Protocol Conformance with Choreographic PlusCal

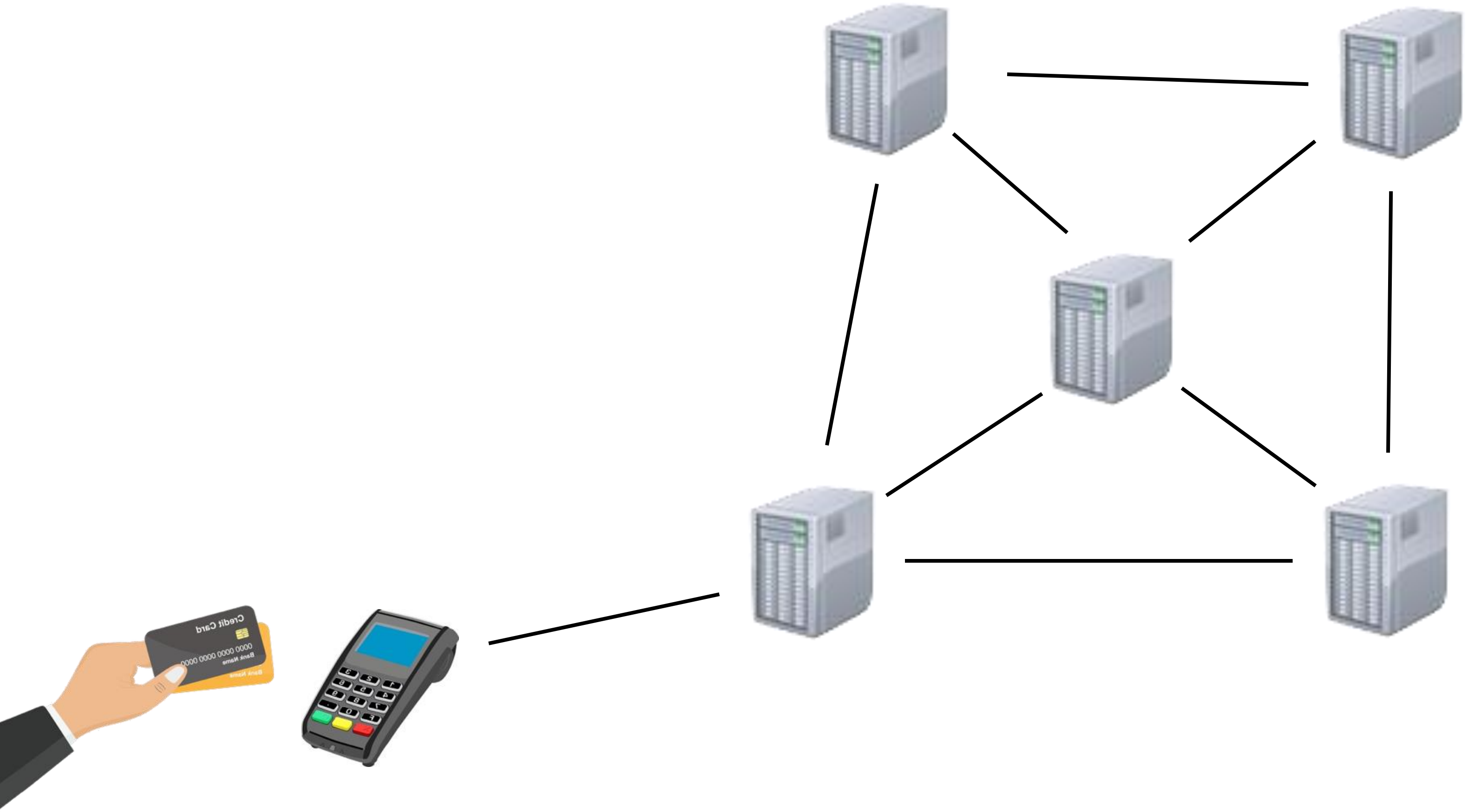
Darius Foo, Andreea Costea, and Wei-Ngan Chin

