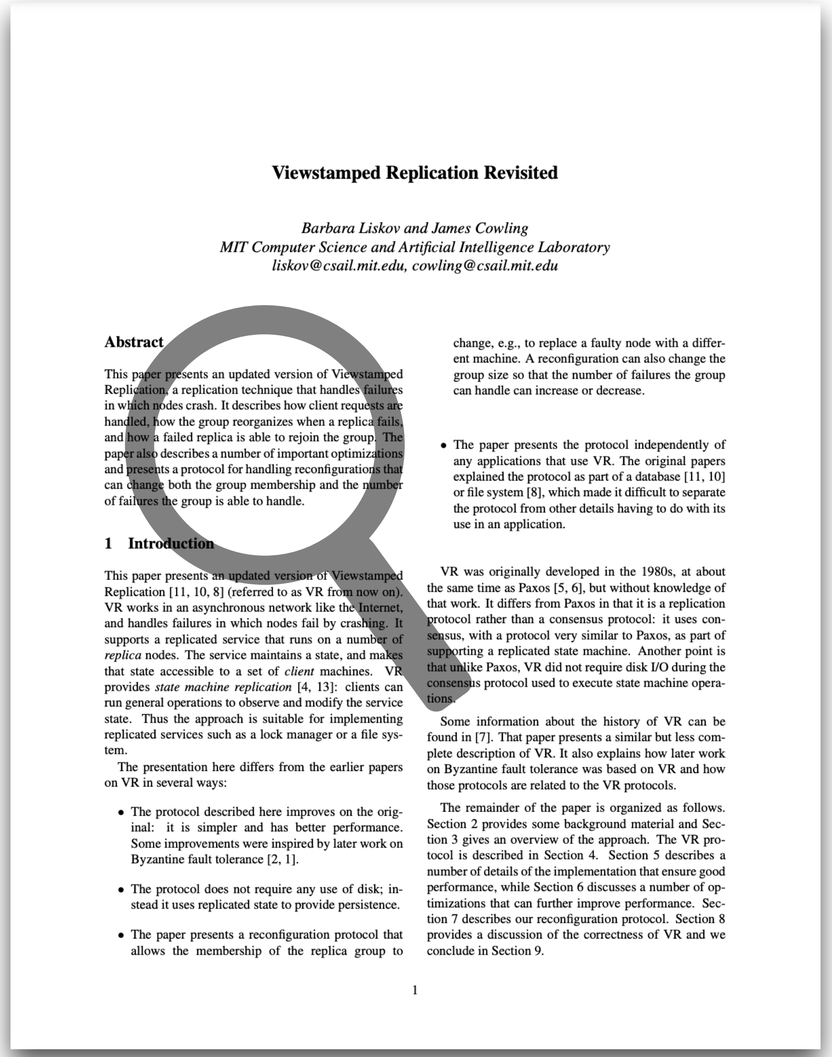
viewstamped replication 代码结构（初步）

蔡之恒 2022年4月



Viewstamped Replication Revisited

Barbara Liskov and James Cowling
MIT Computer Science and Artificial Intelligence Laboratory
liskov@csail.mit.edu, cowling@csail.mit.edu

Abstract

This paper presents an updated version of Viewstamped Replication, a replication technique that handles failures in which nodes crash. It describes how client requests are handled, how the group reorganizes when a replica fails, and how a failed replica is able to rejoin the group. The paper also describes a number of important optimizations and presents a protocol for handling reconfigurations that can change both the group membership and the number of failures the group is able to handle.

1 Introduction

This paper presents an updated version of Viewstamped Replication [11, 10, 8] (referred to as VR from now on). VR works in an asynchronous network like the Internet, and handles failures in which nodes fail by crashing. It supports a replicated service that runs on a number of *replica* nodes. The service maintains a state, and makes that state accessible to a set of *client* machines. VR provides *state machine replication* [4, 13]: clients can run general operations to observe and modify the service state. Thus the approach is suitable for implementing replicated services such as a lock manager or a file system.

