An Empirical Study on the Correctness of Formally Verified Distributed Systems

Pedro Fonseca, Kaiyuan Zhang, Xi Wang,
Arvind Krishnamurthy

UNIVERSITY of WASHINGTON

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

We need **robust** distributed systems

- Distributed systems are critical!
- Reasoning about **concurrency** and **fault-tolerance** is extremely challenging

Verification of distributed systems

Recently applied to **implementations** of DSs

MultiPaxos

Raft

Causal KV

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

Abstract

Distributed systems are notorious for harboring subtle bugs. Verification can, in principle, eliminate these bugs a priori, but verification has historically been difficult to apply at full-program scale, much less distributed-system scale.

We describe a methodology for building practical and provably correct distributed systems based on a unique blend of TLA-style state-machine refinement and Hoare-logic verification. We demonstrate the methodology on a complex implementation of a Paxos-based replicated state machine library and a lease-based sharded key-value store. We prove that each obeys a concise safety specification, as well as desirable liveness requirements. Each implementation achieves performance competitive with a reference system. With our methodology and lessons learned, we aim to raise the standard for distributed systems from "tested" to "correct."

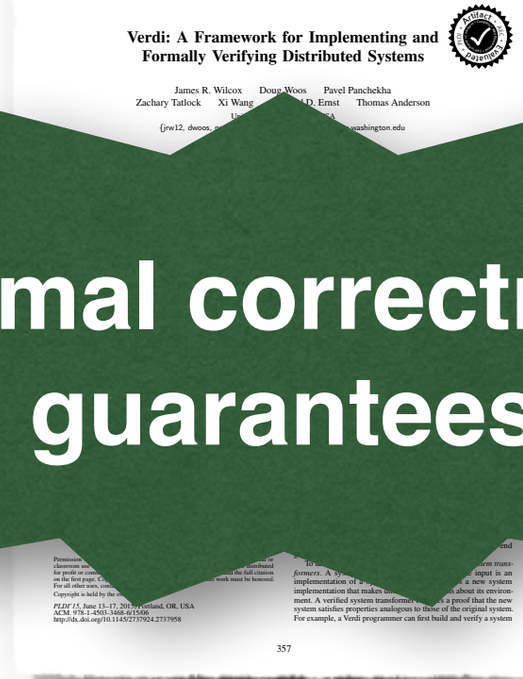
1. Introduction

Distributed systems are notoriously hard to get right. Protocol designers struggle to reason about concurrent execution on multiple machines, which leads to subtle errors. 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.

In theory, formal verification can categorically eliminate errors from distributed systems. However, due to the complexity of these systems, previous work has primarily focused on specifying [4, 13, 27, 41, 48, 54], verifying [3, 52, 53, 59, 61], or at least bug-checking [20, 31, 69] distributed protocols, often in a simplified form, without extending such formal reasoning to the implementations. In principle, one can use model checking to reason about the correctness of both protocols [42, 59] and implementations [46, 47, 69]. In practice, however, model checking is incomplete—the accuracy of the results depends on the accuracy of the model—and does not scale [4].

This paper presents IronFleet, the first methodology for automated machine-checked verification of the safety and liveness of non-trivial distributed system implementations. The IronFleet methodology is practical: it supports complex, feature-rich implementations with reasonable performance and a tolerable proof burden.

Ultimately, IronFleet guarantees that the implementation of a distributed system meets a high-level, centralized specification. For example, a sharded key-value store acts like a key-value store, and a replicated state machine acts like a state machine. This guarantee categorically rules out race conditions, violations of global invariants, integer overflow, bugs in rare



Verdi: A Framework for Implementing and Formally Verifying Distributed Systems

James R. Wilcox Doug Woos Pavel Panchekha
Zachary Tatlock Xi Wang D. Ernst Thomas Anderson

{jrw12, dwoos, washington.edu

357

Chapar: Certified Causally Consistent Distributed Key-Value Stores

Mohsen Lesani Christian J. Bell Adam Chlipala

Massachusetts Institute of Technology, USA
{lesani, cjbell, adamc}@mit.edu

Program 1 (p_1): Uploading a photo and posting a status

```
0 →                                          Alice
    put(Pic, 📷);                            ▷ uploads a new photo
    put(Post, 🏔);                           ▷ announces it to her friends
1 →                                          Bob
    post ← get(Post);                        ▷ checks Alice's post
    photo ← get(Pic);                        ▷ then loads her photo
    assert(post = 🏔 ⇒ photo ≠ ⊥)
```


Figure 1. Inconsistent trace of Photo-Upload example

the downtime of a replica, other replicas can keep the service available, and the locality of replicas enhances responsiveness.

On the flip side, maintaining strong consistency across replicas [30] can limit parallelism [35] and availability. When availability is a must, the CAP theorem [19] formulates a fundamental trade-off between strong consistency and partition tolerance, and PACELC [3] formulates a trade-off between strong consistency and latency [5]. In reaction to these constraints, modern storage systems including Amazon's Dynamo [17], Facebook's Cassandra [27], Yahoo's PNUTS [16], LinkedIn's Voldemort [1], and memcached [2] have adopted relaxed notions of consistency that are collectively called *eventual consistency* [48]. The main guarantee that eventually consistent stores provide is that if clients stop issuing updates, then the replicas will converge to the same state. Researchers [13, 44, 46] have proposed eventually consistent algorithms for common datatypes like registers, counters, and finite sets. Recent work [12, 14, 54] has formalized and verified the eventual-consistency condition for these algorithms.

Weaker consistency is a double-edged sword. It can lead to more efficient and fault-tolerant algorithms, but at the same time it exposes clients to less consistent data. Programming with weak consistency is challenging and error-prone. As an example, consider Program 1, which shows two client routines (0 for Alice and 1 for Bob) running concurrently. An execution of the program with an eventually consistent store is shown in Figure 1. Alice uploads a photo of herself 😊 and then posts a message that she has uploaded a photo 🏔. Bob reads Alice's post announcing the upload. He attempts to see the photo but only sees the default value. The message containing the photo arrives late. The post is issued after the photo is uploaded in Alice's node. We call this a *node-order dependency* from the post to the upload. If Bob can see the

357

Formal correctness guarantees

IronFleet [SOSP'15]

Verdi [PLDI'15]

Chapar [POPL'16]

Are verified systems bug-free?

We found 16 bugs in the three verified systems

	Bug consequence	Component	Trigger
1	Crash server	Client-server	Partial socket read
2	Inject commands	Client-server	Client input
3	Crash server	Recovery	Replica crash
4	Crash server	Recovery	Replica crash
5	Incomplete recovery	Recovery	OS error on recovery
6	Crash server	Server communication	Lagging replica
7	Crash server	Server communication	Lagging replica
8	Crash server	Server communication	Lagging replica
9	Violate causal	Server communication	Packet duplication
10	Return stale results	Server communication	Packet loss
11	Hang and corrupt data	Server communication	Client input
12	Void exactly-once	High-level specification	Packet duplication
13	Void client guarantee	Test case check	-
14	Verify incorrect	Verification framework	Incompatible libraries
15	Verify incorrect	Verification framework	Signal
16	Prevent verification	Binary libraries	-

Are verified systems bug-free?