National University of Singapore

17th International Symposium on Theoretical Aspects of Software Engineering

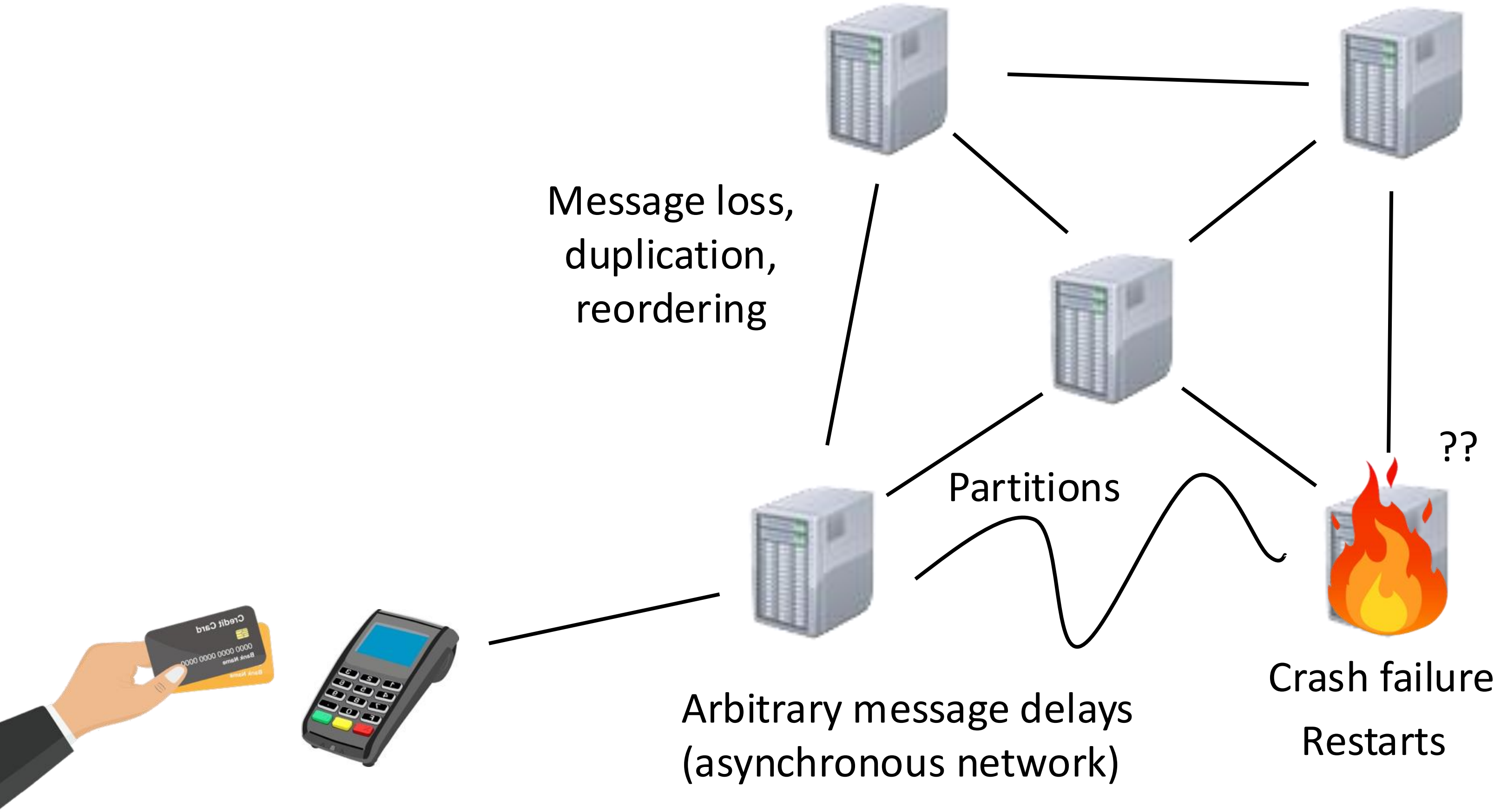
6 July 2023

Distributed Systems



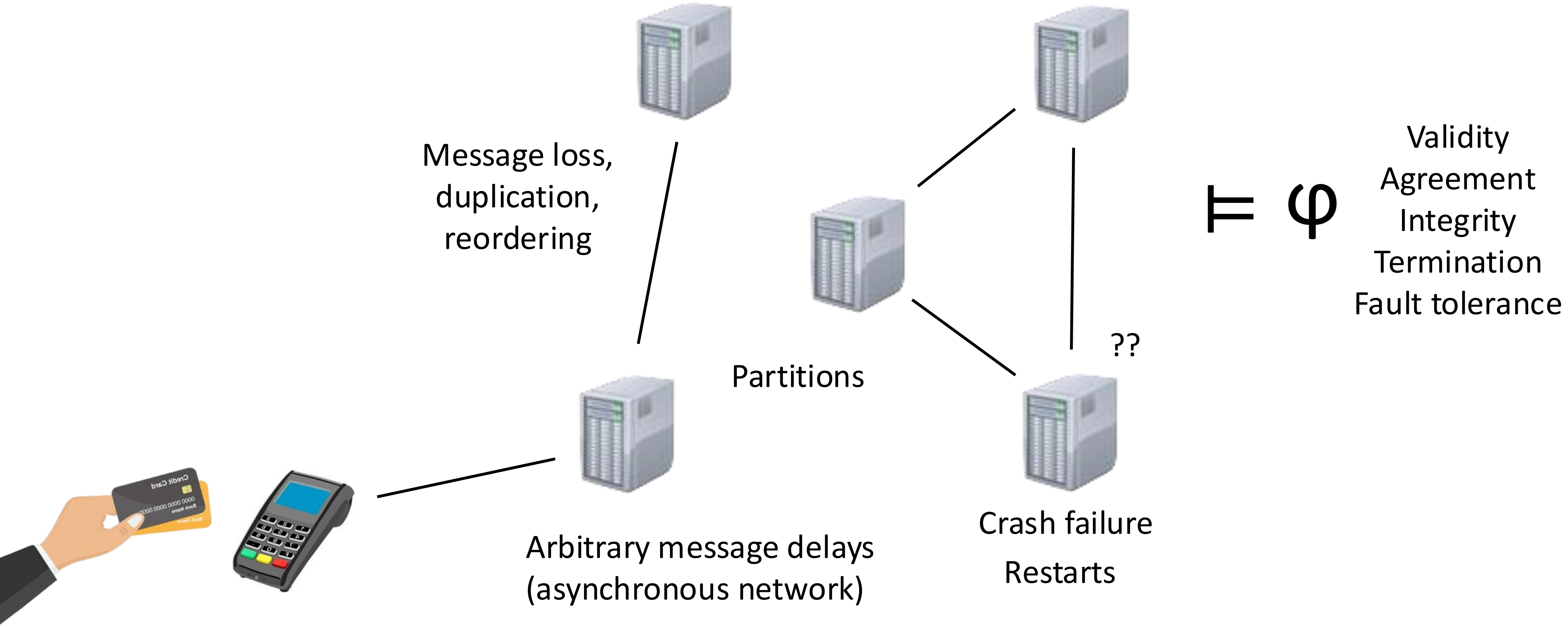
“Transactions shouldn’t be forgotten”
“System should remain available”

Distributed Systems

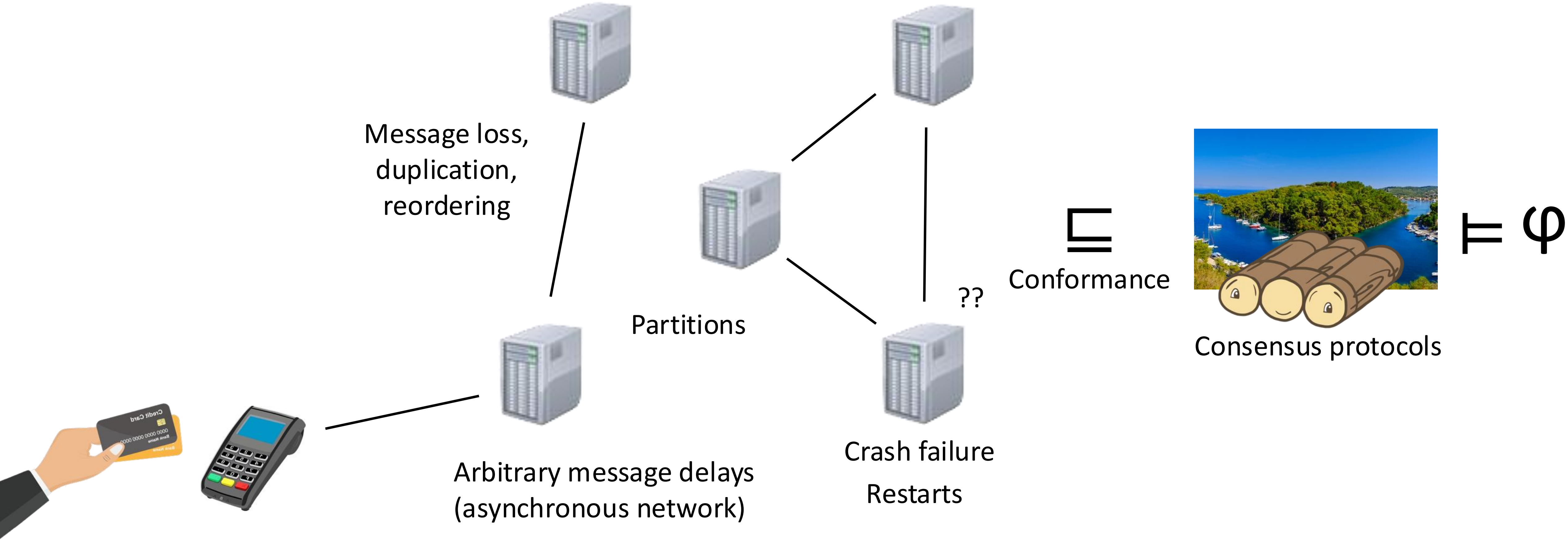


"Transactions shouldn't be forgotten"
"System should remain available"

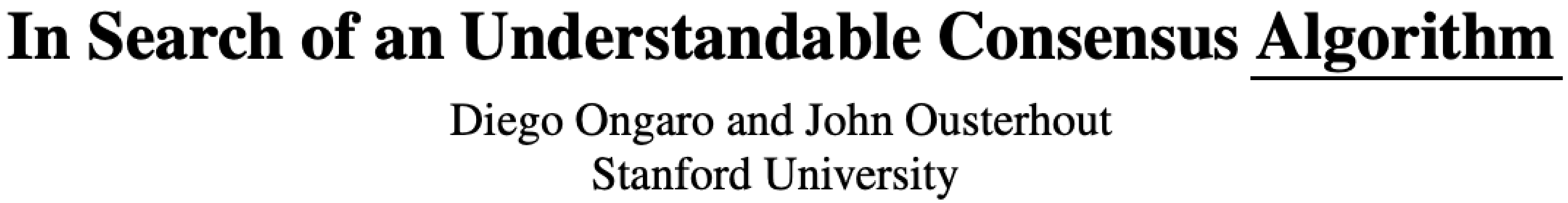
Distributed Systems



Distributed Systems



Consensus protocols in practice


In Search of an Understandable Consensus Algorithm

Diego Ongaro and John Ousterhout
Stanford University

(2014)

Raft is a fully-featured consensus protocol that operates by quorum, performs leader election, maintains logs for durability, handles reconfiguration…

Consensus protocols in practice

Where can I get Raft?

