ISSU in ONOS

Architecture and Implementation

Uyen Chau
Member of Technical Staff @ ONF

Overview

Distributed systems in ONOS

ISSU Protocol

Distributed Systems in ONOS

Distributed Systems in ONOS

Controller

Northbound APIs

Distributed Primitives

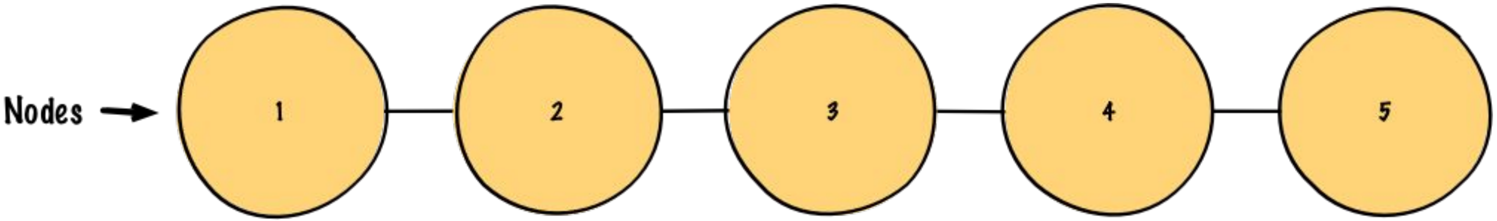
Primitive Protocols

Serialization

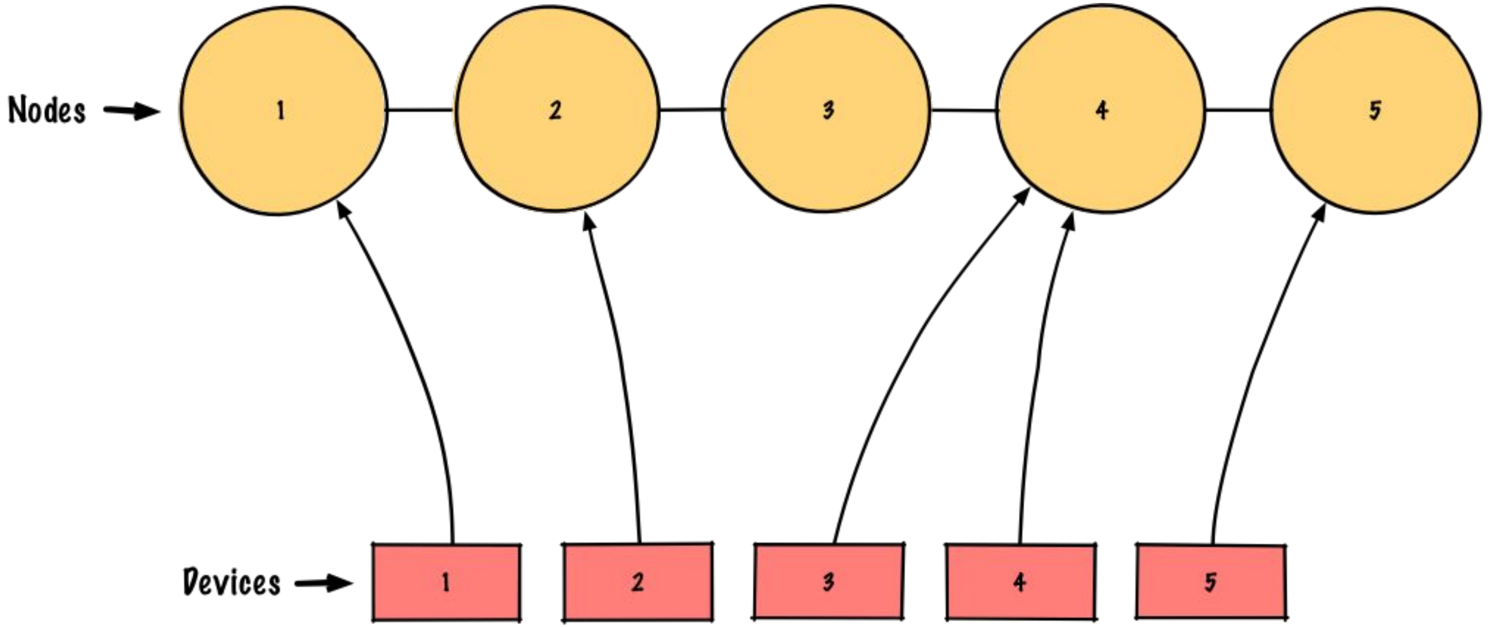
Controller

- ONOS controllers typically consist of an odd number of nodes
- Use peer-to-peer protocols for state replication and coordination
- Elect a master for each device controlled by the controller
- Can tolerate loss of up to a minority of nodes
- New masters elected after node failure

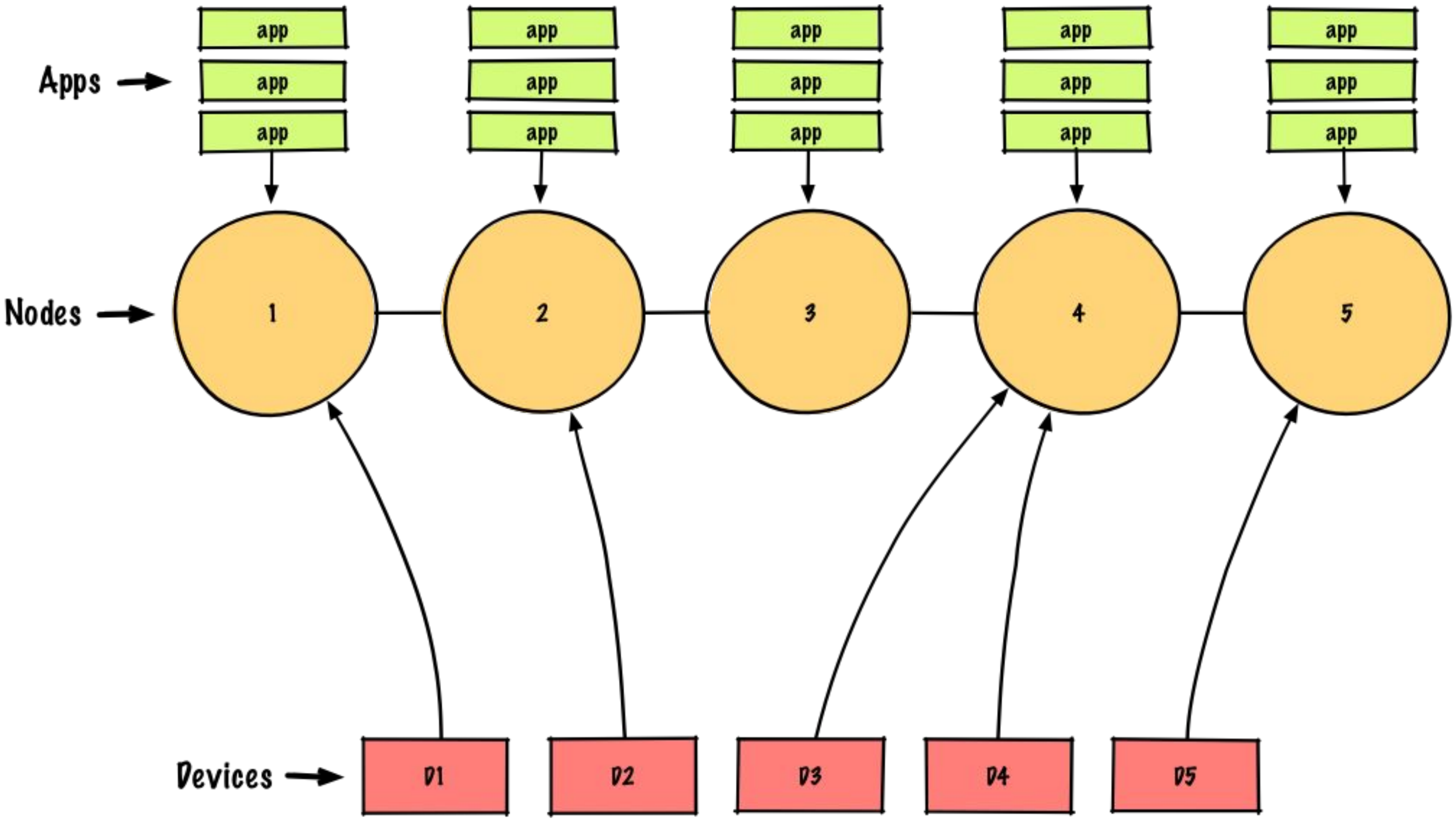
Controller



Controller



Controller



Apps
app
app
app
app
app
app
app
app
app
app
app
app
app
app
app
Nodes
1
2
3
4
5
Devices
D1
D2
D3
D4
D5

Northbound APIs

Applications

- Northbound APIs are accessed via
  - Applications
  - REST APIs
  - CLI
- Applications distributed in each controller node
- Can access all northbound APIs
- Can use primitives for state replication and coordination

Services

- ONOS applications interact with high level services
- Expose information about the network, cluster, configuration, etc
  - DeviceService
  - FlowRuleService
  - IntentService
  - ClusterService
  - NetworkConfigService
  - etc
- Generally stateless

Stores

- Stateful backing to services
  - DeviceStore
  - FlowRuleStore
  - IntentStore
  - ClusterStore
  - NetworkConfigStore
- Usually distributed in multi-node controllers

Stores

- Use a variety of distributed systems protocols
  - Gossip
  - Anti-entropy
  - Consensus
  - Primary-backup
  - Distributed primitives

Distributed Primitives

- Encapsulate complex distributed systems protocols
- Used by both stores and applications
- State replication
  - EventuallyConsistentMap
  - ConsistentMap
  - DistributedSet
  - DocumentTree
  - AtomicCounter
- Coordination
  - LeaderElector
  - DistributedLock
  - WorkQueue

