fmcad.23

Automating Cutoff-based Verification of Distributed Protocols

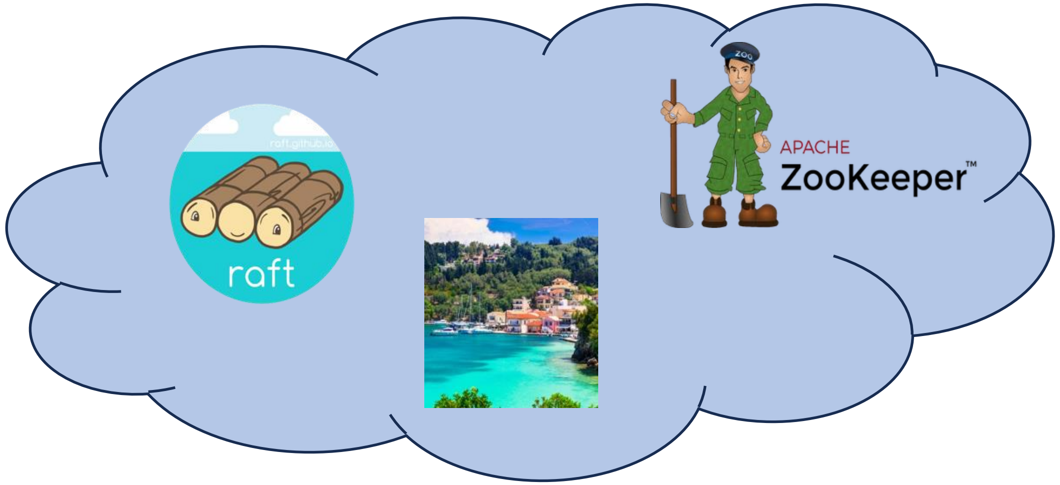
Shreesha G. Bhat* and Kartik Nagar

Department of CSE, IIT Madras

*currently PhD student at University of Illinois Urbana-Champaign

Distributed Protocols

- Independent nodes communicate → accomplish task
- Backbone of modern-day cloud systems
- **Used in correctness critical systems** - incorrect protocols have disastrous consequences
- Need for verification!

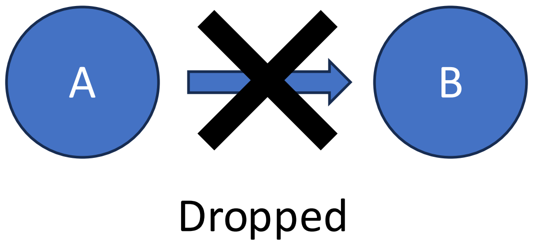


Distributed Protocols

- *Parametric* nature - must work with *any* number of nodes
- Send and receive messages - must work under *adverse network conditions*

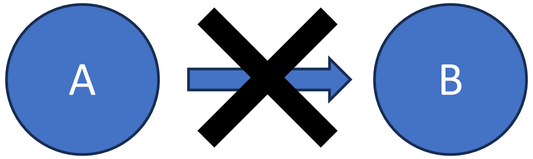
Distributed Protocols

- *Parametric* nature - must work with *any* number of nodes
- Send and receive messages - must work under *adverse network conditions*

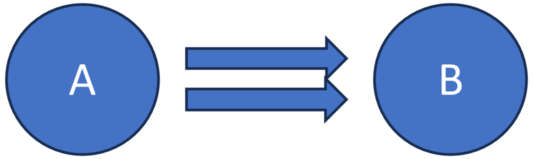


Distributed Protocols

- *Parametric* nature - must work with *any* number of nodes
- Send and receive messages - must work under *adverse network conditions*



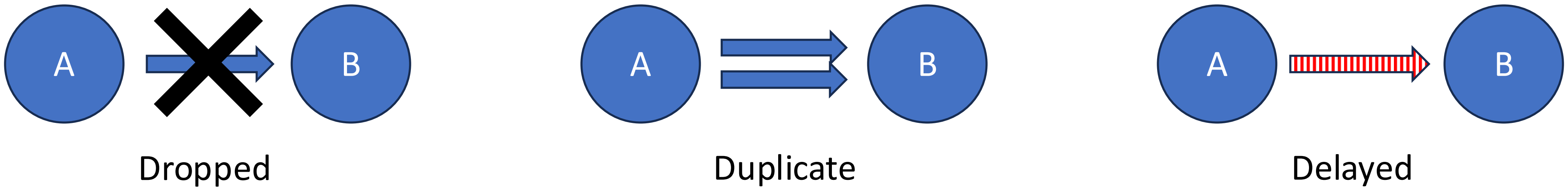
Dropped



Duplicate

Distributed Protocols

- *Parametric* nature - must work with *any* number of nodes
- Send and receive messages - must work under *adverse network conditions*

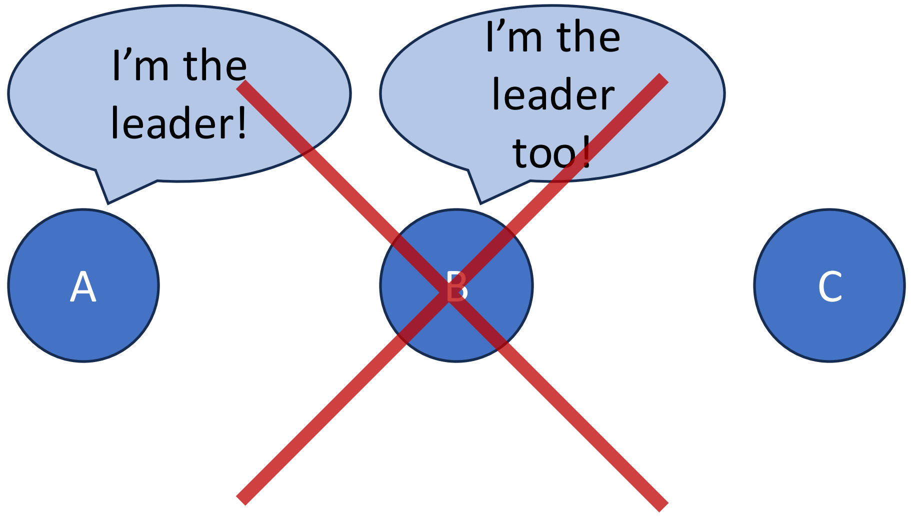


Specifying Correctness

- **Safety Property** - describes ”bad states” that should never be reached
- Must always be obeyed by all nodes

Specifying Correctness

- **Safety Property** - describes ”bad states” that should never be reached
- Must always be obeyed by all nodes



Challenges with Verification

- Complex designs - co-ordination is hard!

