How (Not) to Build Scalable Applications in ONOS

Jordan Halterman

Member of Technical Staff @ ONF

ONF CONNECT

Distributed Applications

```
@Reference(cardinality = ReferenceCardinality.MANDATORY)
private FlowRuleService flowRuleService;

@Activate
public void activate() {
   FlowRuleOperations.Builder ops = FlowRuleOperations.builder();
   // ...
   flowRuleService.apply(ops.build());
}
```

Distributed Applications

```java
private ConsistentMultimap<IpPrefix, Route> routes;

@Activate
public void activate() {
   routes = storageService.<IpPrefix, Route>consistentMultimapBuilder()
            .withName("routes")
            .withSerializer(Serializer.using(KryoNamespaces.API))
            .build();
}

public void addRoute(IpPrefix prefix, Route route) {
   routes.put(prefix, route);
}
```

Distributed Applications

- Distributed primitives designed to make distributed state management accessible to application developers
- But many of the pitfalls of distributed systems engineering are unavoidable no matter how simple the interface
- Time still doesn’t progress at the same rate on all nodes
- The network still introduces significant latency
- Some problems may even be make more likely and more difficult to detect by using distributed primitives

Distributed Applications

- ONOS and Trellis teams have worked on trial deployments over the last year
- Scale testing and production deployment has exposed deficiencies
- Most scalability problems produced by distributed state management
- Most deficiencies correctable by good development practices

onf

Six Lessons

ONF

Lesson #1

onf

DO NOT POLL PRIMITIVES

onf

Do Not Poll Primitives

- A single read requires 1 to 2 round trip time depending on consistency models
- ONOS features a rich event framework
- Rely on event delivery for updates
- Events from consistent primitives delivered in sequential order

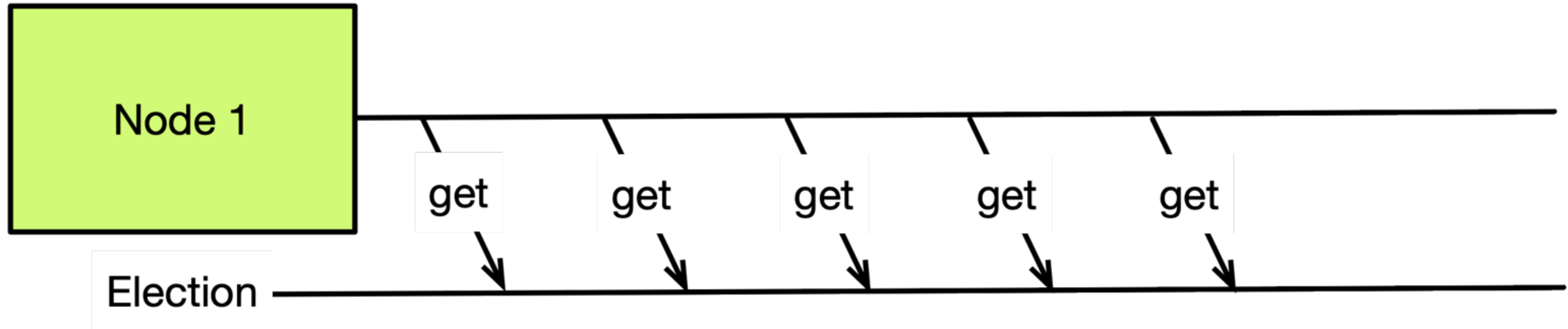
onf

Do Not Poll Primitives

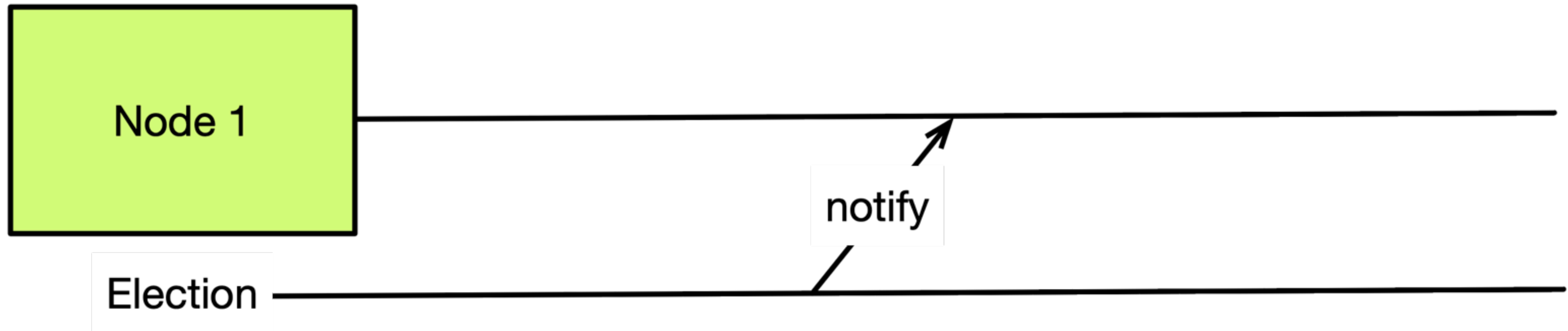
```
LeaderElector leaderElector = storageService.leaderElectorBuilder()
        .withName("election")
        .withElectionTimeout(5, TimeUnit.SECONDS)
        .build()
        .asLeaderElector();

Leadership leadership = leaderElector.getLeadership(topic);
while (leadership == null) {
   Thread.sleep(100);
   leadership = leaderElector.getLeadership(topic);
}

handleLeaderChange(leadership);
```


ONF

Do Not Poll Primitives



Do Not Poll Primitives



Do Not Poll Primitives

```
LeaderElector leaderElector = storageService.leaderElectorBuilder()
        .withName("election")
        .withElectionTimeout(5, TimeUnit.SECONDS)
        .build()
        .asLeaderElector();

leaderElector.addChangeListener(change -> {
   Leadership leadership = change.newValue();
   handleLeaderChange(leadership);
});
```

Lesson #2

DO NOT BLOCK THE EVENT LOOP

Do Not Block the Event Loop

- Event loops are generally single threaded
- Assume synchronous service and primitive methods are blocking
- A blocked event loop prevents additional events from being delivered
- A blocked event loop can prevent synchronous calls from completing
- Core event dispatcher will interrupt threads to unblock the event loop, potentially resulting in lost events

