Actor Systems in Rust:
Techniques and Tradeoffs

Andrew J. Stone

vmware®

Haret

Distributed Coordinator and KV Store

Haret

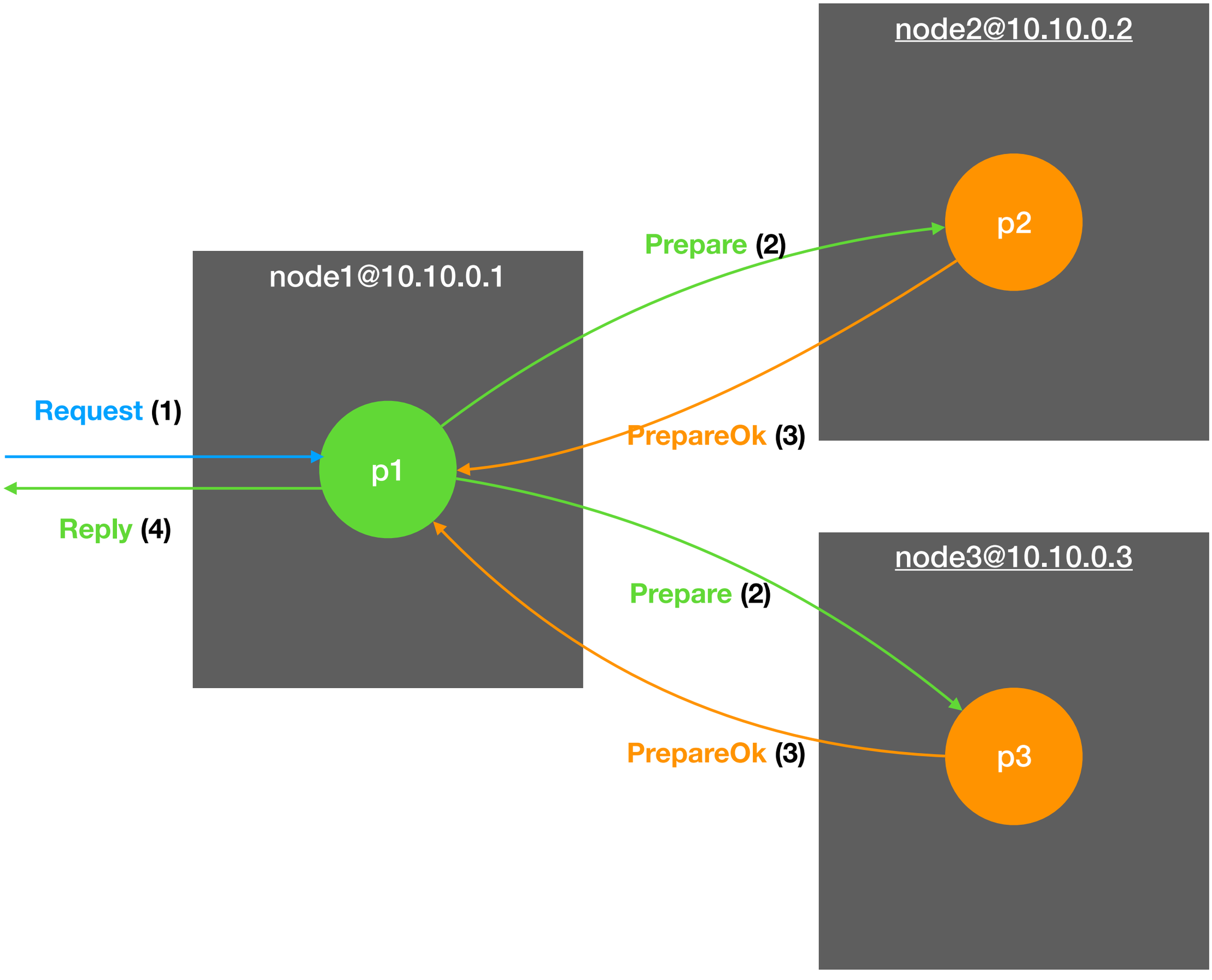
Multiple Independent Consensus Groups

Haret Protocol

Viewstamped Replication Revisited for Consensus

Haret Backend

- Versioned persistent trie per consensus group
  - CAS on entire subtrees
  - Typed Leaves
    - Blob
    - Queue
    - Set


node2@10.10.0.2
p2
Prepare (2)
node1@10.10.0.1
Request (1)
p1
PrepareOk (3)
Reply (4)
node3@10.10.0.3
Prepare (2)
p3
PrepareOk (3)

Actors

Processes That Communicate Only via Asynchronous Msg Passing

Actors

Straightforward Representation of Peers in a Consensus Group

Rabble

Actor Abstractions for Implementing Distributed Algorithms

Rabble

Dynamic Cluster Membership, Transport and Msg Routing

Rabble

Designed for Testability

Rabble

A Work in Progress

Rabble

Provides a clean way to implement VRR in Haret

Naming

```
pub struct Pid {
    pub group: Option<String>,
    pub name: String,
    pub node: NodeId,
}
```

Messages

```
pub enum Msg<T> {
    User(T),
    ClusterStatus(ClusterStatus),
    ExecutorStatus(ExecutorStatus),
    StartTimer(usize), // time in ms
    CancelTimer(Option<CorrelationId>),
    Timeout,
    Shutdown,
    GetMetrics,
    Metrics(Vec<(Name, Metric)>)
}
```

