
OPEN
DAYLIGHT
SUMMIT

OPEN DAYLIGHT SUMMIT

How to Design OpenDaylight Applications for Best Scale and Performance

Jan Medved, Distinguished Engineer, Cisco
Tony Tkacik, Architect, Cisco

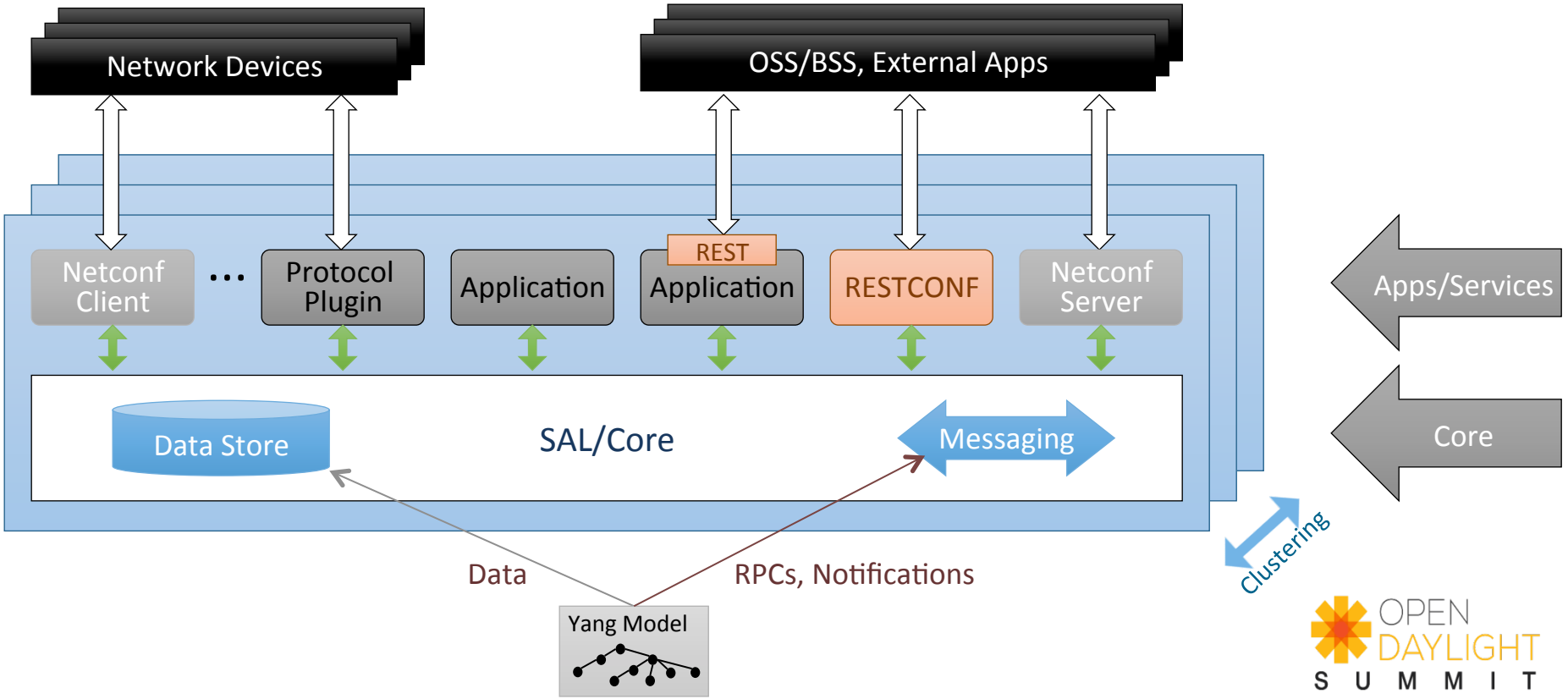
#ODSummit

Agenda

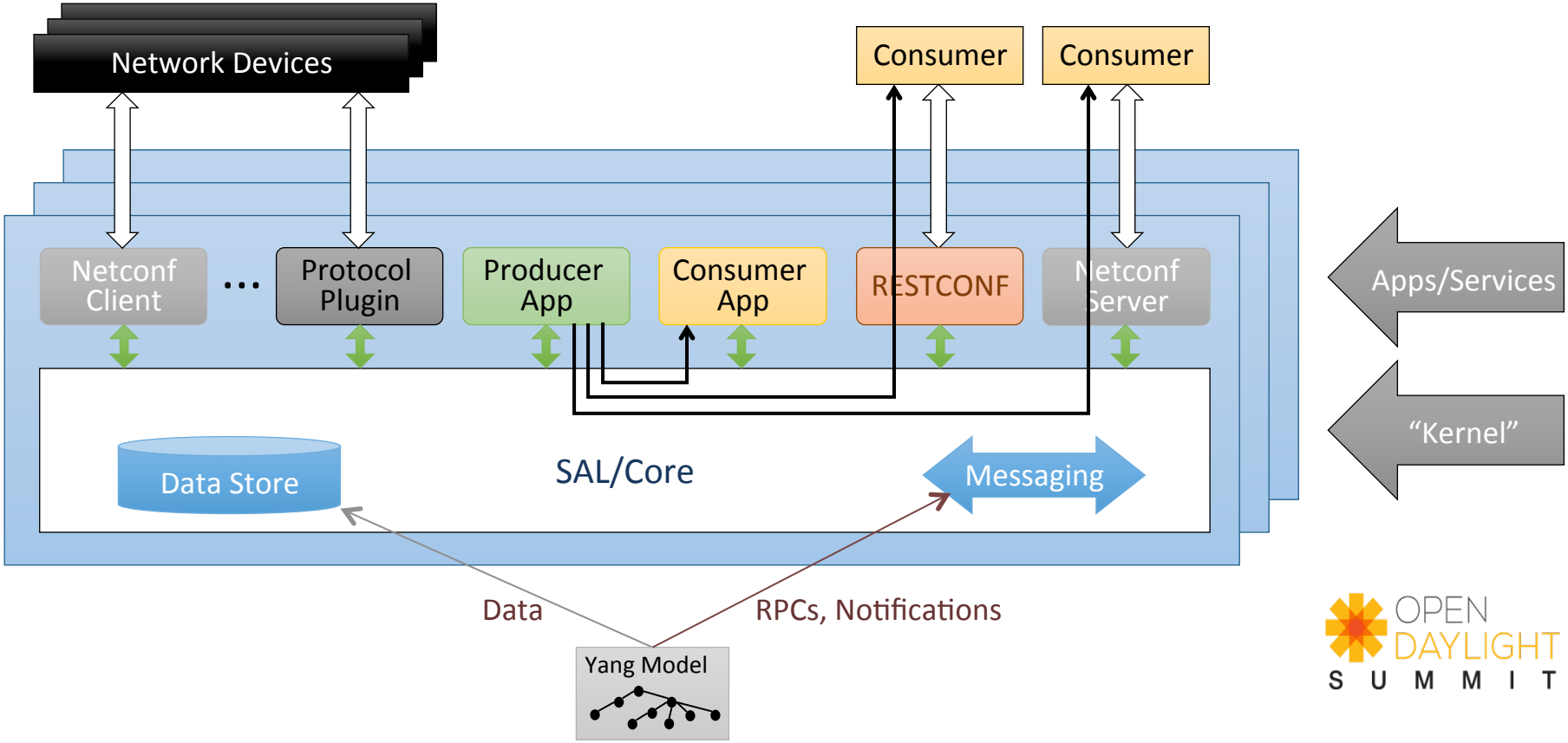
- Architectural Overview & Background
- General Recommendations
- MD-SAL:
  - The Data Store
  - Messaging: RPCs and Notifications
- Coming in Beryllium:
  - MD-SAL APIs
  - Clustering
- Q&A


OPEN
DAYLIGHT
SUMMIT
Architectural Overview &
Background
OPEN
DAYLIGHT
SUMMIT
#ODSummit