Distributed Primitives

```
apps = storageService.<ApplicationId, Application>consistentMapBuilder()
    .withName("onos-apps")
    .withRelaxedReadConsistency()
    .withSerializer(Serializer.using(KryoNamespace.newBuilder()
        .register(KryoNamespaces.API)
        .register(ApplicationId.class)
        .register(Application.class)
        .register(Version.class)
        .register(ApplicationRole.class)
        .build()))
    .build();
```

Distributed Primitives

```
public void storeApplication(Application app) {
    apps.put(app.id(), app);
}

public Application getApplication(ApplicationId appId) {
    return Versioned.valueOrNull(apps.get(appId));
}
```

Primitive Protocols

- Gossip
  - Periodically send updates to peers
  - Use logical/wall clock timestamps for ordering
- Primary-backup/multi-primary
  - Replicate from primaries to backups
- Consensus
  - Embedded Raft clusters for consensus
  - Primitives modelled as replicated state machines
  - Requires a quorum to make progress

Primitive Protocols

- Built on Atomix distributed systems framework
- Multiple distributed systems protocols
    - Raft consensus
    - Primary-backup
    - Partitioning
    - Cluster management
- Supports custom replicated state machines

Serialization

- Kryo for fast binary serialization
- `FieldSerializer`
  - Default serializer
  - Uses reflection to map fields to bytes
- `Serializer`
  - Custom serializer
- `KryoNamespace`
  - Wrapper around Kryo serialization
  - Register serializable types
  - Assigns sequential type IDs
  - Uses `FieldSerializer` by default
  - Supports custom `Serializers`

Serialization

```java
private final Serializer SERIALIZER = Serializer.using(KryoNamespace.newBuilder()
    .register(new HeartbeatMessageSerializer(), HeartbeatMessage.class)
    .register(ControllerNode.class)
    .register(ControllerNode.State.class)
    .register(NodeId.class)
    .build());
```

Serialization

```java
private static class HeartbeatMessageSerializer extends com.esotericsoftware.kryo.Serializer<HeartbeatMessage> {
    @Override
    public void write(Kryo kryo, Output output, HeartbeatMessage message) {
        kryo.writeObject(output, message.source());
        kryo.writeObject(output, message.state());
    }

    @Override
    public HeartbeatMessage read(Kryo kryo, Input input, Class<HeartbeatMessage> type) {
        ControllerNode source = kryo.readObject(input, ControllerNode.class);
        ControllerNode.State state = kryo.readObject(input, ControllerNode.State.class);
        return new HeartbeatMessage(source, state);
    }
}
```

In-Service Software Upgrades

In-Service Software Upgrades

Requirements

The Upgrade Workflow

Fault Tolerance

Compatibility Issues

Upgrading State

Future Work

ISSU Requirements

- Support ISSU for ONOS core and applications
- Only a single controller node down at a time
- Maintain fault tolerance through upgrade process
- Modify replicated state during upgrades
- Introduce new primitives and primitive operations
- Recover from catastrophic failures

ISSU Requirements

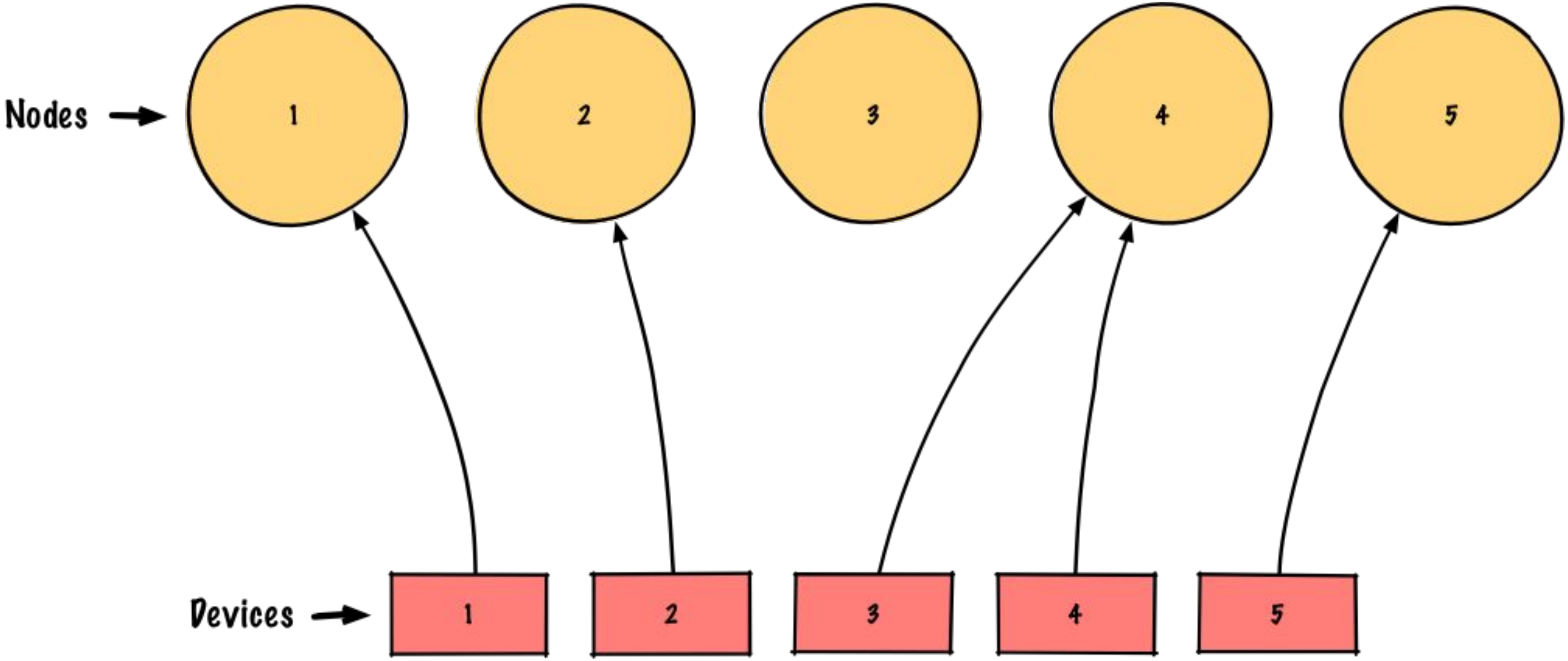
- Encapsulate complexity in distributed primitives
- Require code changes only if state and/or protocols changed
- Use existing bootstrap APIs to modify state

The Upgrade Workflow

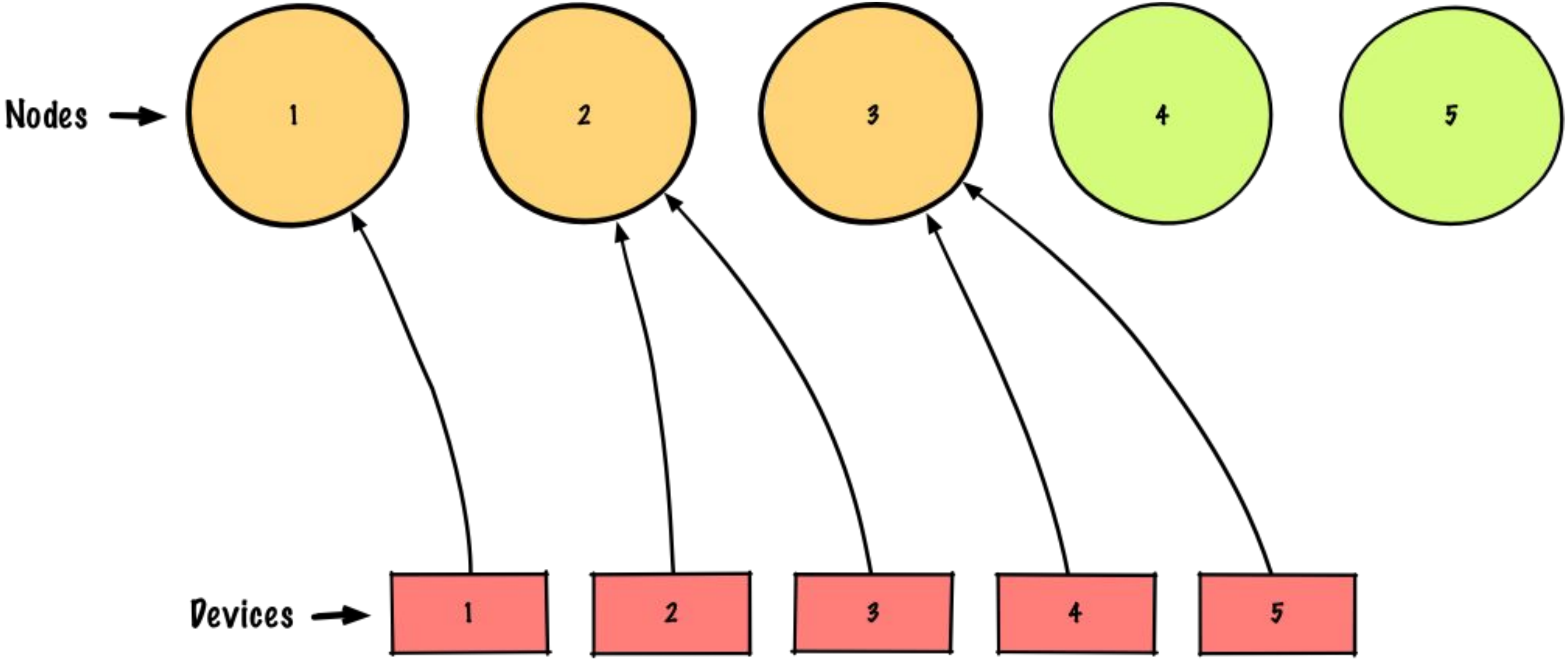
The Upgrade Workflow

- ISSU performed as a partitioned rolling upgrade
- Upgrade a subset of the cluster
- Hand control of the network over to the new version
  - A simple mastership change from one version to the next
- Upgrade the remaining nodes
- Must support multiple compatible versions running in the same cluster at the same time

