ACM SIGACT News Distributed Computing Column 35

Theory and Practice in Large Distributed Systems

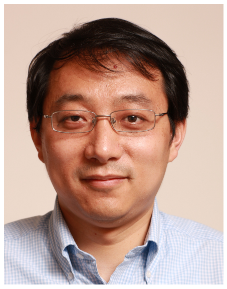
Idit Keidar
Dept. of Electrical Engineering, Technion
Haifa, 32000, Israel
idish@ee.technion.ac.il

In a follow-up on the theme of the previous Distributed Computing Column (SIGACT News 40(2), June 2009, pp. 67–95), which dealt with "Cloud Computing", the current column deals with the tension between theoretical results and practical deployments of large-scale distributed systems. It consists of a single contribution by Lidong Zhou of Microsoft Research Asia, who has experience on both sides of this divide– both in designing distributed algorithms and in building large-scale distributed systems. Many thanks to Lidong for his contribution!

Call for contributions: I welcome suggestions for material to include in this column, including news, reviews, opinions, open problems, tutorials and surveys, either exposing the community to new and interesting topics, or providing new insight on well-studied topics by organizing them in new ways.

Building Reliable Large-Scale Distributed Systems: When Theory Meets Practice

Lidong Zhou
Microsoft Research Asia
lidongz@microsoft.com



1 Introduction

Large-scale distributed systems are in the vogue with the increasing popularity of *cloud computing*, because a *cloud* is usually constructed from a large number of commodity machines. Due to the fail-prone nature of the commodity hardware, reliability has been a central piece in the design of those systems.

The theoretical distributed system community is *in theory* well prepared to embrace this (possibly a bit hyped) new era: the community started uncovering the fundamental principles and theories more than thirty years ago. The core *consensus* problem and the *replicated state-machine approach* [19, 32] based on consensus are directly applicable to the reliability problem in large-scale distributed systems.

Although some basic concepts and core algorithms (such as Paxos [20]) do find their way into the practical systems, the relevance of the research in the community does not appear as high as expected to the practical large-scale distributed systems that have been developed and deployed. There is undeniably a significant gap between theory and practice in this particular context.

This article reflects our rather biased personal perspectives on the gap between theory and practice, based on the limited personal experience of designing and building practical large-scale distributed systems, as well as on our attempts to bridge the gap. The hope is to shed lights on why theory and practice deviate from each other and to call for actions that would eventually bring harmony between the two.

2 Assumptions

Protocol correctness is relative to the underlying assumptions. Assumptions define the problem scope. A theoretical framework is confined within that scope. In practice, assumptions also define a boundary, within which the system is expected to behave correctly. The key difference is that, when building a practical system, it is our responsibility to ensure that the system stays within the boundary most of the time and that appropriate actions can be taken even if the system wanders outside of that boundary. While a theoretical framework focuses on the states within the boundary, the designer of a practical system pays more attention to those outside of the boundary.

A classic assumption for reliable distributed systems is to impose a threshold f on the number of concurrent failures allowed in a *replica group*, where a replica group consists of the set of servers implementing a replicated state machine. A practical system cannot take such an assumption for granted. For long-term reliability, fast recovery from failures becomes crucial in upholding this assumption. This helps explain the discrepancies related to the importance of recovery and reconfiguration: in theory, the issue receives relatively little attention, while in practice replica-group *reconfiguration* after replica failures has always been a key focus. For example, Brewer [3] made a case for the importance of mean-time-to-repair (MTTR) on system availability based on his observations on and experiences with giant-scale Internet services in the late 90s. (Examples of those servers include the infrastructure for the Inktomi search engine.) More recently, the Google File System [12] employs a replica placement design that enables fast and parallel replica recreation in response to failures, further validating the significance of fast repair to reliability of large-scale storage systems.

