Introduction to formal verification

TechTalk @ Google
2022-08-11

Jieung Kim

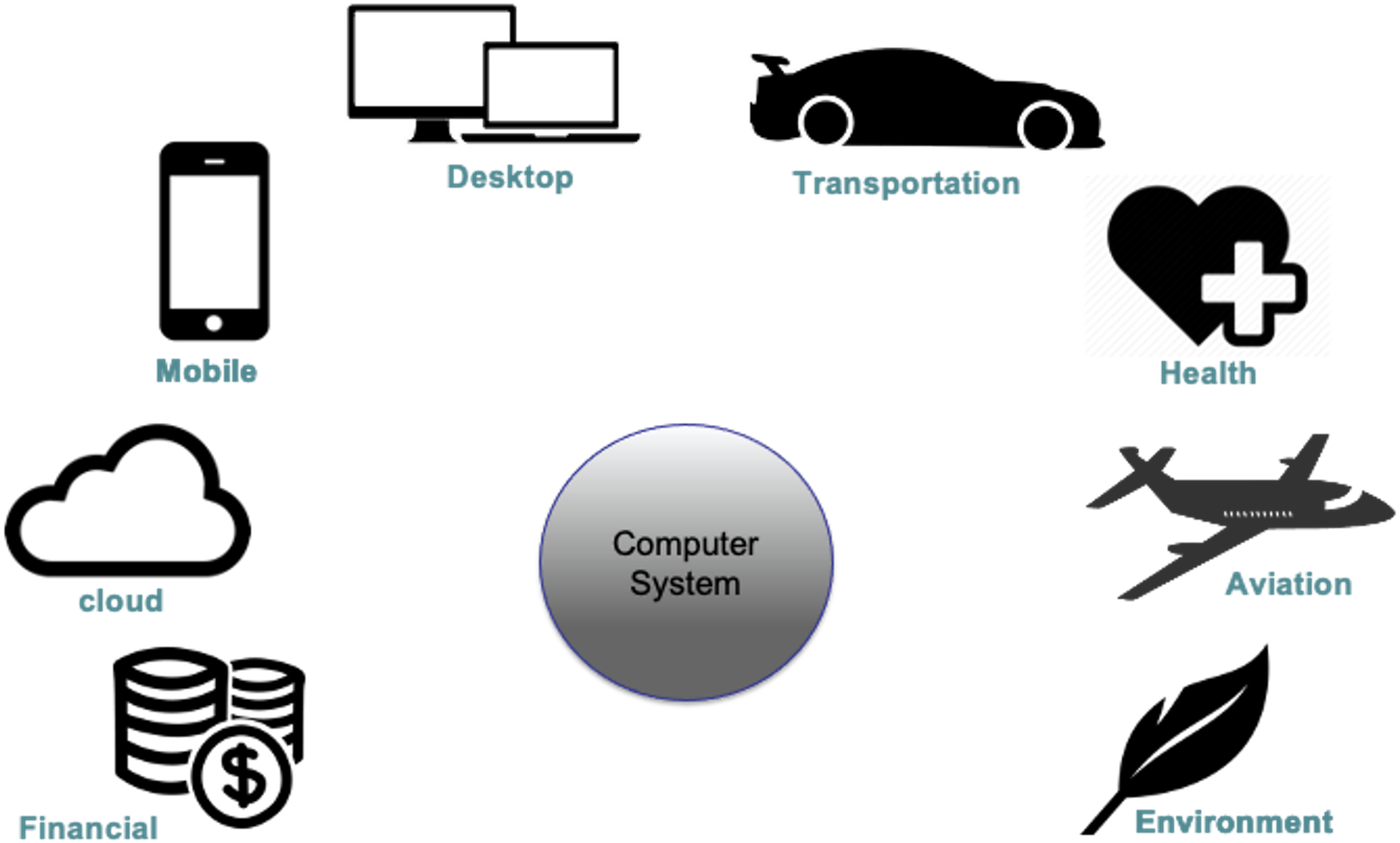
Contents

Contents

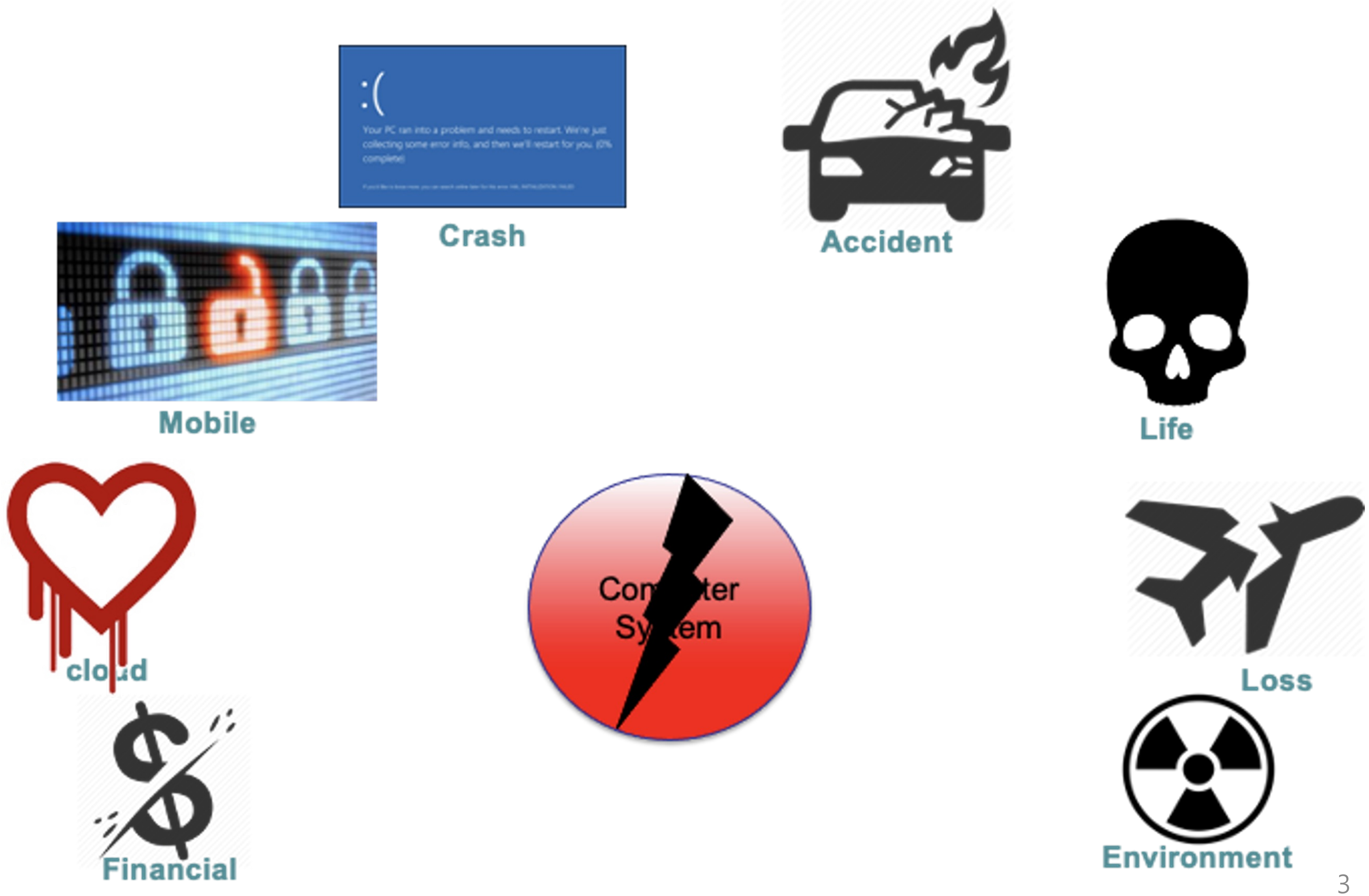
- Intro - Do we need formal verification?
- Formal verification intro with examples
- Formal verification project example
- Conclusion

Intro –
Do we need
formal verification?

Software in the world



Software failure



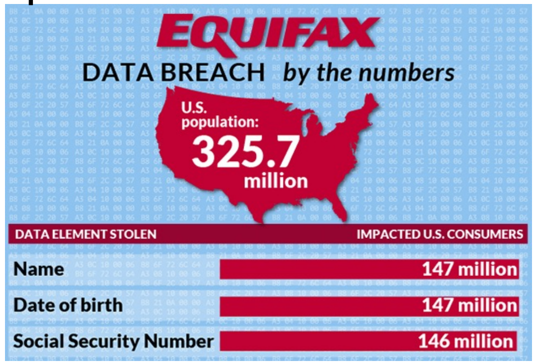
Software failure

Ariane 5 explosion
$370 million

1996

…

50% of American
personal record



2018

Recalls More than
150,000 vehicles

2021~2022

Software failure

AUTHOR

HERB KRASNER

CISQ Advisory Board Member and retired Professor of Software Engineering at the University of Texas at Austin.

He can be reached at hkrasner@utexas.edu.

THE COST OF POOR SOFTWARE QUALITY IN THE US: A 2020 REPORT


The Cost of Poor Software Quality in the US: A 2020 Report

CISQ

The Consortium for Information & Software Quality™ (CISQ™) released new research: The Cost of Poor Software Quality in the US: A 2020

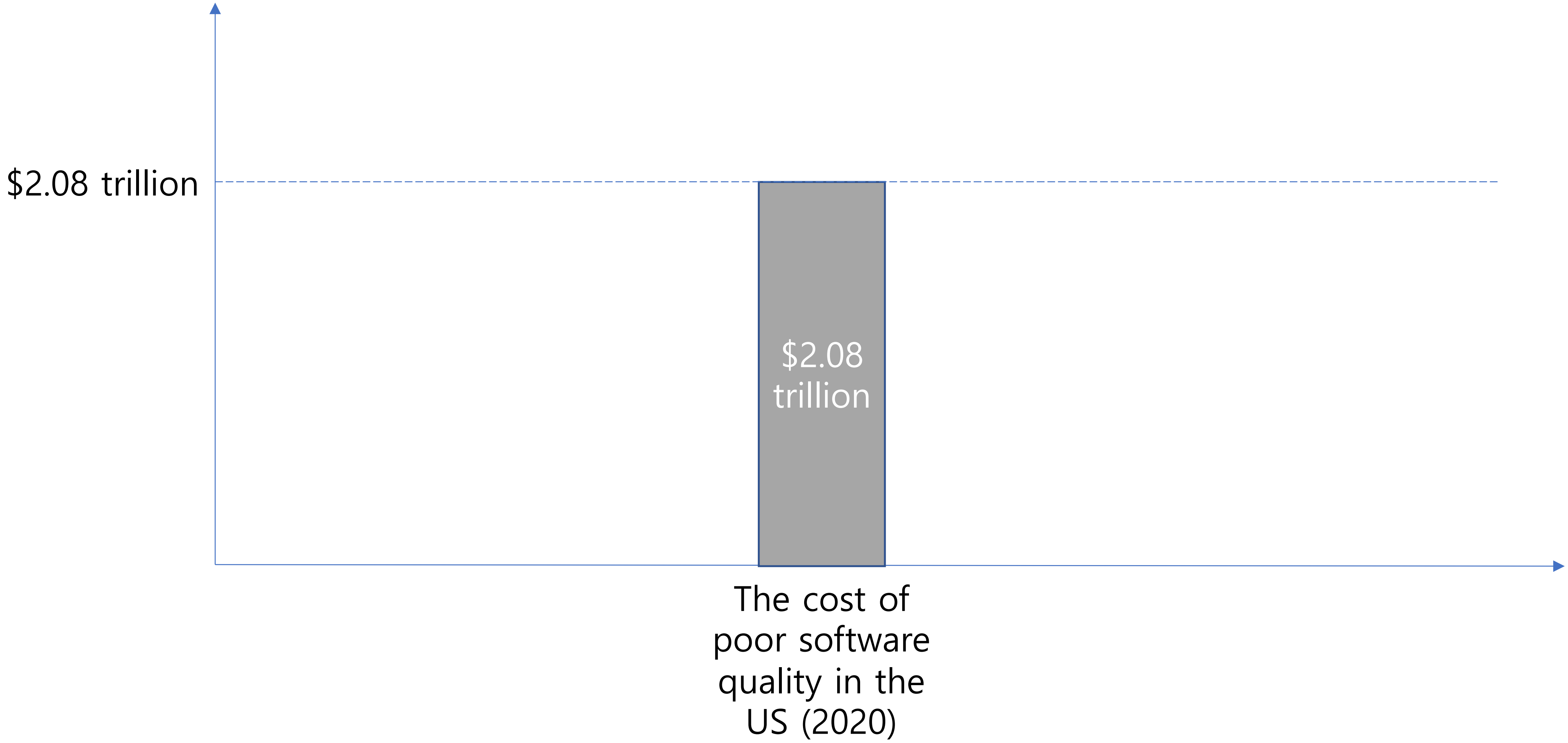
SYNOPSYS® Silicon to Software™

Software Engineering Institute Carnegie Mellon

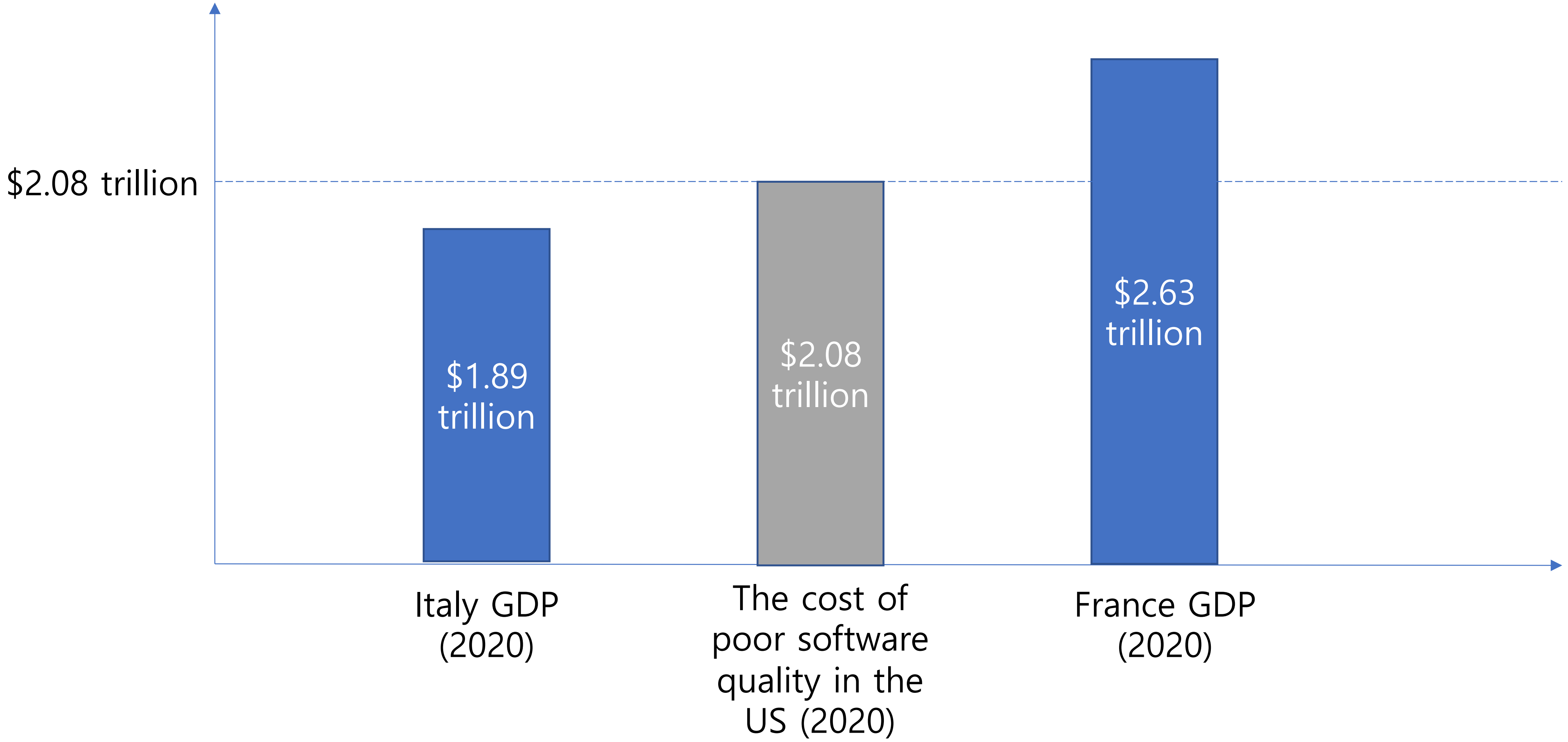
NORTHROP GRUMMAN

- Unsuccessful IT/software projects - $260 billion (up from $177.5 billion in 2018)
- Poor quality in legacy systems - $520 billion (down from $635 billion in 2018)
- Operational software failures - $1.56 trillion (up from $1.275 trillion in 2018)

