Correct by Design with TLA+
Work In Progress

Richard Tang

January 31, 2025

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 of (possibly hundreds) 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 power island has been shut off, the physical hardware debug port is also unavailable, leaving the developer with *no way* of live debugging-related problems. At this point, the developer needs to siphon through (possibly hundreds) of instructions to catch invariant violation *manually*.

If the microcode attempts to access the memory interface when the power island has been shut off, the processor will hang. Since the power island has been shut off, the physical hardware debug port is also unavailable, leaving the developer with no way to debug related problems. At this point, the developer needs to manually inspect (possibly hundreds) of instructions to catch invariant violations.

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 an emulator for this proprietary processor to verify the microcode before deploying it on target. The emulator 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 arbitrarysystem 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.

The Turing Award winner Leslie Lamport invented the TLA+ in 1999, but TLA+ didn't appear to have caught on until the 2010s. My opinion is that TLA+ was invented ahead of its time, and the problem complexity finally caught up in the past decade or so, to allow TLA+ to demonstrate its strength.

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 many low-end hardware instead of a few 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 and 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 critical tools 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 computing

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 and 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 state.

For every specification, the designer can specify *safety* property (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* property. These are properties to be satisfied by a *sequence of state*. 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 if one hasn't seen it before but like any other programming language an 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)

2.3 Spec

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

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

$\triangleq$ 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 variable in its next state. The *Next* action for the digital clock can be defined as:

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

Here's a breakdown of what the spec does:

- *Next* can take either *NextHour* or *NextMinute*
- *Next* takes *NextMinute* when *minute* is not 59. *NextMinute* doesn't update *Hour* and increments *Minute*.
- *Next* takes *NextHour* when *minute* is 59. *NextHour* increments *hour* modulus 24 and sets *minute* to 0.

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:

$$
\begin{array}{l}
vars \triangleq \langle hour,\ minute \rangle \\
Spec \triangleq \\
\quad \land\ Init \\
\quad \land\ \Box[Next]_{vars}
\end{array}
$$

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.
- $\Box$ operator asserts the formula is always true for every step in the behavior.
- And steps in the behaviour is defined as $[Next]_{vars}$, where *Next* describe 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

Safety property describes 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:

$$
\begin{array}{l}
Type\_OK \triangleq \\
\quad \land\ hour \in 0\ldots 23
\end{array}
$$

$\land\ minute \in 0\,..\,59$

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:

$$
\begin{array}{l}
Liveness \triangleq \\
\quad \land\ hour = 23 \land minute = 59 \leadsto hour = 0 \land minute = 0
\end{array}
$$

$\leadsto$ 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.

$$
\begin{array}{l}
Spec \triangleq \\
\quad \land\ Init \\
\quad \land\ \Box[Next]_{vars} \\
\quad \land\ \mathrm{WF}_{vars}(Next)
\end{array}
$$

$WF_{vars}(Next)$ 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 [7].

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*
$vars \triangleq \langle hour, minute \rangle$
$Type\_OK \triangleq$
  $\land\ hour \in 0 \,..\, 23$
  $\land\ minute \in 0 \,..\, 59$
$Liveness \triangleq$
  $\land\ hour = 23 \land minute = 59 \rightsquigarrow hour = 0 \land minute = 0$
$Init \triangleq$
  $\land\ hour = 0$
  $\land\ minute = 0$
$NextMinute \triangleq$
  $\land\ minute = 59$
  $\land\ hour' = (hour + 1)\%24$
  $\land\ minute' = 0$
$NextHour \triangleq$
  $\land\ minute \neq 59$
  $\land\ hour' = hour$
  $\land\ minute' = minute + 1$
$Next \triangleq$
  $\lor\ NextMinute$
  $\lor\ NextHour$
$Spec \triangleq$
  $\land\ Init$
  $\land\ \Box[Next]_{vars}$
  $\land\ \mathrm{WF}_{vars}(Next)$

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.
  Estimates of the probability that TLC did not check all reachable states
```

```
  because two distinct states had the same fingerprint:
  calculated (optimistic):  val = 7.8E-17
1441 states generated, 1440 distinct states found, 0 states left on queue.
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+

Chapter 3

Simple Gossip Protocol

In a distributed system, a cluster of nodes collectively provides a service. A distributed system may have 10s to 100s of nodes working together to offer the service in a geo-diverse environment to maximize uptime. The nodes often have requirements to know about each other. In the context of a distributed database, a node 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 nodes to fetch the latest cluster information in a distributed fashion. Before the gossip protocol, nodes 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 nodes to initiate the data exchange, and the nodes in the cluster periodically select a set of neighbors to gossip with.

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

$$propagation\_time = \log_k N * gossip\_interval \tag{3.1}$$

With the total number of messages exchanged:

$$messages\_exchanged = \log_k N * k \tag{3.2}$$

3.1 Design

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

- Each node has a version.
- Each node caches the version of all other nodes.

