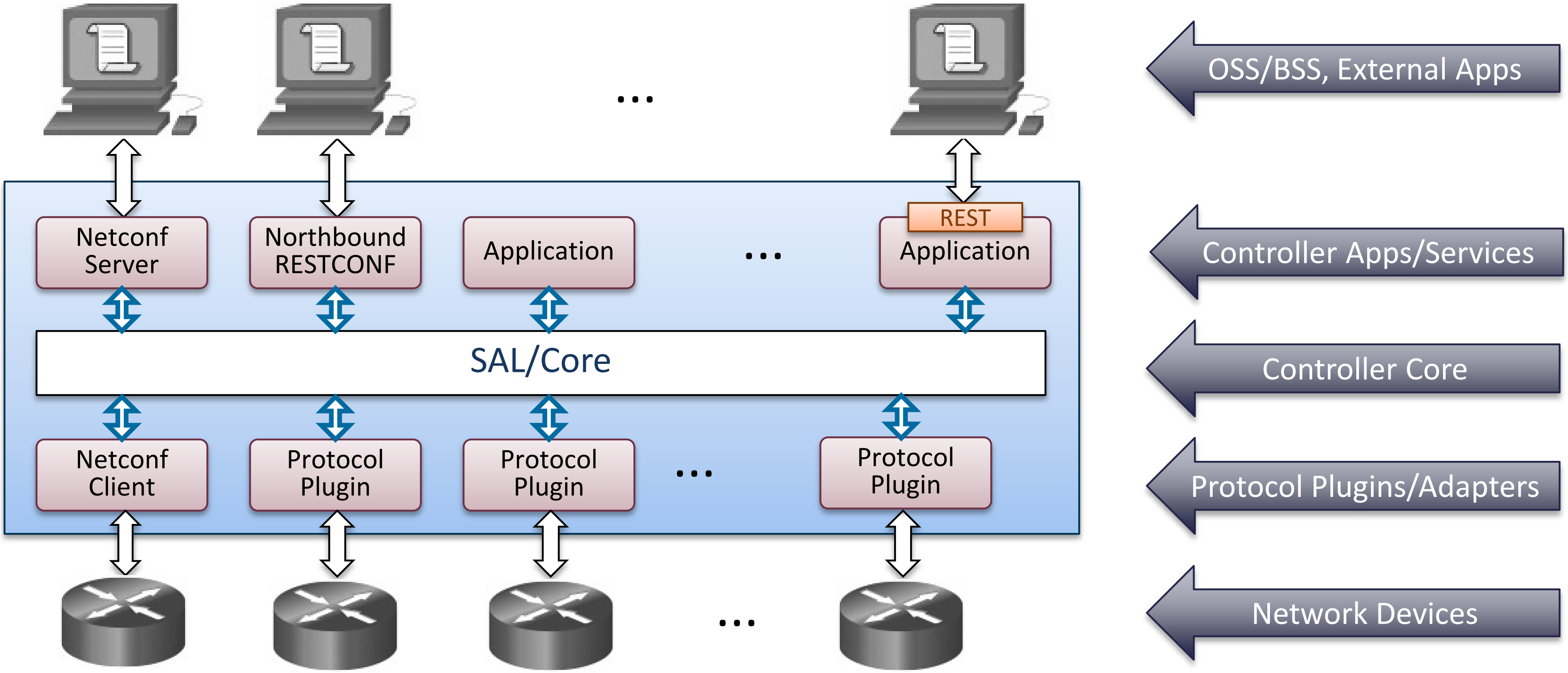
OpenDaylight YANG Data Store: A High-Performance Data Store for SDN and IoT Applications

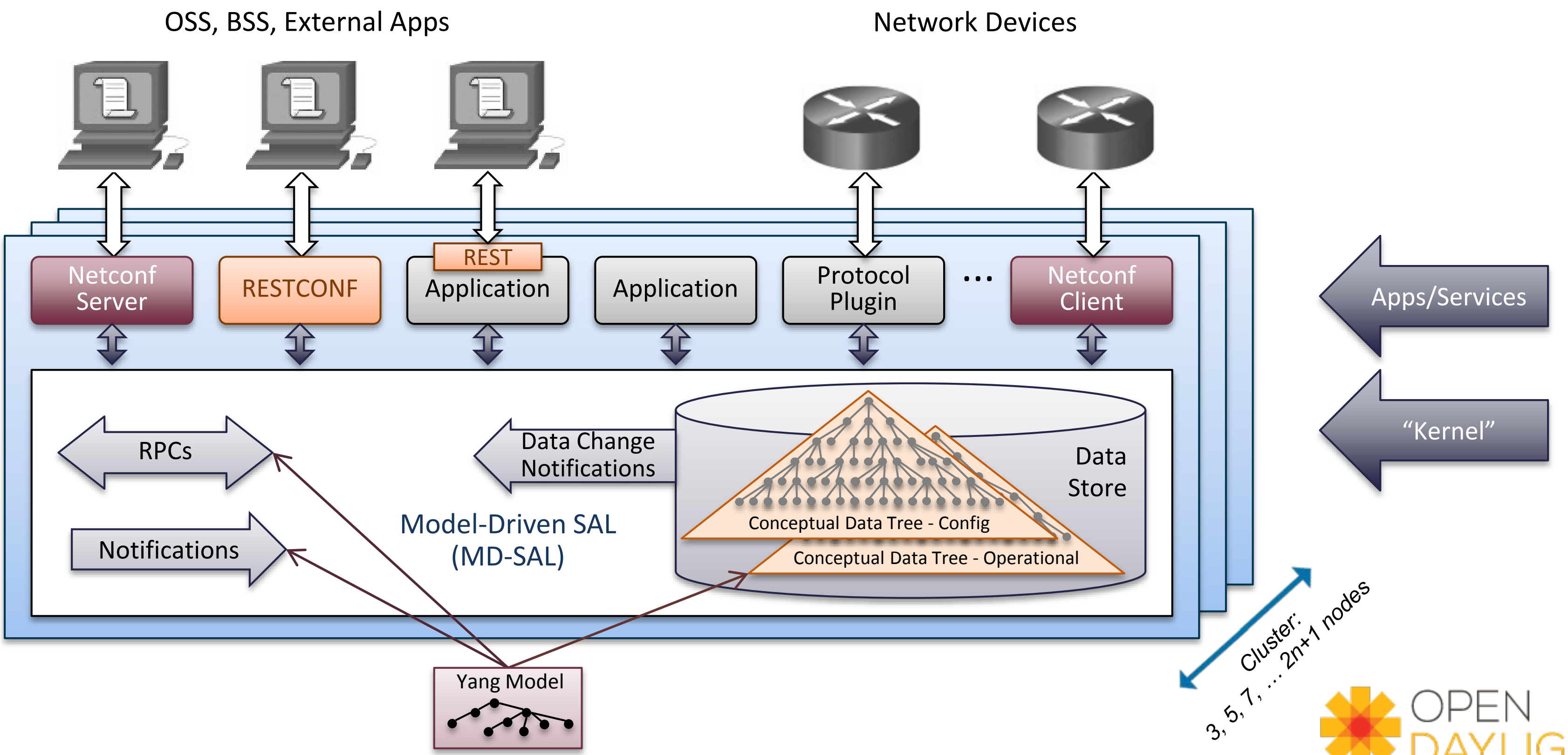
Jan Medved

With contributions from Robert Varga, John Burns, Kun Chen, Branislav Janosik

OpenDaylight is an SDN Controller, Right?



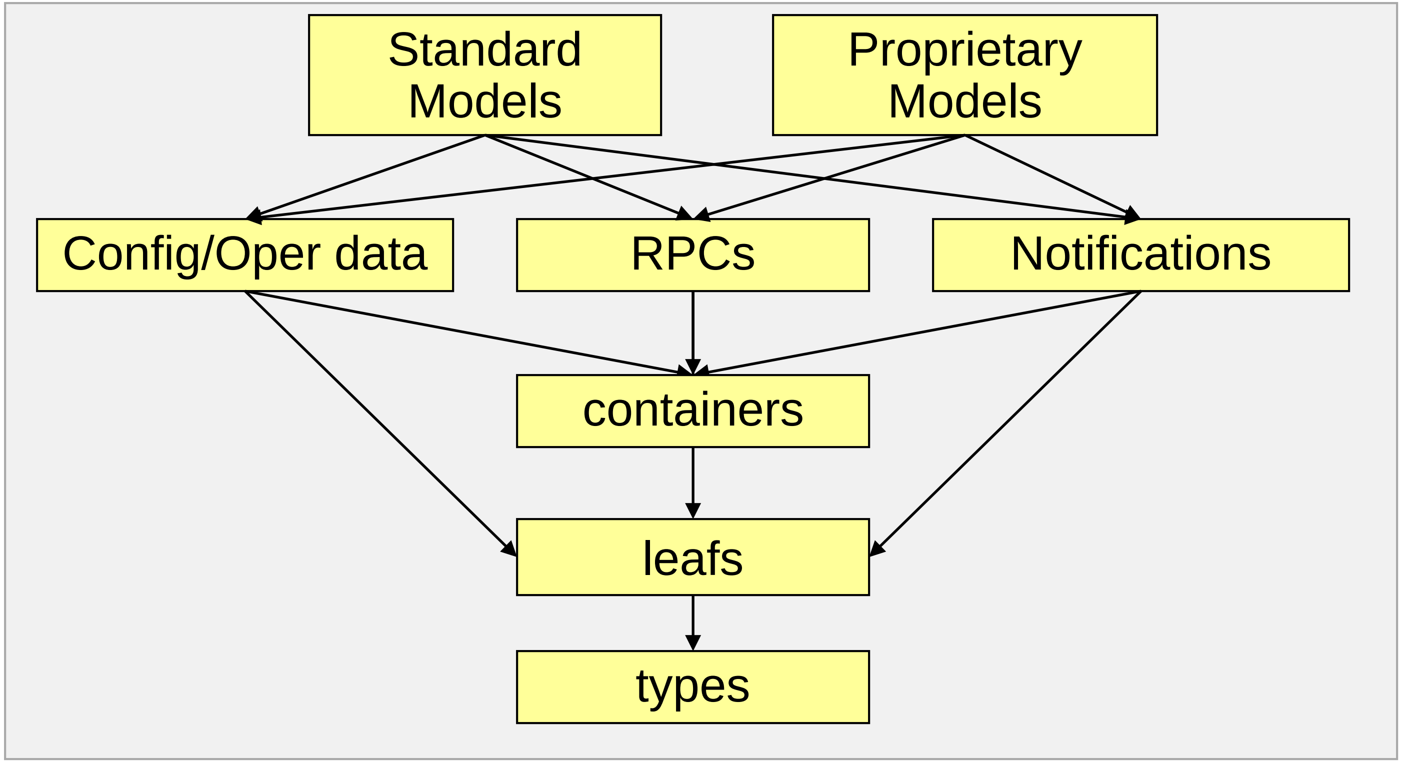
ODL Architecture: an Application Development Platform



YANG is ….

- A NETCONF modeling language
  – Think SMI for NETCONF
- Models semantics and data organization
  – Syntax falls out of semantics
- Able to model config data, state data, RPCs, and notifications
- Based on SMIng syntax
  – Text-based
    - Email, patch, and RFC friendly
- *Also used a Interface Description Language in OpenDaylight*

YANG Concepts



YANG ....