Do Not Block the Event Loop

```java
@Activate
public void activate() throws Exception {
    ConsistentMap<NodeId, MastershipRole> mastershipRoles =
        storageService.<NodeId, MastershipRole>consistentMapBuilder()
            .withName("mastership-roles")
            .withSerializer(Serializer.using(KryoNamespaces.API))
            .build();

    mastershipService.addListener(event -> {
        MastershipInfo mastershipInfo = event.mastershipInfo();
        mastershipRoles.put(mastershipInfo.master().get(), MastershipRole.MASTER);
        mastershipInfo.backups()
            .forEach(nodeId -> mastershipRoles.put(nodeId, MastershipRole.STANDBY));
    });
}
```

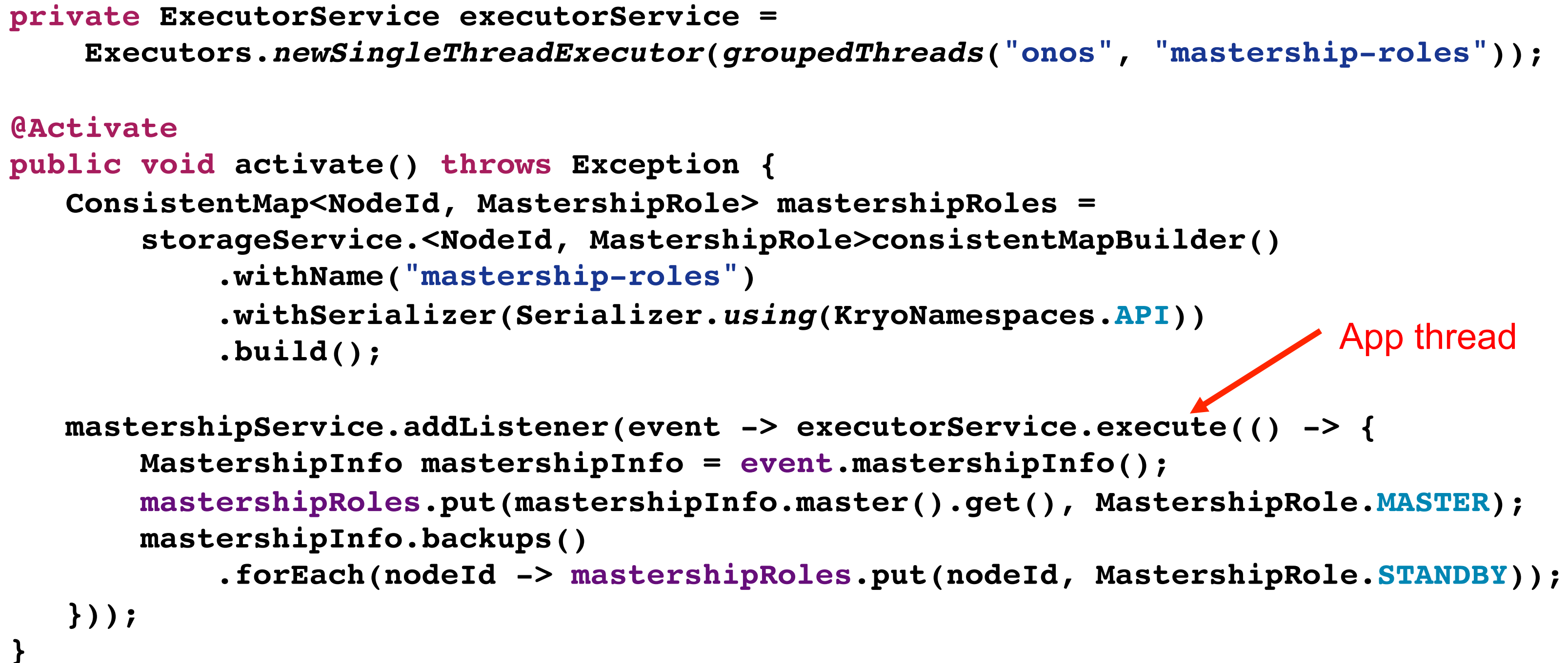
Do Not Block the Event Loop

```
@Activate
public void activate() throws Exception {
    ConsistentMap<NodeId, MastershipRole> mastershipRoles =
        storageService.<NodeId, MastershipRole>consistentMapBuilder()
            .withName("mastership-roles")
            .withSerializer(Serializer.using(KryoNamespaces.API))
            .build();

    mastershipService.addListener(event -> {
        MastershipInfo mastershipInfo = event.mastershipInfo();
        mastershipRoles.put(mastershipInfo.master().get(), MastershipRole.MASTER);
        mastershipInfo.backups()
            .forEach(nodeId -> mastershipRoles.put(nodeId, MastershipRole.STANDBY));
    });
}
```

Do Not Block the Event Loop

```
private ExecutorService executorService =
     Executors.newSingleThreadExecutor(groupedThreads("onos", "mastership-roles"));

@Activate
public void activate() throws Exception {
   ConsistentMap<NodeId, MastershipRole> mastershipRoles =
       storageService.<NodeId, MastershipRole>consistentMapBuilder()
           .withName("mastership-roles")
           .withSerializer(Serializer.using(KryoNamespaces.API))
           .build();

   mastershipService.addListener(event -> executorService.execute(() -> {
       MastershipInfo mastershipInfo = event.mastershipInfo();
       mastershipRoles.put(mastershipInfo.master().get(), MastershipRole.MASTER);
       mastershipInfo.backups()
           .forEach(nodeId -> mastershipRoles.put(nodeId, MastershipRole.STANDBY));
   }));
}
```

Do Not Block the Event Loop



Do Not Block the Event Loop

```
@Activate
public void activate() throws Exception {
   AsyncConsistentMap<NodeId, MastershipRole> mastershipRoles =
       storageService.<NodeId, MastershipRole>consistentMapBuilder()
           .withName("mastership-roles")
           .withSerializer(Serializer.using(KryoNamespaces.API))
           .buildAsyncMap();

   mastershipService.addListener(event -> {
       MastershipInfo mastershipInfo = event.mastershipInfo();
       mastershipRoles.put(mastershipInfo.master().get(), MastershipRole.MASTER);
       mastershipInfo.backups()
           .forEach(nodeId -> mastershipRoles.put(nodeId, MastershipRole.STANDBY));
   });
}
```


onf

Do Not Block the Event Loop

```java
@Activate
public void activate() throws Exception {
    AsyncConsistentMap<NodeId, MastershipRole> mastershipRoles =
        storageService.<NodeId, MastershipRole>consistentMapBuilder()
            .withName("mastership-roles")
            .withSerializer(Serializer.using(KryoNamespaces.API))
            .buildAsyncMap();

    mastershipService.addListener(event -> {
        MastershipInfo mastershipInfo = event.mastershipInfo();
        mastershipRoles.put(mastershipInfo.master().get(), MastershipRole.MASTER);
        mastershipInfo.backups()
            .forEach(nodeId -> mastershipRoles.put(nodeId, MastershipRole.STANDBY));
    });
}
```

