In Search of Understandable Distributed Systems: Revisiting the Raft Vision in the Vibe Coding Era

Amy Babay

School of Computing and Information
University of Pittsburgh

University of Pittsburgh

Where are we going in this talk?

- **Understandable distributed systems**: ones that human developers can reason about, be confident are correct, and modify (while maintaining correctness)

- **Raft**: Consensus protocol introduced in 2014, claiming to be more understandable than the classic Paxos algorithm
- **Vibe Coding**: Building systems via natural language prompts to generative AI tools

Why? What will we be working on in the next 5 years? (a personal view)

- Continuing to secure / modernize legacy infrastructure
  - **Intrusion tolerance** offers solutions that can withstand stealthy, zero-day attacks (but makes some assumptions that need to be supported)
  - Ability to **upgrade** infrastructure becomes crucial as cyberattacks become easier to discover and execute
  - Regulation-driven sectors require **policy solutions**, not only technical
- Trying to "get it right" for emerging applications
  - Dynamic **spectrum sharing**, **quantum** internet, …
- Trying to keep up with the next generation of "legacy" systems
  - As new systems are being rapidly built and deployed, how do we ensure long-term reliability and maintainability?
  - As AI dramatically lowers the barrier to creating software, does it lower the barrier to creating *dependable* software?

The Understandability Gap: What happens when no one understands the code?

Bad code is not new

```
// Dear programmer:
// When I wrote this code, only god and
// I knew how it worked.
// Now, only god knows it!
//
// Therefore, if you are trying to optimize
// this routine and it fails (most surely),
// please increase this counter as a
// warning for the next person:
//
// total_hours_wasted_here = 254
//
```

Source: reposted on reddit, stack overflow, etc. for at least 15 years

But, arguably is getting worse…

- Systems become more complex over time (technical debt)
- Original architects retire
- Fault tolerance techniques chase performance, at the expense of simplicity

The Understandability Gap: What happens when no one understands the code?

Bad code is not new

```
// Dear programmer:
// When I wrote this code, only god and
// I knew how it worked.
// Now, only god knows it!
//
// Therefore, if you are trying to optimize
// this routine and it fails (most surely),
// please increase this counter as a
// warning for the next person:
//
// total_hours_wasted_here = 254
//
```

Claude

Source: reposted on reddit, stack overflow, etc. for at least 15 years

And generative AI changes the game…

- We can now have running code that **no** human has thought about the correctness of…
- Should we care?

Does understandability matter?

Illustration by Shoshana Gordon/ProPublica

- Federal Risk and Authorization Management Program (**FedRAMP**) authorization process of Microsoft's Government Community Cloud High (**GCC High**)
  - Claimed producing data flow diagrams showing how data is encrypted in transit for each service was too challenging
  - "But even **Microsoft's own engineers had struggled over the years to map the architecture of its products**, according to two people involved in building cloud services used by federal customers. At issue, according to people familiar with Microsoft's technology, was the **decades-old code of its legacy software**, which the company used in building its cloud services."

Thanks Roy, for the reference! https://www.propublica.org/article/microsoft-cloud-fedramp-cybersecurity-government