Even with protocols that have been proven correct, a practical system offers only best-effort guarantees [16] because guarantees are relative to the underlying assumptions. In a large-scale distributed system that is expected to run for years, no rare events can be safely ignored: such events eventually show up in the system, often violating assumptions that are required for reasoning about correctness and for ensuring reasonable system performance. Therefore, *graceful degradation* becomes an important notion in practical systems, as advocated by Brewer [3] and Hamilton [13], although the arguments were mostly focusing on degradation in performance or functionality. From a correctness point of view, it is also possible to offer different levels of guarantees under different circumstances in order to enable graceful degradation [14, 38]. Fail-awareness [11] represents another viable approach that embraces the same principle, where a system is expected to perform correctly in normal cases, while emitting an exception signal to affected clients when experiencing "performance failures" or safety violations. System designs embracing this concept have recently emerged [6], where the service detects safety violations and also provides graceful degradation at the semantics level: whenever the server is correct, stronger semantics are ensured; if the server is faulty, only weaker semantics are ensured, and failure notifications are eventually received.

Because degradations could occur in terms of performance, functionality, or semantics (e.g., consistency), different systems might make different choices. The Paxos protocol implicitly chooses to preserve the safety properties (e.g., consistency of the value chosen on different replicas) at all times: when experiencing more failures than it is designed to tolerate or remains in an adverse asynchronous environment, the system stops making progress, but never violates the safety properties of consensus.

3 Simplicity

Simplicity is a widely accepted guideline, both in theory and in practice. Interestingly, the different interpretations of simplicity are source of disconnection between the two. In a theoretical study, simplicity tends to manifest itself through abstractions and generalizations, while removing all unnecessary details.

The Paxos protocol [20] is a classic example: it is beautifully (and deceivingly) simple. The Paxos protocol implements a replicated state machine by executing a series of consensus instances with continuously increasing instance numbers, where the consensus decision for instance i becomes the i^{th} command for the replicated state machine. In each consensus instance, processes designated as *leaders* and *acceptors* execute a two-phase protocol to choose a consensus value for that instance. Processes assigned the role of *learners* learn the consensus values to drive transitions in the state machine.

The two-phase protocol between a single leader and the acceptors is proven to ensure the safety properties of consensus; for example, it ensures that only a single value is chosen. In the first phase, a new leader declares a new *ballot* and collects all the proposed values with a lower ballot from the acceptors. In the second phase, depending on the responses from the first phase, the leader might choose to propose a value obtained in the first phase or a new value from a client. Each phase involves a *quorum* of acceptors; any pair of quorums intersects. The Paxos protocol captures the key elements needed for consensus succinctly, while leaving other "non-essential" details unspecified. For example, liveness conditions hinge on a *leader election* component, but no algorithm is given. In practice, liveness conditions correspond directly to system performance, often at the center of attention for practical systems and tricky to implement due to handling of failures. It is not surprising to see leader-election related bugs in our recent effort [37] of model-checking a production Paxos implementation.

The Paxos protocol further makes simplifying assumptions that are unrealistic. For example, acceptors are required to keep track of proposed values for every consensus instance. The Paxos protocol conveniently assumes that each *acceptor* maintains an infinite log to record proposed values in all instances. In practice, the chosen commands must be learned and applied, with log entries garbage-collected periodically. This seemingly simple engineering detail introduces significant complexity in the implementation, as reported in other efforts of implementing the Paxos protocol for deployed systems [9].

For practical systems, simple designs are favorable [13], especially in a large-scale distributed system, even at the expense of reduced performance. Those design choices sometimes cause the implementation to deviate from the proven protocols. For example, the Paxos implementation we studied [37] embraces an interesting principle of "When in doubt, reboot", to simplify handling of corner cases that occur rarely: a replica always returns to a well-defined initial state after rebooting and this is the only recovery situation for the protocol to handle correctly. The aggressive rebooting strategy is not without cost: it can potentially cause performance hiccups and could even reduce the resilience temporarily.

Practical systems tend to prefer simple solutions over general, but complicated, ones. An interesting example concerns replica-group reconfiguration in Paxos. The standard Paxos protocol assumes a fixed set of acceptors. When acceptors fail, new acceptors might need to be added to replace the faulty ones to maintain the desirable level of resilience. Ideally, reconfiguration is done with no interruptions to users; that is, the system should continue accepting and choosing new values.