There are many implementations of Raft available in various stages of development. This table lists the implementations we know about with source code available. The most popular and/or recently updated implementations are towards the top. This information will inevitably get out of date; please submit a pull request or an issue to update it.

Stars	Name	Primary Authors	Language	License	Leader Election + Log Replication?	Persistence?	Membership Changes?	Log Compaction?
13,312★	TiKV	Jay, ngaut, siddontang, tiancaiamao	Rust	Apache-2.0	Yes	Yes	Yes	Yes
9,211★	nebula-graph-storage	Sherman Ye, Doodle Wang	C++	Apache-2.0	Yes	Yes	Yes	Yes
26,166★	RethinkDB		C++	Apache-2.0	Yes	Yes	Yes	Yes
10,501★	Seastar Raft	Gleb Natapov, Konstantin Osipov, Pavel Solodovnikov, Alejo Sanchez, Kamil Braun, Tomash Grabiec	C++20	AGPL	Yes	Yes	Yes	Yes
5,439★	hazelcast-raft	Mehmet Dogan, Ensar Basri Kahveci	Java	Apache-2.0	Yes	Yes	Yes	Yes
7,220★	hashicorp/raft	Armon Dadgar	Go	MPL-2.0	Yes	Yes	Yes	Yes
3,560★	braft	Zhangyi Chen, Yao Wang	C++	Apache-2.0	Yes	Yes	Yes	Yes

… 137 more implementations

How hard is it to implement a protocol correctly?

Minimizing Faulty Executions of Distributed Systems (2015)

Colin Scott⋆ Aurojit Panda⋆ Vjekoslav Brajkovic⋄ George Necula⋆
Arvind Krishnamurthy† Scott Shenker⋆⋄
⋆*UC Berkeley* ⋄*ICSI* †*University of Washington*

8 bugs

Abstract

When troubleshooting buggy executions of distributed ...s, developers typically start by manually separat- ...ents that are responsible for triggering the ... from those that are extraneous (noise). We ...Mi, a tool for automatically performing this ...n. We apply DEMi to buggy executions of two ... distributed systems, Raft and Spark, and ...t produces minimized executions that are be- ...1X and 4.6X the size of optimal executions.

much more costly than machine time, automated minimization tools for *sequential* test cases [24, 86, 94] have already proven themselves valuable, and are routinely applied to bug reports for software projects such as Firefox [1], LLVM [7], and GCC [6].

In this paper we address the problem of automatically minimizing executions of distributed systems. We focus on executions generated by fuzz testing, but we also illustrate how one might minimize production traces.

Distributed executions have two distinguishing fea-

How hard is it to implement a protocol correctly?

Fuzz testing distributed systems with QuickCheck

(2016)

14 bugs

https://pusher.com/blog/fuzz-testing-distributed-systems-with-quickcheck/

How hard is it to implement a protocol correctly?

Distributed System Fuzzing (2023)

Ruijie Meng*†
National University of Singapore
Singapore
ruijie_meng@u.nus.edu

George Pîrlea*
National University of Singapore
Singapore
gpirlea@comp.nus.edu.sg

Abhik Roychoudhury‡
National University of Singapore
Singapore
abhik@comp.nus.edu.sg

Ilya Sergey
National University of Singapore
Singapore
ilya@nus.edu.sg

22 bugs
10 CVEs

...the lightweight approach of choice for finding ...rograms. It provides a balance between effi- ...s by conducting a biased random search over ... inputs using a feedback function from ob- ... For distributed system testing, however, the ...resented today by only black-box tools that ...ner and exploit any knowledge of the system's ...uide the search for bugs.

are generated in a purely random fashion, or it can be guided by knowledge of the program's internal structure (white-box). The most popular fuzzers are *grey-box*, where the search is guided by run-time observations of program behaviour, collected, as tests execute, for artefacts instrumented at compile time. Thanks to the ease of its deployment and use, grey-box fuzzing is the state-of-the-practice for automatically discovering bugs in sequential programs.

A common approach to finding bugs in distributed systems in practice is *stress-testing*, in which the system is subjected to ...

How hard is it to implement a protocol correctly?

ModelFuzz: Model guided fuzzing of distributed systems. (2025)

Srinidhi Nagendra*

Max Planck Institute for Software Systems

February 12, 2025

13 bugs

Large-scale distributed systems form the core infrastructure for many software applications. It is well-known that designing such systems is difficult due to interactions between concurrency and faults, and subtle bugs often show up in production. Thus, designing testing techniques that cover diverse and interesting program behaviors to find subtle bugs has been an important research challenge.

Coverage-guided fuzzing, which guides test generation toward more coverage, has been effective in exploring diverse executions, mainly in the sequential setting, using structural coverage criteria as a feedback mechanism [1, 2]. However, adopting coverage-guided fuzzing for testing distributed system implementations is nontrivial since there is no common notion of *coverage* for distributed system executions. Unfortunately, structural code coverage criteria such as line coverage can ignore the orderings of message interactions in a system, thus missing interesting schedules. On the other hand, more detailed criteria, such

Why is conformance hard?

> Distributed systems are notoriously hard to get right. Protocol designers struggle to reason about concurrent execution on multiple machines, which leads to subtle errors. Engineers implementing such protocols face the same subtleties and, worse, must improvise to fill in gaps between abstract protocol descriptions and practical constraints, e.g., that real logs cannot grow without bound. Thorough testing is considered best practice, but its efficacy is limited by distributed systems' combinatorially large state spaces.

IronFleet: Proving Practical Distributed Systems Correct

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch,
Bryan Parno, Michael L. Roberts, Srinath Setty, Brian Zill

Microsoft Research

Why is conformance hard?

- Underspecification
  - Fully-fledged protocols are large and complex
    - Basic Raft: 485 LoC
    - TLC-optimized Raft: 653 LoC
    - Raft with reconfiguration: 1083 LoC
  - Large state machines (TLA$^+$) are hard to extend; conventional wisdom is to keep them abstract
  - PlusCal allows specifying implementation concerns, but is not used much in practice (25% of protocols in tlaplus/Examples)

```
\* Defines how the variables may transition.
Next == /\ \/ \E i \in Server : Restart(i)
           \/ \E i \in Server : Timeout(i)
           \/ \E i,j \in Server : RequestVote(i, j)
           \/ \E i \in Server : BecomeLeader(i)
           \/ \E i \in Server, v \in Value : ClientRequest(i, v)
           \/ \E i \in Server : AdvanceCommitIndex(i)
           \/ \E i,j \in Server : AppendEntries(i, j)
           \/ \E m \in DOMAIN messages : Receive(m)
           \/ \E m \in DOMAIN messages : DuplicateMessage(m)
           \/ \E m \in DOMAIN messages : DropMessage(m)
           \* History variable that tracks every log ever:
        /\ allLogs' = allLogs \cup {log[i] : i \in Server}
```

Why is conformance hard?