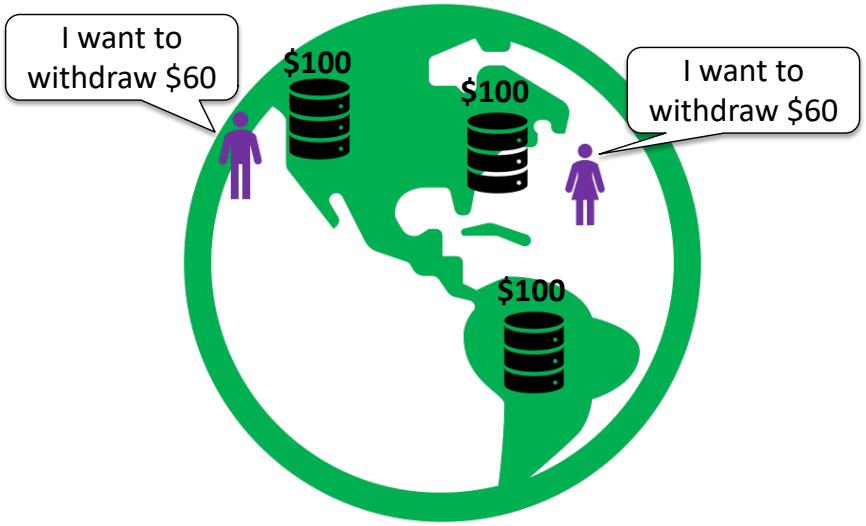
Some questions for the next 5 years…

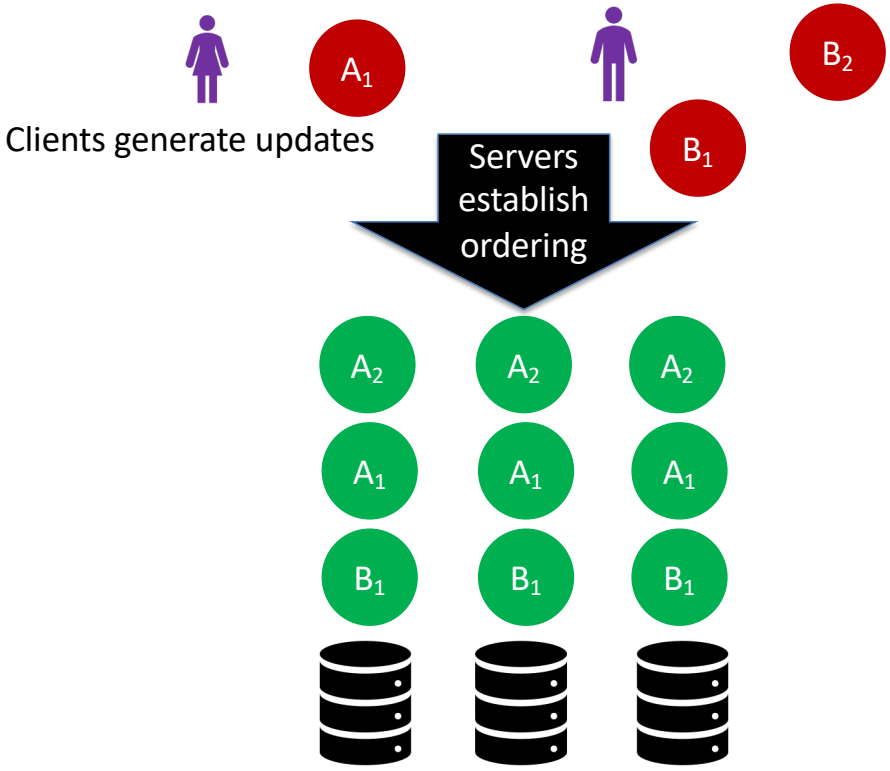
1. Does understandability matter?
   a. What is the relationship between *understandability* and *correctness*?

2. How do we make systems more understandable?
   a. At the protocol / architecture level?
   b. At the code level?

3. How should we measure understandability?

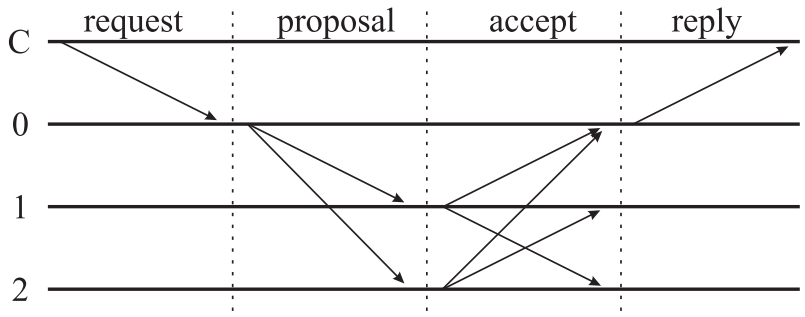
My starting point for understandability:
Fault-tolerant distributed protocols, e.g. State Machine Replication





Servers apply updates in the same order

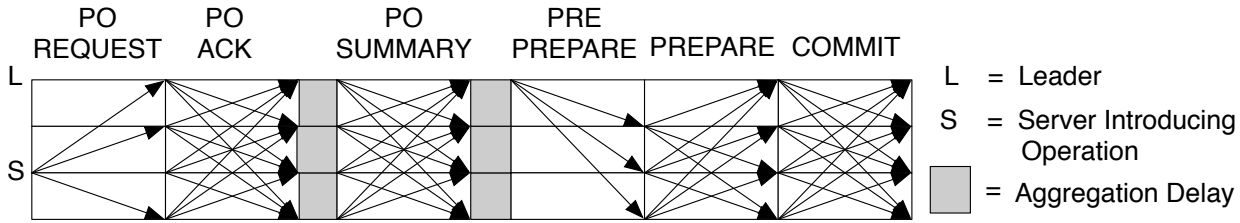
My starting point for understandability:
Fault-tolerant distributed protocols, e.g. State Machine Replication