We found 16 bugs in the three verified systems

	Bug consequence	Component	Trigger
1	**Crash server**	**Client-server**	**Partial socket read**
2	Inject commands	Client-server	Client input
3	**Crash server**	**Recovery**	**Replica crash**
4	**Crash server**	**Recovery**	**Replica crash**
5	Incomplete recovery	Recovery	OS error on recovery
6	**Crash server**	**Server communication**	**Lagging replica**
7	**Crash server**	**Server communication**	**Lagging replica**
8	**Crash server**	**Server communication**	**Lagging replica**
9	Violate causal	Server communication	Packet duplication
10	Return stale results	Server communication	Packet loss
11	Hang and corrupt data	Server communication	Client input
12	Void exactly-once	High-level specification	Packet duplication
13	Void client guarantee	Test case check	-
14	Verify incorrect	Verification framework	Incompatible libraries
15	Verify incorrect	Verification framework	Signal
16	Prevent verification	Binary libraries	-

Are verified systems bug-free?

We found 16 bugs in the three verified systems

	Bug consequence	Component	Trigger
1	**Crash server**	**Client-server**	**Partial socket read**
2	**Inject commands**	**Client-server**	**Client input**
3	**Crash server**	**Recovery**	**Replica crash**
4	**Crash server**	**Recovery**	**Replica crash**
5	**Incomplete recovery**	**Recovery**	**OS error on recovery**
6	**Crash server**	**Server communication**	**Lagging replica**
7	**Crash server**	**Server communication**	**Lagging replica**
8	**Crash server**	**Server communication**	**Lagging replica**
9	**Violate causal**	**Server communication**	**Packet duplication**
10	**Return stale results**	**Server communication**	**Packet loss**
11	**Hang and corrupt data**	**Server communication**	**Client input**
12	Void exactly-once	High-level specification	Packet duplication
13	Void client guarantee	Test case check	-
14	Verify incorrect	Verification framework	Incompatible libraries
15	Verify incorrect	Verification framework	Signal
16	Prevent verification	Binary libraries	-

Are verified systems bug-free?

We found 16 bugs in the three verified systems

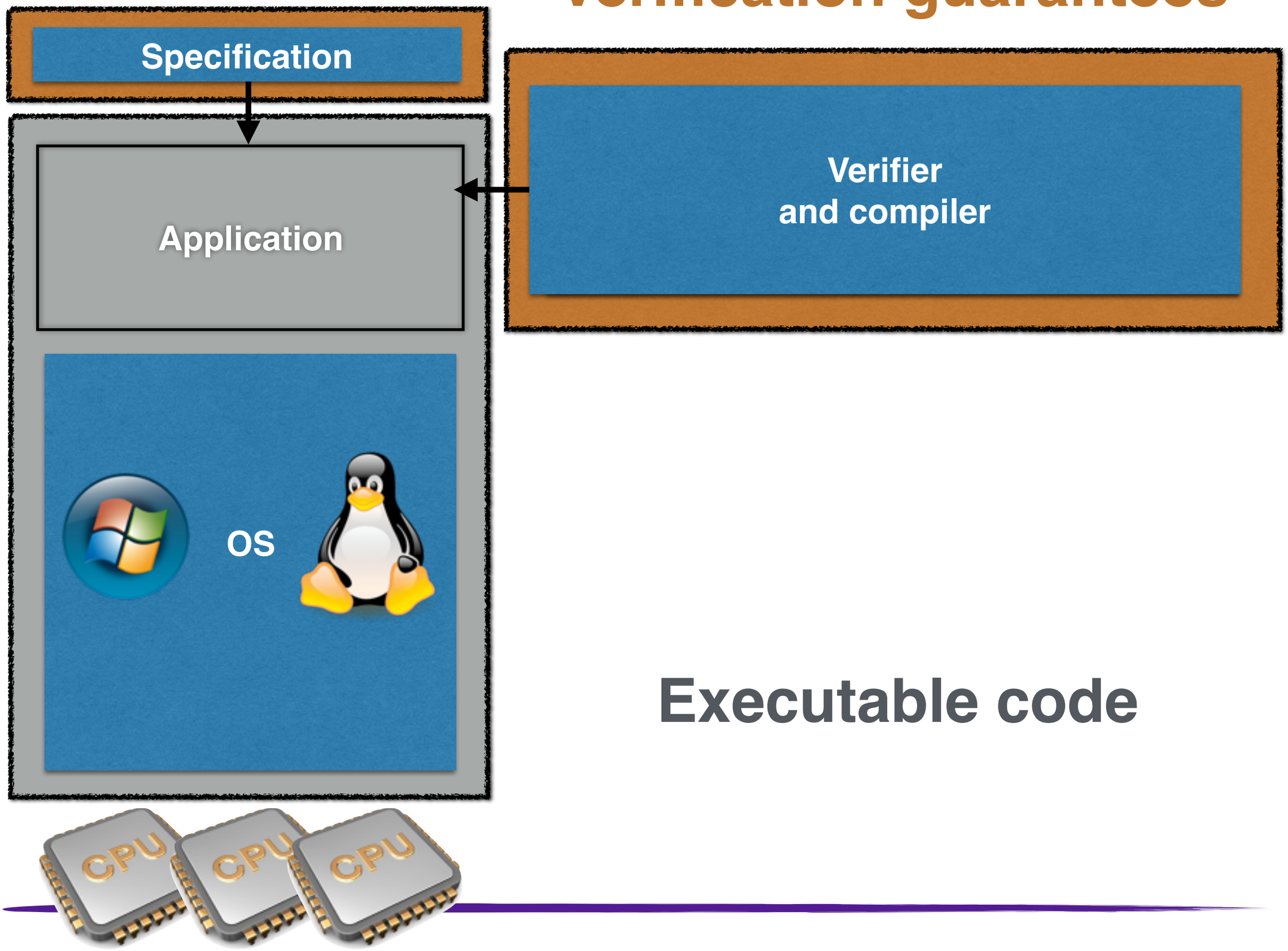
No protocol bugs found

	Bug consequence	Component	Trigger
1	Crash server	Client-server	Partial socket read
2	Inject commands	Client-server	Client input
3	Crash server	Recovery	Replica crash
4	Crash server	Recovery	Replica crash
5	Incomplete recovery	Recovery	OS error on recovery
6	Crash server	Server communication	Lagging replica
7	Crash server	Server communication	Lagging replica
8	Crash server	Server communication	Lagging replica
9	Violate causal	Server communication	Packet duplication
10	Return stale results	Server communication	Packet loss
11	Hang and corrupt data	Server communication	Client input
12	Void exactly-once	High-level specification	Packet duplication
13	Void client guarantee	Test case check	-
14	Verify incorrect	Verification framework	Incompatible libraries
15	Verify incorrect	Verification framework	Signal
16	Prevent verification	Binary libraries	-

All bugs were found in the trusted computing base

What are the components of the TCB?