ODL Software Architecture

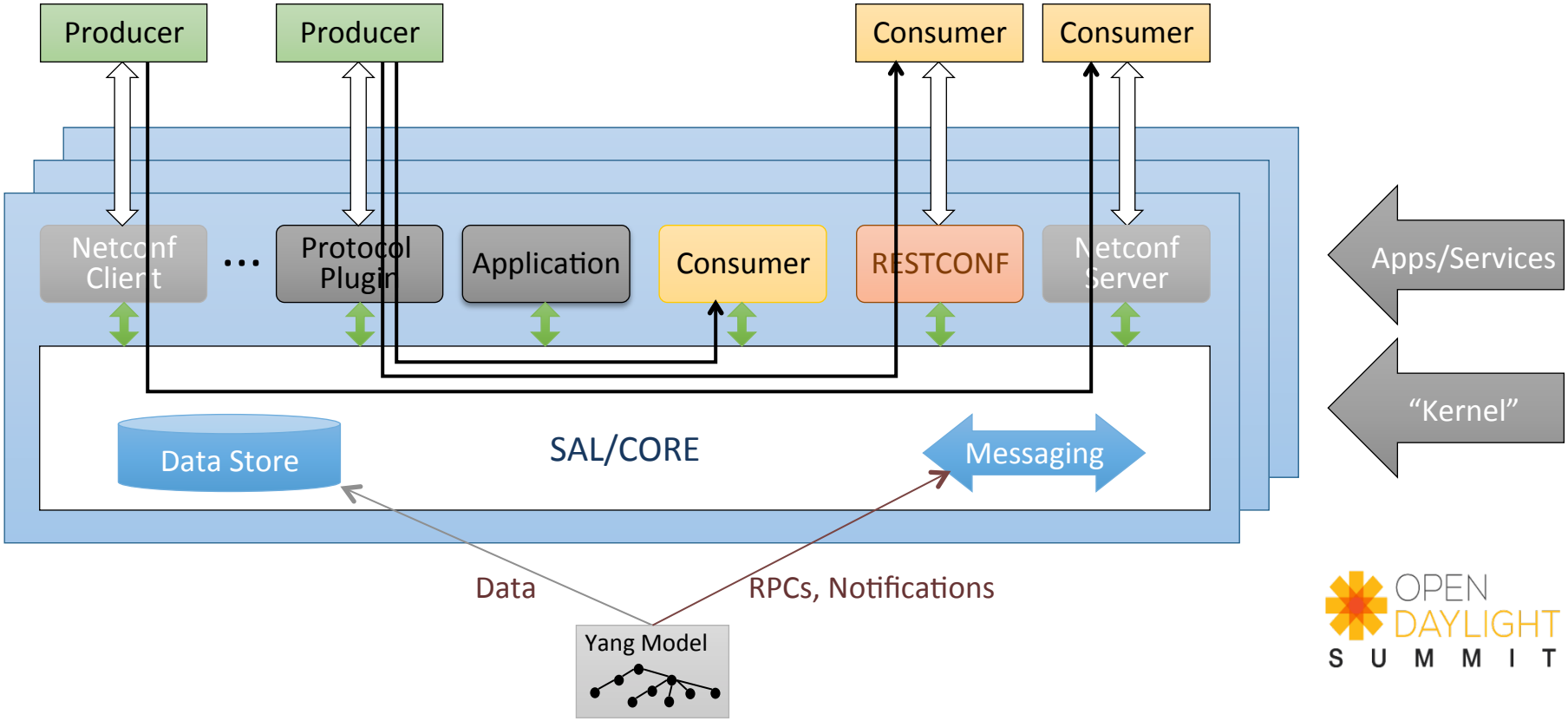


MD-SAL: Connecting Producers & Consumers



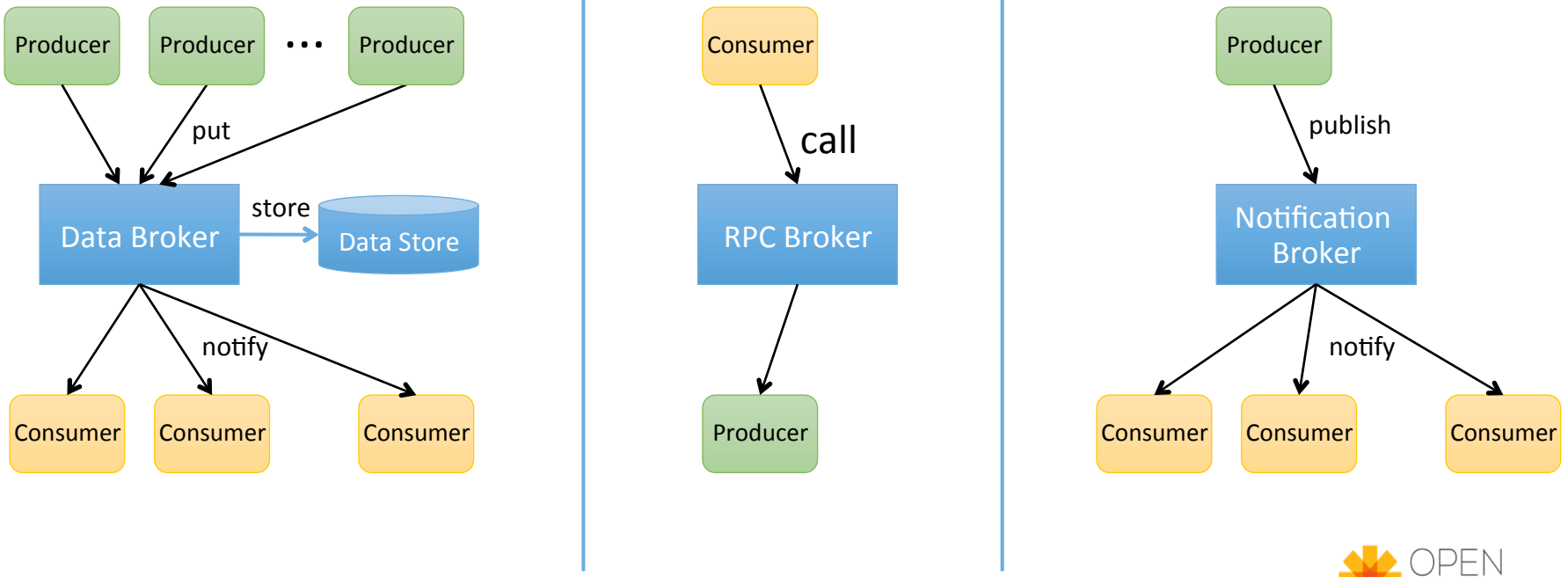
OPEN DAYLIGHT SUMMIT

MD-SAL: Connecting Producers & Consumers

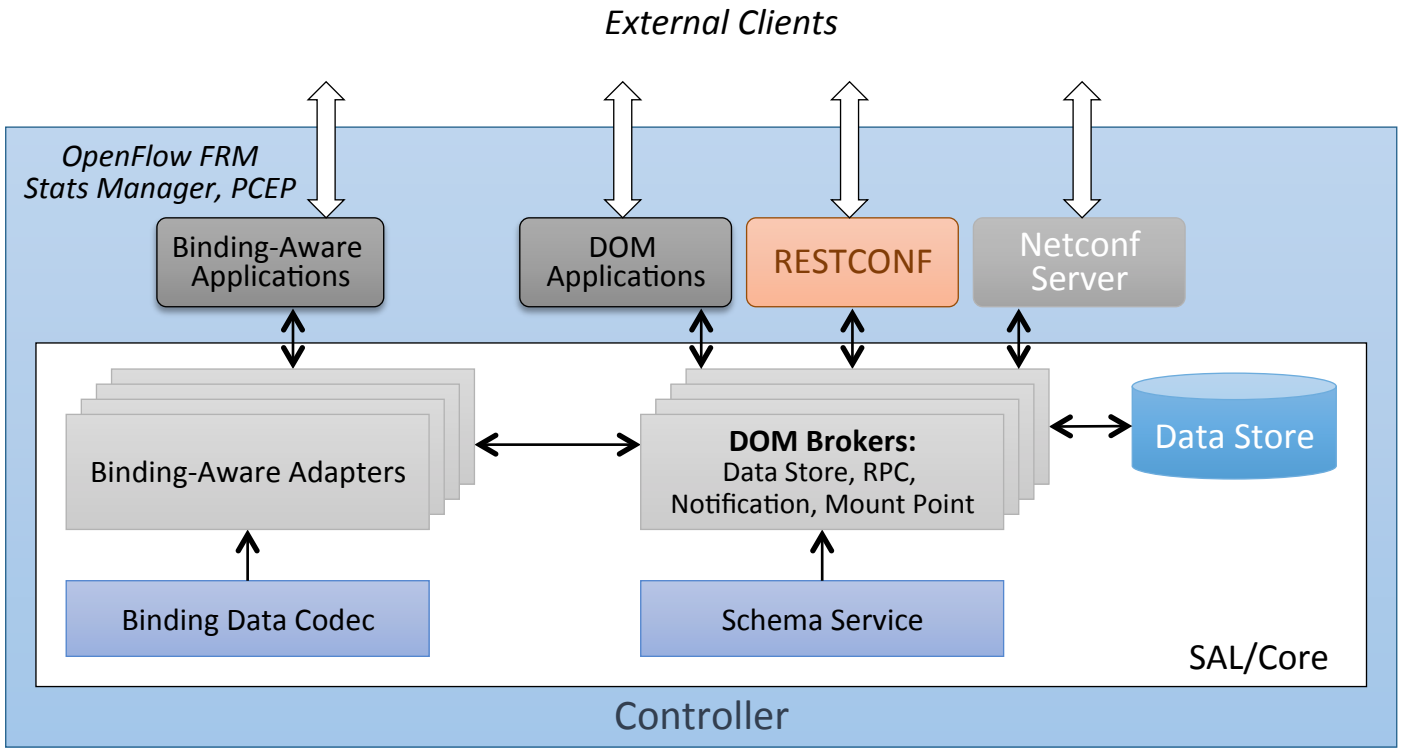


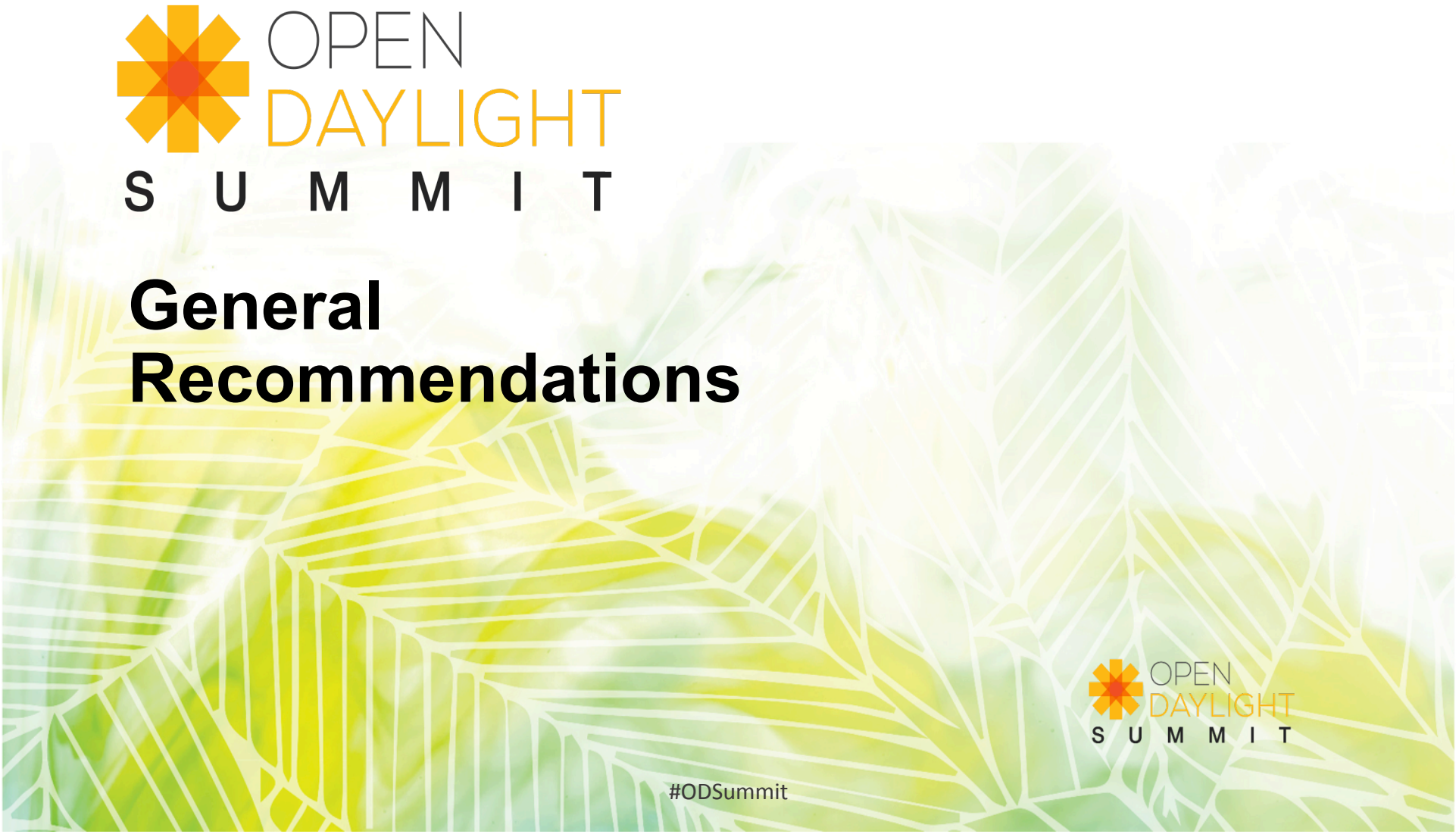
OPEN DAYLIGHT SUMMIT

MD-SAL: The Brokers



MD-SAL: The Internals



