CS6213 Project Proposal

George Pîrlea and Darius Foo

April 2nd, 2021

# Outline

We are looking at **lightweight formal methods** (= an engineer can learn all they need to know to produce something useful in 2-3 weeks [11]) for distributed systems. Specifically, the following workflow:

1. Engineers describe the system *design* as a state transition system (in TLA) and debug it by model-checking key properties.
2. Engineers check that the system *implementation* matches the design by identifying properties that can be checked either at runtime via monitoring [6], or by symbolic model checking [4], or as part of a continuous integration (CI) pipeline via testing [5].

Can we come up with a “pay-as-you-go”[[1]](#footnote-2) workflow for the verification of distributed system implementations? Is it possible to make lightweight verification of distributed systems as easy as unit testing?

# Research Questions

1. **Conformance testing.** Given a TLA model that satisfies some simple properties and a system implementation, what is the easiest way to check that the implementation satisfies the properties?
2. **Refinement mappings.** Given a model and an implementation, how can we best define the correspondence between the two, perhaps via a combination of manual/inferred annotations?
3. **Empirical evaluation of conformance.** For real-world distributed systems that come with both TLA models and implementations, *empirically*, is there a correspondence between the model and the actual implementation?
4. **Automatic discovery of refinement mappings.** Can we automatically concretise models, i.e., make them closer to the implementation, to apply techniques like *model-based trace checking* (MBTC)?
5. **Automation of conformance testing.** Given a model with a known refinement mapping to the implementation, can we automatically instrument the code for model-based trace checking?

1. **Model creation.** If we have an implementation, but no model, what is the best approach to writing a model that is suitable for conformance testing?
2. **Discover invariants to check in the implementation.** Can we automatically discover invariants of the model [9] and *check* that the implementation conformsto these same invariants (after the appropriate refinement mapping is applied), akin to *model-based test-case generation* (MBTCG)?

# Project Scope & Motivation

For the project, we will focus on research questions Q1 – Q3, and will tangentially explore Q6.

Specifically, we will select three open-source distributed system implementations and evaluate whether the implementations satisfy their TLA specification.

**Verifying Raft implementations.** To ease the workload (and potentially make for a better story in an empirical evaluation paper), all three of the systems we choose will be [open-source implementations](https://raft.github.io/#implementations) of Raft, i.e., the implementations will share a single specification. Some of these implementations are very widely used, so finding a serious bug in them would be impactful.

**TLA models.** There is an existing [TLA+ model for Raft](https://github.com/ongardie/raft.tla/blob/master/raft.tla), written by Diego Ongaro himself, but that model *does not* include membership changes and log compaction, features that are error-prone to implement. A later model[[2]](#footnote-3), developed by PhD students at Carnegie Mellon University, extended the original with cluster membership changes, finding a safety violation in the Raft protocol in the process [2]. To our knowledge, no model featuring log compaction exists. The combination of the two features might uncover further errors in the protocol since the log is used to track membership changes.

**Coq proof.** Woos et al. developed a Raft implementation (without membership changes and log compaction) in Coq and proved its safety [15]. We might be able to mine this proof’s lemmas for properties that are cheaper to check than linearizability.

**Different descriptions of Raft.** Of note is the fact that there are differences between the Raft algorithm described in the original paper [13] and the one described in Ongaro’s dissertation [12], and that both algorithms have (known) errors: [the paper version is not live](https://decentralizedthoughts.github.io/2020-12-12-raft-liveness-full-omission/) and [the dissertation version is not safe](https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E/m/d2D9LrWRza8J) [2] (the dissertation was later updated to include the fix). Moreover, realistic implementations often make changes to the protocol which are not formally specified [2].

In our view, the combination of (1) log compaction not being formally specified, (2) errors in the original protocol descriptions, and (3) ad hoc protocol changes might be a potential source of bugs even in widely used implementations.

**State of the art.** The state of the art in terms of validating the correctness of these systems seems to be differential testing [3, 7], fault injection [14], and linearizability testing [8]. For instance, **etcd**, a reliable key-value store used in production at companies like Yandex, Tencent, and Grab, is tested by running the system in a cluster under high load, injecting [specific faults](https://pkg.go.dev/github.com/coreos/etcd/functional/rpcpb?utm_source=godoc#Case) and then checking that the system is consistent and live.