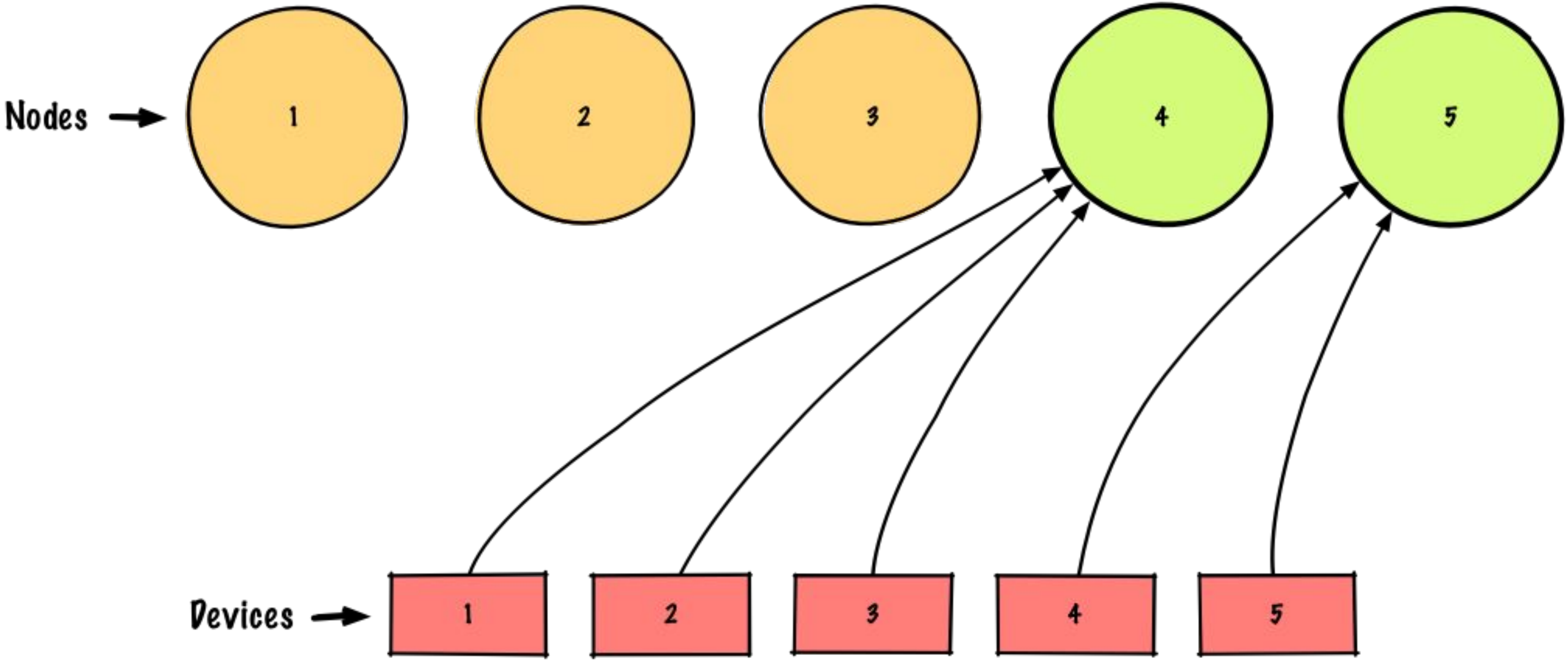
The Upgrade Workflow



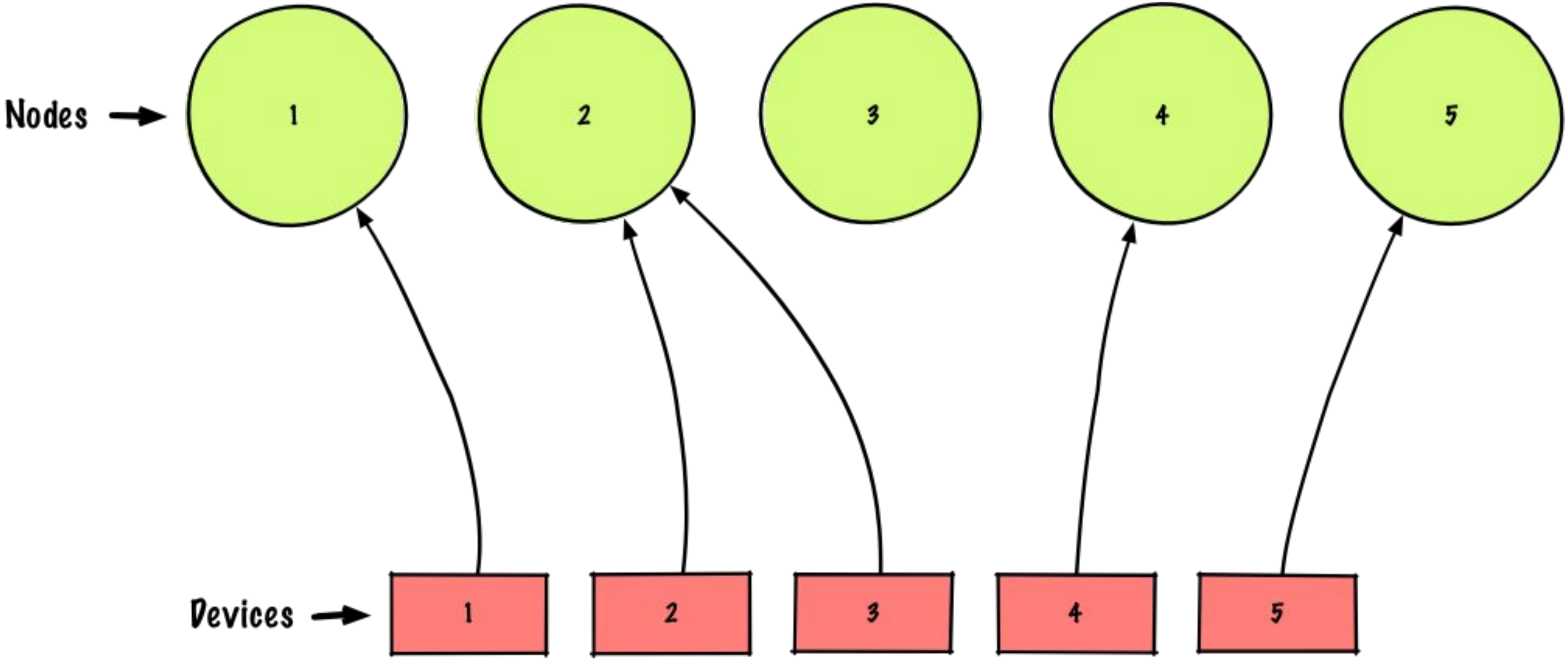
The Upgrade Workflow



The Upgrade Workflow

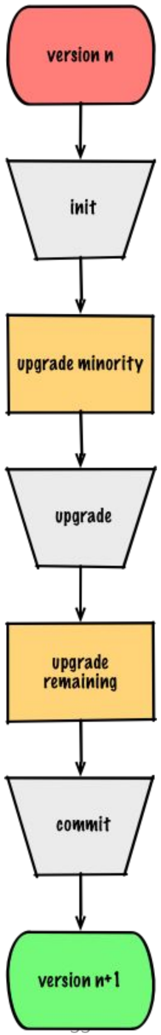


The Upgrade Workflow



The Upgrade Workflow

- `issu init`
  - Begin an upgrade
- `issu upgrade`
  - Transfer control from one version to the next
- `issu commit`
  - Complete an upgrade


version n
init
upgrade minority
upgrade
upgrade remaining
commit
version n+1

Demo

Fault Tolerance

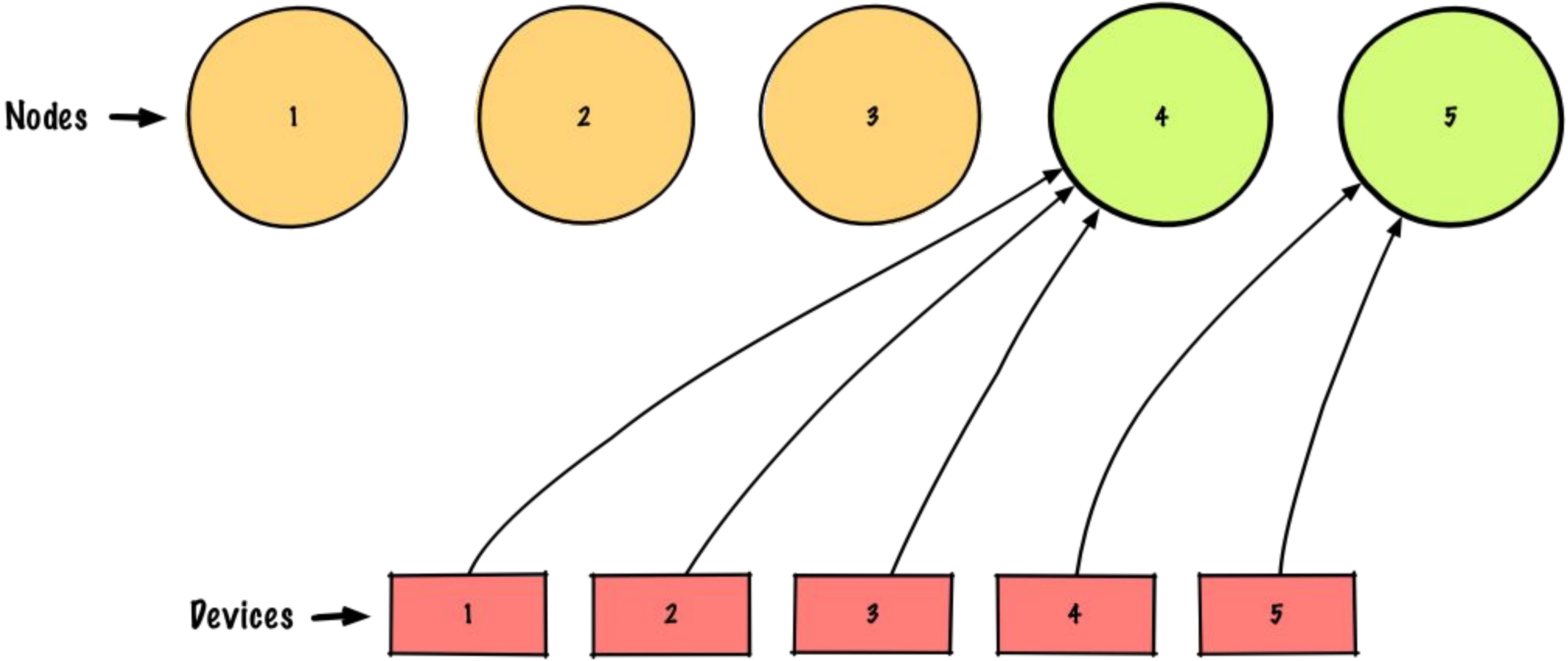
Fault Tolerance

- An upgrade itself can fail
  - New version fails to operate network correctly
  - Applications fail to activate
  - Node/network problems prevent control from being handed over
- Upgrade testing
  - Upgrades should be tested thoroughly in prior to production

