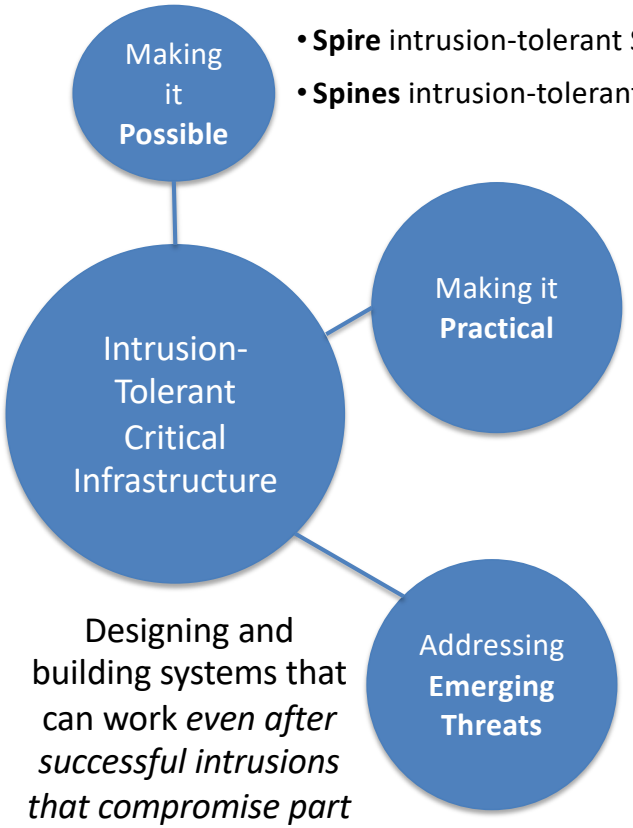
Toward Understandable Distributed Systems

Amy Babay

School of Computing and Information
University of Pittsburgh

University of Pittsburgh

Toward Deployable, Manageable, Intrusion-Tolerant Critical Infrastructure



Designing and building systems that can work *even after successful intrusions that compromise part of the system*

- **Spire** intrusion-tolerant SCADA for the power grid — www.spire-sys.org DSN 2018, 2019
- **Spines** intrusion-tolerant network — www.spines-org.github.io ICDCS 2016

- **Cloud-based** intrusion-tolerant SCADA systems can reduce the cost of resilience, without exposing sensitive data to cloud providers — DSN 2021 (Best paper runner-up), SRDS 2023, Maher Khan PhD thesis (2024)
- **Digital twins** can provide a transition path to intrusion-tolerant architectures — DOE Cyber Energy Center project, DT4DRS (DSN-W) 2025
- **Policy support** is needed for adoption — Pitt Momentum Funds project
- **Understandable replication protocol** implementations can improve long-term maintainability

- **Compound threats** involving cyberattacks coinciding with natural hazards require reconfigurable architectures — DoD/DOE SERDP project, SRDS 2024 (Best paper award)
- **Benign failures** reveal differences between architectures that "on paper" tolerate the same threat models — SRDS 2024, TNSM 2025

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

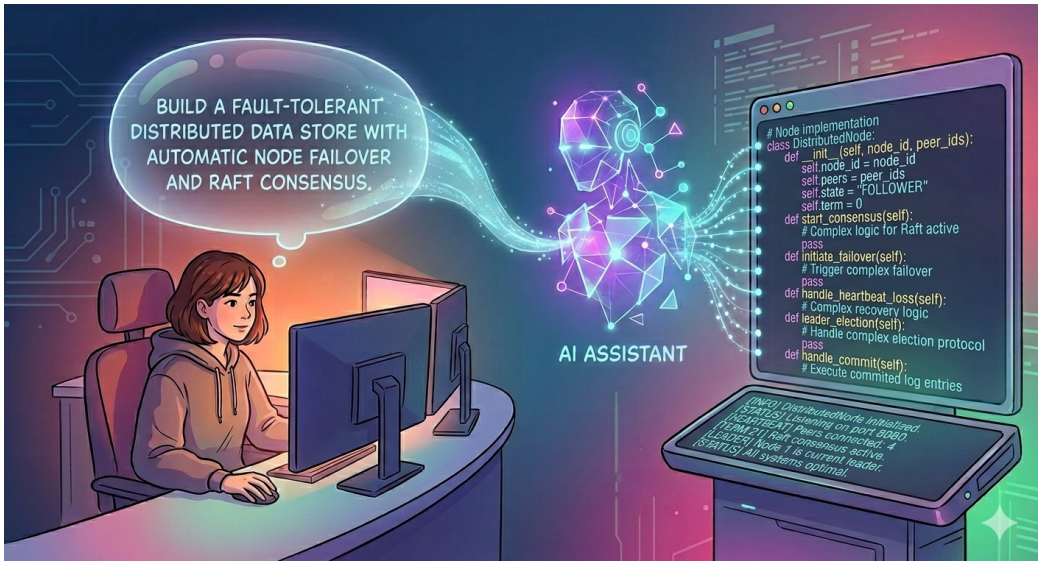
Source: reposted on reddit, stack overflow, etc. for at least 15 years