While this kind of testing is effective for discovering partition-tolerance bugs [10], it is unclear whether it is capable of discovering “deep” bugs. If, for example, we manually introduce the known bugs in Raft in the **etcd** implementation (supposing they are not already present), would the existing testing framework catch them? We suspect that the answer is no.

Model-based checking, on the other hand, might be better at uncovering “deep” bugs. The goal of this project is to test whether this is indeed the case.

**Feasibility.** We are starting from an almost-complete TLA model and testing widely used implementations. Some of these already have extensive testing frameworks, so it is unlikely that we will encounter major difficulties in instrumenting the projects with extra checks. Given this, we can hope that most of our effort will be spent on answering research questions Q1–Q3, rather than on low-level engineering.

Next steps (not necessarily in order):

* Check if Raft’s safety bug could have been caught by **etcd**’s testing framework
* Request the extended Raft model (with configuration changes) from the CMU authors
* Extend the TLA model with log compaction
* Introduce bugs into a minimal Raft implementation and see how difficult it is to find each of them
* Decide on first system and technique to try on it
  + Instrumentation for MBTC
  + Specification mutation for MBTCG

# References

[1] Alvaro, P. and Tymon, S. 2017. Abstracting the Geniuses Away from Failure Testing. *Communications of the ACM*.

[2] Amos, B. and Zhang, H. 2015. *15-812 Term Paper: Specifying and proving cluster membership for the Raft distributed consensus algorithm*.

[3] Chakoumakos, R. and Trusheim, S. 2014. Raft Monkey: Diﬀerential Testing for Raft Implementations. (Dec. 2014), 6.

[4] Chong, N. et al. 2020. Code-level model checking in the software development workflow. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice* (Seoul South Korea, Jun. 2020), 11–20.

[5] Davis, A.J.J. et al. 2020. Extreme modelling in practice. *Proceedings of the VLDB Endowment*. 13, 9 (May 2020), 1346–1358. DOI:https://doi.org/10.14778/3397230.3397233.

[6] Francalanza, A. et al. 2018. Runtime Verification for Decentralised and Distributed Systems. *Lectures on Runtime Verification*. E. Bartocci and Y. Falcone, eds. Springer International Publishing. 176–210.

[7] Gilbert, C. 2014. *Flotsam: Evaluating Implementations of the Raft Consensus Algorithm*.

[8] Kingsbury, K. 2013. *Jepsen: A framework for distributed systems verification, with fault injection*. Jepsen.

[9] Ma, H. et al. 2019. I4: incremental inference of inductive invariants for verification of distributed protocols. *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, Oct. 2019), 370–384.

[10] Majumdar, R. and Niksic, F. 2017. Why is random testing effective for partition tolerance bugs? *Proceedings of the ACM on Programming Languages*. 2, POPL (Dec. 2017), 1–24. DOI:https://doi.org/10/gf9vnf.

[11] Newcombe, C. et al. 2015. How Amazon web services uses formal methods. *Communications of the ACM*.

[12] Ongaro, D. 2014. *Consensus: Bridging Theory and Practice*. Stanford University.

[13] Ongaro, D. and Ousterhout, J.K. 2014. In search of an understandable consensus algorithm. *USENIX Annual Technical Conference* (2014), 305–319.

[14] Vishwanath, D. 2018. *Validating Key-value Based Implementations of the Raft Consensus Algorithm for Distributed Systems*. San Jose State University.

[15] Woos, D. et al. 2016. Planning for change in a formal verification of the Raft consensus protocol. *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (New York, NY, USA, Jan. 2016), 154–165.

1. Byron Cook from Amazon Web Services uses this terminology to refer to formal methods techniques that produce useful results for engineers even with little invested effort. It might have been coined by Peter Alvaro in [1]. [↑](#footnote-ref-2)
2. We have not been able to find a TLA source file for this model, but [2] contains the model exported to PDF. [↑](#footnote-ref-3)