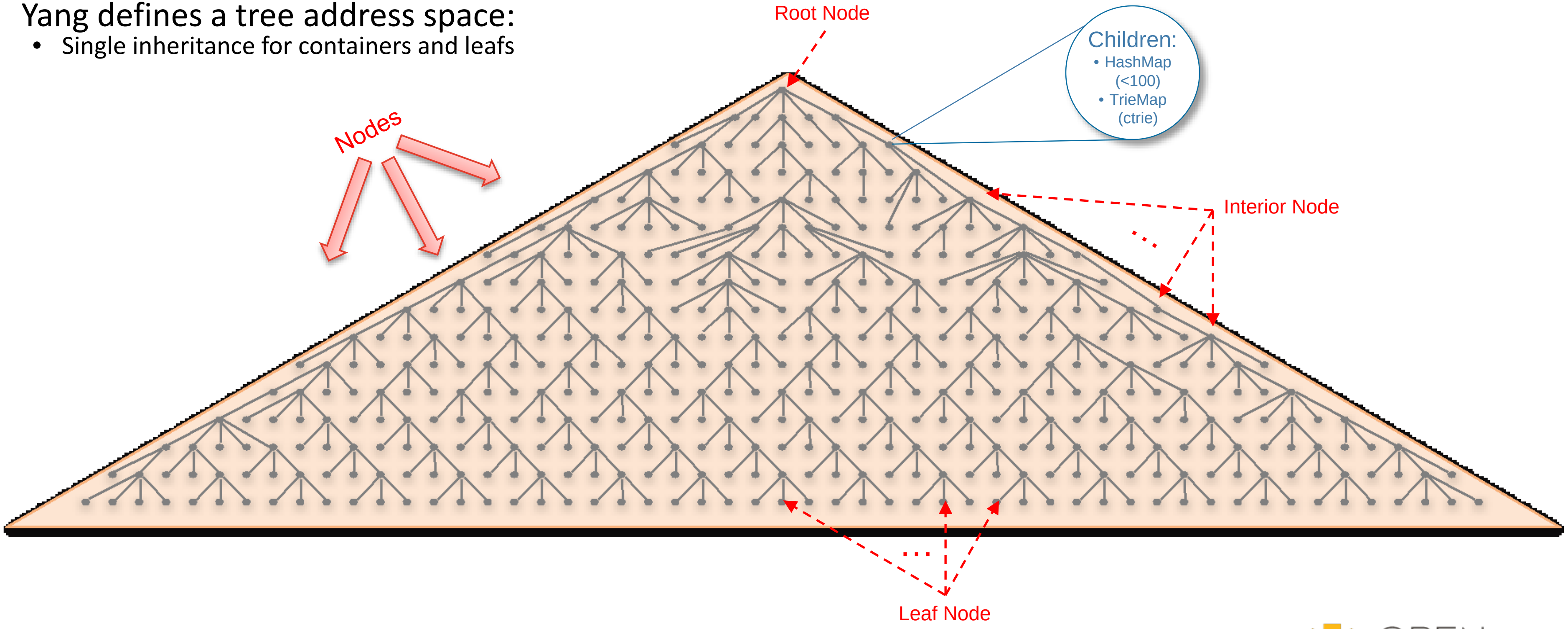
- Directly maps to XML or JSON content on the wire
- Extensible
  – Add new content to existing data models
    - Without changing the original model
  – Add new statements to the YANG language
    - Vendor extensions and future proofing
- Tools in OpenDaylight to generate Java Code from yang models
  – Compile and runtime
- See tools at www.yang-central.org

OPEN DAYLIGHT

Yang Data Store Challenge: Conceptual Data Tree

Yang defines a tree address space:

- Single inheritance for containers and leafs



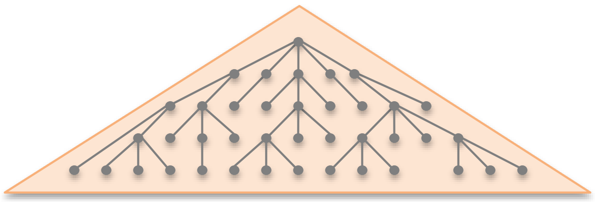
OPEN DAYLIGHT

The OpenDaylight Data Store…

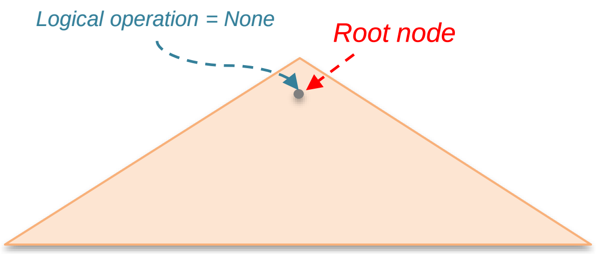
- Implements the equivalent of a W3C DOM Document tree
  – No parent axis maintained -> efficient copy-on-write snapshot
- MVCC-based atomic updates in a single-writer, multiple-readers environment
  1. Modifications prepared concurrently
  2. Single update thread:
     - Requested mods are applied and and made visible to subsequent snapshots
  3. Efficient physical replication
- Enforcement of structural integrity according to yang schemas

Data Store Operation:
Step 1- Transaction Created

Initial snapshot when transaction created



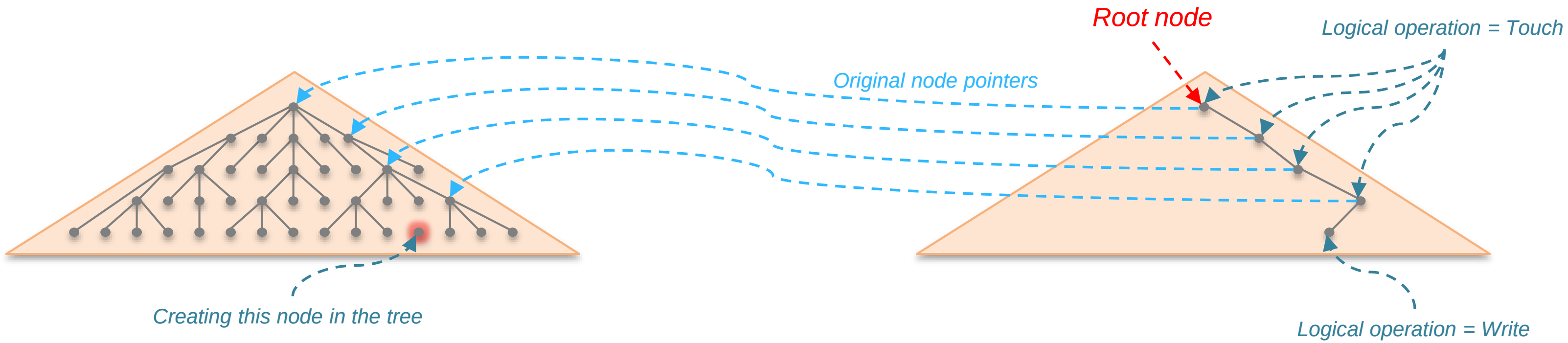
Modification tree (populated as mods in the transaction are performed)



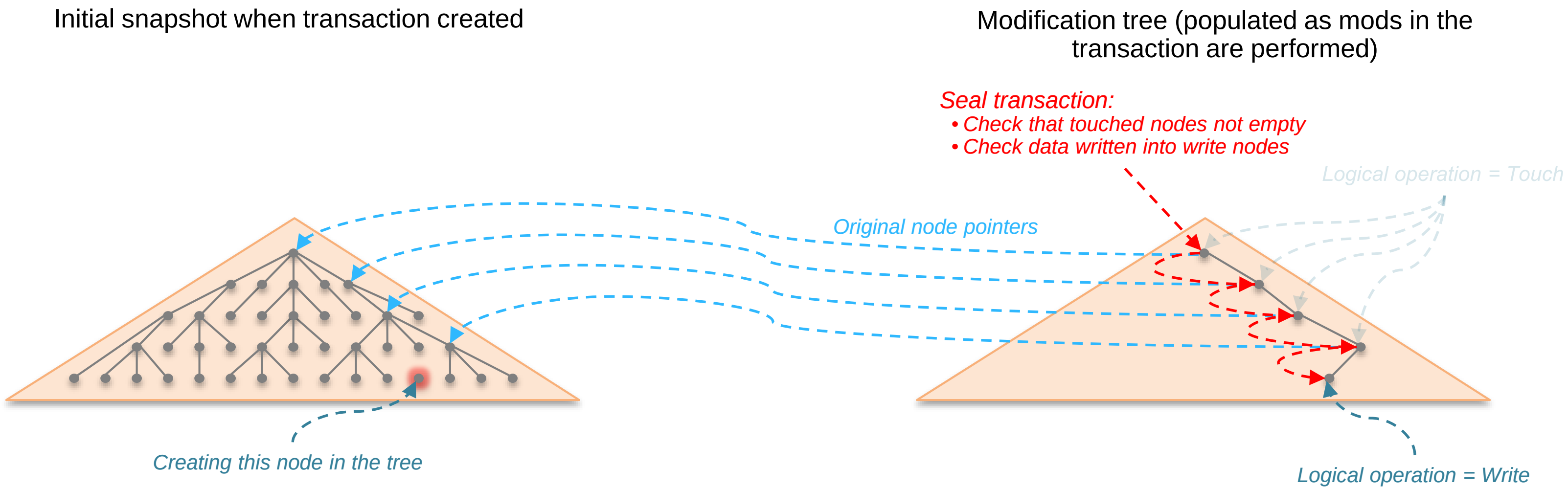
Data Store Operation:
Step 2 - Creating a New Node (Write)

Initial snapshot when transaction created

Modification tree (populated as mods in the transaction are performed)



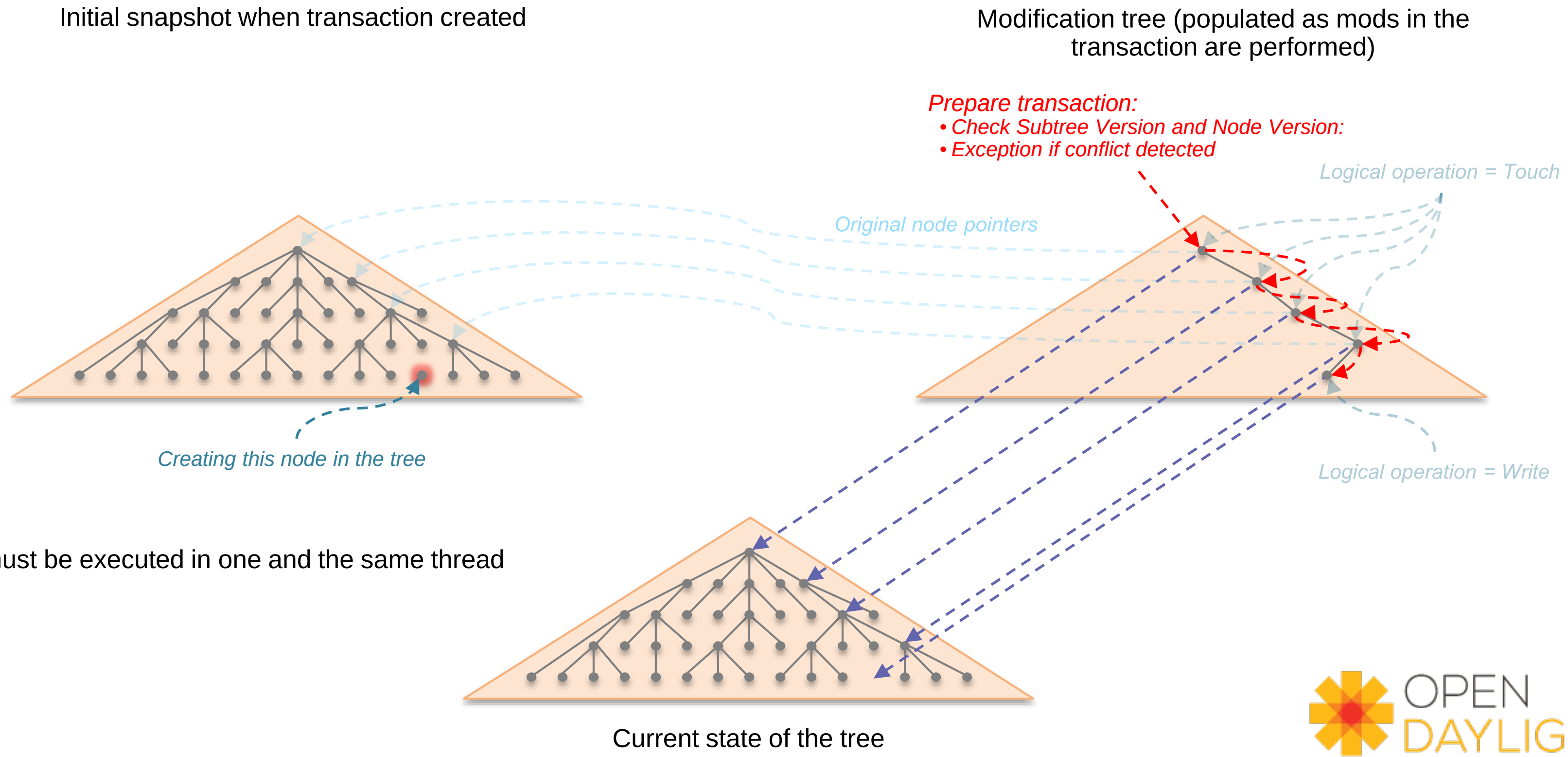
Data Store Operation:
Step 3 - Transaction Seal (Ready)