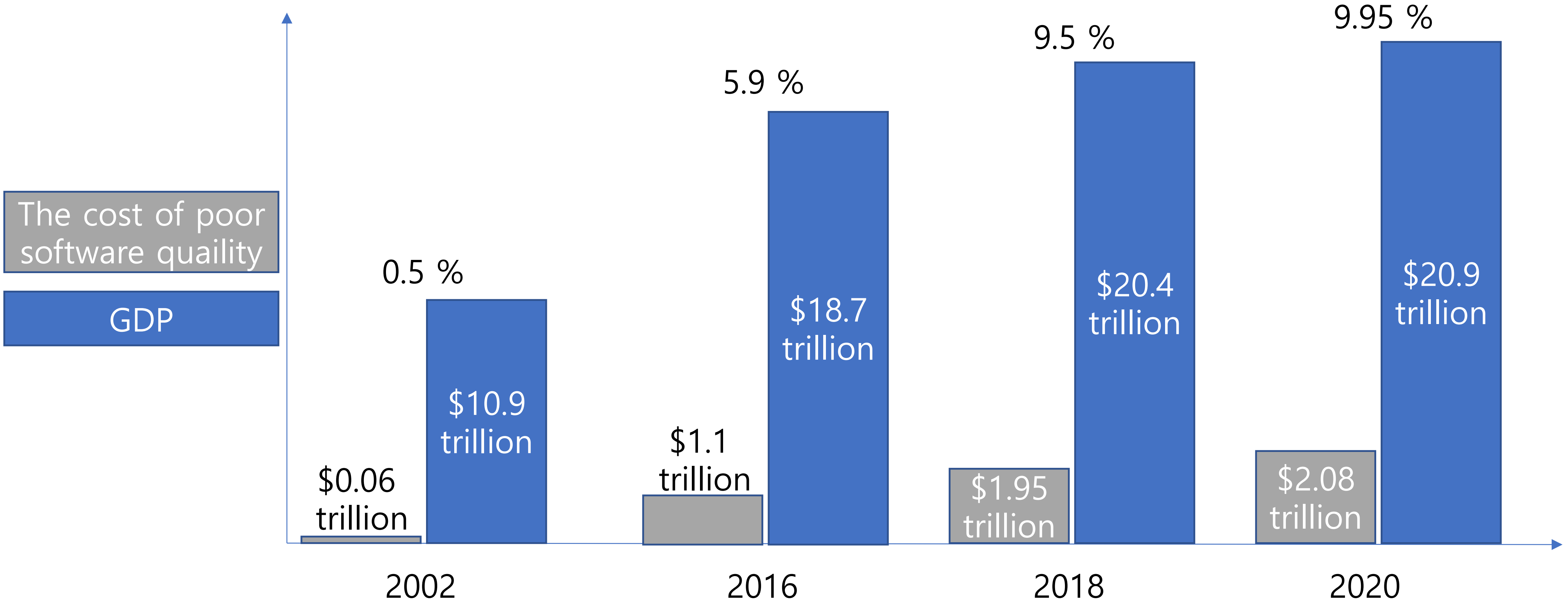
Software failure



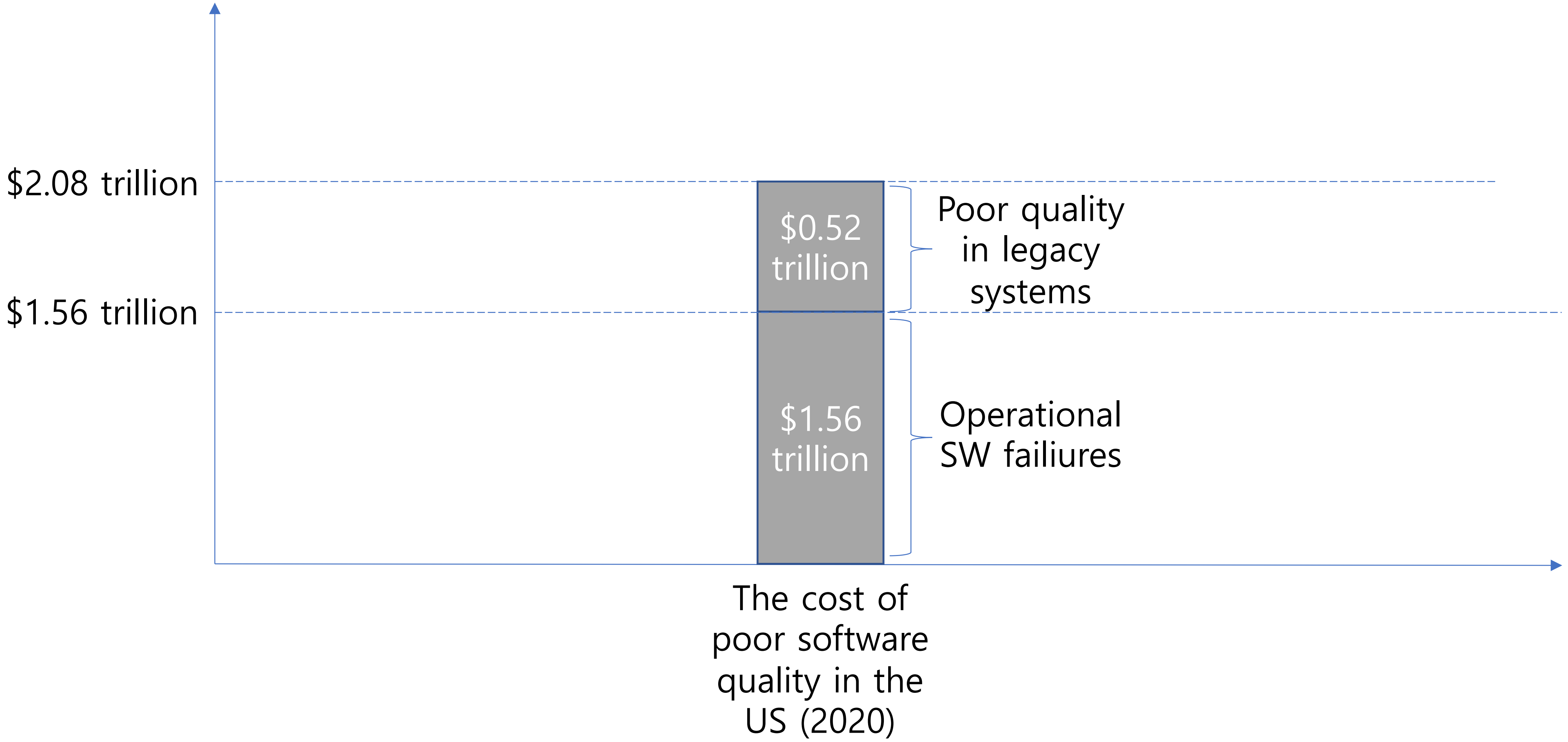
Software failure



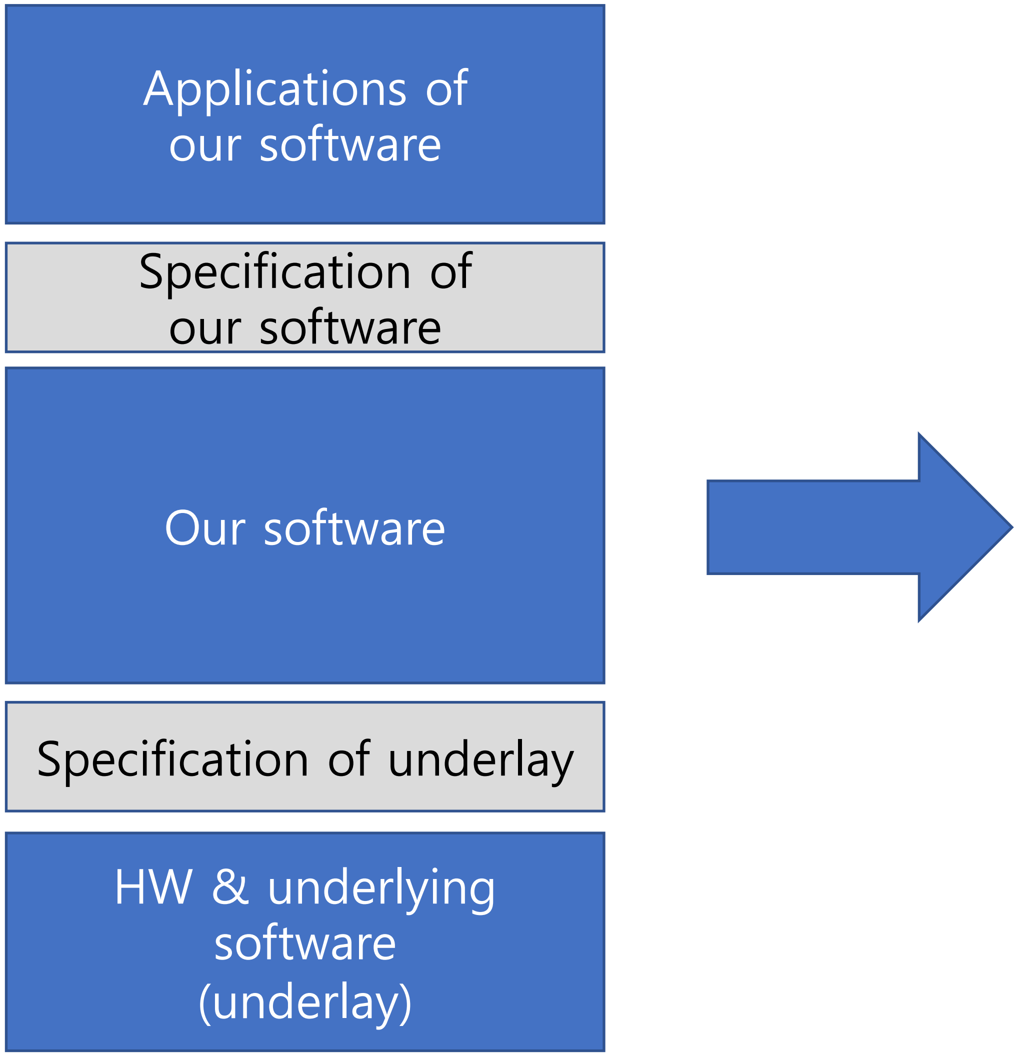
Software failure



Software failure

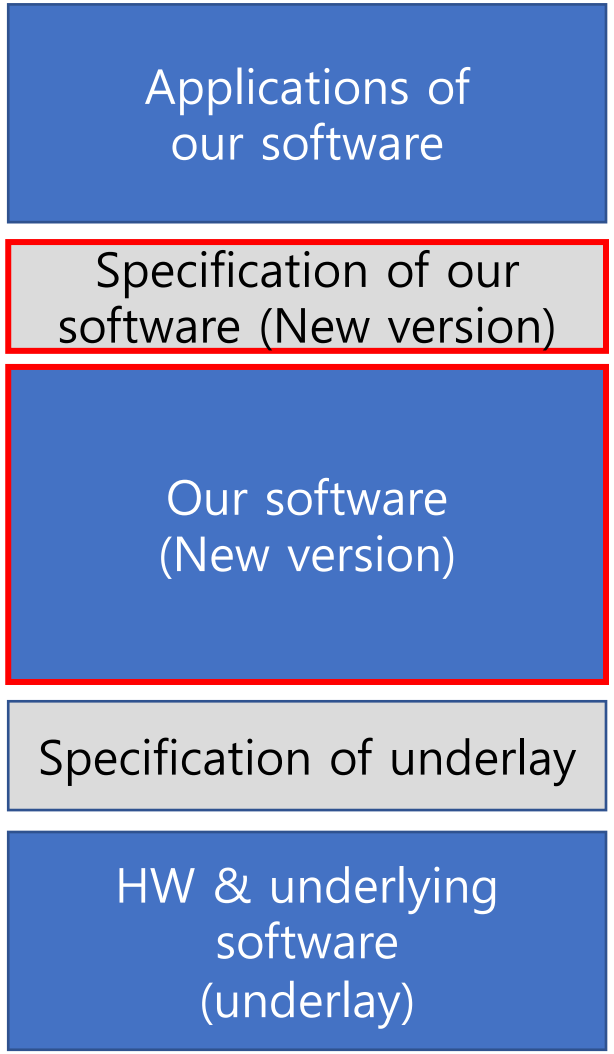


Reduce operational software failures



Our software faithfully implements the specification based on underlying HW and software specifications

Replace poor legacy software



Specification of our software (New version) $\supseteq$ Specification of our software (old version)

Our <u>software (new version)</u> <u>faithfully implements</u> <u>the specification (new version)</u> based on <u>underlying HW and software specifications</u>

Tools for software assurance

	Expressiveness level	Assurance level	Cost level
Code review	Very high	Very low	Medium
Testing	Medium	Low	Medium
Type checker (Java, Haskell, Rust)	Low	High	low
Static alaysis (Coverity, Infer)	Medium	Medium	low

Can those tools entirely tackle previous two challenges?

→ **NO!**

Tools for software assurance

practical

	Expressiveness level	Assurance level	Cost level
Code review	Very high	Very low	Medium
Testing	Medium	Low	Medium
Type checker (Java, Haskell, Rust)	Low	High	low
Static alaysis (Coverity, Infer)	Medium	Medium	low
Formal verificaiton (Z3, Adga, Coq)	Medium ~ High	High ~ Very high	Medium ~ Very high

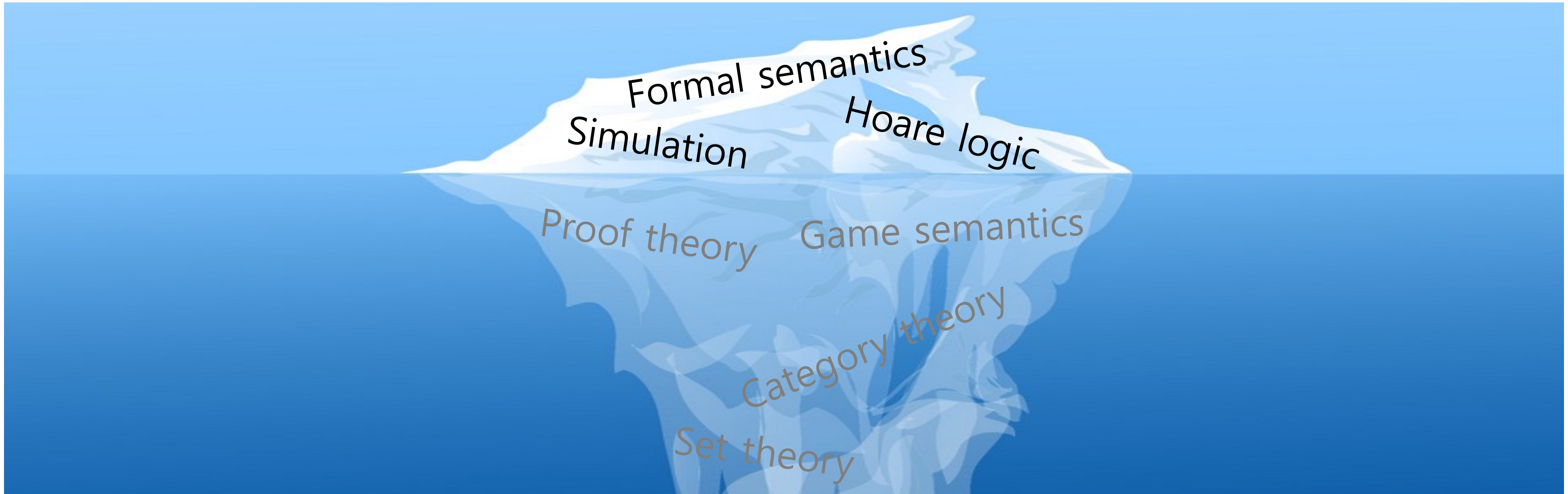
How can we effectively use high expressiveness?

How can we avoid very high cost?

Tools for software assurance

What do we need to know for formal verification?

- It is built on top of lots of unerlying theories
- But, verification engineers can only focus on the tiny subset that is actually required for the verfication target



Can it actually remove bugs?

An Empirical Study on the Correctness of Formally Verified Distributed Systems

Pedro Fonseca Kaiyuan Zhang Xi Wang Arvind Krishnamurthy

University of Washington

{pfonseca, kaiyuanz, xi, arvind}@cs.washington.edu

Abstract

Recent advances in formal verification techniques enabled the implementation of distributed systems with machine-checked proofs. While results are encouraging, the importance of distributed systems warrants a large scale evaluation of the results and verification practices.

This paper thoroughly analyzes three state-of-the-art, formally verified implementations of distributed systems: IronFleet, Verdi, and Chapar. Through code review and testing, we found a total of 16 bugs, many of which produce serious consequences, including crashing servers, returning incorrect results to clients, and invalidating verification guarantees. These bugs were caused by violations of a wide-range of assumptions on which the verified components relied. Our results revealed that these assumptions referred to a small fraction of the trusted computing base, mostly at the interface of verified and unverified components. Based on our observations, we have built a testing toolkit called PK, which focuses on testing these parts and is able to automate the detection of 13 (out of 16) bugs.

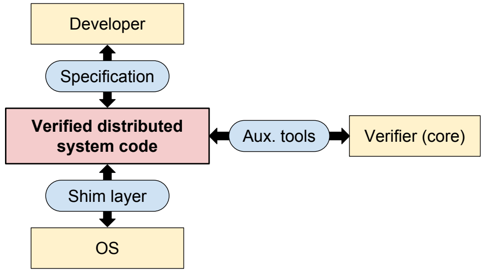
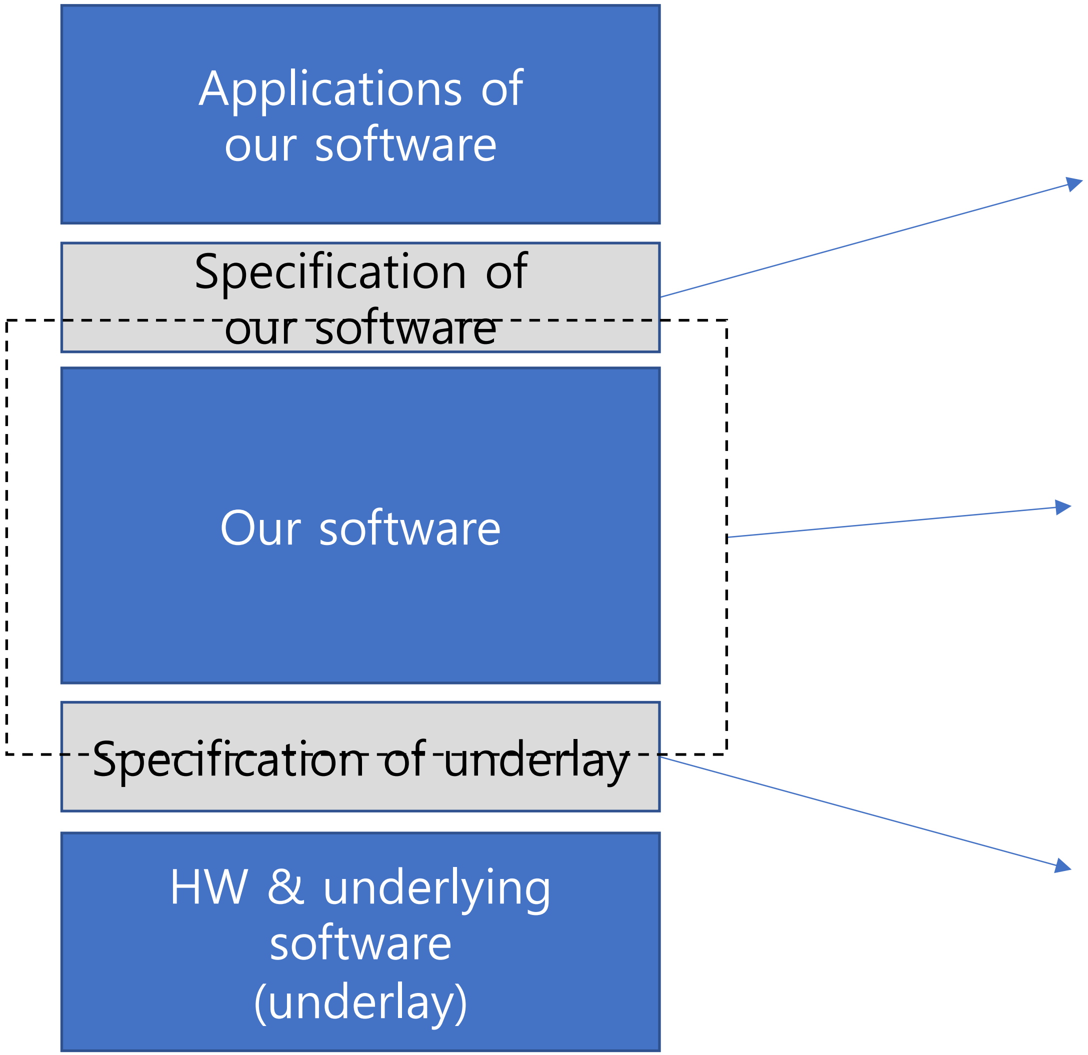


Figure 1: An overview of the workflow to verify a distributed system implementation.

Formal verification, in particular, offers an appealing approach because it provides a strong correctness guarantee of the absence of bugs under certain assumptions. Over the last few decades, the dramatic advances in formal verification techniques have allowed these techniques to scale to complex systems. They were successfully applied to build large single-node implementations, such as the seL4 OS kernel [28] and the CompCert compiler [35]. More recently,

Can it actually remove bugs?



Invariants provides correctness property, but it might have bugs that are not described in invariants

Formal verification can **guarantee** the **correctness** of **target software** module

Assumptions about unverified components, so it may have bugs

Formal verification intro with examples

Formal verification

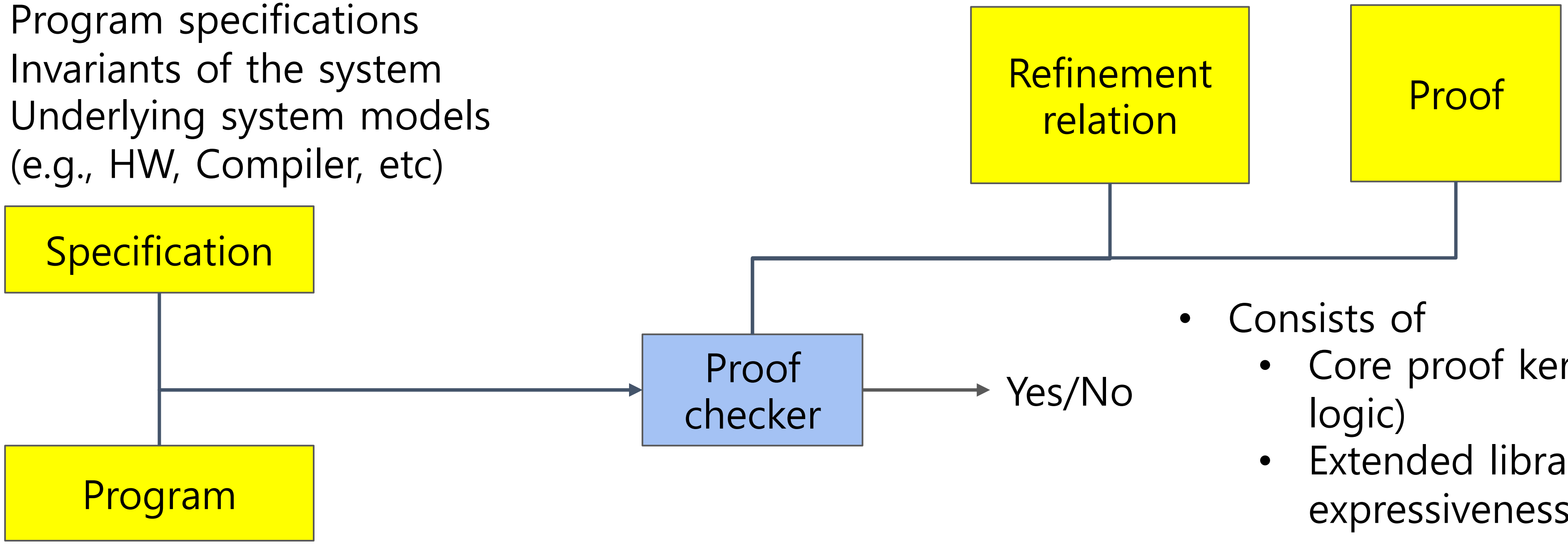
Definition

The act of **proving the correctness of software** with respect to a **certain formal specification** using **mathematics**

Key components

- Proofs for
  - Program meet specifications
  - Specficiations are consistent (i.e., all Invariants are well-defined)

- Mathematical notations for
  - Program specifications
  - Invariants of the system
  - Underlying system models (e.g., HW, Compiler, etc)



- Consists of
  - Core proof kernel (underlying logic)
  - Extended libraries for better expressiveness

- Subject of formal verificaiton

Verification tutorial: simple stateless function

"given two positive numbers, find sum of all numbers between two"

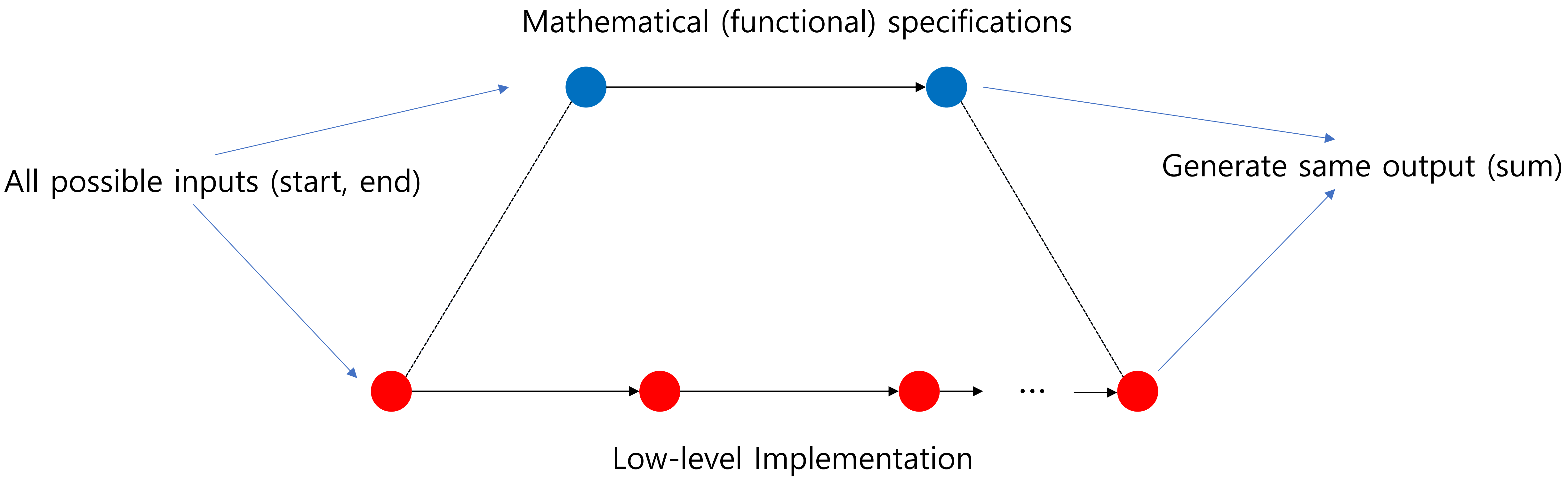
- Mathematical (functional) specs:

```
Definition range_sum (start end : nat)
  : nat :=
  …
  (end * (end - 1)
   - start * (start - 1)) / 2
end.
```

Program example:

```
int range_sum (int start, int end) {
  int sum = 0;
  …
  for (int i = start; i <= end; i++) {
    sum += i;
  }
  return sum;
}
```

Verification tutorial: simple stateless function



Verification tutorial: abstract state

Software usually facilitates hardware states, memory and registers.