- Implementations are large and complex
  - Real-world Raft: etcd, 20k LoC, with concurrency, I/O, etc.
  - Implementation bugs can compromise protocol guarantees
  - Lack of lightweight tools for justifying parts of the implementation and supporting automated checks

Challenges

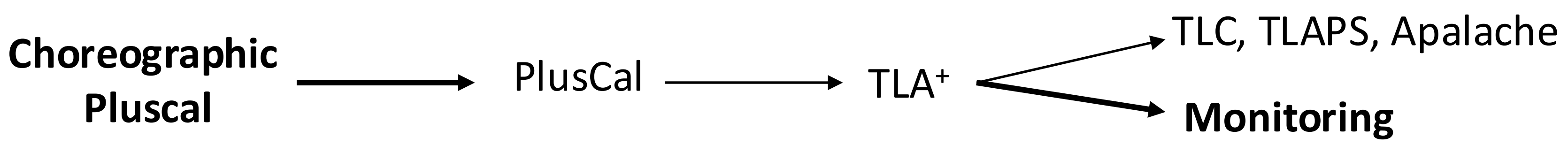
1. Underspecification due to inadequate specification medium

2. Conformance of real-world consensus implementations

Contributions

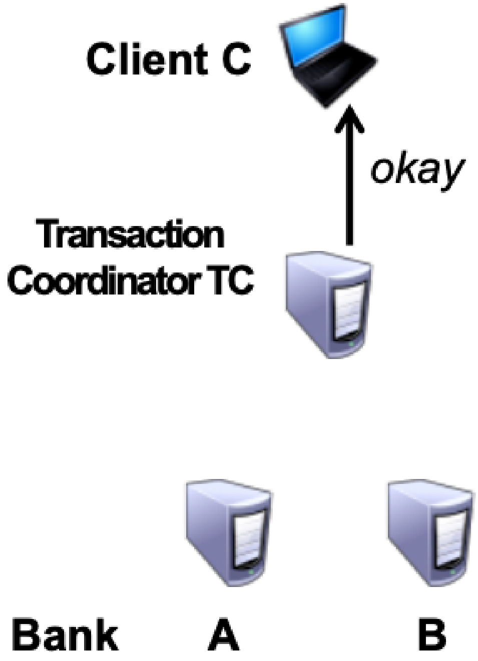
1. Choreographic PlusCal

2. Practical monitoring using *existing* TLA+ specifications



Two-phase commit

A *correct* atomic commit protocol



1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → TC:** vote *"yes"* or *"no"*

(after acquiring all resources, e.g. locks, they will need)

4. **TC → A, B:** *"commit!"* or *"abort!"*
   – **TC** sends ***commit*** if **both** say *yes*
   – **TC** sends ***abort*** if **either** say *no*
5. **TC → C:** *"okay"* or *"failed"*

- **A, B** commit on receipt of commit message

https://web.kaust.edu.sa/Faculty/MarcoCanini/classes/CS240/F19/docs/L10-2pc.pdf

Two-phase commit in PlusCal

```
process (C \in coordinators)
  variables temp = participants,
            aborted = FALSE; {
    while (temp /= {}) {
      with (r \in temp) {
        Send(self, r, "prepare");
        temp := temp \ {r};
      } };
    temp := participants;
    while (temp /= {} \/ aborted) {
      with (r \in temp) {
        either {
        Receive(r, self, "prepared");
        } or {
        Receive(r, self, "abort");
        aborted := TRUE;
        };
        temp := temp \ {r};
      } };
    if (aborted) {
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Send(coord, r, "abort");
        temp := temp \ {r};
      } };
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Receive(r, coord, "aborted");
        temp := temp \ {r};
      } }
    } else {
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Send(coord, r, "commit");
        temp := temp \ {r};
      } }
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Receive(r, coord, "committed");
        temp := temp \ {r};
      } } } }
```

```
process (P \in participants) {
  Receive(coord, self, "prepare");
  either {
  psend:
    Send(self, coord, "prepared");
  } or {
    Send(self, coord, "abort");
  };
  either {
    Receive(coord, self, "commit");
    Send(self, coord, "committed");
  } or {
    Receive(coord, self, "abort");
    Send(self, coord, "aborted");
  } }
```

Two-phase commit in PlusCal

```
process (C \in coordinators)
  variables temp = participants,
            aborted = FALSE; {
    while (temp /= {}) {
      with (r \in temp) {
        Send(self, r, "prepare");
        temp := temp \ {r};
      } };
    temp := participants;
    while (temp /= {} \/ aborted) {
      with (r \in temp) {
        either {
        Receive(r, self, "prepared");
        } or {
        Receive(r, self, "abort");
        aborted := TRUE;
        };
        temp := temp \ {r};
      } };
    if (aborted) {
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Send(coord, r, "abort");
        temp := temp \ {r};
      } };
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Receive(r, coord, "aborted");
        temp := temp \ {r};
      } }
    } else {
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Send(coord, r, "commit");
        temp := temp \ {r};
      } }
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Receive(r, coord, "committed");
        temp := temp \ {r};
      } } } }
```

```
process (P \in participants) {
  Receive(coord, self, "prepare");
  either {
  psend:
    Send(self, coord, "prepared");
  } or {
    Send(self, coord, "abort");
  };
  either {
    Receive(coord, self, "commit");
    Send(self, coord, "committed");
  } or {
    Receive(coord, self, "abort");
    Send(self, coord, "aborted");
  } }
```

Two-phase commit in PlusCal

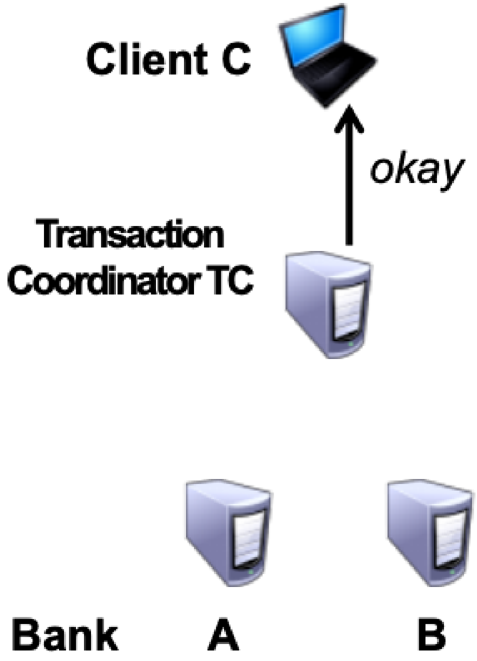
```
process (C \in coordinators)
  variables temp = participants,
            aborted = FALSE; {
    while (temp /= {}) {
      with (r \in temp) {
        Send(self, r, "prepare");
        temp := temp \ {r};
      } };
    temp := participants;
    while (temp /= {} \/ aborted) {
      with (r \in temp) {
        either {
        Receive(r, self, "prepared");
        } or {
        Receive(r, self, "abort");
        aborted := TRUE;
        };
        temp := temp \ {r};
      } };
    if (aborted) {
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Send(coord, r, "abort");
        temp := temp \ {r};
```

```
process (P \in participants) {
  Receive(coord, self, "prepare");
  either {
  psend:
    Send(self, coord, "prepared");
  } or {
    Send(self, coord, "abort");
  };
  either {
    Receive(coord, self, "commit");
    Send(self, coord, "committed");
  } or {
    Receive(coord, self, "abort");
    Send(self, coord, "aborted");
  } }
```