Paxos for System Builders normal case execution

Normal case looks fairly simple, but leaves out a lot

- Leader election / view change, retransmission, flow control, garbage collection



Prime normal case execution

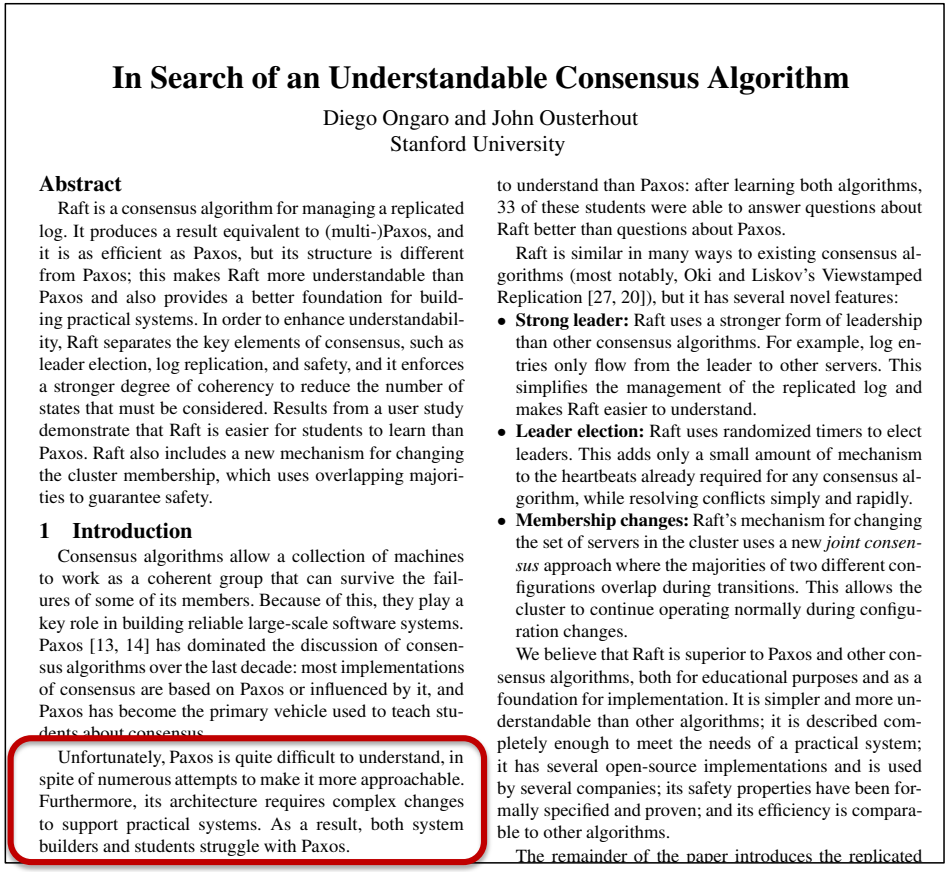
Stronger guarantees require more complexity

- Byzantine fault tolerance
- Performance guarantees under attack

Kirsch, Jonathan, and Yair Amir. "Paxos for system builders: An overview." In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pp. 1-6. 2008.

Amir, Yair, Brian Coan, Jonathan Kirsch, and John Lane. "Prime: Byzantine replication under attack." *IEEE transactions on dependable and secure computing* 8, no. 4 (2010): 564-577.

Prioritizing *Protocol* Understandability: The Raft Vision (2014)



In Search of an Understandable Consensus Algorithm

Diego Ongaro and John Ousterhout
Stanford University

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

1 Introduction

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems. Paxos [13, 14] has dominated the discussion of consensus algorithms over the last decade: most implementations of consensus are based on Paxos or influenced by it, and Paxos has become the primary vehicle used to teach students about consensus.

Unfortunately, Paxos is quite difficult to understand, in spite of numerous attempts to make it more approachable. Furthermore, its architecture requires complex changes to support practical systems. As a result, both system builders and students struggle with Paxos.

to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [27, 20]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.
- **Leader election:** Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly.
- **Membership changes:** Raft's mechanism for changing the set of servers in the cluster uses a new *joint consensus* approach where the majorities of two different configurations overlap during transitions. This allows the cluster to continue operating normally during configuration changes.