Lesson #3

AVOID HOTSPOTTING

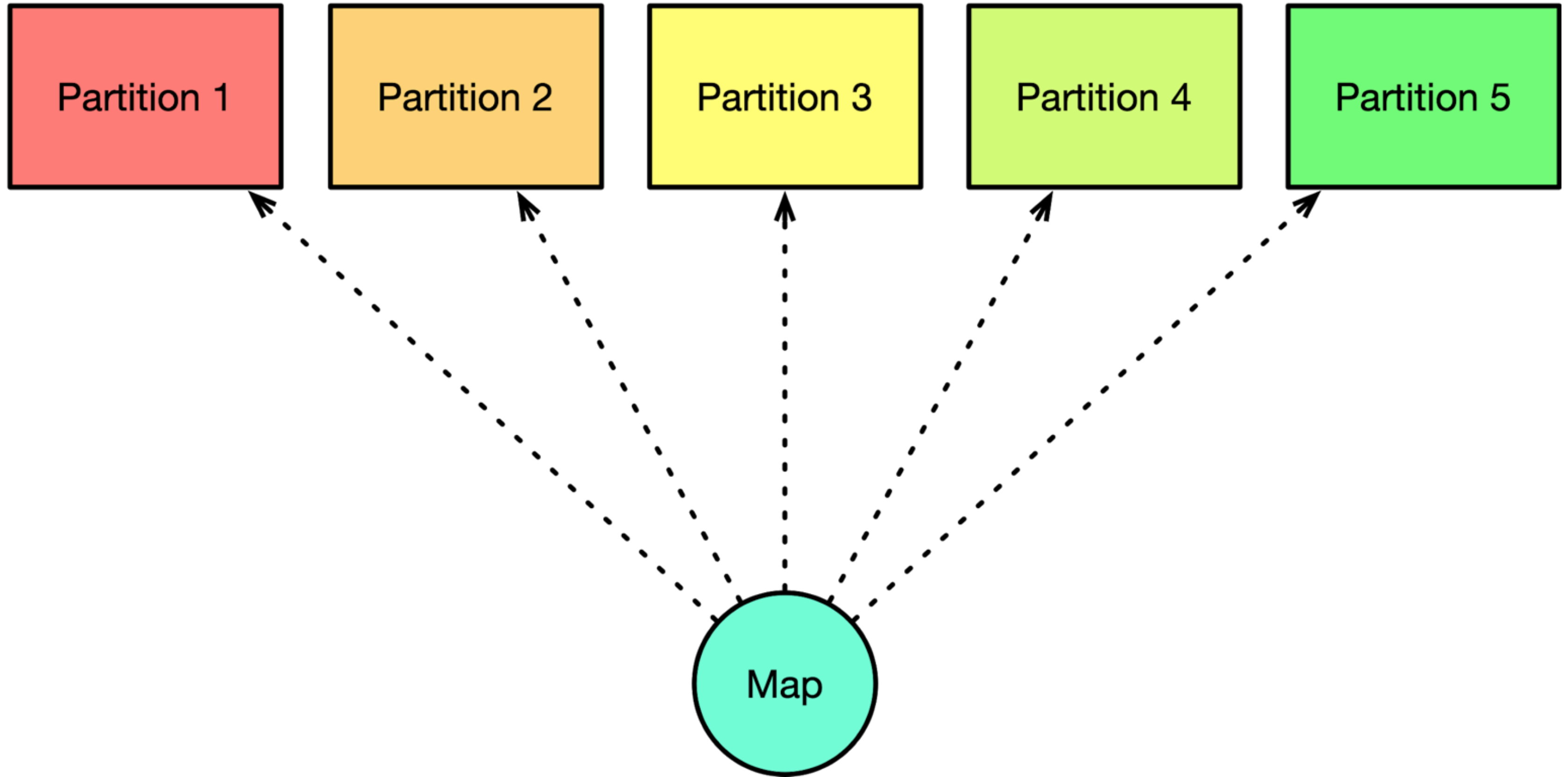
Avoid Hotspotting

- Partitioning is the primary mechanism for achieving scalability in ONOS
- Not all primitives can be partitioned
- Global total order usually relies on a single partition
- Limit or optimize usage of primitives that cannot be partitioned