Mathematical state could be much simplier than those physical states.

Mathmatical (functional) list:

```
Variable A : Type.

Inductive list : Type :=
  | nil : list
  | cons : A -> list -> list.
```

Program example:

1) With array

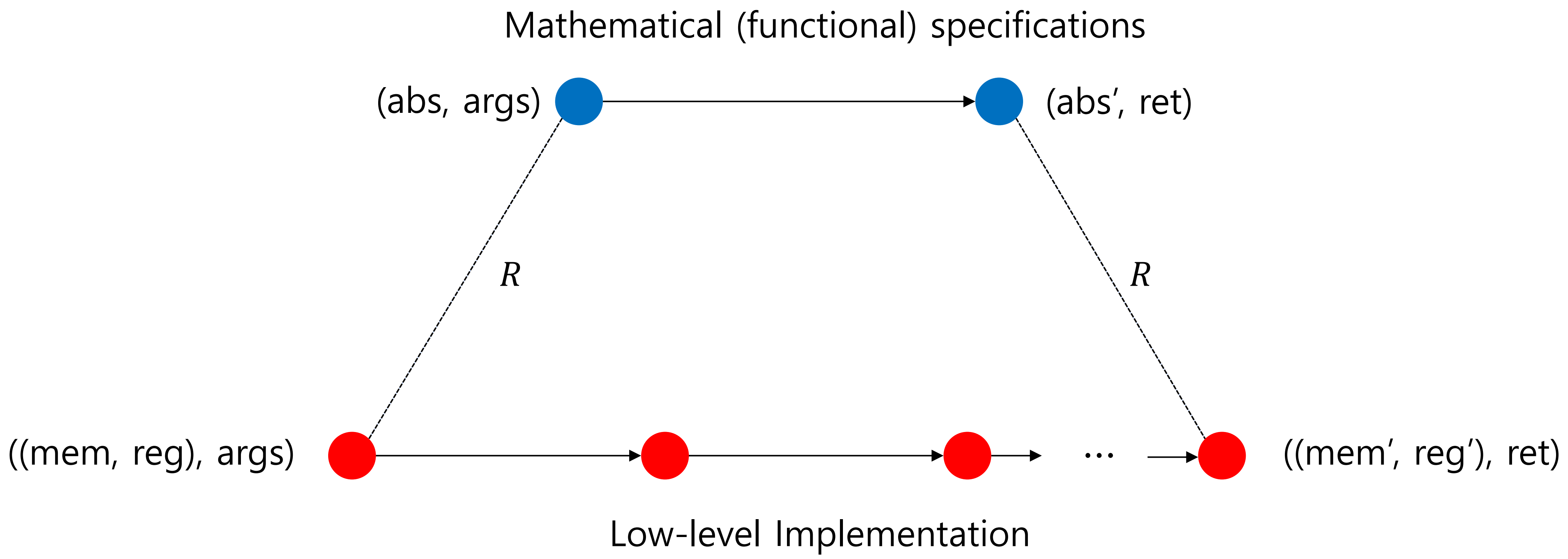
```
int array_list[kMaxLength];
```

1) With linked list

```
struct Node {
    int data;
    Node* next;
    Node* prev;
};
```

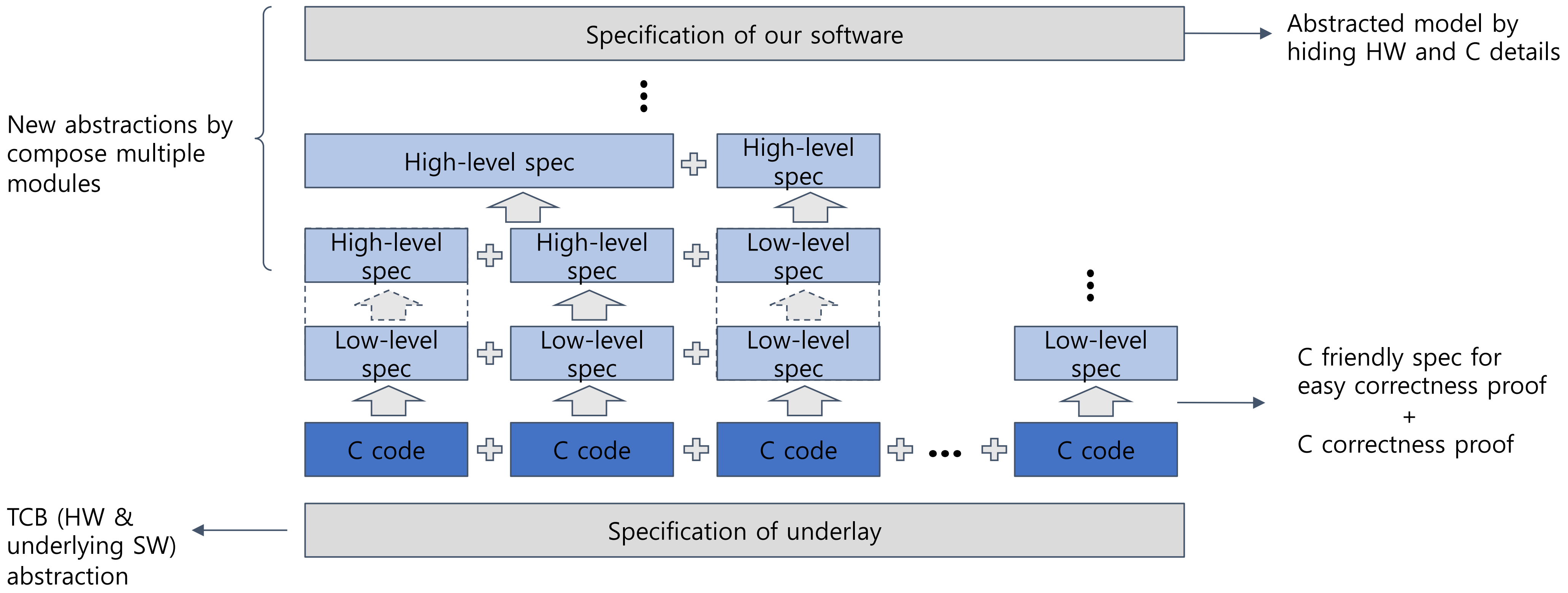
Refinement relation (R): how mathematical list is related to the low-level structure.

Verification tutorial: abstract state

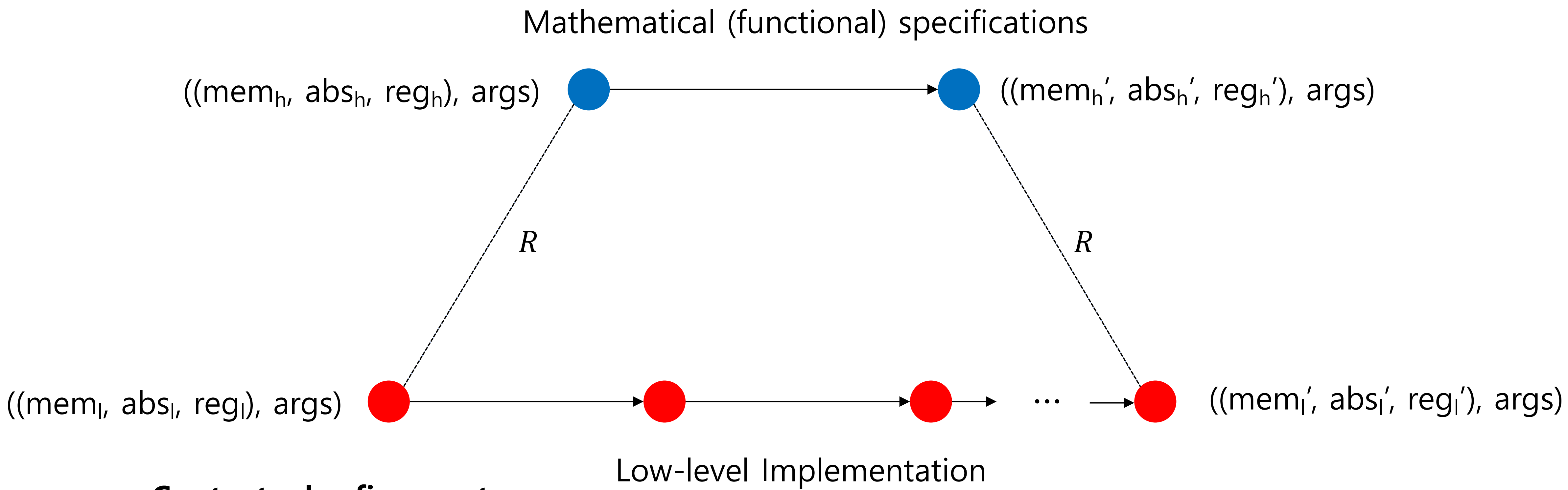


Verification tutorial: modularity

Decompose the entire software into multiple sub components, verifying them, and combine their proofs together.

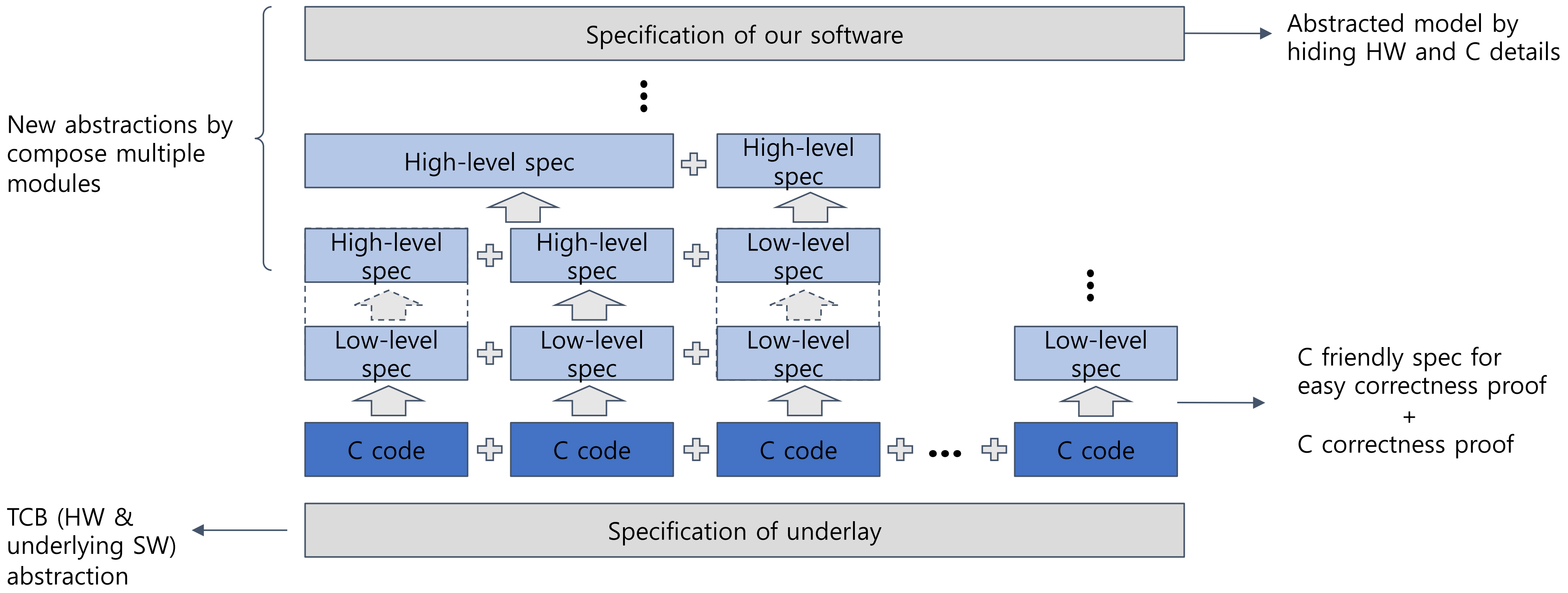


Verification tutorial: modularity



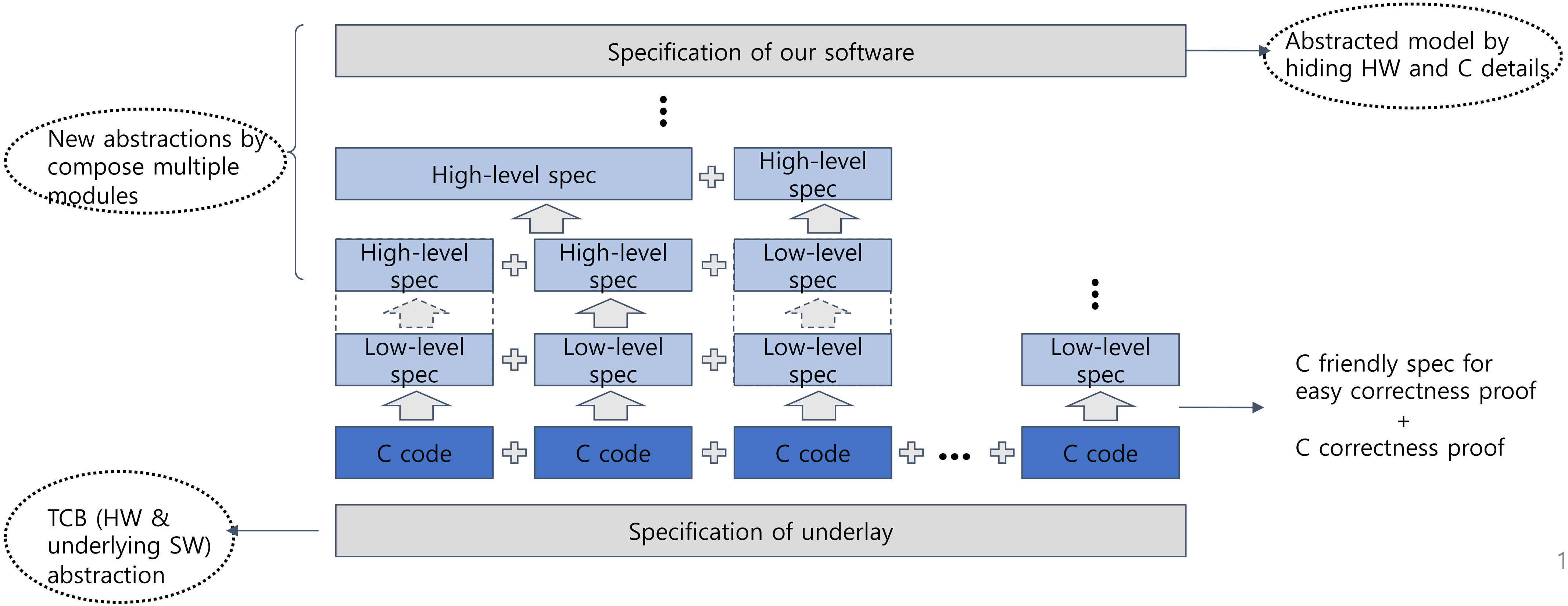
- **Contextual refinement**
  - Compositional approach to compositional verification of concurrent objects.
  - Combined with several program logics, it can show consistency between the object implementation and its abstract specification.

Verification tutorial: modularity



Verification tutorial: modularity

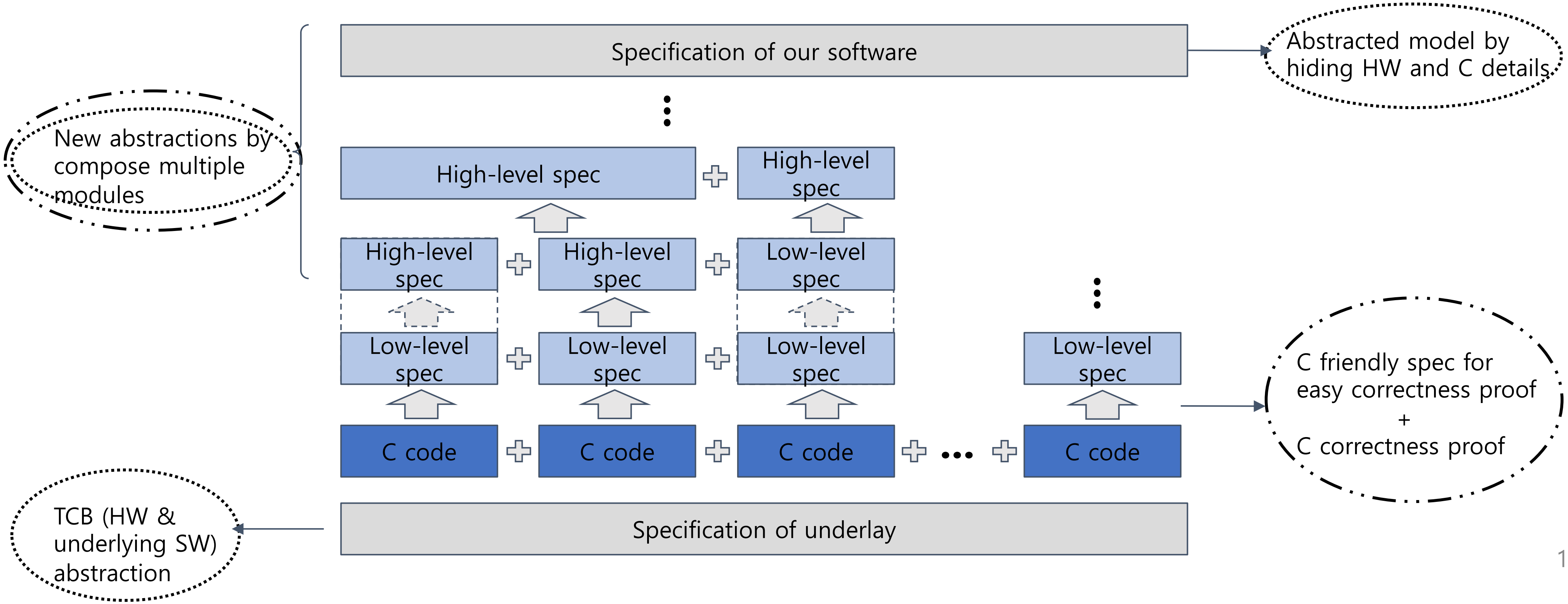
How can we effectively use high expressiveness?



Verification tutorial: modularity

How can we effectively use high expressiveness?

How can we reduce the very high cost?