We believe that Raft is superior to Paxos and other consensus algorithms, both for educational purposes and as a foundation for implementation. It is simpler and more understandable than other algorithms; it is described completely enough to meet the needs of a practical system; it has several open-source implementations and is used by several companies; its safety properties have been formally specified and proven; and its efficiency is comparable to other algorithms.

The remainder of the paper introduces the replicated

- Systems-oriented pseudocode
- Design principles
  - Decomposition
    - Leader election, log replication, safety, membership change
  - State space reduction
    - "Strong" up-to-date leader, no log holes
- Evaluating understandability
  - Course-based user study
    - Lecture + quiz on basic operation and corner cases
    - Avg 25.7/60 for Raft, 20.8/60 for Paxos

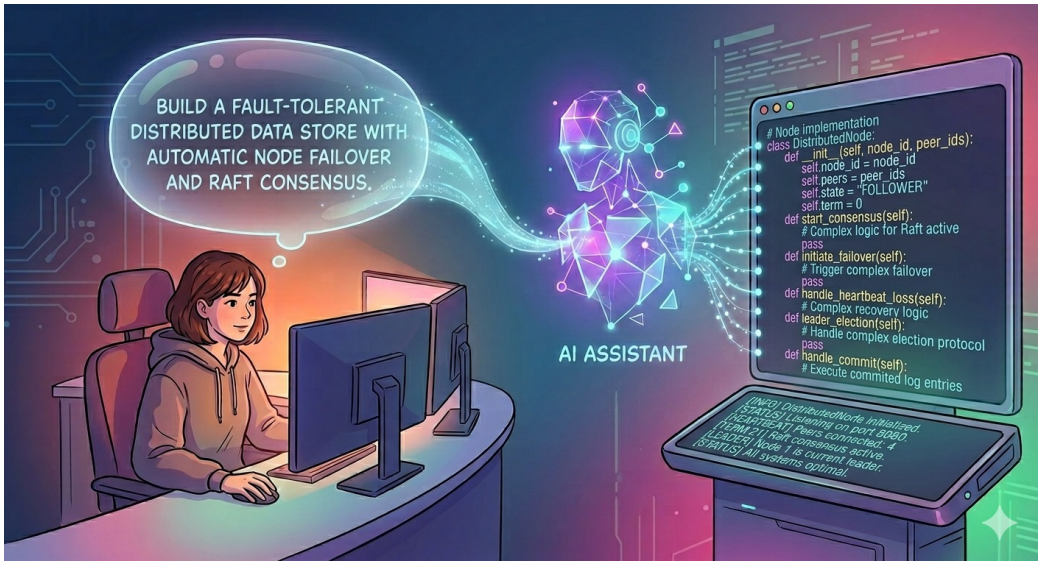
The Raft Reality (2026)

- What actually happened in the last 12 years?
  - **Community takeaway: Raft is a better replication protocol than Paxos**
  - Follow-up: lots of Raft implementations and evaluations; a few BFT Raft protocols that adapt the protocol itself; not much picking up the idea of explicitly designing for understandability

- A big gap – what about the code?
  - Original Raft implementation: ~2000 lines C++ (actual code only)
  - Original Raft TLA+ spec: 470 lines
    - Not executable, not model checked at practical size, used for manual proof
    - More work on this since the original paper
  - Understandable protocol != understandable implementation

The Vibe Coding Vision

- Specify *intent* for what I want the code to do in natural language, AI produces the code
  - No required mental model for *how* the code works
  - How to verify that the implementation matches my intent?
    - Code review
    - Test cases
    - Formal verification