Verification guarantees



Executable code

Verification guarantees

2 bugs

Specification

Verified code

11 bugs

Shim layer

OS

3 bugs

Aux. tools

Verifier
and compiler

Tiny fraction of the TCB

Executable code

Study methodology

Overall server correctness
(including non-verified components)

+

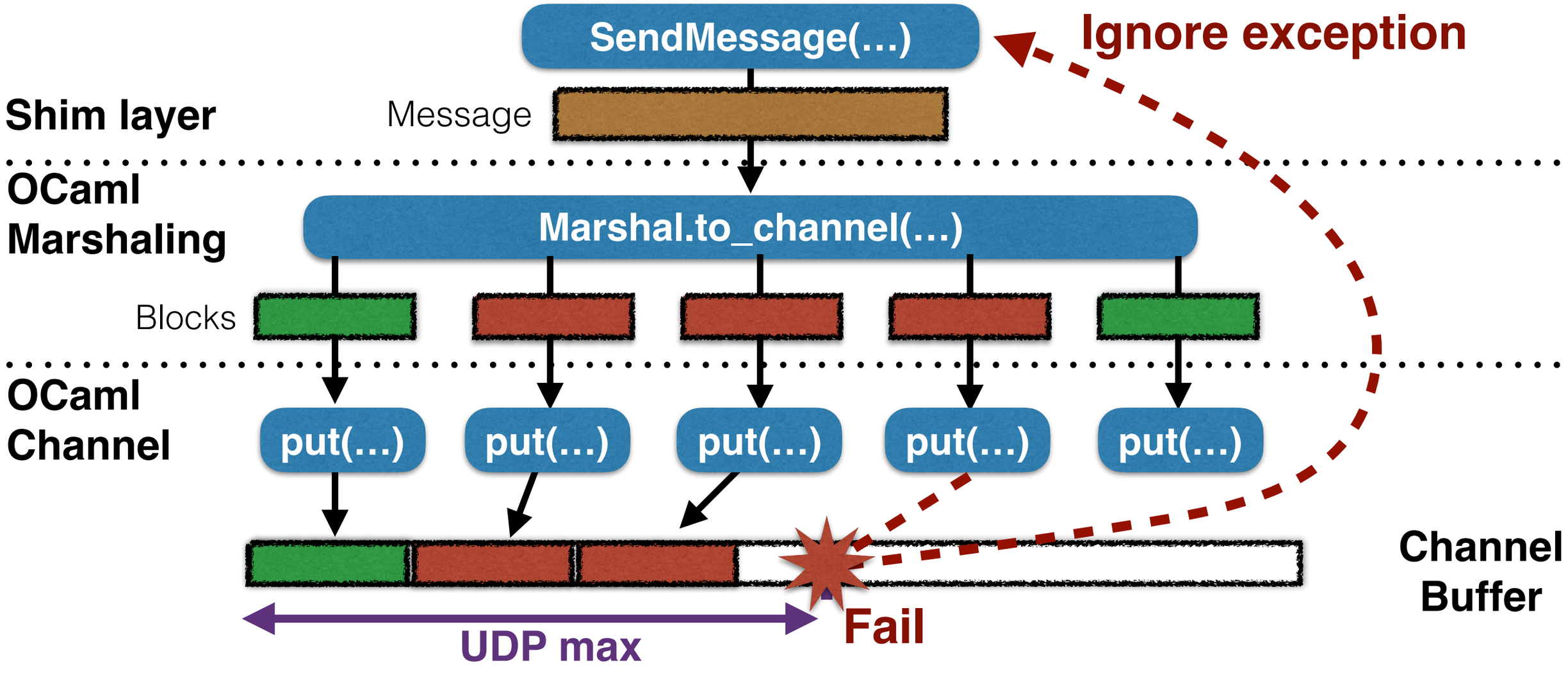
Verification guarantees

- Relied on code review, testing tools, and comparison between systems
- Analyzed source code, documentation, specification
- PK testing toolkit

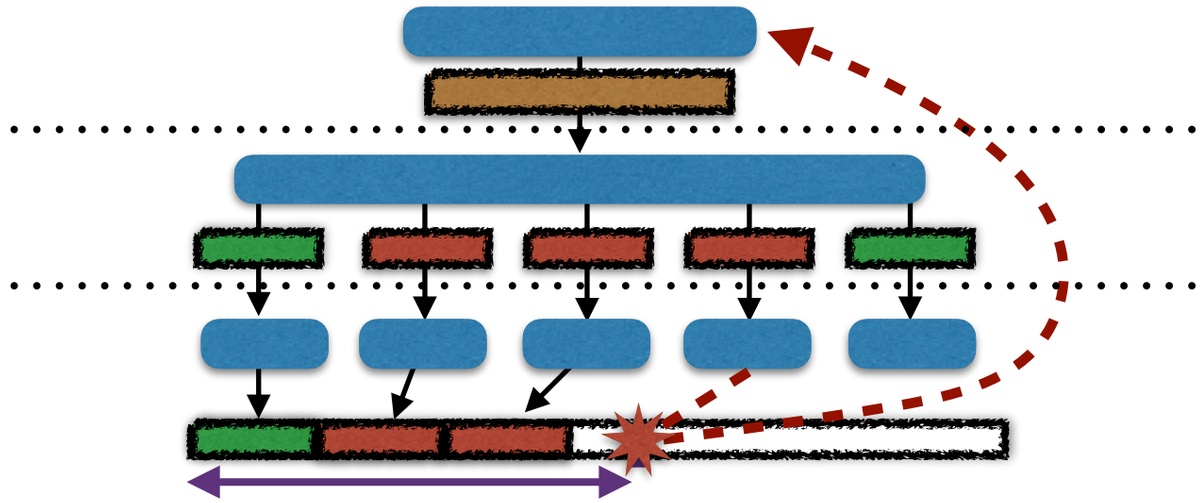
1. Shim layer bugs
2. Specification bugs
3. Verifier bugs
4. Towards “bug-free” distributed system

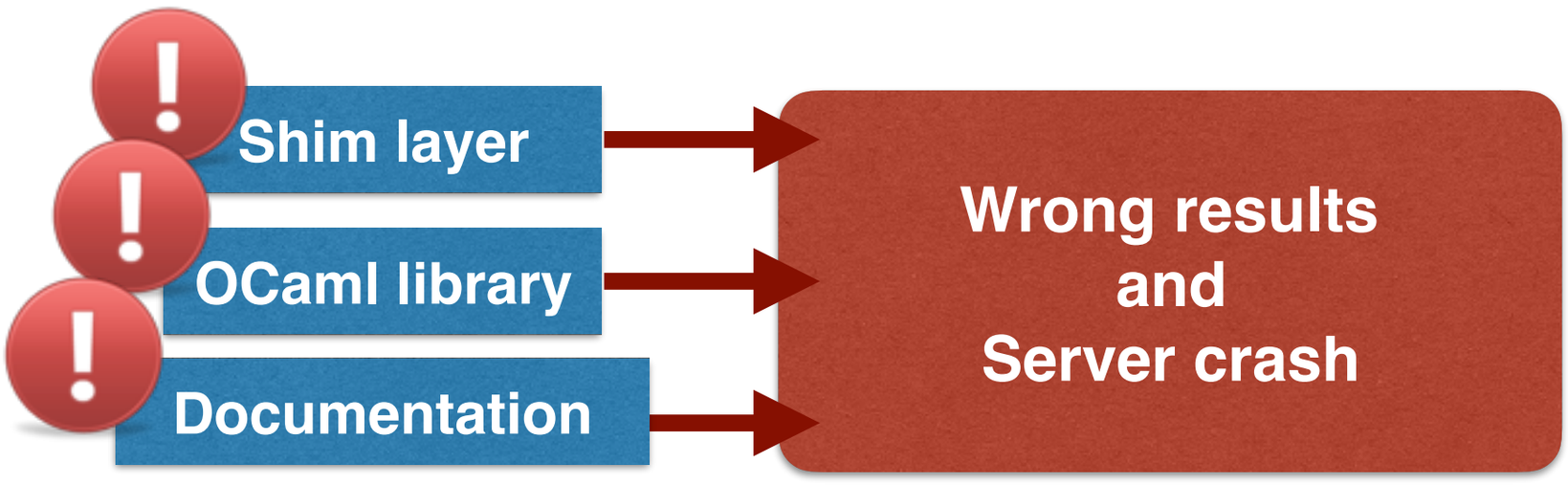
1. **Shim layer bugs**
2. Specification bugs
3. Verifier bugs
4. Towards “bug-free” distributed system