The presentation here differs from the earlier papers on VR in several ways:

- The protocol described here improves on the original: it is simpler and has better performance. Some improvements were inspired by later work on Byzantine fault tolerance [2, 1].
- The protocol does not require any use of disk; instead it uses replicated state to provide persistence.
- The paper presents a reconfiguration protocol that allows the membership of the replica group to change, e.g., to replace a faulty node with a different machine. A reconfiguration can also change the group size so that the number of failures the group can handle can increase or decrease.
- The paper presents the protocol independently of any applications that use VR. The original papers explained the protocol as part of a database [11, 10] or file system [8], which made it difficult to separate the protocol from other details having to do with its use in an application.

VR was originally developed in the 1980s, at about the same time as Paxos [5, 6], but without knowledge of that work. It differs from Paxos in that it is a replication protocol rather than a consensus protocol: it uses consensus, with a protocol very similar to Paxos, as part of supporting a replicated state machine. Another point is that unlike Paxos, VR did not require disk I/O during the consensus protocol used to execute state machine operations.

Some information about the history of VR can be found in [7]. That paper presents a similar but less complete description of VR. It also explains how later work on Byzantine fault tolerance was based on VR and how those protocols are related to the VR protocols.

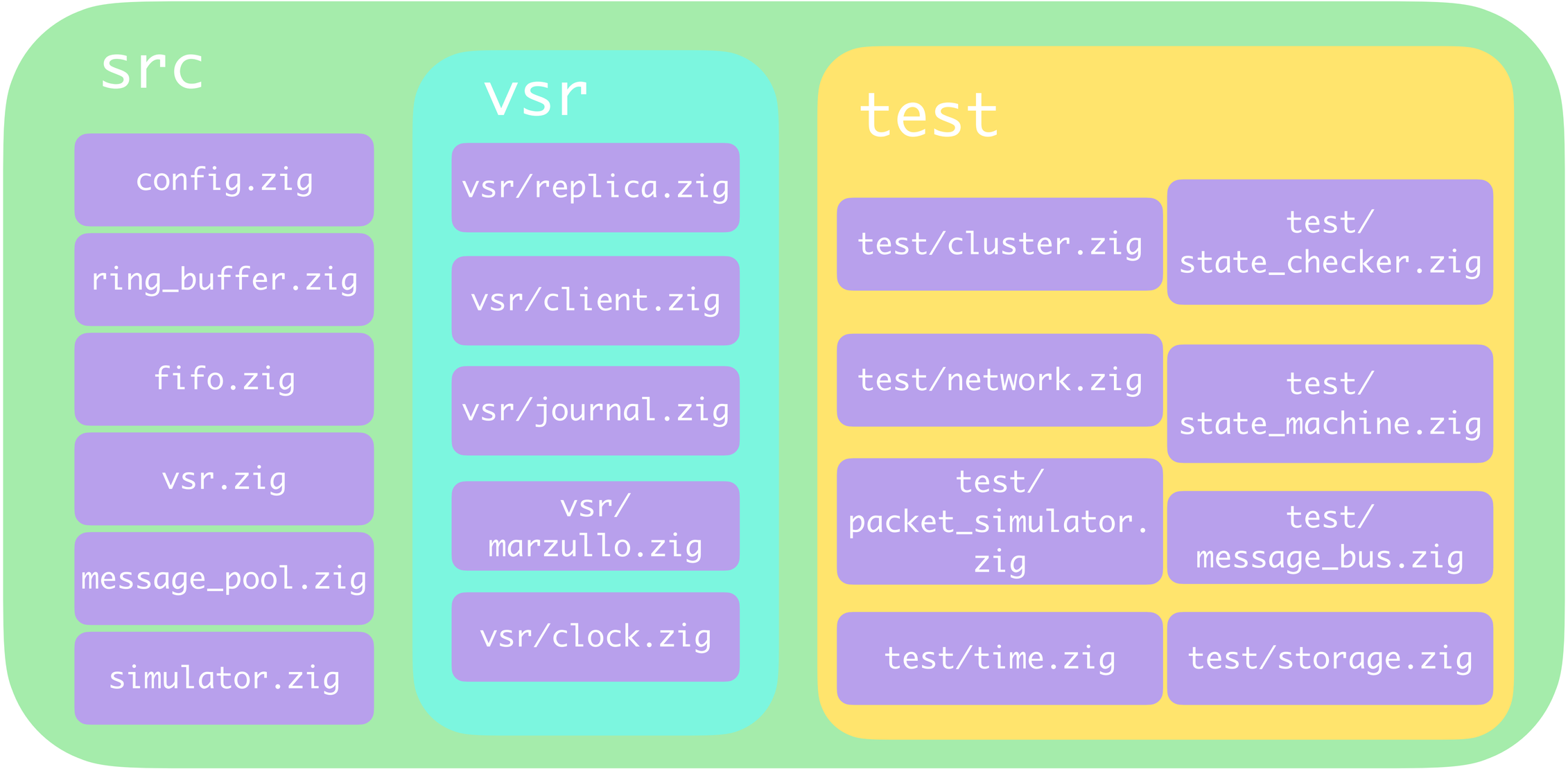
The remainder of the paper is organized as follows. Section 2 provides some background material and Section 3 gives an overview of the approach. The VR protocol is described in Section 4. Section 5 describes a number of details of the implementation that ensure good performance, while Section 6 discusses a number of optimizations that can further improve performance. Section 7 describes our reconfiguration protocol. Section 8 provides a discussion of the correctness of VR and we conclude in Section 9.