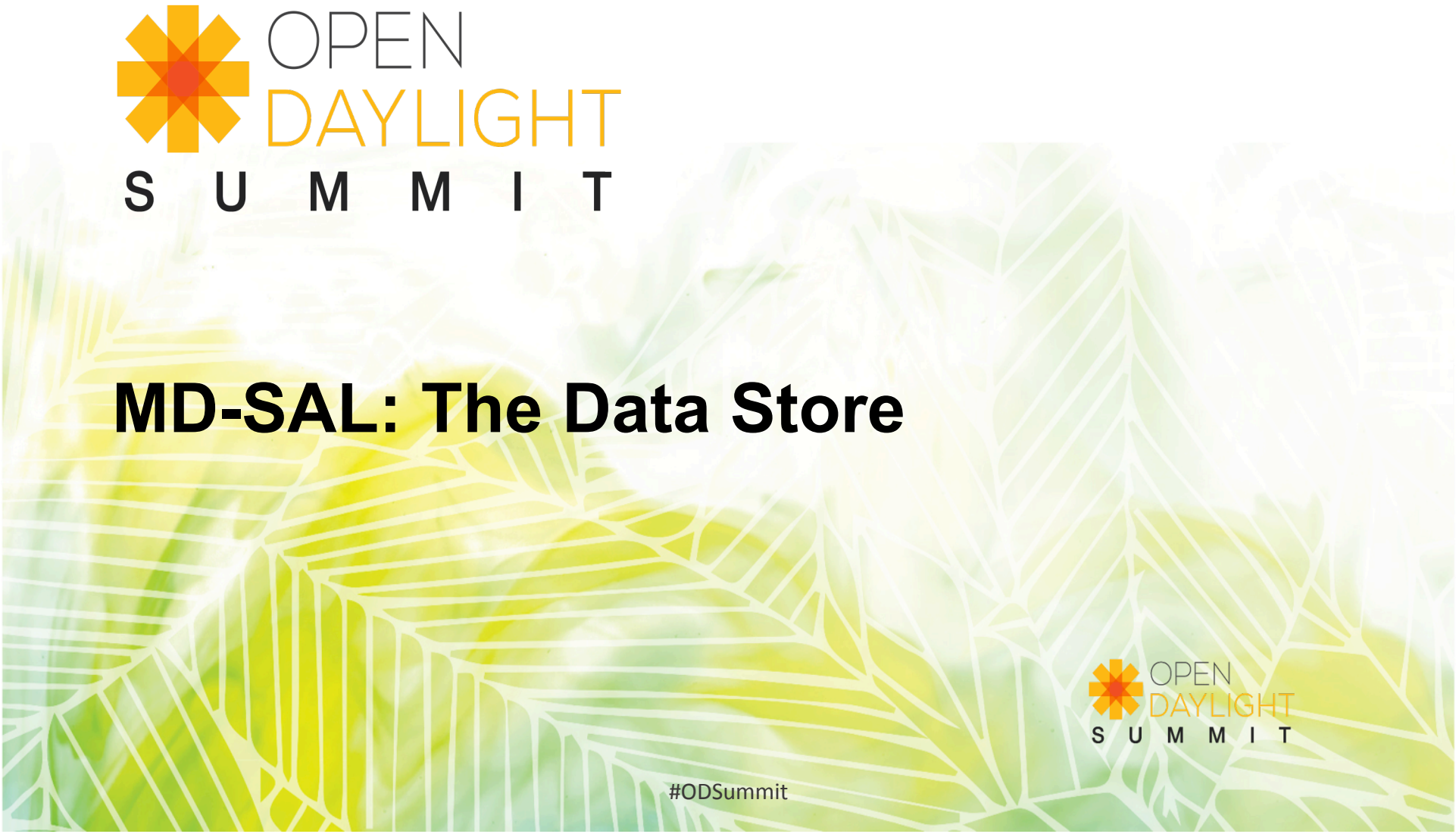

OPEN
DAYLIGHT
SUMMIT
General
Recommendations
OPEN
DAYLIGHT
SUMMIT
#ODSummit

Using Java Futures correctly

- Avoid blocking on Futures unless necessary
- Register a callback invoked once the operation is completed:
  - Use **Futures.addCallback()**
- Assume that code returning Java Futures runs in a different thread and will take some time to complete