Example #1: Library semantics



Example #1: Library semantics





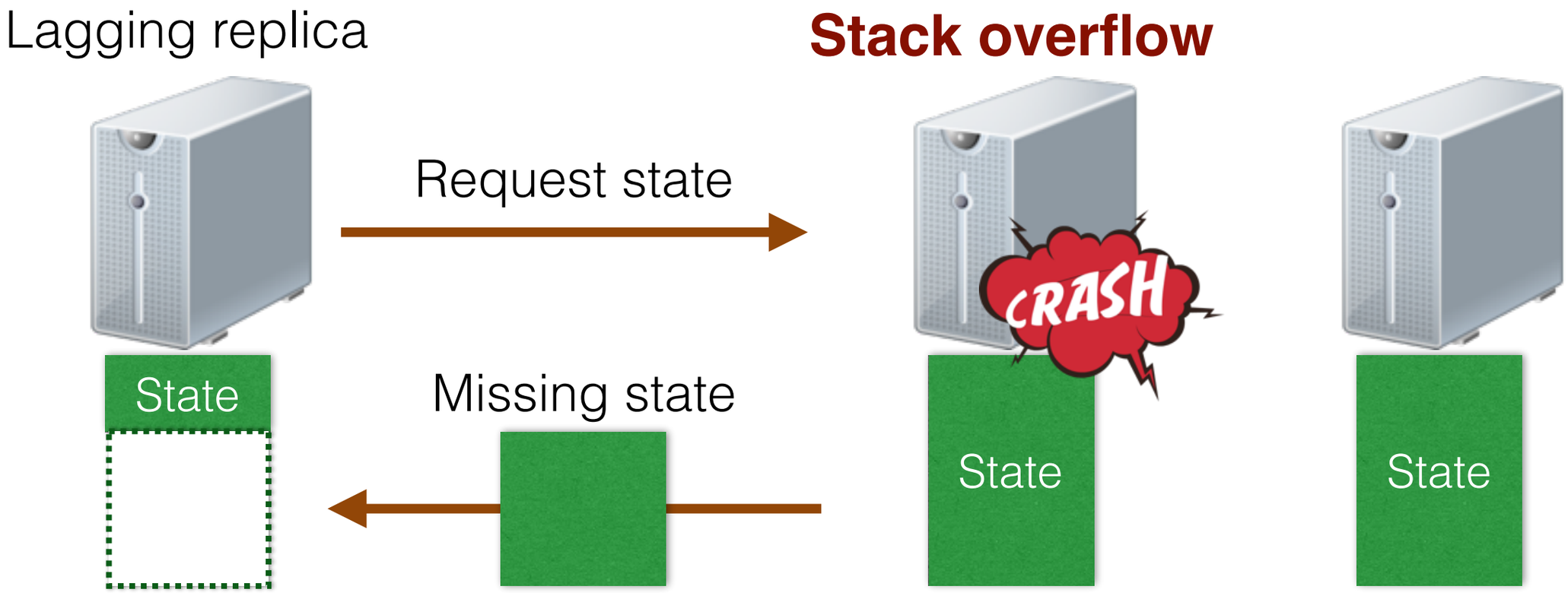
Example #2: Resource limits

State

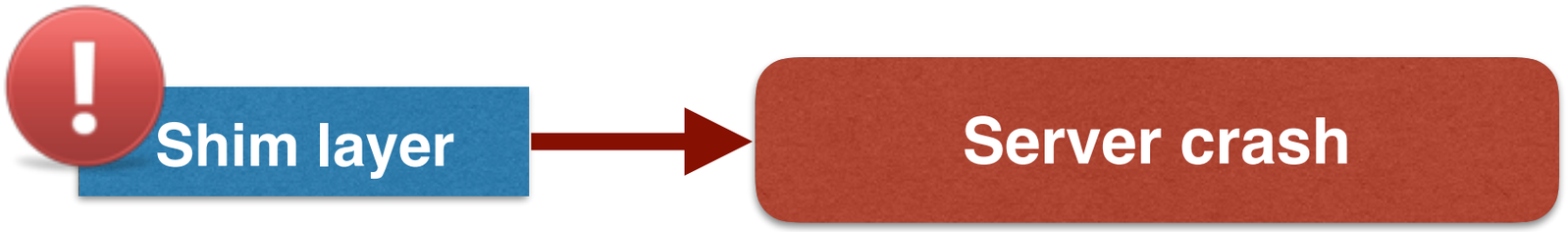
State

State

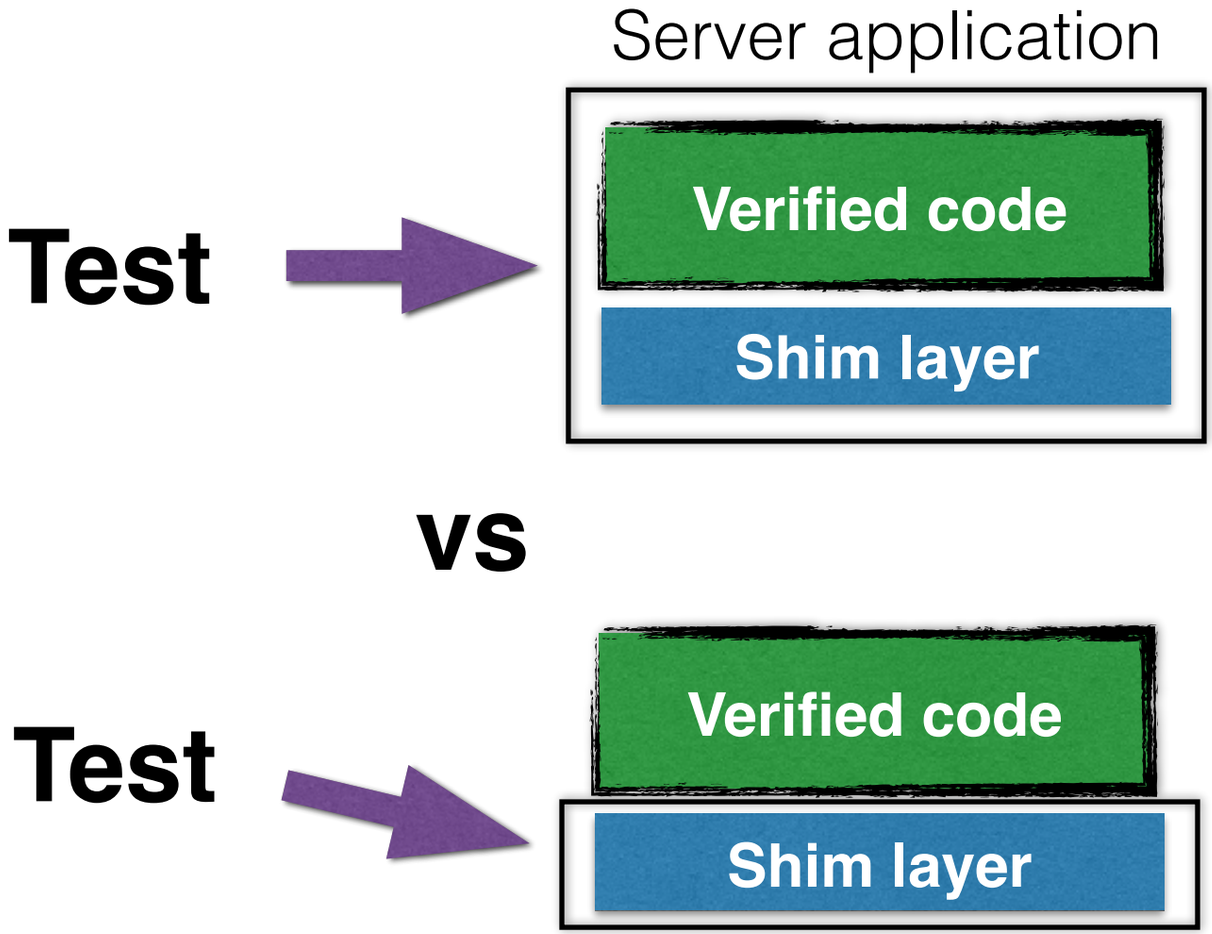
Example #2: Resource limits



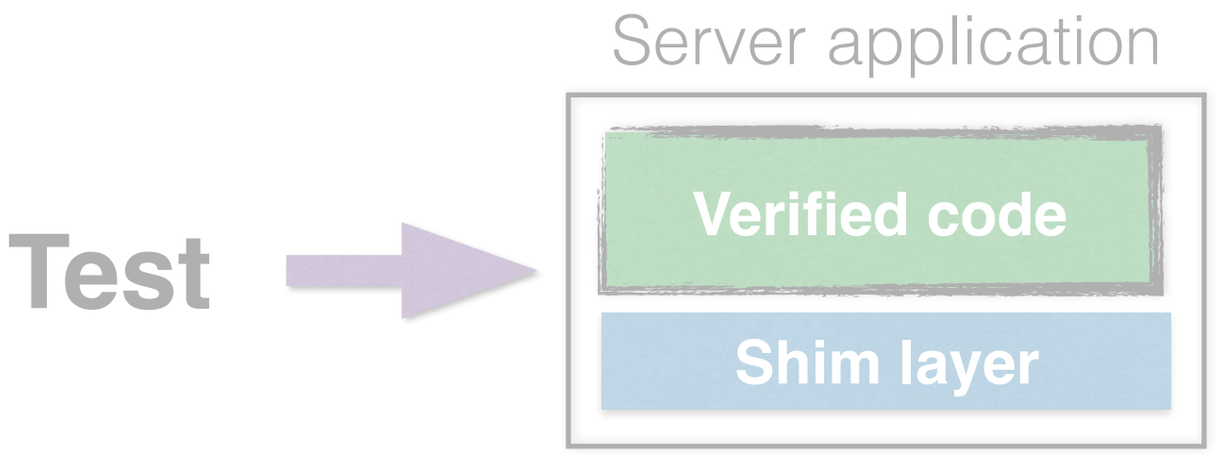
