Overview of Model-Driven SAL and Creating an Application based on MD-SAL

Radhika Hirannaiah, OpenDaylight Project
Open Networking Summit 2016
Goal

• Overviewing MD-SAL internals for application development
OpenDaylight SDN Controller

• Java-based, model-driven controller using YANG modeling language
• Relies on the following technologies:
  • **OSGI** - Back-end framework of OpenDaylight that allows dynamically loading of bundles and packages JAR files, and binding bundles together for exchanging information
  • **Karaf** - Application container built on top of OSGI, which simplifies operational aspects of packaging and installing applications
  • **YANG** - a data modeling language used to model configuration and state data manipulated by the applications, remote procedure calls, and notifications
• Tools
  • IDE - IntelliJ IDEA or Eclipse
  • Java 1.7 or 1.8
  • Maven 3.2.3/3.3.x
  • Postman REST Client
What is Model-Driven SAL (MD-SAL)?

- OpenDaylight kernel, that interfaces between different layers and modules
- Uses APIs to connect and bind requests and services
- Service Abstraction Layer (SAL) introduces an extra layer containing all the necessary logic to receive and delegate requests
- Forwards incoming requests to Service Providers (SPs) that returns service results through API to the Clients (Consumers)
OpenDaylight Software Architecture

Network Devices

Applications

Model-Driven SAL (MD-SAL)

Controller

Messaging

Data Store

Remote Controller Instance

Remote Controller Instance

Protocols

- NETCONF
- RESTCONF

Plugins & Applications

Network Applications Orchestrations & Services

Controller Platform

Clustering
Why MD-SAL?

• Evolutionary step forward from object-oriented design of network management systems
• Agnostic model that supports any device and/or service models
• Designed to stitch together the modules horizontally by allowing the developer to use generic interfaces for service discovery and consumption
• Provides “common” REST API to access data and functions defined in models
YANG (RFC 6020)

- Modeling language that models semantics and data organization
- Models can be ‘augmented’
- Can model:
  - Config/Operational data as a tree
  - RPCs
  - Notifications
Remote Procedure Calls (RPCs)

- Used for any call or invocation that crosses the plugin or module boundaries
- Triggered by consumers
- Communication is Unicast between consumer and provider
- Consumer sends request message to provider responding with reply message
- Models any procedure call implemented by a Provider (Server) exposing functionality to Consumers (Clients)
- Two types of RPCs –
  - Global – one service instance per controller container
  - Routed – multiple service instance per controller container
RPCs – Sending a Message
E.g: HelloWorld application

HelloService helloService =
    session.getRpcService(HelloService.class);

Future<RpcResult<HelloWorldOutput>> future;
future = helloService
    .helloWorld(helloWorldInput);
HelloWorldOutput helloWorldOutput =
    future.get().getResult();
Global RPCs – processing a message – Sync HelloWorld application

```java
public class HelloWorldImpl
    implements HelloService {

    public HelloWorldImpl(ProviderContext session) {
        session.addRpcImplementation(
            HelloService.class,
            this);
    }

    @Override
    public Future<RpcResult<HelloWorldOutput>>
    helloWorld(HelloWorldInput input) {
        /* construct output */
        return RpcResultBuilder
            .success(helloWorldOutput)
            .buildFuture();
    }
}
```

**Diagram:**
- **MD-SAL**
  - `addRpcImplementation(this)`
- **helloWorldImpl**
  - `helloWorld(HelloWorldInput)`
  - `return: future`
Routed RPCs – E.g: HelloWorld application

```java
public class HelloWorldImpl1 implements HelloService {
    public HelloWorldImpl1(ProviderContext session) {
        RoutedRpcRegistration<HelloService> reg1 = session.addRoutedRpcImplementation(
            HelloService.class, this);
        reg1.registerPath(MyContext.class, iid1);
    }
    /* helloWorld() implementation works as before */
}

public class HelloWorldImpl2 implements HelloService {
    public HelloWorldImpl2(ProviderContext session) {
        RoutedRpcRegistration<HelloService> reg2 = session.addRoutedRpcImplementation(
            HelloService.class, this);
        reg2.registerPath(MyContext.class, iid2);
    }
    /* helloWorld() implementation works as before */
}
```
RPC – OpenFlow application