Other Recommendations

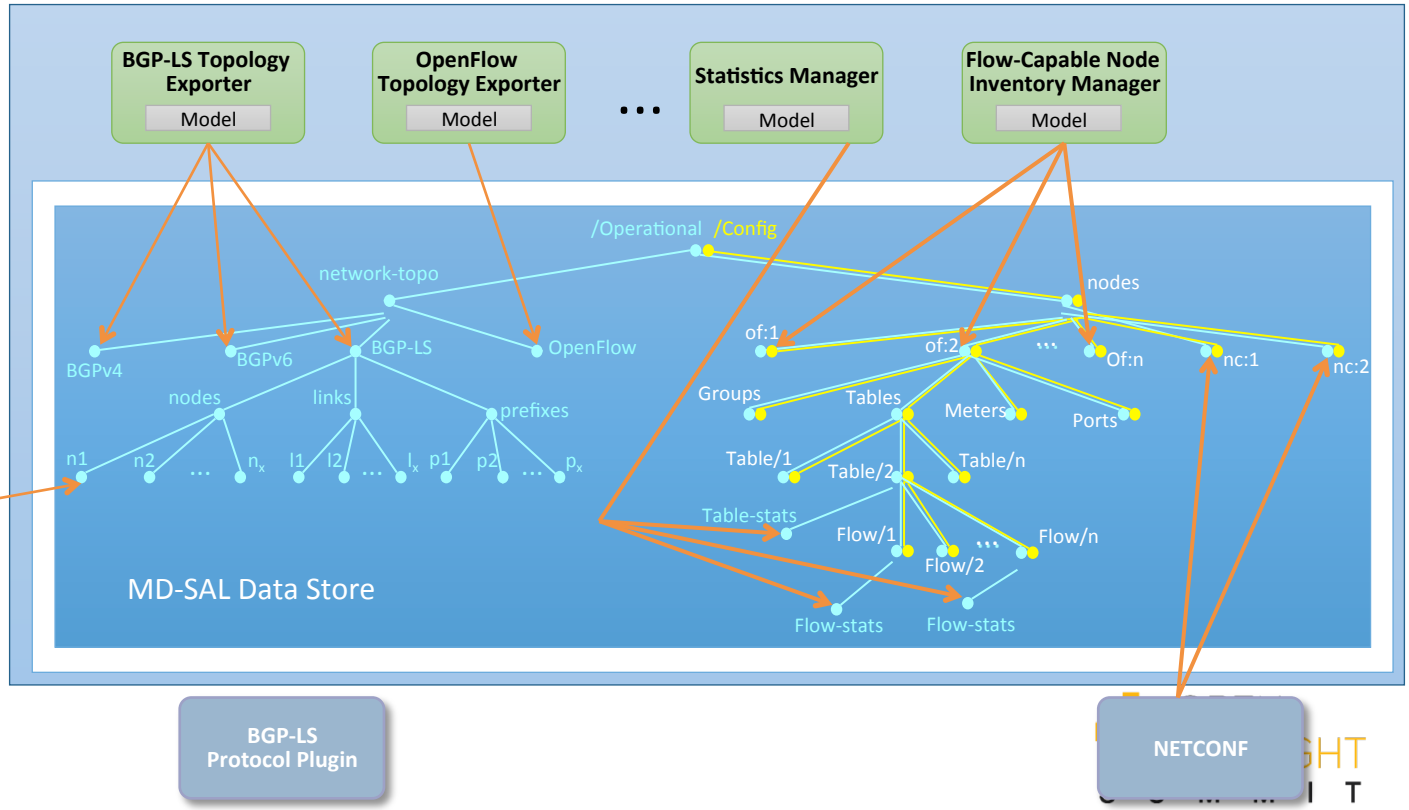
- Programming:
  - Logging:
    - Incorrect use of logging may decrease performance:
    - Use `LOG.isDebugEnabled()` if log statement computation is CPU-heavy
    - Logging best practices: https://wiki.opendaylight.org/view/BestPractices/Logging_Best_Practices
  - Avoid unnecesary synchronization
- Environment tuning:
  - Garbage collection
  - TCP stack configuration



MD-SAL: The Data Store

Data Store Key Concepts

- Yang data is a tree
- Two Logical Data Stores:
  - Config
  - Operational
- Unified View
- InstanceIdentifier:
  - Pointer to a node



Data Store Recommendations

- Design proper structure of your data (depth vs width)
- Use **DataTreeChangeListener**
  - instead of repetitive READs
  - instead DataChangeListener
- Use **WriteOnlyTransaction** over ReadWriteTransaction
- Use **TransactionChain** if your transactions depends on each other
- Where possible, use **PUT** over MERGE.
- Use **„pingpong-broker“** for best write batching performance with WriteOnlyTransactions
- **Single writer** for any given subtree

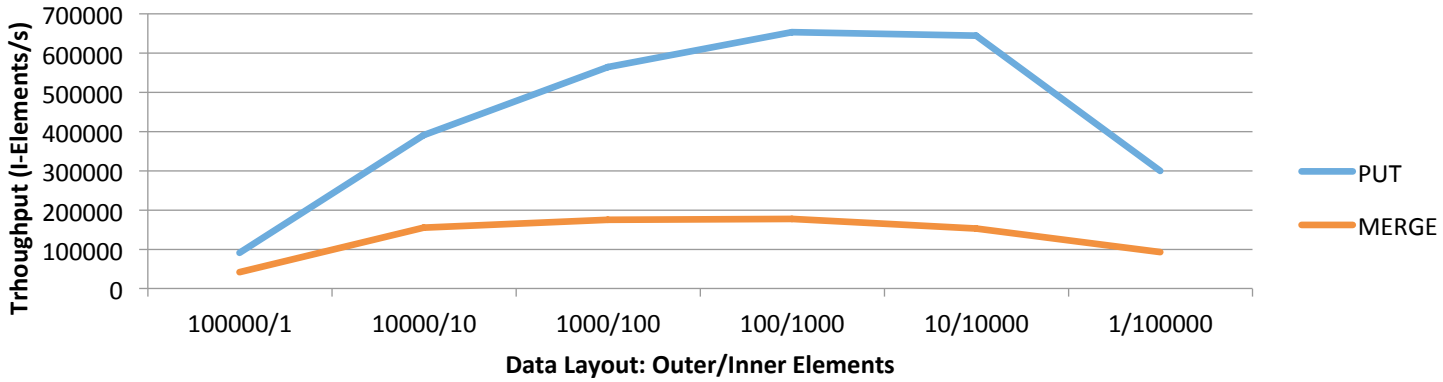
Data Store Performance Considerations

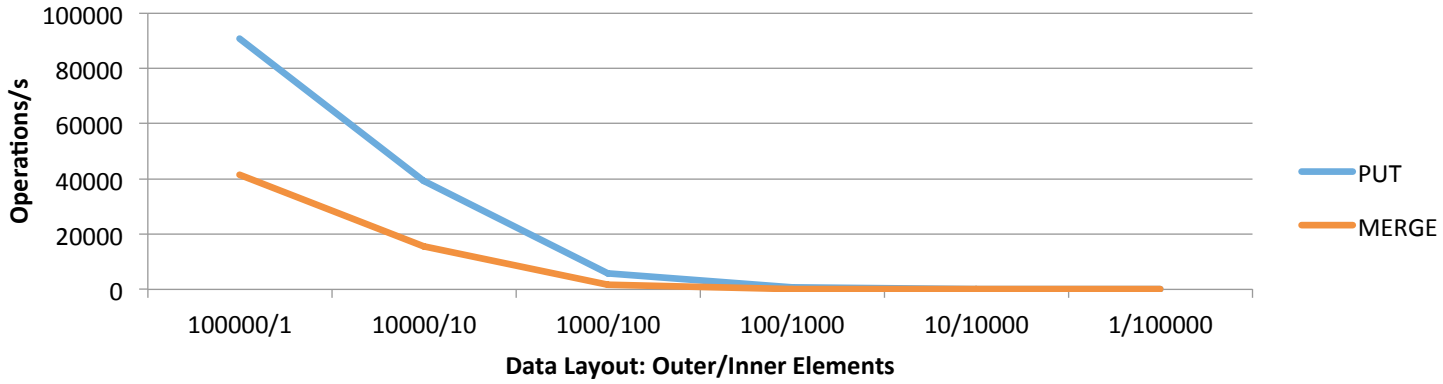
- Structure of data: breadth vs. depth
- Merge vs. Put vs. Delete
- Persistence: On/Off
- Data Broker API: Binding-Aware vs. DOM
- Simple Transaction vs. Transaction Chain
- Number of operations per submit

- Test Environment:
  - 2014 Mac:
    - 16 Gig 1600 MHz DDR3 RAM/ 2.2 GHz Intel Core i7 / OSX 10.10.3 / SSD
  - Java 8 / 4G heap / G1GC
  - No clustering, single node operation

Data Store Benchmark: Data Layout

Depth vs. breadth: Data structure impact (Persistence off, transaction chain, pingpong buffer)

