Tutorial: Consensus Algorithms from Classical to Blockchain: Quickly Program, Configure, Run, and Check

Yanhong A. Liu
Stony Brook University
liu@cs.stonybrook.edu

Scott D. Stoller
Stony Brook University
stoller@cs.stonybrook.edu

***Abstract*—This tutorial examines well-known algorithms for distributed consensus problems, from classical consensus to blockchain consensus. We discuss exact algorithms that are high-level as in pseudocode and directly executable as programs at the same time, focusing on how to quickly program, configure, run, and check these algorithms as well as distributed algorithms and systems in general.**

I. Introduction

Distributed consensus is at the core of distributed systems and services, which are increasingly indispensable in daily life. There is an increasing interest in algorithms for both classical consensus and blockchain consensus, especially the exact algorithms for deeper understanding and possible improvements by researchers, practitioners, and students.

Especially desired is the ability to quickly program, configure, run, and check these algorithms, through methods and languages for expressing distributed algorithms and protocols precisely at a very high level, to facilitate the design and implementation, as well as analysis and verification, of these algorithms and protocols.

This tutorial examines well-known algorithms for distributed consensus problems, from classical consensus to Byzantine consensus to blockchain consensus. We discuss exact algorithms that are high-level as in pseudocode and directly executable at the same time, focusing on how to quickly program, configure, run, and check these algorithms as well as distributed algorithms and systems in general. The tutorial consists of five parts:

1) An introduction to different distributed consensus problems, from classical consensus to Byzantine consensus to blockchain consensus.
2) An overview of well-known algorithms, from Paxos for classical consensus to the Bitcoin algorithm for blockchain consensus, including important variants such as Viewstamped Replication and Virtual Synchrony, as well as Proof-of-Stake vs. Proof-of-Work and Byzantine fault tolerance (BFT) protocols.
3) An overview of a method and language, DistAlgo, for expressing distributed algorithms precisely at a high-level as pseudocode and having them be directly executable at the same time, and be easily configured and checked for safety and liveness properties.
4) A study of exact algorithms expressed at a high level for the most extensively studied algorithm variants, including Lamport's Paxos for classical consensus, Nakamoto's Bitcoin algorithm for blockchain consensus, and more recent BFT protocols, focusing on their exact programming, configuration, execution, and checking.
5) A demo of direct execution and checking of these algorithms running as distributed processes.

A closely related tutorial was given at PODC 2019 (on PODC'19 page: http://www.podc.org/podc2019/workshops-and-tutorials/tutorial-descriptions; in PODC'19 proceedings: https://dl.acm.org/doi/10.1145/3293611.3338022).[1] This new tutorial focuses on ease of programming, configuration, running, and checking, and will include a variety of new and old concensus algorithms, especially including BFT algorithms.

II. Distributed consensus

Distributed consensus is the problem of having a set of distributed processes—also knowns as servers, parties, players, agents, entities, etc.—agree on a single value or a continuing sequence of values in the presence of faulty processes and communication channels.

- Classical consensus considers benign failures where processes may crash and may later recover, and where messages may be lost, delayed, reordered, and duplicated.
- Byzantine consensus considers arbitrary failures, where processes may also perform malicious actions, such as sending arbitrary messages.
- Blockchain consensus further considers Sybil attacks where a system is subverted by forging process identities.

Different consensus problems are for very different application senarios.

- Classical consensus is essential for distributed services using replications to provide fault tolerance, where replicas must agree on the system state or the sequence of system actions. This is by far the vast majority of uses of consensus protocols.

[1]The Google site created for the tutorial, containing programs, specifications, and proofs, became unavailable last year when Google stopped supporting advanced features we used for the site, but the tutorial slides are available at https://distalgo.cs.stonybrook.edu/tutorial.

- Byzantine consensus is needed for fault tolerance against potentially malicious processes, such as services involving external entities, or processes being taken over by intruders.
- Blockchain consensus is used for transactions involving entities that are totally anonymous and where all of them can be malicious.