Steps 2 and 3 can run concurrently in multiple threads

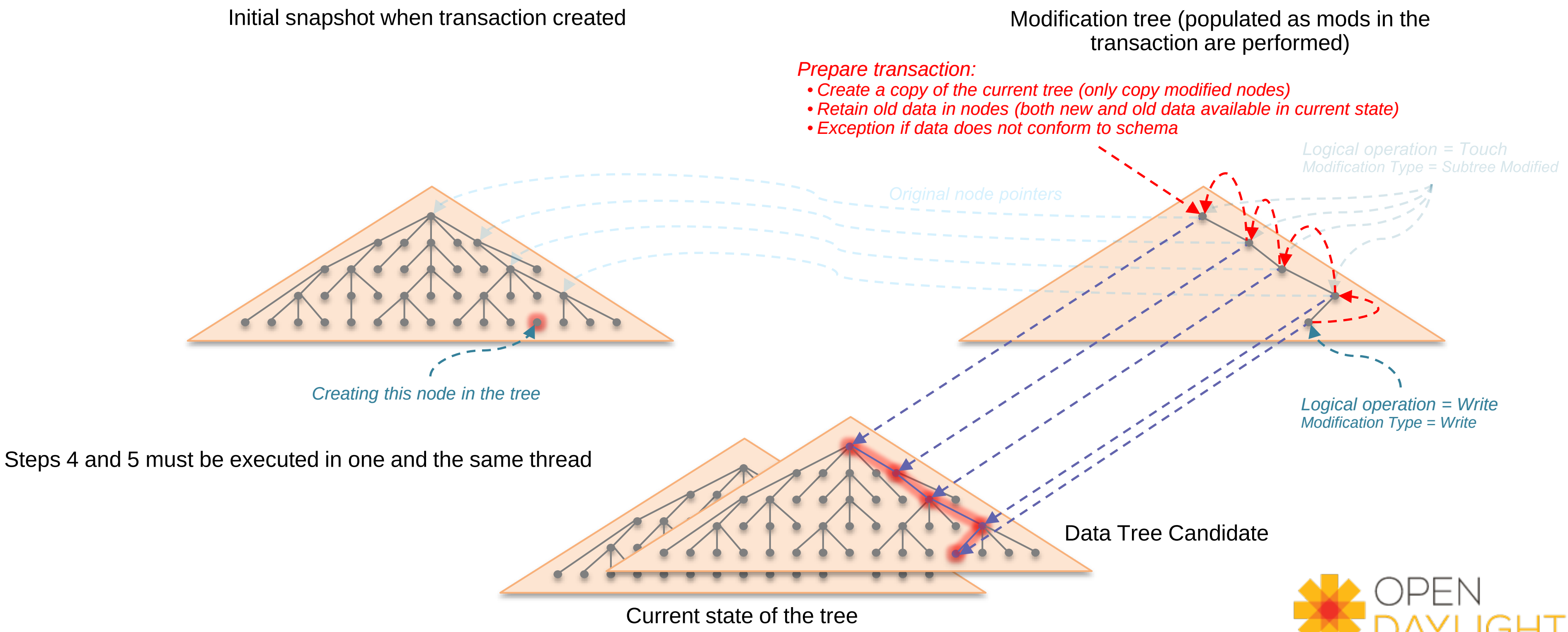
OPEN DAYLIGHT

Data Store Operation:
Step 4a - Transaction Prepare (Walk Down)



Steps 4 and 5 must be executed in one and the same thread

Data Store Operation:
Step 4b -Transaction Prepare (Walk Up)



OPEN DAYLIGHT

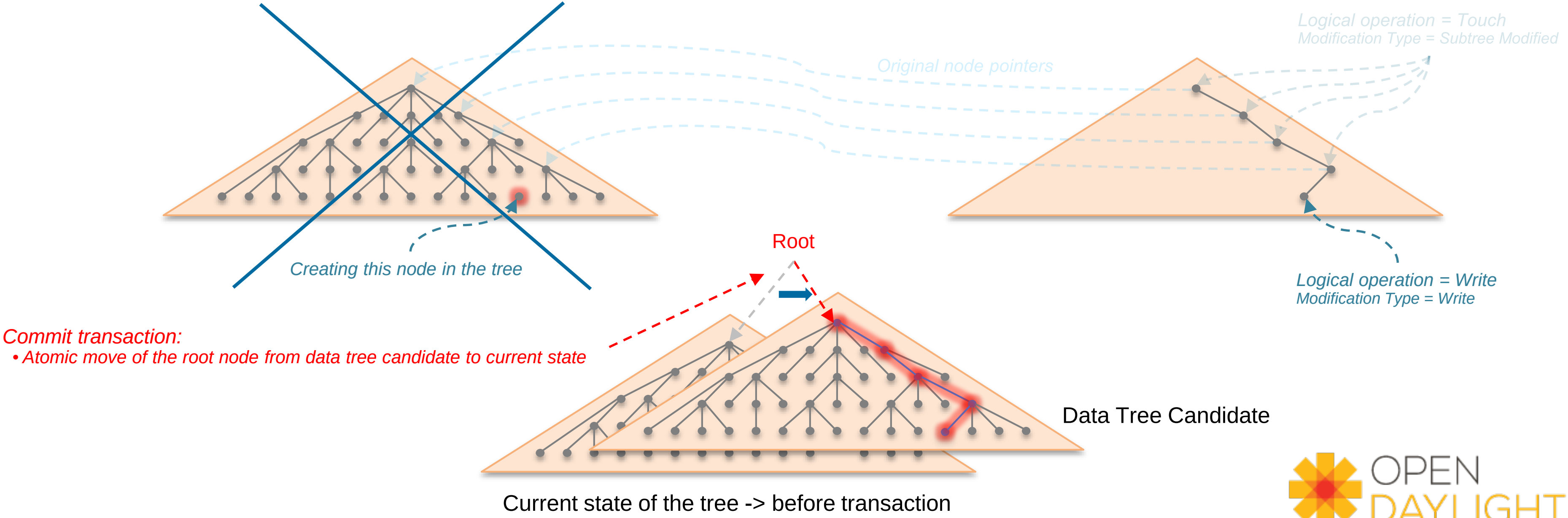
Data Store Operation: Step 5: Commit

Steps 4 and 5 must be executed in one and the same thread

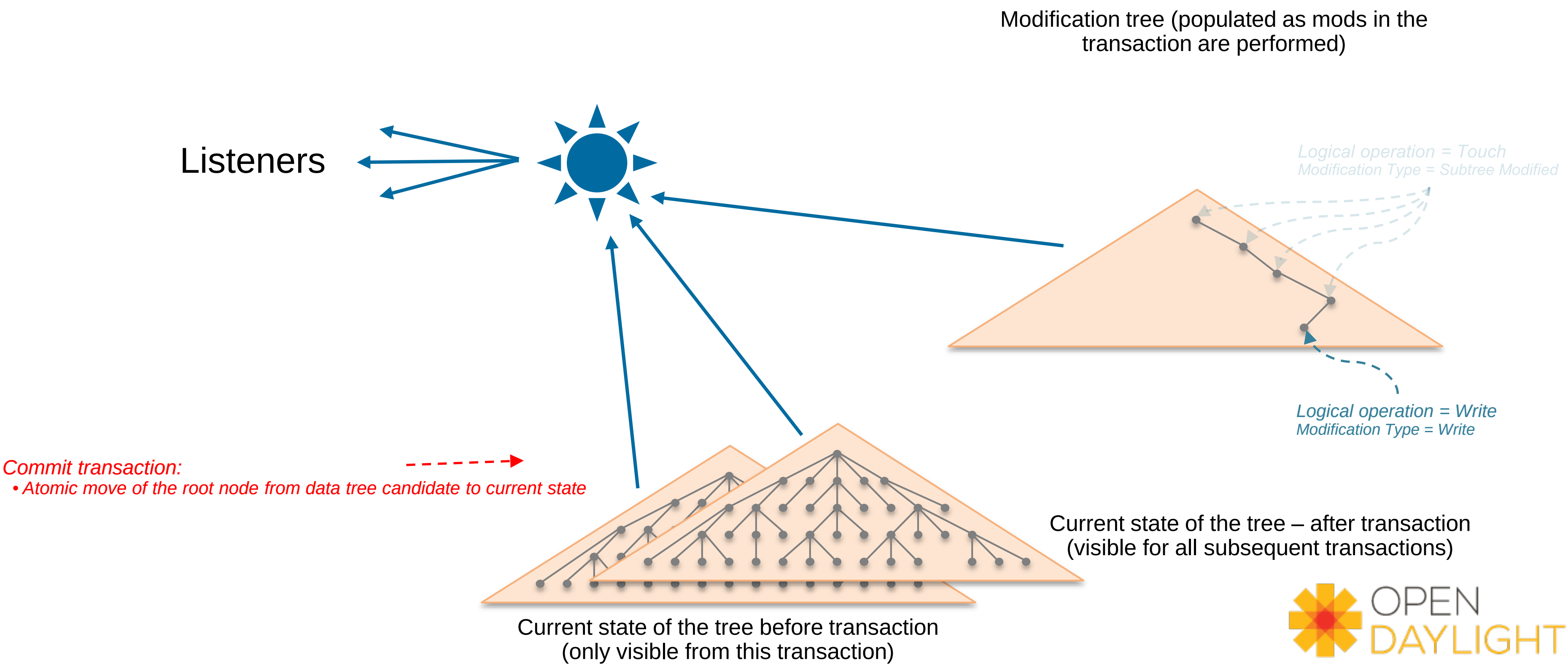
Initial snapshot when transaction created:

- If commit successful, only used in transactions that were created on top of the just committed transaction
- will be garbage collected if not needed anymore

Modification tree (populated as mods in the transaction are performed)