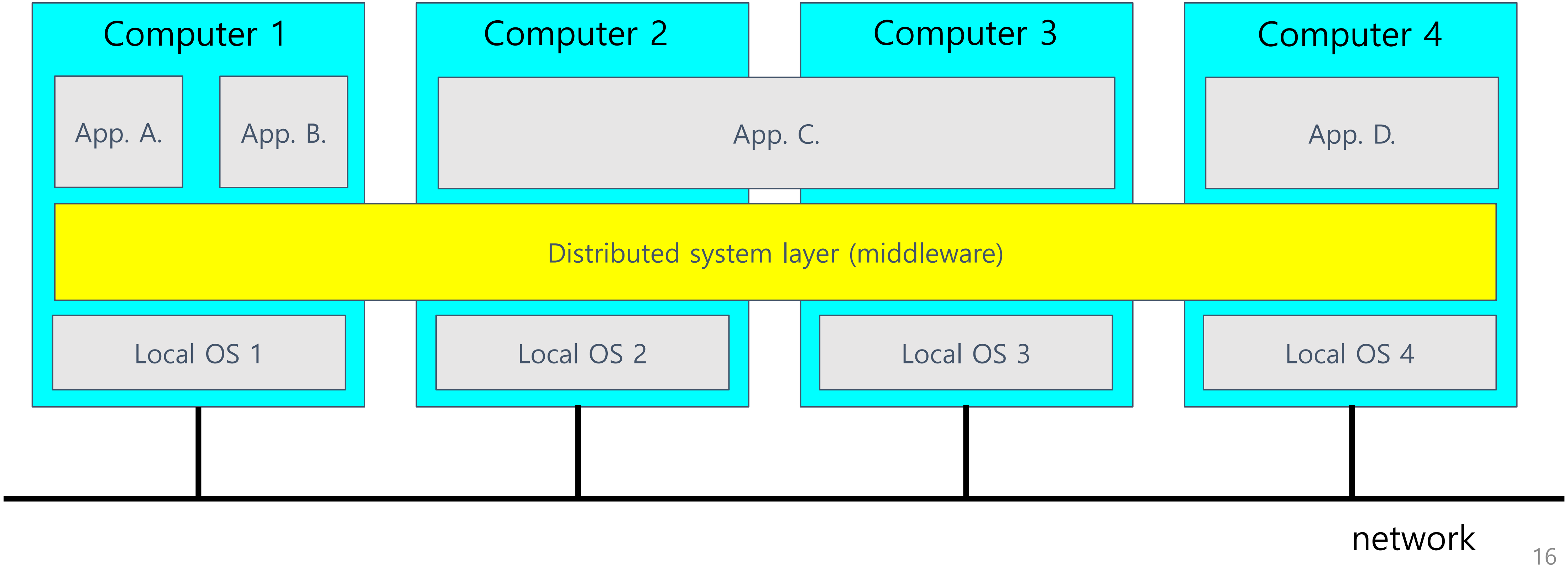
Formal verification projects

My formal verification researches

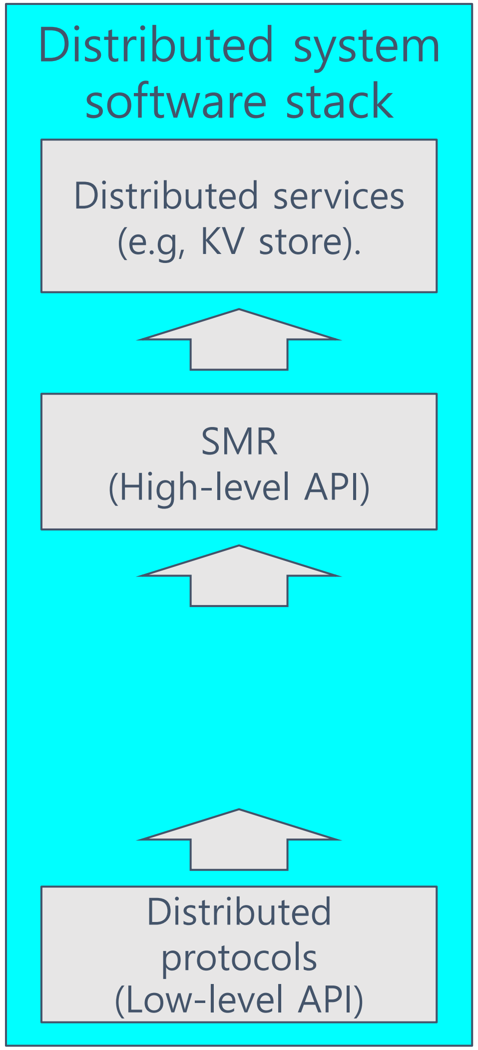
- CertiKOS – Small OS and hypervisor
  - **CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels**. OSDI 2016
  - **Safety and Liveness of MCS Lock - Layer by Layer**. APLAS 2017
  - **Certified concurrent abstraction layers**. PLDI 2018
  - **Building certified concurrent OS kernels**. Comm. of ACM 62(10) 2019
- ADO (Atomic Distributed Object) – Distributed system
  - **WormSpace: A Modular Foundation for Simple, Verifiable Distributed Systems**. SoCC 2019
  - **Much ADO about failures: a fault-aware model for compositional verification of strongly consistent distributed systems**. Proc. ACM Program. Lang. 5(OOPSLA)
  - **Adore: Atomic Distributed Objects with Certified Reconfiguration**, PLDI 2022
- pKVM formal verificaiton – Practical hypervisor

Distributed system verification

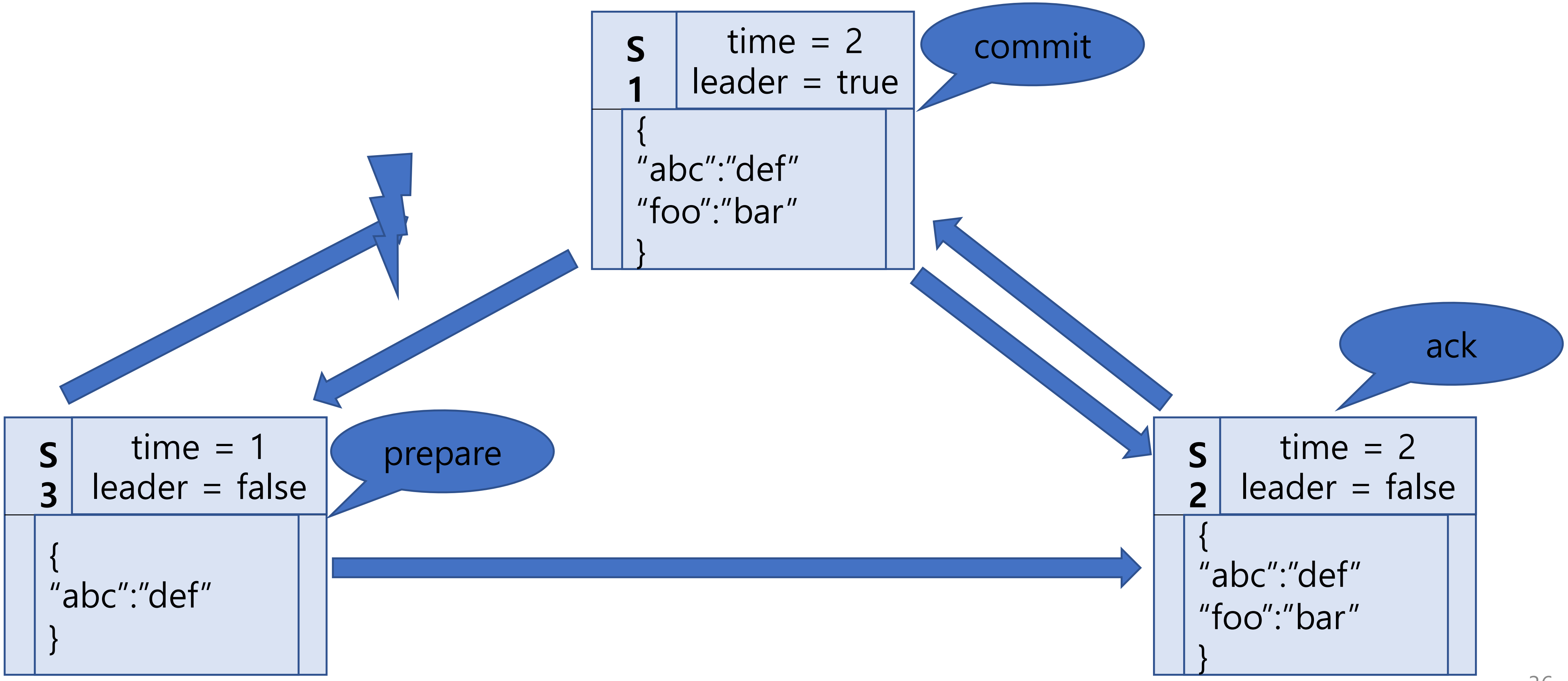
Distributed system



Distributed system software stack

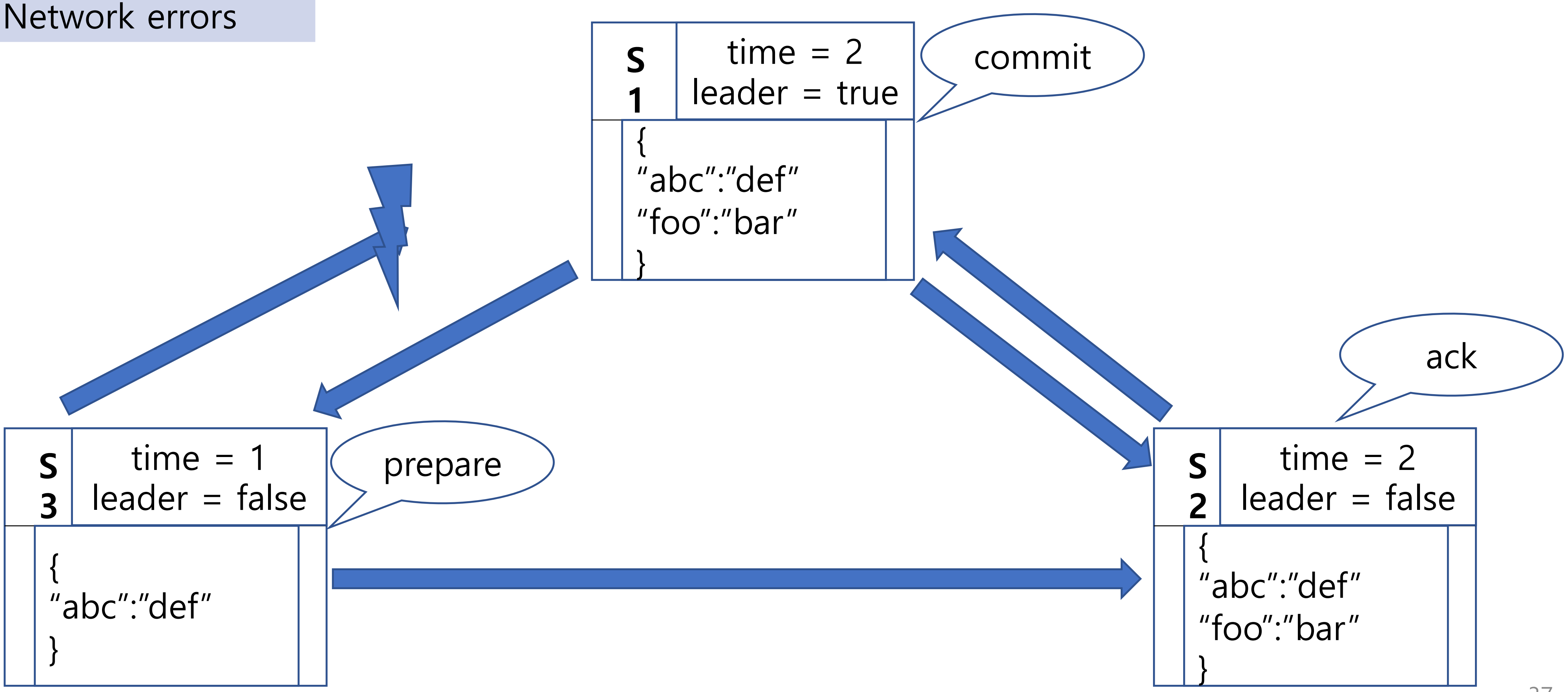


Network-based models too complex



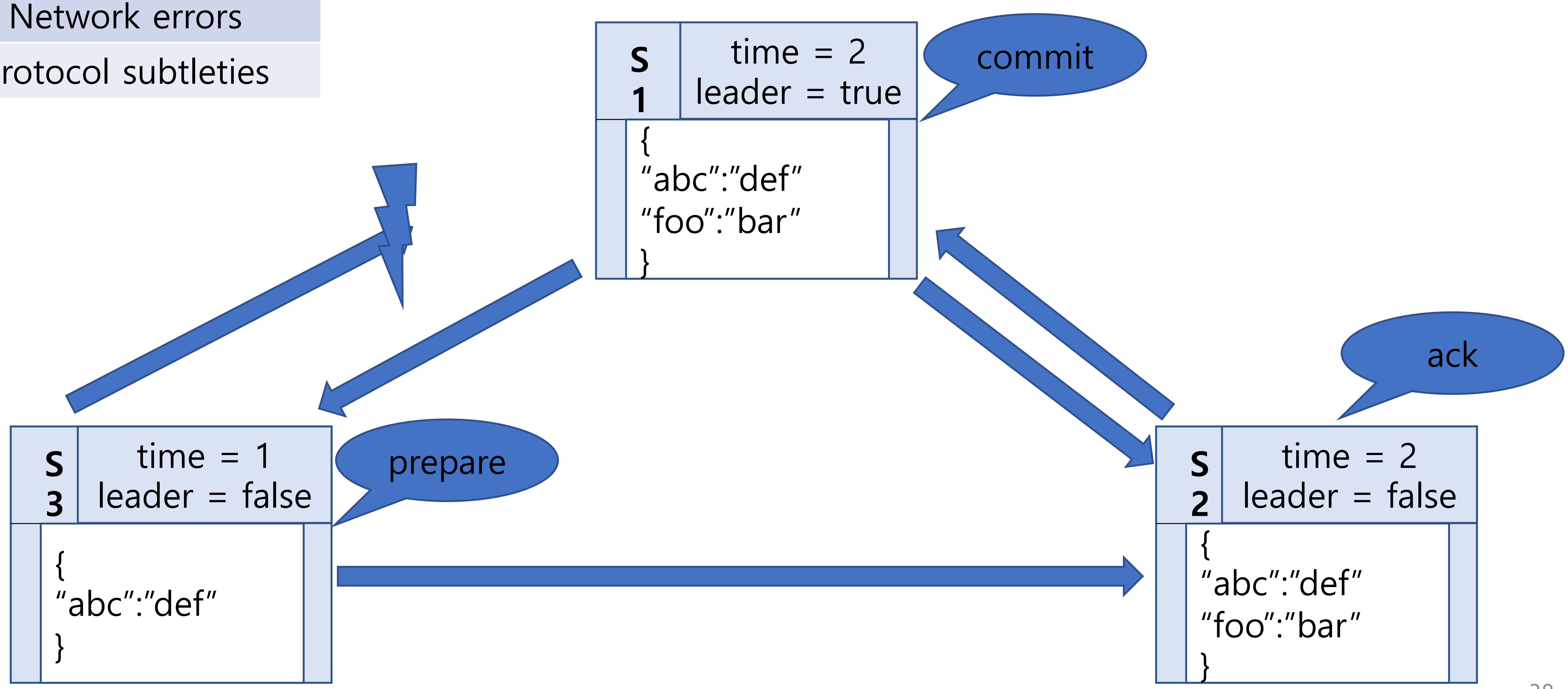
Network-based models too complex

Challenges
Network errors



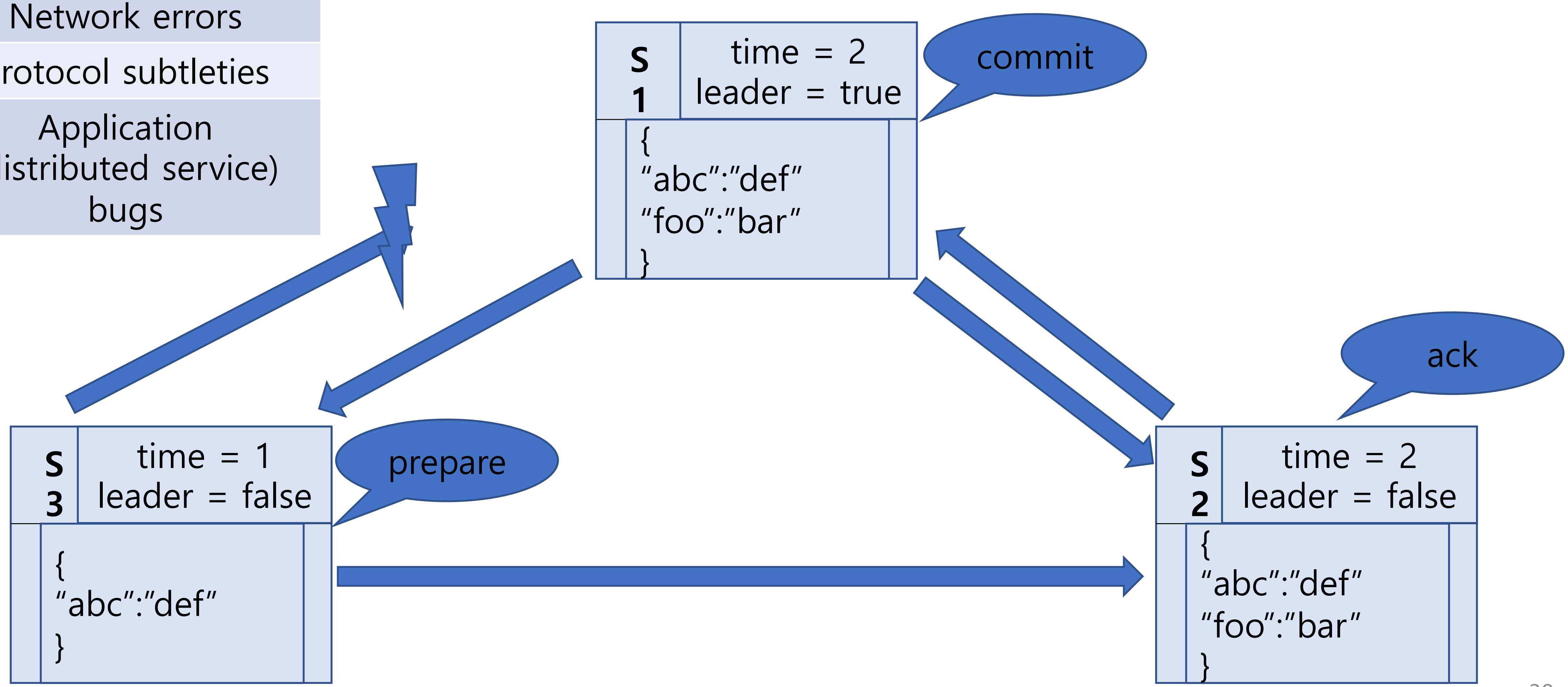
Network-based models too complex

Challenges
Network errors
Protocol subtleties

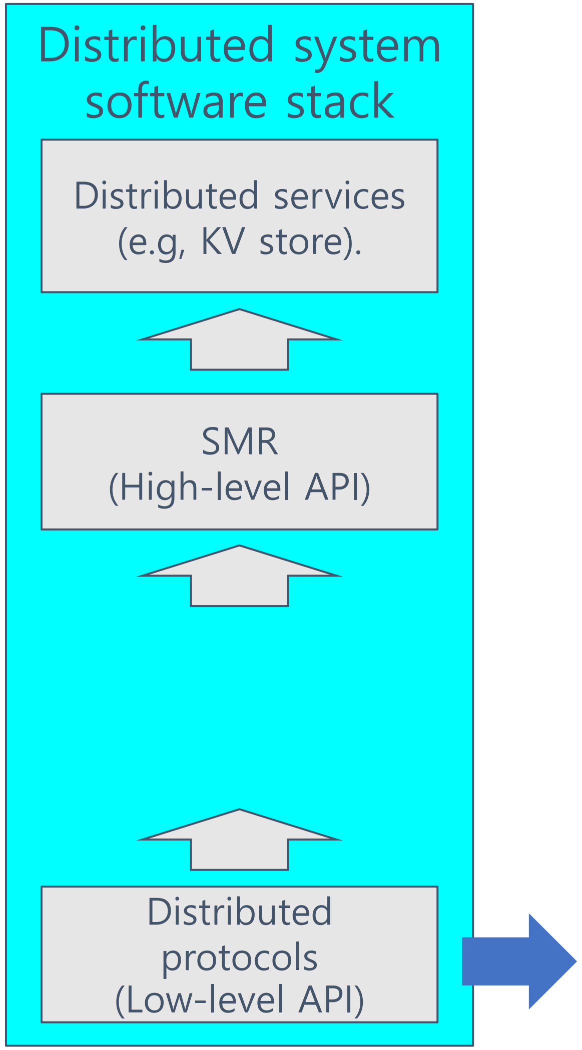


Network-based models too complex

Challenges
Network errors
Protocol subtleties
Application (distributed service) bugs