- Openflowplugindin/openflowplugin-impl/src/main/java/org.opendaylight/openflowplugin/impl/OpenFlowPluginProviderImpl.java

```java
import org.opendaylight.controller.sal.binding.api.RpcProviderRegistry;

@Override
public void setRpcProviderRegistry(final RpcProviderRegistry rpcProviderRegistry) {
    this.rpcProviderRegistry = rpcProviderRegistry;
}
```

- openflowplugin/test-provider/src/main/java/org.opendaylight/openflowplugin/test/OpenflowpluginTestServiceProvider.java

```java
RoutedRpcRegistration<SalFlowService> addRoutedRpcImplementation = ctx
    .<SalFlowService> addRoutedRpcImplementation(
        SalFlowService.class, this);

public RoutedRpcRegistration<SalFlowService> getFlowRegistration() {
    return flowRegistration;
}
```
Data Store

- Yang data is a tree-related data are modeled and represented as data tree to address any element / subtree
- Two Logical Data Stores
  - config
  - operational
- Unified View
- InstanceIdentifier:
  - Pointer to a node
Data Store Transactions

• Multicast asynchronous communications, sent by **Data Broker** if there is change in conceptual data tree, and is delivered to subscribed consumers

• Data Stores:
  • Operational Data Tree - published by the providers using MD-SAL, represents a feedback loop for applications to observe state of the network / system.
  • Configuration Data Tree - populated by consumers, represents intended state of the system / network

• Transaction Types
  • Transactional modification to conceptual data tree - write transactions `newWriteOnlyTransaction()`
  • Transactional reads from current data tree and not affected by any subsequent write - read-only transactions `newReadOnlyTransaction()`
  • Transaction provides both read and write capabilities – read-write transactions `newReadWriteTransaction()`

Modifications on Data tree:

• Put – stores a piece of data on specified path, act as add/replace
  • `void put(LogicalDatastoreType store, InstanceIdentifier<T> path, T data);`

• Merge – merges a piece of data on the existing data on specified path
  • `void merge(LogicalDatastoreType store, InstanceIdentifier<T> path, T data);`

• Delete – removes the whole subtree from the specified path
  • `void delete(LogicalDatastoreType store, InstanceIdentifier<> path);`
Simple Data Transaction Steps

Let assume initial state of data tree for PATH is A

1. Allocates new ReadWriteTransaction
   ```java
   ReadWriteTransaction rwTx = broker.newReadWriteTransaction();
   ```

2. Read from rwTx will return value A for PATH
   ```java
   rwRx.read(OPERATIONAL, PATH).get();
   ```

3. Writes value B to PATH using rwTx
   ```java
   rwRx.put(OPERATIONAL, PATH, B);
   ```

4. Read will return value B for PATH, since previous write occurred in same transaction
   ```java
   rwRx.read(OPERATIONAL, PATH).get();
   ```

5. Writes value C to PATH using rwTx
   ```java
   rwRx.put(OPERATIONAL, PATH, C);
   ```

6. Read will return value C for PATH, since previous write occurred in same transaction
   ```java
   rwRx.read(OPERATIONAL, PATH).get();
   ```
ReadWriteTransaction transaction = dataBroker.newReadWriteTransaction();
Optional<Node> nodeOptional;
nodeOptional = transaction.read(
    LogicalDataStore.OPERATIONAL,
    n1InstanceIdentifier);
transaction.put(
    LogicalDataStore.CONFIG,
    n2InstanceIdentifier, 
    topologyNodeBuilder.build());
transaction.delete(
    LogicalDataStore.CONFIG,
    n3InstanceIdentifier);
CheckedFuture future;
future = transaction.submit();
E.g: HelloWorld & OpenFlow applications

```java
public HelloWorldImpl(DataBroker db) {
    this.db = db;
    initializeDataTree(this.db);
}

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

```java
openflowplugin/openflowplugin-
impl/src/main/java/org/opendaylight/openflowplugin/impl/OpenFlowPluginProvi-
derImpl.java
@override
    public void setDataBroker(final DataBroker dataBroker) {
        this.dataBroker = dataBroker;
    }