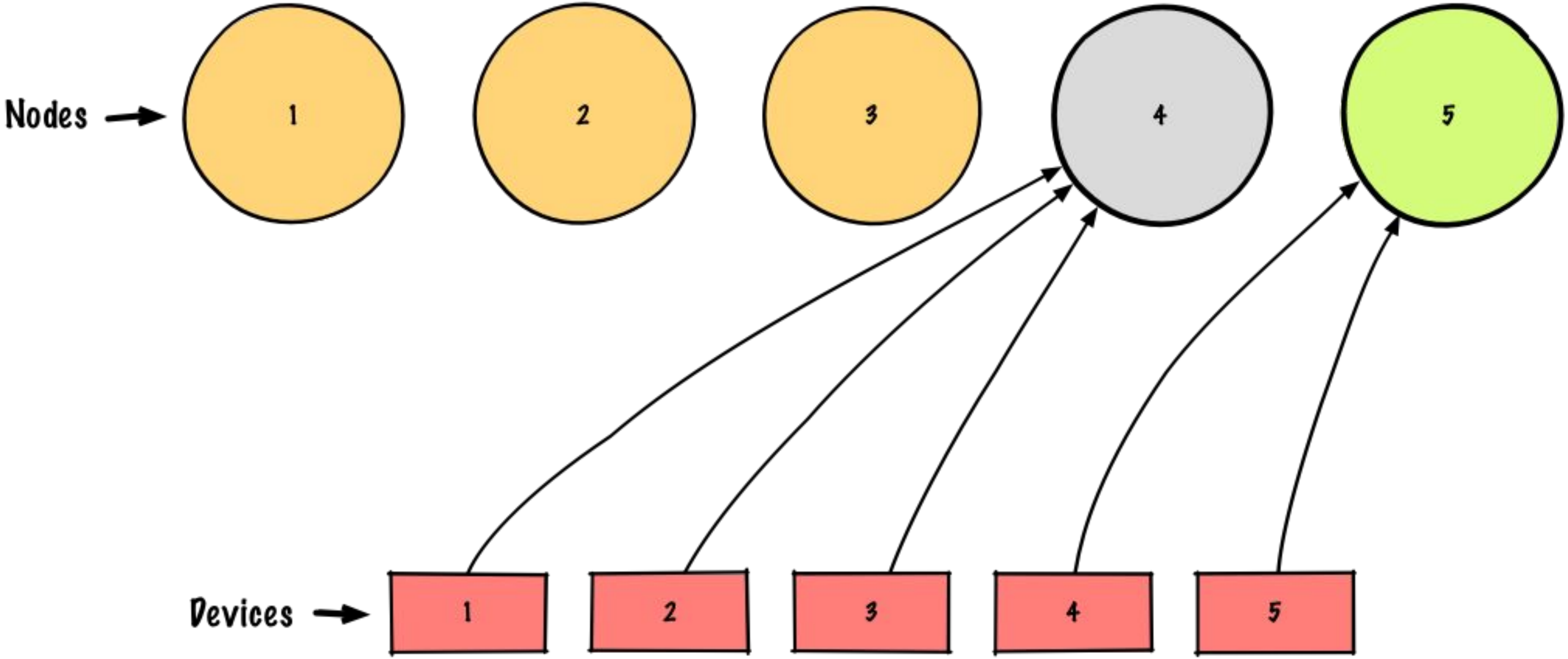
Fault Tolerance

- Production upgrade failure recovery
  - Rollback upgrade to restore to pre-upgrade state
  - Switch mastership from new version back to old
  - Revert stores to previous revisions
- Automatic rollbacks
  - Failures can cause undue strain on either version
  - Detect failures on upgraded nodes and rollback

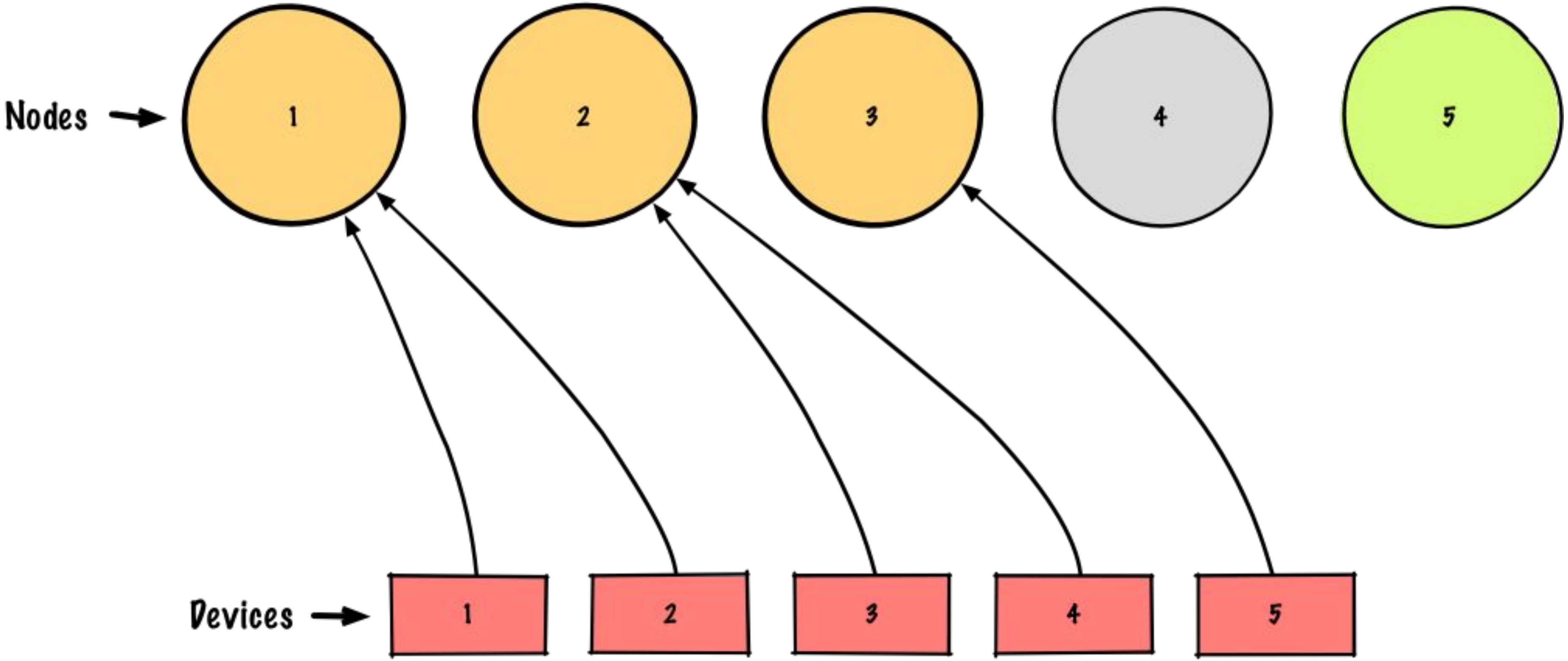
Fault Tolerance



Fault Tolerance

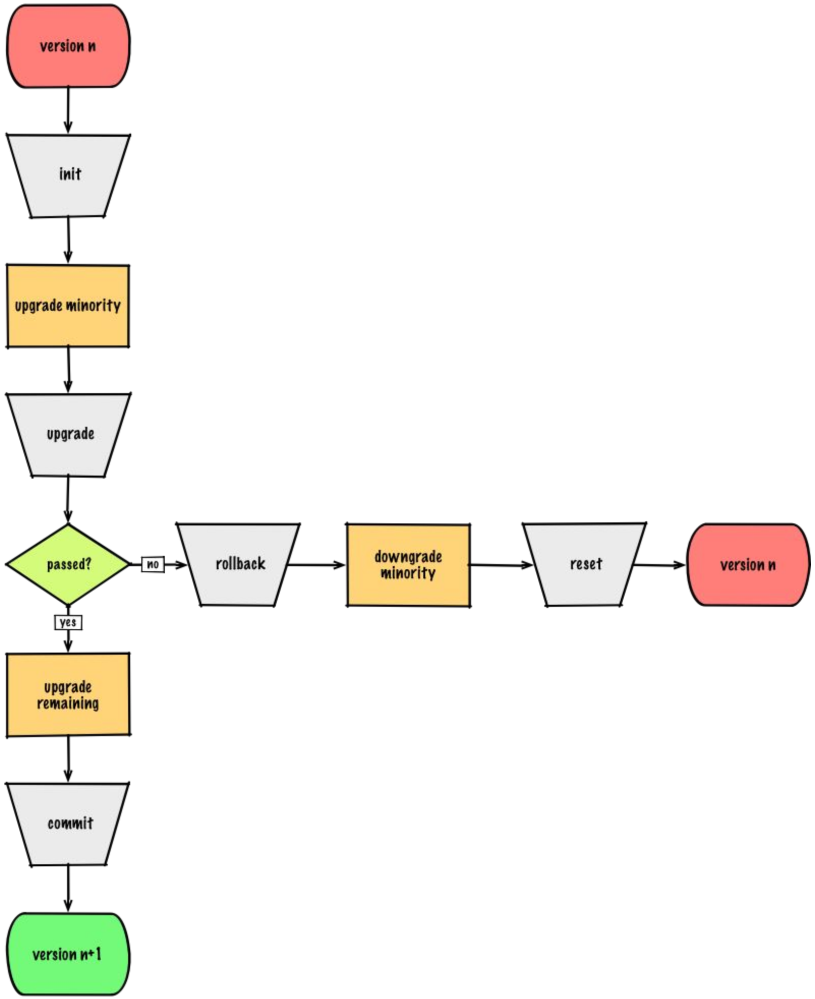


Fault Tolerance



The Upgrade Workflow

- `issu init`
  - Begin an upgrade
- `issu upgrade`
  - Transfer control from one version to the next
- `issu commit`
  - Complete an upgrade
- `issu rollback`
  - Transfer control back to the old version and revert the state of the cluster
- `issu reset`
  - Reset the upgrade protocol after rollback


version n
init
upgrade minority
upgrade
passed?
no
rollback
downgrade minority
reset
version n
yes
upgrade remaining
commit
version n+1