onf

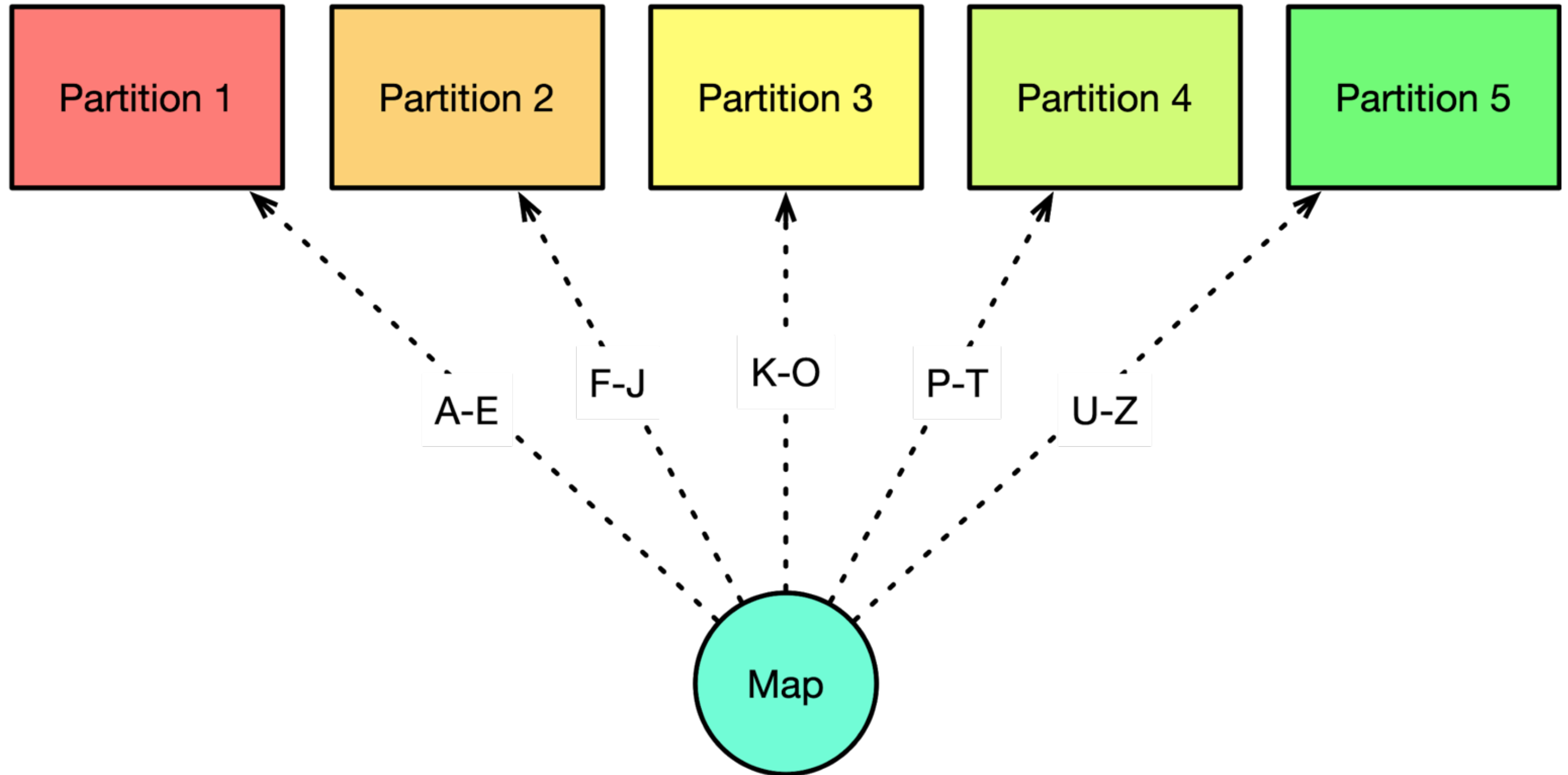
Avoid Hotspotting

Avoid Hotspotting

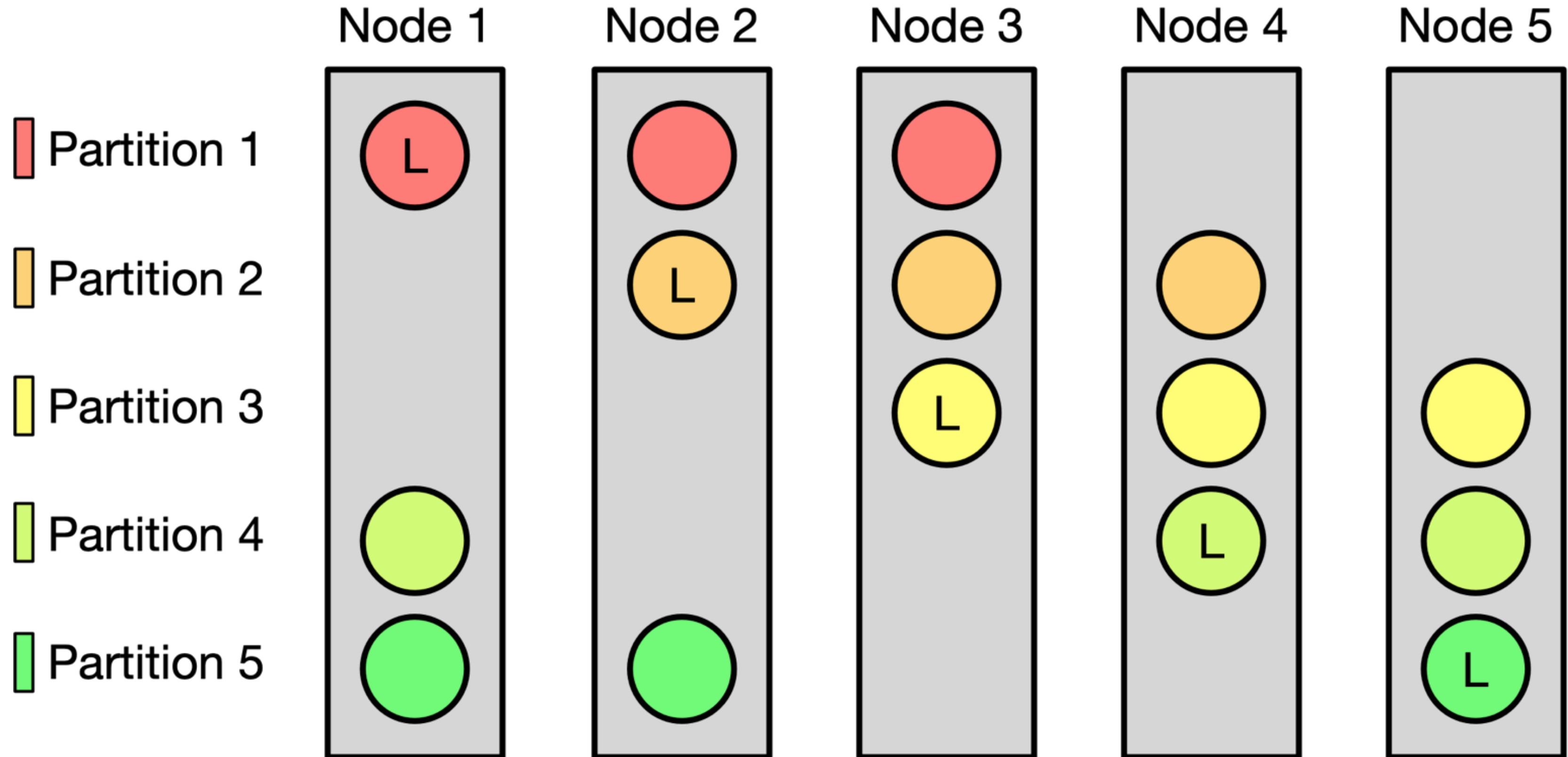


onf

Avoid Hotspotting



Avoid Hotspotting



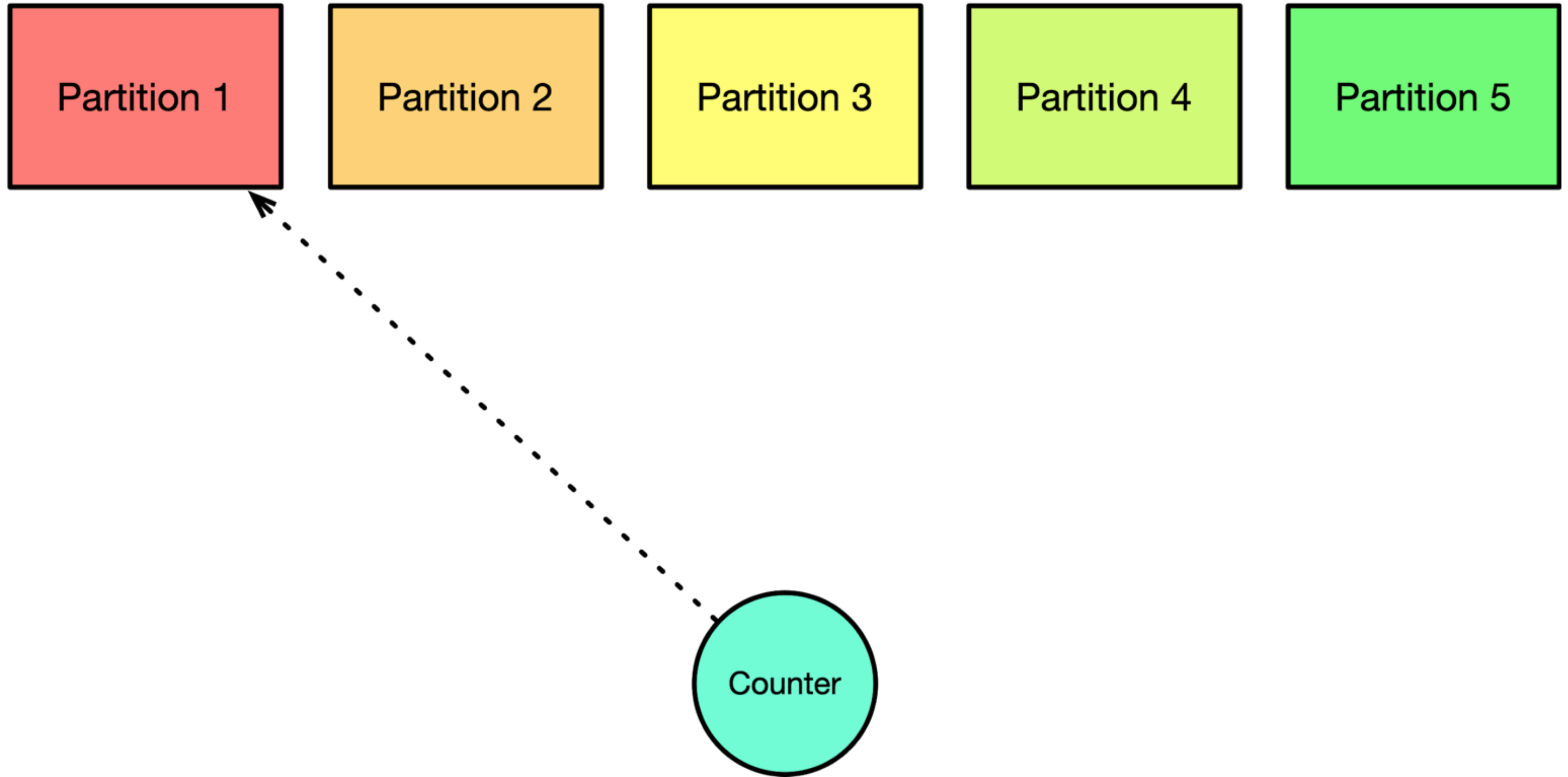
NOT ALL PRIMITIVES ARE HORIZONTALLY SCALABLE

Avoid Hotspotting

```
AtomicCounter counter = storageService.atomicCounterBuilder()
        .withName("uuid")
        .build()
        .asAtomicCounter();

long uuid = counter.incrementAndGet();
```


ONF

Avoid Hotspotting



Avoid Hotspotting

```
AtomicCounter counter = storageService.atomicCounterBuilder()
        .withName("uuid")
        .build()
        .asAtomicCounter();

final int batchSize = 1000;
long base = counter.getAndAdd(batchSize);

long uuid;
for (int i = 0; i < batchSize; i++) {
   uuid = base + i;
}
```

Avoid Hotspotting

```
AtomicIdGenerator idGenerator = storageService.atomicIdGeneratorBuilder()
        .withName("uuid")
        .build()
        .asAtomicIdGenerator();

long uuid = idGenerator.nextId();
```


ONF

Lesson #4

onf

OPTIMISTIC LOCKING IS YOUR FRIEND

Optimistic Locking is Your Friend

- All primitive operations (method calls) are atomic