- A pair of nodes are randomly selected to gossip
- A node can restart. Restarting a node clears the node's version cache of the other nodes.
- A node can bump its own version.

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

3.2 Spec

In gossip protocol, every node needs to remember all its peers current version:

```
Init ≜
    ∧ version = [i ∈ Servers ↦ [j ∈ Servers ↦ 0]]
Next ≜
    ∨ ∃ i ∈ Servers :
      ∧ Bump(i)
    ∨ ∃ i, j ∈ Servers :
      ∧ Gossip(i, j)
    ∨ ∃ i ∈ Servers :
      ∧ Restart(i)
```

The *Init* formula simply declares version to be a two dimension array with all elements initialized to 0. *Next* allows either bumping version of a server, pick a pair of nodes to gossip, or restart a server.

The following provides definition to these steps:

```
Gossip(i, j) ≜
    LET
        Max(a, b) ≜ IF a > b THEN a ELSE b
        updated ≜ [k ∈ Servers ↦ Max(version[i][k], version[j][k])]
        version_a ≜ [version EXCEPT ![i] = updated]
        version_ab ≜ [version_a EXCEPT ![j] = updated]
    IN
        ∧ version' = version_ab
```

When two server gossip, they gossip about all the nodes (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) ≜
```

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

The action only permits version bump if the Server hasn't made it to *MaxVersion*. When the Server bumps the version, it only bumps its own version and keeps all other version in its version cache as is.

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

Upon *Restart*, a server reloads from its local storage (so it's own version persist), but the server needs to re-learn the cluster status (all other entries in its version cache is wiped).

The *Spec* defines three actions: *Bump*, *Restart*, *Gossip*. Without any fairness description, *Any* permutation of these actions are allowed by the Spec, and *will* be checked by the model checker:

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

We will discuss how to specify fairness in the Liveness section of this chapter.

3.3 Safety

3.4 Liveness

The expected behaviour of a system using gossip protocol is to ensure the cluster converges towards higher version number for all servers even in the presence of failure (represented by *Restart*). This means we need to gaurantee *Bump* is always being called. Without this gaurantee, the Spec can trap in a *Restart* and *Gossip* loop. To ensure *Bump* is always called, we need to add fairness description:

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

The the model checker explores all possible transitions permitted by the Spec. This includes calling Restart repeatedly, calling Gossip repeatedly, calling any subset of the actions repeatedly. The fairness description gaurantees that if the enabling condition of an action is true, the action will take. 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 the Spec ensuring the system always mgirate torwards higher version number, we can now define the Liveness property:

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

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

Since the 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, refer to the last section of the book.

Chapter 4

Simple Scheduler

Task schedulers are fairly 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 for security reasons. Context switching process requires the OS to "clean" the hardware before running the new process.

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 sort of scheduler to manage the tasks.

To solve the C10k [9] problem, some languages (eg. Rust) support asynchronous programming, allowing the user to enable task switching *within* in the same process to scale up system throughput. However, language like Rust only provides the *language support* for asynchronous programming, and user must supply their async runtime. The async runtime must also include a scheduler to manage the tasks.

Hopefully, this provides enough context as to why implementing a scheduler may be of interest. In this chapter, we will implement a very simple cooperative scheduler with tasks that share a single lock.

4.1 Design

In this section we will define a spec for a simple task scheduler. The task sechdeuler has the following requirements:

- Supporting N execution context (ie. CPUs)
- Supporting T number of tasks

- Tasks have identical priority and are scheduled coopertively
- System has a single global lock
- Any task can attempt to acquire the lock, Any task attempting to acquire the lock are gauranteed to be scheduled.
- If multiple tasks attempt to grab the lock, the tasks will be scheduled in lock request order.

4.2 Spec

We will model scheduler using the following variables:

$$
\begin{array}{l}
Init \triangleq \\
\quad \land\ cpus = [i \in 0\,..\,N-1 \mapsto \text{“”}] \\