- Designed to work under all possible network behaviors. Must reason about protocol behavior under these conditions

- Parametric nature → ∞ number of possible instantiations

- Subtle behaviors and corner cases are easy to miss

Example Protocol: Sharded Key-Value Store

A	
k_1	v_1
k_2	v_2

B	
k_3	v_3

C	
k_4	v_4

Example Protocol: Sharded Key-Value Store

put(k_1, v_1')

A	
k_1	v_1
k_2	v_2

B	
k_3	v_3

C	
k_4	v_4

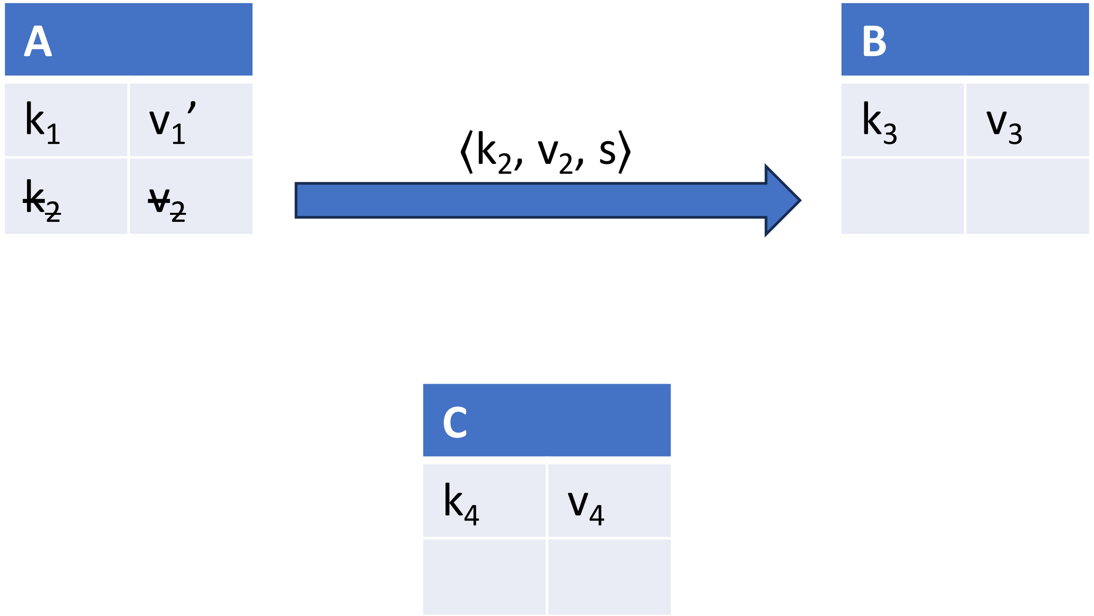
Example Protocol: Sharded Key-Value Store