1

This paper presents an updated version of Viewstamped Replication, a replication technique that handles failures in which nodes crash. It describes how client requests are handled, how the group reorganizes when a replica fails, and how a failed replica is able to rejoin the group. The paper also describes a number of important optimizations and presents a protocol for handling reconfigurations that can change both the group membership and the number of failures the group is able to handle.


ZIG

2 directories, 19 files



config.zig

数据结构

fifo.zig

ring_buffer.zig

message_pool.zig

协议本体

vsr.zig

vsr

vsr/replica.zig

vsr/client.zig

vsr/journal.zig

vsr/marzullo.zig

vsr/clock.zig

模拟与测试

simulator.zig

test

test/cluster.zig

test/state_checker.zig

test/network.zig

test/state_machine.zig

test/packet_simulator.zig

test/message_bus.zig

test/time.zig

test/storage.zig

config.zig

“配置文件”，绝大多数都是注释

```zig
/// The maximum number of clients allowed per cluster, where each client has a unique 128-bit ID.
/// This impacts the amount of memory allocated at initialization by the server.
/// This determines the size of the VR client table used to cache replies to clients by client ID.
/// Each client has one entry in the VR client table to store the latest `message_size_max` reply.
pub const clients_max = 3;

/// The minimum number of nodes required to form a quorum for replication:
/// Majority quorums are only required across view change and replication phases (not within).
/// As per Flexible Paxos, provided `quorum_replication + quorum_view_change > replicas`:
/// 1. you may increase `quorum_view_change` above a majority, so that
/// 2. you can decrease `quorum_replication` below a majority, to optimize the common case.
/// This improves latency by reducing the number of nodes required for synchronous replication.
/// This reduces redundancy only in the short term, asynchronous replication will still continue.
/// The size of the replication quorum is limited to the minimum of this value and actual majority.
/// The size of the view change quorum will then be automatically inferred from quorum_replication.
pub const quorum_replication_max = 3;
```