Because failures are inevitable, consensus algorithms have to pay to be fault tolerant. Generally, blockchain consensus is drastically more costly than Byzantine consensus, and Byzantine consensus is significantly more costly than classical consensus.

The pursuit of consensus algorithm design is exactly to minimize the cost to provide maximum reliability. As a result, efficient consensus algorithms are highly nontrivial and challenging to understand and to design.

III. DistAlgo language

DistAlgo is developed exactly for easy and clear expression of complex distributed algorithms—expressing them at a high level as with pseudocode languages, but precisely as with formal specification languages, and directly executable as with programming languages.

For expressing distributed algorithms at a high level, DistAlgo supports four key concepts:

1) distributed processes that can send messages,
2) high-level control flows for handling received messages, both synchronously and asynchronously,
3) high-level queries for synchronization conditions, and
4) declarative configuration for setting up and running.

DistAlgo minimally extends the Python programming language to support these concepts.

- DistAlgo has been used to express a wide variety of important distributed algorithms, including over 20 well-known algorithms and variants for classical consensus, Byzantine consensus, and blockchain consensus.
- DistAlgo has been used in over 100 different course and research projects at Stony Brook and elsewhere, including by high school students, implementing the core of network protocols, distributed graph algorithms, distributed coordination services, distributed hash tables, distributed file systems, distributed databases, parallel processing platforms, security protocols, and more.
- The algorithms and systems can be programmed much more easily and clearly compared to using conventional programming languages, e.g., in 20 lines instead of 200 lines, or 300 lines instead of 3000 lines or more.

DistAlgo is open-source (https://github.com/DistAlgo/distalgo; it is also being extended to support Datalog rules and more for advanced analysis and optimizations https://github.com/DistAlgo/alda).

IV. Concensus algorithms: quickly program, configure, run, and check

The tutorial content is based on our studies of distributed consensus algorithms, e.g., [1]–[4], especially by using the DistAlgo language [5], [6] that has enabled the study of many more algorithms and protocols, e.g., [7]–[9]. Students in various course projects and research projects have implemented essentially all interesting distributed agorithms we could find, including a wide variety of Byzantine consensus protocols from oldest to newest. The most relevant references are:

- From Clarity to Efficiency for Distributed Algorithms [5] (OOPSLA 2012) — DistAlgo language, compilation, optimization, implementation, and experiments with a dozen examples including Basic Paxos and Byzantine Paxos
- High-Level Executable Specifications of Distributed Algorithms [7] (SSS 2012) — Methods for writing high-level specifications, with parts of Multi-Paxos as a main example. Best Student Paper Award
- From Clarity to Efficiency for Distributed Algorithms [6] (TOPLAS 39(3) 2017) — Extended description of DistAlgo language and optimization method, with a formal operational semantics
- Moderately Complex Paxos Made Simple: High-Level Specification of Distributed Algorithms [2] (PPDP 2019) — Basic Paxos and Multi-Paxos algorithms for distributed consensus written at a high level in DistAlgo
- What's Live? Understanding Distributed Consensus [10], long version of [3] (PODC 2021) with a video — Specifications of a wide range of liveness properties for over 30 consensus algorithms and variants
- Specification and Runtime Checking of Derecho, A Protocol for Fast Replication for Cloud Services [4] (APPLIED 2023 at PODC 2023) — A complete specification that is directly executable, after fixing some issues discovered in the pseudocode

We have not written about our studies of algorithms for Byzantine consensus and blockchain consensus for publication, but our results have been used in continued teaching and research. For example, exact algorithms for Bitcoin backbone were discussed in the PODC 2019 tutorial [11] (with slides at https://distalgo.cs.stonybrook.edu/tutorial).