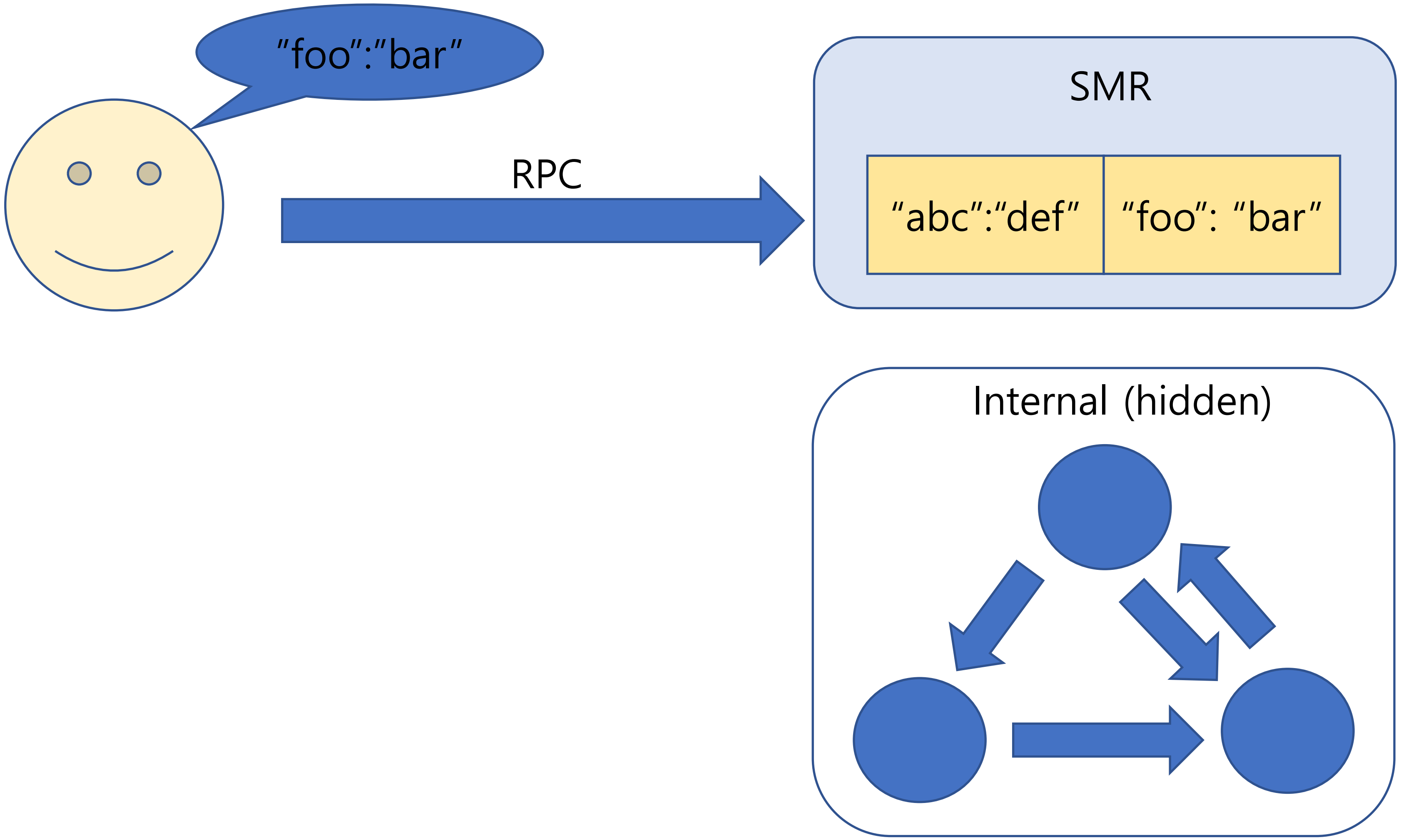


Network-based models too complex

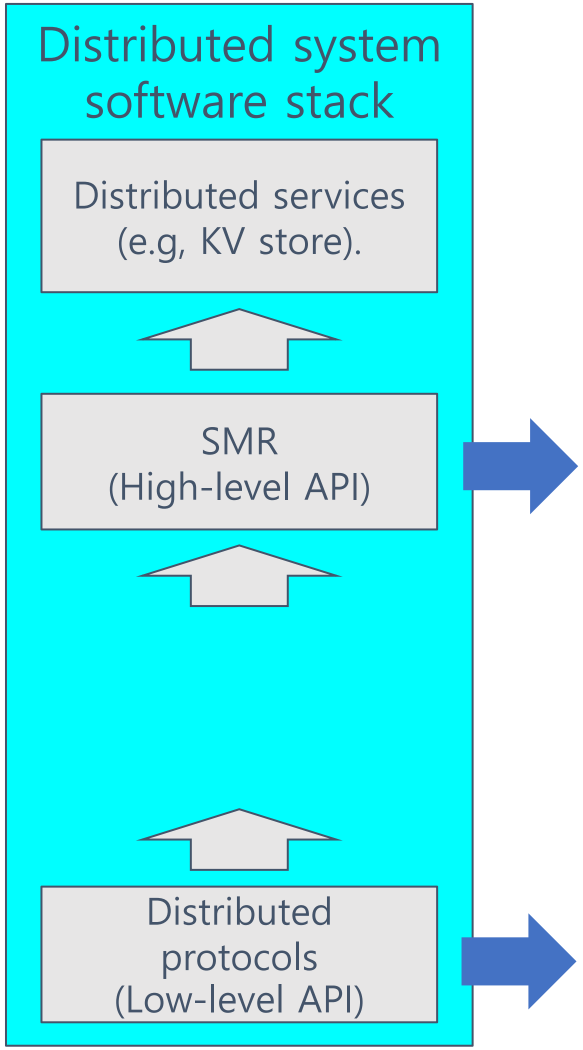


- Non-determinism
  - Complex interleaving
  - Network & node errors
- Several protocols and implementations (paxos, raft, chain-replication, etc)
- Lots of verificaiton works done

State machine replication too abstract

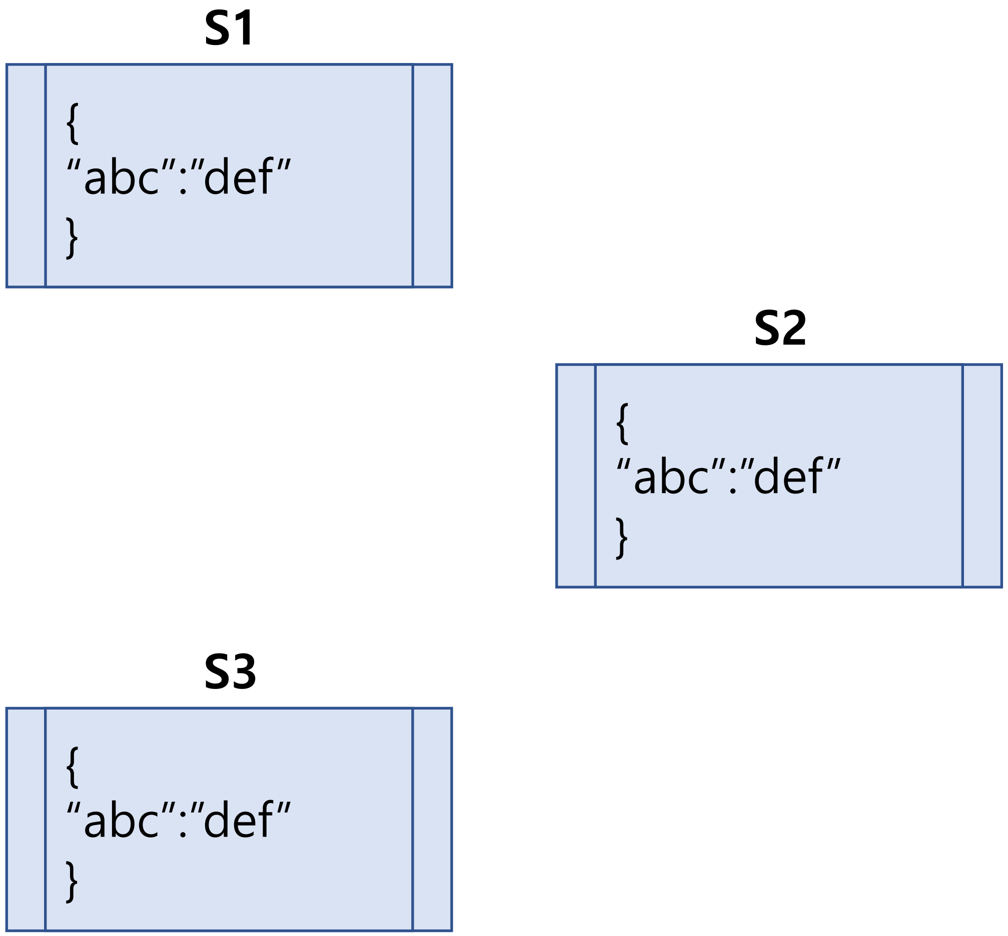


State machine replication too abstract

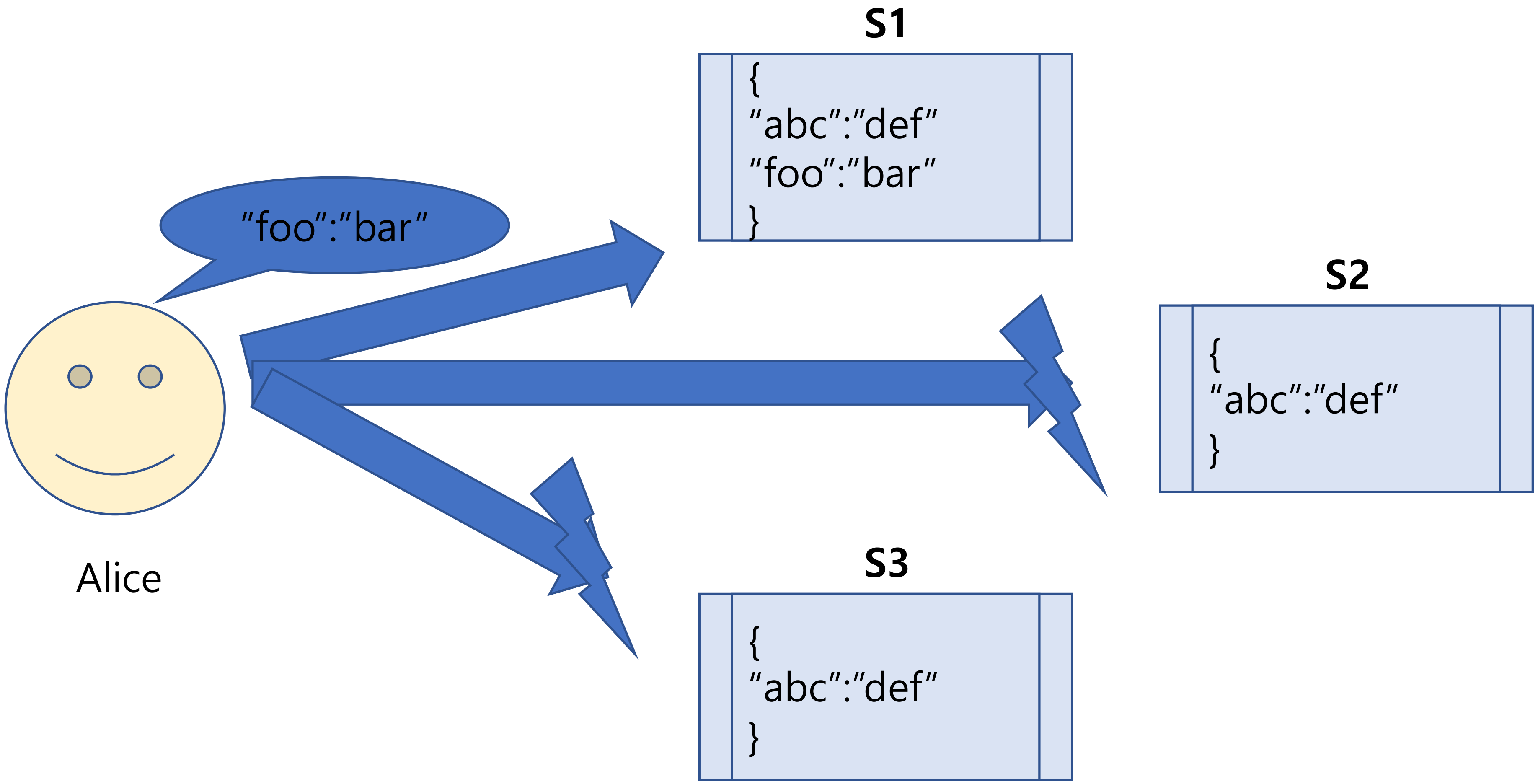


- Deterministic
- Unified abstraction
- Non-determinism
  - Complex interleaving
  - Network & node errors
- Several protocols and implementations (paxos, raft, chain-replication, etc)
- Lots of verificaiton works done

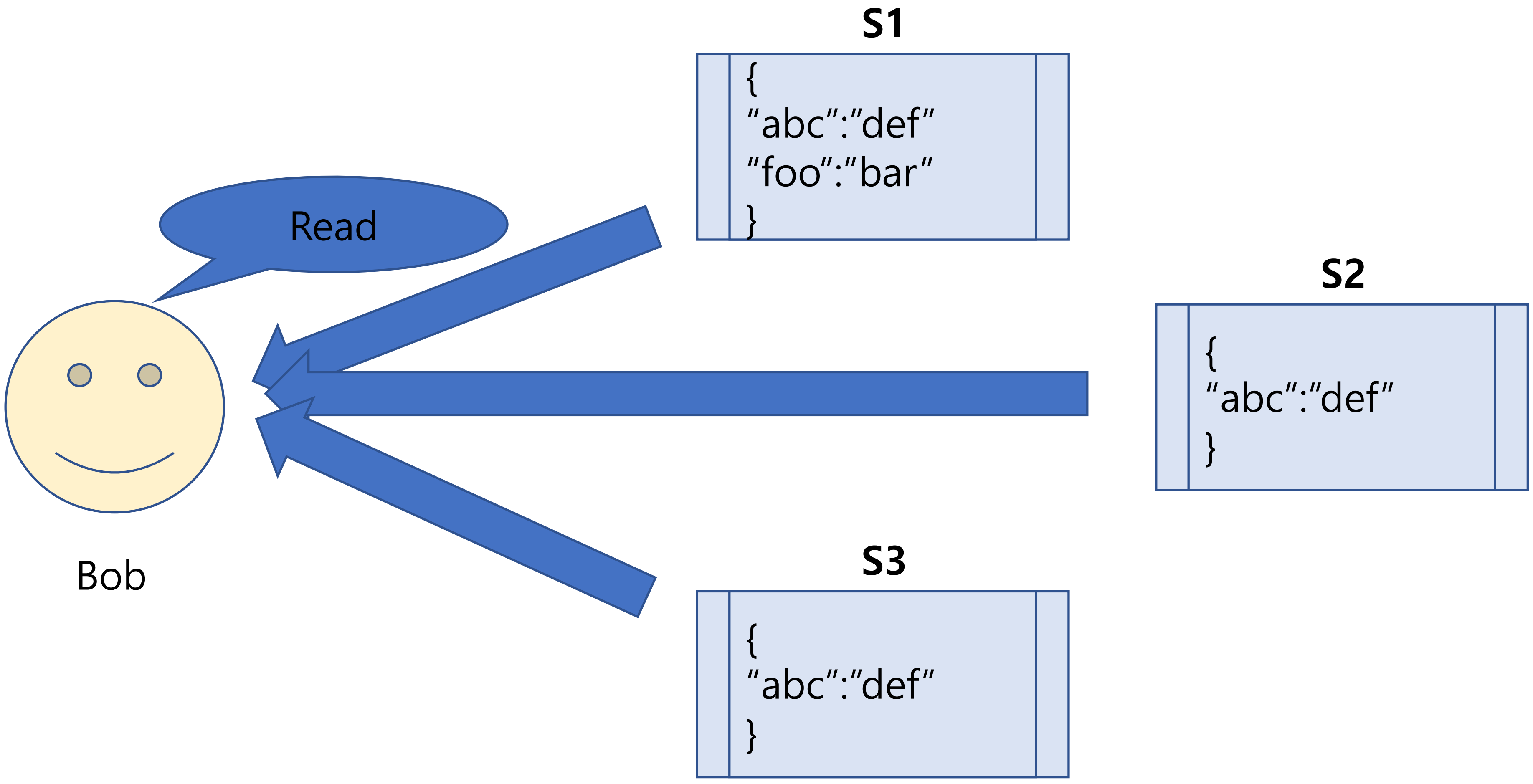
Partial failure



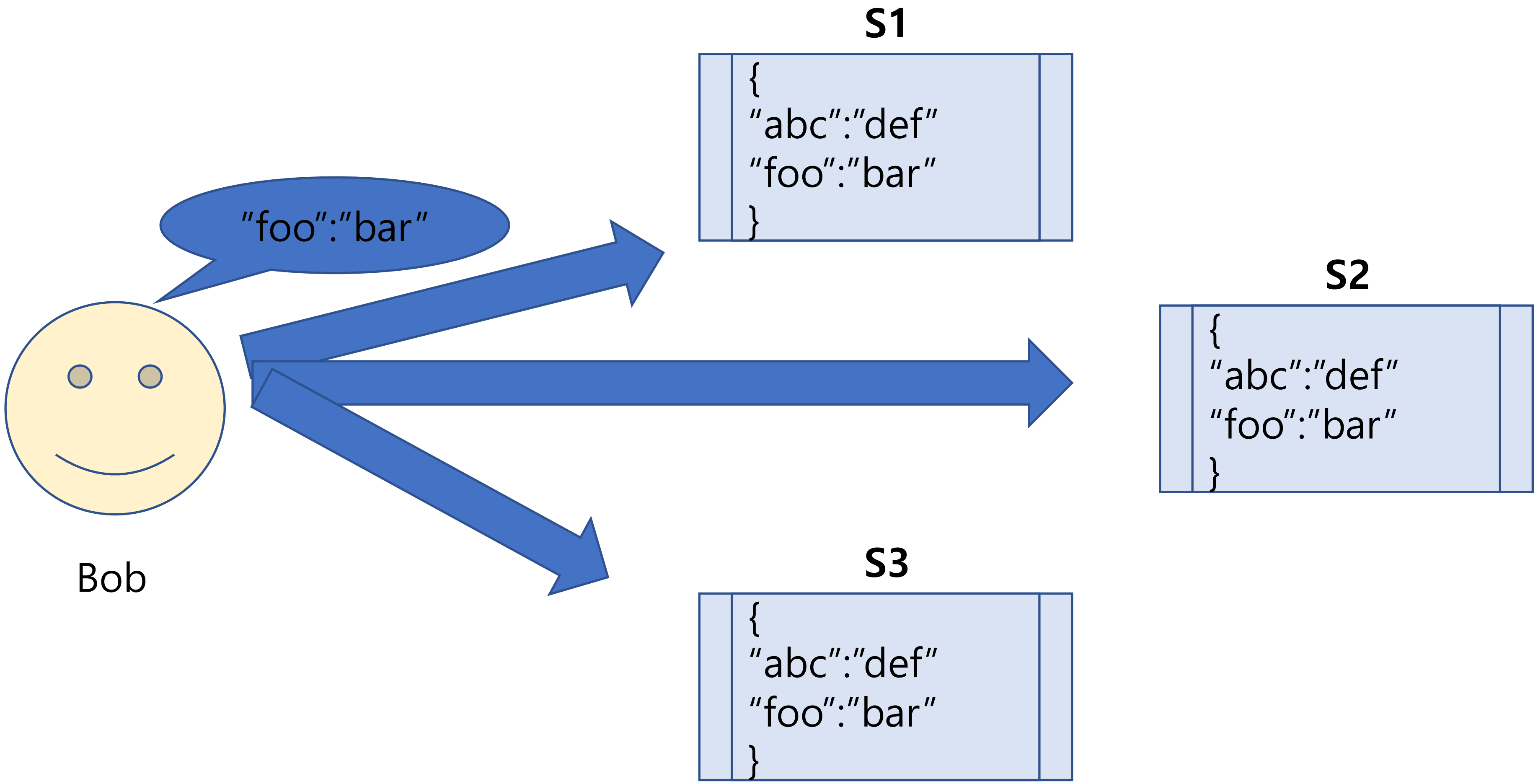
Partial failure



Partial failure



Partial failure

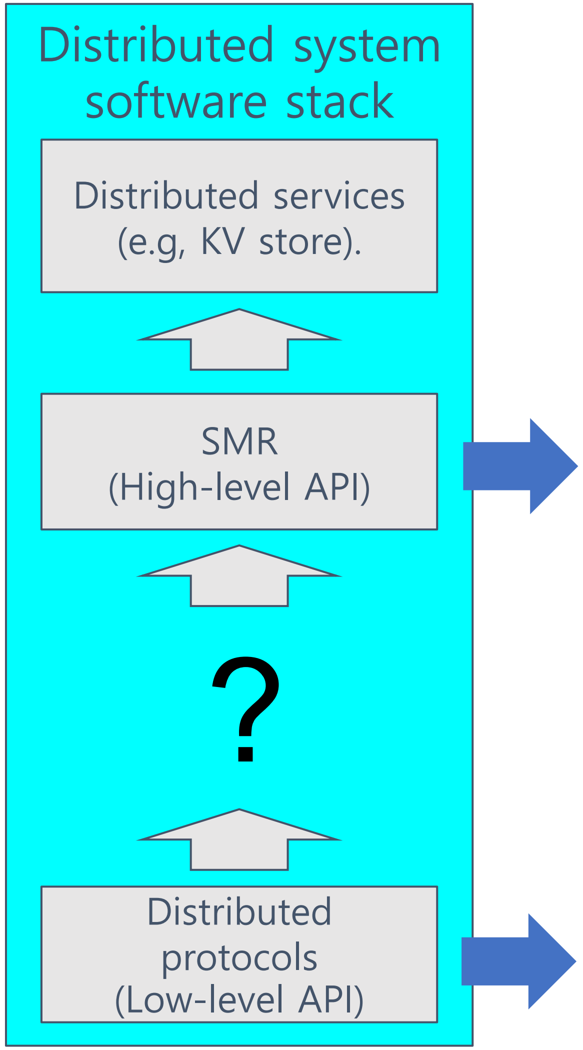


Partial failure is important

Partial failure is a central reality of distributed computing. [. . .] Being robust in the face of partial failure requires some expression at the interface level.
(Jim Waldo. A Note on Distributed Computing. 1994)

