfy this liveness property, we will need to *remove* the the transition from 3 to 2, which updates the state diagram like below:



We need to remove the corresponding *Dec* action from Next:

$$\begin{aligned} &Next \triangleq \\ &\quad \land\ counter \neq 3 \\ &\quad \land\ Inc \end{aligned}$$

The system now *eventually always* remains in state 3, satisfying the liveness property.

Note that the system still *always eventually* makes it to state 3, so the updated spec satisfies both *AlwaysEventually* and *EventuallyAlways* liveness properties. This is not to say the designer should always use *eventually always.* Some system may *never* converge onto a fixed state. For example, in a consensus system, any given server may crash and disturb the converged state. In such case *eventually always* will never be true, but *always eventually* can be true.

14.3 Leads To

Leads to provides a *cause-and-effect* description. In this example, we can describe state 0 *leads to* state 3:

state 0 leads to state 3: TRUE

$$LeadsTo \triangleq counter = 0 \leadsto counter = 3$$

state 0 leads to state 4: FALSE - model checker reports a violation

$$LeadsTo \triangleq counter = 0 \leadsto counter = 4$$

Note that *leads to* is only evaluated if the left-hand side is *true.* If the right-hand side is updated to counter = 4, the liveness the property will fail as expected. However, if the left-hand side is false, then the liveness property is not evaluated since there isn't a state that satisfies the cause condition. For example, the model checker will not report violations for the following liveness property:

model checker will NOT report violation because cause condition never occur

$$LeadsTo \triangleq counter = 4 \leadsto counter = 3$$

Chapter 15

General Guideline

15.1 Debug

Debugging in TLC is a bit different than debugging with normal programs. A step in the model checker is a state transition. Even if the model checker completes, it's still worthwhile to dump and audit the states just to make sure *Spec* is defined correctly.

```
tlc elevator -dump out > /dev/null && cat out.dump | head -n5
State 1:
a = 1
State 2:
a = 2
```

You may want to grep the output to look for the state being set to expected value to confirm your spec is working as intended.

15.2 Dead Lock

Deadlock typically happens when the model checker runs out of things to do. This can be a result of an incomplete *Spec* definition, where certain edge cases were not accounted for. The model checker typically provides a fairly comprehensive backtrace leading up to the deadlock to simplify debugging.

15.3 Live Lock

Livelock happens when the model checker identifies a case where the liveness property is violated. An example is the elevator stuck going between two floors instead of going to the top floor, or the system is stuck dropping and retransmitting the same packet.

These are typically fixed by providing additional fairness descriptions to the Spec, telling the model checker how to continue in the case of a live lock.

For a detailed fairness description please refer to Chapter 13.

15.4 Model Reduction

This is the *art* of enabling a TLA+ spec to be verifiable. Model checking is only valuable if it can be verified within a reasonable amount of time. Since the model complexity grows exponentially, there's little value in attempting to hyper-optimize the details. Designers should focus on simplifying the model by removing non-critical features and focus on features with highest return on investment.

One useful way of trimming out the low-value portion of a spec is to audit the state dump. Even in the case of a non-terminating run, a partial state dump may help identify low-value abstractions that can be removed from the spec.

One key value of TLA+ is it highlights all the corner cases in the system. Even if the designer ends up simplifying the spec, it still likely highlights certain conditions the designer was previously unaware of.

In general, when the spec has millions or higher more states, it likely cannot be verified within a few seconds. If a fault is caught after verifying a million states, the model likely can be simplified to reproduce the fault in much less states.

Chapter 16

Data Structure

Like other languages, TLA+ provides its data structure. Readers are assumed familiar with common data structures, and this chapter will only focus on the TLA+ language semantics.

16.1 Set

Set is an unordered set where every element in the set is unique. TLA+ Set includes common set operation including union, intersection, membership check, and more.