Large requests cause servers to crash



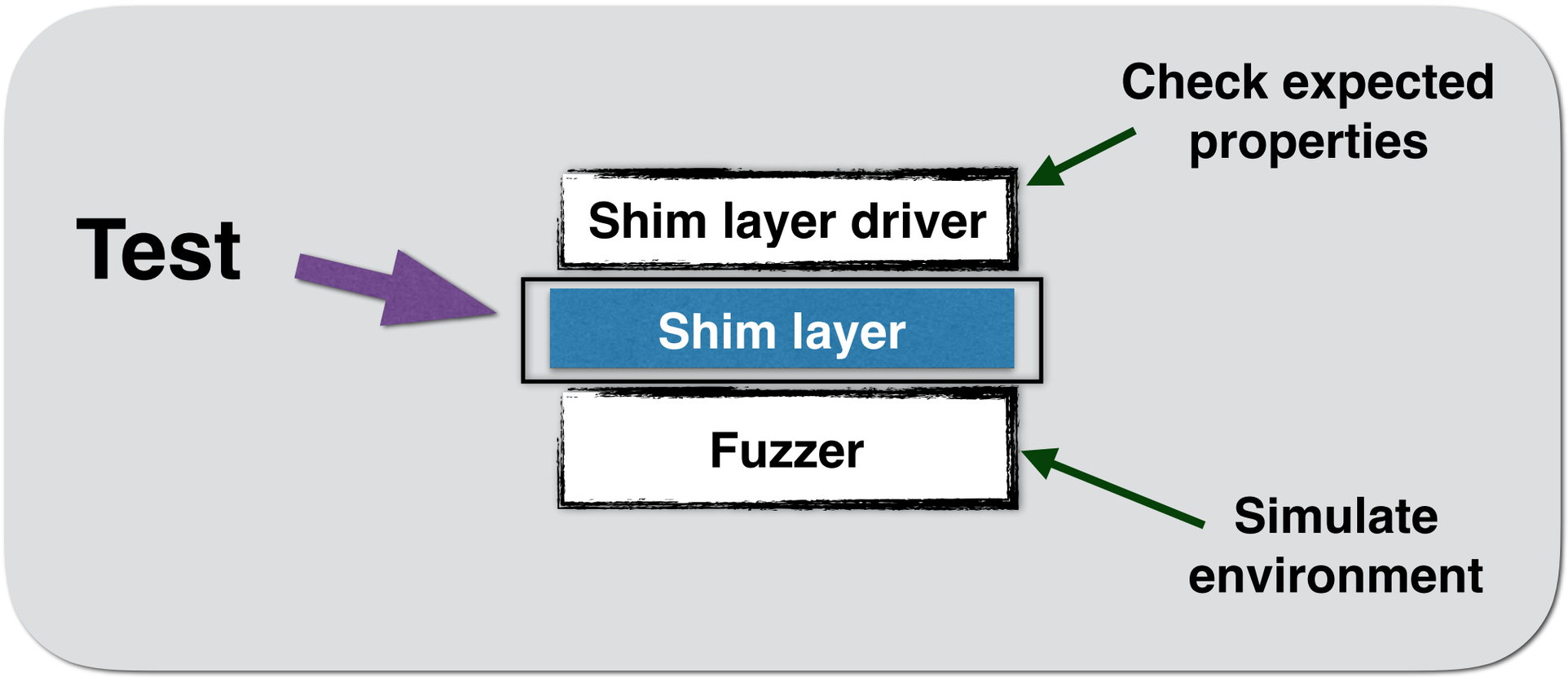
Preventing shim-layer bugs



Preventing shim-layer bugs



PK testing toolkit



1 Shim layer bugs

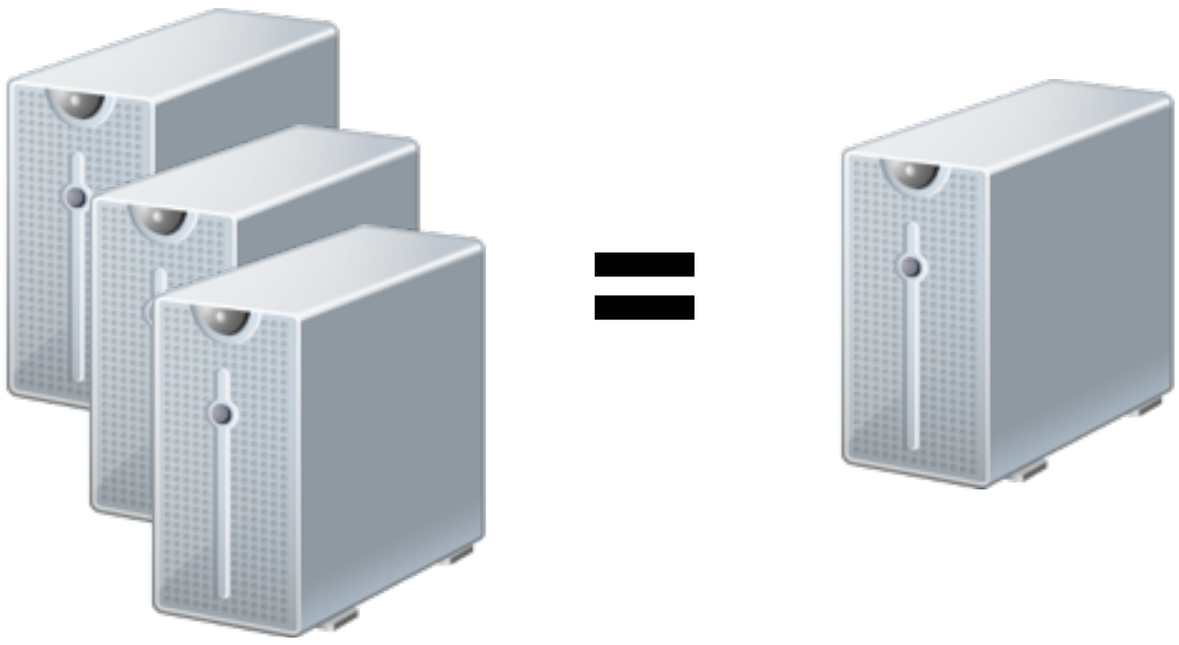
2 **Specification bugs**

3 Verifier bugs