- But multiple method calls are not atomic unless a lock is used
- Distributed locks are expensive
- Optimistic locks perform as well as a single atomic operation in the optimal case

ONF

Optimistic Locking is Your Friend

```
ConsistentMap<IpPrefix, Set<Route>> routes =
    storageService.<IpPrefix, Set<Route>>consistentMapBuilder()
        .withName("routes")
        .withSerializer(Serializer.using(KryoNamespaces.API))
        .build();

Set<Route> prefixRoutes = Versioned.valueOrNull(routes.get(prefix));
if (prefixRoutes == null) {
   prefixRoutes = new HashSet<>();
}
prefixRoutes.add(route);
routes.put(prefix, prefixRoutes);
```

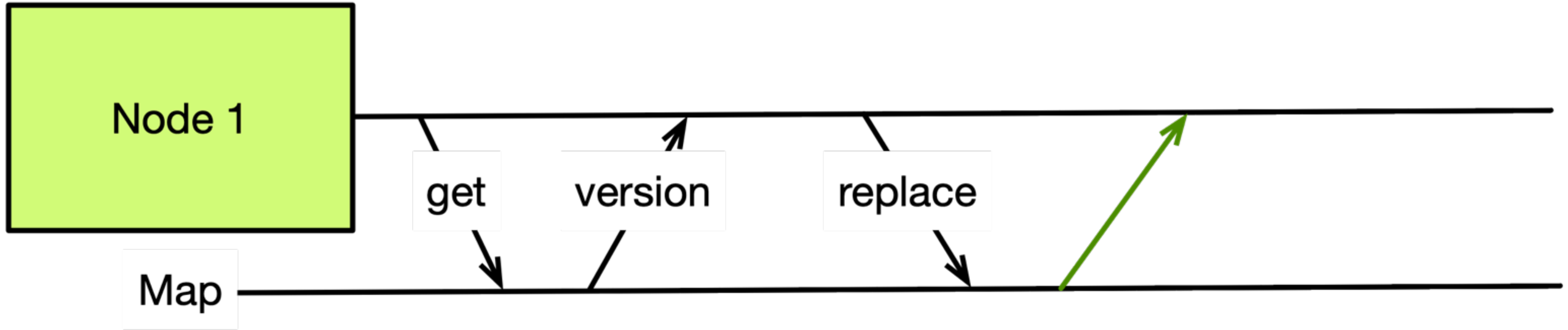
Optimistic Locking is Your Friend

```
ConsistentMap<IpPrefix, Set<Route>> routes =
    storageService.<IpPrefix, Set<Route>>consistentMapBuilder()
        .withName("routes")
        .withSerializer(Serializer.using(KryoNamespaces.API))
        .build();

Set<Route> prefixRoutes = Versioned.valueOrNull(routes.get(prefix));
if (prefixRoutes == null) {
   prefixRoutes = new HashSet<>();
}
prefixRoutes.add(route);
routes.put(prefix, prefixRoutes);
```