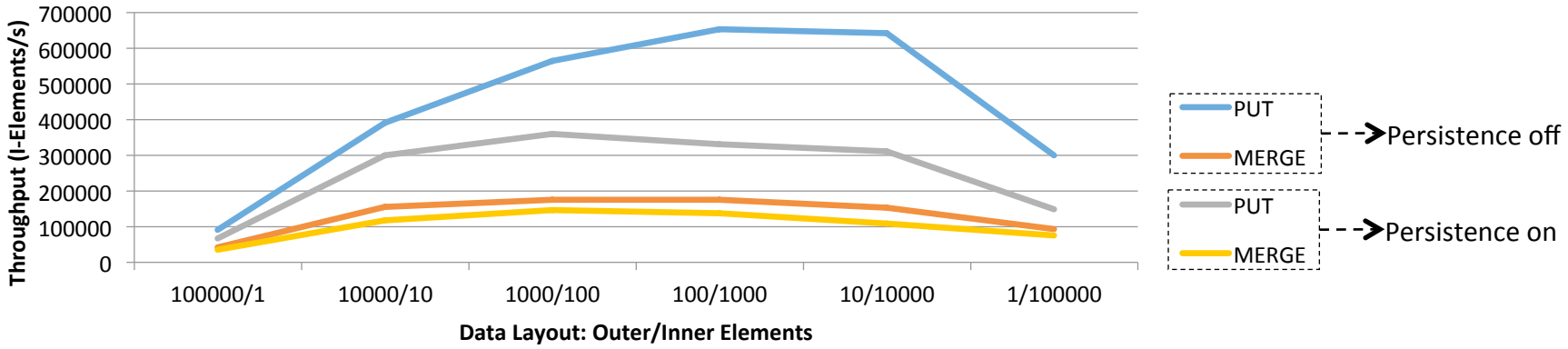


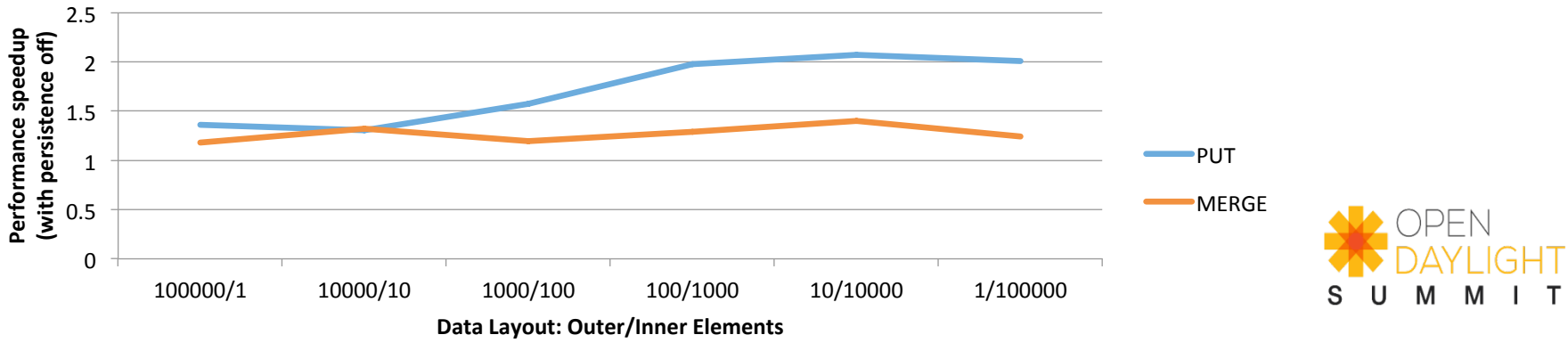


OPEN DAYLIGHT SUMMIT

Data Store Benchmark: Persistence

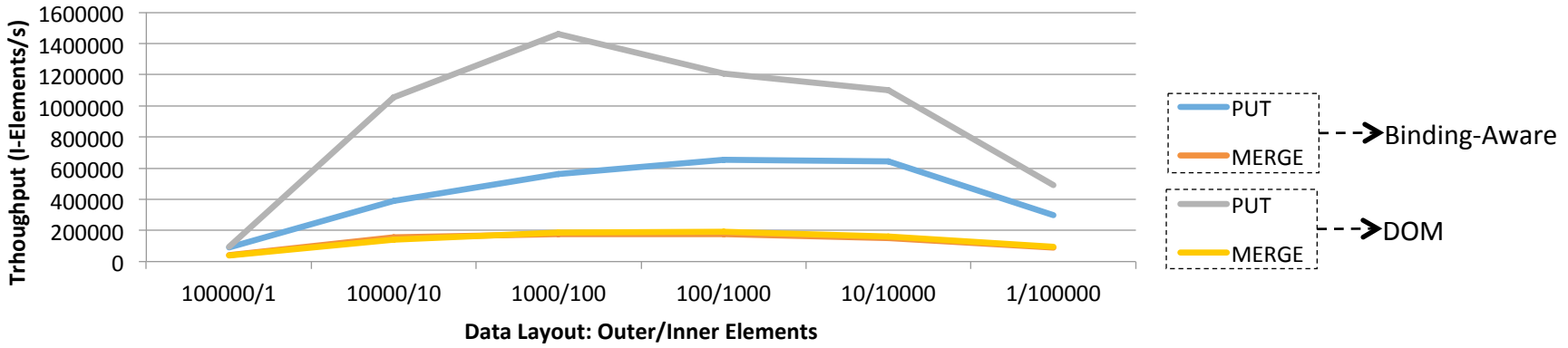
Persistence on/off, transaction chain, pingpong buffer

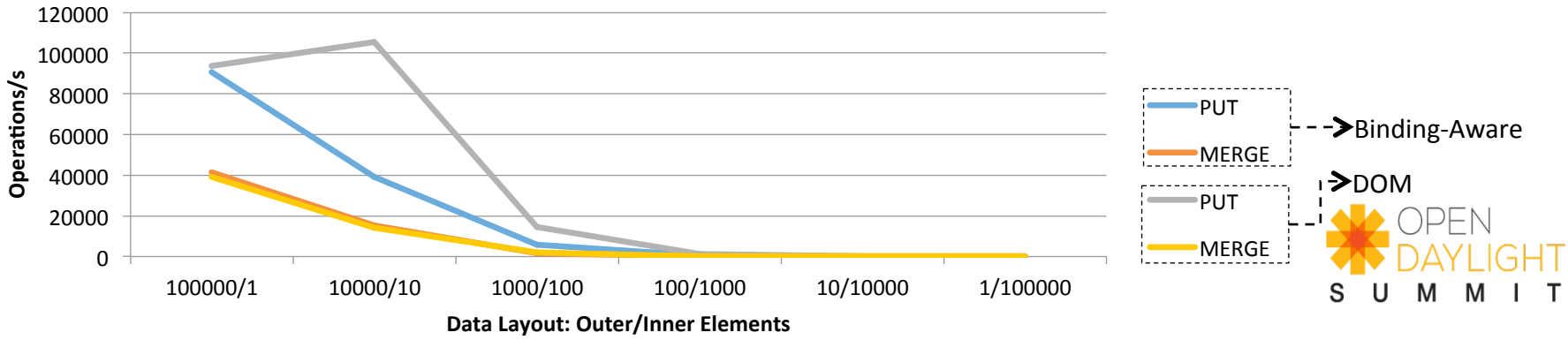




Data Store Benchmark: Binding-Aware vs. DOM

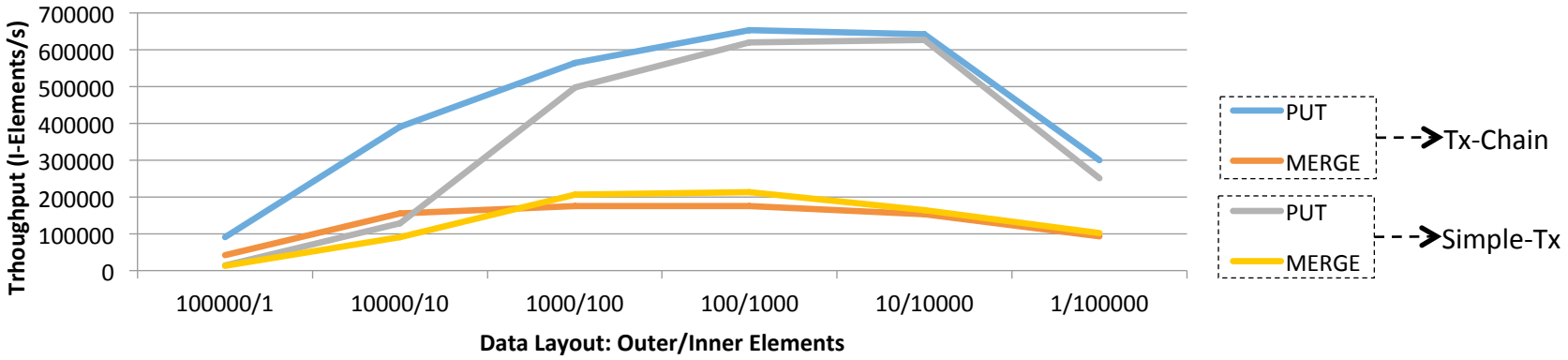
Persistence off, transaction chain, pingpong buffer

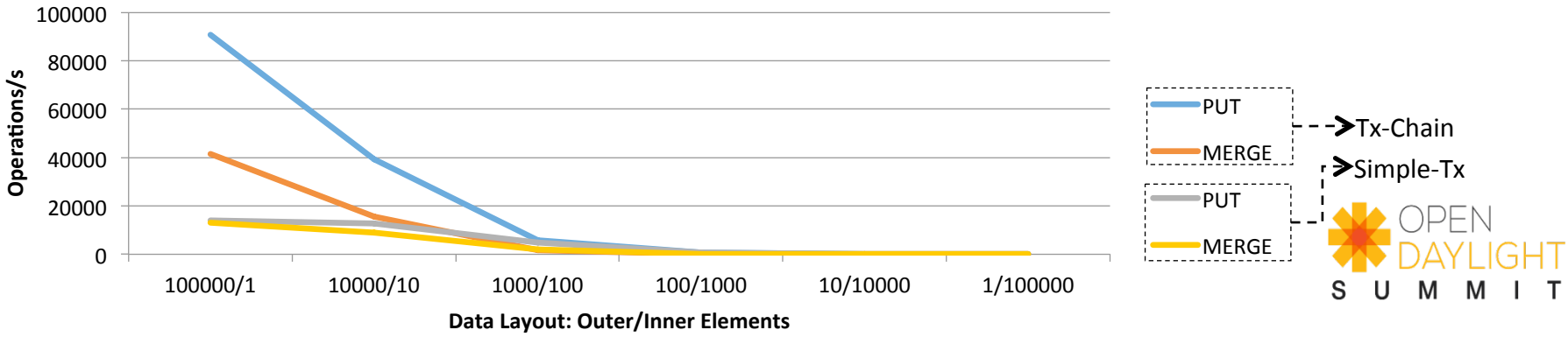




Data Store Benchmark: Simple-Tx vs. Tx-Chain

(***Persistence off***, binding-aware, transaction chain with pingpong buffer)