4 Towards “bug-free” distributed system

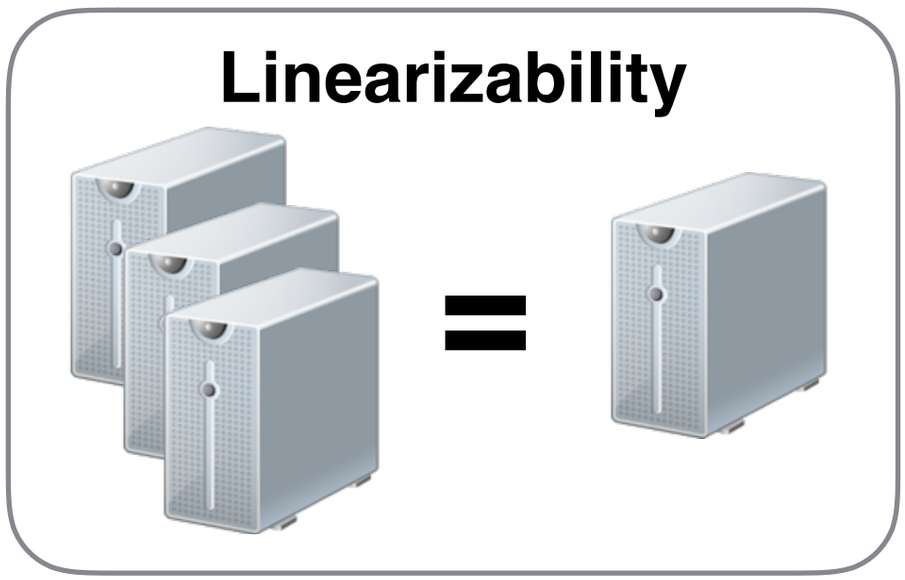
Example #3: Specification bug

Replicated state machine protocols

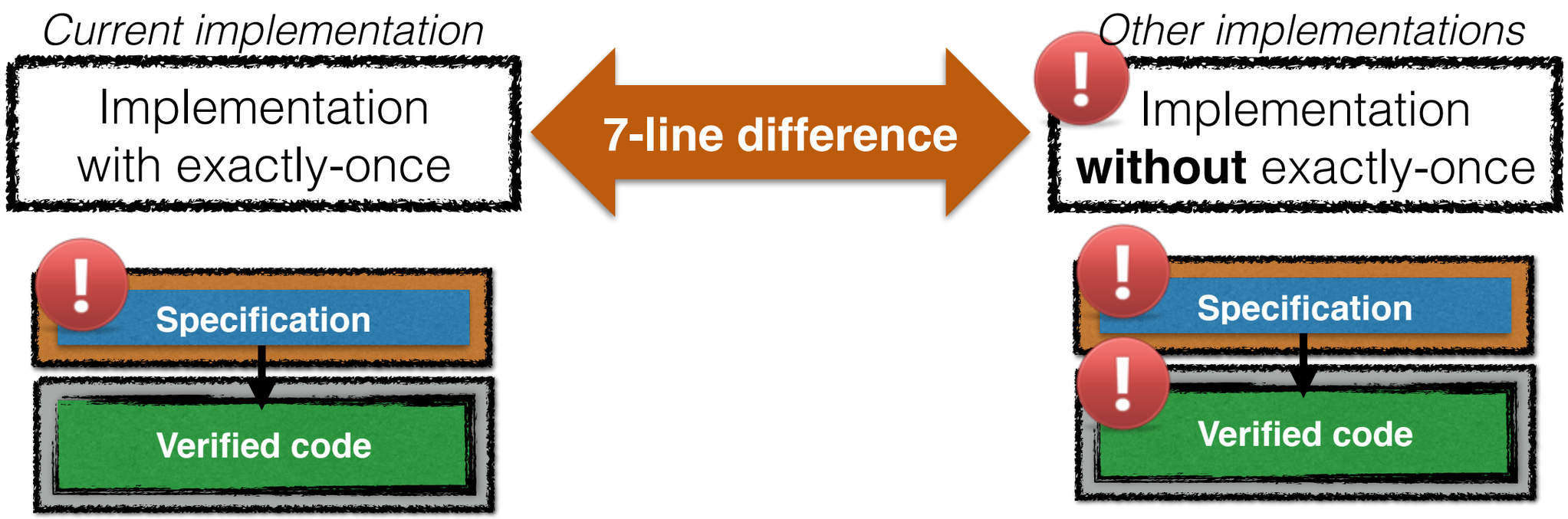


Linearizability

Example #3: Specification bug

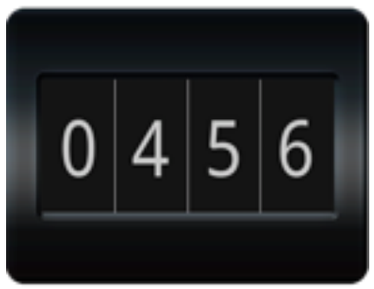


Ensure that operations