\quad \land\ ready\_q = SetToSeq(Tasks) \\
\quad \land\ blocked\_q = \langle\rangle \\
\quad \land\ lock\_owner = \text{“”}
\end{array}
$$

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 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 [10] to converts a set into a ordered tuple. To use community module, one can install required .tla files into the tla project source directly.
- *lock_owner* represents the task that is current holding the lock.

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

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

$\lor\ Lock(k)$
$\lor\ Unlock(k)$
$Next\ \triangleq$
$\lor\ Running$
$\lor\ Ready$

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 a 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:

$MoveToReady(k)\ \triangleq$
$\land\ cpus[k] \neq$ ""
$\land\ lock\_owner \neq cpus[k]$
$\land\ ready\_q' = Append(ready\_q,\ cpus[k])$
$\land\ cpus' = [cpus$ EXCEPT $![k] =$ ""$]$
$\land$ UNCHANGED $\langle lock\_owner,\ blocked\_q,\ blocked\rangle$

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

$Lock(k)\ \triangleq$
$\lor\ \land\ cpus[k] \neq$ ""
$\ \land\ lock\_owner =$ ""
$\ \land\ lock\_owner' = cpus[k]$
$\ \land$ UNCHANGED $\langle ready\_q,\ cpus,\ blocked\_q,\ blocked\rangle$
$\lor\ \land\ cpus[k] \neq$ ""
$\ \land\ lock\_owner \neq$ ""
$\ \land\ lock\_owner \neq cpus[k]$ cannot double lock
$\ \land\ blocked\_q' = Append(blocked\_q,\ cpus[k])$
$\ \land\ blocked' = [blocked$ EXCEPT $![cpus[k]] = 1]$
$\ \land\ cpus' = [cpus$ EXCEPT $![k] =$ ""$]$
$\ \land$ 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 move on. When the lock is already held, the task moves into 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.

$Unlock(k)\ \triangleq$

$$
\begin{array}{ll}
\lor & \land\ cpus[k] \neq \text{“”} \\
& \land\ Len(blocked\_q) \neq 0 \\
& \land\ lock\_owner = cpus[k] \\
& \land\ lock\_owner' = Head(blocked\_q) \\
& \land\ cpus' = [cpus \text{ EXCEPT } ![k] = Head(blocked\_q)] \\
& \land\ ready\_q' = ready\_q \circ \langle cpus[k] \rangle \\
& \land\ blocked\_q' = Tail(blocked\_q) \\
& \land\ blocked' = [blocked \text{ EXCEPT } ![Head(blocked\_q)] = 0] \\
\lor & \land\ cpus[k] \neq \text{“”} \\
& \land\ Len(blocked\_q) = 0 \\
& \land\ lock\_owner' = \text{“”} \\
& \land \text{UNCHANGED } \langle ready\_q,\ blocked\_q,\ blocked,\ cpus \rangle
\end{array}
$$

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 running task is inserted at the end of the ready queue.

4.3 Safety

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

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

4.4 Liveness

Any tasks attempting to acquire the lock when the lock is already taken becomes blocked. A liveness property we can define is to check the scheduler gaurantee any blocked task eventually acquire 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*):

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

$\wedge$ *Fairness*

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:

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

Chapter 5

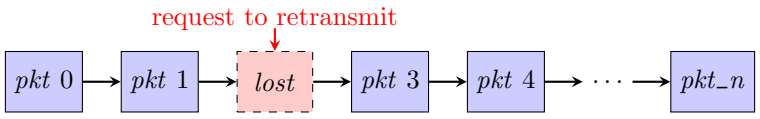
Selective Retransmit

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 artefacts 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.

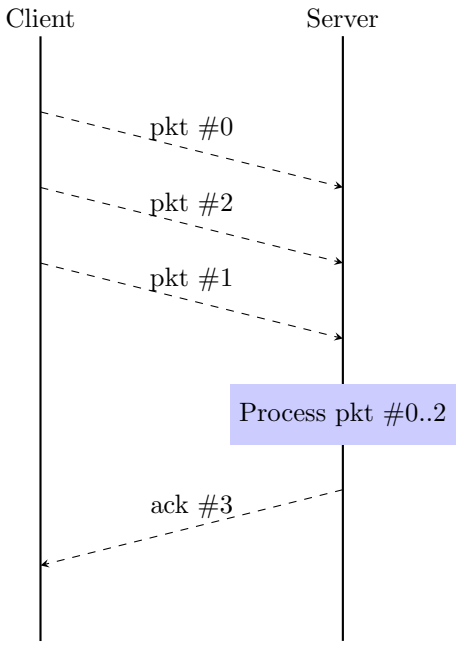
In this chapter, we will implement a simple selective retransmit algorithm.



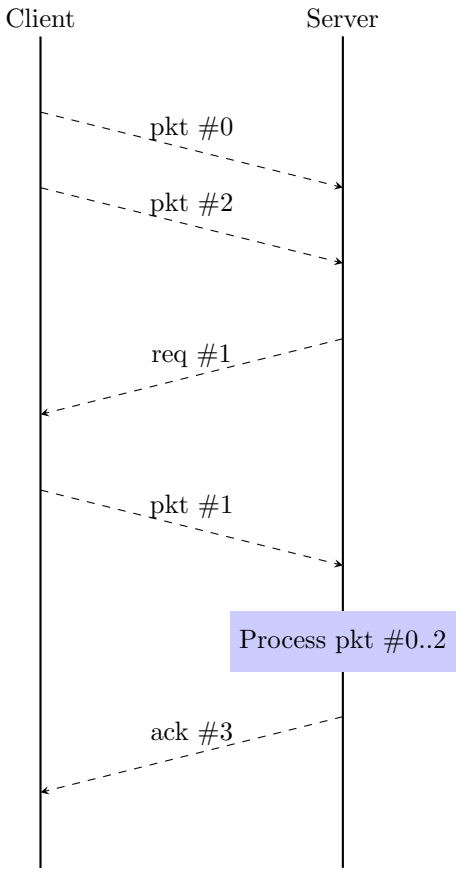
Since packets may arrive out-of-order, the server stamps the packets with sequence numbers to allow the client to order the packets as they arrive. Once the client has a set of ordered packets, it moves the packets from the receive buffer into the decoding engine to be displayed.

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 the server assumes once an acknowledgement of packet N is received, then all packets before N has also been received.

5.1 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*.

5.2 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:

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

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

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

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

$minv$

$Range \triangleq$
IF $MaxIndex \geq MinIndex$
THEN
$MaxIndex - MinIndex + 1$
ELSE
$MaxIndex + 1 + N - MinIndex$

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:

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

$ClientAcknowledgement \triangleq$
$\land\ client\_buffer \neq \{\}$
$\land\ MergeReady$
$\land\ client\_buffer' = \{\}$
$\land\ client\_rx' = MaxIndex$
$\land\ network' = AddMessage([dst \mapsto$ "server",
$type \mapsto$ "ack",
$ack \mapsto MaxIndex]$,
$network)$
$\land$ 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.

$ClientReceive(pp) \triangleq$