A	
k_1	v_1'
k_2	v_2

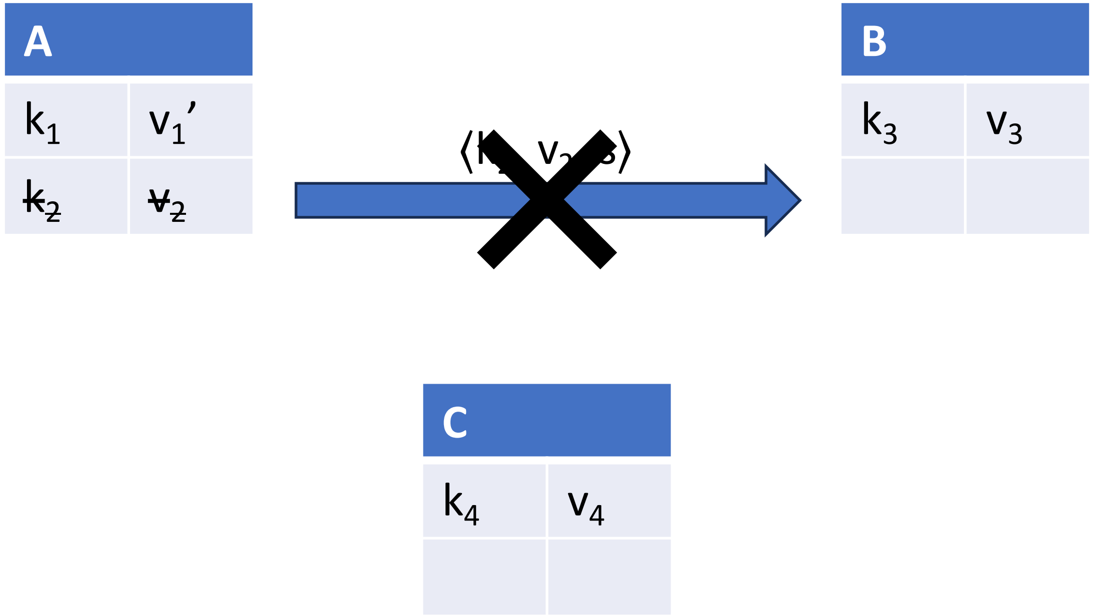
B	
k_3	v_3

C	
k_4	v_4

Example Protocol: Sharded Key-Value Store



Example Protocol: Sharded Key-Value Store



Example Protocol: Sharded Key-Value Store

A	
k_1	v_1’
~~k_2~~	~~v_2~~

B	
k_3	v_3

C	
k_4	v_4

Example Protocol: Sharded Key-Value Store

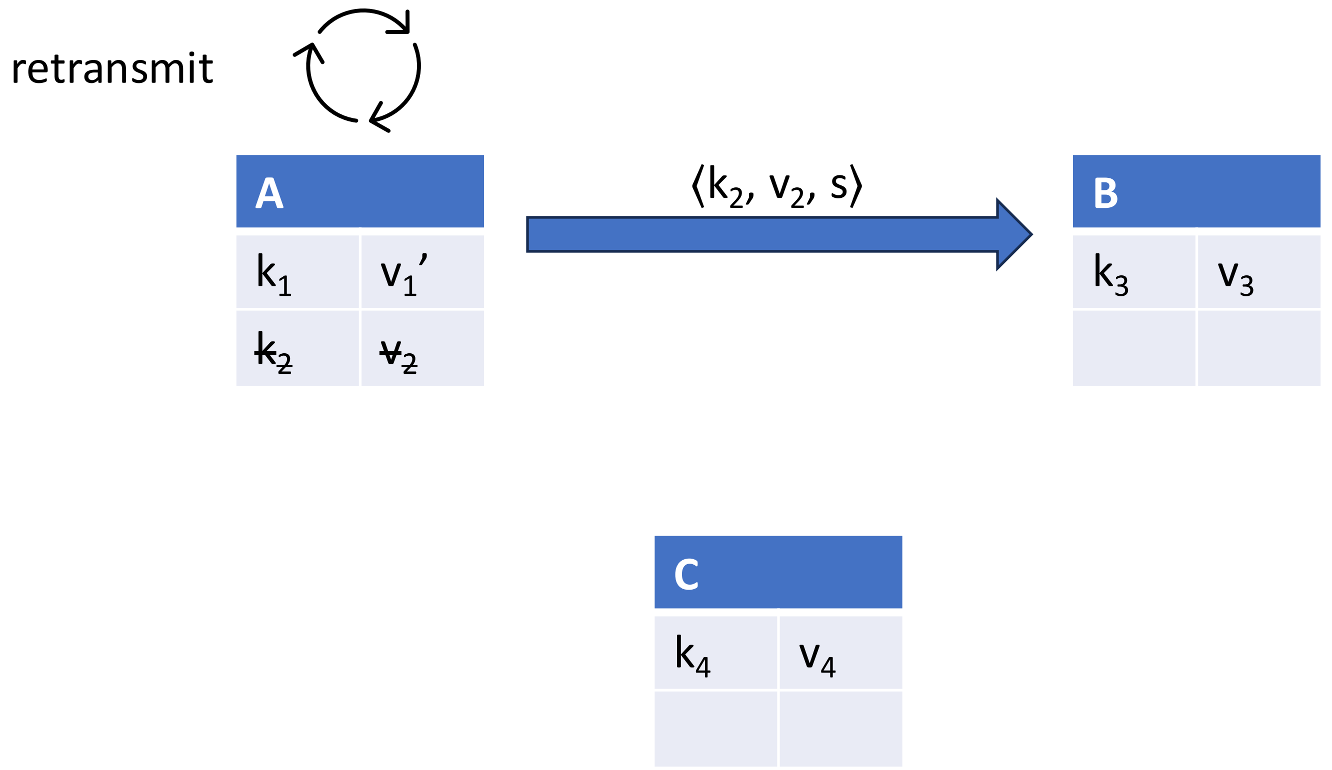
retransmit

A	
k_1	v_1’
$\cancel{k_2}$	$\cancel{v_2}$

B	
k_3	v_3

C	
k_4	v_4

Example Protocol: Sharded Key-Value Store



Example Protocol: Sharded Key-Value Store

A	
k_1	v_1'
$\cancel{k_2}$	$\cancel{v_2}$

B	
k_3	v_3
k_2	v_2

C	
k_4	v_4

Example Protocol: Sharded Key-Value Store

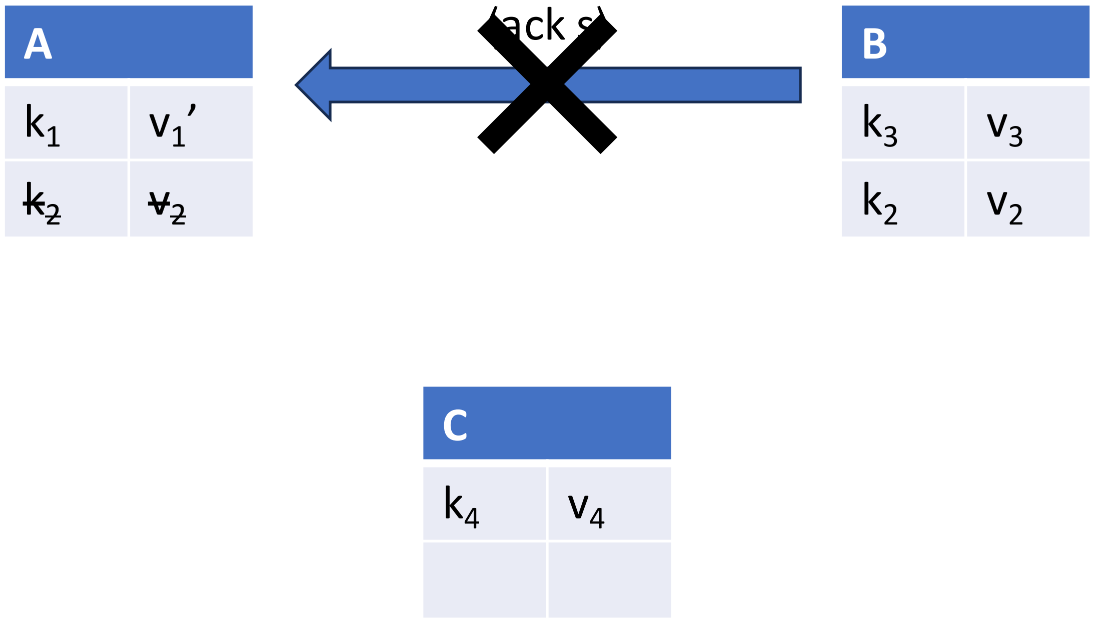
A	
k_1	v_1'
$\cancel{k_2}$	$\cancel{v_2}$