are executed exactly once

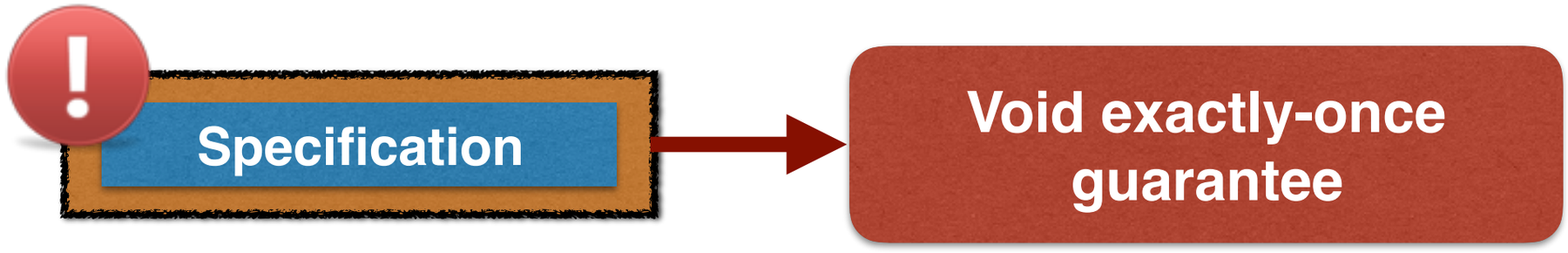


Example #3: Specification bug

- Exactly-once semantics is critical for applications

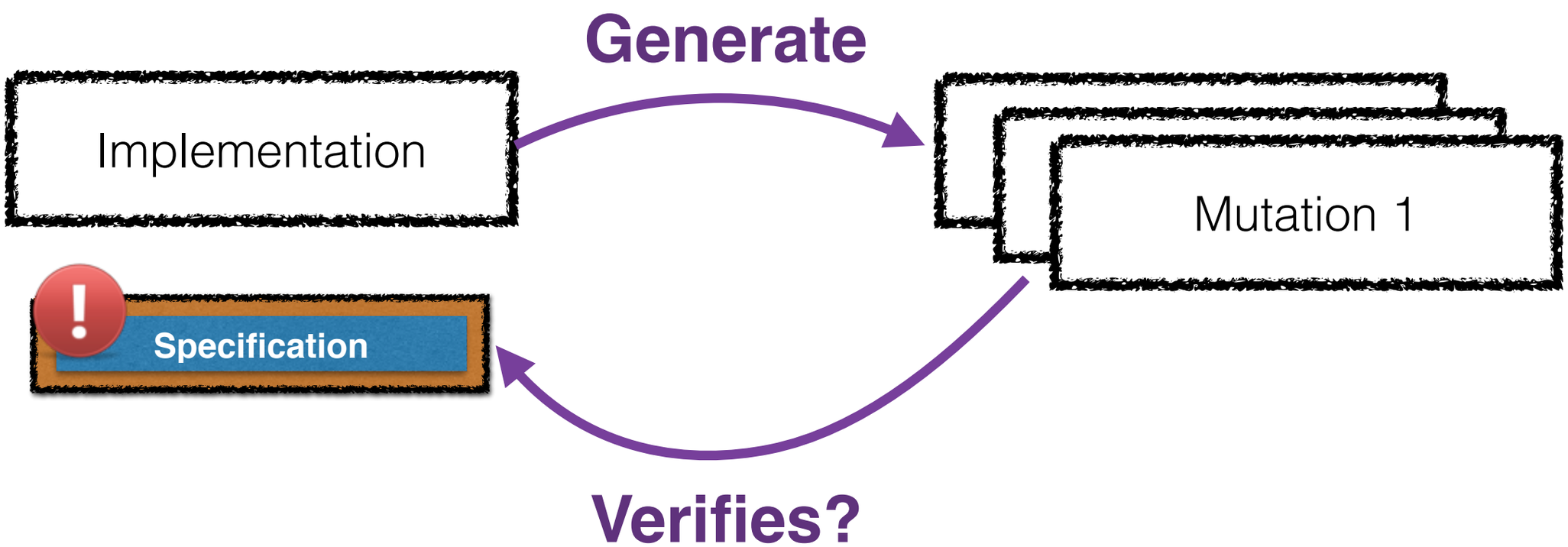


- Fixing: Remove semantics from implementation or Add semantics to specification and verify it