Step 6: Notify Listeners

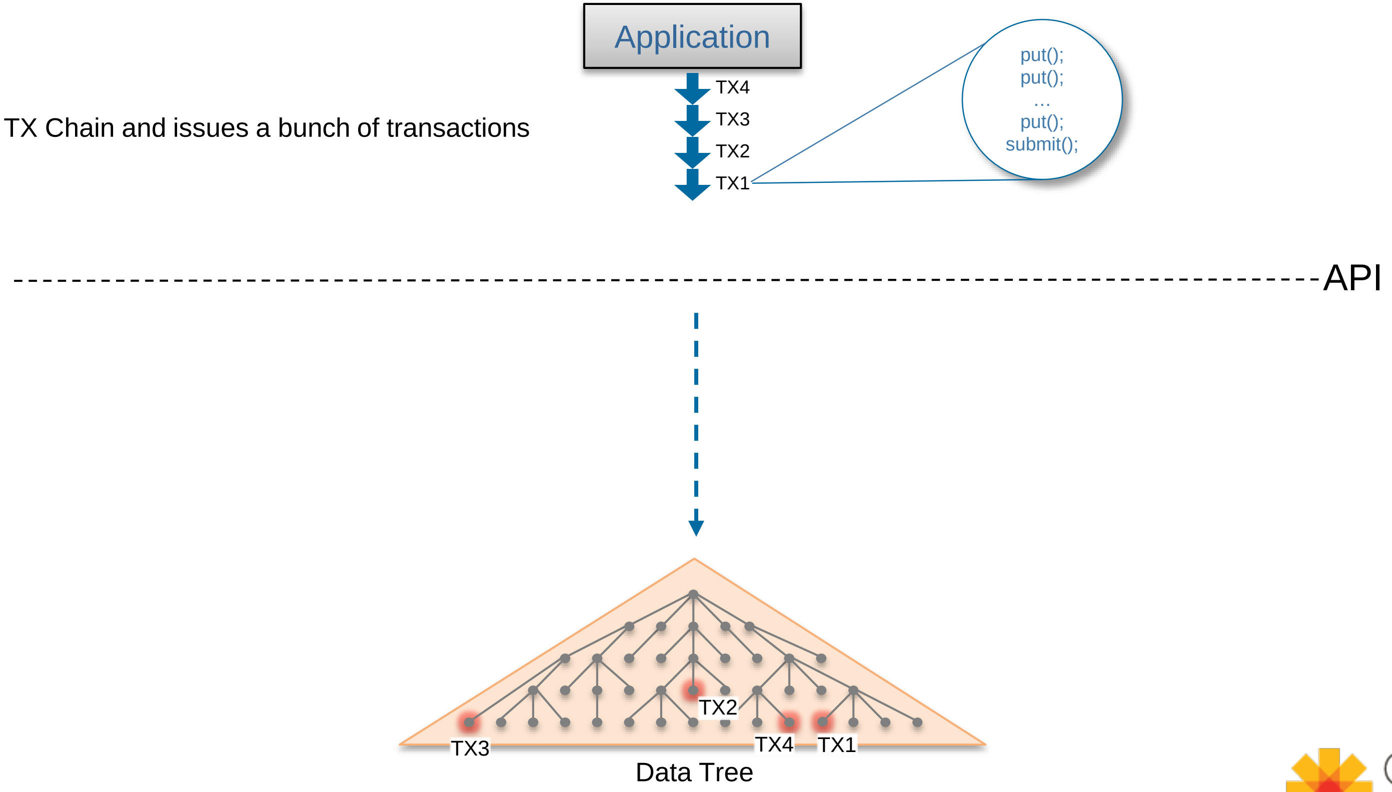


How to Improve Transaction Performance

- State Compression
  - Multiple fine-grained transactions compressed into one bigger transaction
    - “Fate sharing” -> faster success path, but longer recovery
  - Transaction Chaining with ping-pong buffer
    - Single writer
- Sharding (Parallelism)

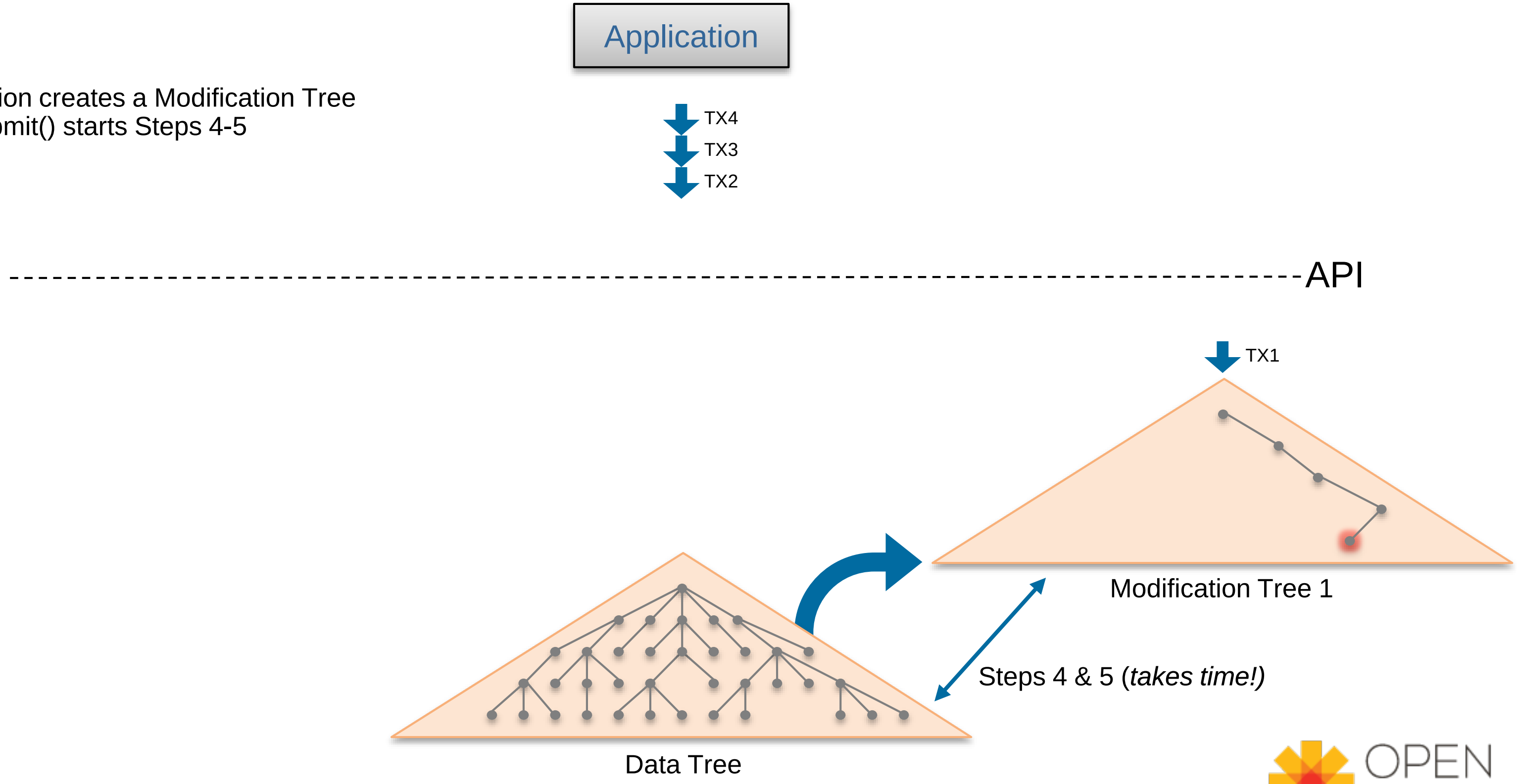
Transaction Chaining with Ping-Pong Buffer

1. App creates a TX Chain and issues a bunch of transactions



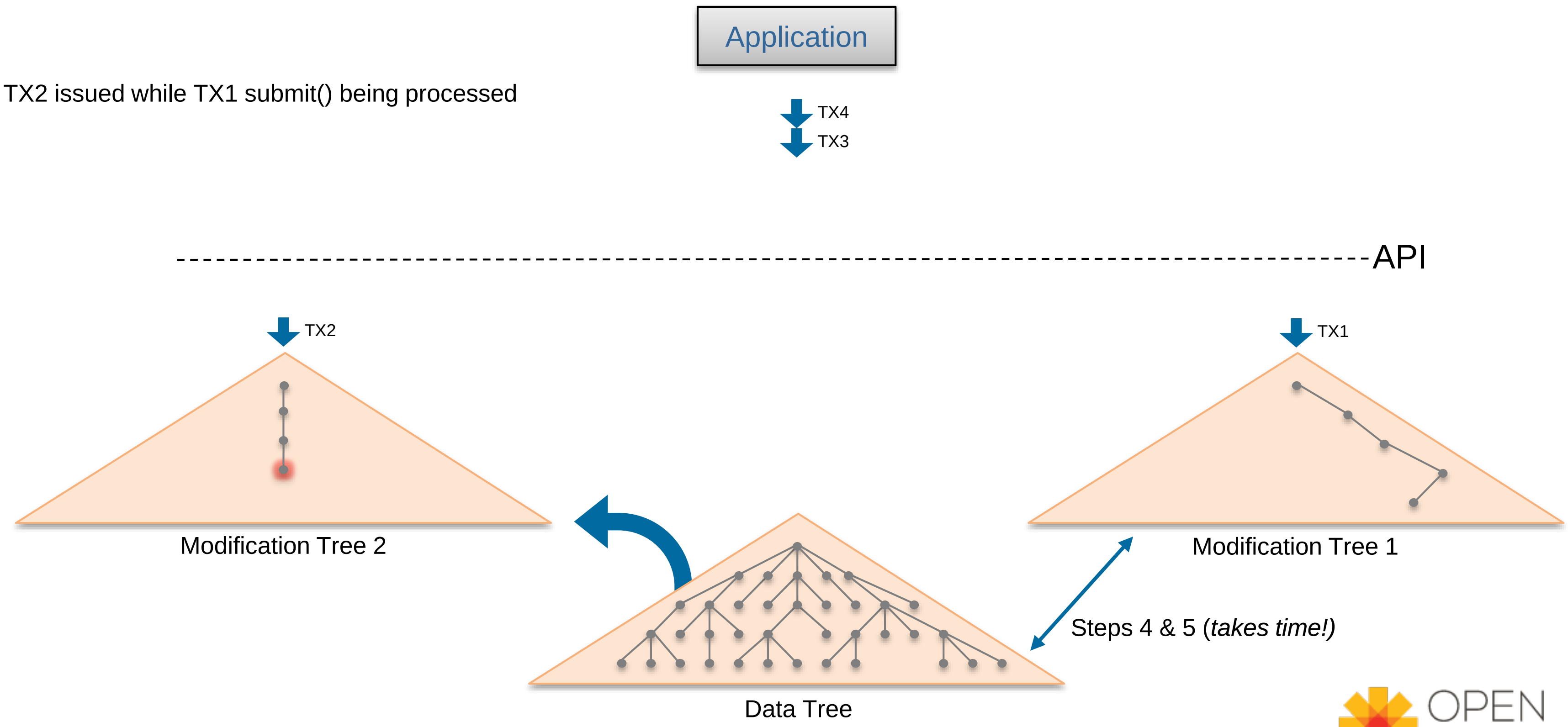
Transaction Chaining with Ping-Pong Buffer