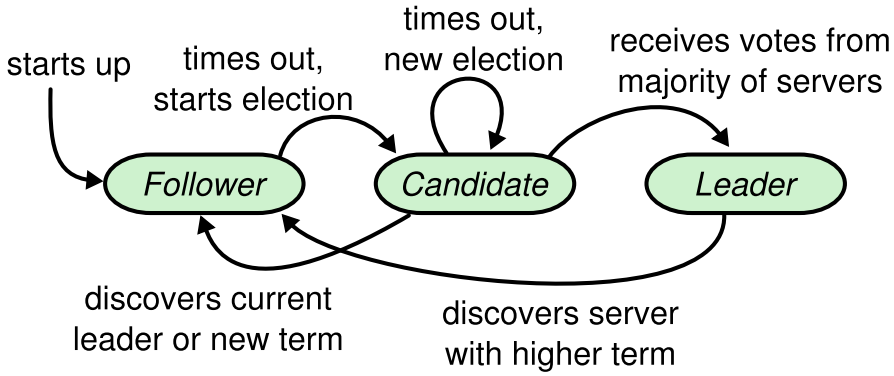
Source: Gemini

A different perspective: leveraging new language support

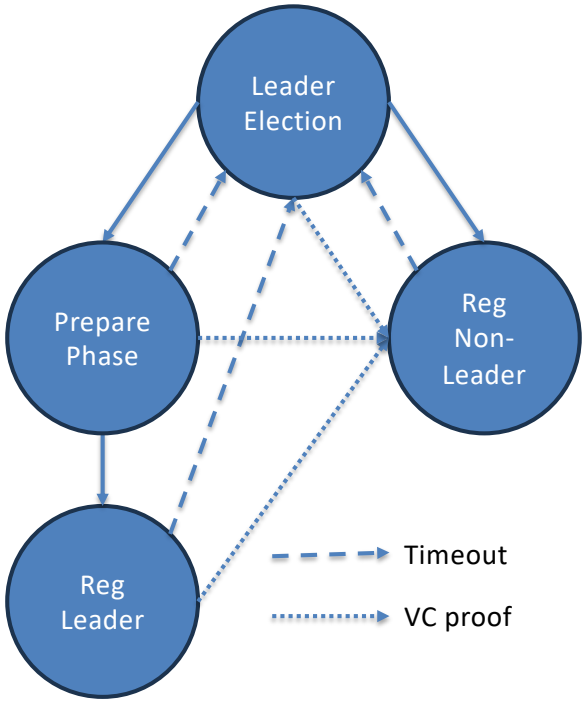
- **Specify the intent *precisely*** in a language designed for distributed systems
  - Deterministic result – intuitive specification **is the implementation**! No gap between intent and code
- **DistAlgo** supports this capability today!
  - By Annie Liu's group at Stony Brook – I'm not the creator, just a fan ☺
  - https://distalgo.cs.stonybrook.edu/
- But, still need principles for applying language capabilities to support understandability

Traditional Event-Driven State Machine Approach: Attempt Understandability via Decomposition

- Break the protocol into smaller states
- Detailed pseudocode specifies for each state:
  - What events (message arrivals or timeouts) can happen?
  - What action is taken in response to each event?



Raft state machine



Paxos for System Builders state machine

DistAlgo Capability 1: Run the (Event-Driven) Pseudcode

```
C1. Upon receiving Accept(server_id, view, seq):
C2.   Apply Accept to data structures
C3.   if Globally_Ordered_Ready(seq)
C4.     globally_ordered_update ← Construct_Globally_Ordered_Update(seq)
C5.     Apply globally_ordered_update to data structures
C6.     Advance_Aru()
```

```
def Receive_Accept(message):
  Update_Data_Structures(message)
  if Globally_Ordered_Ready(message.seq):
    prop = global_history[message.seq].proposal
    globally_ordered_update = Globally_Ordered_Update(
      prop.server_id, prop.seq, prop.update)
    Update_Data_Structures(globally_ordered_update)
    Advance_Aru()
```

* Server is a DistAlgo *process*; its *receive handler* abstracts out network and event handling details.

But, pseudocode isn't *quite* as simple as it looks

```
def receive(msg=message, from_= p):
  if Conflict(message): return
  ...
  elif isinstance(message, Accept):
    Receive_Accept(message)
```

```
def Receive_Accept(message):
  Update_Data_Structures(message)
  if Globally_Ordered_Ready(message.seq):
    prop = global_history[message.seq].proposal
    globally_ordered_update = Globally_Ordered_Update(
      prop.server_id, prop.seq, prop.update)
    Update_Data_Structures(globally_ordered_update)
    Advance_Aru()
```

Conflict check: should I handle this message?

Update data structures: store message (if needed)

Globally ordered ready?: Do I have a quorum of Proposal + Accepts?