ReadWriteTransaction modification =
dataBroker.newReadWriteTransaction();
    InstanceIdentifier<Group> path1 =
        InstanceIdentifier.create(Nodes.class)
            .child(Node.class,
                  testNode12.getKey()).augmentation(FlowCapableNode.class)
            .child(Group.class, new GroupKey(group.getGroupId()));
    modification.merge(LogicalDatastoreType.CONFIGURATION,
                       nodeToInstanceID(testNode12), testNode12, true);
    modification.merge(LogicalDatastoreType.CONFIGURATION,
                       path1, group, true);
    CheckedFuture<Void, TransactionCommitFailedException> commitFuture = modification.submit();
```
Notifications

• Represent asynchronous events with multicast communication, published by providers for consumers
• Used to model events originating in network devices or southbound plugins that are exposed to consumers for listening
• Defined with the notification statement:

```yaml
notification {
    ....
}
```
• NotificationService – is a notification broker that allows clients to subscribe and publish YANG-modeled notifications
• NotificationPublishService.java - interface to publish any YANG-modeled notification which will be delivered to all subscribed listeners
import org.opendaylight.controller.md.sal.binding.api.NotificationPublishService;
import org.opendaylight.controller.md.sal.binding.api.NotificationService;

@Override
public void setNotificationProviderService(final NotificationService notificationProviderService) {
    this.notificationProviderService = notificationProviderService;
}

@Override
public void setNotificationPublishService(final NotificationPublishService notificationPublishProviderService) {
    this.notificationPublishProviderService = notificationPublishProvide

Openflowplugin/test-provider/src/main/java/org.opendaylight/openflowplugin/test/OpenflowpluginTestCommandProvider.java

public void onSessionInitiated(final ProviderContext session) {
    notificationService = session.getSALService(NotificationService.class);
    // For switch events
    notificationService.registerNotificationListener(flowEventListener);
    notificationService.registerNotificationListener(nodeErrorListener);

    dataBroker = session.getSALService(DataBroker.class);

    ctx.registerService(CommandProvider.class.getName(), this, null);
    createTestFlow(createTestNode(null), null, null);
}
E.g. OpenFlow Pipeline Processing

- **OF Topology**
- **OF Switch**
  - **OpenFlow Java**
  - **OF Plugin**
- **Forwarding Rules Manager**
- **OF Inventory** (Operational)
- **OF Inventory** (Configuration)
- **Toplogy Manager**
- **Apps**
  - **NETCONF**
  - **RESTCONF**

**Data Change Notification**
**Data Store Write**
**RPCs/Notifications**
Using Futures for Logging

E.g. HelloWorld application - hello-world/impl/src/main/java/org.opendaylight/hello/impl/LoggingFuturesCallBack.java

```java
public class LoggingFuturesCallBack<
 implements FutureCallback<
     private Logger LOG;
     private String message;
     public LoggingFuturesCallBack(String message, Logger LOG) {
         this.message = message;
         this.LOG = LOG;
     }
     @Override
     public void onFailure(Throwable e) {
         LOG.warn(message, e);
     }
 }
```

E.g. In OpenFlow application - openflowplugin/test-provider/src/main/java/org.opendaylight/openflowplugin/test/OpenflowPluginBulkGroupTransactionProvider.java

```java
Futures.addCallback(commitFuture, new FutureCallback<Void>() {
    @Override
    public void onSuccess(Void aVoid) {
        ci.println("Status of Group Data Loaded Transaction: success.");
    }
});
```
Take-away: 3 Brokers

Data Broker
- put
- notify
- store

RPC Broker
- call

Notification Broker
- publish
- notify
Take-away: for Application Development

• RPCs:
  registerToHandle( RPC, implementation )
  call( RPC, input ) => output

• Data Store:
  put( key, value )
  get( key ) => value

• Notifications:
  subscribe( notification, callback )
  create( notification, data )
References

4. YANG - RFC 6020 - https://tools.ietf.org/html/rfc6020#section-4.2.10
5. GitHub - https://github.com/opendaylight/openflowplugin
controller-dev@lists.opendaylight.org
https://lists.opendaylight.org/mailman/listinfo
radhikamh at gmail dot com