# Consistent Regions - Toolkit Development Guide for Java Operators

For an operator to participate in a consistent region, the operator needs to implement StateHandler.  A state handler is responsible for handling the following call backs:

- Drain: Submit all pending tuples in the operator.  If your operator has internal buffers for storing tuples before they are submitted, the internal buffer must be drained.  If your operator is sending data to an external system, send all pending data.
- Checkpoint:  Save all applicable internal state into the checkpoint.
- Reset:  Reset operator internal state to those stored in the checkpoint.
- ResetToInitialState: Reset operator internal state to its initial state.  This is called if an application failure is detected before the first checkpoint can be saved.

Follow these steps to enable consistent region support in a Java Operator:

# Implement StateHandler Interface

1. In your Java operator class, implements the StateHandler interface:

```
public class MyJavaOperator extends AbstractOperator implements StateHandler {
     // operator code
     :
     :
}
```

2. Implement methods required by the StateHandler interface:

```
@Override
public void close() throws IOException {
 TRACE.log(TraceLevel.INFO, "StateHandler close", CONSISTENT_ASPECT);
}

@Override
public void checkpoint(Checkpoint checkpoint) throws Exception {
 TRACE.log(TraceLevel.INFO, "Checkpoint " + checkpoint.getSequenceId(), CONSISTENT_ASPECT);
}

@Override
public void drain() throws Exception {
 TRACE.log(TraceLevel.INFO, "Drain...", CONSISTEN_ASPECT);
}
@Override
public void reset(Checkpoint checkpoint) throws Exception {
 TRACE.log(TraceLevel.INFO, "Reset to checkpoint " + checkpoint.getSequenceId(), CONSISTENT_ASPECT);
}
@Override
public void resetToInitialState() throws Exception {
 TRACE.log(TraceLevel.INFO, "Reset to initial state", CONSISTENT_ASPECT);
}

@Override
public void retireCheckpoint(long id) throws Exception {
 TRACE.log(TraceLevel.INFO, "Retire checkpoint", CONSISTENT_ASPECT);
}
```

# Test Consistent Region API Implementations

At this point, you are ready to test that you have hooked up the StateHandler APIs correctly. In this test, we are not concerned that tuples are guaranteed to be processed.  Instead we want to make sure that the APIs are called as expected.

1. Create a SPL application that calls your operator
2. In the application, add a JobControlPlane operator.
3. Make your operator part of a consistent region
4. Compile the application and submit it to a Streams instance.  The Streams instance should be set up with checkpointing enabled.  When you submit the job, make sure the trace level is set to TRACE.
5. Let the job run for a while, so that checkpoints can be saved.
6. After a few checkpoints have been saved, select one of the operators in your application.  It is best to select an operator that you are not currently working on, so you can see how your operator will be called when an application failure is detected.  Restart the PE of the operator.
7. Restarting the PE will be viewed as an application failure by the runtime.  Your operator should be told to reset.
8. Wait for the job to become healthy again.

At this point, gather the PE trace for the operator that you are working on.  You should see the trace statements of your operator, and see that your operator has been called to drain, checkpoint and reset.

## Persisting and Restoring States

Once you have verified that your operator is called properly upon checkpoint and reset, you can now try to persist internal operator state to checkpoint and reset to it when needed. You need to identify all internal state that need to be persisted and restored. Streams provides APIs for you to easily persist and restore states to checkpoints. The APIs support any class that is **Serializable**. As an operator developer, you do not need to worry about what checkpoint backend is used.

Example code to persist and restore internal states:

```
String rulesetPath;
String ruleAppDirectory;

@Override
public void checkpoint(Checkpoint checkpoint) throws Exception {
    checkpoint.getOutputStream().writeObject(rulesetPath);
    checkpoint.getOutputStream().writeObject(ruleAppDirectory);
}

@Override
public void reset(Checkpoint checkpoint) throws Exception {
    try {
        rulesetPath = (String)checkpoint.getInputStream().readObject();
        ruleAppDirectory = (String)checkpoint.getInputStream().readObject();

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

## Persisting and Restoring Window

If your operator supports windowing, you will also need to handle saving and restoring of the window during checkpoint and reset. Streams provides built-in support to handle persistence and restoration of a window. For C++ operator, you have to add some simple code in the checkpoint and reset methods to handle this. For Java, this is done under the covers automatically and no extra code is required. However, it is a good idea to test your operator with consistent region and windowing enabled.

## Multi-threaded Considerations

When the Streams application is in checkpoint state or reset state, tuple flow is stopped. This is done by the runtime, and requires that a consistent region permit is acquired before an operator can submit tuples. A consistent region permit can be acquired by using the following code snippet:

In the initialize(…) method:

```
// crContext is a private variable in your operator class
crContext = context.getOptionalContext(ConsistentRegionContext.class);
```

Before you submit tuple:

```
Tuple dataTuple = dataOutputPort.newTuple();
try {
    crContext.acquirePermit();
    dataOutputPort.submit(dataTuple);
}
finally {
    crContext.releasePermit();
}
```

Please note that a consistent region permit is only required if you are submitting tuples on a background thread. A background thread is a thread that your operator has manually spawned off. This is usually done in a source operator. If you are submitting tuples in the **public void process(StreamingInput<Tuple> stream, Tuple tuple)** method, you do not need to explicitly acquire the permit before tuple submission. The permit is acquired for your operator before the process method is called.

### References

The Streams Knolwedge Center has detailed information on implementing operators that use consistent regions.

StateHandler API Reference