$\xleftarrow{\langle \text{ack s} \rangle}$

B	
k_3	v_3
k_2	v_2

C	
k_4	v_4

Example Protocol: Sharded Key-Value Store



Example Protocol: Sharded Key-Value Store

A	
k_1	v_1'
$\cancel{k_2}$	$\cancel{v_2}$

B	
k_3	v_3
k_2	v_2

C	
k_4	v_4

Example Protocol: Sharded Key-Value Store

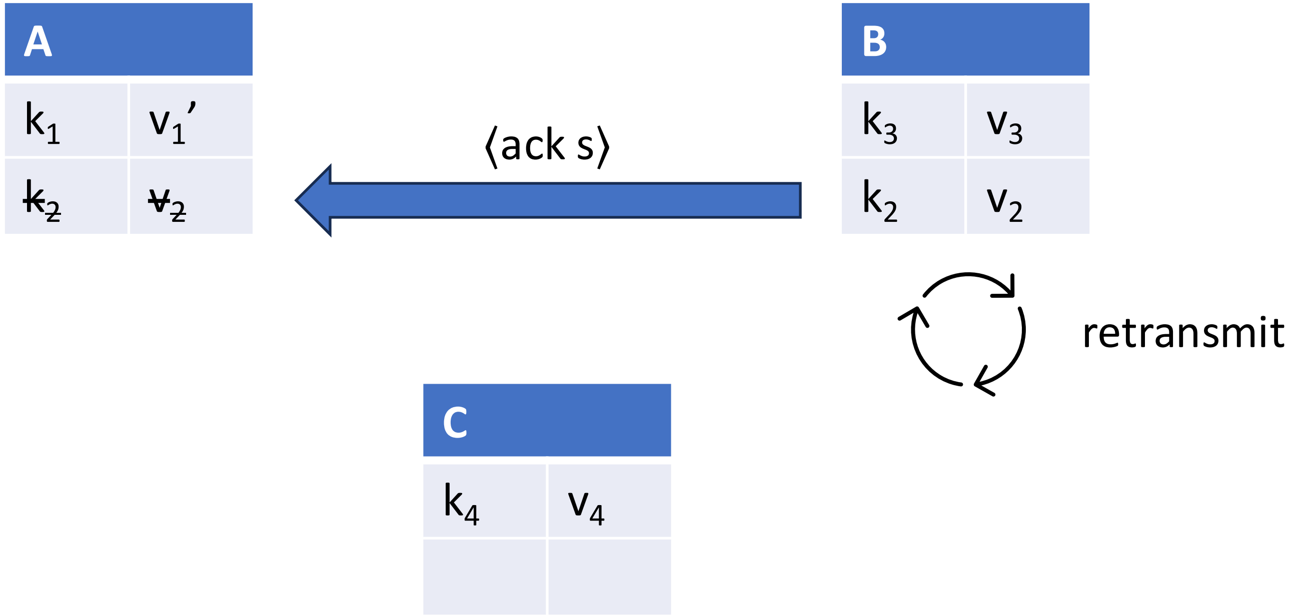
A	
k_1	v_1'
$\cancel{k_2}$	$\cancel{v_2}$

B	
k_3	v_3
k_2	v_2

retransmit

C	
k_4	v_4

Example Protocol: Sharded Key-Value Store



Example Protocol: Sharded Key-Value Store

A	
k_1	v_1'

B	
k_3	v_3
k_2	v_2

C	
k_4	v_4

Example Protocol: Sharded Key-Value Store

Safety – a single associated value per key across the nodes

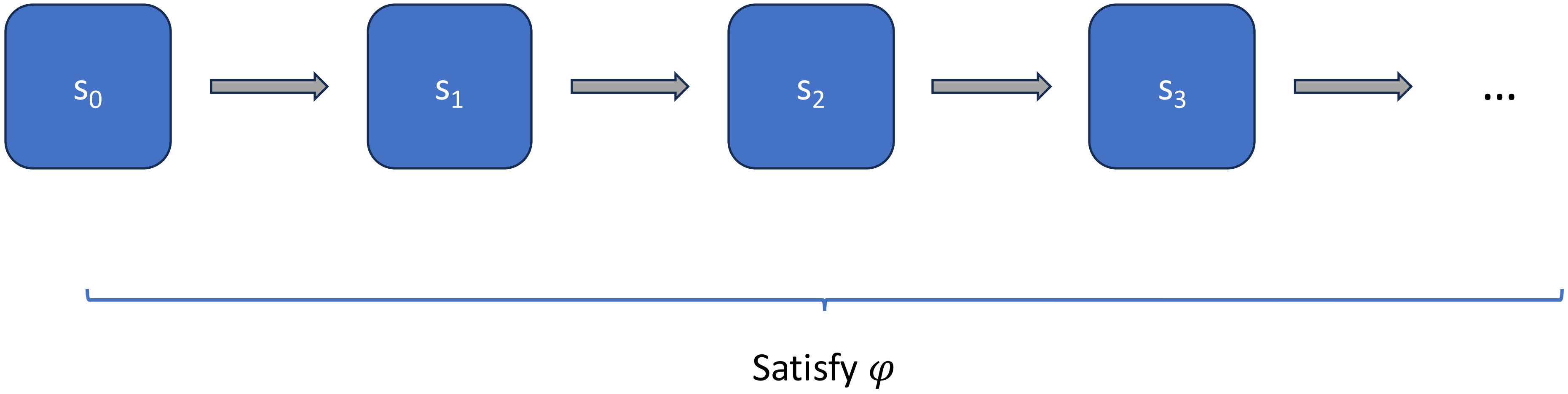
Why sequence numbers?

- Unique, not reused
- Distinguish stale transfers from new ones
- Prevent safety issues
  - Old key is re-entered after subsequent transfer
  - Old key is re-entered after subsequent KVP modification

Complex logic!