Demo

Compatibility Issues

Compatibility Issues

- New versions may break compatibility of serialized objects
  - Fields added/removed/changed
- Enabling ISSU in ONOS enables backwards/forwards compatible serialization by default
  - `onos.cluster.issu.enabled` system property
  - Implemented with Kryo's `CompatibleFieldSerializer`
  - Significant overhead to compatible serialization (20-50%)
  - Can be disabled for specific serializers for better performance

Compatibility Issues

```java
private final Serializer SERIALIZER = Serializer.using(KryoNamespace.newBuilder()
    .setCompatible(true)
    .register(HeartbeatMessage.class)
    .register(ControllerNode.class)
    .register(ControllerNode.State.class)
    .register(NodeId.class)
    .build());
```

Compatibility Issues

- Custom serializers are not inherently capable of handling schema evolution
- Custom serializers must either be static or designed to handle changes
  - Create new classes to introduce changes
  - Encode a 1-byte format version number

Compatibility Issues

```java
private static class HeartbeatMessageSerializer extends com.esotericsoftware.kryo.Serializer<HeartbeatMessage> {
    private final byte VERSION = 1;

    @Override
    public void write(Kryo kryo, Output output, HeartbeatMessage message) {
        output.writeByte(VERSION);
        kryo.writeObject(output, message.source());
        kryo.writeObject(output, message.state());
    }

    @Override
    public HeartbeatMessage read(Kryo kryo, Input input, Class<HeartbeatMessage> type) {
        byte version = input.readByte();
        ControllerNode source = kryo.readObject(input, ControllerNode.class);
        ControllerNode.State state = kryo.readObject(input, ControllerNode.State.class);
        return new HeartbeatMessage(source, state);
    }
}
```

Compatibility Issues

```java
private static class HeartbeatMessageSerializer extends com.esotericsoftware.kryo.Serializer<HeartbeatMessage> {
    private final byte VERSION = 2;

    @Override
    public void write(Kryo kryo, Output output, HeartbeatMessage message) {
        output.writeByte(VERSION);
        kryo.writeObject(output, message.source());
        kryo.writeObject(output, message.state());
        output.writeLong(message.timestamp());
    }

    @Override
    public HeartbeatMessage read(Kryo kryo, Input input, Class<HeartbeatMessage> type) {
        byte version = input.readByte();
        ControllerNode source = kryo.readObject(input, ControllerNode.class);
        ControllerNode.State state = kryo.readObject(input, ControllerNode.State.class);
        switch (version) {
            case 1:
                return new HeartbeatMessage(source, state, System.currentTimeMillis());
            case 2:
                long timestamp = input.readLong();
                return new HeartbeatMessage(source, state, timestamp);
            default:
                throw new AssertionError();
        }
    }
}
```

Compatibility Issues

- KryoNamespace objects must be modified carefully
  - Changes to ordering of registrations can change type ID mappings
  - Serialized objects will no longer reference the correct type during deserialization
  - Use static registration IDs to avoid breaking changes during refactoring

Compatibility Issues

```
private final Serializer SERIALIZER = Serializer.using(KryoNamespace.newBuilder()
    .setCompatible(true)
    .register(HeartbeatMessage.class)
    .register(ControllerNode.class)
    .register(ControllerNode.State.class)
    .register(NodeId.class)
    .build());
```

Upgrading State

Upgrading State

- Two types of state
  - Shared state
  - Isolated state
- Network state is shared across versions
  - DeviceStore
  - LinkStore
  - HostStore
- Controller state is versioned
  - Changes made in newer versions not visible in older versions
  - Not propagated to network until after mastership change

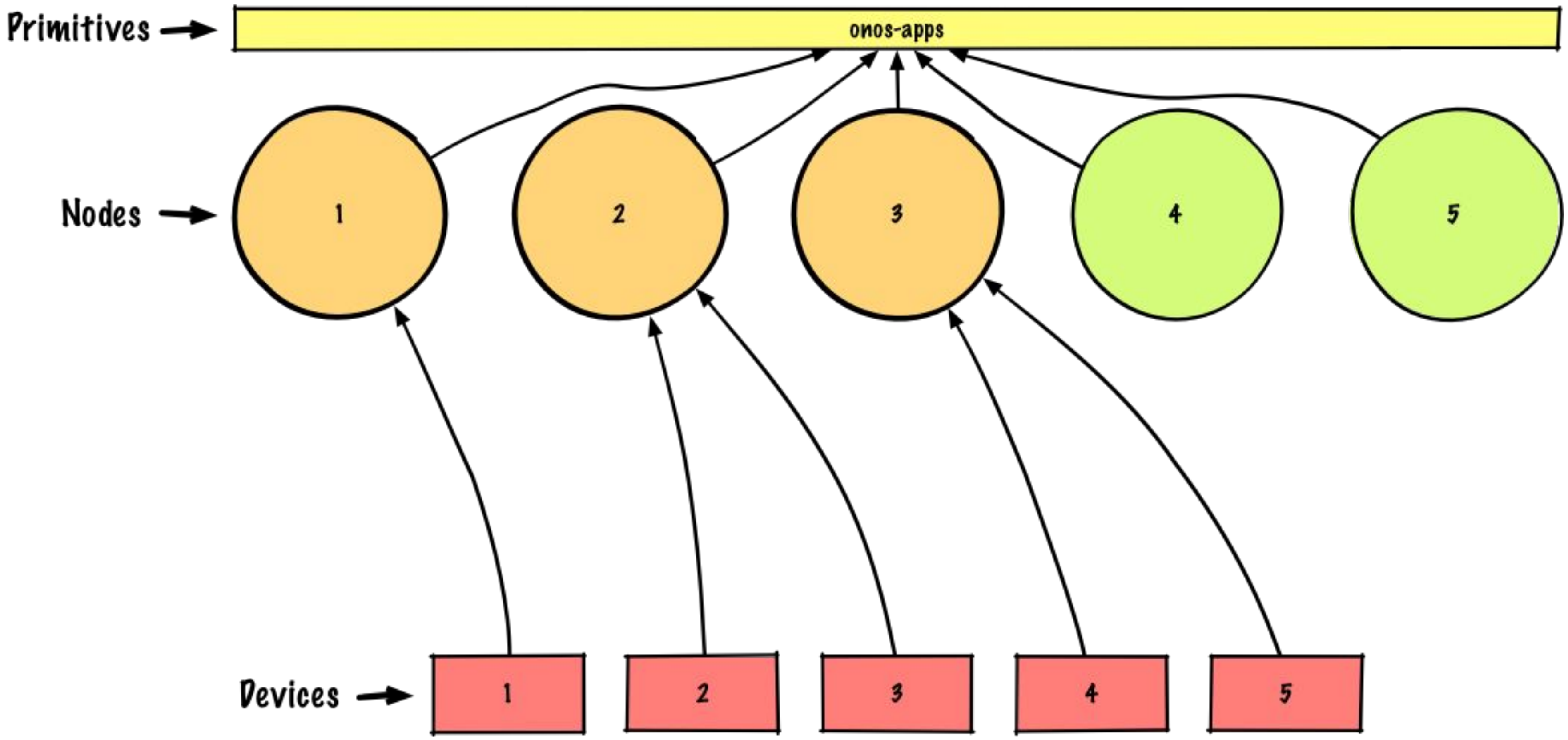
Upgrading State

- Upgraded nodes must be able to modify primitive state
- But modified state may not be compatible with older versions
- Two approaches to state modification
  - Modify on write
  - Modify on read