数据结构

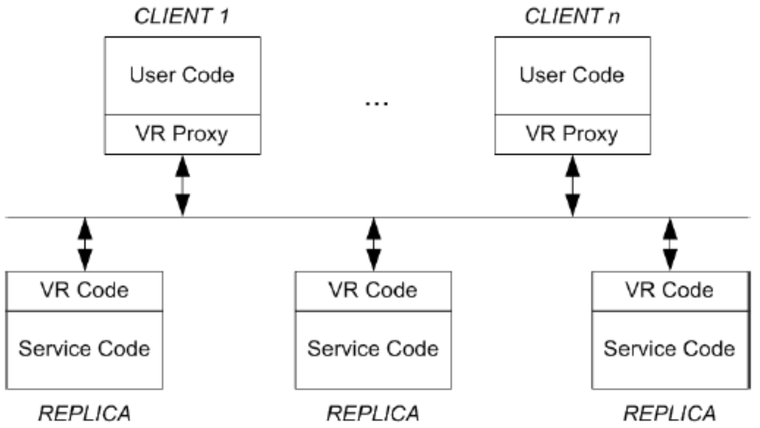
- fifo.zig ： 普通的队列
- ring_buffer.zig ： 有最大容量的循环队列
- message_pool.zig ?: 定义了 Message；储存 Message 的普通链表。

```
61  /// A pool of reference-counted Messages, memory for which is allocated only once during
62  /// initialization and reused thereafter. The messages_max values determine the size of this pool.
63  pub const MessagePool = struct {
64  ›       pub const Message = struct {…
82          };
83
84          /// List of currently unused messages of message_size_max_padded
85          free_list: ?*Message,
```

协议本体(vsr/*.zig)

面向对象?

- client.zig : client 类
- replica.zig : replica 类
- clock.zig : 机器时钟, 依赖 marzullo.zig
- marzullo.zig : Marzullo 算法, 维护时钟
- journal.zig : 写到硬盘中的日志



协议本体(vsr.zig)

- “整合”了 vsr/*.zig 的内容

```zig
pub const Replica = @import("vsr/replica.zig").Replica;
pub const Client = @import("vsr/client.zig").Client;
pub const Clock = @import("vsr/clock.zig").Clock;
pub const Journal = @import("vsr/journal.zig").Journal;
```

- 定义了所有的操作
- 定义了 Header 类和 Timeout 类

```zig
/// Network message and journal entry header:
/// We reuse the same header for both so that prepare messages from the leader can simply be
/// journalled as is by the followers without requiring any further modification.
/// TODO Move from packed struct to extern struct for C ABI:
pub const Header = packed struct {…
};
```