```
    ∧ network' = RemoveMessage(pp, network)
    ∧ client_buffer' = client_buffer ∪ {pp.seq}
    ∧ UNCHANGED ⟨server_tx, client_rx,
     server_tx_ack, server_tx_limit, lost⟩

Missing ≜
    LET
        full_seq ≜
            IF MaxIndex ≥ client_rx + 1
             THEN
                {x ∈ client_rx + 1 .. MaxIndex : TRUE}
             ELSE
                {x ∈ 0 .. MaxIndex : TRUE} ∪ {x ∈ client_rx + 1 .. N − 1 : TRUE}
        all_client_msgs ≜ {m ∈ network : m.dst = "client"}
        all_client_seqs ≜ {m.seq : m ∈ all_client_msgs}
        network_missing ≜ full_seq \ all_client_seqs
        client_missing ≜ full_seq \ client_buffer
        to_request ≜ network_missing ∩ client_missing
    IN
        to_request

ClientRetransmitRequest ≜
    ∧ ¬MergeReady
    ∧ client_buffer ≠ {}
    ∧ Missing ≠ {}
    ∧ network' = AddMessage([dst ↦ "server",
    type ↦ "retransmit",
    seq ↦ CHOOSE x ∈ Missing : TRUE],
      network)
    ∧ UNCHANGED ⟨server_tx, server_tx_limit,
     client_rx, client_buffer, server_tx_ack, lost⟩
```

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) ≜
    LET
```

```
        acks ≜ {(ack − k + N)%N : k ∈ 1 .. W}
    IN
        {m ∈ msgs : ¬(m.dst = "server" ∧ m.type = "ack" ∧ m.ack ∈ acks)}

ServerReceive(pp) ≜
    ∨ ∧ pp.type = "ack"
      ∧ server_tx_ack′ = pp.ack
      ∧ server_tx_limit′ = (pp.ack + W)%N
      ∧ network′ = RemoveStaleAck(pp.ack, RemoveMessage(pp, network))
      ∧ UNCHANGED ⟨server_tx, client_rx, client_buffer, lost⟩
    ∨ ∧ pp.type = "retransmit"
      ∧ network′ = AddMessage([dst ↦ "client", seq ↦ pp.seq],
      RemoveMessage(pp, network))
      ∧ lost′ = lost − 1
      ∧ UNCHANGED ⟨server_tx, server_tx_limit,
        client_rx, client_buffer, server_tx_ack⟩
```

When the server receives an acknowledgement for sequence number K, it assumes K-1 and prior were all received by the client. *RemoveStaleAck* is amodel optimization to drop all acknowledgements with sequence numbers less thanK. Note that sequence numbers k, k-N, and k-2*N, are all represented as k, and in theory, 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.

5.3 Refinement

5.3.1 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 refinement 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.

5.3.2 Retransmit Request Assumed Reliable

Retransmit requests packets 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 modelled, it would increase the runtime without providing many benefits. For this Spec, we simply assume all control packets are reliable.

5.4 Safety

5.5 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.

$$
\begin{array}{l}
\mathit{Liveness} \triangleq \\
\quad \forall\, i \in 0 \,..\, N-1 : \\
\quad\quad \mathit{client\_buffer} = \{\} \rightsquigarrow \exists\, j \in \mathit{client\_buffer} : i = j
\end{array}
$$

Chapter 6

Raft Consensus Protocol

6.1 Design

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 database replication protocol. With a replication factor of 3 (eg. data is replicated across 3 nodes) and hard drive failure rate of 0.81% per year, the possibility of the total failure where the entire replication group goes down is $1 - 0.0081^3 = 99.9999\%$ uptime [4].

This chapter implements only the leader election portion of the protocol to limit the scope of the discussion. For a full description of the the Raft protocol, please refer to the original paper [5].