User Defined → `User(T),`

Rabble built-ins → `ClusterStatus` … `Metrics`

Envelopes

```
pub struct Envelope<T> {
    pub to: Pid,
    pub from: Pid,
    pub msg: Msg<T>,
    pub correlation_id: Option<CorrelationId>
}
```

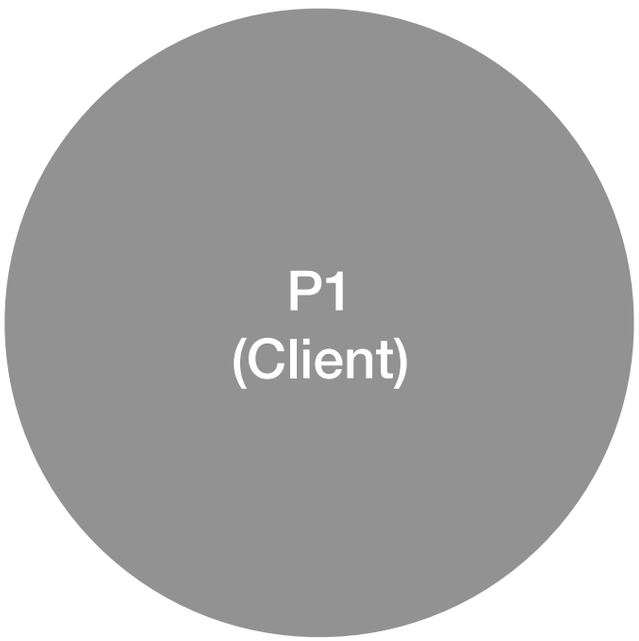
Processes

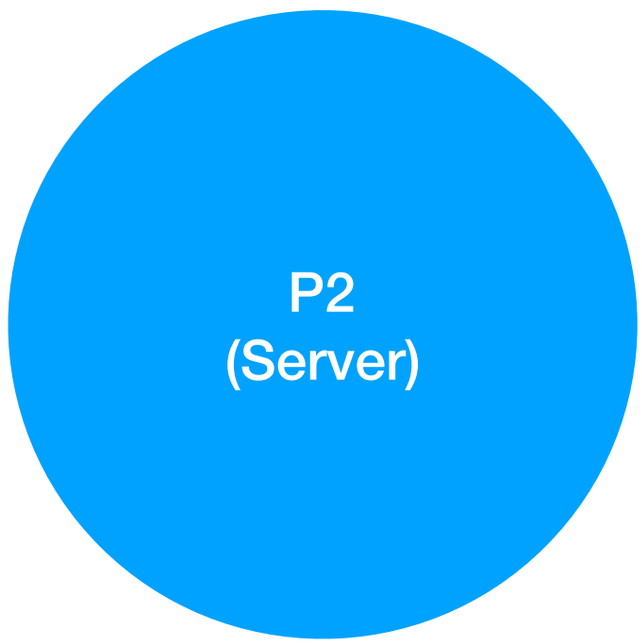
```
pub trait Process<T> : Send {

    /// Handle messages from other actors
    fn handle(&mut self,
              msg: Msg<T>,
              from: Pid,
              correlation_id: Option<CorrelationId>,
              output: &mut Vec<Envelope<T>>);
}
```

Echo Server

```
Envelope {
    to: P2,
    from: P1,
    msg: Msg::User<Hello>,
    correlation_id: Some(P1)
}
```






```
Envelope {
    to: P1,
    from: P2,
    msg: Msg::User<Hello>,
    correlation_id: Some(P1)
}
```

```rust
impl Process<T> for EchoServer<T> {
    fn handle(&mut self,
              msg: Msg<T>,
              from: Pid,
              cid: Option<CorrelationId>,
              output: &mut Vec<Envelope<T>>)
    {
        match msg {
            Msg::User(Hello) => {
                let to = from; // P1
                let from = self.pid.clone(); // P2
                let msg = Msg::User(Hello);
                let reply = Envelope::new(to, from, msg, cid);
                output.push(reply);
            },
            _ => ()
        }
    }
```

Executor

Locates and Runs Processes

Thread

Executor Channel

Hashmap<Pid, Process>

P1	P2	P3

To: P1

Handle

Property Based Testing

1. Construct a group of processes in an initial state
2. Generate a schedule of test messages
3. Call the handle method of a process with a test message
4. Collect output messages of handle method
5. Schedule output messages

Test Scheduling

Trigger Protocol State Transitions with Explicit Timeouts

Test Assertions

- Pre/Post conditions
- Global state invariants

Failing Tests

Failing Schedules Run as a Regression Suite

Debugging

Each Message Receipt is One Step in a Test Schedule

Cluster Server

Non-blocking Single Threaded Server with TCP Connections to every node

Membership

Dynamic Membership Maintained in an Observed-Remove Set

Services

Actors that Run Blocking and CPU Intensive Operations

Takeaways

Takeaway

Dynamic Messaging is Complex Expensive, and Unergonomic for Distributed Actors in Rust

Takeaway

Writing Good Schedulers is Hard

Takeaway

Design your systems to make deterministic testing easier

Takeaway

Compromise and Pragmatism allow a good enough solution that works NOW

Links

- https://github.com/andrewjstone/rabble
- https://github.com/andrewjstone/orset
- https://github.com/vmware/haret

Thanks!