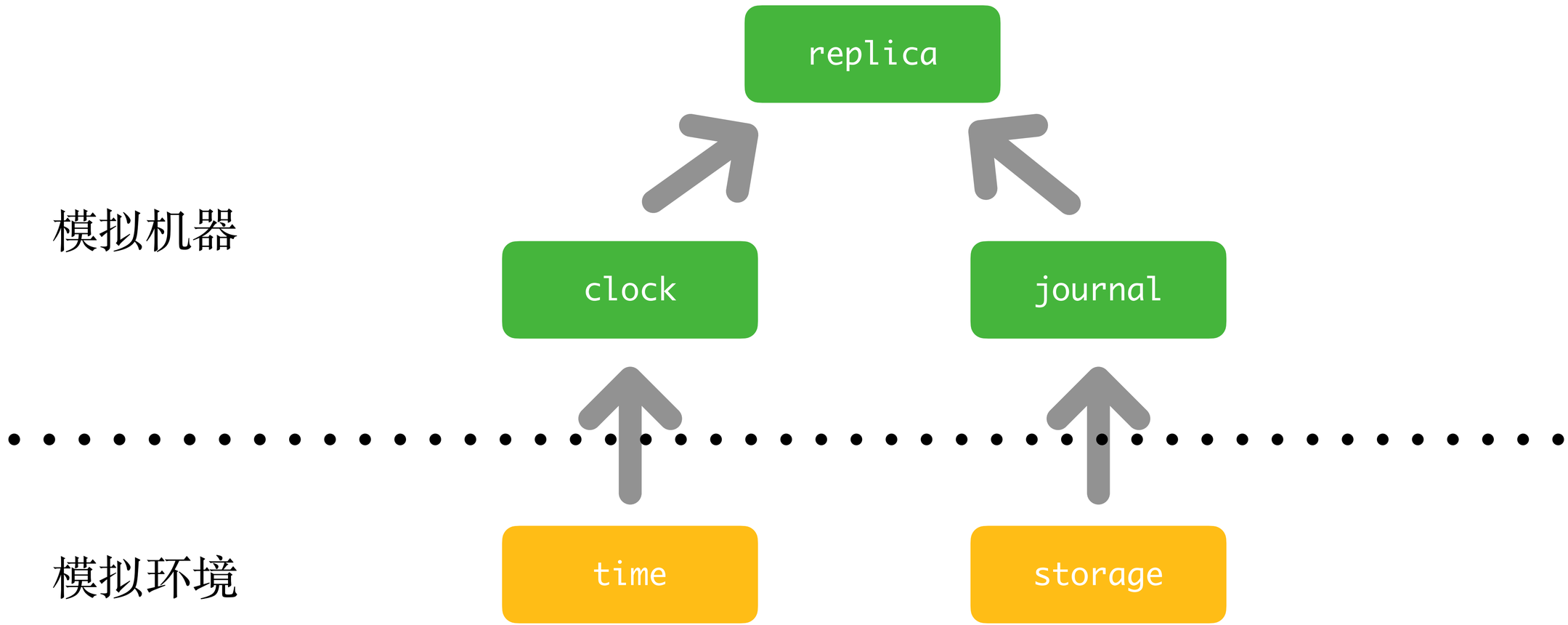
- ……

```zig
/// Viewstamped Replication protocol commands:
pub const Command = enum(u8) {
    reserved,

    ping,
    pong,

    request,
    prepare,
    prepare_ok,
    reply,
    commit,

    start_view_change,
    do_view_change,
    start_view,

    recovery,
    recovery_response,

    request_start_view,
    request_headers,
    request_prepare,
    headers,
    nack_prepare,

    eviction,
};
```

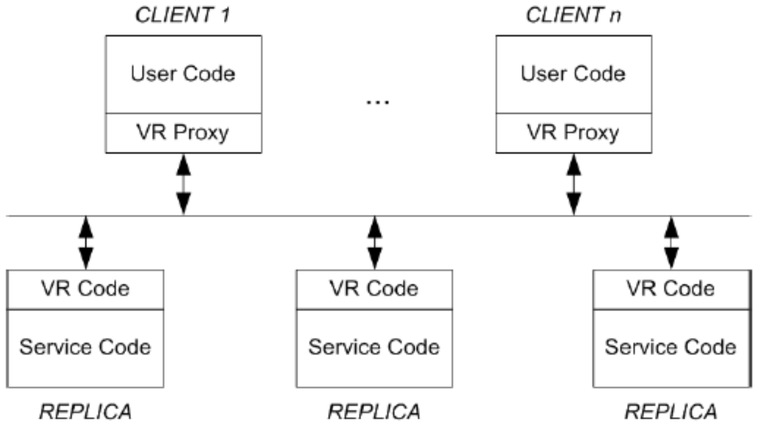
模拟与测试(test/*.zig)

- packet_simulator.zig : 模拟传输过程中的包
- network.zig : 模拟网络
- message_bus.zig : 模拟消息（message）传输的过程
- storage.zig : 模拟磁盘
- time.zig : 模拟真正的时间
- cluster.zig : cluster 类 replicas, clients, storage, time, …
- state_machine.zig : （单个结点）状态机建模
- state_checker.zig : 全局状态检查器