The tutorial introduces consensus problems, the DistAlgo language, and consensus algorithms at a level accessible to a broad audience:

- The tutorial is for all of researchers, practitioners, and students interested in programming, configuring, running, and checking distributed algorithms and systems.
- At the same time, the tutorial is aimed to be accessible to anyone who has basic algorithm background and programming skills.
- The tutorial may also be of interest to people interested in verification of distributed algorithms, because high-level programming allows properties of programs to be much more easily proved.
- The tutorial may be of interest to compiler builders, because high-level programming creates more opportunities for optimization and code generation.

For complex distributed algorithms, especially consensus algorithms, our experience is that precise high-level executable

```
process Proposer:
  def setup(acceptors):                                  # take in set of acceptors
    self.majority := acceptors                           # any majority of acceptors; we use all
  def run():
    n := self                                                    # Phase 1a: select proposal num n
    send ('prepare',n) to majority                               #  send prepare n to majority
    await count {a: received ('respond',=n,_) from a}            # Phase 2a: wait for response to n
          > (count acceptors)/2:                                 #  from a majority of acceptors
      v := any ({v: received ('respond',=n,(n2,v)),              #  find val in accepted prop, if any
                 n2 = max {n2: received ('respond',=n,(n2,_))} } # having max prop num
               or {any 1..100})                                  #  or any value; use any in 1..100
      responded := {a: received ('respond',=n,_) from a}         #  find responded
      send ('accept',n,v) to responded                           #  send accept for proposal n,v
process Acceptor:
  def setup(learners): pass                              # take in set of learners
  def run(): await false                                 # wait for nothing, only to handle messages
  receive ('prepare',n) from p:                                  # Phase 1b: receive prepare n
    if each sent ('respond',n2,_) has n > n2:                    #  if n > each responded n2
      max_prop := any {(n,v): sent ('accepted',n,v),             #  find accepted proposal, if any,
                           n = max {n: sent ('accepted',n,_)} }  # having max prop num
      send ('respond',n,max_prop) to p                           #  respond with n,max_prop
  receive ('accept',n,v):                                        # Phase 2b: receive proposal n,v
    if not some sent ('respond',n2,_) has n2 > n:                #  if not responded with larger n2
      send ('accepted',n,v) to learners                          #  send accepted proposal n,v
process Learner:
  def setup(acceptors): pass                             # take in set of acceptors
  def run():
    await some received ('accepted',n,v) has                     # wait for some accepted proposal
              count {a: received ('accepted',=n,=v) from a}      # that has been accepted by
              > (count acceptors)/2:                             # a majority of acceptors
      output('chosen',v)                                 # output chosen value v
def main():
  acceptors := 3 new Acceptor                            # create 3 Acceptor processes
  proposers := 3 new Proposer(acceptors)                 # create 3 Proposer procs, pass in acceptors
  learners := 3 new Learner(acceptors)                   # create 3 Learner procs, pass in acceptors
  acceptors.setup(learners)                              # to acceptors, pass in learners
  (acceptors + proposers + learners).start()             # start acceptors, proposers, learners
```

Fig. 1. A high-level specification of Basic Paxos in DistAlgo, including setting up and running 3 each of Proposer, Acceptor, and Learner processes and outputting the result.

specifications are critical. They have allowed us to understand the algorithms drastically better and then to improve both correctness and efficiency much more easily than was possible before.

- For example, for Multi-Paxos, our DistAlgo specification improves over the original pseudocode in several ways and also led us to easily discover liveness violations when messages can be lost [2].
- For complex algorithms, high-level specification and execution of checkers for safety and liveness properties, as enabled by the power of DistAlgo [12], help significantly too, e.g., in finding buffer overwriting and leader-election deadlock in Derecho pseudocode [4].

In fact, for almost all algorithms we have studied, we have found issues and improvements not known before, even for simple algorithms, e.g., [8].