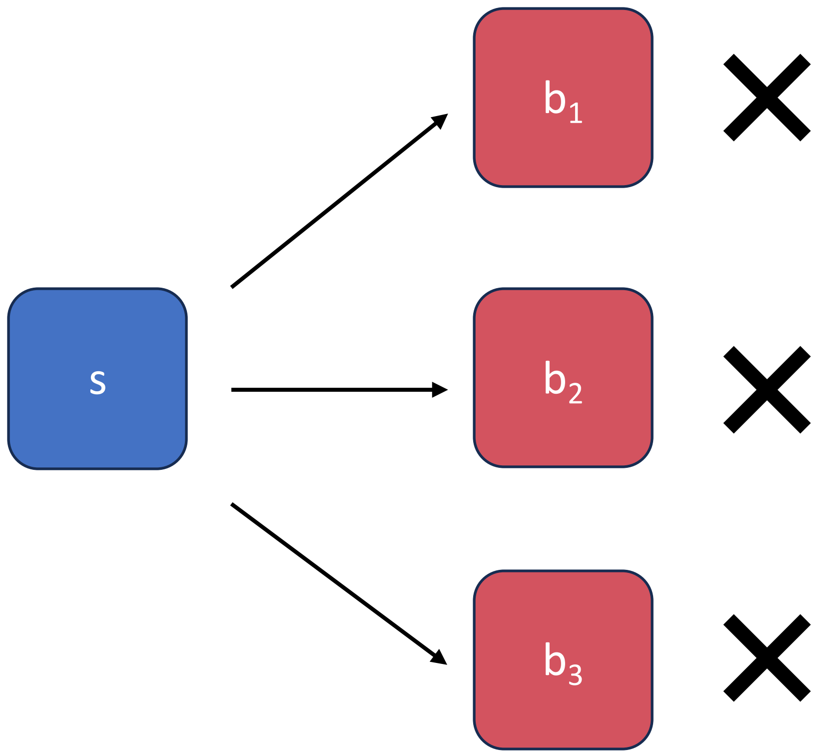
Verifying Distributed Protocols: Inductive Invariants

- Property φ that is satisfied at each step
- Strong enough to imply the safety property



Verifying Distributed Protocols: Inductive Invariants

- Traditional approach to verifying distributed protocols
- Hard to automatically synthesize
  - Must address how protocol blocks all bad behaviors
  - Cannot avoid intricacies of protocol



Cutoff-based Approach

- ‘small scope hypothesis’ – erroneous behaviors occur within small scopes
- Cutoff instance – fixed size instance such that a violation in any arbitrary sized instance *can be re-produced* in the cutoff instance
- Correctness of cutoff instance implies correctness of any arbitrary sized instance
- Cutoff instance is of fixed, finite size → correctness established by finite-state model checking

Cutoff-based Approach: Advantages

Cleanly separates the two main roadblocks with verification

Dealing with infinite instantiations

↓

Having a *correct* cutoff instance absolves us from dealing with infinite instantiations

Reasoning about complex protocol and network behavior

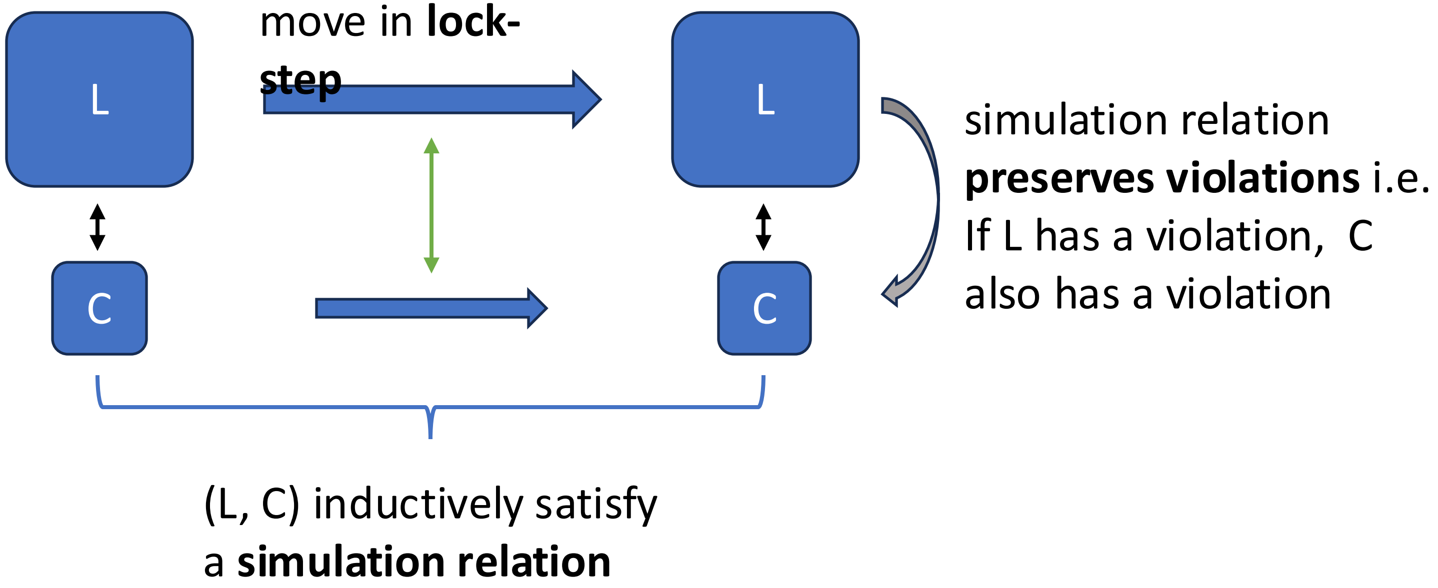
↓

Complexity encapsulated within the cutoff instance. Delegated to finite-state model checking!

Our Contributions

- Automate the process of finding and proving a cutoff instance
- *Static Analysis* - identify key state components and actions responsible for a violation in any instance
- *Simulate* this violation in the cutoff instance
- Efficient encoding of validity of cutoff instance in SMT
- **Generalizable** across classes of protocols!

