Automatic Compartmentalization of Distributed Protocols

Graduate

Abstract

Promises of better cost and scalability have driven the migration of systems to the cloud. Once systems become distributed, they must handle complications resulting from asynchrony, inconsistency, and failures. To resolve these complications, consensus protocols like Paxos [14] have been incorporated into distributed systems [6, 7, 13]. Consensus protocols are inherently complex and difficult to reason about, and they often become bottlenecks in practice. This has driven the design of scalable protocol variants [8, 10, 15, 16]. Unfortunately, these variants are even more intricate and often error-ridden [1, 12, 17, 18, 20].

Recent results suggest a simpler path to high-throughput consensus. Compartmentalized Paxos [21] "decouples" a complex consensus protocol into small, independent software *modules* that can be individually scaled. Then it identifies modules that are bottlenecks to throughput, and scales them up when possible. Comparmentalizing correctly in the context of a complex protocol like MultiPaxos [21] (or even more complex, Mencius [22]) required significant insight into their'inner workings.

Compartmentalized Paxos is Yet Another Consensus Protocol (YACP) that needs to be implemented and deployed. The goal of our work is to stop inventing protocols, and instead systematize the scalability ideas from Compartmentalized Paxos so they can be applied *automatically* to a wide variety of complex protocols, including transactional concurrency control, BFT, etc. Our vision of Automatic Compartmentalization proposes to increase throughput while preserving correctness and liveness, expanding the impact of compartmentalization to a broad range of programs. The end result, we believe, will be distributed systems that are both more elastic and more reliable.

—

There are two dimensions to the task of Automatic Compartmentalization: as a building block, we need to classify the scaling potential of individual modules. More holistically, we need to refactor monolithic code into modules that maximize the potential for scaling.

Our work distills from Compartmentalized Paxos three classes of modules in distributed protocols and techniques to classify them:

Embarassingly parallel modules can be scaled up and down arbitrarily. For example, relay nodes in PigPaxos [8] receive messages from the Paxos leader and broadcast them to followers; the number and choice of relay nodes is immaterial to correctness. We can classify modules for embarrassing parallelism via *monotonicity* analysis, as per the CALM theorem [11].

Key-partitioned modules can be partitioned and scaled based on a key in the request payload. For example, Compartmentalized Paxos shards acceptors into groups based on log index number; each group can execute commands independently of groups for other log indices. Independence between keys in a collection can be classified via functional dependency analysis [2].

Fundamentally sequential (single-threaded) modules cannot be scaled. These modules need to be small to avoid becoming a bottleneck. For example, the total-ordering of individual requests in Paxos is fundamentally unscalable. To maximize throughput, Corfu [5] singles out an individual sequencer whose sole purpose is to perform total-ordering.

To perform these analyses automatically, we require a DSL that allows us to automatically classify and refactor protocols. Logic programming [3, 4, 9] is a good fit because monotonicity and functional dependencies can be extracted easily [2, 3].

The challenge of refactoring a program into modules amenable to classifying remains. Typically, protocols are not factored based on their scalable modules; they are factored based on semantic units. For example, MultiPaxos has proposers and acceptors, but Compartmentalized Paxos refactors those modules for scalability. Hence we desire a language that allows code organization based on program semantics, while exposing features required for Automatic Compartmentalization. We plan to build on Dedalus [4] and Bloom [3] to include formalisms of physical machines and their (potentially dynamic) mapping to program modules.

Compartmentalization attempts to ensure correct behavior while scaling, but there are nuances that require further investigation. For example, protocol specifications like Paxos have semantic agents (e.g. a "proposer") that could be refactored into modules for scalability. After compartmentalization, these agents can exhibit partial failure of a module, and not the fail-stop behavior per agent as assumed in protocol correctness proofs [19]. In general, there are constraints on when certain optimization techniques can be applied, which requires further investigation into their formalisms.

Merely knowing *how* to apply optimizations is not enough; the optimizer must choose which portion of the program to optimize, given constraints and workloads. For example, Compartmentalized Paxos optimizes message replication at the expense of leader election, because it assumes that leader failures are rare. Profiling, bottleneck analysis, code generation, and automatic reconfiguration are all significant challenges.

We aim to ultimately combine these systems into a full pipeline such that users can write a distributed protocol, deployed with our system, that optimizes and redeploys on-the-fly to provide the best performance and cost for any given workload.

Preliminary results. We are currently working on classifying and refactoring programs implemented in Bloom. Our current analysis is able to arrive at Compartmentalized Paxos by manually applying individual steps of classifying and refactoring to a Bloom implementation of MultiPaxos. We are working to automate that process next.

References

[1] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J. Martin. Revisiting fast practical byzantine fault tolerance. *CoRR*, abs/1712.01367, 2017.

[2] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: Coordination analysis and placement for distributed programs. *ACM Trans. Database Syst.*, 42(4), Oct. 2017.

[3] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011.

[4] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in time and space. In O. de Moor, G. Gottlob, T. Furche, and A. Sellers, editors, *Datalog Reloaded*, pages 262–281, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[5] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis. CORFU: A shared log design for flash clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 1–14, San Jose, CA, Apr. 2012. USENIX Association.

[6] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):70–93, Jan. 2016.

[7] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 335–350, USA, 2006. USENIX Association.

[8] A. Charapko, A. Ailijiang, and M. Demirbas. PigPaxos: Devouring the communication bottlenecks in distributed consensus. In *Proceedings of the 2021 International Conference on Management of Data*. ACM, June 2021.

[9] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[10] C. Ding, D. Chu, E. Zhao, X. Li, L. Alvisi, and R. V. Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 325–338, Santa Clara, CA, Feb. 2020. USENIX Association.

[11] J. M. Hellerstein and P. Alvaro. Keeping CALM. *Communications of the ACM*, 63(9):72–81, Aug. 2020.

[12] H. Howard and I. Abraham. Raft does not guarantee liveness in the face of network faults. https://decentralizedthoughts.github.io/2020-12-12-raft-liveness-full-omission/, Dec 2020.

[13] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010.

[14] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[15] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for wans. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA, Dec. 2008. USENIX Association.

[16] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, Nov. 2013.

[17] D. Ongaro. *Consensus : bridging theory and practice*. PhD thesis, Stanford University, 2014.

[18] P. Sutra. On the correctness of egalitarian paxos. *Information Processing Letters*, 156:105901, 2020.

[19] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3), Feb. 2015.

[20] M. Whittaker. mwhittaker/craq_bug. https://github.com/mwhittaker/craq_bug, Jun 2020.

[21] M. Whittaker, A. Ailijiang, A. Charapko, M. Demirbas, N. Giridharan, J. M. Hellerstein, H. Howard, I. Stoica, and A. Szekeres. Scaling replicated state machines with compartmentalization. *arXiv preprint arXiv:2012.15762*, 2020.

[22] M. Whittaker, A. Ailijiang, A. Charapko, M. Demirbas, N. Giridharan, J. M. Hellerstein, H. Howard, I. Stoica, and A. Szekeres. Scaling replicated state machines with compartmentalization [technical report], 2021.