2. First transaction creates a Modification Tree
   - submit() starts Steps 4-5



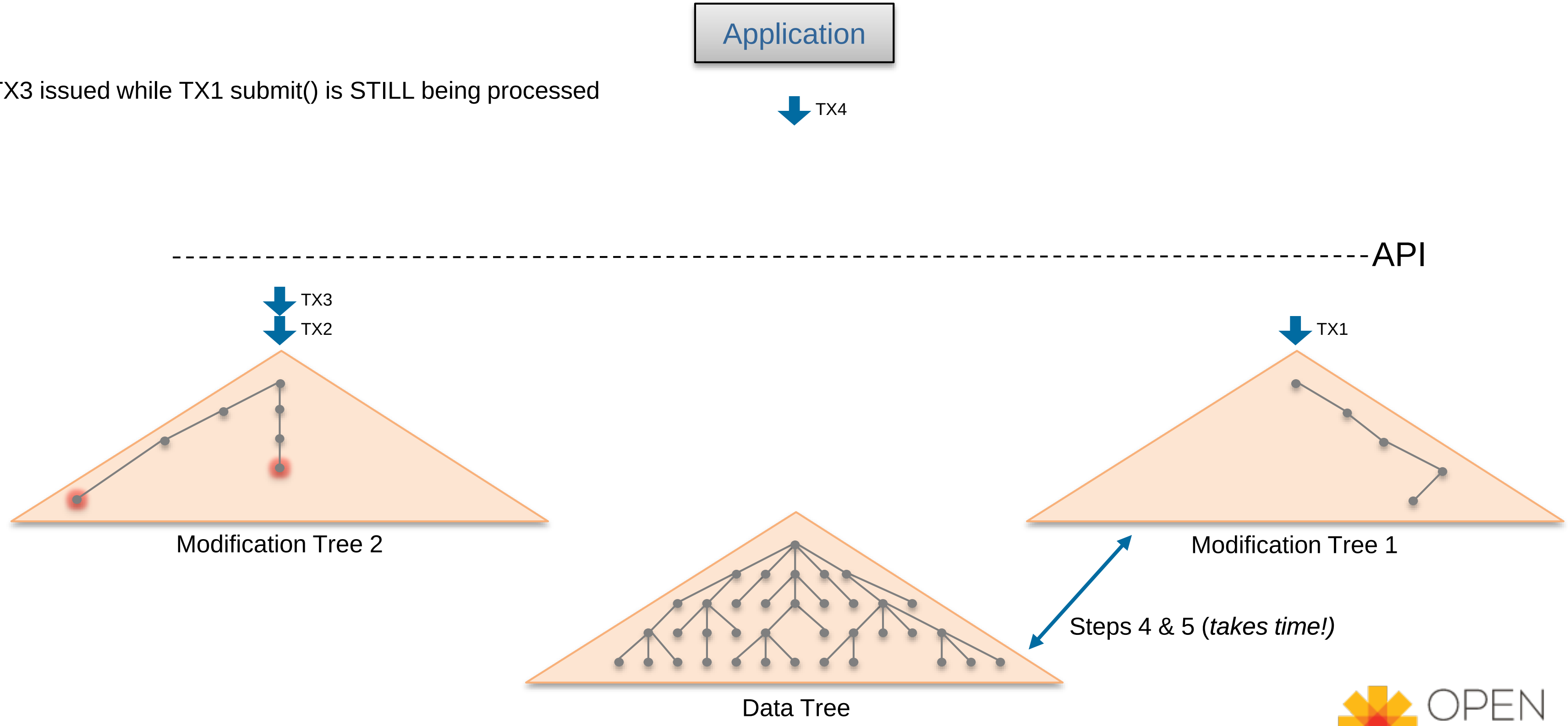
Transaction Chaining with Ping-Pong Buffer

3. TX2 issued while TX1 submit() being processed



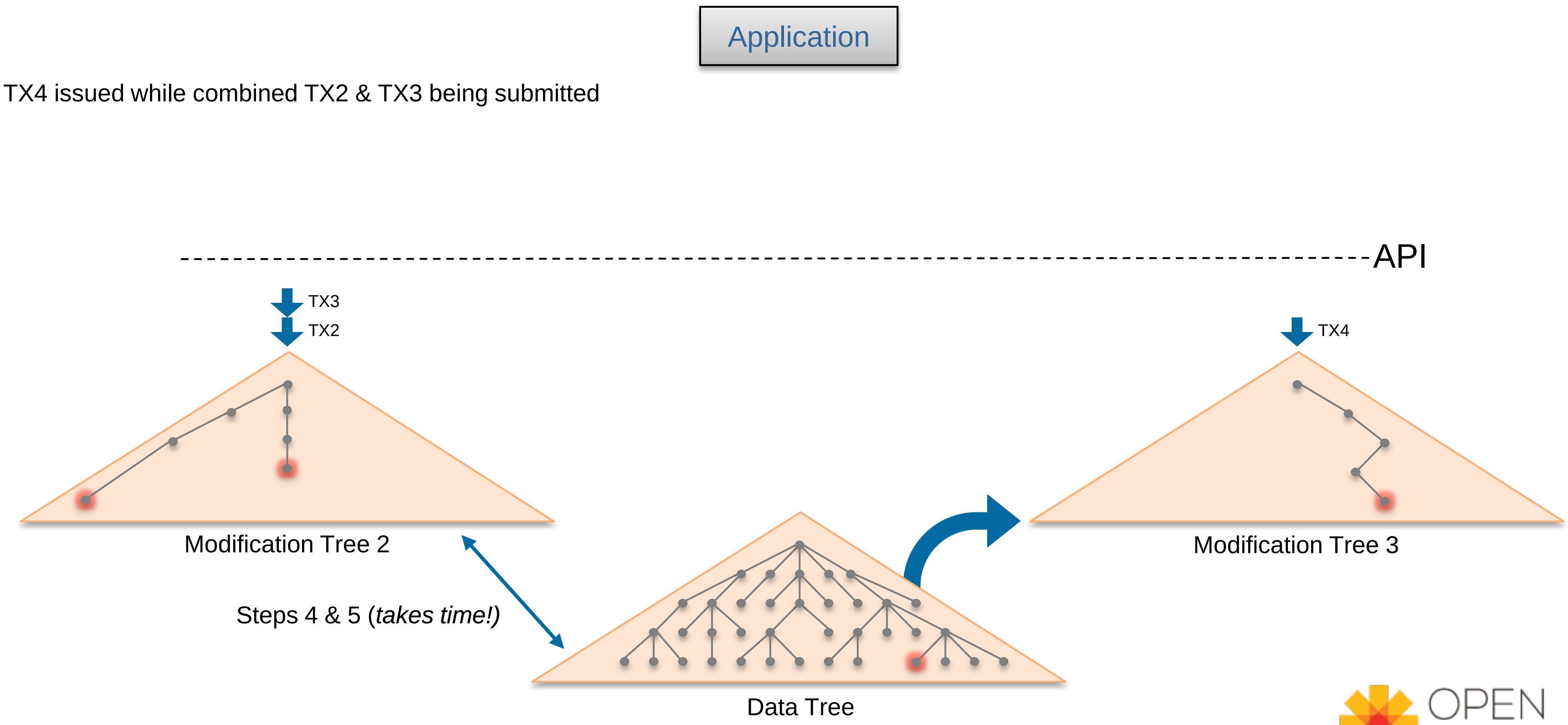
Transaction Chaining with Ping-Pong Buffer