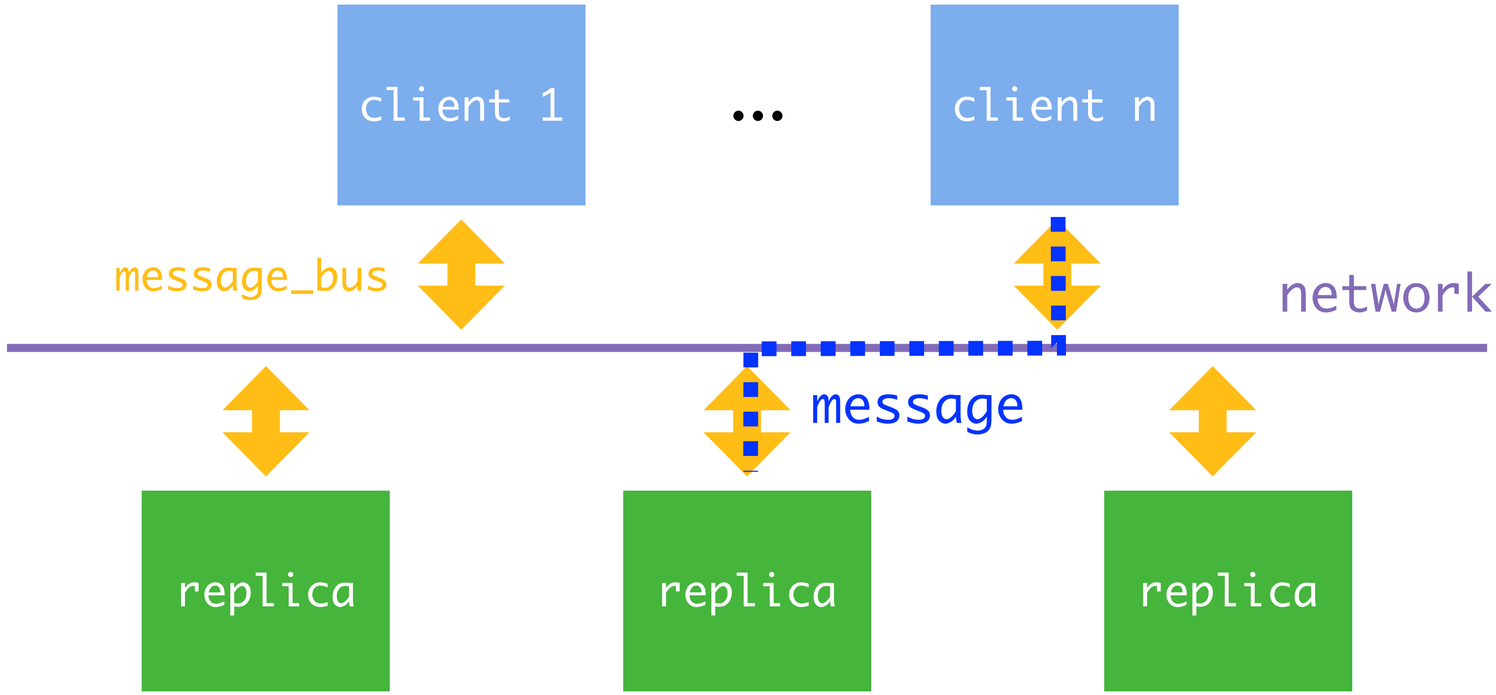
replica



cluster

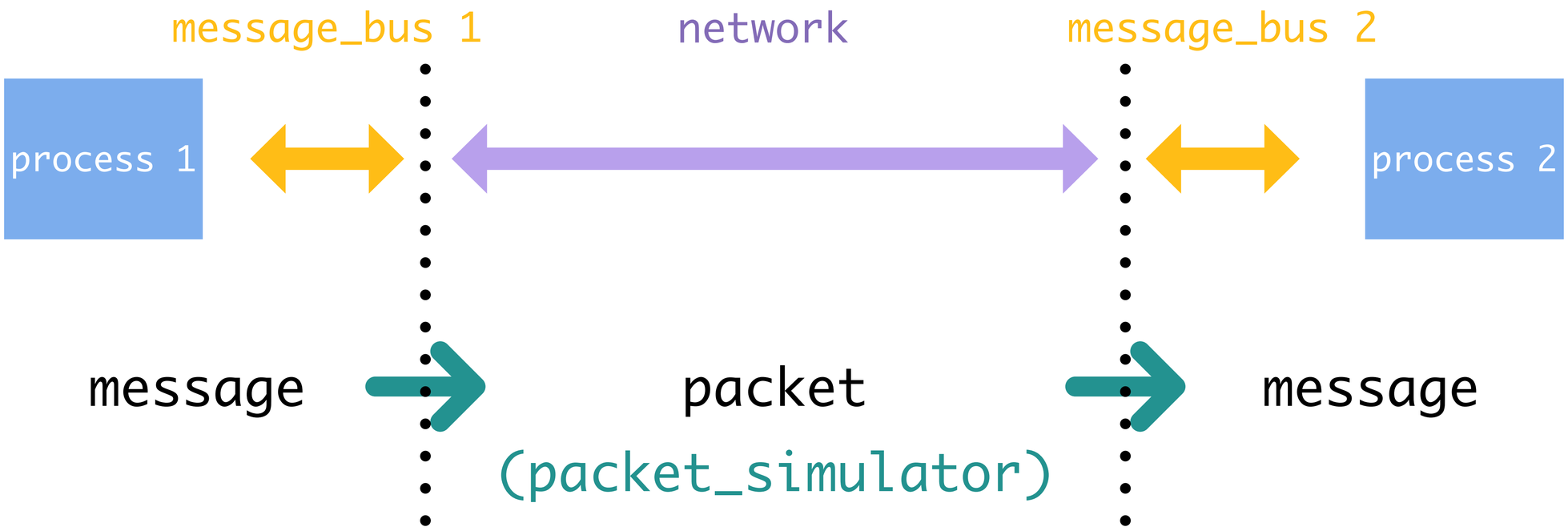


简约线条

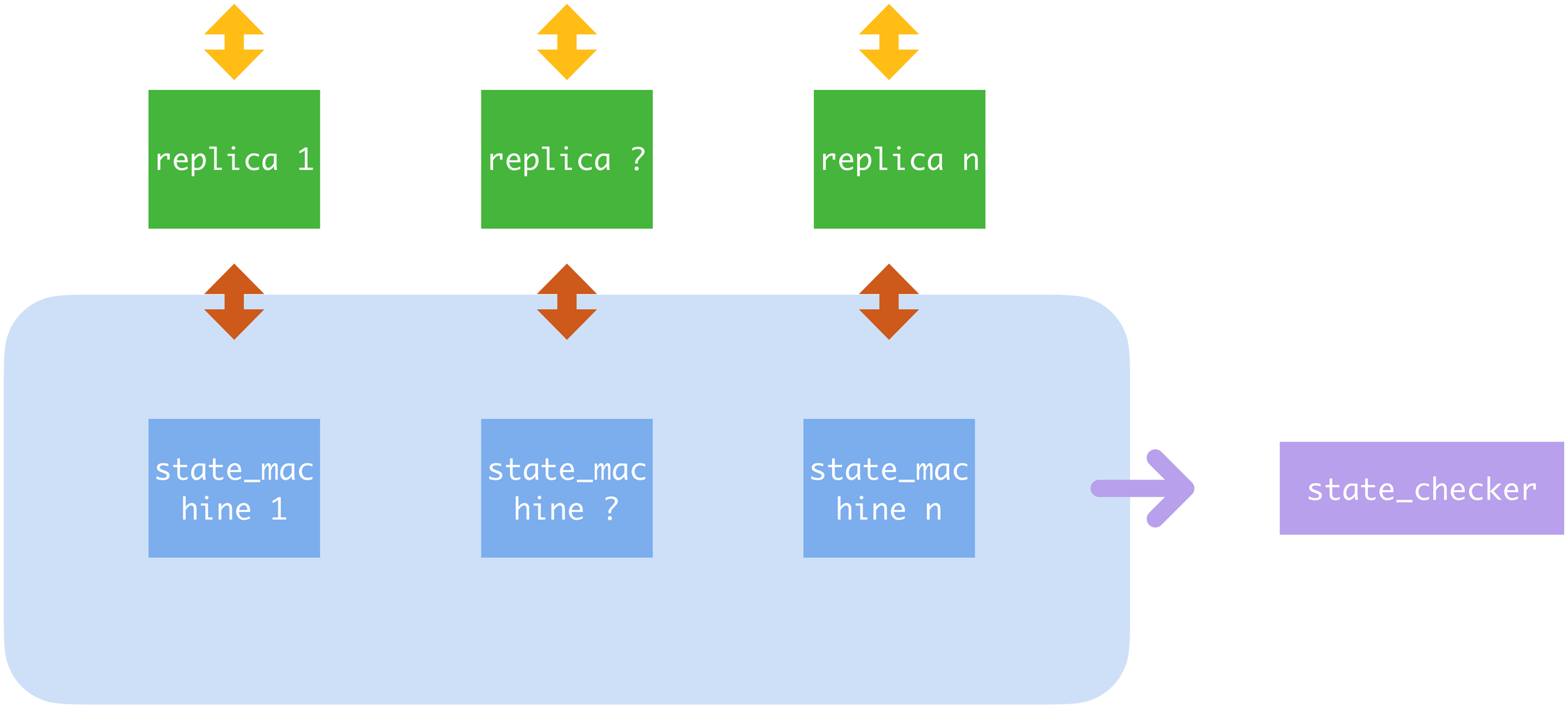


极致色彩

path


message_bus 1
network
message_bus 2
process 1
process 2
message
packet
message
(packet_simulator)

cluster


replica 1
replica ?
replica n
state_machine 1
state_machine ?
state_machine n
state_checker

模拟与测试(simulator.zig)

初始化 & 随机数生成 & main loop

- main 函数

```zig
pub fn main() !void {
    // TODO Use std.testing.allocator when all deinit() leaks are fixed.
    const allocator = std.heap.page_allocator;

    var args = std.process.args();

    // Skip argv[0] which is the name of this executable:
    _ = args_next(&args, allocator);