Simulation-based approach



Motivating Example: Sharded Key-Value Store

type $key, value, node, seqnum$
relation $table : node, key, value$
relation $transfer\_msg : node, node, key, value, seqnum$
relation $ack\_msg : node, node, seqnum$
relation $seqnum\_sent : node, seqnum$
relation $unacked : node, node, key, value, seqnum$
relation $seqnum\_recvd : node, node, seqnum$
init $\forall n_1, n_2, k, v_1 .\ table(n_1, k, v_1) \wedge table(n_2, k, v_2) \implies n_1 = n_2 \wedge v_1 = v_2$ $\triangleright$ *All other relations are empty*

Feldman, Y.M.Y., Wilcox, J.R., Shoham, S., Sagiv, M. (2019). Inferring Inductive Invariants from Phase Structures. In: Dillig, I., Tasiran, S. (eds) Computer Aided Verification. CAV 2019. Lecture Notes in Computer Science(), vol 11562. Springer, Cham. https://doi.org/10.1007/978-3-030-25543-5_23

Motivating Example: Sharded Key-Value Store

parameters

action Put($n : node, k : key, v : value$)
  require $\exists v'. table(n, k, v')$
  $table(n, k, *) \leftarrow false$
  $table(n, k, v) \leftarrow true$

guard

Feldman, Y.M.Y., Wilcox, J.R., Shoham, S., Sagiv, M. (2019). Inferring Inductive Invariants from Phase Structures. In: Dillig, I., Tasiran, S. (eds) Computer Aided Verification. CAV 2019. Lecture Notes in Computer Science(), vol 11562. Springer, Cham. https://doi.org/10.1007/978-3-030-25543-5_23

Motivating Example: Sharded Key-Value Store

parameters

action Put($n : node, k : key, v : value$)
    require $\exists v'.\, table(n, k, v')$
    $table(n, k, *) \leftarrow false$
    $table(n, k, v) \leftarrow true$

guard

action Reshard($n\_old : node, n\_new : node, k : key, v : value, s : seqnum$)
    require $table(n\_old, k, v) \wedge \neg seqnum\_sent(s)$
    $seqnum\_sent(s) \leftarrow true$
    $table(n\_old, k, v) \leftarrow false$
    $transfer\_msg(n\_old, n\_new, k, v, s) \leftarrow true$
    $unacked(n\_old, n\_new, k, v, s) \leftarrow true$

Feldman, Y.M.Y., Wilcox, J.R., Shoham, S., Sagiv, M. (2019). Inferring Inductive Invariants from Phase Structures. In: Dillig, I., Tasiran, S. (eds) Computer Aided Verification. CAV 2019. Lecture Notes in Computer Science(), vol 11562. Springer, Cham. https://doi.org/10.1007/978-3-030-25543-5_23

Motivating Example: Sharded Key-Value Store

parameters

action Put($n : node, k : key, v : value$)
  require $\exists v'.\, table(n, k, v')$
  $table(n, k, *) \leftarrow false$
  $table(n, k, v) \leftarrow true$

guard

action DropTransferMsg($src : node, dst : node, k : key, v : value, s : seqnum$)
  require $transfer\_msg(src, dst, k, v, s)$
  $transfer\_msg(src, dst, k, v, s) \leftarrow false$
action Retransmit($src : node, dst : node, k : key, v : value, s : seqnum$)
  require $unacked(src, dst, k, v, s)$
  $transfer\_msg(src, dst, k, v, s) \leftarrow true$

action Reshard($n\_old : node, n\_new : node, k : key, v : value, s : seqnum$)
  require $table(n\_old, k, v) \wedge \neg seqnum\_sent(s)$
  $seqnum\_sent(s) \leftarrow true$
  $table(n\_old, k, v) \leftarrow false$
  $transfer\_msg(n\_old, n\_new, k, v, s) \leftarrow true$
  $unacked(n\_old, n\_new, k, v, s) \leftarrow true$

Feldman, Y.M.Y., Wilcox, J.R., Shoham, S., Sagiv, M. (2019). Inferring Inductive Invariants from Phase Structures. In: Dillig, I., Tasiran, S. (eds) Computer Aided Verification. CAV 2019. Lecture Notes in Computer Science(), vol 11562. Springer, Cham. https://doi.org/10.1007/978-3-030-25543-5_23

Motivating Example: Sharded Key-Value Store

action RecvTransferMsg($src : node, dst : node, k : key, v : value, s : seqnum$)
    require $transfer\_msg(src, dst, k, v, s) \wedge \neg seqnum\_recvd(s)$
    $seqnum\_recvd(s) \leftarrow true$
    $table(dst, k, v) \leftarrow true$

Feldman, Y.M.Y., Wilcox, J.R., Shoham, S., Sagiv, M. (2019). Inferring Inductive Invariants from Phase Structures. In: Dillig, I., Tasiran, S. (eds) Computer Aided Verification. CAV 2019. Lecture Notes in Computer Science(), vol 11562. Springer, Cham. https://doi.org/10.1007/978-3-030-25543-5_23

Motivating Example: Sharded Key-Value Store

action RecvTransferMsg$(src : node, dst : node, k : key, v : value, s : seqnum)$
  require $transfer\_msg(src, dst, k, v, s) \wedge \neg seqnum\_recvd(s)$
  $seqnum\_recvd(s) \leftarrow true$
  $table(dst, k, v) \leftarrow true$

action SendAck$(src : node, dst : node, k : key, v : value, s : seqnum)$
  require $transfer\_msg(src, dst, k, v, s) \wedge seqnum\_recvd(s)$
  $ack\_msg(s) \leftarrow true$
action DropAckMsg$(src : node, dst : node, k : key, v : value, s : seqnum)$
  require $ack\_msg(s)$
  $ack\_msg(s) \leftarrow false$
action RecvAckMsg$(src : node, dst : node, k : key, v : value, s : seqnum)$
  require $ack\_msg(s)$
  $unacked(src, dst, k, v, s) \leftarrow false$

Feldman, Y.M.Y., Wilcox, J.R., Shoham, S., Sagiv, M. (2019). Inferring Inductive Invariants from Phase Structures. In: Dillig, I., Tasiran, S. (eds) Computer Aided Verification. CAV 2019. Lecture Notes in Computer Science(), vol 11562. Springer, Cham. https://doi.org/10.1007/978-3-030-25543-5_23