3. TX3 issued while TX1 submit() is STILL being processed

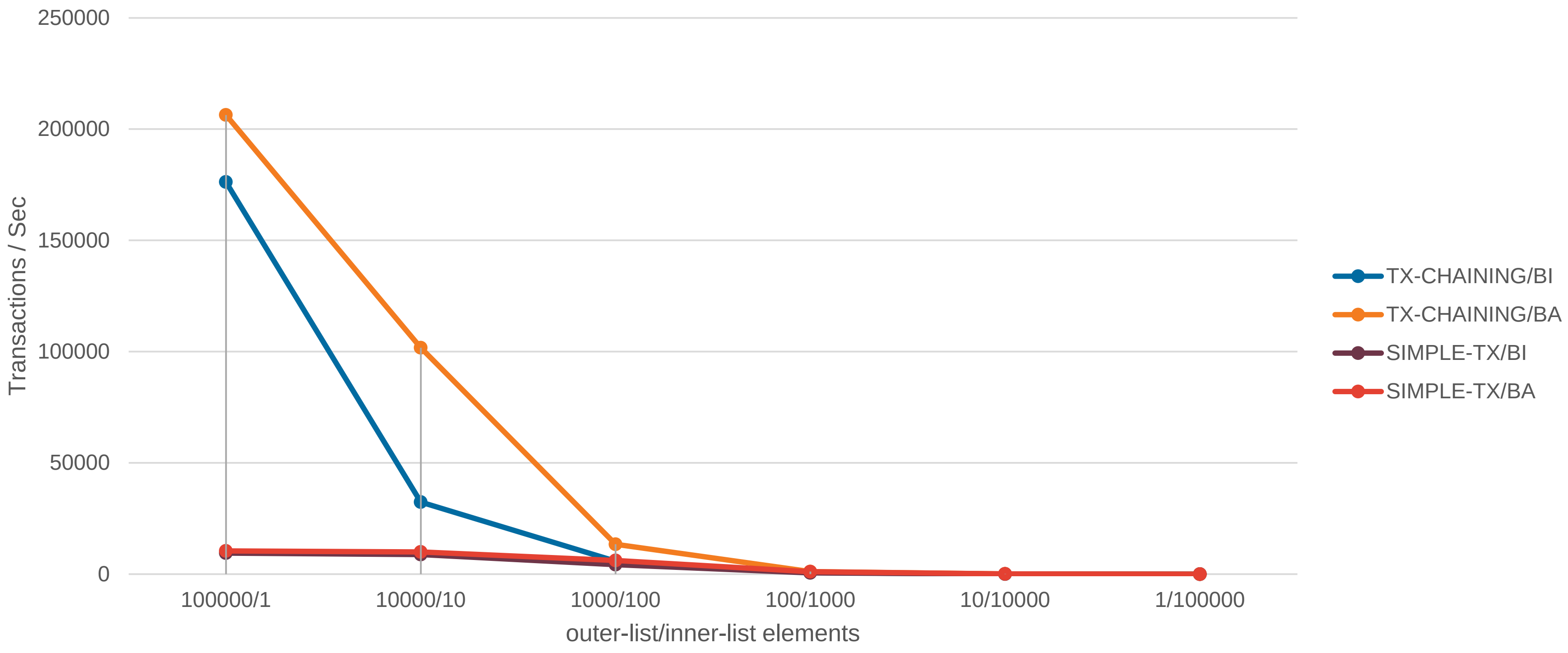


Transaction Chaining with Ping-Pong Buffer

4. TX4 issued while combined TX2 & TX3 being submitted



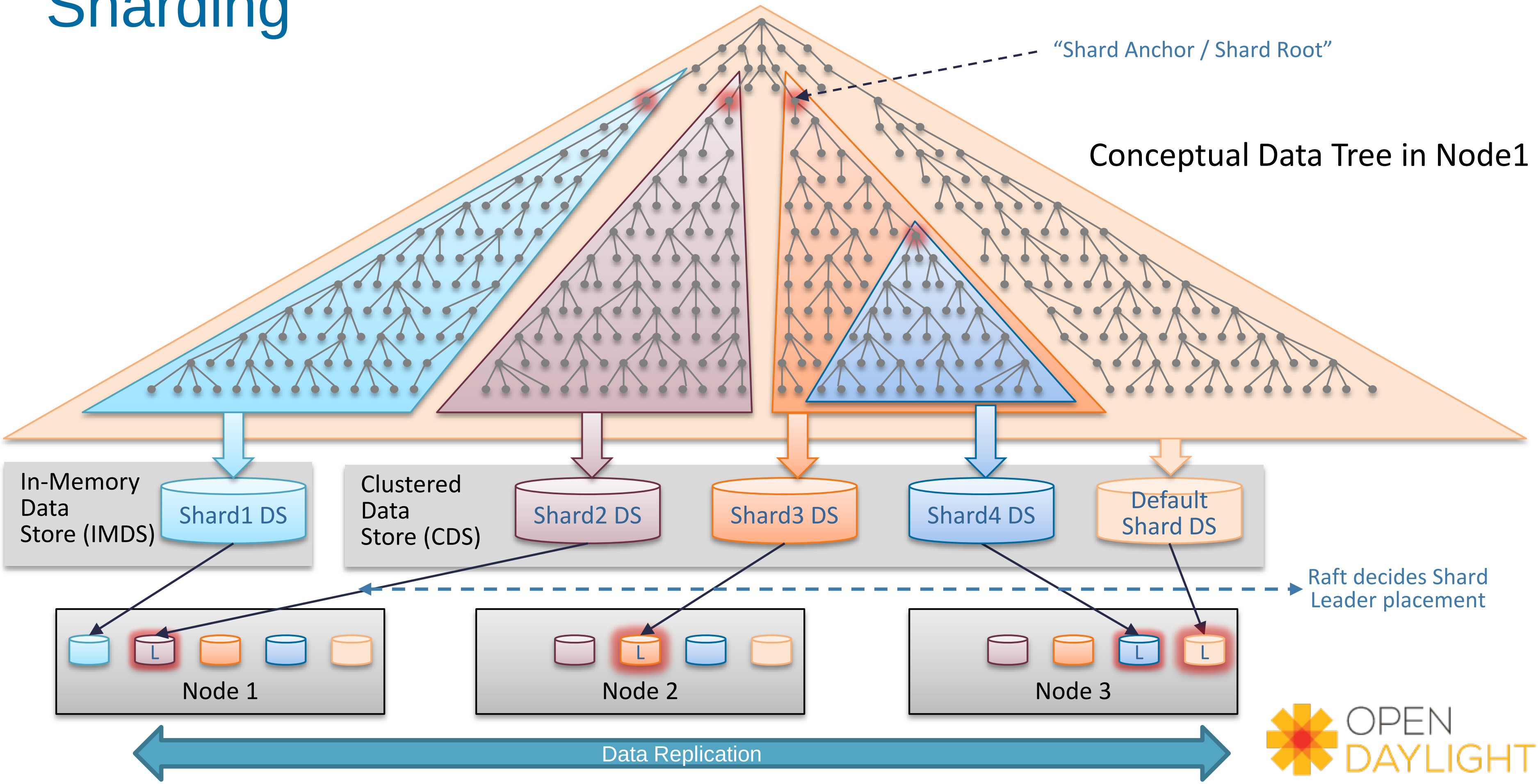
Transaction Chain vs Single Transactions Performance



Test Data: List of lists

OPEN DAYLIGHT

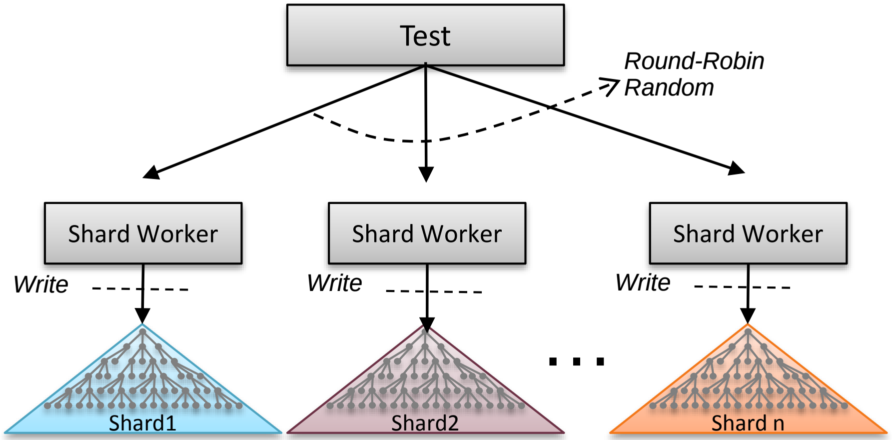
Sharding



OPEN DAYLIGHT

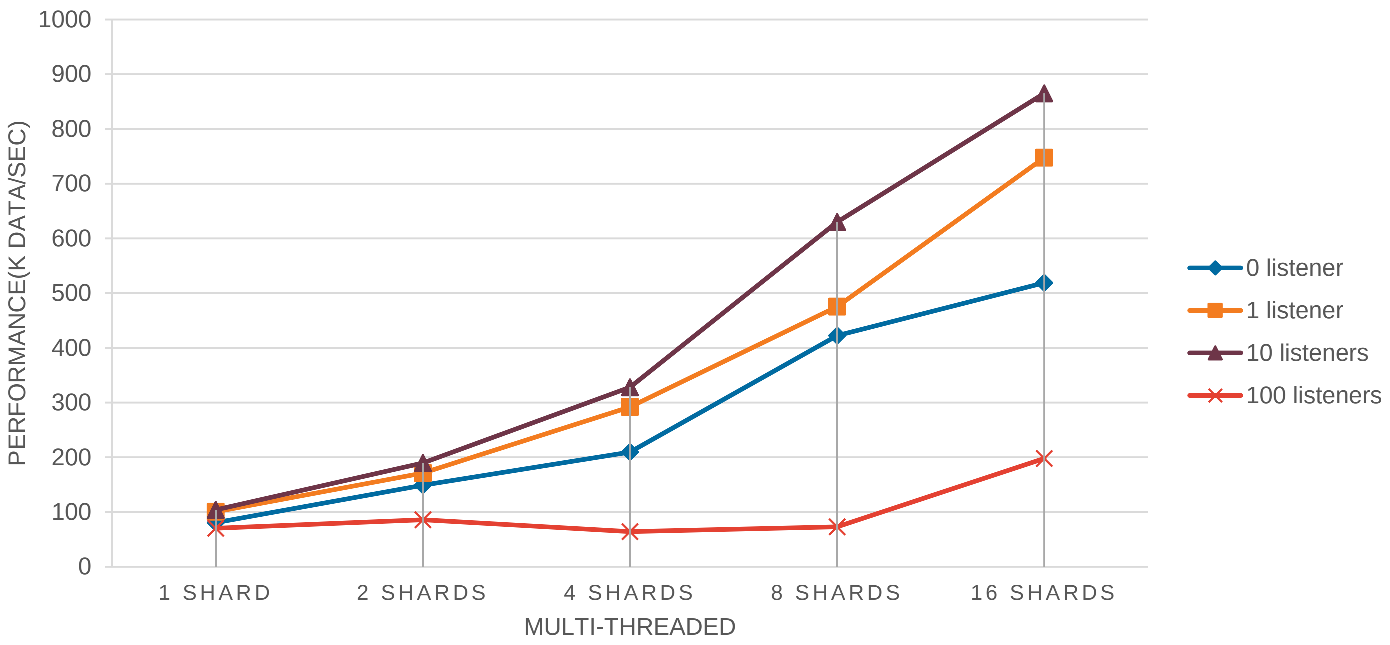
Shard Performance Testing: Test Setup

Multi-Producer Test



- List/Array of x string elements divided into 1 … n shards
- (Test) writes data to Shard Workers (Producers) in Round-Robin or Random fashion
- Worker handles tx responses asynchronously (Futures)

Performance Results: Multiple Producers

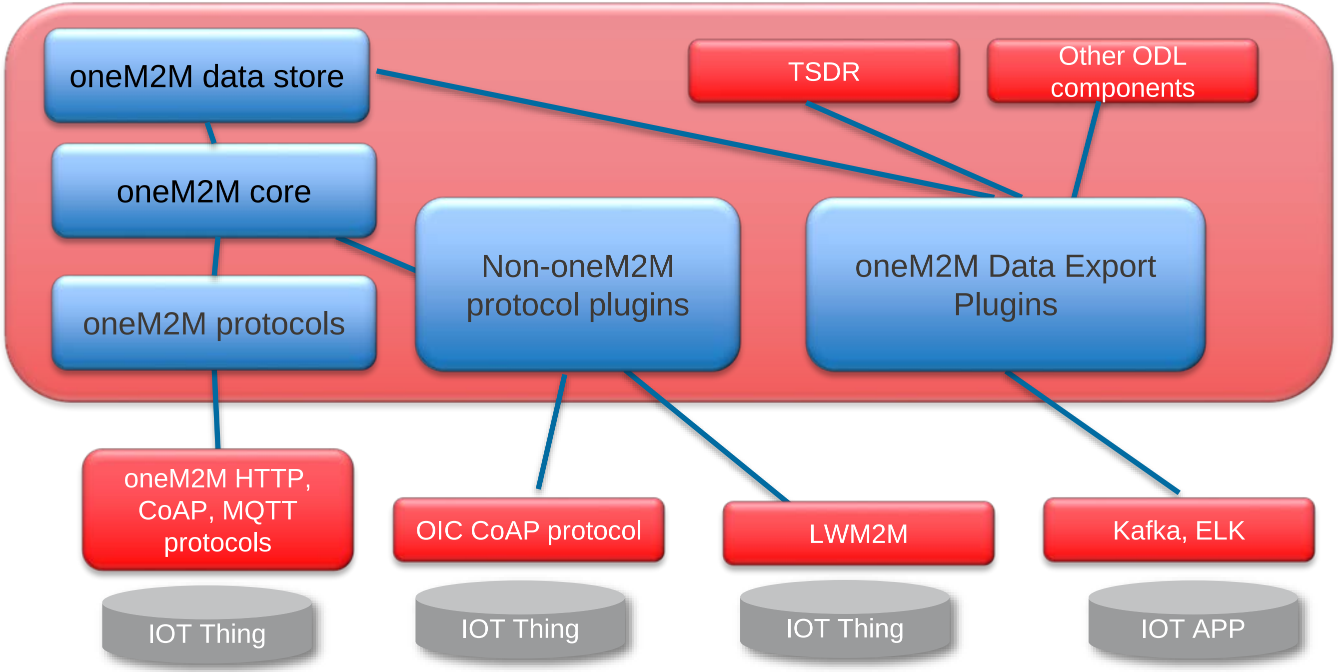


IoTDM

- What is IoTDM?
  - A collection of projects for IOT data management.
    - git clone https://git.opendaylight.org/gerrit/p/iotdm.git
    - https://wiki.opendaylight.org/view/IoTDM:Main
  - Initial IOT standard chosen to implement is onem2m.org
  - Model IOT things and their data in the ODL data store
    - onem2m resources and resource tree modeled generically in the onem2m.yang file
  - Other standards in the future such as
    - OIC, LWM2M, …
    - Initial strategy for other standards is to adapt /interwork the other standards, normalized to onem2m