Advance ARU: execute update, reply to client, advance ARU (executed up through this seq), restart view timeout

* Server is a DistAlgo *process*; its *receive handler* abstracts out network and event handling details.

```
class Global_History:
  def __init__(self, proposal, accepts, globally_ordered_update):
    self.proposal = proposal
    self.accepts = accepts
    self.globally_ordered_update = globally_ordered_update

class Server(process):
  def Conflict(message):
    if isinstance(message, Accept):
      if message.server_id == My_server_id:
        return True
      if message.view != Last_Installed:
        return True
      if (not message.seq in global_history or not global_history[message.seq].proposal
          or message.view != global_history[message.seq].proposal.view):
        return True
      return False

  def Update_Data_Structures(message):
    if isinstance(message, Accept):
      if message.seq in global_history and
         global_history[message.seq].globally_ordered_update != None:
        return
      if message.seq in global_history and
         len([a for a in global_history[message.seq].accepts if a != None]) >= int(N/2):
        return
      if message.seq in global_history and
         global_history[message.seq].accepts[message.server_id] != None:
        return
      if message.seq not in global_history:
        global_history[message.seq] = Global_History(None, [None for _ in servers],
                                                     None)
      global_history[message.seq].accepts[message.server_id] = message

    if isinstance(message, Globally_Ordered_Update):
      if message.seq in global_history and
         global_history[message.seq].globally_ordered_update == None:
         global_history[message.seq].globally_ordered_update = message
```

```
def Receive_Accept(message):
  Update_Data_Structures(message)
  if Globally_Ordered_Ready(message.seq):
    prop = global_history[message.seq].proposal
    globally_ordered_update = Globally_Ordered_Update(
      prop.server_id, prop.seq, prop.update)
    Update_Data_Structures(globally_ordered_update)
    Advance_Aru()

def Globally_Ordered_Ready(seq):
  if (seq in global_history and global_history[seq].proposal != None and
      len([a for a in global_history[seq].accepts if a != None]) >= int(N/2)):
    return True
  else:
    return False

def Advance_Aru():
  i = Local_Aru + 1
  while True:
    if i in global_history and global_history[i].globally_ordered_update != None:
      Local_Aru += 1
      Upon_Executing_Client_Update(global_history[Local_Aru].globally_ordered_update.update)
      i += 1
    else:
      return
def Upon_Executing_Client_Update(message):
  if message.server_id == My_server_id:
    Reply_to_client = 'Executed Update ' + str(message.timestamp)
    send(('Reply', Reply_to_client, message.timestamp), to= clients[message.client_id])
    if message.client_id in Pending_Updates:
      Update_Timer_[message.client_id].cancel()
      Pending_Updates.pop(message.client_id)
  Last_Executed[message.client_id] = message.timestamp
  if state != State.LEADER_ELECTION:
    Progress_Timer_.cancel()
    Progress_Timer = DEFAULT_TIMER
    Progress_Timer_ = Timer(Progress_Timer, Expiration_Progress_Timer)
    Progress_Timer_.start()
  if state == State.REG_LEADER:
    Send_Proposal()
```

Complete Accept handling implementation is longer, but still directly matches the pseudocode

Outcome of runnable event-driven pseudocode

- ***Action*** taken on each event is clear
- "Pseudocode" is real code – it runs and we can test it

- But: **What is the condition under which an update is executed?**
  - This is the critical property for proving safety

- Intuition: Server is in a **regular ordering state** (not leader election), and it has a **majority of matching Proposal+Accept messages** in its **current view** for the **next expected sequence number**

DistAlgo Capability 2: Express Protocol as Message History Queries

Write the condition directly!

```
elif ((is_elected_leader(view) or is_nonleader(view)) and
      some(received(('Proposal', _view, Aru + 1,
                     (client, server, timestamp, op))), has=
           len(setof(p, received(('Accept', _view, Aru + 1), from_= p)))
             + 1 > len(servers)/2)):
  Aru += 1
  ordered.add((client, server, timestamp, op))
  app_state, result = apply(op, app_state)
  if server == self:
    send(('Reply', timestamp, op, result), to= client)
  if attempted == view:
    if view_timer: view_timer.cancel()
    view_timer = Timer(View_TIMEOUT, send_View_timeout, args=[attempted + 1])
    view_timer.start()
```

DistAlgo Capability 2: Express Protocol as Message History Queries

Requires just few additional definitions (also written as message history queries):