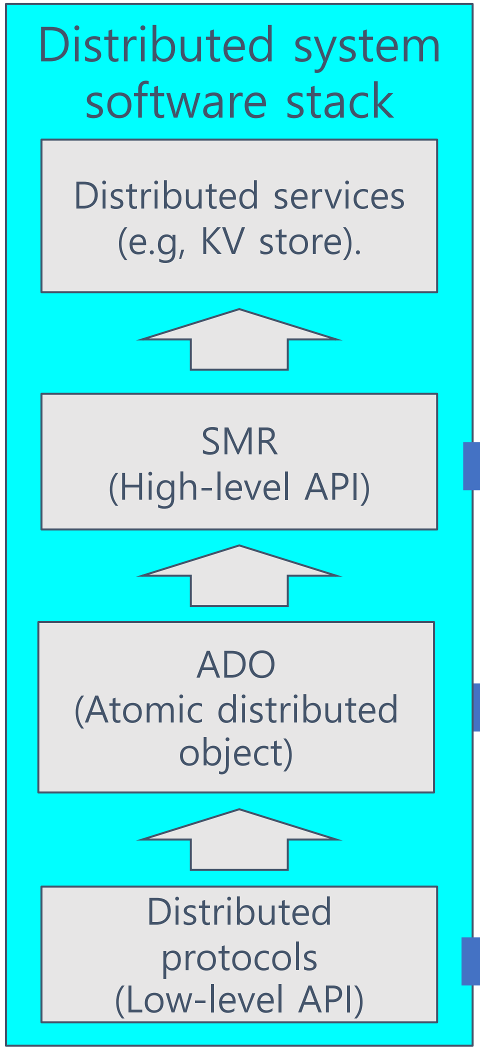
- Unavoidable feature unique to distributed systems
- Influence with all aspects of distributed protocols (e.g., leader election and reconfiguration)
- Can be used for performance optimizations
  - TAPIR (SOSP '15): Transactions with out-of-order commits
  - Speculator (SOSP '05): Speculative distributed file system

Partial failure is important



- Deterministic
- Unified abstraction
- Non-determinism
  - Complex interleaving
  - Network & node errors
- Several protocols and implementations (paxos, raft, chain-replication, etc)
- Lots of verificaiton works done

ADO (Atomic distributed object)



- Deterministic
- Unified abstraction
- Non-determinism
  - Complex interleaving
  - Network & node errors
- Several protocols and implementations (paxos, raft, chain-replication, etc)
- Lots of verificaiton works done

- Simple, but non-deterministic abstraction
- Covers all protocols
- Make connection between two APIs possible

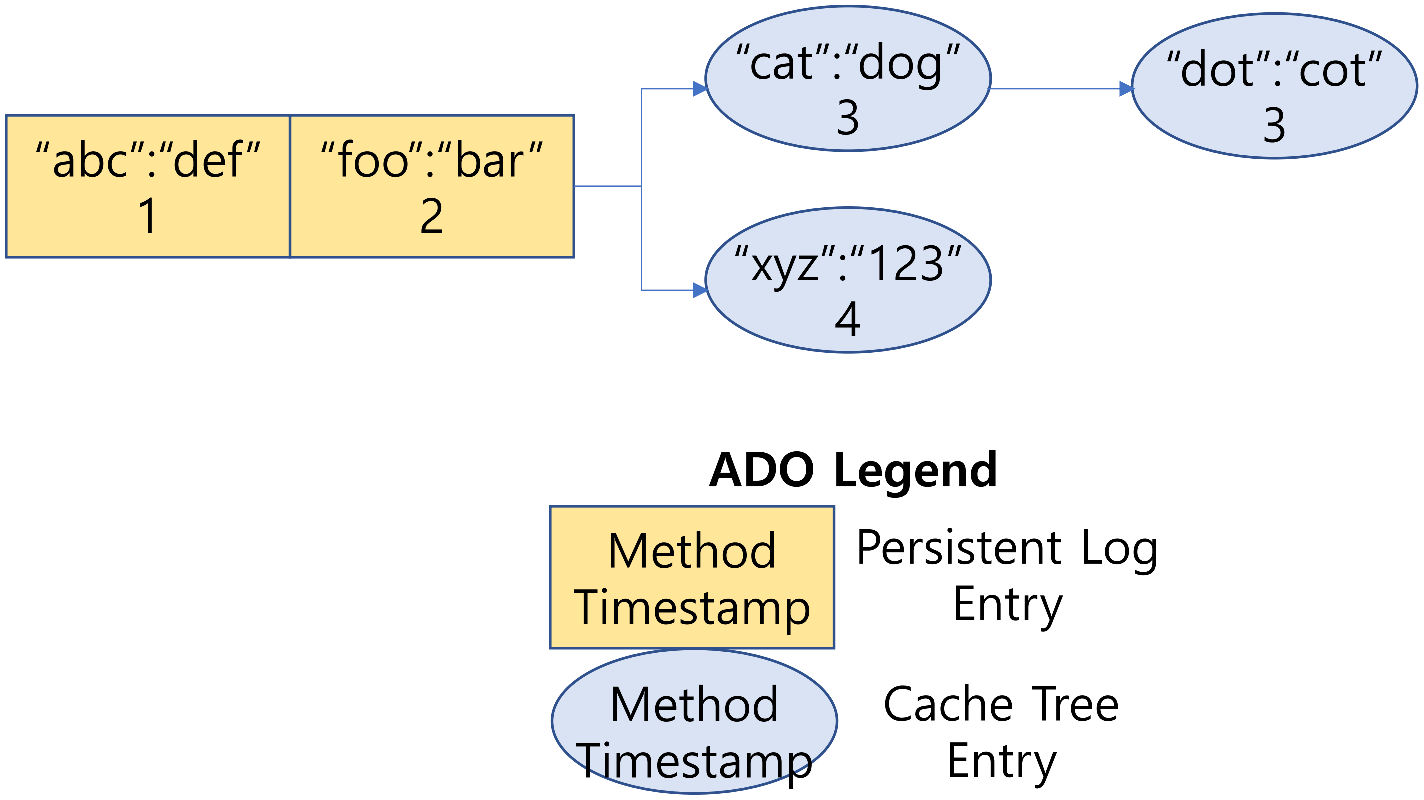
ADO state

"abc":"def"	"foo":"bar"
1	2

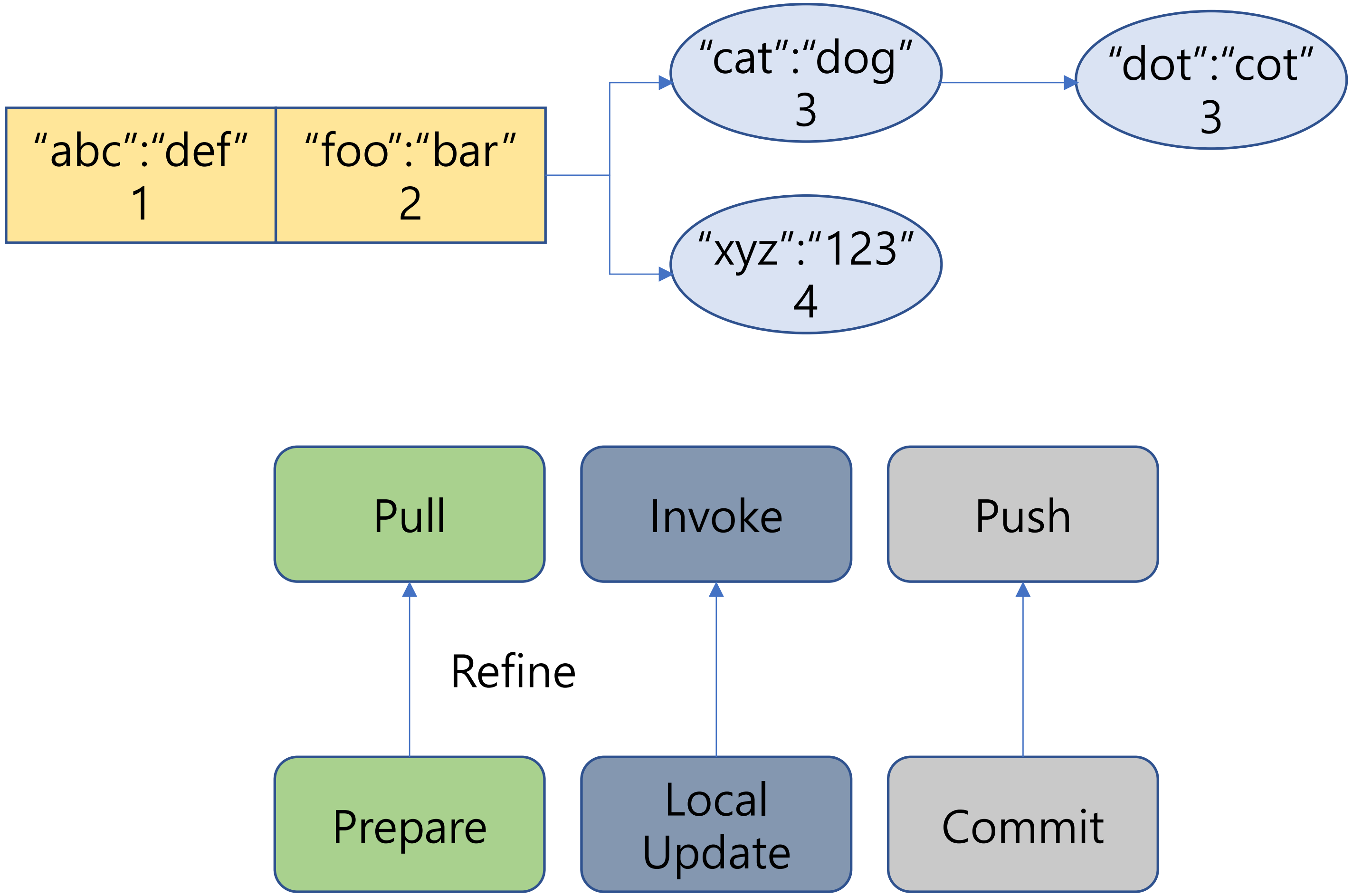
ADO Legend

Method Timestamp	Persistent Log Entry

ADO state

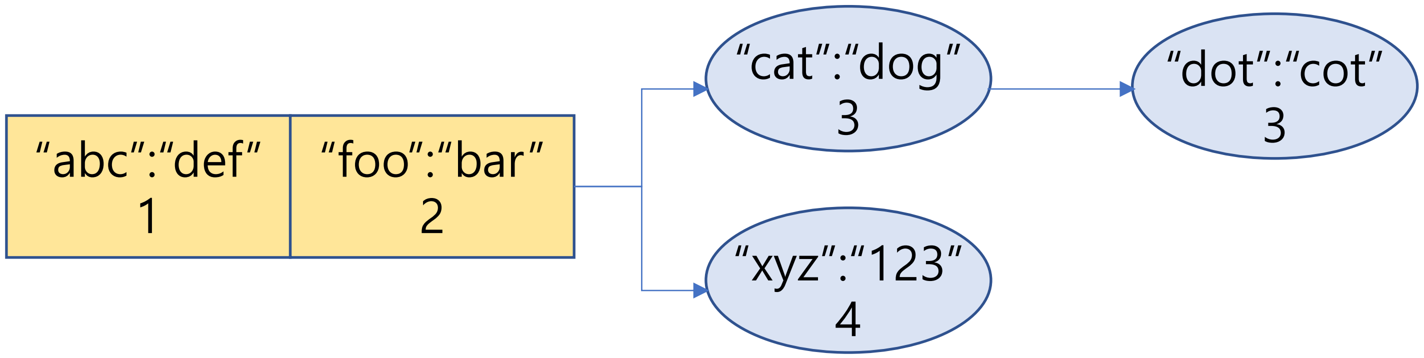

"abc":"def"
1
"foo":"bar"
2
"cat":"dog"
3
"dot":"cot"
3
"xyz":"123"
4
ADO Legend
Method
Timestamp
Persistent Log
Entry
Method
Timestamp
Cache Tree
Entry