…

Two-phase commit

A *correct* atomic commit protocol



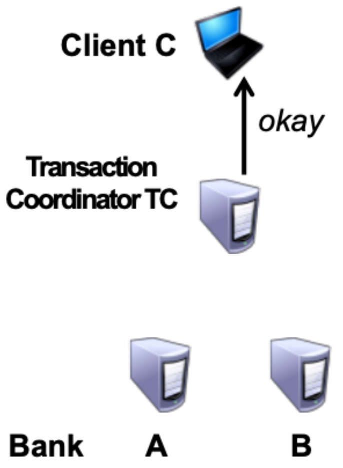
1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → TC:** vote *"yes"* or *"no"*
4. **TC → A, B:** *"commit!"* or *"abort!"*
   - **TC** sends ***commit*** if **both** say *yes*
   - **TC** sends ***abort*** if **either** say *no*
5. **TC → C:** *"okay" or "failed"*

- **A, B** commit on receipt of commit message

https://web.kaust.edu.sa/Faculty/MarcoCanini/classes/CS240/F19/docs/L10-2pc.pdf

Choreographic PlusCal: **choreography**

A *correct* atomic commit protocol



1. **C → TC:** *“go!”*
2. **TC → A, B:** *“prepare!”*
3. **A, B → TC:** vote *“yes”* or *“no”*
4. **TC → A, B:** *“commit!”* or *“abort!”*
   - **TC** sends ***commit*** if **both** say *yes*
   - **TC** sends ***abort*** if **either** say *no*
5. **TC → C:** *“okay”* or *“failed”*

- **A, B** commit on receipt of commit message

```
choreography
  (P \in participants),
  (C \in coordinators) {

    all (p \in participants) {
      Transmit(coord, p, "prepare");
      either {
        Transmit(p, coord, "prepared");
      } or {
        Transmit(p, coord, "aborted");

      } } };
  if (aborted) {
    all (p \in participants) {
      Transmit(coord, p, "abort");
      Transmit(p, coord, "aborted");
    }
  } else {
    all (p \in participants) {
      Transmit(coord, p, "commit");
      Transmit(p, coord, "committed");
    } } }
```

Choreographic PlusCal: **all**

```
process (C \in coordinators)
  variables temp = participants,
            aborted = FALSE; {
    while (temp /= {}) {
      with (r \in temp) {
        Send(self, r, "prepare");
        temp := temp \ {r};
      } };
    temp := participants;
    while (temp /= {} \/ aborted) {
      with (r \in temp) {
        either {
        Receive(r, self, "prepared");
        } or {
        Receive(r, self, "abort");
        aborted := TRUE;
        };
        temp := temp \ {r};
      } };
    if (aborted) {
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
```

Encoding a multicast

```
all (p \in participants) {
  Transmit(coord, p, "prepare");
  either {
    Transmit(p, coord, "prepared");
  } or {
    Transmit(p, coord, "aborted");
    cancel "phase1";
  } }
```

Choreographic PlusCal: **task** and **cancel**

```
process (C \in coordinators)
  variables temp = participants,
            aborted = FALSE; {
    while (temp /= {}) {
      with (r \in temp) {
        Send(self, r, "prepare");
        temp := temp \ {r};
      } };
    temp := participants;
    while (temp /= {} \/ aborted) {
      with (r \in temp) {
        either {
        Receive(r, self, "prepared");
        } or {
        Receive(r, self, "abort");
        aborted := TRUE;
        };
        temp := temp \ {r};
      } };
    if (aborted) {
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
```

```
task coordinators "phase1" {
  all (p \in participants) {
    Transmit(coord, p, "prepare");
    either {
      Transmit(p, coord, "prepared");
    } or {
      Transmit(p, coord, "aborted");
      cancel "phase1";
    } } };
```

Stop if participant aborts

Choreographic PlusCal

```
process (C \in coordinators)
  variables temp = participants,
            aborted = FALSE; {
    while (temp /= {}) {
      with (r \in temp) {
        Send(self, r, "prepare");
        temp := temp \ {r};
      } };
    temp := participants;
    while (temp /= {} \/ aborted) {
      with (r \in temp) {
        either {
        Receive(r, self, "prepared");
        } or {
        Receive(r, self, "abort");
        aborted := TRUE;
        };
        temp := temp \ {r};
      } };
    if (aborted) {
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Send(coord, r, "abort");
        temp := temp \ {r};
      } };
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Receive(r, coord, "aborted");
        temp := temp \ {r};
      } }
    } else {
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Send(coord, r, "commit");
        temp := temp \ {r};
      } }
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Receive(r, coord, "committed");
        temp := temp \ {r};
      } } } }
```

```
process (P \in participants) {
  Receive(coord, self, "prepare");
  either {
  psend:
    Send(self, coord, "prepared");
  } or {
    Send(self, coord, "abort");
  };
  either {
    Receive(coord, self, "commit");
    Send(self, coord, "committed");
  } or {
    Receive(coord, self, "abort");
    Send(self, coord, "aborted");
  } }
```

```
choreography
  (P \in participants),
  (C \in coordinators) {
  task coordinators "phase1" {
    all (p \in participants) {
      Transmit(coord, p, "prepare");
      either {
        Transmit(p, coord, "prepared");
      } or {
        Transmit(p, coord, "aborted");
        cancel "phase1";
      } } };
  if (aborted) {
    all (p \in participants) {
      Transmit(coord, p, "abort");
      Transmit(p, coord, "aborted");
    }
  } else {
    all (p \in participants) {
      Transmit(coord, p, "commit");
      Transmit(p, coord, "committed");
    } } }
```

Choreographic PlusCal

Protocol	Ch. PlusCal	TLA^+
Two-phase commit [23]	23	66
Non-blocking atomic commit [35]	36	96
Raft leader election [32]	46	186

Table 1: Relative specification sizes (LoC)

```
choreography
  (P \in participants),
  (C \in coordinators) {
  task coordinators "phase1" {
    all (p \in participants) {
      Transmit(coord, p, "prepare");
      either {
        Transmit(p, coord, "prepared");
      } or {
        Transmit(p, coord, "aborted");
        cancel "phase1";
      } } };
  if (aborted) {
    all (p \in participants) {
      Transmit(coord, p, "abort");
      Transmit(p, coord, "aborted");
    }
  } else {
    all (p \in participants) {
      Transmit(coord, p, "commit");
      Transmit(p, coord, "committed");
    } } }
```

Projection & monitoring

```
choreography
  (P \in participants),
  (C \in coordinators) {
  task coordinators "phase1" {
    all (p \in participants) {
      Transmit(coord, p, "prepare");
      either {
        Transmit(p, coord, "prepared");
      } or {
        Transmit(p, coord, "aborted");
        cancel "phase1";
      } } };
  if (aborted) {
    all (p \in participants) {
      Transmit(coord, p, "abort");
      Transmit(p, coord, "aborted");
    }
  } else {
    all (p \in participants) {
      Transmit(coord, p, "commit");
      Transmit(p, coord, "committed");
    } } }
```