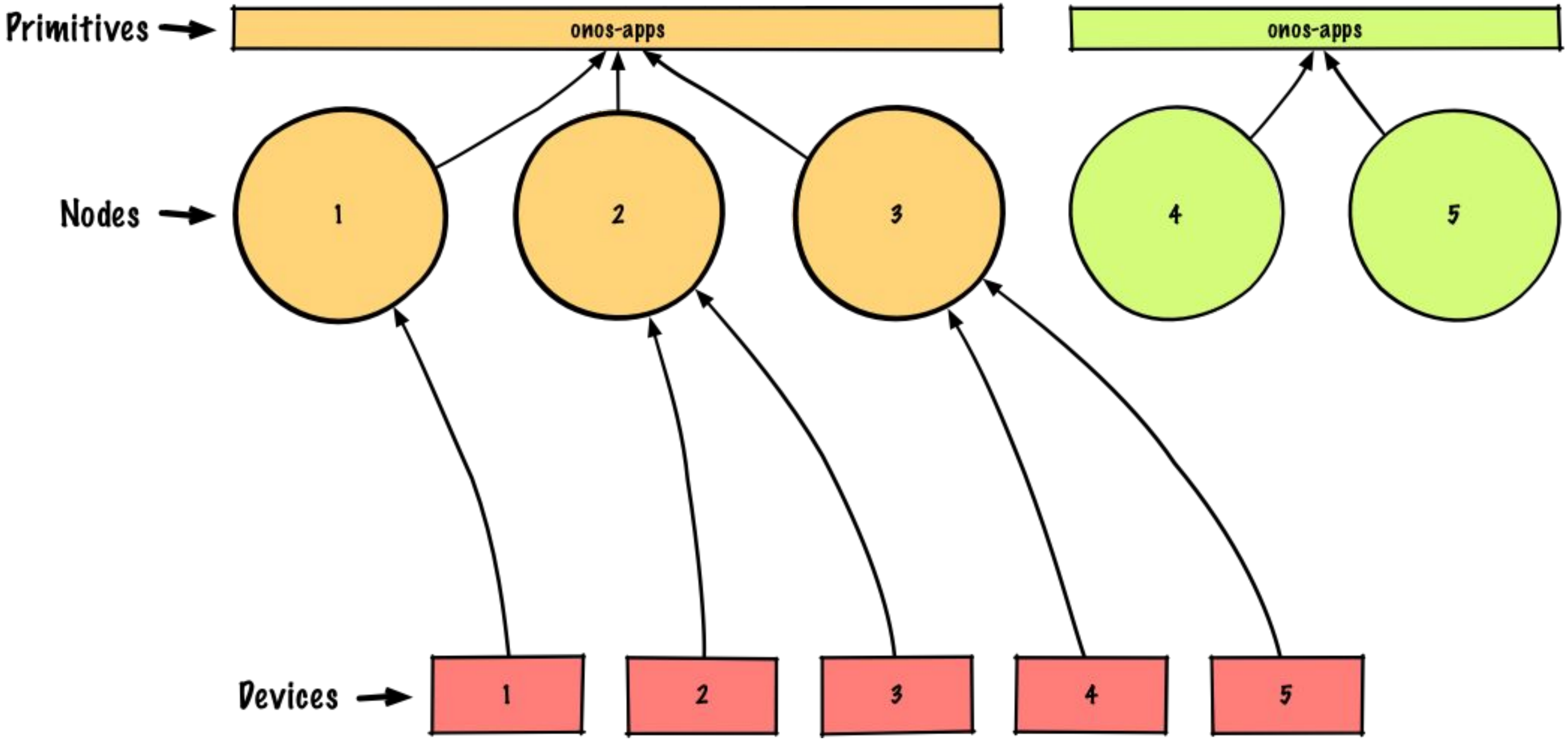
Modify on Write

- Primitive revisions
  - The primary mechanism for isolating primitive changes
  - Fork primitive state machines
  - Preserves consistency guarantees
- Three revision isolation levels
  - Full isolation
  - Simple versioning
  - Forward propagation

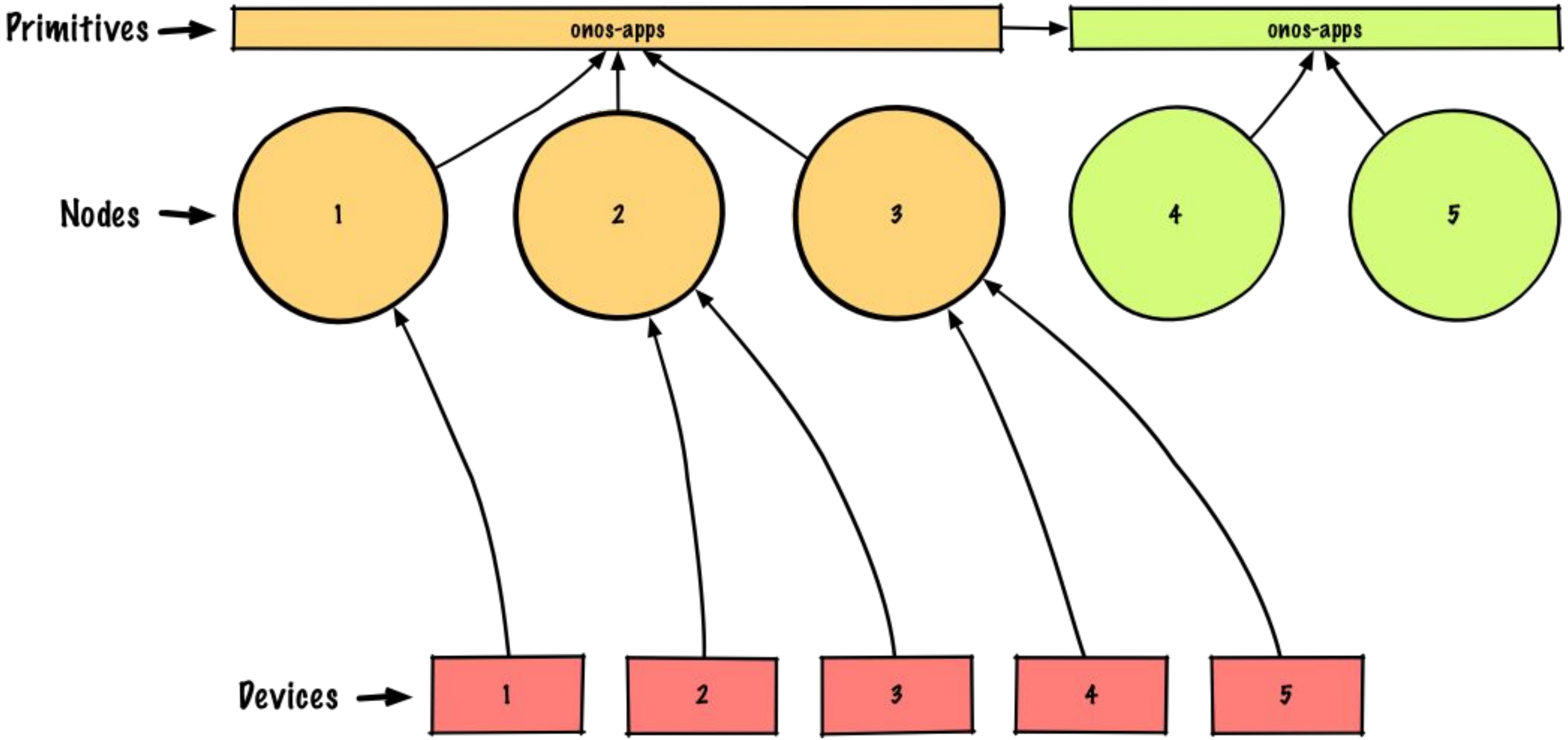
Modify on Write



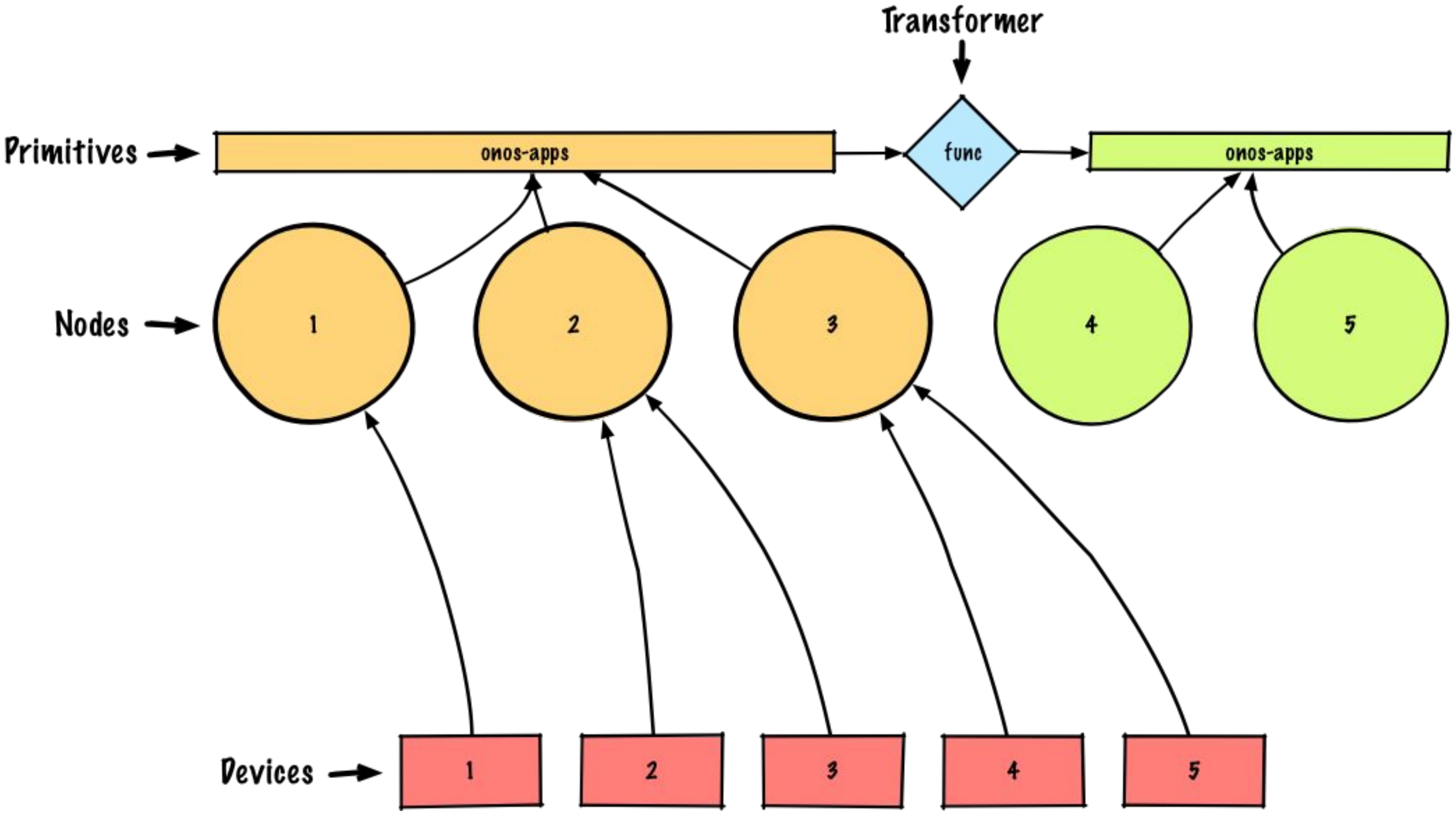
Modify on Write



Modify on Write



Modify on Write



Modify on Read

- Apply transformers to state on read
  - Write to primitives with a node version number
  - When reading state from another node, apply transformations
  - Used in the ONOS `ApplicationStore`

Modify on Read

```
apps = storageService.<ApplicationId, Application>consistentMapBuilder()
    .withName("onos-apps")
    .withRelaxedReadConsistency()
    .withSerializer(Serializer.using(KryoNamespace.newBuilder()
        .register(KryoNamespaces.API)
        .register(ApplicationId.class)
        .register(Application.class)
        .register(Version.class)
        .register(ApplicationRole.class)
        .build()))
    .withCompatibilityFunction((app, version) -> {
        // Load the application description from disk. If the version doesn't match the persisted
        // version, update the stored application with the new version.
        ApplicationDescription appDesc = getApplicationDescription(app.id().name());
        if (!appDesc.version().equals(appHolder.app().version())) {
            return DefaultApplication.builder(app)
                .withVersion(appDesc.version())
                .build();
        }
        return app;
    })
    .build();
```

Future Work

Future Work

- How do we upgrade applications independently of the core?
- How do we prevent breaking changes to serializer configurations?
- How do we monitor refactoring for breaking changes?
- What happens if a single application breaks the upgrade path?
- How do we introduce new features to distributed primitives themselves?
- How can we simplify isolating applications?