V. Example: Basic Paxos

Figure 1 shows Lamport's Basic Paxos in DistAlgo's ideal syntax, as specified in [2]. The comments aligned with the right # signs paraphrase Lamport's English description. Figure 2 shows the same specification in DistAlgo's Python syntax.

VI. Biographies

Y. Annie Liu is a Professor of Computer Science at Stony Brook University. She received her B,S. from Peking University, M.Eng. from Tsinghua University, and Ph.D. from Cornell University, all in Computer Science. Her primary research is in languages and algorithms, and focuses specially on systematic methods for design and optimizations. The methods are centered around incrementalization—the discrete counterpart of differentiation in calculus. She has strong other interests in interactive environments, real-time and embedded systems, database, knowledge representation and reasoning, distributed systems, and security. She led the development of DistAlgo language and compiler for the precise specification, optimization, and checking of distributed algorithms, especially including consensus algorithms. She has published in many prestigious venues, taught in a wide range of Computer Science areas, and presented over 140 conference and invited talks worldwide.

Scott D. Stoller is a Professor of Computer Science at Stony Brook University. He received his B.S. in Physics, *summa cum laude*, from Princeton University and his Ph.D. in Computer Science from Cornell University. His primary research areas are distributed systems, cybersecurity, languages, and cyber-

```
def anyof(s): return next(iter(s)) if s else None
class Proposer(process):
  def setup(acceptors):
    self.majority = acceptors
  def run():
    n = self
    send(('prepare', n), to= majority)
    if await(len(setof(a, received(('respond', _n, _),
                                   from_= a)))
             > len(acceptors)/2):
      v = anyof(setof(v, received(('respond', _n, (n2,v))),
                      n2==max(setof(n2, received(('respond',
                                                  _n,
                                                  (n2,_))))))
                or {randint(1,100)})
      responded = setof(a, received(('respond', _n, _),
                                    from_= a))
      send(('accept', n, v), to= responded)
class Acceptor(process):
  def setup(learners): pass
  def run(): await(False)
  def receive(msg= ('prepare', n), from_= p):
    if each(sent(('respond', n2, _)), has= n > n2):
      max_prop = anyof(setof((n,v), sent(('accepted', n, v)),
                             n==max(setof(n, sent(('accepted',
                                                   n, _))))))
      send(('respond', n, max_prop), to= p)
  def receive(msg= ('accept', n, v)):
    if not some(sent(('respond', n2, _)), has= n2 > n):
      send(('accepted', n, v), to= learners)
class Learner(process):
  def setup(acceptors): pass
  def run():
    if await(some(received(('accepted', n, v)), has=
                  len(setof(a, received(('accepted', _n, _v),
                                        from_= a)))
                  > len(acceptors)/2)):
      output('chosen', n, v)
def main():
  acceptors = new(Acceptor, num= 3)
  proposers = new(Proposer, (acceptors,), num= 3)
  learners = new(Learner, (acceptors,), num= 3)
  setup(acceptors, (learners,))
  start(proposers | acceptors | learners)
```

Fig. 2. Basic Paxos in DistAlgo's Python syntax.

physical systems. He received an NSF CAREER Award, an ONR Young Investigator Award, a NASA Turning Goals Into Reality Award for Engineering Innovation, Best Paper Awards at Haifa Verification 2005, Runtime Verification 2011, and Data and Applications Security and Privacy 2016. He taught a graduate-level Distributed Systems class, covering theoretical algorithms and practical system design, to over 400 students.

VII. Acknowledgments

We thank Leslie Lamport, Robbert van Renesse, Yair Amir, and Ken Birman for their clear explanations and helpful discussions about various consensus algorithms. We thank Bo Lin for his robust DistAlgo compiler with excellent support, and Yi Tong for her excellent upgrade to new Python versions while extending DistAlgo with logic rules. We thank hundreds of students in distributed algorithms and distributed systems courses and projects for extending, developing variants of, testing, evaluating, and model checking our executable specifications, including running them on distributed machines, in the cloud, etc.