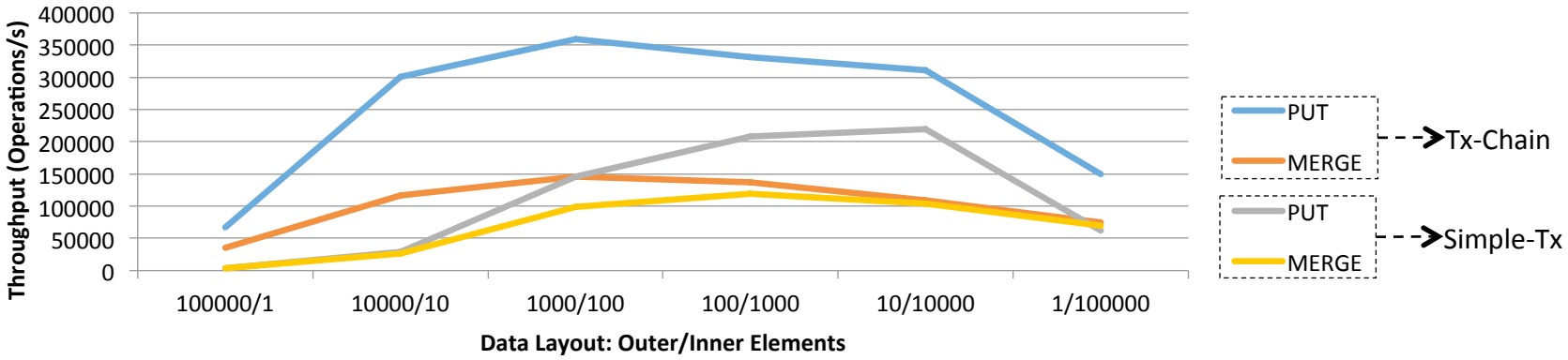




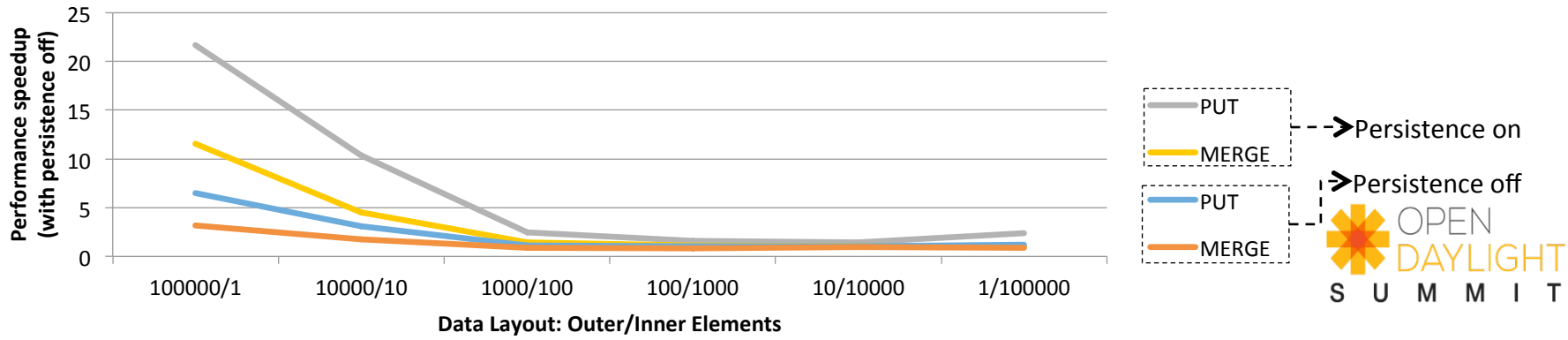
OPEN DAYLIGHT SUMMIT

Data Store Benchmark: Simple-Tx vs. Tx-Chain

(***Persistence on***, binding-aware, transaction chain with pingpong buffer vs simple-tx)

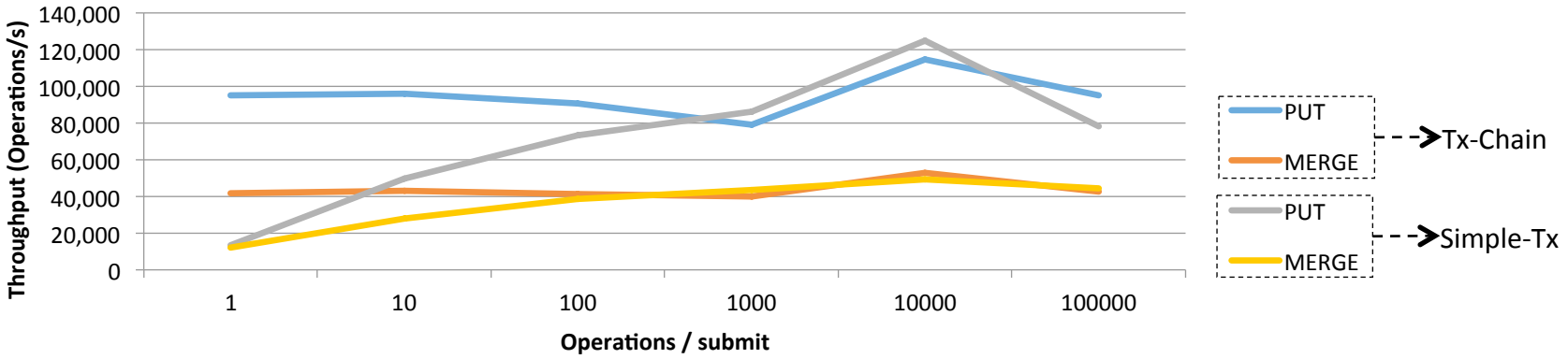


(***Performance speedup*** transaction chain with pingpong buffer vs. simple-tx, binding-aware)

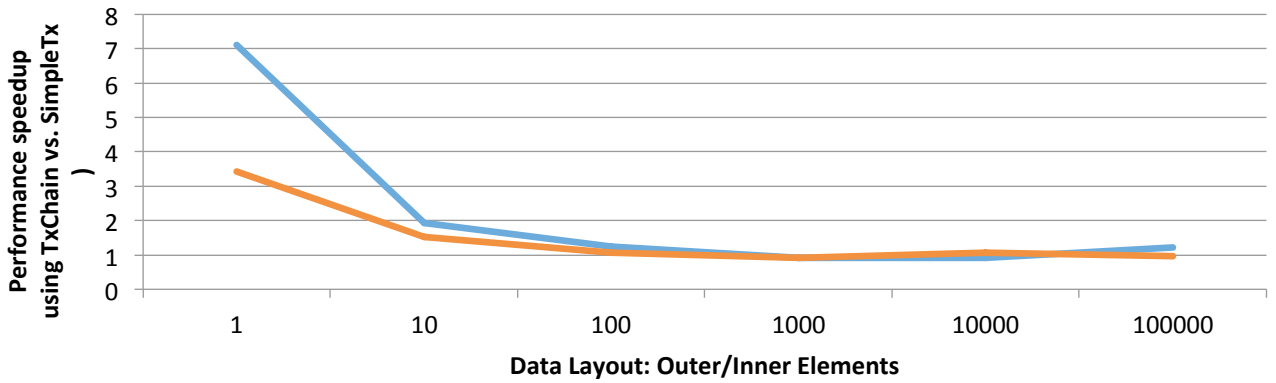


Data Store Benchmark: Batching Operations

100k/1 list, ***Persistence off***, Tx-Chain vs. Simple Tx, Binding-Aware

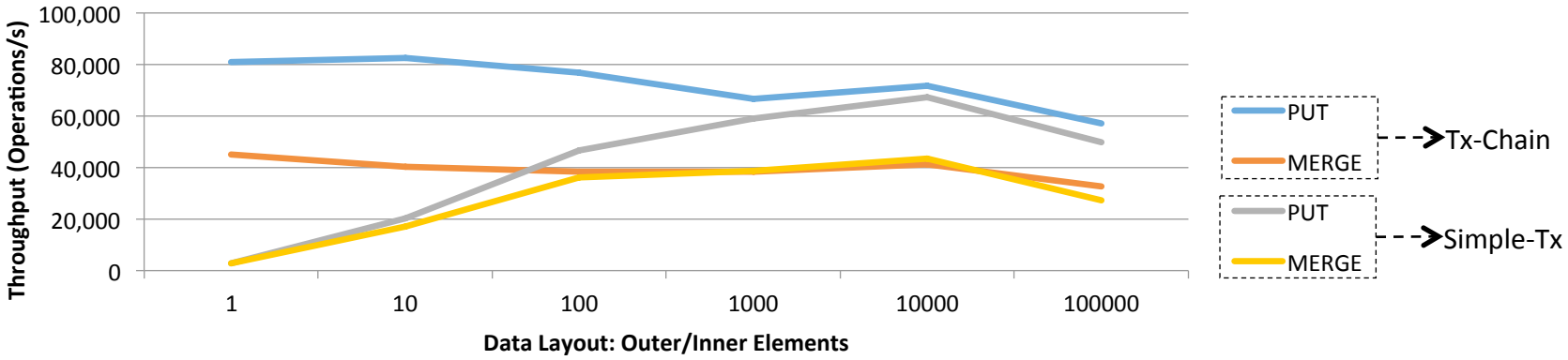


100k/1 list, ***Performance speedup***, Persistence off, Tx-Chain vs. Simple-Tx, Binding-Aware