Preventing specification bugs

- Testing for underspecified implementations

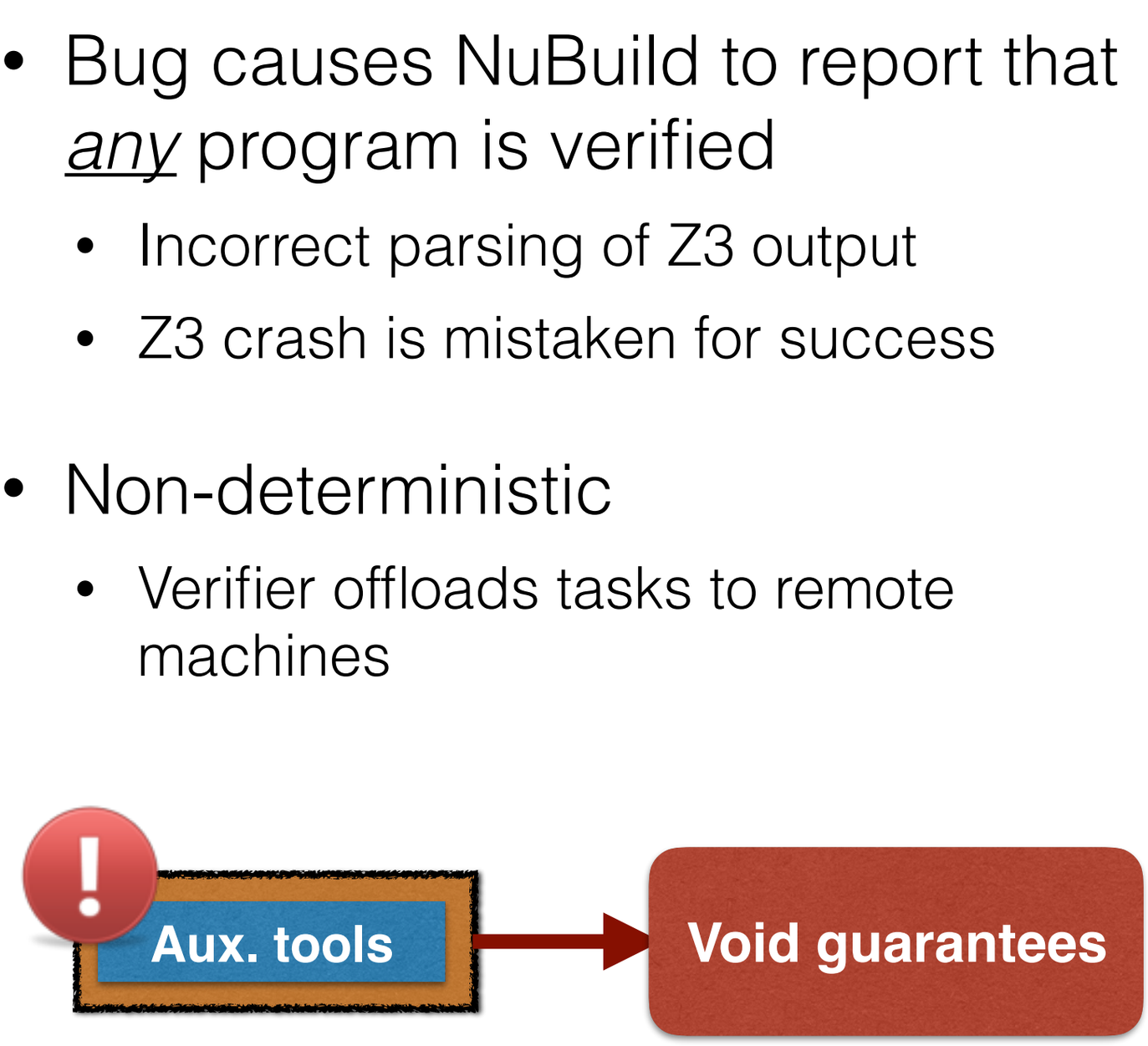


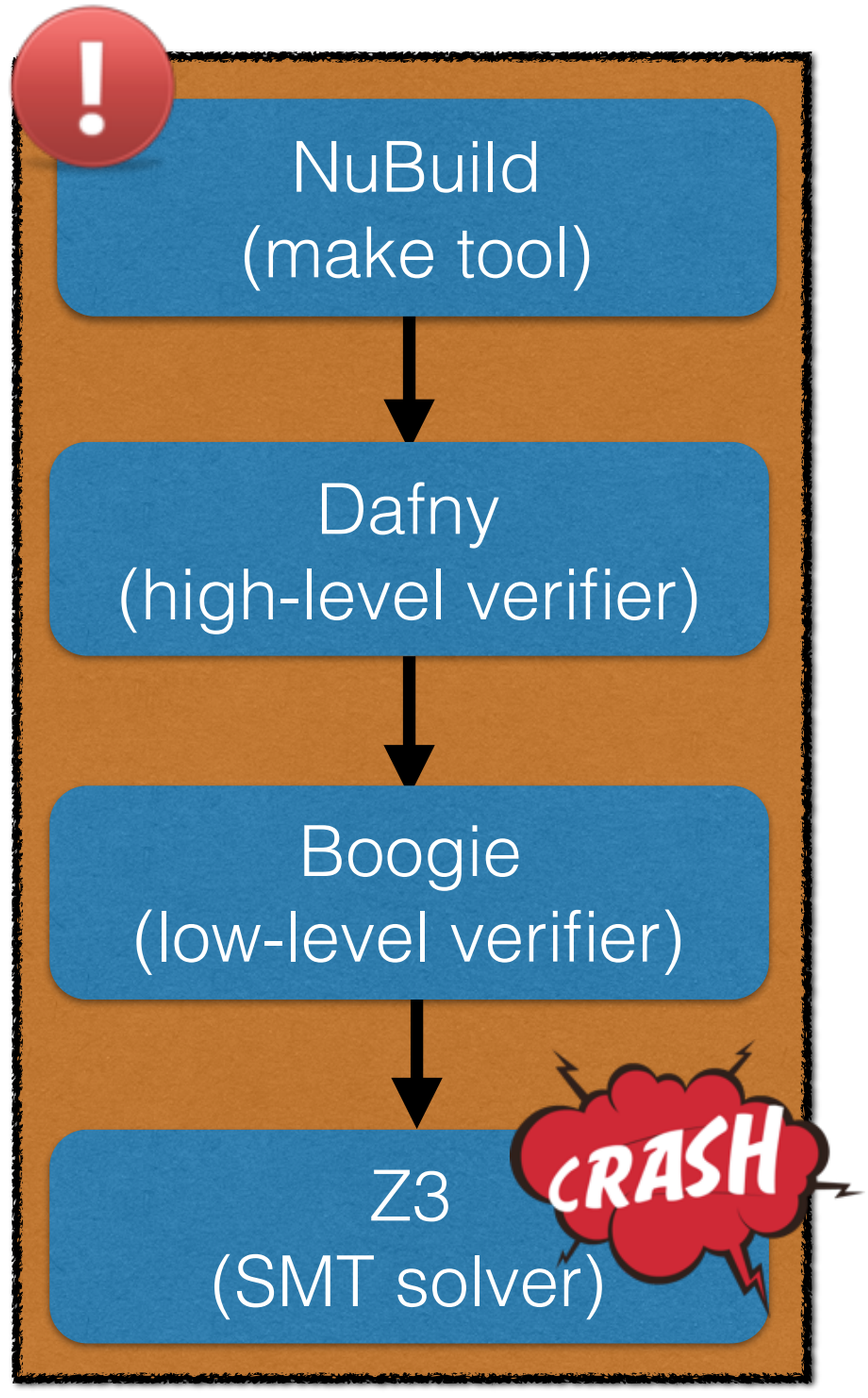
- Proving specification properties

1. Shim layer bugs
2. Specification bugs
3. **Verifier bugs**
4. Towards “bug-free” distributed system

Example #4: Verifier bug

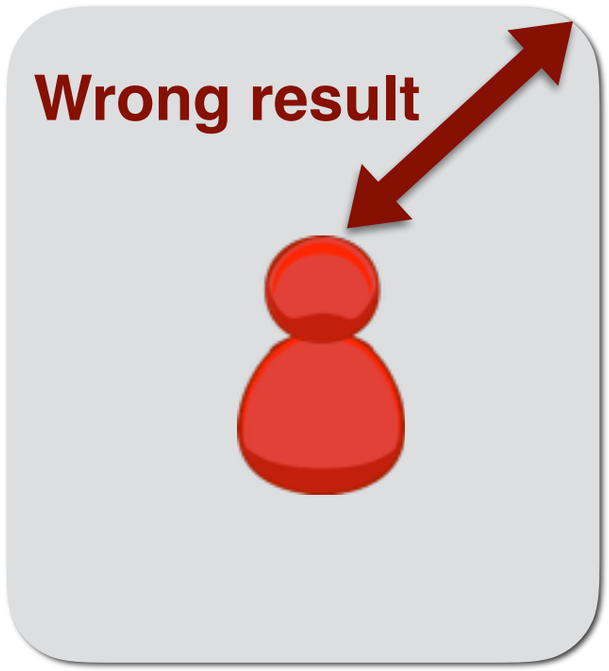
- Bug causes NuBuild to report that *any* program is verified
  - Incorrect parsing of Z3 output
  - Z3 crash is mistaken for success
- Non-deterministic
  - Verifier offloads tasks to remote machines





Preventing verifier bugs

- Construct and apply sanity-checks
  - Detect obvious problems in solvers, offloading, cache
- Design fail-safe verifiers



1 Shim layer bugs

2 Specification bugs

3 Verifier bugs

4 **Towards “bug-free” distributed system**

Existing **real-world deployed** systems

- Analyzed bug reports of *unverified DSs*
  - 1-year span
  - Differences: system size, maturity, etc.

Component	Total
Communication	17
Recovery	8
Logging	21
Protocol	12
Configuration	3
Reconfiguration	42
Management	160
Storage	230
Concurrency	24

Protocol bugs remain a problem

Management and storage have most of the bugs

Conclusion

- Empirical study on verified systems
- No protocol-level bugs found in verified systems
- 16 bugs found suggest **interface** between verified code and the TCB is bug-prone
  - Specification, shim-layer, and auxiliary tools
  - Testing toolchains complement verification