This material is based upon work supported in part by the National Science Foundation under Grants CCF-1954837, CCF 1414078, CCF 1248184, and CNS 1421893, and Office of Naval Research under Grants N000142112719, N000142012751, and N000141512208. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] S. Chand, Y. A. Liu, and S. D. Stoller, "Formal verification of Multi-Paxos for distributed consensus," in *Proceedings of the 21st International Symposium on Formal Methods*, ser. LNCS, vol. 9995. Springer, Nov. 2016, pp. 119–136. [Online]. Available: https://doi.org/10.1007/978-3-319-48989-6_8

[2] Y. A. Liu, S. Chand, and S. D. Stoller, "Moderately complex Paxos made simple: High-level executable specification of distributed algorithm," in *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*. ACM Press, Oct. 2019, pp. 15:1–15:15. [Online]. Available: https://doi.org/10.1145/3354166.3354180

[3] S. Chand and Y. A. Liu, "Brief announcement: What's live? understanding distributed consensus," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. ACM Press, July 2021, pp. 565–568. [Online]. Available: https://doi.org/10.1145/3465084.3467947

[4] K. Shivam, V. Paladugu, and Y. A. Liu, "Specification and runtime checking of derecho, a protocol for fast replication for cloud services," in *Proceedings of the Workshop on Advanced Tools, Programming Languages, and Platforms for Implementing and Evaluating Algorithms for Distributed Systems*. ACM Press, 2023. [Online]. Available: https://doi.org/10.1145/3584684.3597275

[5] Y. A. Liu, S. D. Stoller, B. Lin, and M. Gorbovitski, "From clarity to efficiency for distributed algorithms," in *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2012, pp. 395–410. [Online]. Available: https://doi.org/10.1145/2384616.2384645

[6] Y. A. Liu, S. D. Stoller, and B. Lin, "From clarity to efficiency for distributed algorithms," *ACM Transactions on Programming Languages and Systems*, vol. 39, no. 3, pp. 12:1–12:41, May 2017. [Online]. Available: https://doi.org/10.1145/2994595

[7] ——, "High-level executable specifications of distributed algorithms," in *Proceedings of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, 2012, pp. 95–110. [Online]. Available: https://doi.org/10.1007/978-3-642-33536-5_11

[8] Y. A. Liu, "Logical clocks are not fair: What is fair? A case study of high-level language and optimization," in *Proceedings of the Workshop on Advanced Tools, Programming Languages, and Platforms for Implementing and Evaluating Algorithms for Distributed Systems*. ACM Press, 2018, pp. 21–27. [Online]. Available: https://doi.org/10.1145/3231104.3231109

[9] C. Kane, B. Lin, S. Chand, S. D. Stoller, and Y. A. Liu, "High-level cryptographic abstractions," in *Proceedings of the ACM SIGSAC 14th Workshop on Programming Languages and Analysis for Security*. London, U.K.: ACM Press, Nov. 2019, pp. 31–43. [Online]. Available: https://doi.org/10.1145/3338504.3357343

[10] S. Chand and Y. A. Liu, "What's live? understanding distributed consensus," *Computing Research Repository*, vol. cs.DC, no. arXiv:2001.04787, Jan. 2020. [Online]. Available: https://doi.org/10.48550/arXiv.2001.04787

[11] Y. A. Liu and S. D. Stoller, "From classical to blockchain consensus: What are the exact algorithms?" in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. ACM Press, July-Aug. 2019, pp. 544–545. [Online]. Available: https://doi.org/10.1145/3293611.3338022

[12] ——, "Assurance of distributed algorithms and systems: Runtime checking of safety and liveness," in *Proceedings of the 20th International Conference on Runtime Verification*, ser. LNCS, vol. 12399. Springer, Oct. 2020, pp. 47–66. [Online]. Available: https://doi.org/10.1007/978-3-030-60508-7_3