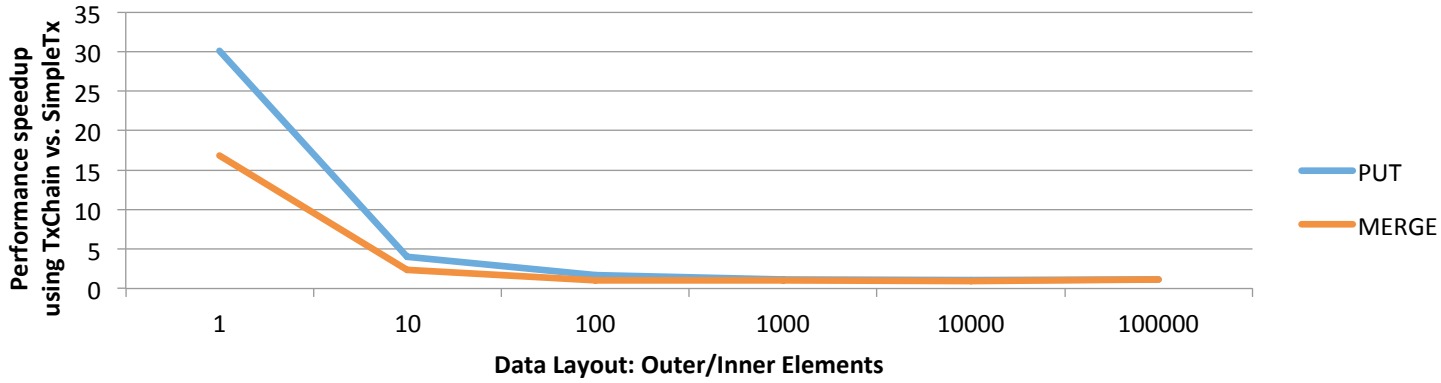


Data Store Benchmark: Batching Operations

100k/1 list, ***Persistence on***, Tx-Chain vs. Simple Tx, Binding-Aware



100k/1 list, ***Performance speedup***, Persistence on, Tx-Chain vs. Simple-Tx, Binding-Aware



MD-SAL Messaging: RPCs and Notifications

OPEN
DAYLIGHT
SUMMIT

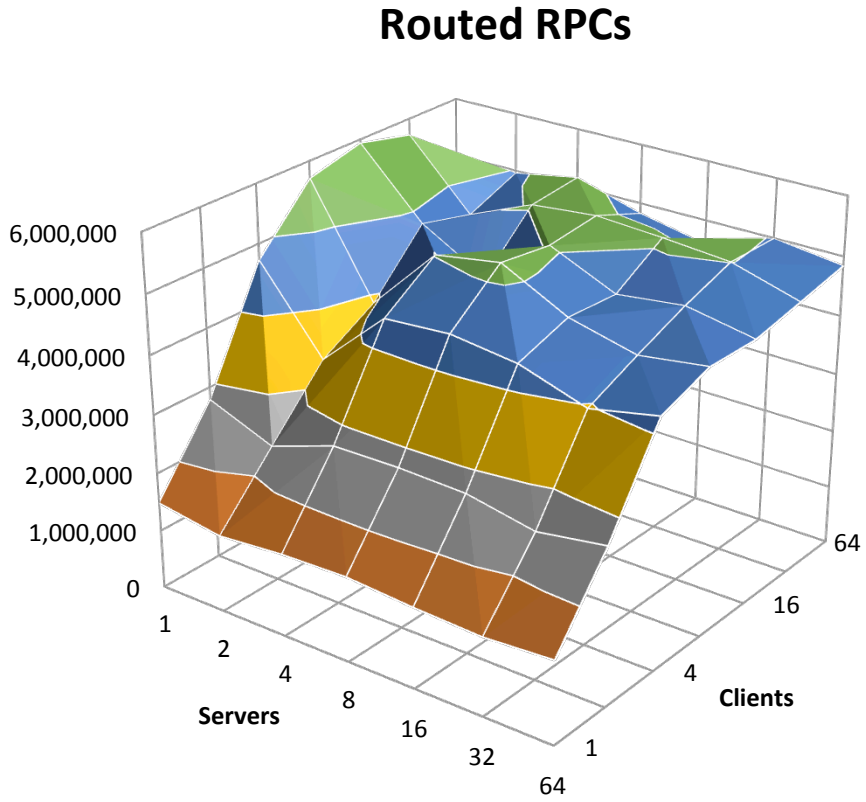
Using RPCs (Client)

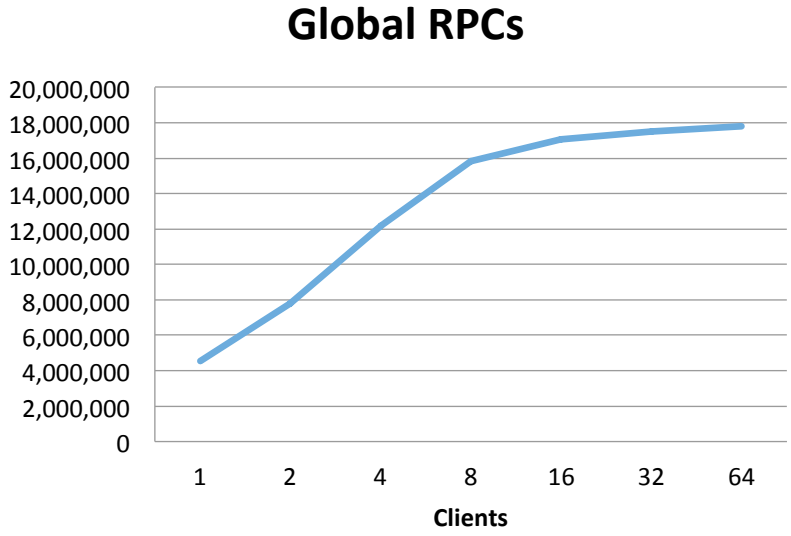
- **Do not block on** returned RPC future; use **Futures.addCallback()** instead.
- Assume that the invocation of an RPC may throw the RuntimeException on an incorrect input -> handle if necessary.

Implementing RPCs (Server)

- Do not do CPU-heavy operations in the RPC callback:
  - Dispatch and process requesst asynchronously
  - For simplicity use **ListeningExecutorService**
- Where possible return **ListenableFuture**
  - Consider using **SettableFuture** if delegating to a component that has a different asynchronous / callback mechanism.