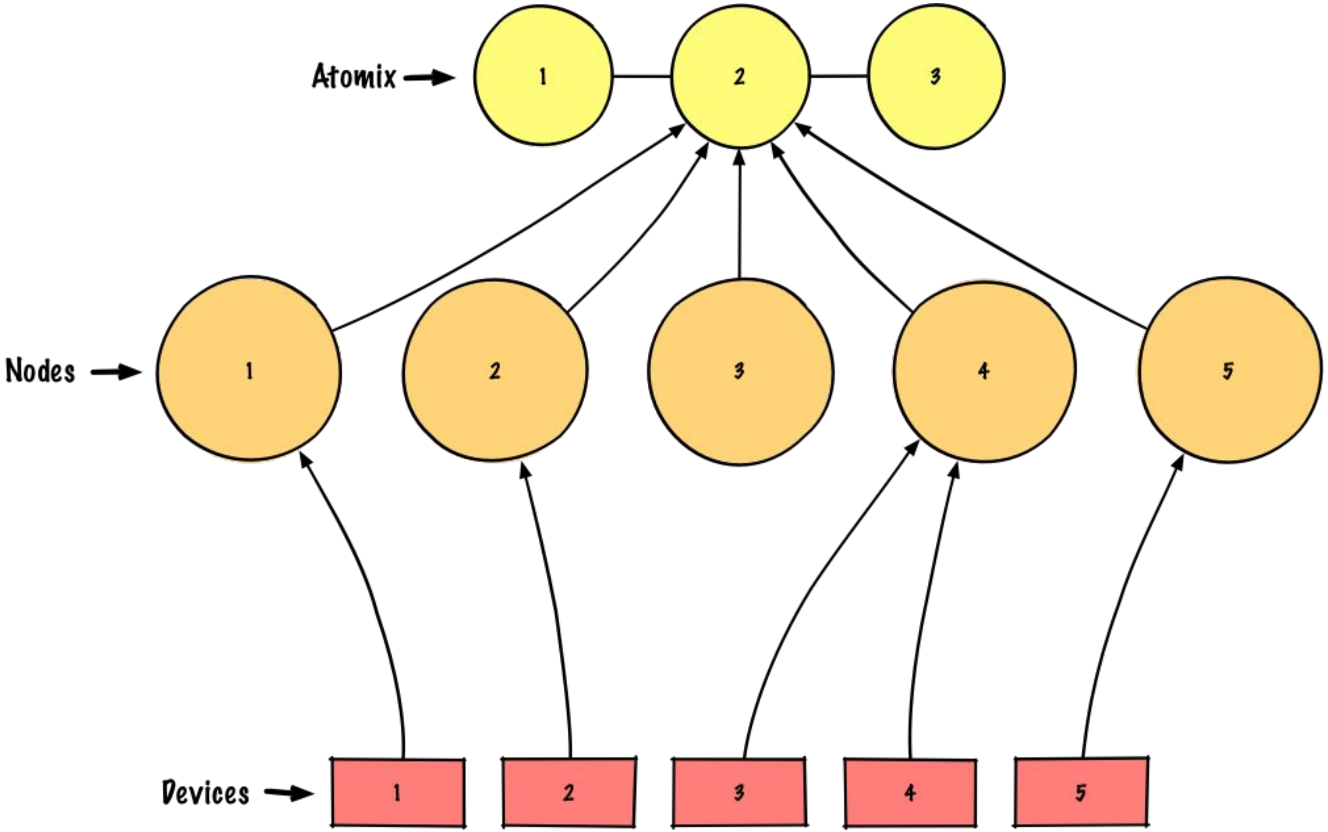
Application Upgrades

- Currently possible, but not elegant
  - Assign unique VERSION for each application upgrade
- Add hash of installed applications to VERSION?
- Upgrade a subset of components (app dependencies)?

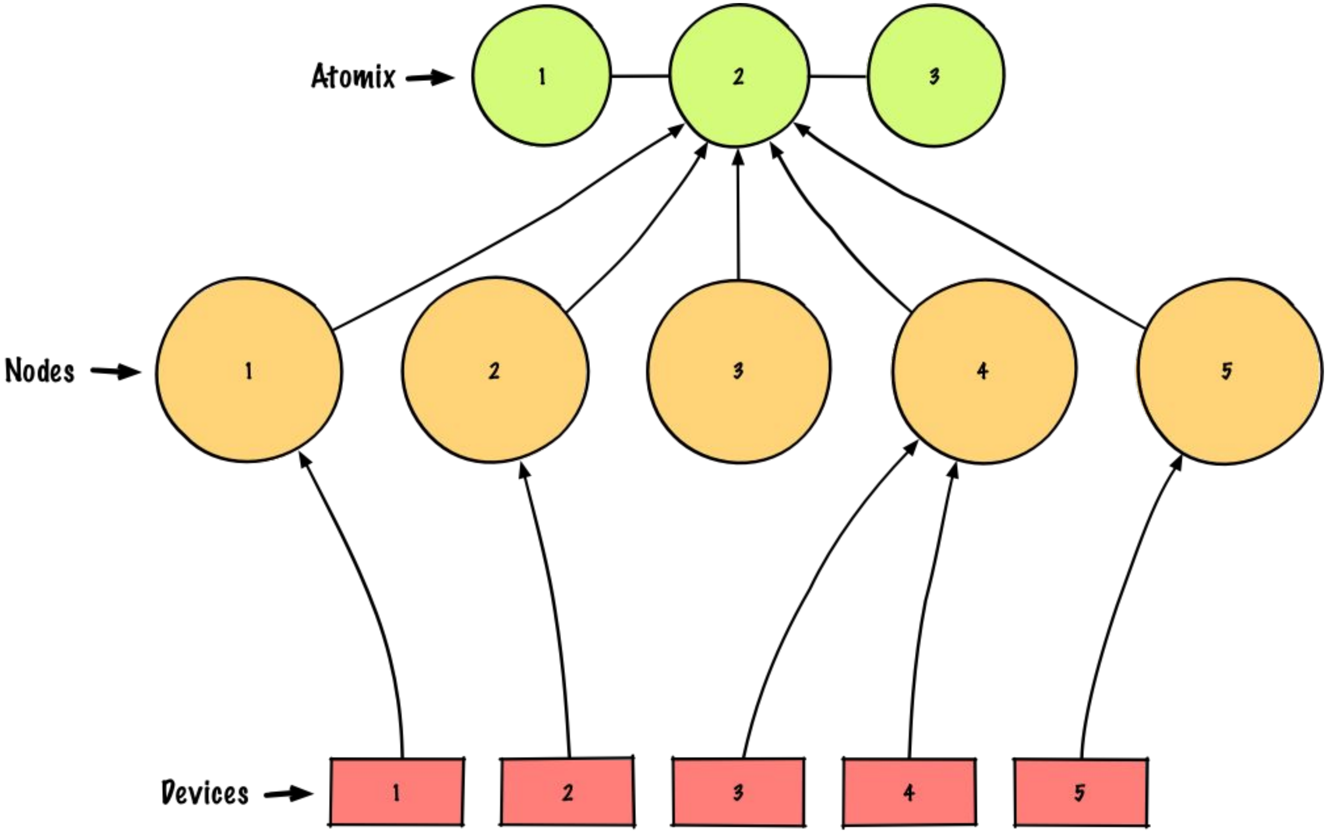
Introducing New Features

- New Raft primitives cannot be introduced because of quorum requirements
- New Raft operations cannot be introduced for the same reason
- Initial solution
  - Fork Raft partitions on upgrade
  - Difficult to synchronize across Raft partitions
  - Increases overhead during ISSU by doubling number of partitions
- Future solution
  - Separate Atomix/Raft from ONOS controller
  - Upgrade Atomix independently of ONOS
  - Introduce new primitives/operations prior to controller upgrade