One approach [20] is to introduce reconfiguration as a command to the state machine and therefore a value to be proposed in any consensus instance. Because a consensus instance cannot be activated until the set of acceptors is known, there is an inherent tension between the level of concurrency among consensus instances and how fast a reconfiguration command takes effect. If a reconfiguration command is chosen as the consensus value for the i^{th} instance and it takes effect in the next instance $i+1$, then the replicated state machine cannot activate consensus instance $i+1$ until a consensus decision is reached at instance i. This is because the consensus decision at i could be a reconfiguration command, which will change the set of acceptors for instance $i+1$.

Lamport introduces a parameter α to address this tension: having a reconfiguration command chosen at instance i to take effect starting from instance $i+\alpha$ allows up to α consensus instances to be activated concurrently. Implementing the general protocol for any value of α can be tricky and complicated [25]. Setting α to 1 significantly simplifies the protocol at the expense of reducing the level of concurrency. In some practical systems (e.g., [10, 5, 28, 35, 27]), the Paxos protocol is used to implement a reliable central service for maintaining global configuration and for facilitating reconfigurations of individual replica groups in the system. In those settings, concurrency of consensus instances is arguably less important, making the simplification attractive. Furthermore, *batching* can be used to improve service throughput by proposing a combined value for all the commands received during the processing of a previous instance. (Interestingly, the best practice might be setting α to ∞, leading to a two-dimensional space for instance numbers, according to personal communication with Leslie Lamport.)

4 Taking a Global View

Traditional research on consensus and replicated state machines tends to focus on a single replica group. This seems sensible given that each replica group is expected to run the same protocol. However, in practice, taking a global view could offer more attractive architectural choices, especially for large-scale distributed storage systems. For example, the Paxos protocol requires $2f + 1$ acceptors in order to tolerate f failures. However, with a global service in many practical systems, each replica group can get away with $f+1$ servers to tolerate f failures, using primary/backup replication [23, 33]. The global service is usually implemented using Paxos and is called upon during replica-group reconfigurations. Furthermore, global placement of replicas on servers is important in practice: it directly affects recovery from failures and load balance. Interestingly, practical solutions [26] to the placement problem manage to find a theoretical underpinning: the classic bins-and-balls problem.

The focus on individual replica groups also fails to capture important global interplays across groups. For a large-scale distributed system, the system dynamics in response to changes such as failures, incremental expansion, and load balancing are particularly hard to predict and cope with: they can easily evolve into emergent mis-behavior [29]. For example, when a replica in a replica group fails, from a reliability point of view, it is ideal to build up a new replica to replace the failed one before more failures cause data losses. However, building up a new replica requires *state transfer* from available replicas to the new replica, which is often expensive and incurs significant network traffic. Because failure detection often depends on beacons and is not perfect, any temporary networking problem could cause false suspicions. If the system adds new replicas too aggressively, it might exacerbate the networking problem and cause more replicas to be classified as faulty incorrectly, eventually bringing down the entire system.

5 From Practice to Theory, and Back to Practice

The gap between theory and practice will always exist due to their different focuses and values. But their evolutions should go hand in hand. Practical systems evolve with the advances in underlying hardware and with emerging new applications and requirements. New theoretical foundations naturally arise from practical challenges in building those real systems.

For example, chain replication [35] is a new replication protocol that achieves both high throughput and availability in a large-scale distributed system. Replicas in a replica group form a one-dimensional chain and the protocol messages for an update flow through the chain forward (from *head* to *tail*) and then backward. Any reconfiguration as a result of replica failures or additions involves a central service and preserves the chain structure. The protocol as presented [35] is already a result of proper abstraction. The protocol is further formalized and verified with the Nuprl library [1].

The process of identifying key elements of practical systems and framing them in an interesting theoretical framework could often in turn lead to better understanding of practical systems and provide guidance to building better systems. For example, large-scale distributed storage systems tend to use a primary/backup protocol for data replication, where all requests go to the *primary* and the primary has all updates committed on all the secondaries. In our attempt [22] to understand the correctness of such protocols, we have made a connection of those protocols to the Paxos protocol (i) by allowing multiple different configurations (in terms of the set of acceptors and its quorum structure) to be associated with each ballot and (ii) by distinguishing *read quorums* required in the first phase of Paxos from the *write quorums* required in the second phase of Paxos. The result is a generalization of the primary/backup protocols used in practice: in a primary/backup protocol, each reconfiguration creates a new configuration with its version as the corresponding new ballot, with a centralized service maintaining the mapping from the ballots to the corresponding configurations; primary/backup protocols use a specific, and probably the most interesting, read/write quorum construction, where all replicas form the single write quorum and each replica constitutes a read quorum.