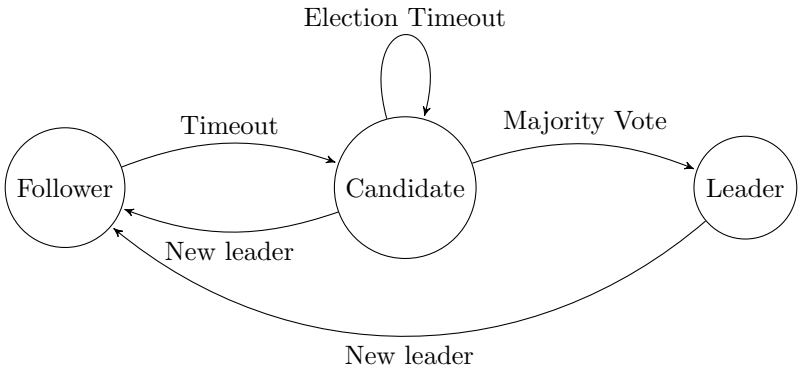
We will briefly describe Raft and its leader election process below:

- A Raft cluster have N nodes, the cluster work collective 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 have 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 heartbeat to the follower, indicate its alive
- If a follower fails to receive heartbeat from the leader after timeout, it will become a candidate, vote for itself, and campaign to be leader
- A candidate collect the majority of the vote becomes the leader

- If multiple candidates are campaigning and a split vote happen, candidates will eventually declare election timeout and start new round of election
- The cluster can have multiple leaders due to unfavourable 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 request and responses include the sender's term, allowing the receiver to react accordingly

The protocol also included description about log synchronization, state recovery, and more. Many details are ommitted in chis chapter to reduce modeling cost. The N nodes in the cluster operate *independently* following the above hueristics. Hopefully this highlights the complexity around verifying the the correctness of the protocol.

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



6.2 Spec

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

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

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

```
    [fSrc ↦ i, fDst ↦ s, fType ↦ "RequestVoteReq", fTerm ↦ term[i]]
        : s ∈ Servers \ {i}
}

Campaign(i) ≜
    ∧ vote_requested[i] = 0
    ∧ vote_requested' = [vote_requested EXCEPT ![i] = 1]
    ∧ messages' = messages ∪ RequestVoteSet(i)
    ∧ UNCHANGED ⟨state, term, vote_granted, voted_for⟩

KeepAliveSet(i) ≜ {
    [fSrc ↦ i, fDst ↦ s, fType ↦ "AppendEntryReq", fTerm ↦ term[i]]
        : s ∈ Servers \ {i}
}

Leader(i) ≜
    ∧ state[i] = "Leader"
    ∧ messages' = messages ∪ KeepAliveSet(i)
    ∧ UNCHANGED ⟨state, voted_for, term, vote_granted, vote_requested⟩

BecomeLeader(i) ≜
    ∧ Cardinality(vote_granted[i]) > Cardinality(Servers) ÷ 2
    ∧ state' = [state EXCEPT ![i] = "Leader"]
    ∧ UNCHANGED ⟨messages, voted_for, term, vote_granted, vote_requested⟩

Candidate(i) ≜
    ∧ state[i] = "Candidate"
    ∧ ∨ Campaign(i)
      ∨ BecomeLeader(i)
      ∨ Timeout(i)

Follower(i) ≜
    ∧ state[i] = "Follower"
    ∧ Timeout(i)

Receive(msg) ≜
    ∨  ∧ msg.fType = "AppendEntryReq"
       ∧ AppendEntryReq(msg)
    ∨  ∧ msg.fType = "AppendEntryResp"
       ∧ AppendEntryResp(msg)
    ∨  ∧ msg.fType = "RequestVoteReq"
       ∧ RequestVoteReq(msg)
    ∨  ∧ msg.fType = "RequestVoteResp"
       ∧ RequestVoteResp(msg)

Next ≜
    ∨ ∃ i ∈ Servers :
        ∨ Leader(i)
```

```
    ∨ Candidate(i)
    ∨ Follower(i)
∨ ∃ msg ∈ 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 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

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

```
RequestVoteReq(msg) ≜
    LET
        i ≜ msg.fDst
        j ≜ msg.fSrc
        type ≜ msg.fType
        t ≜ msg.fTerm
    IN
         haven't voted, or whom we voted re-requested
        ∨ ∧ t = term[i]
          ∧ ∨ voted_for[i] = j
            ∨ voted_for[i] = ""
          ∧ voted_for' = [voted_for EXCEPT ![i] = j]
          ∧ messages' = AddMessage([fSrc ↦ i,
                                    fDst ↦ j,
                                    fType ↦ "RequestVoteResp",
```