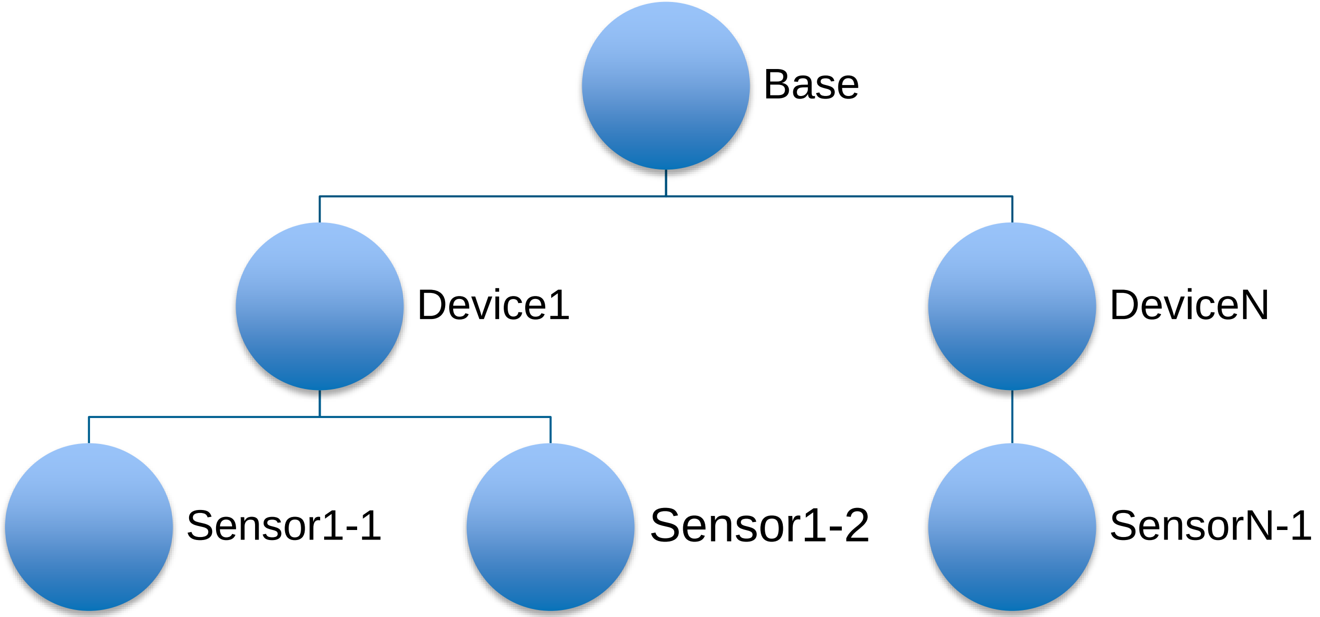
26

IoTDM Architecture



OPEN DAYLIGHT

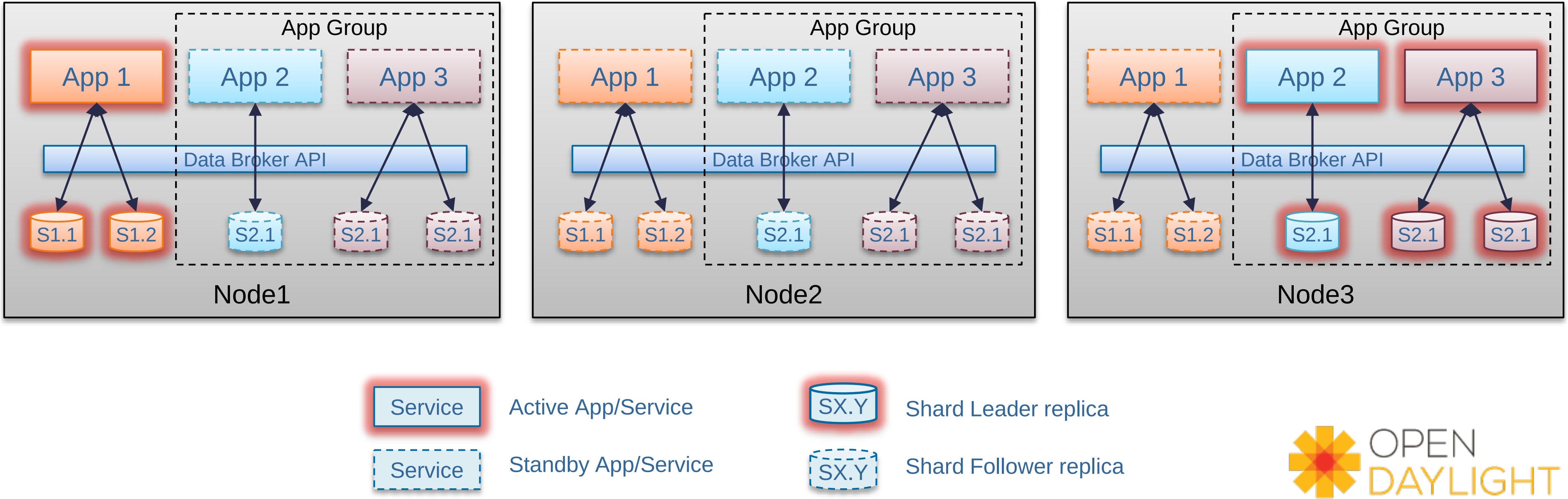
oneM2M Tree for Performance



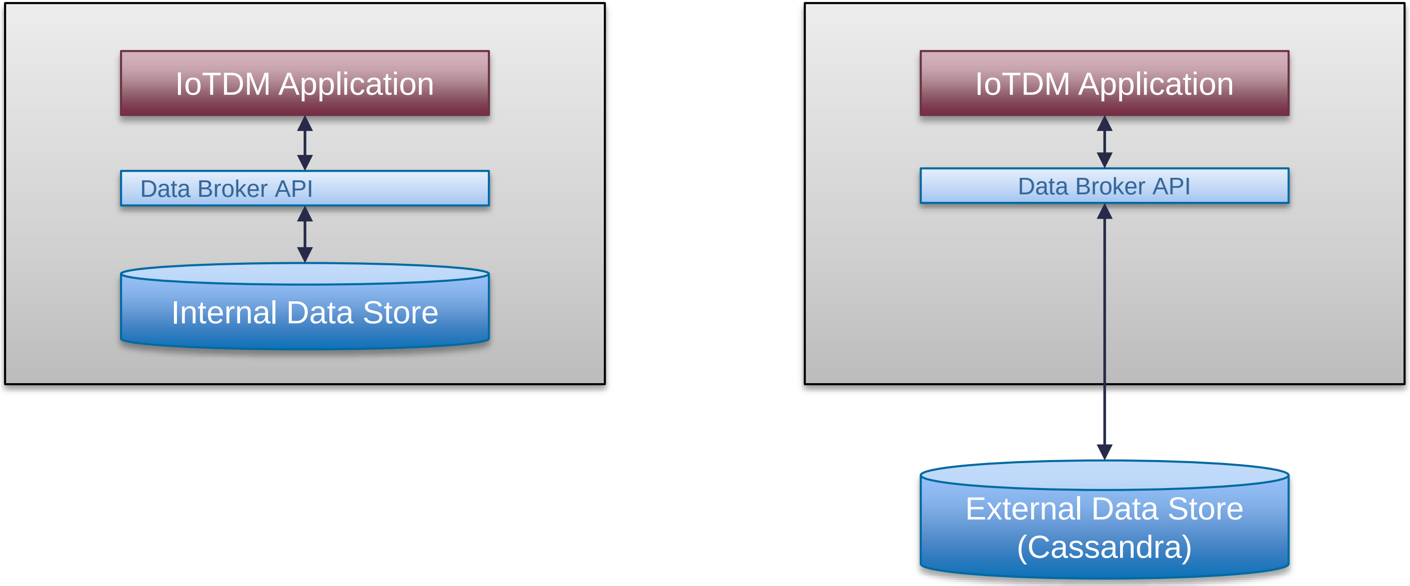
OpenDaylight Application Model

Service/Application:

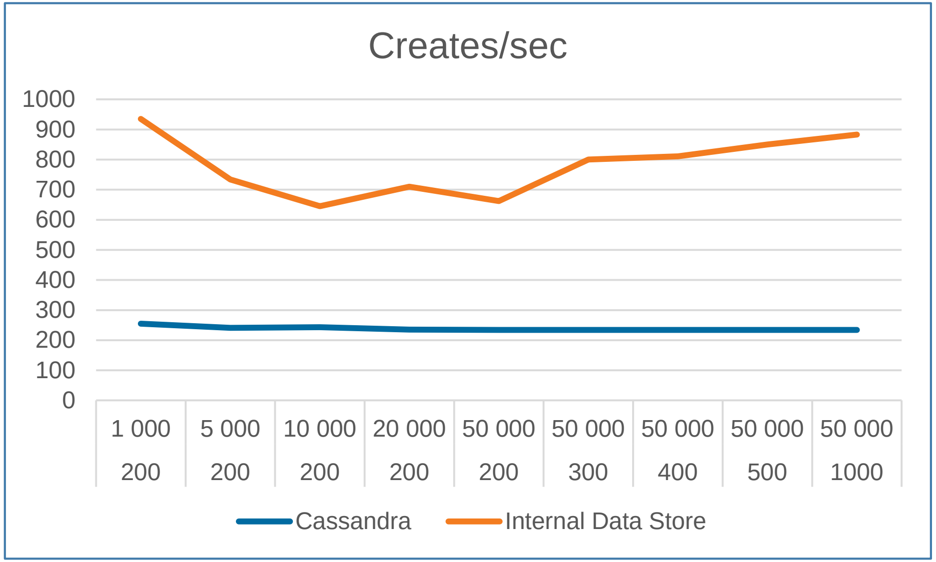
- Code + Data (Shards, subtrees of the Conceptual Data Tree)
- Service/Application instances SHOULD be co-located with shard replicas
- Active Service/Application instance SHOULD be co-located with all Shard Leaders it “owns”
- Apps can be grouped for ”fate sharing