ADO operations

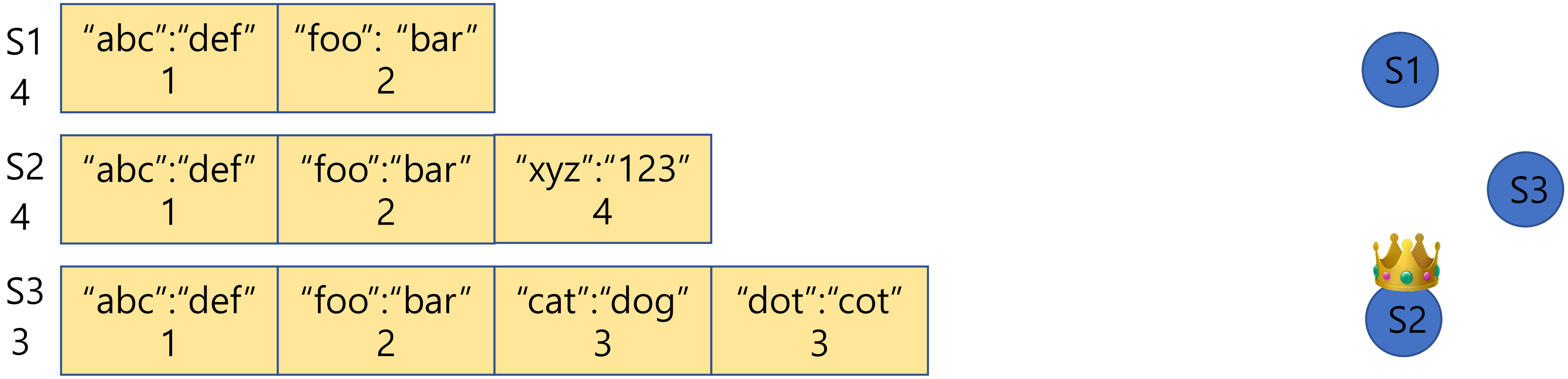


ADO operations

ADO

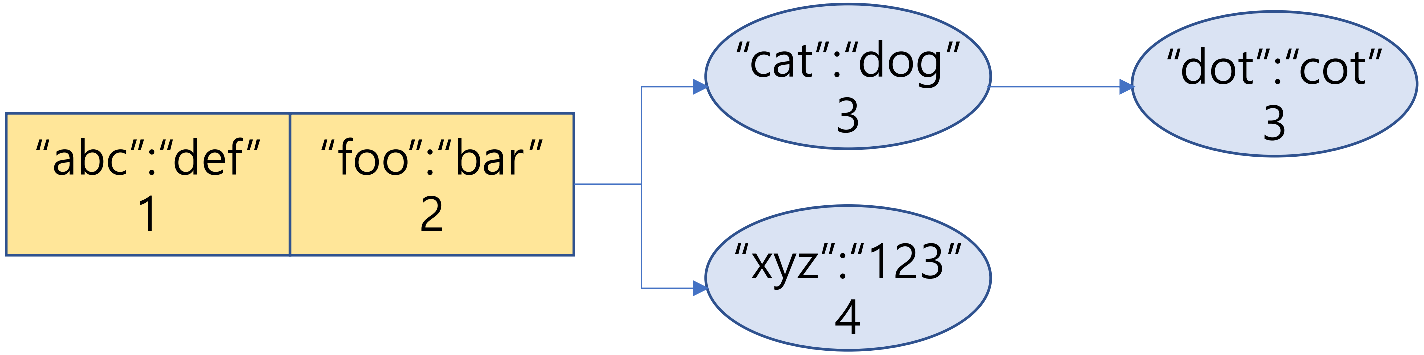


Multi-Paxos

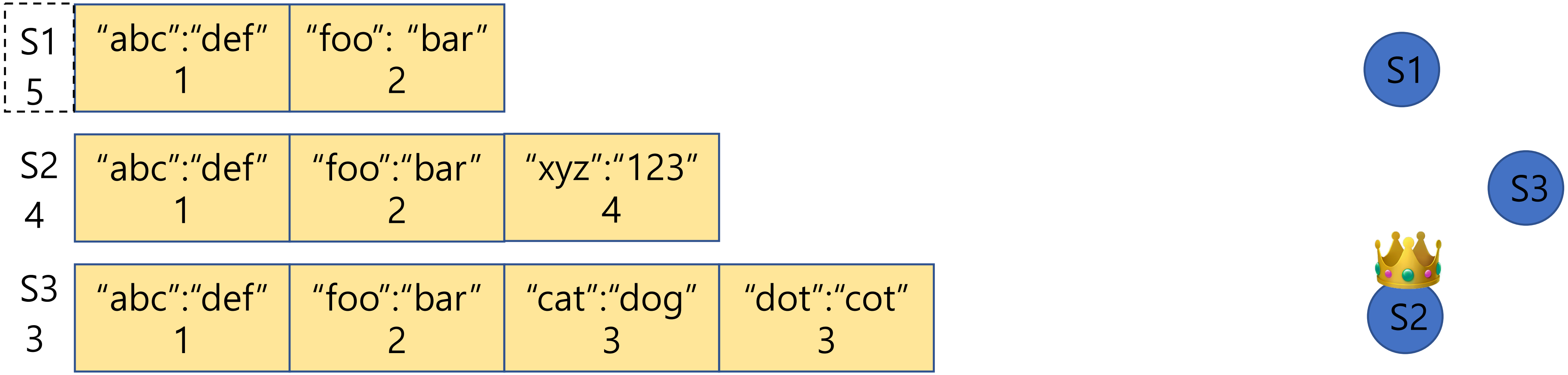


ADO operations

ADO

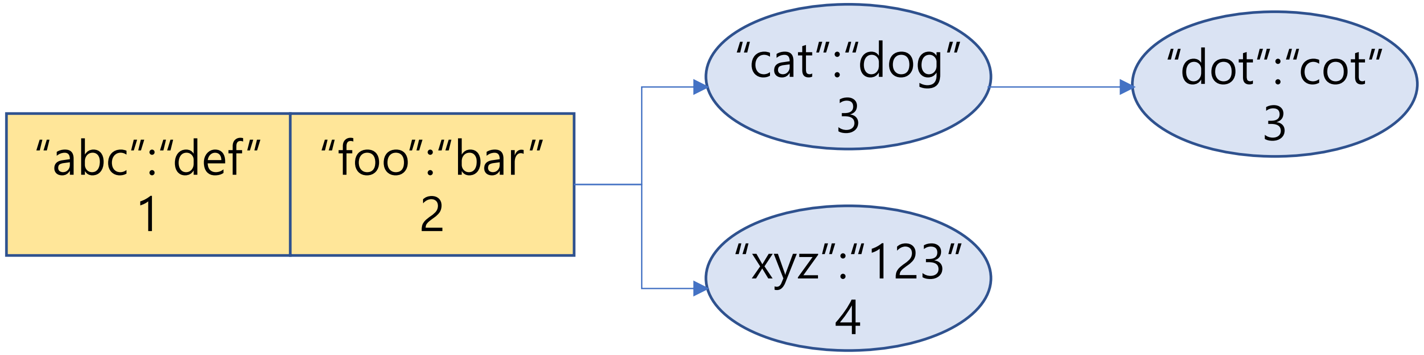


Multi-Paxos

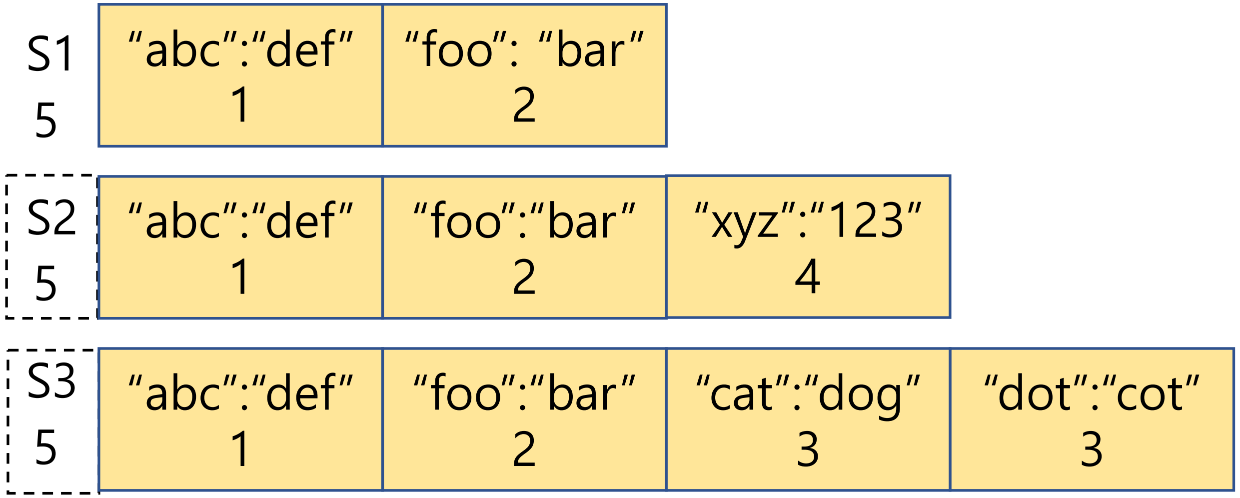


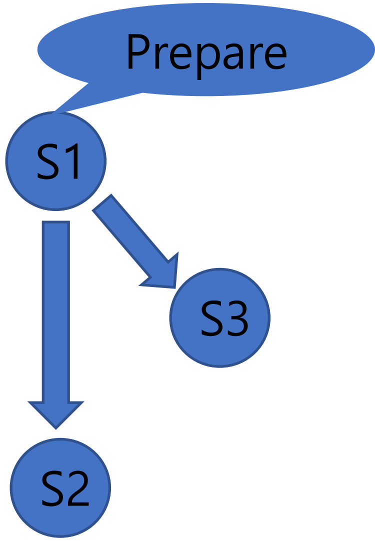
ADO operations

ADO



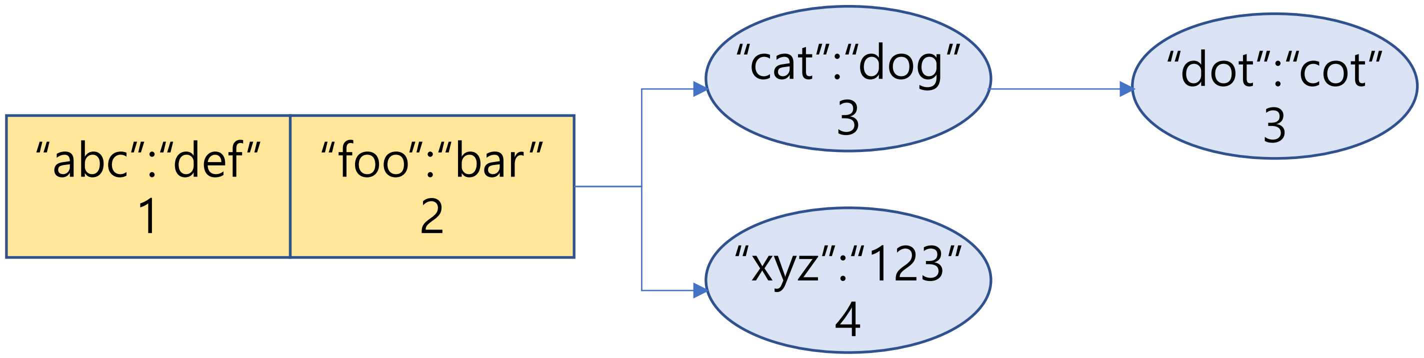
Multi-Paxos



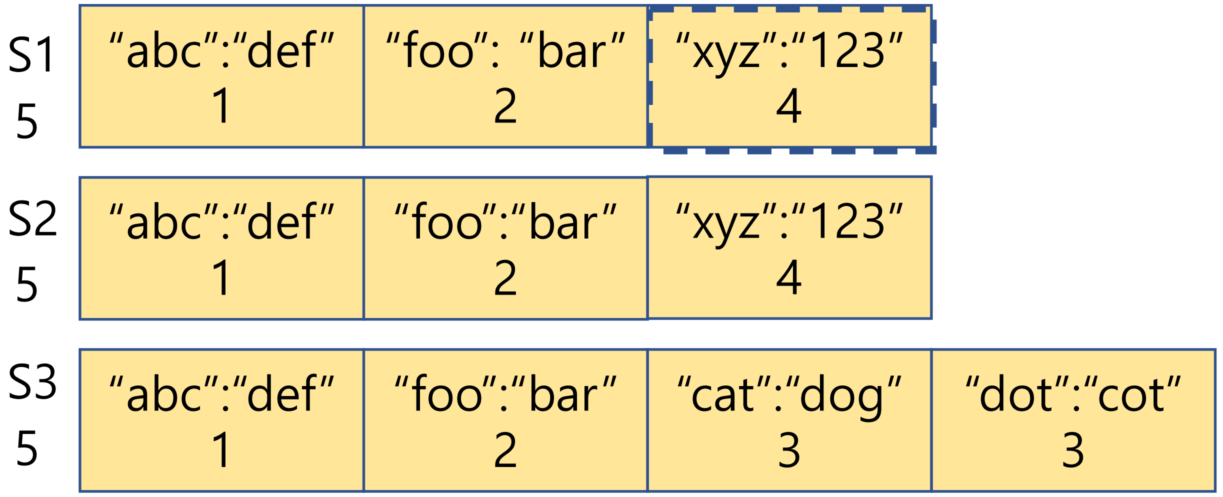


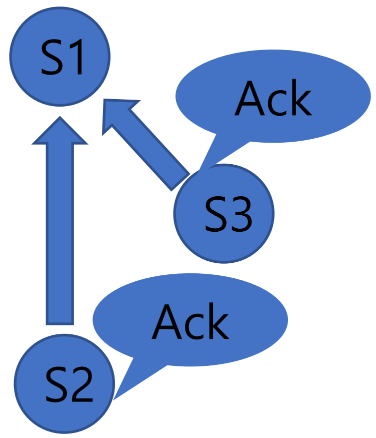
ADO operations

ADO

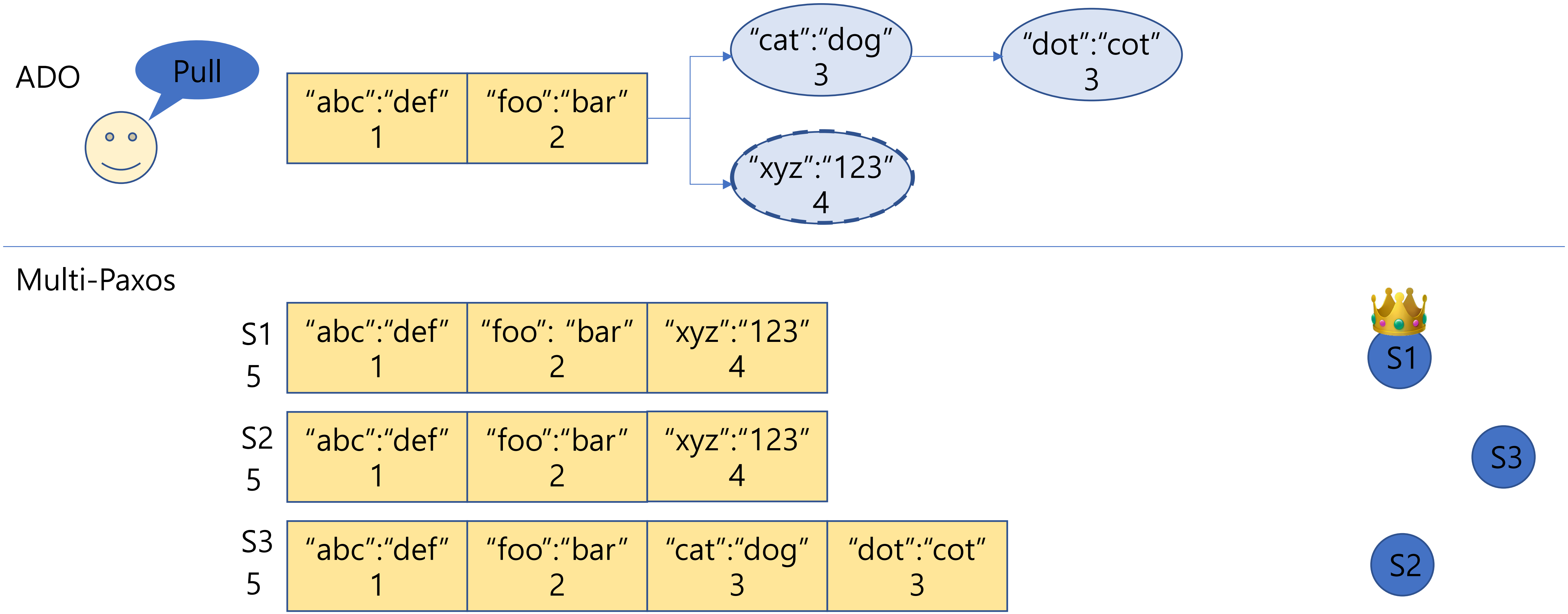


Multi-Paxos





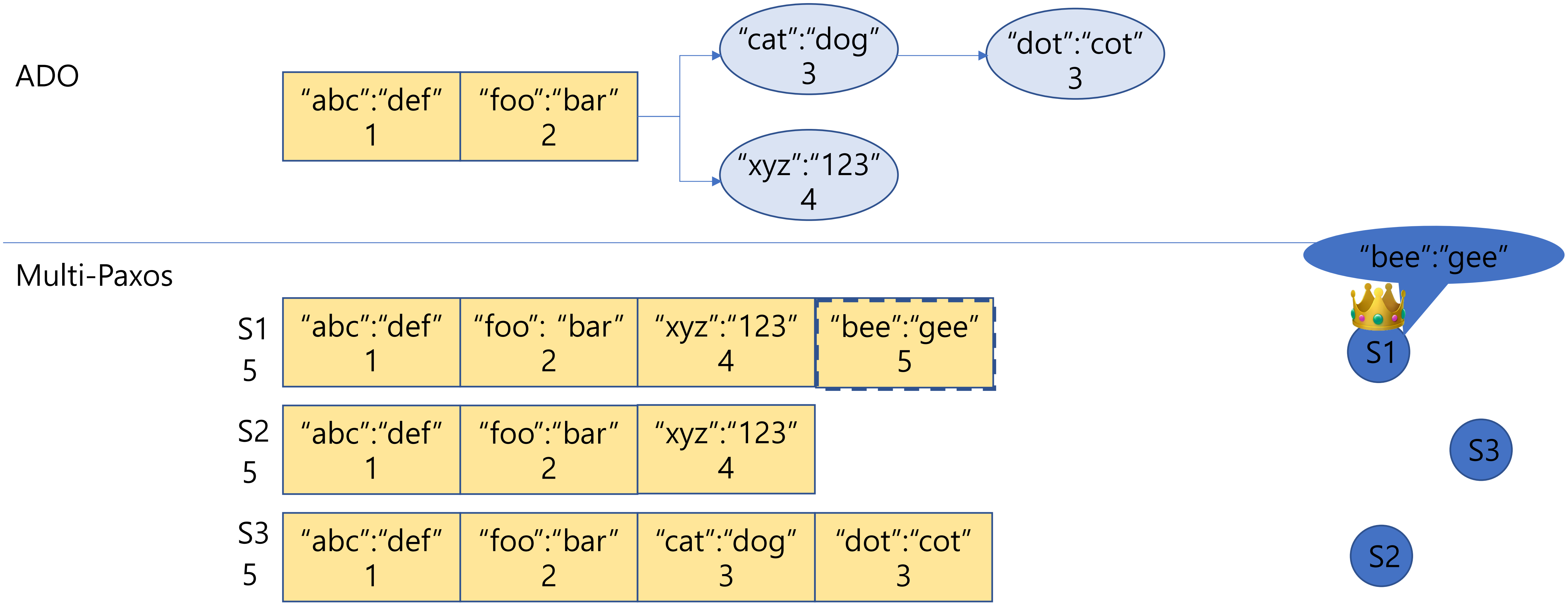
ADO operations



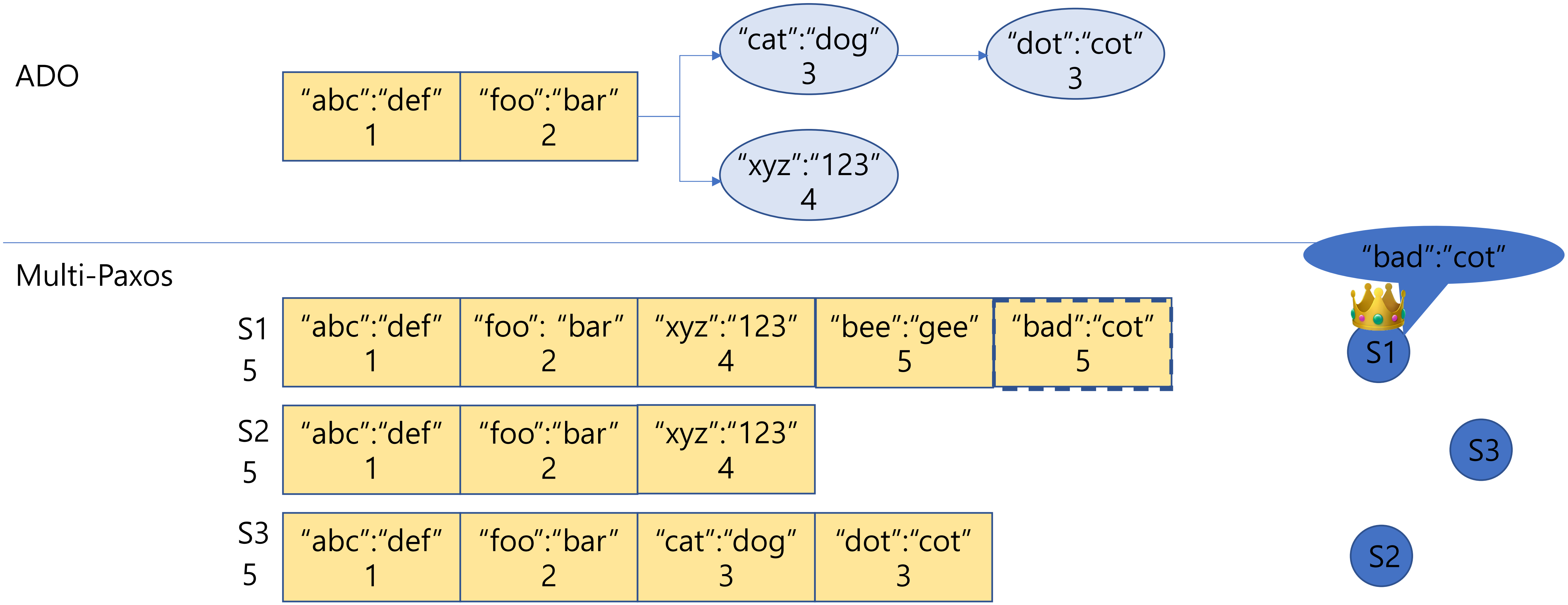
Pull

Get permission to update and select a starting point in the cache tree.

ADO Operations

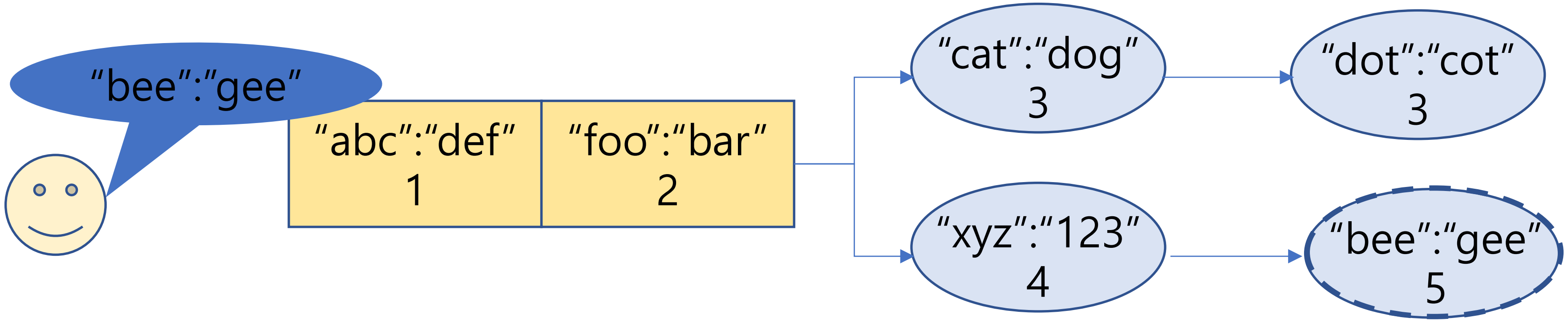


ADO operations



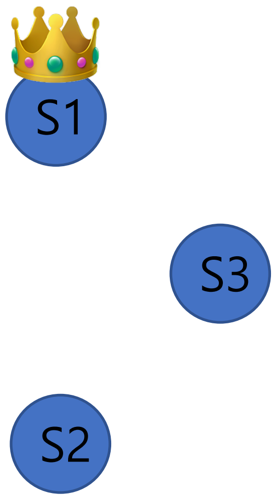
ADO operations

ADO



Multi-Paxos

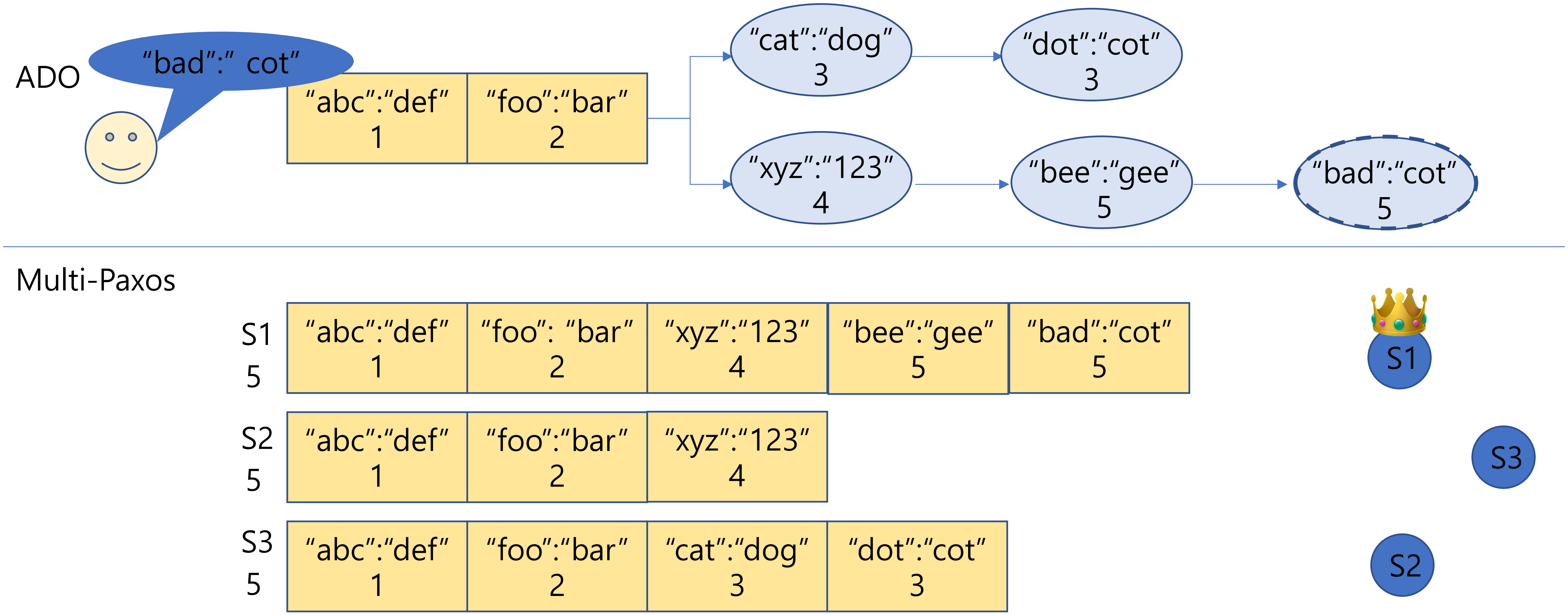
S1 5	"abc":"def" 1	"foo": "bar" 2	"xyz":"123" 4	"bee":"gee" 5	"bad":"cot" 5
S2 5	"abc":"def" 1	"foo":"bar" 2	"xyz":"123" 4		
S3 5	"abc":"def" 1	"foo":"bar" 2	"cat":"dog" 3	"dot":"cot" 3	



Invoking a Method

Add a new entry to the cache tree.

ADO operations

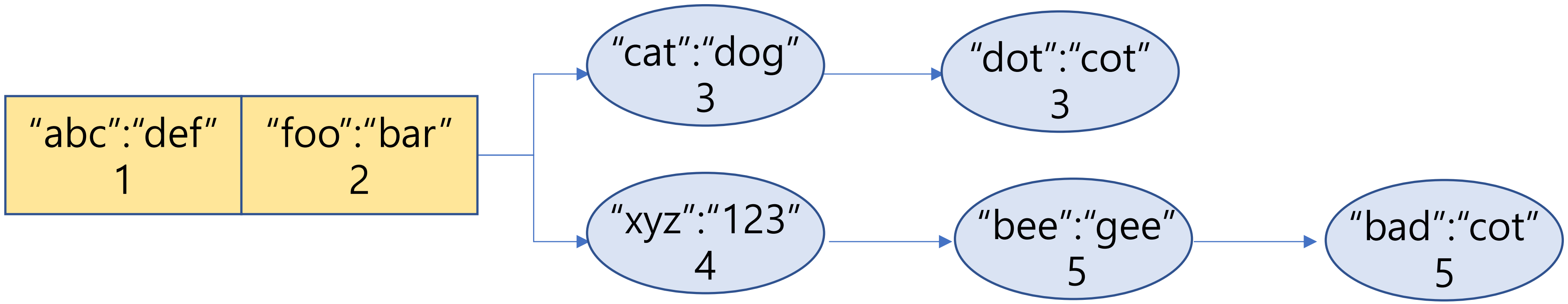


Invoking a Method

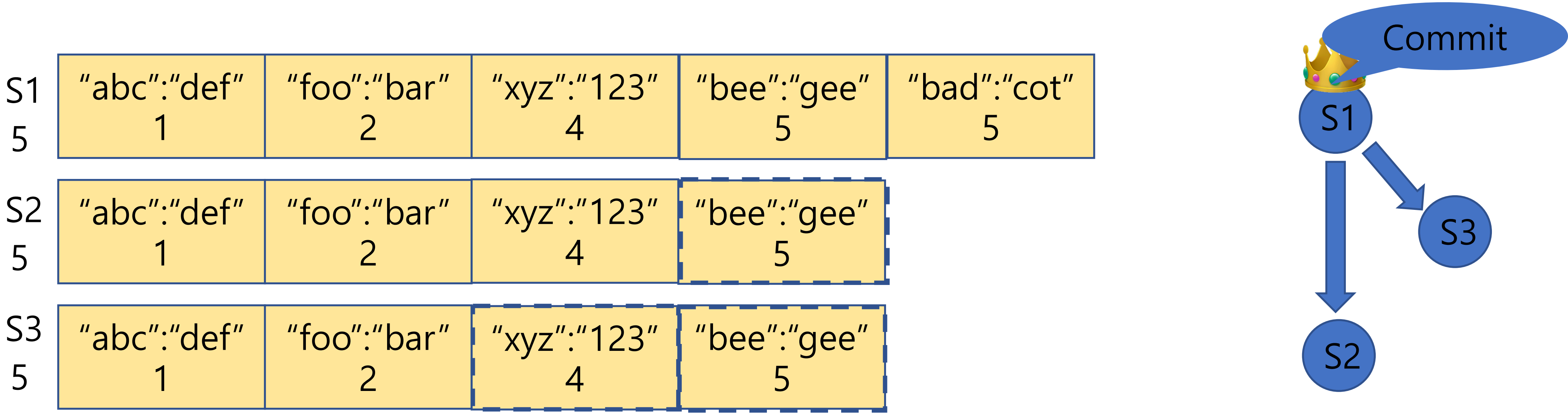
Add a new entry to the cache tree.