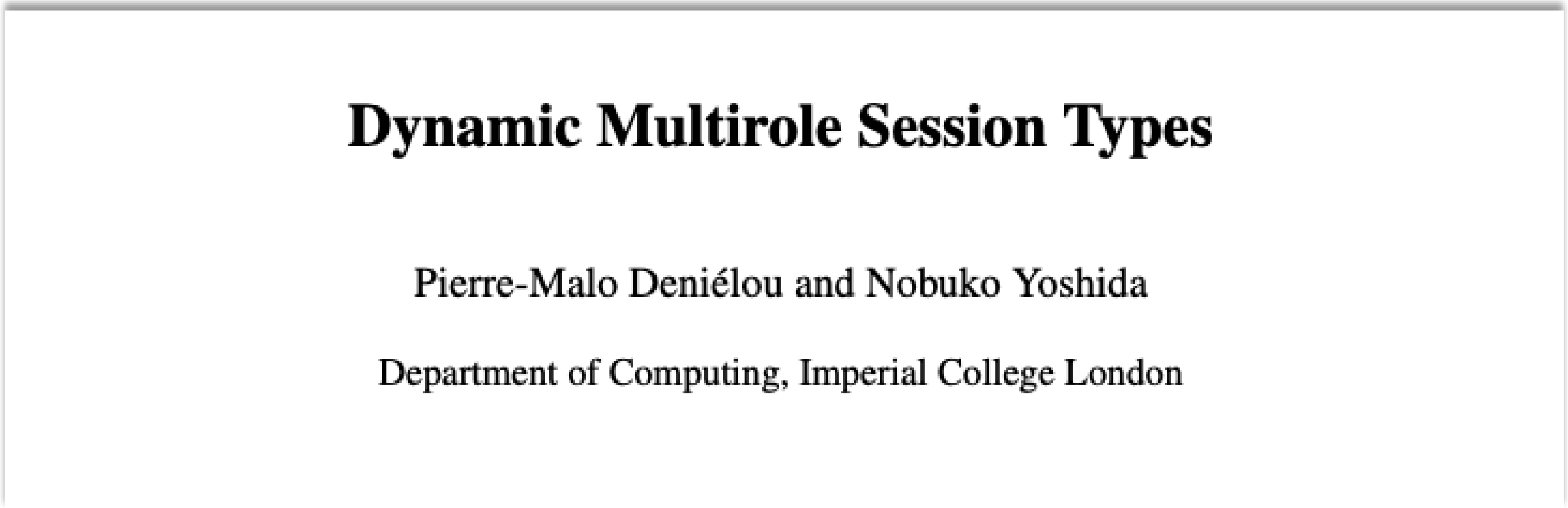
```
process (C \in coordinators)
  variables temp = participants,
            aborted = FALSE; {
    while (temp /= {}) {
      with (r \in temp) {
        Send(self, r, "prepare");
        temp := temp \ {r};
      } };
    temp := participants;
    while (temp /= {} \/ aborted) {
      with (r \in temp) {
        either {
        Receive(r, self, "prepared");
        } or {
        Receive(r, self, "abort");
        aborted := TRUE;
        };
        temp := temp \ {r};
      } };
    if (aborted) {
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Send(coord, r, "abort");
        temp := temp \ {r};
      } };
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Receive(r, coord, "aborted");
        temp := temp \ {r};
      } }
    } else {
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Send(coord, r, "commit");
        temp := temp \ {r};
      } }
      temp := participants;
      while (temp /= {}) {
        with (r \in temp) {
        Receive(r, coord, "committed");
        temp := temp \ {r};
      } } } }
```

```
process (P \in participants) {
  Receive(coord, self, "prepare");
  either {
  psend:
    Send(self, coord, "prepared");
  } or {
    Send(self, coord, "abort");
  };
  either {
    Receive(coord, self, "commit");
    Send(self, coord, "committed");
  } or {
    Receive(coord, self, "abort");
    Send(self, coord, "aborted");
  } }
```

Projection & monitoring

- Choreographic languages/logics (e.g. session types) have a *projection* operation to derive local programs for verification, monitoring, and/or code generation

$$project(a \to b) = \{!\, b, ?\, a\}$$



Dynamic Multirole Session Types

Pierre-Malo Deniélou and Nobuko Yoshida

Department of Computing, Imperial College London

Projection & monitoring

- We define projection across *both* Choreographic PlusCal and TLA$^+$
  - Integrates with existing toolchain
  - Monitoring works for vanilla TLA$^+$ as well (assuming some syntactic conditions)

Projection

Ch. PlusCal

```
VARIABLES v,
inbox, outbox
```

```
choreography (a \in A, b \in B) {
  Transmit(a, b, v, "msg")
}
```

PlusCal

```
VARIABLES v,
inbox, outbox
```

```
process (a \in A) {
  Send(b, "msg")
}
```

```
process (b \in B) {
  v = Receive(a)
}
```

TLA+

```
VARIABLES v,
inbox, outbox
```

```
A_send(self, b) ==
  /\ Send(b, "msg")
  /\ UNCHANGED <<inbox, v>>
```

```
B_send(self, a) ==
  /\ v'[self] := Receive(a)
  /\ UNCHANGED <<outbox>>
```

Multiple TLA+ models

```
VARIABLES inbox, outbox
A_send(b) ==
  /\ Send(b, "msg”)
  /\ UNCHANGED <<inbox>>
```

```
VARIABLES v, inbox, outbox
B_send(a) ==
  /\ v := Receive(a)
  /\ UNCHANGED <<outbox>>
```