$a \triangleq \{0, 1, 2\}$
$b \triangleq \{2, 3, 4\}$

{0, 1, 2, 3, 4}
$c \triangleq a \cup b$

{2}
$d \triangleq a \cap b$

TRUE - because 4 in c is bigger than 3
$e \triangleq \exists x \in c : x > 3$

FALSE - nothing in c is bigger than 5
$f \triangleq \exists x \in c : x > 5$

FALSE - not all elements in c are smaller than 3
$g \triangleq \forall x \in c : x < 3$

TRUE - all elements in c are smaller than 3
$h \triangleq \forall x \in c : x < 5$

{0, 1, 2} - all elementse less than 3
$i \triangleq \{x \in c : x < 3\}$

5 - the number of elements in c
$j \triangleq Cardinality(c)$

{0, 1, 3, 4} - c substracts d
$k \triangleq c \setminus d$

16.2 Tuple

A tuple is an ordered data structure, similar to a queue in other languages. Common operation supported by tuple include Append to push and Tail to pop.

$a \triangleq \langle 0, 1, 2\rangle$
$b \triangleq \langle 2, 3, 4\rangle$

tuple: 0, 1, 2, 2, 3, 4
$c \triangleq A \circ B$

tuple: 0, 1, 2, 3
$c2 \triangleq Append(A, 3)$

gets head: 0
$c3 \triangleq Head(A)$

removes head: 1, 2
$c4 \triangleq Tail(A)$

6
$d \triangleq Len(c)$

TRUE - every c[x] is not 10

First tuple element is at index 1 (not 0)
$e \triangleq \forall x \in 1 \, .. \, Len(c) : c[x] \neq 10$

TRUE - there exists a c[x] that is 2
$f \triangleq \exists x \in 1 \, .. \, Len(c) : c[x] = 2$

{3, 4} - when index is 3 or 4, c[x] = 2
$g \triangleq \{x \in 1 \, .. \, Len(c) : c[x] = 2\}$

16.3 Function

Function is similar to map in other data structures, supporting key value lookup.

$SetA \triangleq \{\text{“a”}, \text{“b”}, \text{“c”}\}$
$SetB \triangleq \{\text{“c”}, \text{“d”}, \text{“e”}\}$

Create a mapping with keys a, b, c with values 0, 0, 0
$a \triangleq [k \in SetA \mapsto 0]$
$b \triangleq [k \in SetB \mapsto 1]$

Concatenate
$c \triangleq a @@ b$

Subtraction
$d \triangleq [x \in (\text{DOMAIN } c \setminus \text{DOMAIN } b) \mapsto c[x]]$

Create a mapping with keys a, b, c with values {}, {}, {}
$e \triangleq [k \in SetA \mapsto \{\}]$

Create a mapping that is the same as e, except key a's value is "a", "b", "c"
$f \triangleq [e \text{ EXCEPT } ![\text{“a”}] = \{\text{“a”}, \text{“b”}, \text{“c”}\}]$

Chapter 17

Reference

Bibliography

[1] Specifying Systems, https://lamport.azurewebsites.net/tla/book.html

[2] Verifying Hyperproperties with TLA, https://lamport.azurewebsites.net/pubs/hyper2.pdf

[3] TLA Toolbox, https://github.com/tlaplus/tlaplus

[4] TLA+ Community Modules, https://github.com/tlaplus/CommunityModules

[5] Fairness in TLA+, https://sriku.org/posts/fairness-in-tlaplus/

[6] Backblaze Durability Calculates at 99.999999999% — And Why It Doesn't Matter, https://www.backblaze.com/blog/cloud-storage-durability/

[7] In Search of an Understandable Consensus Algorithm, https://raft.github.io/raft.pdf

[8] raft.tla, https://github.com/ongardie/raft.tla

[9] Wrangling monotonic systems in TLA+, https://ahelwer.ca/post/2023-11-01-tla-finite-monotonic/

[10] C10k problem, https://en.wikipedia.org/wiki/C10k_problem

[11] Dining Philosophers, https://en.wikipedia.org/wiki/Dining_philosophers_problem

[12] A GPU Algorithm for Detecting Strongly Connected Components, https://userweb.cs.txstate.edu/ mb92/papers/sc23c.pdf

[13] Keynote Fifteen years of formal methods at AWS, https://www.youtube.com/watch?v=HxP4wi4DhA0