Motivating Example: Sharded Key-Value Store

action RecvTransferMsg$(src : node, dst : node, k : key, v : value, s : seqnum)$
  require $transfer\_msg(src, dst, k, v, s) \wedge \neg seqnum\_recvd(s)$
  $seqnum\_recvd(s) \leftarrow true$
  $table(dst, k, v) \leftarrow true$

action SendAck$(src : node, dst : node, k : key, v : value, s : seqnum)$
  require $transfer\_msg(src, dst, k, v, s) \wedge seqnum\_recvd(s)$
  $ack\_msg(s) \leftarrow true$
action DropAckMsg$(src : node, dst : node, k : key, v : value, s : seqnum)$
  require $ack\_msg(s)$
  $ack\_msg(s) \leftarrow false$
action RecvAckMsg$(src : node, dst : node, k : key, v : value, s : seqnum)$
  require $ack\_msg(s)$
  $unacked(src, dst, k, v, s) \leftarrow false$

safety $\forall k, n_1, n_2, v_1, v_2, k.\ table(n_1, k, v_1) \wedge table(n_2, k, v_2) \implies n_1 = n_2 \wedge v_1 = v_2$

Feldman, Y.M.Y., Wilcox, J.R., Shoham, S., Sagiv, M. (2019). Inferring Inductive Invariants from Phase Structures. In: Dillig, I., Tasiran, S. (eds) Computer Aided Verification. CAV 2019. Lecture Notes in Computer Science(), vol 11562. Springer, Cham. https://doi.org/10.1007/978-3-030-25543-5_23

Identifying Key State Components and Actions

- Instantiate a violation in an arbitrary instance

$\texttt{table}(A_L, K, V_1) \wedge \texttt{table}(B_L, K, V_2) \longrightarrow S_{init} = \{ \texttt{table}(\langle A_L/B_L\rangle, K, \langle V_1/V_2\rangle) \}$

- How would such a state manifest? What actions set these relation entries?

action RecvTransferMsg$(src : node, dst : node, k : key, v : value, s : seqnum)$
  require $transfer\_msg(src, dst, k, v, s) \wedge \neg seqnum\_recvd(s)$
  $seqnum\_recvd(s) \leftarrow true$
  $\boxed{table(dst, k, v) \leftarrow true}$

action Put$(n : node, k : key, v : value)$
  require $\exists v'.\, table(n, k, v')$
  $table(n, k, *) \leftarrow false$
  $\boxed{table(n, k, v) \leftarrow true}$

- Collect relevant actions

$A = \{ \texttt{RecvTransferMsg}(*, \langle A_L/B_L\rangle, K, \langle V_1/V_2\rangle, *), \texttt{Put}(\langle A_L/B_L\rangle, K, \langle V_1/V_2\rangle) \}$

Identifying Key State Components and Actions

- What state components are required for these actions to fire → examine guards of actions

action RecvTransferMsg($src : node, dst : node, k : key, v : value, s : seqnum$)
  require $\boxed{transfer\_msg(src, dst, k, v, s) \wedge \neg seqnum\_recvd(s)}$
  $seqnum\_recvd(s) \leftarrow true$
  $table(dst, k, v) \leftarrow true$

action Put($n : node, k : key, v : value$)
  require $\boxed{\exists v'.\, table(n, k, v')}$
  $table(n, k, *) \leftarrow false$
  $table(n, k, v) \leftarrow true$

- Add these entries to the set of relevant state components

$$S = \{\ table(\langle A_L/B_L\rangle, K, \langle V_1/V_2\rangle),\ transfer\_msg(*, \langle A_L/B_L\rangle, K, \langle V_1/V_2\rangle, *),\ table(\langle A_L/B_L\rangle, K, *)\ \}$$

- Iterate until convergence!

Identifying Key Components and Actions

Algorithm 4 STATICANALYSIS

Arguments: P the program, S_{init} a set of clauses
Returns: S a set of clauses, A a set of action invocations

procedure STATICANALYSIS(P, S_{init})
  $S \leftarrow S_{init}$
  $S_{prev} \leftarrow \emptyset$
  $A \leftarrow \emptyset$
  while $S \neq S_{prev}$ **do**
    $S_d \leftarrow S \setminus S_{prev}$
    $S_{prev} \leftarrow S$
    for each clause c in S_d **do** ▷ *For each new clause*
      $A_t \leftarrow$ ACTIONSTHATSET(P, c)
      for each action invocation act in A_t **do**
        $S \leftarrow S \cup$ GUARDSFOR(P, act)
      $A \leftarrow A \cup A_t$
  return S, A

```
S = {  table(*, K, *),
       transfer_msg(*, *, K, *, *),
       ¬seqnum_recvd(*),
       ¬seqnum_sent(*),
       unacked(*, *, K, *, *) }

A = {  Put(*, K, *),
       RecvTransferMsg(*, *, K, *, *),
       Reshard(*, *, K, *, *),
       Retransmit(*, *, K, *, *) }
```

Observations

- Original protocol → 8 actions, static analysis → 4 actions – Most actions are not relevant to simulate a violation!
- How does protocol avoid violation → complex!!
- How to simulate hypothetical violation → avoid complexity
- Need only ensure that these components are preserved in cutoff system to simulate violation

Cutoff Instance

How many nodes do we need?

- Instantiate violation of safety property → `table(`A_L`, K, `V_1`)` ∧ `table(`B_L`, K, `V_2`)`

- Violations require 2 instantiated nodes → use cutoff instance of size 2

Synthesizing the components of the simulation

Consider the arbitrary sized instance L and cutoff instance C

- **Lock-step** – Which action(s) taken in C for every action in L?
- **Simulation relation** – Inductive property satisfied by twin systems when they progress as per lock-step. Must preserve violations!