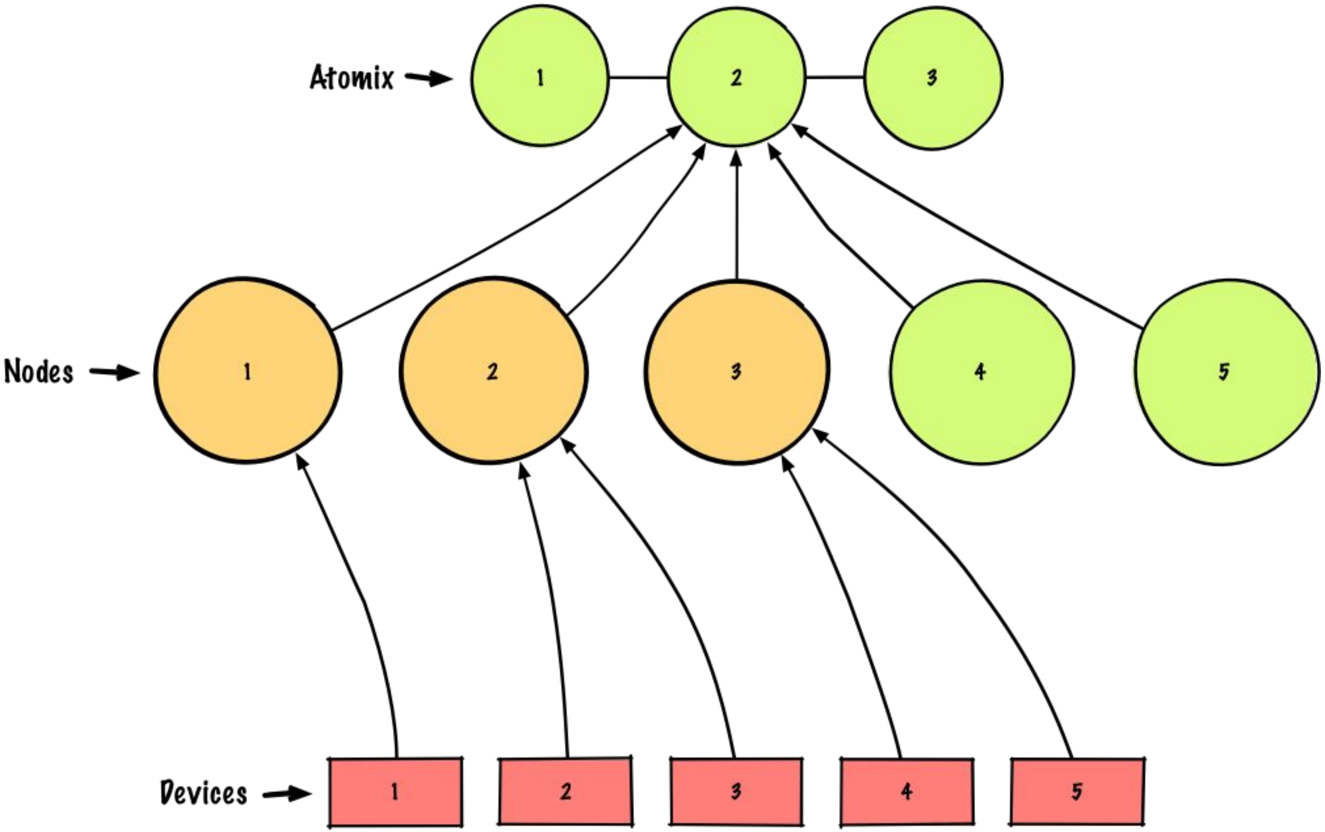
Introducing New Features



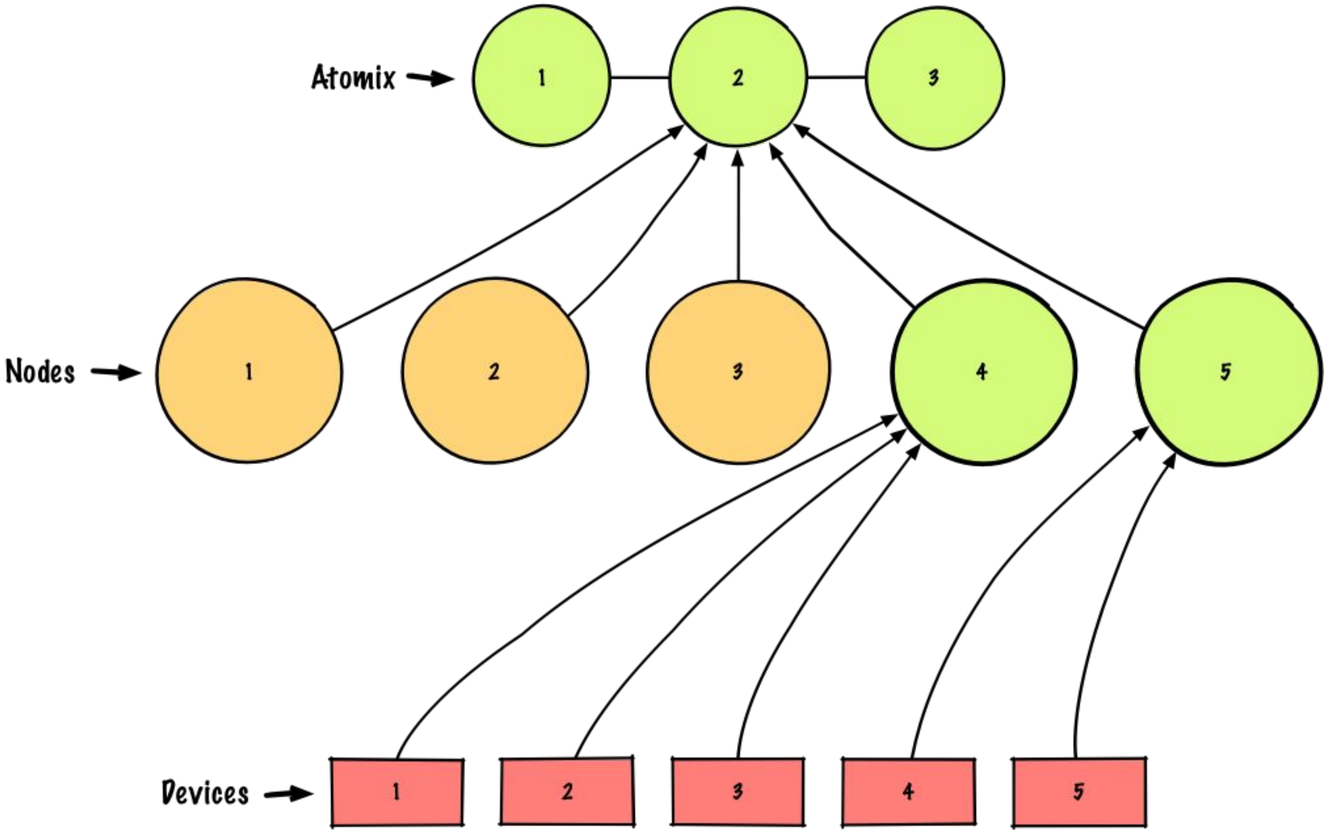
Introducing New Features



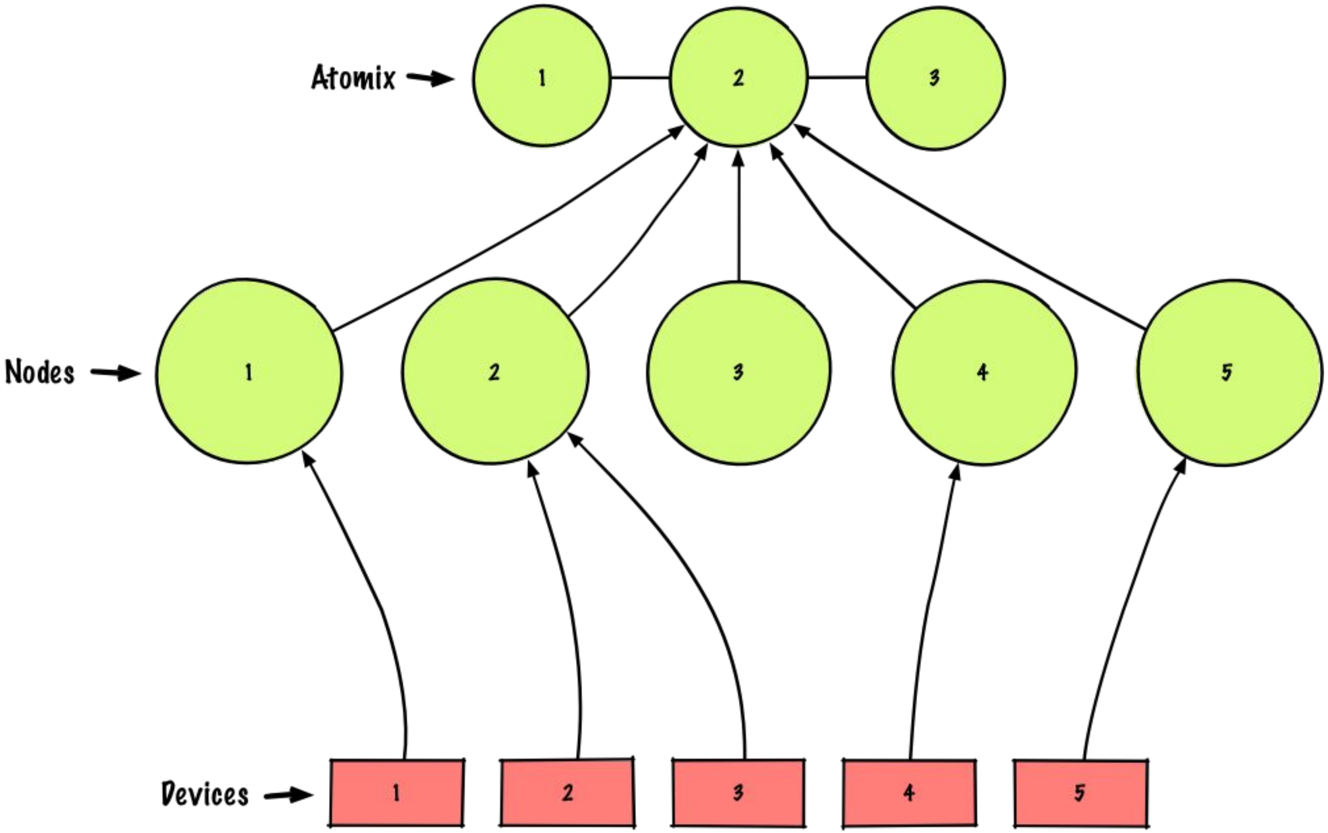
Introducing New Features



Introducing New Features



Introducing New Features



Contributing

- ONOS ISSU Brigade Wiki: https://wiki.onosproject.org/display/ONOS/ISSU
- Google Group: https://groups.google.com/a/onosproject.org/forum/#!forum/brigade-issu
- Meetings weekly on Tuesdays