    const seed_random = std.crypto.random.int(u64);
    const seed = seed_from_arg: {
        const arg_two = args_next(&args, allocator) orelse break :seed_from_arg seed_random;
        defer allocator.free(arg_two);
        break :seed_from_arg parse_seed(arg_two);
    };
```

模拟与测试(simulator.zig)

随机数生成 & main loop

- main loop

```
var requests_sent: u64 = 0;
var idle = false;

var tick: u64 = 0;
while (tick < ticks_max) : (tick += 1) {
    for (cluster.storages) |*storage| storage.tick();

    for (cluster.replicas) |*replica, i| {
        replica.tick();
        cluster.state_checker.check_state(@intCast(u8, i));
    }

    cluster.network.packet_simulator.tick();

    for (cluster.clients) |*client| client.tick();

    if (cluster.state_checker.transitions == transitions_max) {
        if (cluster.state_checker.convergence()) break;
        continue;
    } else {
        assert(cluster.state_checker.transitions < transitions_max);
    }

    if (requests_sent < transitions_max) {
        if (idle) {
            if (chance(random, idle_off_probability)) idle = false;
        } else {
            if (chance(random, request_probability)) {
                if (send_request(random)) requests_sent += 1;
            }
            if (chance(random, idle_on_probability)) idle = true;
        }
    }
}
```

感受和总结

- 希望在充分理解协议之后看懂代码中的所有细节和作者划分职责的思路。
- 希望能够看懂作者（在注释中）提到的部分优化并尝试证明 / 推翻。
- zig 语言确实有其优越性。