op

op

Optimistic Locking is Your Friend



Optimistic Locking is Your Friend

```
ConsistentMap<IpPrefix, Set<Route>> routes =
    storageService.<IpPrefix, Set<Route>>consistentMapBuilder()
        .withName("routes")
        .withSerializer(Serializer.using(KryoNamespaces.API))
        .build();

Versioned<Set<Route>> versioned = routes.get(prefix);
Set<Route> prefixRoutes;
if (versioned == null) {
   prefixRoutes = new HashSet<>();
   prefixRoutes.add(route);
   routes.putIfAbsent(prefix, prefixRoutes);
} else {
   prefixRoutes = new HashSet<>(versioned.value());
   prefixRoutes.add(route);
   routes.replace(prefix, versioned.version(), prefixRoutes);
}
```

Optimistic Locking is Your Friend

```
ConsistentMap<IpPrefix, Set<Route>> routes =
    storageService.<IpPrefix, Set<Route>>consistentMapBuilder()
        .withName("routes")
        .withSerializer(Serializer.using(KryoNamespaces.API))
        .build();

Versioned<Set<Route>> versioned = routes.get(prefix);
Set<Route> prefixRoutes;
if (versioned == null) {
   prefixRoutes = new HashSet<>();
   prefixRoutes.add(route);
   routes.putIfAbsent(prefix, prefixRoutes);
} else {
   prefixRoutes = new HashSet<>(versioned.value());
   prefixRoutes.add(route);
   routes.replace(prefix, versioned.version(), prefixRoutes);
}
```

Optimistic Locking is Your Friend

```
ConsistentMap<IpPrefix, Set<Route>> routes =
    storageService.<IpPrefix, Set<Route>>consistentMapBuilder()
        .withName("routes")
        .withSerializer(Serializer.using(KryoNamespaces.API))
        .build();

routes.compute(prefix, (p, r) -> {
   if (r == null) {
       r = new HashSet<>();
   }
   r.add(route);
   return r;
});
```

Lesson #5

ONF

OPTIMISTIC LOCKING CAN BE A BAD FRIEND

onf

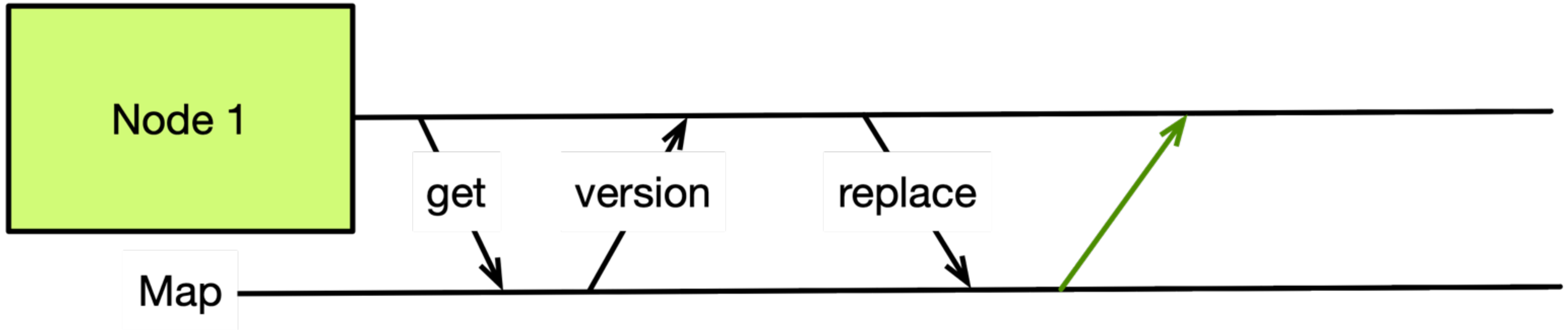
Optimistic Locking Can Be a Bad Friend

- While optimistic locks perform well in optimal cases, high lock contention can lead to significant load on the cluster
- Pay attention to optimistic locks for bottlenecks
- Use lock free primitives if possible
- Consider pessimistic locks or leader election otherwise

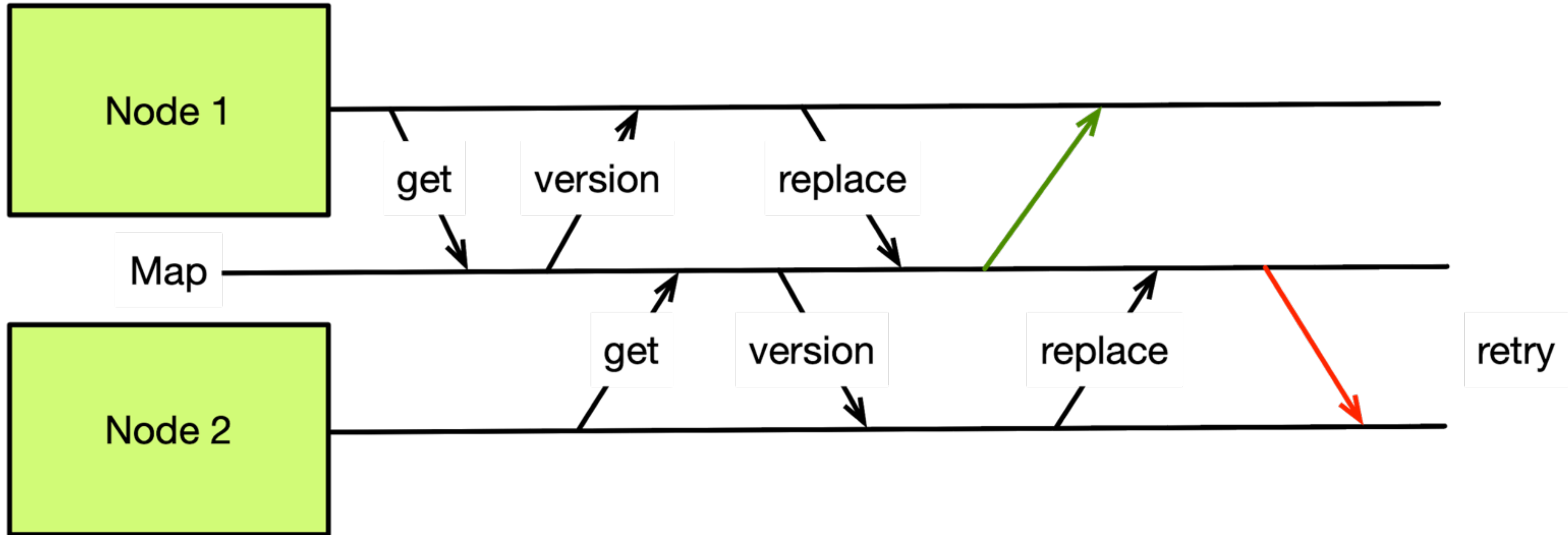
Optimistic Locking Can Be a Bad Friend

```
ConsistentMap<IpPrefix, Set<Route>> routes =
    storageService.<IpPrefix, Set<Route>>consistentMapBuilder()
        .withName("routes")
        .withSerializer(Serializer.using(KryoNamespaces.API))
        .build();

routes.compute(prefix, (p, r) -> {
   if (r == null) {
        r = new HashSet<>();
   }
   r.add(route);
   return r;
});
```