RPC Benchmark





OPEN DAYLIGHT SUMMIT

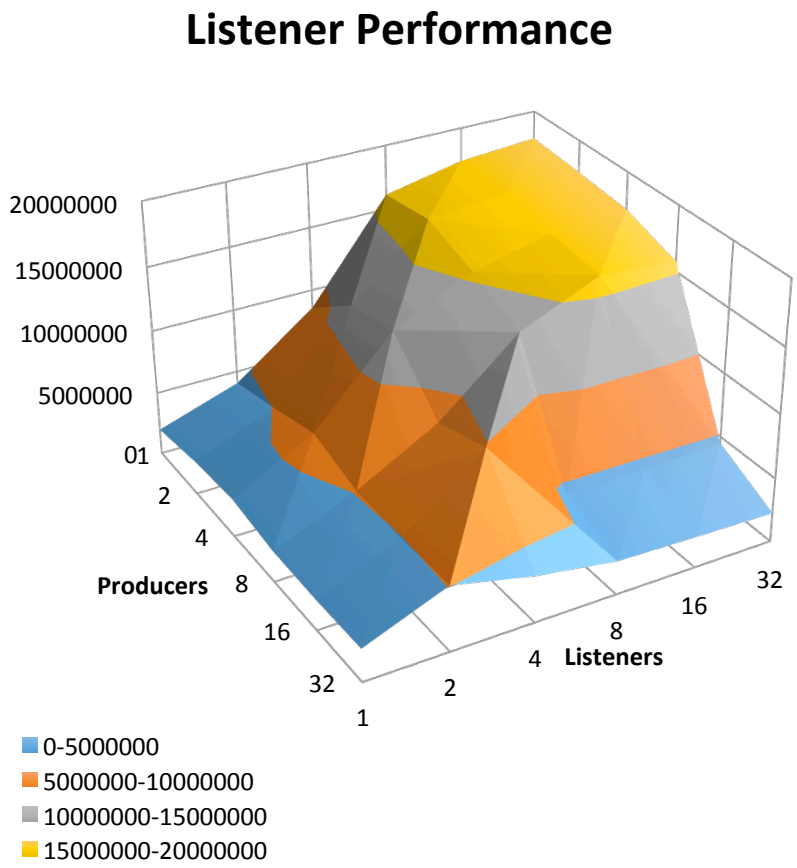
Consuming Notifications

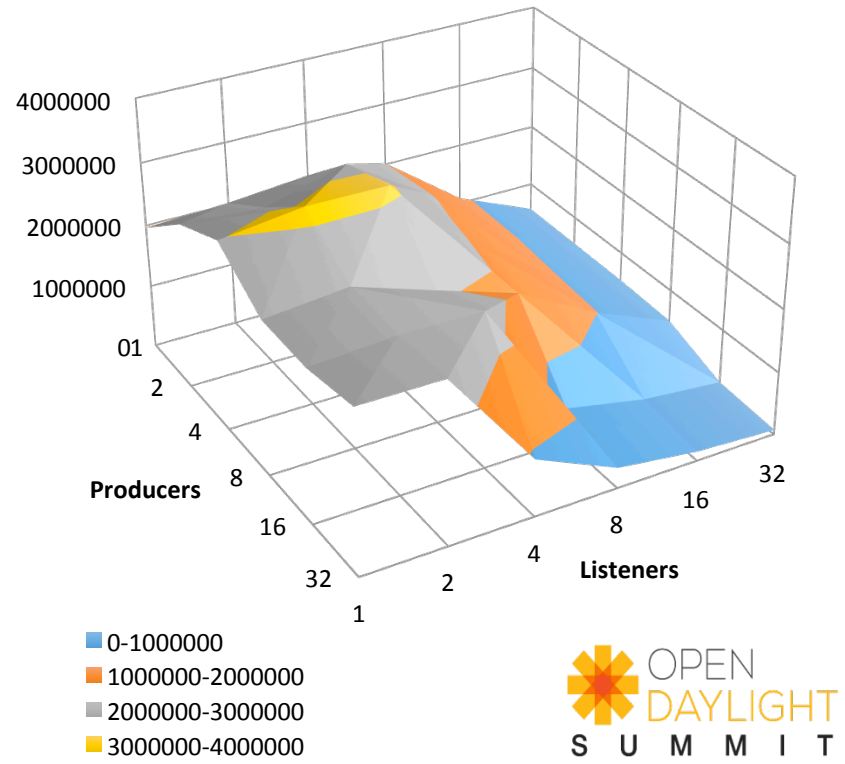
- **Do NOT perform any CPU-heavy operations** in your callback
  - Dispatch and process notifications asynchronously if CPU-heavy processing is needed
  - For simplicity use ListeningExecutorService
  - Use advanced dispatch methods if you need more contro

Publishing Notifications

- Use **blocking APIs** (`putNotification()`), only if necessary.
  - **DO NOT use blocking APIs** in callbacks from MD-SAL.
- Always use `offerNotification()` for **non-critical notifications** (i.e. those that can be dropped)
  - If you get back the **`NotificationPublishService.REJECTED`** future, the publishing queue is full -> **back-off** or **slow-down**
  - A future completes once notifications are delivered to the immediate listeners (in the same JVM); may be useful for back-off implementation

Notifications Benchmark





OPEN DAYLIGHT SUMMIT

Coming in Beryllium

What? Yet another of new MD-SAL APIs?

- NO, **just improved APIs**, which will allow the MD-SAL to **have a better insight into** how **data** is produced and consumed:
  - Allows for **better under-the-hood optimization** of data flows
  - Captures the causality of data changes in the system
  - Introduces **micro-sharding**
  - Introduces **more usable APIs:**
    - **For data change listeners** based on feedback from applications
    - **For transaction chaining**
    - **For operation batching**
  - **Actor-like** model for listeners / producers

How Will This Affect My Code?

- Affects the **instantiation** of listeners and transactions, not their use:
  - The new APIs have a lot in common with the current APIs
  - The concepts are similar
  - Migration Guide on the way
- **Examples:**
  - **DataTreeChangeListener -> DataChangeListener**
    - You get all the state that you registered for **– no need for READs**
  - **TransactionChain -> DataTreeProducer**
    - Subtree scoped, easier to use, allows for opt-in / opt-out of batching of transaction operations


OPEN
DAYLIGHT
SUMMIT
Questions?
OPEN
DAYLIGHT
SUMMIT
#ODSummit

Where to Find More Information

- Details:
  https://wiki.opendaylight.org/view/Controller_Core_Functionality_Tutorials:Tutorials:Data_Store_Benchmarking_and_Data_Access_Patterns
- Code: coretutorials/dsbenchmark

Thank You

#ODSummit

Test Data: List of Lists

Yang Model:

```
container test-exec {
    config true;

    list outer-list {
        key id;
        leaf id {
            type int32;
        }

        list inner-list {
            key name;
            leaf name {
                type int32;
            }

            leaf value {
                type string;
            }
        }
    }
}
```

Using Generated Java DTOs:

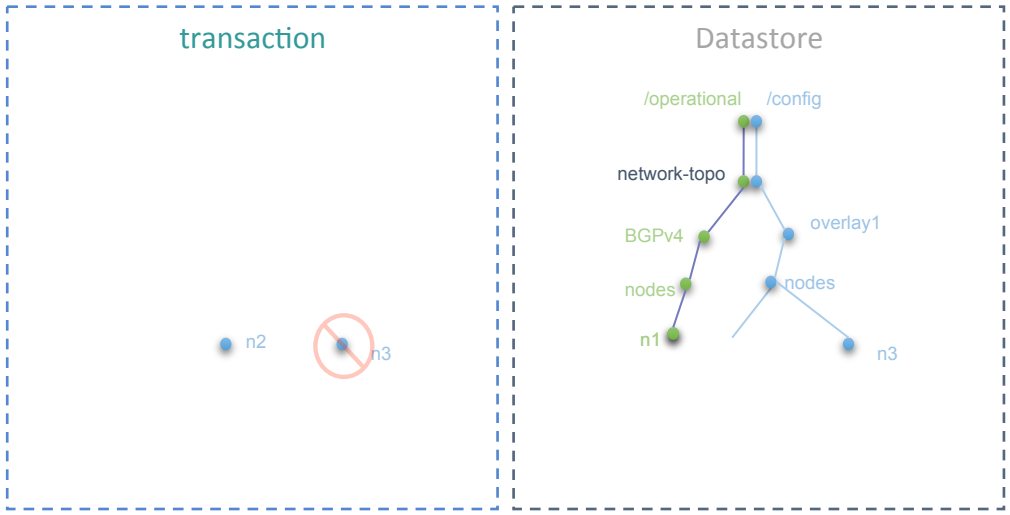
```
static public List<OuterList> buildOuterList(int outerElements,
                                              int innerElements) {
    List<OuterList> outerList = new ArrayList<OuterList>(outerElements);
        for (int j = 0; j < outerElements; j++) {
            outerList.add(new OuterListBuilder().setId( j )
                                .setInnerList(buildInnerList(j, innerElements))
                                .setKey(new OuterListKey( j )).build());
        }
    return outerList;
}

static private List<InnerList> buildInnerList( int index, int elements ) {
    List<InnerList> innerList = new ArrayList<InnerList>( elements );

    final String itemStr = "Item-" + String.valueOf(index) + "-";
    for( int i = 0; i < elements; i++ ) {
        innerList.add(new InnerListBuilder().setKey( new InnerListKey( i ) )
                            .setName(i).setValue( itemStr + String.valueOf( i ) )
                            .build());
    }
    return innerList;
}
```

Transactions: Read/Write/Delete

```
ReadWriteTransaction transaction =
dataBroker.newReadWriteTransaction();
Optional<Node> nodeOptional;
nodeOptional = transaction.read(
   LogicalDataStore.OPERATIONAL,
   n1InstanceIdentifier);
transaction.put(
   LogicalDataStore.CONFIGURATION,
   n2InstanceIdentifier,
   topologyNodeBuilder.build());
transaction.delete(
   LogicalDataStore.CONFIGURATION,
   n3InstanceIdentifier);
CheckedFuture future;
future = transaction.submit();
```



Transactions: Merge vs. Put

```
WriteOnlyTransaction transaction =
dataBroker.newWriteOnlyTransaction();
InstanceIdentifier<Node> path =
   InstanceIdentifier
      .create(NetworkTopology.class)
      .child(Topology.class,
             new TopologyKey(
                  “overlay1”));
transaction.merge(
   LogicalDataStore.CONFIG,
   path,
   topologyBuilder.build());
CheckedFuture future;
future = transaction.submit();
```