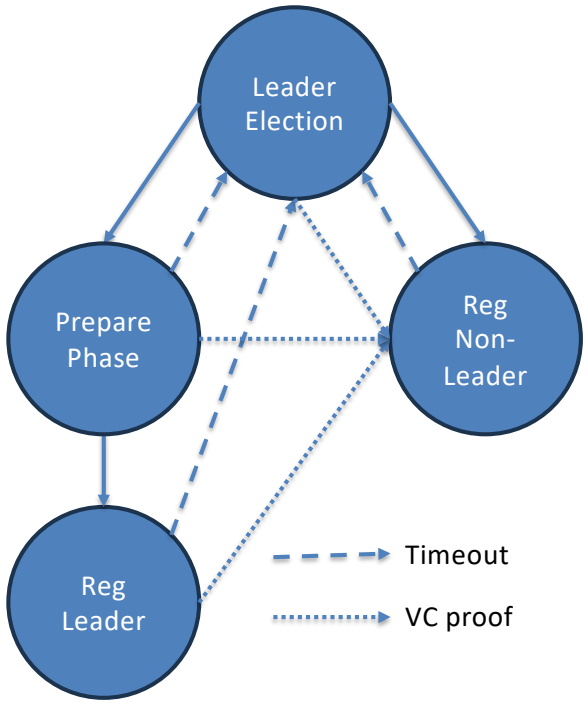
```
def is_elected_leader(view):
  return (leader(view) == self and
          len(setof(p, received(('Prepare_OK', _view, data_), from_= p)))
          > len(servers)/2 and
          not some(sent(('View_Change', view2)), has= view2 > view))

def is_nonleader(view):
  return (some(sent(('Prepare_OK', _view, data_list_)))
          and not some(sent(('View_Change', view2)), has= view2 > view)
          and not leader(view) == self)

def leader(view):  return servers[((view-1) % len(servers))]
```

Outcome of message-history-driven implementation

- Core protocol logic captured in **7 message history conditions**
  - normal-case leader election (2), prepare phase (2), global ordering (3)
- A few auxiliary conditions to support:
  - client handling (1), partitions/merges/recoveries (3), retransmissions (3)
- **Less than 250 total lines** (including comments, blank lines, etc)
- Closely **matches design *intent***
  - Most of the work involved re-creating what the pseudocode was supposed to be doing. English text descriptions were a better guide.

Message-history-driven implementation enables new design insights

- Concise implementation **expands the scope of what the designer/developer can reason about**
- Traditionally, actions are strictly limited based on state (decomposition)
  - NOT necessarily required for correctness



Paxos for System Builders state machine

Protocol improvement based on message-history-driven implementation

These "state" restrictions are NOT needed (and make the protocol slightly less live)

```
elif ((is_elected_leader(view) or is_nonleader(view)) and
      some(received(('Proposal', _view, Aru + 1,
                     (client, server, timestamp, op))), has=
           len(setof(p, received(('Accept', _view, Aru + 1), from_= p)))
             + 1 > len(servers)/2)):
  [Execute update]
```

is_nonleader check was actually not needed at all

```
def is_nonleader(view):
  return (some(sent(('Prepare_OK', _view, data_list_)))
          and not some(sent(('View_Change', view2)), has= view2 > view)
          and not leader(view) == self)
```

Declarative, Message-History-Based Implementation as a Path to Understandability

- Offers **concise specification** that makes protocol **logic clearer** than event-driven pseudocode, and is real **running code**
- Clearer logic supports new **design insights**
- BUT, message history implementation is not *directly* practical today
  - Queries become extremely expensive as history grows
  - All messages are stored indefinitely
  - **Incrementalization** can address this
    - Mechanical process of translating queries to data structures with incremental updates; in theory can be automated

Liu, Yanhong A., Scott D. Stoller, and Bo Lin. "From clarity to efficiency for distributed algorithms." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39, no. 3 (2017): 1-41.

Some questions for the next 5 years…and initial thoughts

1. Does understandability matter?
   - Yes, if we want to be able to improve (and invent new) protocols/systems

   a. What is the relationship between *understandability* and *correctness*?
      - Understandable -> Correct in terms of matching design *intent*
      - Message-history-based implementation is more amenable to proofs than traditional event-driven code by making conditions for each action clear

2. How do we make systems more understandable?
   a. At the protocol / architecture level?
   b. At the code level?

   } DistAlgo approach can unify these

3. How should we measure understandability?
   - So far, informal in our work. But, the fact that it led us to new protocol insights gives some evidence
   - How to make it more rigorous? User studies, panel of experts; how well do existing code complexity metrics map to understanding?