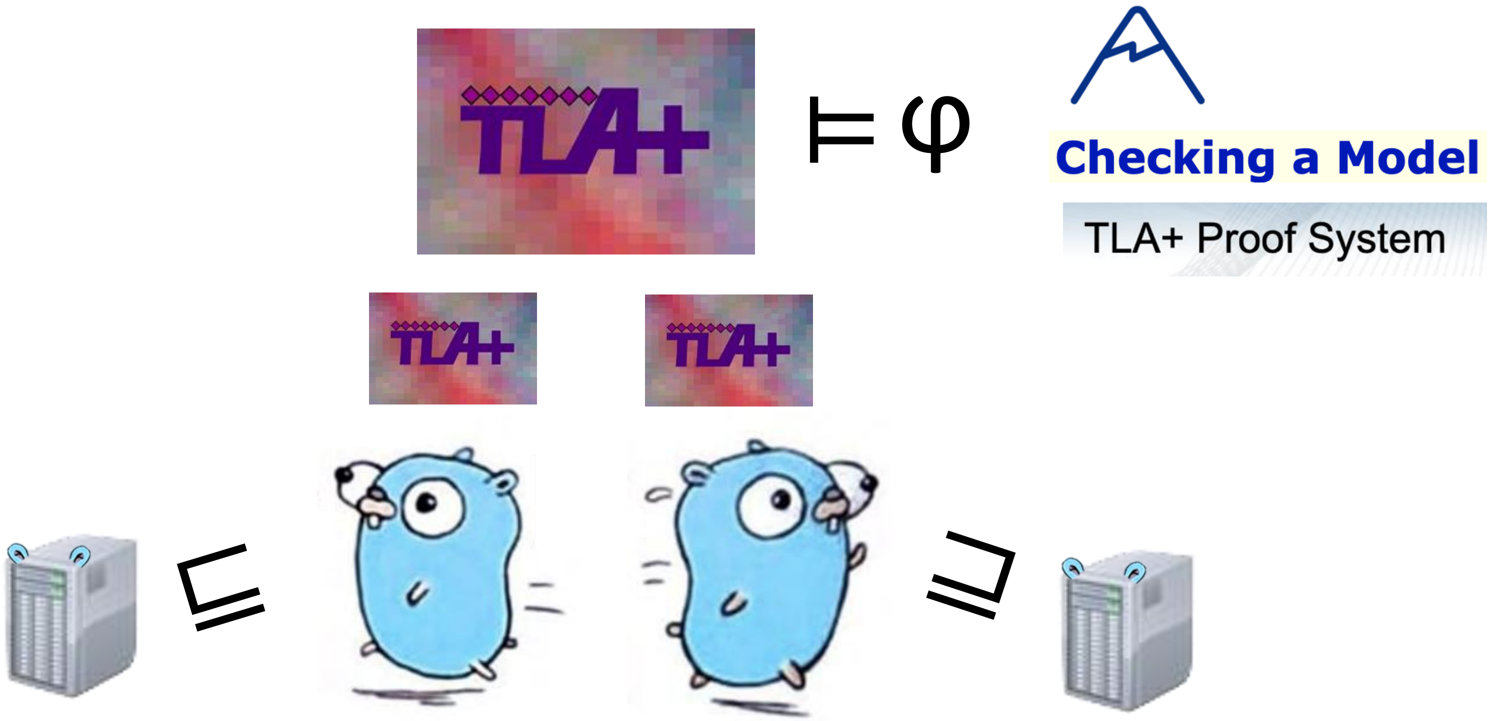
Monitoring

$$
\begin{tikzcd}
\text{Ch. PlusCal } \hat{S} \text{ (Sec. 3)} \arrow[rr, "\rightsquigarrow"] \arrow[d, "\upharpoonright \text{ (Sec. 4)}"] & & \{\tau, \ldots\} \arrow[d, equal] \\
\text{Ex. PlusCal } \bar{S} \arrow[rr, "\rightsquigarrow"] \arrow[d, "\mathit{Tr} \text{ (Sec. 5)}"] & & \{\tau, \ldots\} \arrow[d, equal] \\
\text{PlusCal } S \arrow[rr, "\rightsquigarrow"] \arrow[d, "\text{pcal.trans [22]}"] & & \{\tau, \ldots\} \\
\text{TLA}^{+} \arrow[rr, "\text{Inv.}"] \arrow[d, "\text{Comp. (Sec. 6)}"] & & \text{TLC} \\
\text{Monitor} \arrow[rr, "\text{Instr.}"] & & \text{Test}
\end{tikzcd}
$$

Fig. 4: Overview

Choreographic PlusCal

The PlusCal Algorithm Language



Monitoring

- Instrument system to collect traces
  - Refinement mapping
    - Function from concrete to abstract state
    - Abstracts away details, reinterprets system behavior in terms of the model’s
    - May require auxiliary state to define
    - Deep embedding of TLA+ formulae in Go

```
type TLA interface {
  String() string
  MarshalJSON() ([]byte, error)
}
```

Monitoring

- Instrument system to collect traces
  - Refinement mapping
  - Linearization points
    - Program locations where state changes become visible
    - Can vary significantly between implementations
    - May require auxiliary state to define

Monitoring

- Instrument system to collect traces
- Validate behaviors
  - Model-based trace checking [Pressler 18, Davis 20]
  - Compile model into monitor and validate on the fly
    - Online or offline
    - Scalable, possible to enable in production/fuzzing

Project	Protocol	LoC	Overhead
vadiminshakov/committer	2PC	3032	19% (5 ms)
etcd-io/raft	Raft leader election	21,064	2% (4 ms)

Table 2: Monitor overhead

Conclusion

- Choreographic PlusCal + monitoring
- What's in the paper?
  - Details, formalization, soundness of new features and projection
- Future work
  - Liveness: runtime verification
  - New classes of protocols, e.g. role-parametric
  - User-provided refinement mapping and linearization points are all trusted – statically check

Thank you!

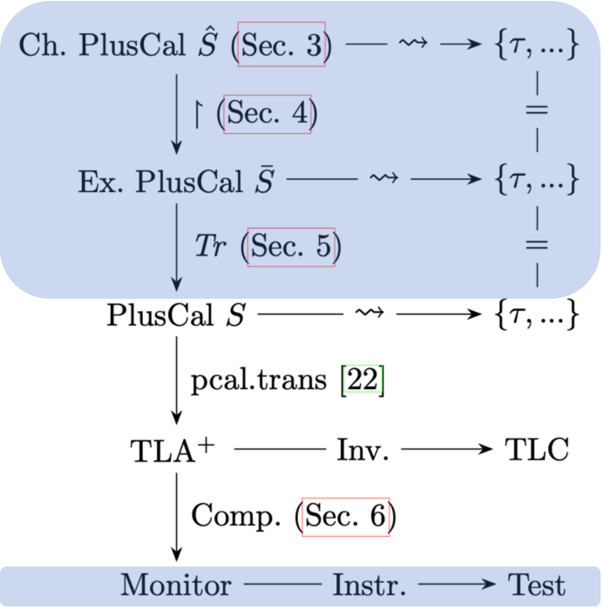
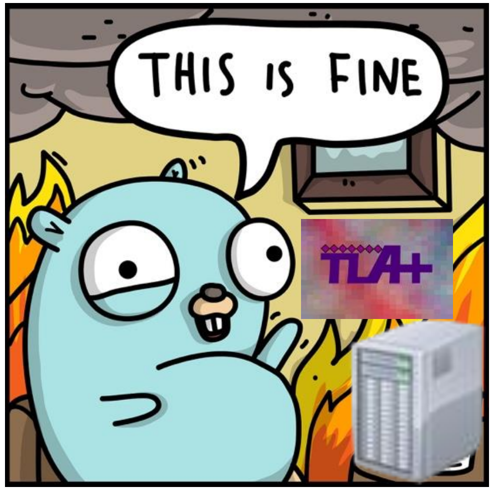


Fig. 4: Overview

Protocol Conformance with Choreographic PlusCal

Darius Foo, Andreea Costea, and Wei-Ngan Chin

National University of Singapore
{dariusf,andreeac,chinwn}@comp.nus.edu.sg



https://github.com/dariusf/tlaplus/tree/cpcal

Monitoring

```go
func psend(prev state, this state, self TLA) bool {
  if !(reflect.DeepEqual(prev.pc, Str("psend"))) {
    return false
  }
  // ... outbox check elided
  if !(reflect.DeepEqual(this.pc, Str("Lbl_2"))) {
    return false
  }
  return true
}
```

Fig. 6: Go rendering of **psend** in generated monitor

Specification with TLA$^+$

```
\* Defines how the variables may transition.
Next == /\ \/ \E i \in Server : Restart(i)
           \/ \E i \in Server : Timeout(i)
           \/ \E i,j \in Server : RequestVote(i, j)
           \/ \E i \in Server : BecomeLeader(i)
           \/ \E i \in Server, v \in Value : ClientRequest(i, v)
           \/ \E i \in Server : AdvanceCommitIndex(i)
           \/ \E i,j \in Server : AppendEntries(i, j)
           \/ \E m \in DOMAIN messages : Receive(m)
           \/ \E m \in DOMAIN messages : DuplicateMessage(m)
           \/ \E m \in DOMAIN messages : DropMessage(m)
           \* History variable that tracks every log ever:
        /\ allLogs' = allLogs \cup {log[i] : i \in Server}
```

Specification with TLA+

Figuring out how actions are related is tedious, e.g. sequentially

```
\* Server i times out and starts a new election.
Timeout(i) == /\ state[i] \in {Follower, Candidate}
              /\ state' = [state EXCEPT ![i] = Candidate]
              /\ currentTerm' = [currentTerm EXCEPT ![i] = currentTerm[i] + 1]
              \* Most implementations would probably just set the local vote
              \* atomically, but messaging localhost for it is weaker.
              /\ votedFor' = [votedFor EXCEPT ![i] = Nil]
              /\ votesResponded' = [votesResponded EXCEPT ![i] = {}]
              /\ votesGranted'   = [votesGranted EXCEPT ![i] = {}]
              /\ voterLog'       = [voterLog EXCEPT ![i] = [j \in {} |-> <<>>]]
              /\ UNCHANGED <<messages, leaderVars, logVars>>
```

```
\* Candidate i sends j a RequestVote request.
RequestVote(i, j) ==
    /\ state[i] = Candidate
    /\ j \notin votesResponded[i]
    /\ Send([mtype         |-> RequestVoteRequest,
             mterm         |-> currentTerm[i],
             mlastLogTerm  |-> LastTerm(log[i]),
             mlastLogIndex |-> Len(log[i]),
             msource       |-> i,
             mdest         |-> j])
    /\ UNCHANGED <<serverVars, candidateVars, leaderVars, logVars>>
```