```
                                        fTerm ↦ t,
                                        fSuccess ↦ 1],
                                        RemoveMessage(msg, messages))
    ∧ UNCHANGED ⟨state, term, vote_granted, vote_requested, establish_leadership⟩
  already voted someone else
∨ ∧ t = term[i]
  ∧ voted_for[i] ≠ j
  ∧ voted_for[i] ≠ ""
  ∧ messages' = AddMessage([fSrc ↦ i,
                                        fDst ↦ j,
                                        fType ↦ "RequestVoteResp",
                                        fTerm ↦ t,
                                        fSuccess ↦ 0],
                                        RemoveMessage(msg, messages))
    ∧ UNCHANGED ⟨state, voted_for, term, vote_granted, vote_requested, establish_leadership⟩
∨ ∧ t < term[i]
  ∧ messages' = AddMessage([fSrc ↦ i,
                                        fDst ↦ j,
                                        fType ↦ "RequestVoteResp",
                                        fTerm ↦ term[i],
                                        fSuccess ↦ 0],
                                        RemoveMessage(msg, messages))
    ∧ UNCHANGED ⟨state, voted_for, term, vote_granted, vote_requested, establish_leadership⟩
  revert back to follower
∨ ∧ t > term[i]
  ∧ state' = [state EXCEPT ![i] = "Follower"]
  ∧ term' = [term EXCEPT ![i] = t]
  ∧ voted_for' = [voted_for EXCEPT ![i] = j]
  ∧ vote_granted' = [vote_granted EXCEPT ![i] = {}]
  ∧ vote_requested' = [vote_requested EXCEPT ![i] = 0]
  ∧ establish_leadership' = [establish_leadership EXCEPT ![i] = 0]
  ∧ messages' = AddMessage([fSrc ↦ i,
                                        fDst ↦ j,
                                        fType ↦ "RequestVoteResp",
                                        fTerm ↦ t,
                                        fSuccess ↦ 1],
                                        RemoveMessage(msg, messages))
```

The handling is split into three cases:

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

6.3 Refinement

The model checker will run the spec as defined, but due to the expoential growth of states it is unlikely to complete in a reasonable amount of time. We need to simplify the model and possibly trades off some correctness. 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 describe a set of optimization implemented for this example.

6.3.1 Modeling Messages as a Set

In the original Raft TLA+ Spec [6], 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 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.

6.3.2 Limit Term Divergence

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

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

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

$Timeout(i) \triangleq$
$\land$ $LimitDivergence(i)$
$\land$ $state' = [state$ EXCEPT $![i] =$ "Candidate"$]$
$\land$ $voted\_for' = [voted\_for$ EXCEPT $![i] = i]$ voted for myself
$\land$ $vote\_granted' = [vote\_granted$ EXCEPT $![i] = \{i\}]$
$\land$ $vote\_requested' = [vote\_requested$ EXCEPT $![i] = 0]$
$\land$ $term' = [term$ EXCEPT $![i] = @ + 1]$ bump term
$\land$ $establish\_leadership' = [establish\_leadership$ EXCEPT $![i] = 0]$
$\land$ UNCHANGED $\langle messages \rangle$
/ PrintT(state')

6.3.3 Normalize Cluster Term

However, term *itself* can grow unbounded. This is a key tenent converging protocols rely on, an monotonically increasing counter. We want to *normalize* the range of terms in the cluster so the minimum value resets back to 0. This provides an upper bound to the state graph.

$Normalize \triangleq$
LET
$values \triangleq \{term[s] : s \in Servers\}$
$max\_v \triangleq$ CHOOSE $x \in values : \forall\, y \in values : x \geq y$
$min\_v \triangleq$ 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 node is on *MaxTerm*, the cluster terms are normalized.

Another caveat here is in the initial implementation I didn't update messsages. This led to liveness property violation as the messages had terms disagree-

ing with the system state. To simplify the spec I simply cleared all messages. This indirectly verifies a portion of the packet loss handling in the spec as well.

6.3.4 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 are only sent once. The implementation no longer tracks if the responses were received, since the spec should handle packet loss scenarios as well.

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

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

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

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

6.3.5 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:

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

$pruned \cup \{to\_add\}$

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

6.3.6 Enable Symmetry

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

$Perms \triangleq Permutations(Servers)$

6.4 Safety

One of the goals for the protocol is to ensure the cluster only have one leader. It is possible for the clusters to have mulitple leaders due to unfavourable network connections. For example, a leader node is partitioned off and a new leader is elected. However, even when the cluster have mutliple 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:

$$
\begin{aligned}
&LeaderUniqueTerm \triangleq \\
&\quad \forall\, s1,\, s2 \in Servers : \\
&\qquad (state[s1] = \text{“Leader”} \land state[s2] = \text{“Leader”} \land s1 \neq s2) \Rightarrow (term[s1] \neq term[s2])
\end{aligned}
$$

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

6.5 Liveness

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

- A node timed out and starts a new election on a new term
- A partitioned follower receives 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 increase. This can be described as below:

$$
\begin{aligned}
&Converge \triangleq \\
&\quad \forall\, s \in Servers : \\
&\qquad term[s] = 0 \rightsquigarrow term[s] = MaxTerm - MaxDiff
\end{aligned}
$$