And generative AI changes the game…

- We can now have running code that **no** human has thought about the correctness of

The Vibe Coding Vision

- Specify *intent* for what I want the code to do in natural language, AI produces the code
  - How to verify that the implementation matches my intent?



Source: Gemini

Leveraging new language support to create understandable implementations

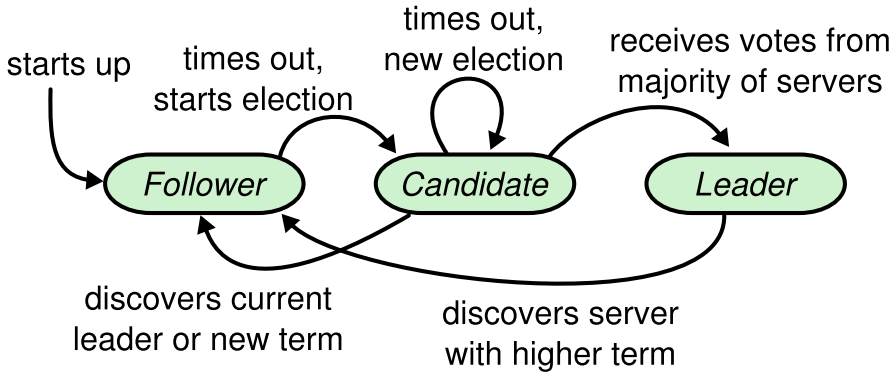
- **Specify the intent *precisely*** in a language designed for distributed systems
  - Deterministic result – intuitive specification **is the implementation**! No gap between intent and code

- **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/

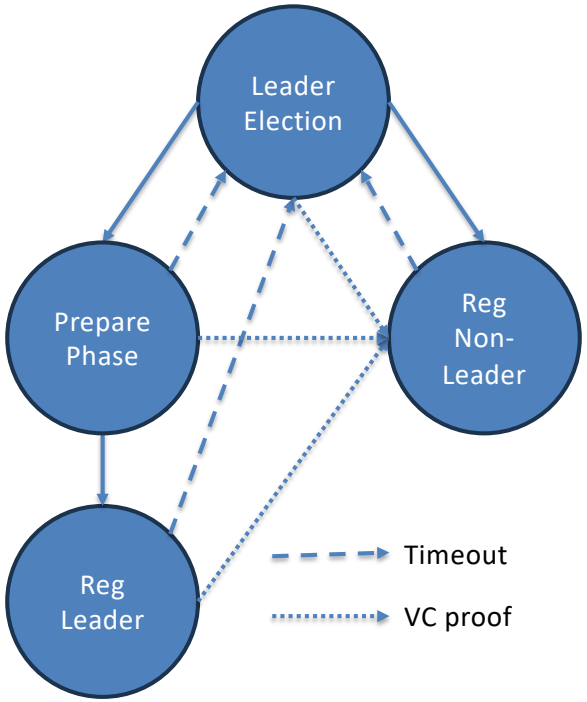
- But, still need principles for applying language capabilities to support understandability

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

```
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()
```

Event-driven Accept handling implementation in DistAlgo directly matches the original Paxos-SB pseudocode

Declarative Approach with DistAlgo: Make the logic clear and compact

How? Express as queries over message history

```
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()
```

Declarative Approach with DistAlgo: Make the logic clear and compact

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.

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**

Some questions:

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

2. How should we measure understandability?