… must also check if other actions are enabled in Candidate state, else nondeterminism

Specification with TLA$^+$

Figuring out how actions are related is tedious, e.g. send-receive

```
\* Server i receives a RequestVote request from server j with
\* m.mterm <= currentTerm[i].
HandleRequestVoteRequest(i, j, m) ==
    LET logOk == \/ m.mlastLogTerm > LastTerm(log[i])
                 \/ /\ m.mlastLogTerm = LastTerm(log[i])
                    /\ m.mlastLogIndex >= Len(log[i])
        grant == /\ m.mterm = currentTerm[i]
                 /\ logOk
                 /\ votedFor[i] \in {Nil, j}
    IN /\ m.mterm <= currentTerm[i]
       /\ \/ grant  /\ votedFor' = [votedFor EXCEPT ![i] = j]
          \/ ~grant /\ UNCHANGED votedFor
       /\ Reply([mtype        |-> RequestVoteResponse,
                 mterm        |-> currentTerm[i],
                 mvoteGranted |-> grant,
                 \* mlog is used just for the `elections' history variable for
                 \* the proof. It would not exist in a real implementation.
                 mlog         |-> log[i],
                 msource      |-> i,
                 mdest        |-> j],
                 m)
       /\ UNCHANGED <<state, currentTerm, candidateVars, leaderVars, logVars>>
```

```
\* Candidate i sends j a RequestVote request.
RequestVote(i, j) ==
    /\ state[i] = Candidate
    /\ j \notin votesResponded[i]
    /\ Send([mtype         |-> RequestVoteRequest,
             mterm         |-> currentTerm[i],
             mlastLogTerm  |-> LastTerm(log[i]),
             mlastLogIndex |-> Len(log[i]),
             msource       |-> i,
             mdest         |-> j])
    /\ UNCHANGED <<serverVars, candidateVars, leaderVars, logVars>>
```

Must do this repeatedly to get a sense of the flow of the protocol

Specification with TLA$^+$

Non-compositionality

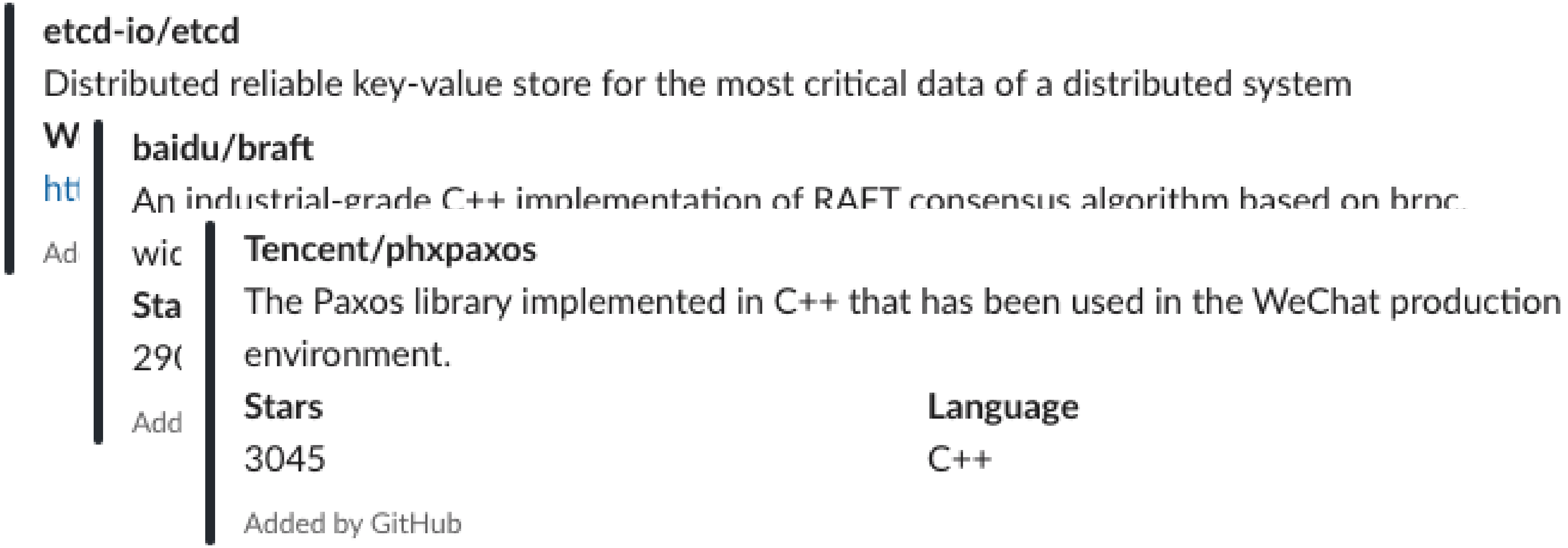
```
\* Add a message to the bag of messages.
Send(m) == messages' = WithMessage(m, messages)
```

```
\* Candidate i sends j a RequestVote request.
RequestVote(i, j) ==
    /\ state[i] = Candidate
    /\ j \notin votesResponded[i]
    /\ Send([mtype         |-> RequestVoteRequest,
             mterm         |-> currentTerm[i],
             mlastLogTerm  |-> LastTerm(log[i]),
             mlastLogIndex |-> Len(log[i]),
             msource       |-> i,
             mdest         |-> j])
    /\ UNCHANGED <<serverVars, candidateVars, leaderVars, logVars>>
```

Must thread state through functions manually

Linking specification to implementation

Many “industrial-grade” unverified protocol implementations…



Linking specification to implementation

… many specifications as well, but unrelated

List of Examples

No	Name	Short description	Spec's authors	TLAPS Proof	TLC Check
39	MultiPaxos	The abstract specification of Generalized Paxos (Lamport, 2004)	Giuliano Losa		✓
45	Paxos	Paxos consensus algorithm (Lamport, 1998)	Leslie Lamport		✓
47	raft	Raft consensus algorithm (Ongaro, 2014)	Diego Ongaro		✓
57	transaction_commit	Consensus on transaction commit (Gray & Lamport, 2006)	Leslie Lamport		✓
67	Tencent-Paxos	PaxosStore: high-availability storage made practical in WeChat. Proceedings of the VLDB Endowment(Zheng et al., 2017)	Xingchen Yi, Hengfeng Wei	✓	✓

59	TwoPhase	Two-phase handshaking	Leslie Lamport, Stephan Merz		✓	Nat
62	Misra Reachability Algorithm	Misra Reachability Algorithm	Leslie Lamport	✓	✓	Int, Seq, FiniteS TLC, TLAPS, NaturalsInductic
63	Loop Invariance	Loop Invariance	Leslie Lamport	✓	✓	Int, Seq, FiniteS TLC, TLAPS, SequenceTheor NaturalsInductic
69	Paxos	Paxos			✓	Int, FiniteSets
75	Lock-Free Set	PlusCal spec of a lock-Free set used by TLC	Markus Kuppe		✓	Sequences, FiniteSets, Integ TLC
77	ParallelRaft	A variant of Raft	Xiaosong Gu, Hengfeng Wei, Yu Huang		✓	Integers, FiniteS Sequences, Naturals
83	Raft (with cluster changes)	Raft with cluster changes, and a version with Apalache type annotations but no cluster changes	George Pîrlea, Darius Foo, Brandon Amos, Huanchen Zhang, Daniel Ricketts		✓	Functions, SequencesExt, FiniteSetsExt, TypedBags