Instead of *MaxTerm*, we use *MaxTerm-MaxDiff* to ensure the liveness property is always upheld even after *Normalization*. However, running the 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:

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

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 7

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.

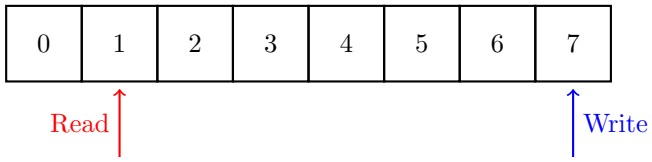
A real implementation needs to account for memory ordering effects specific to the architecture. For example, ARM has a weak memory ordering model where read/write may appear out-of-order between CPUs. In this chapter, we will assume *logical* execution order where each command is perceived as issued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

7.1 Design

The following describes the SPSC queue requirements:

- Two executing context, reader and writer
- Writer pointer advances after write
- Reader pointer advances after read
- 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 a 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.

7.2 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 queue is empty by comparing read and write pointer. If queue is empty, reader early returns. If queue is not empty, 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. The reason works symmetrically with the write index.

7.3 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:

$$MutualExclusion \triangleq \neg((pc[WRITER] = \text{“w\_cs”}) \wedge (pc[READER] = \text{“r\_cs”}) \wedge rptr = wptr)$$

7.4 Liveness

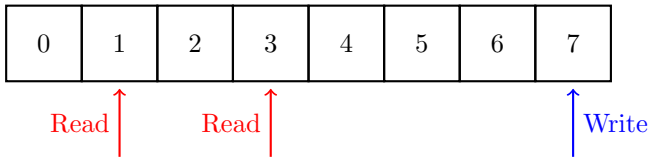
For liveness, we want to check the queue never hangs. This can be described as all indices are eventually used and unused:

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

Chapter 8

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 read 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.

8.1 Design

For the design:

- SPMC is implemented as a circular queue with size of N
- The status of individual index is represented as a status array of size N
- The status of each index is either UNUSED, WRITTEN, or READING

- Each reader maintain its own read pointer
- A *outstanding* counter is incremented by the writer when 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 walk its read pointer until it successfully reserve 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 in implementing SPMC, the above description is what we will implement in this chapter.

8.2 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 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, stores that onto local variable counter.
- Reader early returns if 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 rv is non-success, another reader has *won* the reservation. The current reader can attempt to reserve again if *outstanding* is non-zero.

If rv is a success, 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] ≠ 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 return if queue is full. After 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 haven't done the actual read. If both checks pass, then writer now has an *UNUSED* index it can write to.

8.3 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 critical section, they must operate on different indices:

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

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

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

8.4 Liveness

All indices must be used as the system runs. The following verifies all unused indices are eventually used, and all used indicies 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 describe such scenario, where two non-contiguous indicies have been reserved for reading. In this case we expect the indicies to eventually be re-used. This means the system remains functional after such scenario.

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

Part IV

Reference

Chapter 9

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.

9.1 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
$k \triangleq c \setminus d$

9.2 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
$c \triangleq A \circ B$

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
$g \triangleq \{x \in 1 \,..\, Len(c) : c[x] = 2\}$

9.3 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
$c \triangleq a \mathbin{@@} b$

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 {}, {}, {}

$e \triangleq [k \in SetA \mapsto \{\}]$

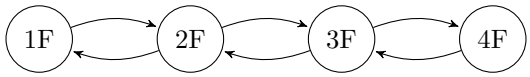
Create a mapping that is the same as e, except key a's value is "a", "b", "c"

$f \triangleq [e \text{ EXCEPT } ![\text{“a”}] = \{\text{“a”}, \text{“b”}, \text{“c”}\}]$

Chapter 10

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:



10.1 Liveness

Consider the following elevator *Spec*:

MODULE *elevator*

EXTENDS *Integers*
VARIABLES a
$vars \triangleq \langle a \rangle$
$TOP \triangleq 4$
$BOTTOM \triangleq 1$
$Init \triangleq$
$\quad \wedge a = BOTTOM$
$Up \triangleq$
$\quad \wedge a \neq TOP$
$\quad \wedge a' = a + 1$
$Down \triangleq$
$\quad \wedge a \neq BOTTOM$
$\quad \wedge a' = a - 1$
$Spec \triangleq$
$\quad \wedge Init$
$\quad \wedge \Box[Up \vee Down]_a$

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 keep 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:

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

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

10.2 Weak Fairness

Weak fairness is defined as:

$$\Diamond\Box(\mathit{ENABLED}\langle A\rangle_v) \Rightarrow \Box\Diamond\langle A\rangle_v \tag{10.1}$$

$\mathit{ENABLED}\langle A\rangle$ 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 is *eventually always* true (ie. elevator decides to stay on 1F but *can* go up), the elevator *always eventually* go up.

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

Running the spec against 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:

$Liveness \triangleq$
$\quad \wedge\, a = BOTTOM \leadsto a = TOP$

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>
```