Optimistic Locking Can Be a Bad Friend



Optimistic Locking Can Be a Bad Friend



Optimistic Locking Can Be a Bad Friend

```
ConsistentMap<IpPrefix, Set<Route>> routes =
    storageService.<IpPrefix, Set<Route>>consistentMapBuilder()
        .withName("routes")
        .withSerializer(Serializer.using(KryoNamespaces.API))
        .build();

DistributedLock lock = storageService.lockBuilder()
        .withName("routes-lock")
        .build()
        .asLock();

lock.lock();
try {
   Set<Route> prefixRoutes = Versioned.valueOrNull(routes.get(prefix));
   if (prefixRoutes == null) {
       prefixRoutes = new HashSet<>();
   }
   prefixRoutes.add(route);
   routes.put(prefix, prefixRoutes);
} finally {
   lock.unlock();
}
```


ONF

Optimistic Locking Can Be a Bad Friend

```
ConsistentMap<IpPrefix, Set<Route>> routes =
    storageService.<IpPrefix, Set<Route>>consistentMapBuilder()
        .withName("routes")
        .withSerializer(Serializer.using(KryoNamespaces.API))
        .build();

DistributedLock lock = storageService.lockBuilder()
        .withName("routes-lock")
        .build()
        .asLock();

lock.lock();        <---- write
try {
   Set<Route> prefixRoutes = Versioned.valueOrNull(routes.get(prefix));
   if (prefixRoutes == null) {
       prefixRoutes = new HashSet<>();
   }
   prefixRoutes.add(route);
   routes.put(prefix, prefixRoutes);
} finally {
   lock.unlock();   <---- write
}
```

Optimistic Locking Can Be a Bad Friend

```
ConsistentMultimap<IpPrefix, Route> routes =
    storageService.<IpPrefix, Route>consistentMultimapBuilder()
        .withName("routes")
        .withSerializer(Serializer.using(KryoNamespaces.API))
        .build();

routes.put(prefix, route);
```