Derive above from static analysis

- Every action in set A is simulated by a corresponding action in C
- Simulation relation ensures every component in set S is present in C

The last piece: *sim* mapping

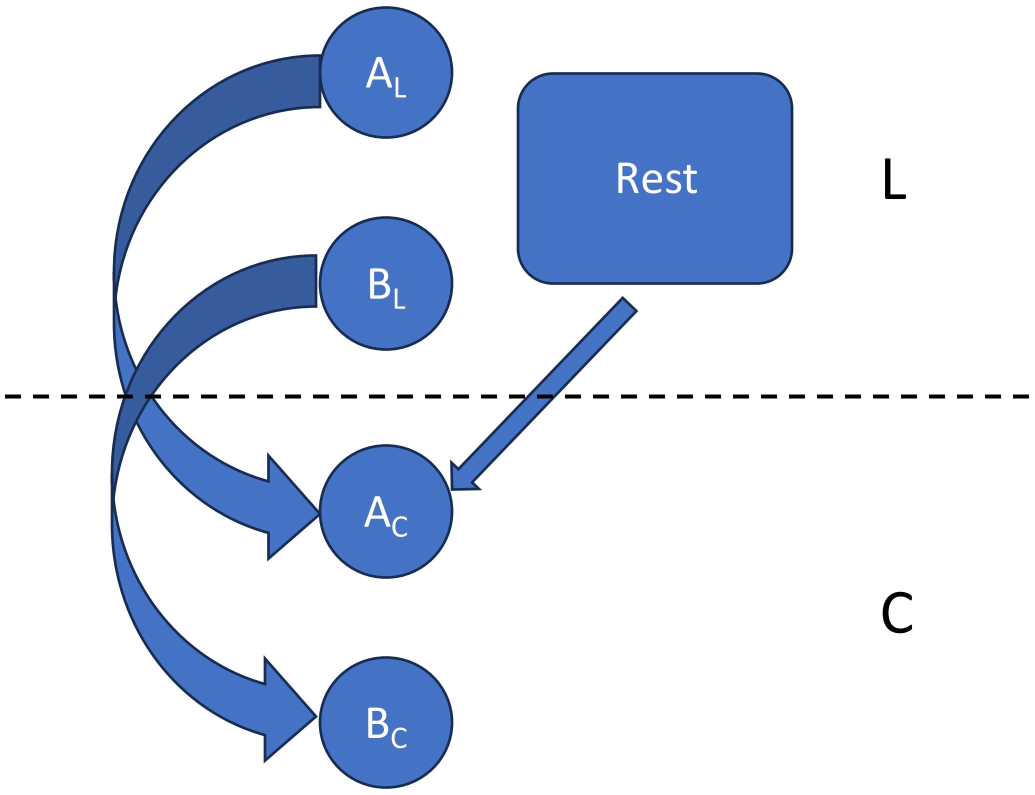
How to map state components and actions from L to C?

- Map nodes from L to C!
- *sim* : nodes in L → nodes in C

Simple mapping strategies work in practice!

Assume nodes in C are A_C and B_C

- Map A_L to A_C and B_L to B_C
- Rest of the nodes in L all map to one of A_C or B_C



Putting it all together

Lock-step

(1) $\mathsf{Put_L}(n, K, v)$ **is simulated as** $\mathsf{Put_C}(sim(n), K, v)$

(2) $\mathsf{Reshard_L}(n_1, n_2, K, v, s)$ **is simulated as** $\mathsf{Reshard_C}(sim(n_1), sim(n_2), K, v, s)$

(3) $\mathsf{Retransmit_L}(n_1, n_2, K, v, s)$ **is simulated as** $\mathsf{Retransmit_C}(sim(n_1), sim(n_2), K, v, s)$

(4) $\mathsf{RecvTransferMsg_L}(n_1, n_2, K, v, s)$ **is simulated as** $\mathsf{RecvTransferMsg_C}(sim(n_1), sim(n_2), K, v, s)$

Simulation relation

(1) $table_L(n, K, v) \implies table_C(sim(n), K, v)$

(2) $unacked_L(n_1, n_2, K, v, s) \implies unacked_C(sim(n_1), sim(n_2), K, v, s)$

(3) $\neg seqnum\_sent_L(s) \implies \neg seqnum\_sent_C(s)$

(4) $\neg seqnum\_recvd_L(s) \implies \neg seqnum\_recvd_C(s)$

(5) $transfer\_msg_L(n_1, n_2, K, v, s) \implies transfer\_msg_C(sim(n_1), sim(n_2), K, v, s)$

SMT Encoding

We encode the correctness of the simulation in SMT. Three properties need to hold

- φ_{init} – initial states of L and C satisfy simulation relation
- φ_{step} – simulation relation holds inductively as L and C move as-per lock-step
- φ_{safety} – simulation relation ensures that if L is in a violating state, so is C

Evaluation and Results

- Using Z3 as backend SMT solver, we apply this approach on a variety of distributed protocols
- Approach is generalizable across classes of protocols
- For more details, refer to our paper

Protocol	**Cutoff**	**Time Taken(s)**	$\|\gamma\|$
Sharded Key-Value Store[15]	2	0.02	5
Leader Election in a Ring[16]	2	0.03	4
Centralized Lock Server[17]	2	0.02	5
Lock Server Sync[18]	2	0.01	2
Ricart Agrawala[19]	2	0.01	6
Two Phase Commit[20]	2	0.02	9
Toy Consensus ForAll[18]	1	0.07	5
Consensus[18]	2	29.7	11

TABLE II: γ *is a FOL formula of the type* $\bigwedge_{i=1}^{|\gamma|}(p \implies q)$ *therefore* $|\gamma|$ *represents the number of clauses of the type* $p \implies q$ *in the simulation relation. Time taken refers to the total time taken by our synthesis+verification procedure.*

Limitations

Current approach can fail in one of two ways

- Cutoff value could be higher than the one chosen in our analysis
- Simulation relation and lock-step do not work i.e. one of φ_{init} , φ_{step} or φ_{safety} do not hold

Our work takes the first step in formalizing and generalizing the ’small scope’ hypothesis for distributed protocols

Conclusions

- We automate and mechanize cutoff-based proofs for distributed protocols
- Cutoff-based approaches allow us to avoid reasoning about protocol intricacies
- Focus on simulating hypothetical violations in a small instance
- Results show that cutoff-approaches are generalizable across classes of protocols
- We hope that this work paves the way for more investigations into automating cutoff proofs for more complex protocols