Besides helping establish correctness of practical primary/backup protocols, the generalization further leads to improvements on existing systems. Current systems usually impose the requirement that a single configuration is active for each replica group, which creates tension between reliability and availability. Because the write quorum consists of all the replicas in the replica group, failure of any replica could prevent the group from accepting new updates to the data it maintains. Reconfiguration is therefore necessary for providing high availability.

The traditional approach is to remove faulty replica(s) from the group: this reduces the level of resilience because new updates are now committed on a smaller number of replicas than in the ideal setting. An alternative is to replace faulty replicas with new ones. However, in the traditional primary/backup protocol, state transfer to new replicas must be completed before new replicas can be added to a replica set. State transfer tends to be expensive for a replica group maintaining a large amount of data; the replica group is unavailable before state transfer and reconfiguration is completed.

Our generalization decouples state transfer from reconfiguration by allowing multiple concurrent configurations to be active, thereby removing the tension between availability and reliability. As a result of removing the constraint of a single active configuration, a system designer now has more options to play with and can choose the most appropriate ones. For example, when a replica r fails in a replica group with current configuration C, a new configuration C' is created with a new replica r' added to replace r. The new configuration therefore maintains the desirable level of resilience. Because old configurations remain in the system, new updates can be committed to the new configuration immediately, while state transfer from the old configurations can be carried out concurrently, if r remains faulty. In cases where r recovers promptly, it might actually be desirable to create a newer configuration as C (but with a higher ballot) to recruit r back to the replica group to minimize the amount of state transfer, assuming that r has accumulated more updates than r'.

6 Moving Systems Closer to the Specifications

The gap between theory and practice also manifests itself in the distance between specification and implementation. In theory, algorithms and protocols are specified formally and proven correct. But the process of turning specifications into implementations is often considered as engineering and remains largely an art, rather than a science.

There are at least three ways to bring systems and specifications closer together. The first is through software verification, which aims at checking the system correctness automatically. Klein [18] reviews the historical landscape in formal machine-checked software verification and reports the latest developments in this direction; in particular, the L4.verified project is on the promising path towards a fully verified OS kernel. The second approach is to generate an (efficient) system implementation from a specification. Quite a few specification languages (e.g., Lotos [2], Estelle [4], SDL [34], PROMELA [15], and TLA+ [21]) have been proposed. Instances of such translations have also been reported [8, 36] in some limited domains. Some form of automatic code generation has also been reported in the development of a practical system based on

Paxos [9]. Finally, both complete verification and automatic translation are ambitious goals, especially for large and complex distributed systems. Various language and tool support for large-scale distributed systems have also emerged. Those languages and tools target testing, debugging, and checking of complex systems. Some recent examples include KLEE [7], Pip [31], Mace [17], D^{3}S [24], CHESS [30], and Modist [37]. All of these directions have shown promise, but more work is clearly needed.

Acknowledgments

This article has benefited tremendously from discussions with Idit Keidar, Leslie Lamport, Dahlia Malkhi, Fred B. Schneider, Robbert van Renesse, Chandramohan A. Thekkath, Ming Wu, Mao Yang, and my colleagues at Microsoft Research Silicon Valley and in the System Research Group at Microsoft Research Asia, as well as from experience of working with development and testing teams responsible for distributed systems behind various on-line services, such as Hotmail and the Bing search engine.

References

[1] Mark Bickford and David Guaspari. Formalizing the chain replication protocol. http://www.cs.cornell.edu/Info/Projects/NuPrl/ /FDLcontentAUXdocs/ChainRepl, September 2006.

[2] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[3] Eric A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, July/August 2001.

[4] Stanislaw Budkowski and Piotr Dembinski. An introduction to Estelle: a specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1):3–23, 1987.