ADO operations

ADO

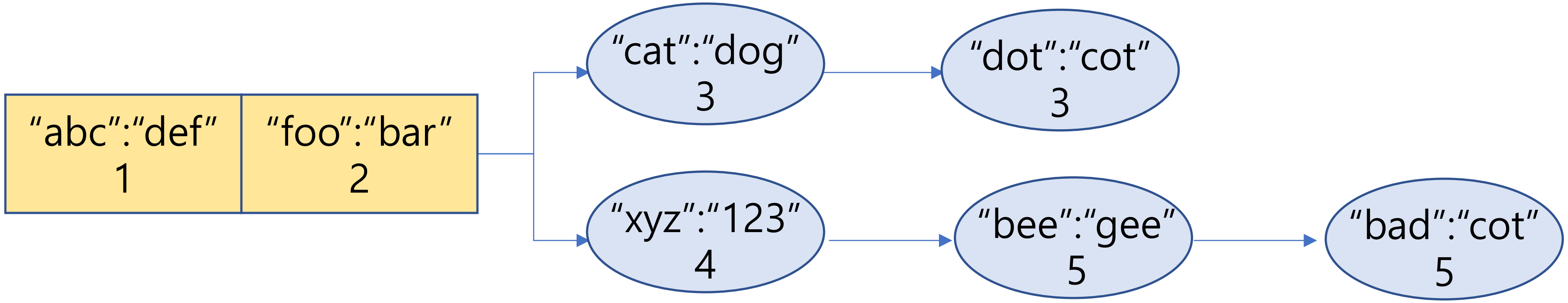


Multi-Paxos



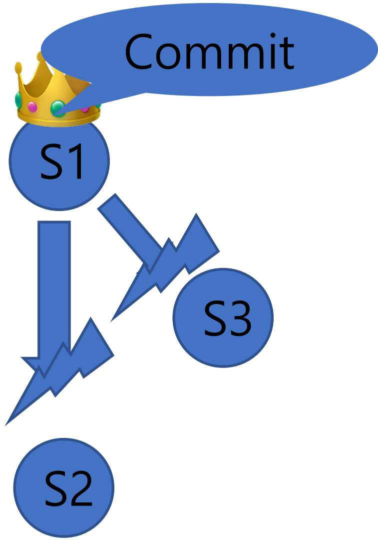
ADO operations

ADO

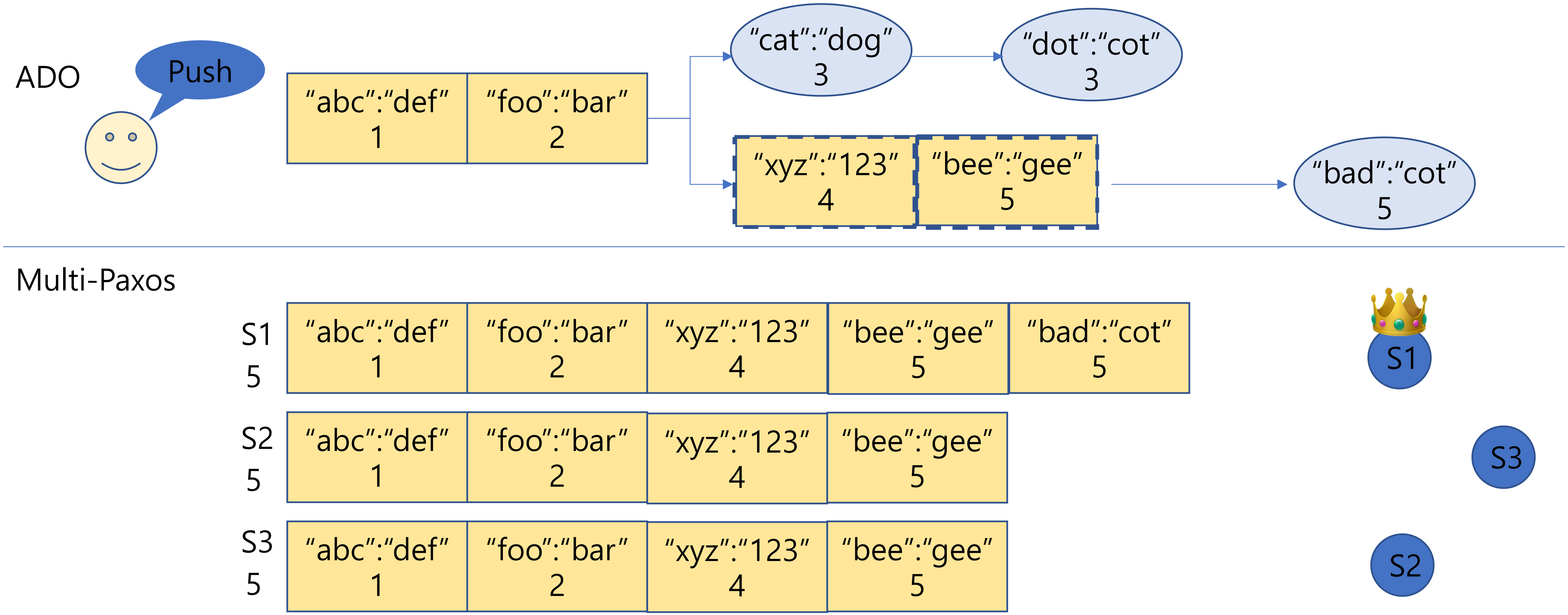


Multi-Paxos

S1 5	"abc":"def" 1	"foo":"bar" 2	"xyz":"123" 4	"bee":"gee" 5	"bad":"cot" 5
S2 5	"abc":"def" 1	"foo":"bar" 2	"xyz":"123" 4	"bee":"gee" 5	
S3 5	"abc":"def" 1	"foo":"bar" 2	"xyz":"123" 4	"bee":"gee" 5	



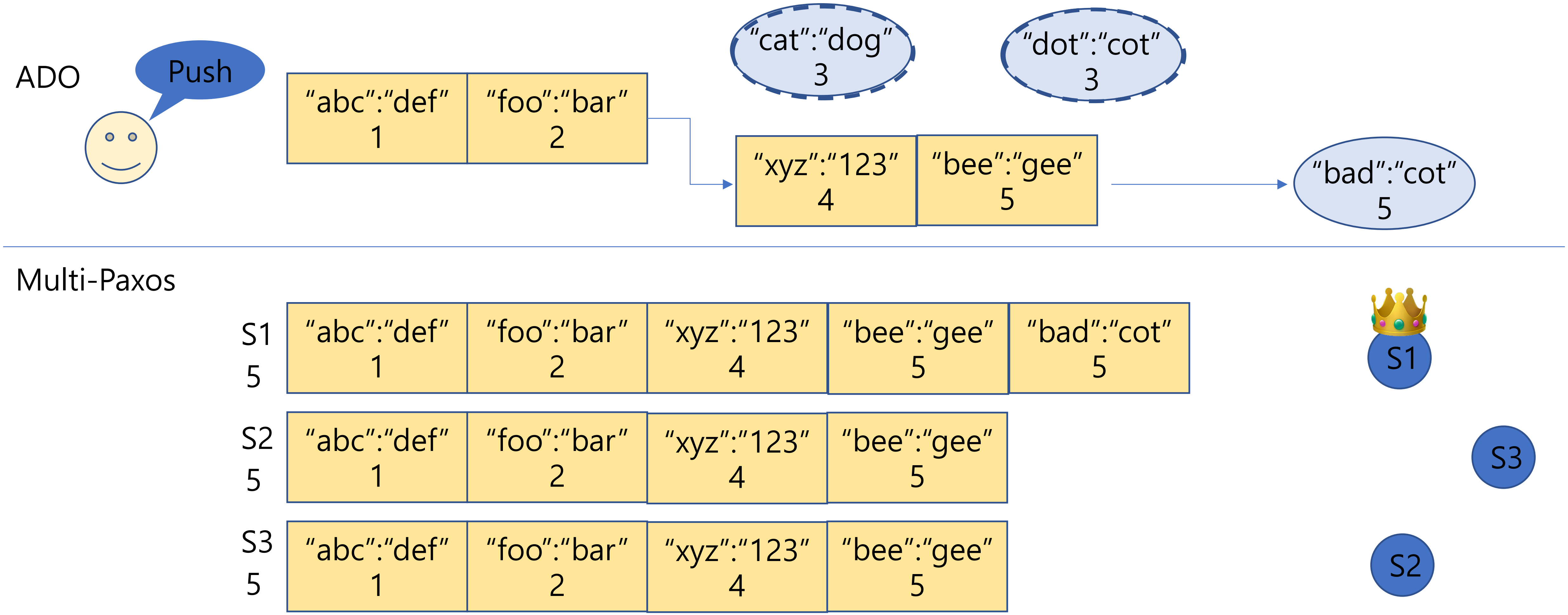
ADO operations



Push

Move committed methods into the log and prune stale states from the tree.

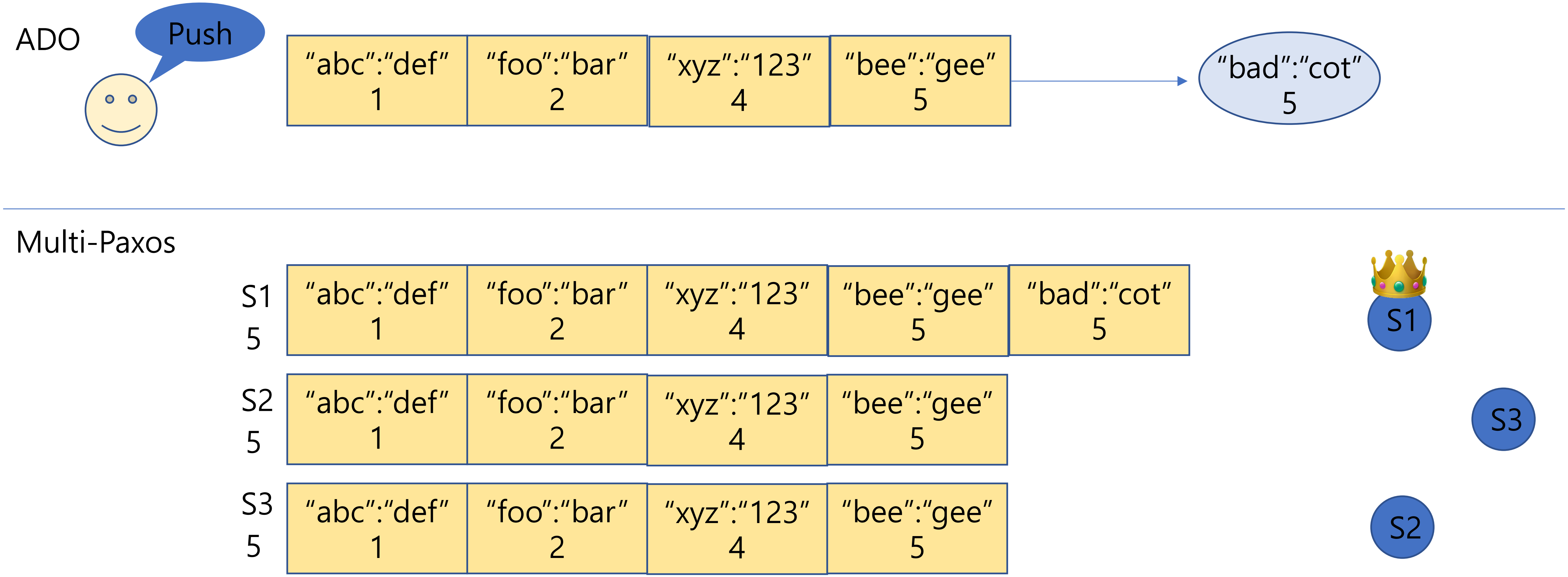
ADO operations



Push

Move committed methods into the log and prune stale states from the tree.

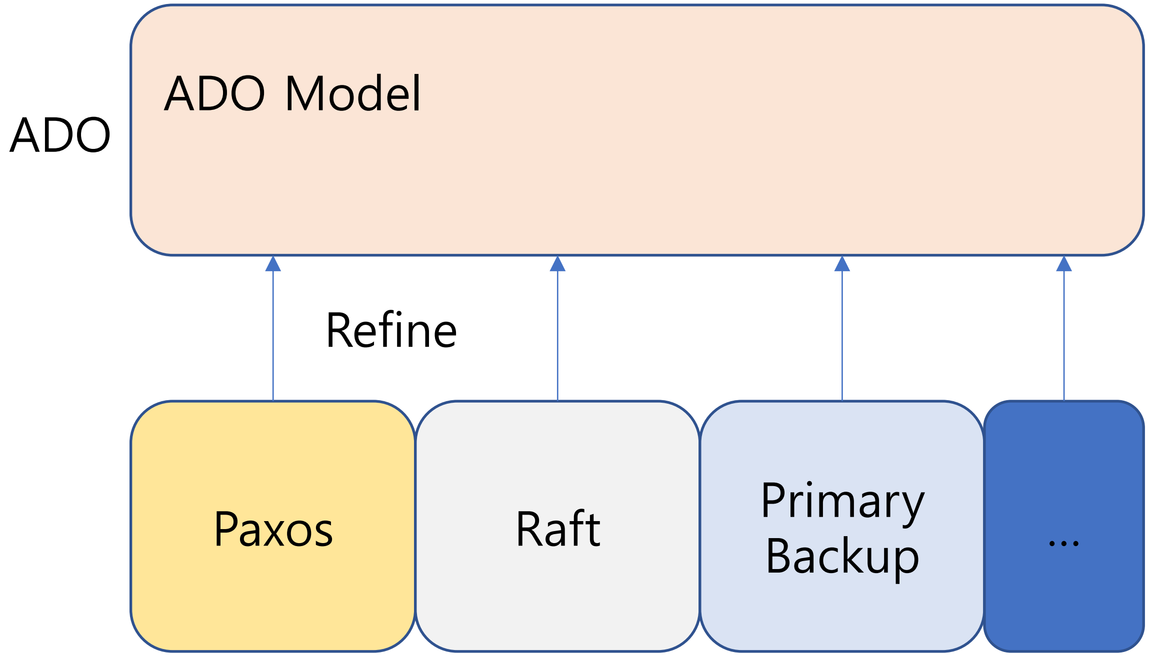
ADO operations



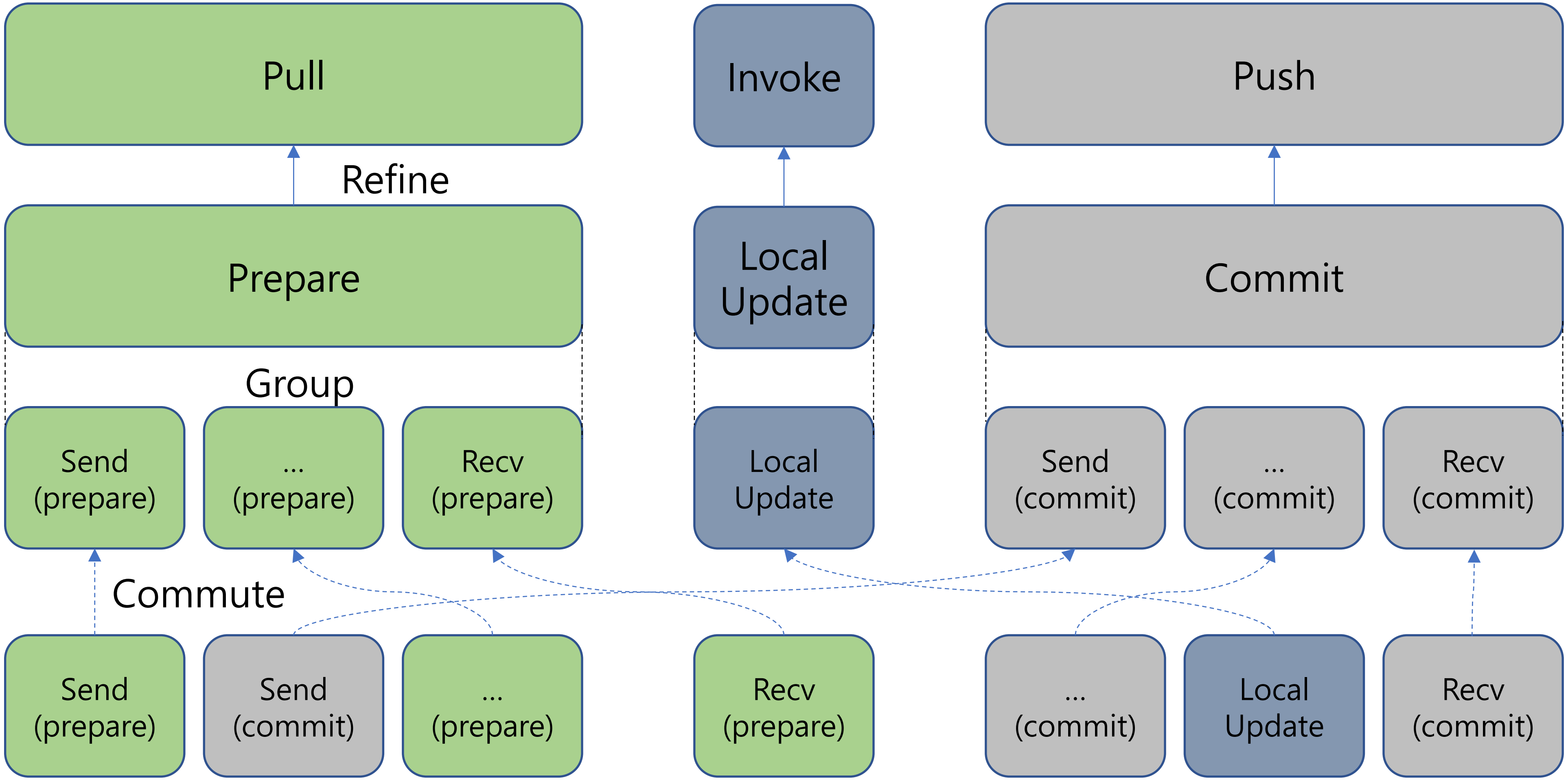
Push

Move committed methods into the log and prune stale states from the tree.

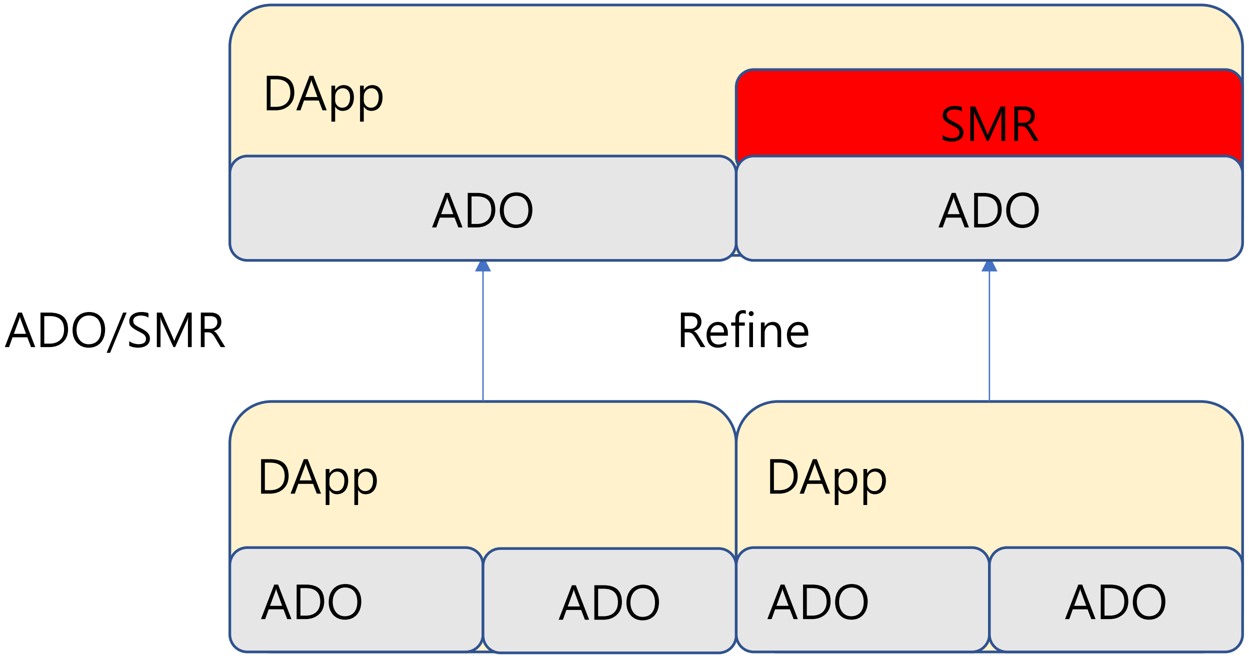
Connection with distributed protocols



Connection with distributed protocols



Distributed applications



Conclusion

Conclusion

- Formal verification can reduce the cost for the poor software
  - Operational software failure cost
  - Cost due to poor legacy systems
- Formal verification
  - What is formal verification
  - Formal verification key concept
  - Modularity in formal verification
- ADO: formal verification project example
  - Distributed system formal verification
  - Unified and modular program abstractions for distributed systems