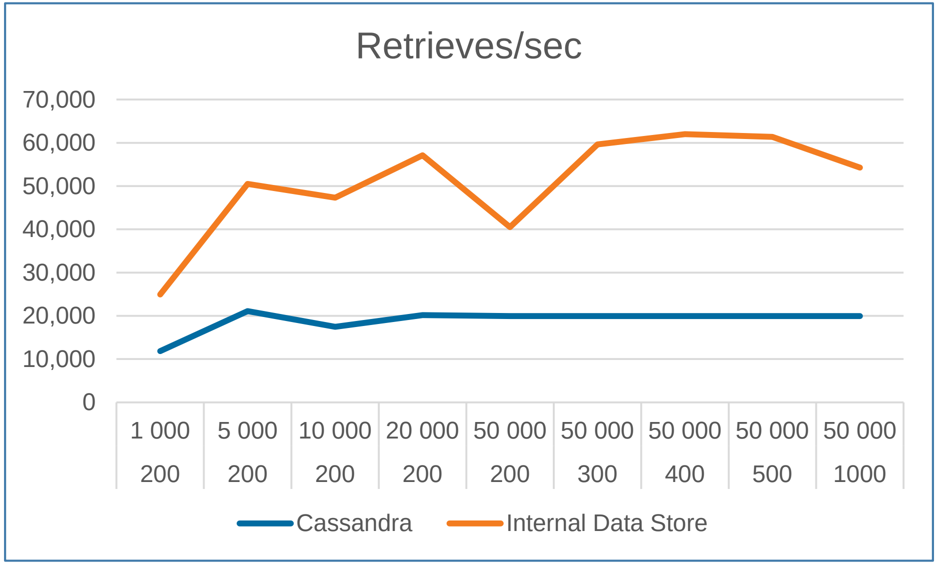


Internal vs. External Data Store



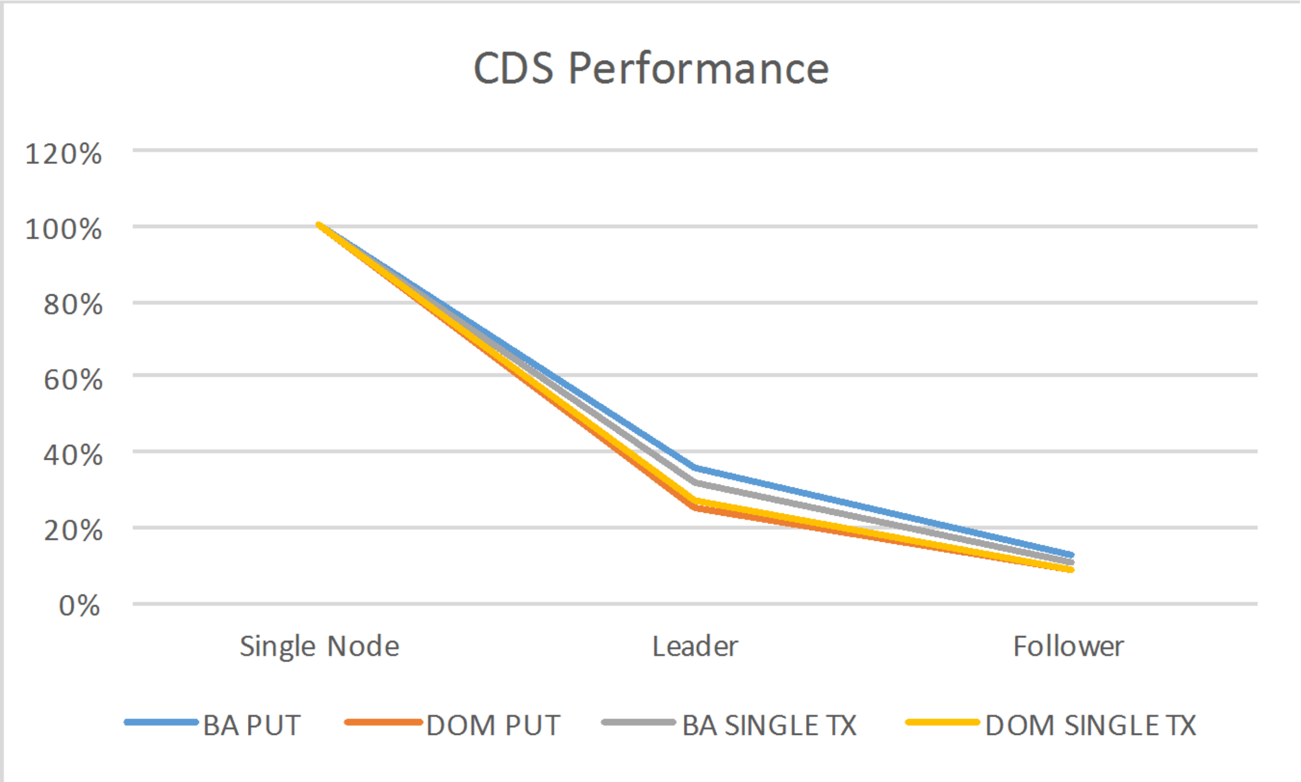
Data Store Performance





Thank you

Performance Results: Impact of Clustering on CDS



OPEN DAYLIGHT

Performance Results: Lessons Learned

- Small transactions are dominated by initialization cost, charged to producer
  - Affects a single thread's ability to saturate backend
- Batching effectiveness goes up with backend latency
  - Many listeners, complex transaction processing, messaging latency
- Listening across shards results in heavy backend contention
  - Increases latency in notification delivery
  - Results in more events being batched hence spikes are observable
- Dynamic sharding improves performance with multiple applications
  - Per-application shards result in better isolation and improved parallelism
  - Single-threaded applications are unable to fully saturate IMDS

Getting & Running the Performance Test Code

- Clone coreturials:

  git clone https://git.opendaylight.org/gerrit/coretutorials.git

- Build a distribution with sharding performance tests:

  cd coreturials/clustering/shardingsimple
  mvn clean install

- Run the distribution:

  ./karaf/target/assembly/bin/karaf –Xmx4096m

- Run the test script:

  cd coreturials/clustering/scripts/sharding
  ./shardingbenchmark.py –help (will print all the parameters in the script)
  More info in coreturials/clustering/scripts/sharding/site/asciidoc/scripts-user-manual.adoc

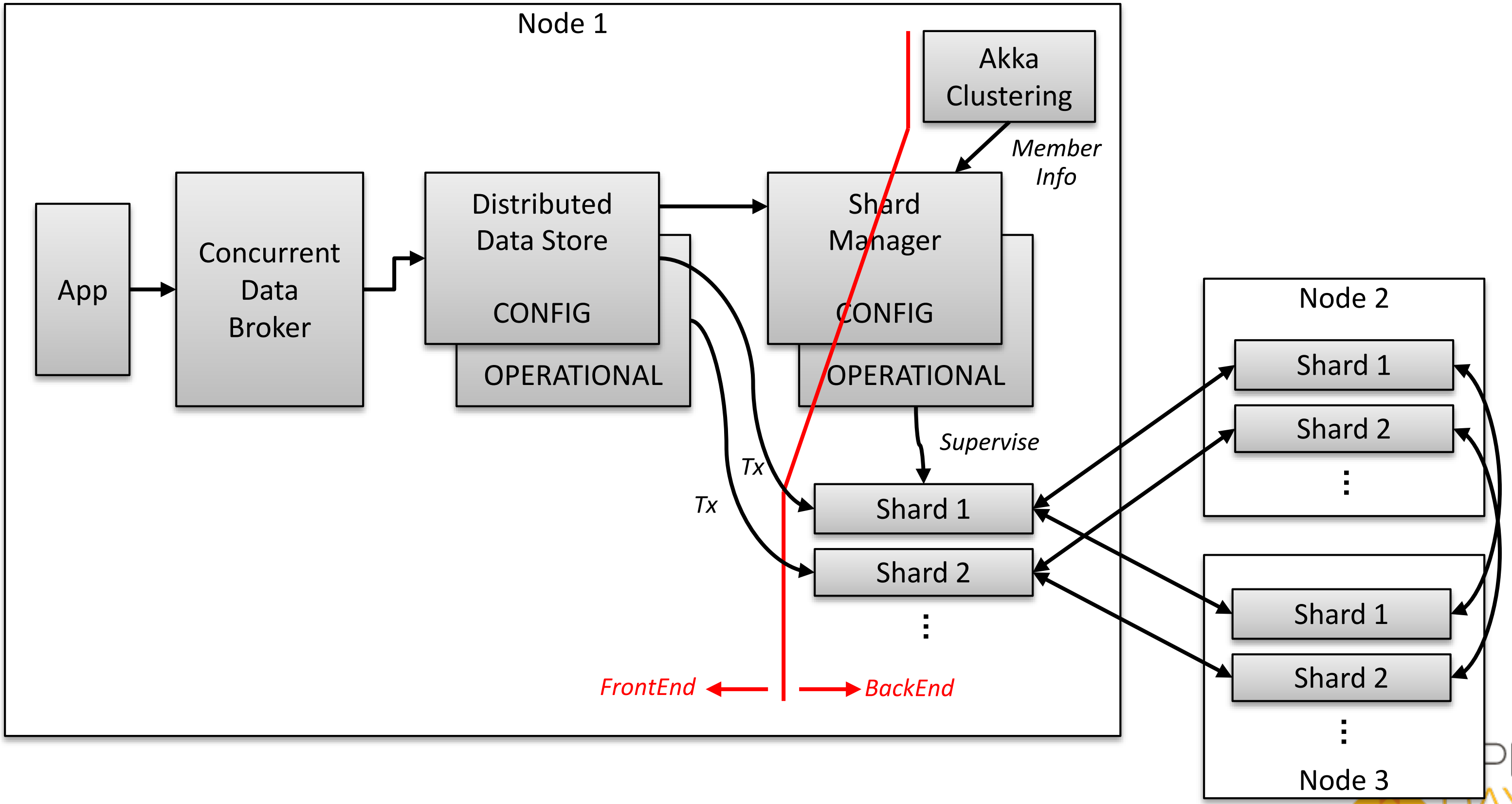
OPEN DAYLIGHT

ODL Yang Resources

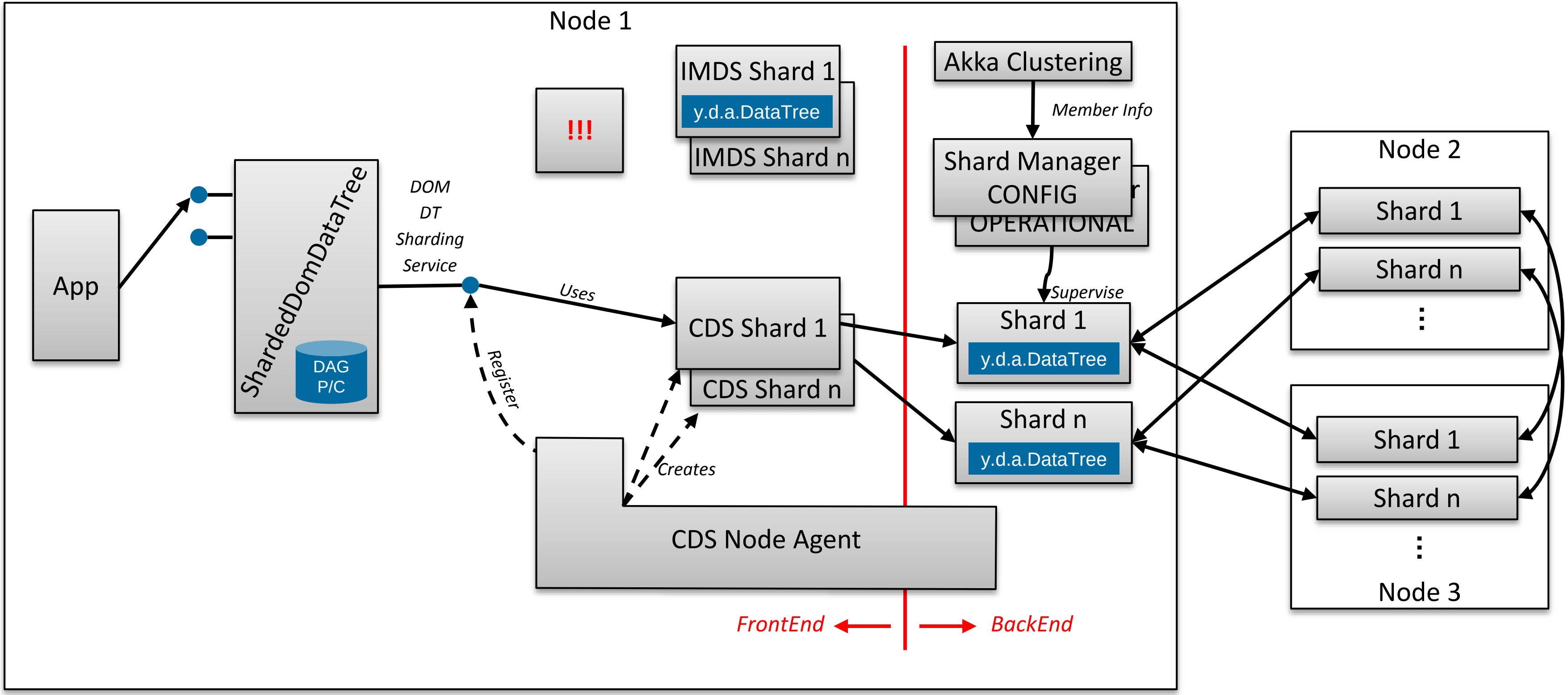
- YangTools main page:
  - https://wiki.opendaylight.org/view/YANG_Tools:Main
- Code Generation demo
  - https://wiki.opendaylight.org/view/Yang_Tools:Code_Generation_Demo
- Java “Binding Specification”:
  - https://wiki.opendaylight.org/view/YANG_Tools:YANG_to_Java_Mapping
- DLUX
  - Main page: https://wiki.opendaylight.org/view/OpenDaylight_dlux:Main
  - YangUI: https://wiki.opendaylight.org/view/OpenDaylight_dlux:yangUI-user
- Controller:
  - Swagger UI Explorer:
    - http://localhost:8181/apidoc/explorer/index.html
  - DLUX (Yangman):
    - http://localhost:8181/dlux/index.html

OPEN DAYLIGHT

Existing Clustered Data Store - Details



Data Store Evolution in Boron and Beyond



y.d.a. = Yang Data API

DAB/PC – Directed Acyclic Graph Producer/Consumer