Model checker identified a case where the elevator is perpetually stuck going between 1F and 2F, but never go 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.

10.3 Strong Fairness

Strong fairness is defined as:

$$\Box\Diamond(ENABLED\langle A\rangle_v) \Rightarrow \Box\Diamond\langle A\rangle_v \tag{10.2}$$

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 transition 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:

$Spec \triangleq$
$\quad \wedge\, Init$
$\quad \wedge\, \Box[Up \vee Down]_a$
$\quad \wedge\, \mathrm{WF}_a(Down)$
$\quad \wedge\, \mathrm{SF}_a(UP)$

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

If we take a closer look at the enabling condition for *Up*, it only requires 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 really want is strong fairness on *Up* for *every floor*, instead of *any floor except top floor*. So if elevator makes to 2F once, it will *always eventaully* go to 3F. If elevator makes to 3F once, it will *always eventaully* go to 4F, so on and so forth. The following is the change required:

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

With this change the model checker will pass.

Chapter 11

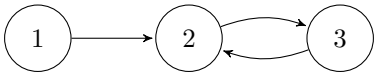
Liveness

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*. We are rarely interested only in the correctness of one state in the system, but rather in the correctness of system behavior *across* a set of states.

This book has already provided a few examples of liveness properties: 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 go through a very simple state machine to demonstrate liveness properties.

Assume a simple three-system system:



This can be described by the following spec:

MODULE *liveness*

EXTENDS *Naturals*
VARIABLES *counter*
$vars \triangleq \langle counter \rangle$

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

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

$$\begin{array}{l} Init \triangleq \\ \quad \land\ counter = 0 \end{array}$$

$$\begin{array}{l} Inc \triangleq \\ \land\ counter' = counter + 1 \end{array}$$

$$\begin{array}{l} Dec \triangleq \\ \land\ counter' = counter - 1 \end{array}$$

$$\begin{array}{l} Next \triangleq \\ \quad \lor\ \land\ counter \neq 3 \\ \quad\ \ \land\ Inc \\ \quad \lor\ \land\ counter = 3 \\ \quad\ \ \land\ Dec \end{array}$$

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

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

11.1 Always Eventually

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

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

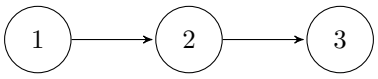
Once the system makes it to state 3, the system is stuck in a loop transitioning states 2 and 3. It doesn't *remain* in state 3, but it does *always eventually* transition to state 3. The system as described fulfills this liveness property.

11.2 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:

$Next \triangleq$
$\quad \wedge\ counter \neq 3$
$\quad \wedge\ Inc$

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.

11.3 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
$LeadsTo \triangleq counter = 0 \rightsquigarrow counter = 3$

state 0 leads to state 4: FALSE - model checker reports a violation
$LeadsTo \triangleq counter = 0 \rightsquigarrow counter = 4$

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

Chapter 12

General Guideline

12.1 Model Checker Debug

Debugging in TLC is a bit different than debugging with normal programs. A step in the model checker is really a state transition. Even if the model cheker completes, it's still worthwhile dump and audit the states just to make sure the 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 state being set to certain value to confirm the Spec is working as intended.

12.1.1 Dead Lock

Deadlock typically happens when the model checker ran out of things to do. This is typically 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 dead lock to simplify debug.

12.1.2 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 keep going to the top floor, or the system is stuck dropping and re-transmit the same packet.

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

For a detail fairness description please refer to Chapter 10.

12.2 Model Refinement

This is, in some sense, the *art* associated with writing model checker verifiable TLA+ spec. 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. Designer should focus on optimizing the broad stroke, such as removing features that are harder to get wrong, and focus the model on the bits that have the highest return on investment.

One useful way of trimming out low value portion of the 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 details that can be removed from the Spec.

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

As a broad stroke principle: when the Spec has millions or higher more states, it is unlikely to terminate within a few seconds. From first principles if a designer can find one failure case in a million states, designer can also likely simplify the model to reproduce the failure in much fewer states.

Chapter 13

Reference

Bibliography

[1] *https://lamport.azurewebsites.net/tla/book.html*

[2] Srikumar Subramanian *https://sriku.org/posts/fairness-in-tlaplus/*, 2015

[3] Richard M. Murray, Nok Wongpiromsarn *Linear Temporal Logic, Lecture 3*, 2012

[4] *https://www.backblaze.com/blog/cloud-storage-durability/*

[5] *https://raft.github.io/raft.pdf*

[6] *https://github.com/ongardie/raft.tla*

[7] *https://github.com/tlaplus/tlaplus*

[8] *https://ahelwer.ca/post/2023-11-01-tla-finite-monotonic/*

[9] *https://en.wikipedia.org/wiki/C10k_problem*

[10] *https://github.com/tlaplus/CommunityModules*