[5] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[6] Christian Cachin, Idit Keidar, and Alexander Shraer. Fail-aware untrusted storage. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2009.

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, 2008.

[8] Claude Castelluccia, Walid Dabbous, and Sean O'Malley. Generating efficient protocol code from an abstract specification. In *IEEE/ACM Transactions on Networking*, pages 514–525, 1997.

[9] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In Indranil Gupta and Roger Wattenhofer, editors, *Proceedings of the Symposium on Principles of Distributed Computing (PODC 2007)*, Portland, Oregon, USA, August 2007. ACM.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, M. Burrows D. A. Wallach, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the Seventh Symposium on Operating System Design and Implementation (OSDI 2006)*, November 2006.

[11] Christoph Fetzer, Christof Fetzer, and King Richard H. Fail-awareness in timed asynchronous systems. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC 1996)*, pages 314–321. ACM, 1996.

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating Systems Principles (SOSP 2003)*, pages 29–43, New York, NY, USA, 2003. ACM.

[13] James Hamilton. On designing and deploying internet-scale services. In *LISA*, 2007.

[14] Maurice. P. Herlihy and Jeannette M. Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, July 1991.

[15] Gerard J. Holzmann. *SPIN Model Checker, The Primer and Reference Manual*. Addison-Wesley, September 2003.

[16] Michael Isard. Autopilot: Automatic data center management. *Operating Systems Review*, 41(2):60–67, April 2007.

[17] Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. Mace: Language support for building distributed systems. In *ACM SIGPLAN Conference onProgramming Language Design and Implementation (PODI 2007)*, 2007.

[18] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, February 2009.

[19] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[20] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[21] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, July 2002.

[22] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and primary-backup replication. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC 2009)*. ACM, August 2009. Brief Announcement.

[23] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles (SOSP 1991)*, pages 226–238. ACM, 1991.

[24] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D^{3}S: Debugging deployed distributed systems. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI 2008)*, 2008.

[25] Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The SMART way to migrate replicated stateful services. In *Proceedings of the 2006 EuroSys Conference*, Leuven, Belgium, April 2006. ACM.

[26] John MacCormick, Nicholas Murphy, Venugopalan Ramasubramanian, Udi Wieder, Junfeng Yang, and Lidong Zhou. Kinesis: A new approach to replica placement in distributed storage systems. *ACM Transactions on Storage (TOS)*, 4(4), 2009.

[27] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *OSDI'04: Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation*, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.

[28] John MacCormick, Chandramohan A. Thekkath, Marcus Jager, Kristof Roomp, Lidong Zhou, and Ryan Peterson. Niobel a practical replication protocol. *ACM Transactions on Storage (TOS)*, 3(4), February 2008.

[29] Jeff C. Mogul. Emergent (mis)behavior vs. complex software systems. *Operating Systems Review*, 40(4):293–304, 2006.

[30] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[31] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, 2006.

[32] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[33] Garret Swart, , Andrew Birrell, Andy Hisgen, and Timothy Mann. Availability in the Echo file system. Research Report 112, System Research Center, Digital Equipment Corporation, September 1993.

[34] Kenneth J. Turner. *Using Formal Description Techniques: An Introduction to Estelle, Lotos, and SDL.* John Wiley & Sons, Inc., New York, NY, USA, 1993.

[35] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 91–104, Berkeley, CA, USA, 2004. USENIX Association.

[36] S. T. Vuong, A. C. Lau, and R. I. Chan. Semiautomatic implementation of protocols using an estelle-c compiler. *IEEE Trans. Softw. Eng.*, 14(3):384–393, 1988.

[37] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2009)*, Boston, MA, USA, April 2009. USENIX.

[38] Lidong Zhou, Vijayan Prabhakaran, Venugopalan Ramasubramanian, Roy Levin, and Chandramohan A. Thekkath. Graceful degradation via versions: specifications and implementations. In Indranil Gupta and Roger Wattenhofer, editors, *Proceedings of the Symposium on Principles of Distributed Computing (PODC 2007)*, pages 264–273, Portland, Oregon, USA, August 2007. ACM.