Turning an incident report into a design issue with TLA+

Finn Hackett (UBC) & Markus A. Kuppe (MSFT)
By Some Not-SREs

A. Finn Hackett
PhD Student at U. of British Columbia
*Programming Languages Researcher*
→Intern at Microsoft Research S’22

Markus A. Kuppe
Principal Research Engineer (MSFT)
*TLA+ eng, Postmortem Reviewer*
→Finn’s internship mentor S’22

Joshua Rowe
*Principal Engineer (MSFT)*
*Azure CosmosDB*
→Domain expert

Thanks to postmortem owner Ben Pannell (MSFT)!
Highlights

We learned about a high-profile 28-day incident at Microsoft Azure.

Mitigation was a feature revert (full repair a costly multi-year redesign).

Reproducing the issue at small scale is impractical…

Idea: use modeling to unambiguously document the problem.

We use TLA+ here, but this applies to similar tools as well.
A Workflow Beyond Incident Reports

Conventional outcomes

Incident -> Incident report -> System(s) -> Specification of correct behavior

Checker

Model

Counter-examples

… which summarize the issue at a high level.
Story Time: an Incident

1. A Performance Optimization is Deployed

2a. Error Metrics Don’t Noticeably Change…

2b. Uncommon Client Request Pattern Starts to Fail Very Often

3. Lots of Debugging Later, Client Considers Azure Might be Wrong

4. Azure Devs Investigate

5. System Running Too Fast, Roll back!
location
FROBLE
deployment
s/${real product name}/FROBLE
What’s the Big Deal?

Client is not seeing errors anymore, but…

😞 That took a long time to figure out.

(28 days to fix)

😊 We know why we rolled back, but design-level insight is missing.

🤔 What about the design made a component being *too fast* a problem?

We want to clearly model and understand what happened at the conceptual level.
Step 1: Understand Underlying System(s)
What is Azure CosmosDB?

Need to understand what the system is supposed to do to model it.

What we present next is based on what we learned by drafting our model.

A planet-scale key-value store - a big distributed system.

Stores your data as key-value mappings, useful for app/service data.

Similar to? Amazon DynamoDB, Google Firestore…

Also using TLA+
Key Detail: Communication with Azure Cosmos DB

1. Write metadata successfully

2. Enqueue work

3. DB replies metadata not found (but only when 2. is “too fast”)

4. Job failed…💀
CosmosDB: Consensus Machine

Clients exchange get/set messages with different servers.

Need consensus to make sense of this.
5 consistency levels control how strictly CosmosDB maintains consensus.

<table>
<thead>
<tr>
<th>Consistency Level</th>
<th>Effect on Client View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Consistency</td>
<td>Global order; clients always see latest versions</td>
</tr>
<tr>
<td>Bounded Staleness</td>
<td>Old values visible for limited (<em>bounded</em>) time</td>
</tr>
<tr>
<td>Session Consistency</td>
<td>Synchronize only between clients with shared token</td>
</tr>
<tr>
<td>Consistent Prefix</td>
<td>Generally similar to Eventual Consistency</td>
</tr>
<tr>
<td>Eventual Consistency</td>
<td>Old values should eventually stop showing up</td>
</tr>
</tbody>
</table>

- **Strict + inefficient**
- **Recommended in docs (but tricky)**
- **Fast + unreliable**
Step 2: Represent Underlying System(s)
CosmosDB: What to Model for Our Purpose?

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."
– Edsger W. Dijkstra

<table>
<thead>
<tr>
<th>No</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>❌ Detailed client API syntax</td>
<td>✔ Plain key-value r/w</td>
</tr>
<tr>
<td>❌ Server management</td>
<td>✔ Client view of servers</td>
</tr>
<tr>
<td>❌ Anything client can’t see</td>
<td>✔ Anything client can see (especially unusual things)</td>
</tr>
</tbody>
</table>
… But Have a Domain Expert Model It
CosmosDB: the TLA+ Model

✔ Simulate concurrent key-value reads and writes at all 5 consistency levels.

⏳ Reusable, complete model: took 3 months to build w/ dev input.

🌟 But it took 1 day to use in this postmortem.

Github: https://github.com/tlaplus/azure-cosmos-tla/tree/master/simple-model

See our conference paper for all the details:
https://doi.org/10.48550/arXiv.2210.13661 (preprint)
Step 3: Model the Incident

Demo Time!
1. Write value to CosmosDB
   
   2a. Value does not get replicated enough
   
   2b. Value is fully replicated
   
   2c. Node failure; value lost
   
   3a. Key not found; bad
   
   3b. Key found; ok actually
   
   3c. Key not found; also bad

TLA+ lets us consider all these possibilities at the same time.
Thinking in States: the Full

(... with failure sim disabled, vars skipped)
The Corrected Design

1. Write metadata successfully

2. Enqueue work with attached token

3. Read with token OK ✨

4. No problem 🎉
What We Learned

💡 Problem is using session consistency without sharing tokens.

📝 Original mitigation changed the probability of the issue, but wasn’t a fix. The issue was always there, just very rare.

📝 Clear when presented this way, but issue was hidden behind APIs.

📝 Engineers were not surprised by our results - we confirmed what was suspected but could not be demonstrated.

📝 Modeling helped us think and investigate.
Summary

Reproduced incident spanning multiple foundational systems. 🎉

Our modeling workflow can go beyond current incident reports.

You can do this too. Learn at http://tlapl.us.

TLA+ model (linked from official docs):

Any Questions?

Conventional outcomes

- Incident
- System(s)

Incident report

Specification of correct behavior

Model

Checker

Counter-examples