Thinking in TLA+
Modeling Judgment for System Design

Murat Demirbas
twitter @muratdemirbas
muratbuffalo.blogspot.com

MongoDB Research
http://mongodb.com

April 12, 2026

The Premise

In the age of LLMs, syntax is no longer the bottleneck for learning TLA+

- The accidental complexity (syntax, tooling) is going away
- The intrinsic complexity remains: *modeling judgment*

TLA+ is not a verification tool you bolt on at the end
It is a thinking tool that shapes how you reason about systems

The Perspective Shift

Engineers think in: code, control flow, local state
TLA+ forces a different mode: mathematical, declarative, and global

- You specify what must hold, not how to achieve it

This shift in perspective is the real value of TLA+. Over many industry projects, I found that TLA+ works because it changes how you think

- After 15+ years of using TLA+, I now think of it is a design accelerator!

7 Mental Models for Thinking in TLA+

These are the thinking patterns behind effective TLA+ use

They are usually acquired by osmosis over years of practice

Let me try to make them explicit and actionable

1. Abstraction, Abstraction, Abstraction

Omission is the default: add only when leaving out breaks your exploration

- Cross-cut to focus on the protocol you want to investigate
- What is the behavioral slice I care about, and what can I safely ignore?

TLA+'s tight feedback loop accelerates design experience

1. Abstraction: CosmosDB (2018)

- Anti-pattern: model the distributed database engine → state-space explosion
- Instead: "history as a log" abstraction for client-facing behavior
- Sort-merge captures replication internals; read index captures consistency
- Five consistency levels became clear predicates over operation histories

A model does not have to capture everything. It just needs to capture the behavior that matters!

1. Abstraction: Secondary Index at Aurora DSQL (2023)

- Engineer kept finding corner cases in his 6-pager design doc
- I wrote an initial TLA+ model: database as a log of operations

- Over the weekend, he'd written variations with no prior TLA+ experience
- TLA+ forced a declarative approach instead of patching corner cases

Don't grow the design by patching, search the protocol solution space!

2. Embrace the Global Shared Memory Model

- A program = a set of variables + a finite set of guarded actions
- {state} action {state'} — Sir Tony Hoare's triples

- GSM is the most minimal way to model; hides message passing
- Be frugal with variables: each one exponentially explodes state space

Safety and liveness become compact predicates over global state!

2. GSM Mapping to Message Passing

- Use indexed local variables: `vote[i]` refers to node *i*'s vote
- Global variable is the array; local version is `vote[i]`

Keep to "localish" guards and local variable assignments, and there would be a refinement possible for the message passing model

3. Refine to Local Monotonic Guards

GSM trap: easy to write guards with illegal knowledge (nonlocal reach)

- Audit every action: what could a real node actually know?

Use stable, monotonic, or locally stable predicates for guards

- If the guard remains true despite stale state then less coordination is needed

For doing coordination efficiently and safely in a distributed system, the insight is almost always to transform the problem in to a monotonic one!

- You gotta keep things moving up! (*For your definition of up*)

3. Slow is Fast: The Key Insight

- A *slow action's* guard tolerates stale information & asynchrony
  - Guard is a stable predicate (once true, stays true)
  - Guard depends only on local variables
  - Guard is locally stable (only this node's actions can falsify it)
- A *fast action* requires fresh global state

Paxos: ballot monotonicity makes voting guards locally evaluable

4. Derive Good Invariants

- Invariants = distilled understanding of your protocol
- They scaffold exploration, composition, fault-tolerance, testing
- Anti-patterns: trivial invariants (always true), end-state-as-invariant
- If TLC passes instantly, your model is too small or invariants too weak

Don't stop at safety, write liveness! $\diamond$ `Termination`, `Init` $\rightsquigarrow$ `Solution`

4. Invariants: Paxos Example

- Consensus level: at most one value is chosen
- Voting level: chosen = values with a quorum of votes in some ballot
- Two rules: (1) no different values in same ballot, (2) only vote if safe at b

These invariants are tight, non-trivial, and scaffold the refinement chain
They show genuine understanding of *why* consensus works

4. Invariants: PowerSet Paxos from Amazon (2022)

After weeks of sketching in Excel, Chuck Carman used TLA+ to capture the protocol

- A week to translate to code
- 75-80% of the code was testing invariants

"Much regret has been expressed around not modeling other parts in TLA+"

The expensive part isn't writing the algorithm. It's trusting it.

5. Explore Through Stepwise Refinement

- Start with the most abstract spec, refine downward
- Each level adds detail while preserving safety from above
- Refinement in TLA+ is implication: concrete behaviors $\subseteq$ abstract behaviors

To explore a variant: go up, change one refinement step, verify!
When stuck patching corner cases, go back up

5. Refinement: Lamport's Paxos Derivation

- Consensus: `chosen` transitions from {} to {v}. That's the entire spec.
- Voting: acceptors vote, majority chooses. Refines Consensus.
- Paxos: adds ballots, leaders, messages. Refines Voting.

At each level, different refinement choices yield different protocols
All satisfy the same high-level safety property

5. Refinement: WPaxos and LeaseGuard

WPaxos (2016): Started modeling early, explored flexible quorums

- Sharpened quorum definitions; used TLA+ snippets directly in the paper

LeaseGuard (2024): Discovered two optimizations we hadn't anticipated

- Including inherited lease reads — wouldn't have found it without TLA+

Start modeling early, don't wait until you think the design is "done"

6. Aggressively Refine Atomicity

Large atomic actions sweep concurrency under the rug

- Start coarse-grained for correctness, then split and verify
- Give the implementation maximum freedom to reorder and parallelize

Connects to mental model 3: monotonic guards enable fine splitting

6. Atomicity: StableEmptySet (2022)

Problem: a distributed set that, once empty, stays empty forever

- No atomic IsEmpty in a distributed system, so can't just check IsEmpty
- Ernie Cohen pushed things to finest granularity & no distributed locking
- Interleaving surface area exploded, but TLC verified without problems

Push for the finest granularity you can. Use TLA+ to verify interleavings are safe.

7. Share Your Mental Models

Specs are communication artifacts, not just verification inputs

- Write for humans: clear action names, TypeOK, and explicit invariants
- A well-structured spec shows essential logic without implementation noise

- Spectacle for visualization: bridge TLA+ precision and operational intuition

7. Sharing — DSQL and MongoDB

Aurora DSQL (2022): TLA+ model served as communication anchor

- Sped up onboarding; kept everyone aligned on protocol design

MongoDB Txns (2025): Modeled after production

- Generated thousands of unit tests from model traces
- Became an authoritative description of what the protocol guarantees

It's never too late to model!
One of the purest intellectual pleasures is using TLA+ to reduce a tangled distributed system to its essential logic.

Closing

As LLMs write more code, TLA+ for design and reasoning grows in value

- The hardest part is learning what to model, what to omit, and when to trust what the model tells you
- This is modeling judgment, and it is a skill you can practice & learn

/ AI + TLA+ stack for software development for the win!/