onf

Lesson #6

ONF

EXPLOIT DATA LOCALITY

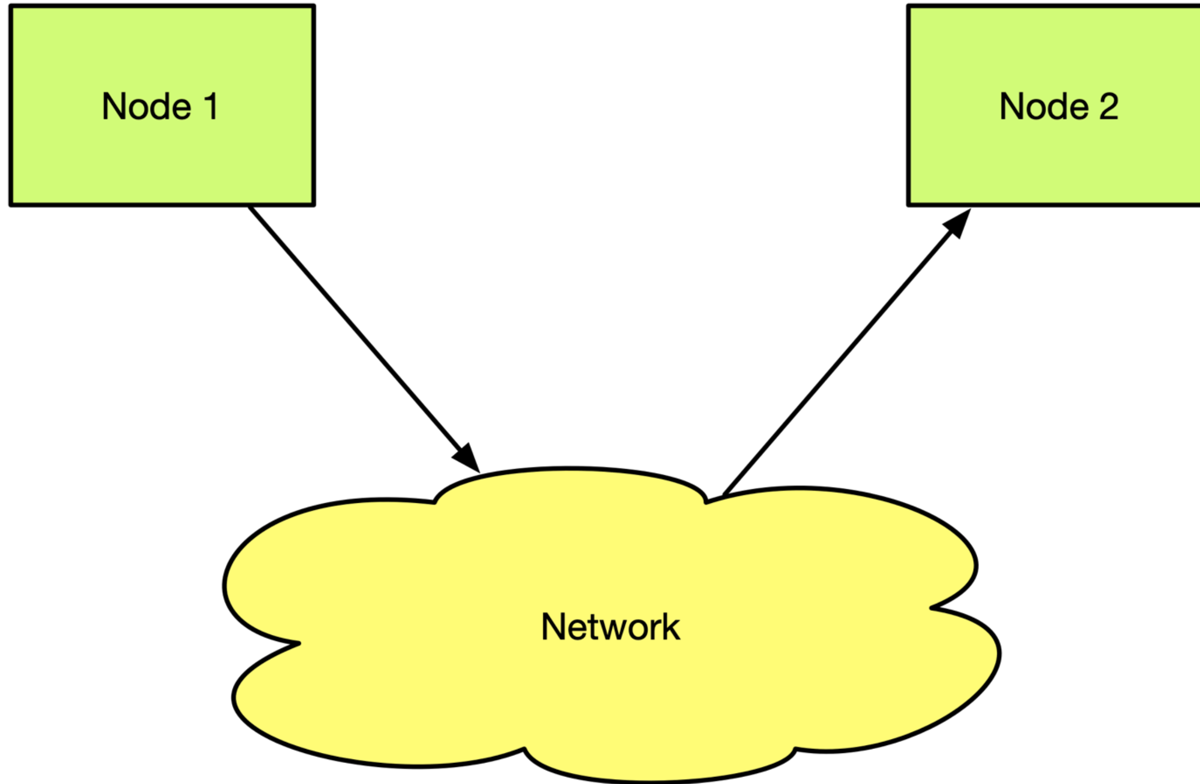
onf

Exploit Data Locality

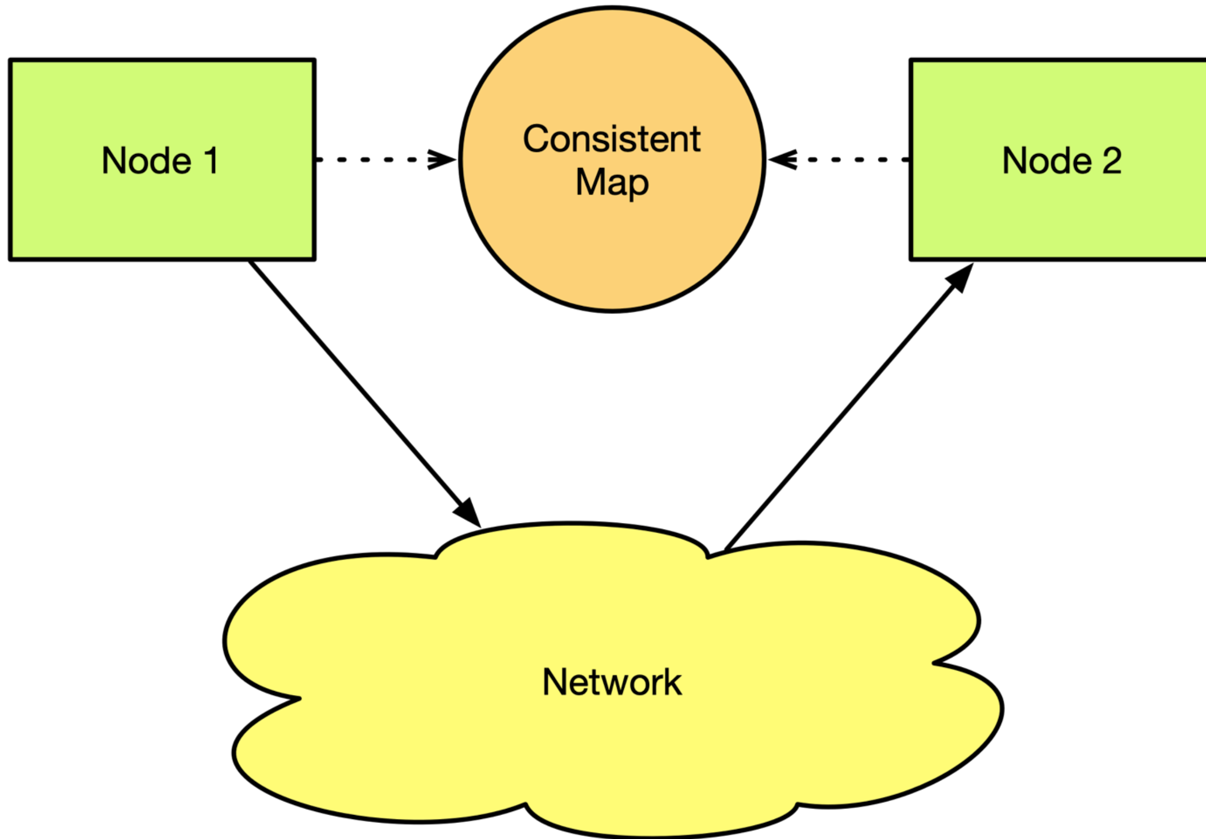
- The network is expensive
- Distributed primitives are even more expensive
- Exploit partitioning schemes, mastership, etc to avoid synchronization altogether
- This may be lesson #6, but it’s rule #1 for scaling!

ONF

Exploit Data Locality

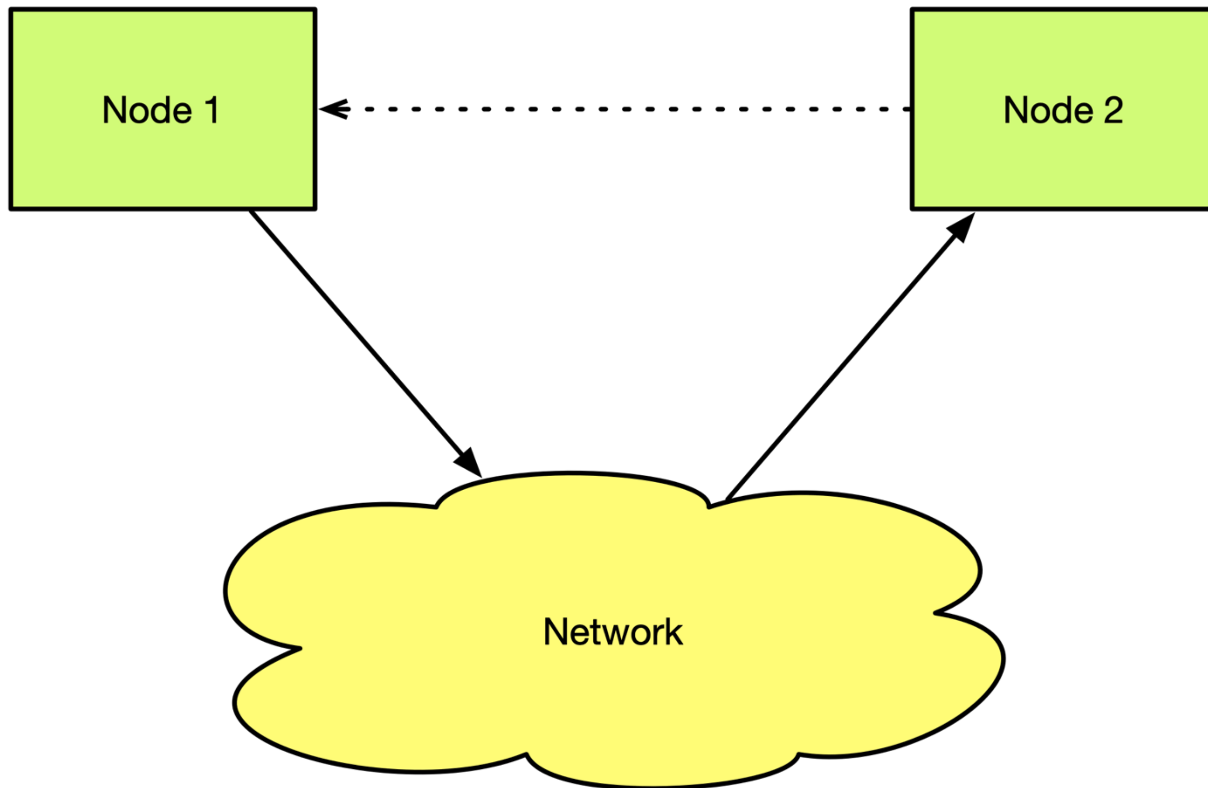


Exploit Data Locality



onf

Exploit Data Locality



onf

Review

- Do not poll primitives
- Do not block the event loop
- Avoid hotspotting
- Optimistic locking for low contention
- Pessimistic locking for high contention
- Data locality is king

onf

Questions?

onf