Consistent Regions - Toolkit Development Guide for C++ Operators

For an operator to participate in a consistent region, the operator needs to be a 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: Serialize 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.

Here's a very useful sample to demonstrate how this can be done: `/samples/spl/feature/ConsistentRegion`

Follow these steps to implement the StateHandler APIs in a C++ operator:

Implement StateHandler API

1. Add StateHandler.h include path to *_h.cgt file:

```
<%
    my $isInConsistentRegion =
    $model->getContext()->getOptionalContext("ConsistentRegion");
    my @includes;
    if ($isInConsistentRegion) {
        push @includes, "#include <SPL/Runtime/Operator/State/StateHandler.h>";
    }
    SPL::CodeGen::headerPrologue($model, \@includes);
%>
```

2. Subclass from StateHandler class in *_h.cgt:

```
class MY_OPERATOR : public MY_BASE_OPERATOR
<%if ($isInConsistentRegion) {%>
    , StateHandler
<%}%>
{
    public:
       // Constructor
       MY_OPERATOR();

      // Destructor
      virtual ~MY_OPERATOR();

      // more code
      ...
}
```

3. Define StateHandler methods in *_h.cgt :

```
 <%if ($isInConsistentRegion) {%>
    // Callbacks from StateHandler.h
    virtual void checkpoint(Checkpoint & ckpt);
    virtual void reset(Checkpoint & ckpt);
    virtual void resetToInitialState();
    virtual void drain();
    virtual void retireCheckpoint(int64_t id);
 <%}%>
```

4. Consistent region context

Optionally create a private member in the operator class for storing consistent region context. This is needed for your operator to get information about consistent region at runtime, and also required for acquiring any locks before tuples can be submitted. Add this in the header file, as a private member:

```
<%if ($isInConsistentRegion) {%>
    ConsistentRegionContext *_crContext;
<%}%>
```

5. Consistent region context at compile time

In *_cpp.cgt file, add a perl variable at the beginning of the file for us to get consistent region context at compile time. This allows us to optionally generate code if the operator is in a consistent region.

```
 <%
    my $isInConsistentRegion = $model->getContext()->getOptionalContext("ConsistentRegion");
 %>
```

6. Implement StateHandler methods in *_cpp.cgt file:

In the *_cpp.cgt file, add the following methods:

```
<%if ($isInConsistentRegion) {%>
void MY_OPERATOR::checkpoint(Checkpoint & ckpt)
{
    // TODO:  persist state when called.
    SPLAPPTRC(L_TRACE, "Checkpoint: " << ckpt.getSequenceId(), "CONSISTENT");
}

void MY_OPERATOR::reset(Checkpoint & ckpt)
{
    // TODO:  Restore state from checkpoint when called
    SPLAPPTRC(L_TRACE, "Reset: " << ckpt.getSequenceId(), "CONSISTENT");
}

void MY_OPERATOR::resetToInitialState()
{
    // TODO:  Reset to operator initial state when called
    SPLAPPTRC(L_TRACE, "Reset to Initial State. ", "CONSISTENT");
    // may have to undo any state changes during the processing
}
void MY_OPERATOR::drain() {
    // TODO:  Drain operator
    SPLAPPTRC(L_TRACE, "Drain Operator", "CONSISTENT");
}
void MY_OPERATOR::retireCheckpoint(int64_t id) {
    SPLAPPTRC(L_TRACE, "Retire Checkpoint: " << id, "CONSISTENT");
}


<%}%>
```

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 state to checkpoints. The APIs support primitives and STL collections. As an operator developer, you do not need to worry about what checkpoint backend is used. Example code to persist and restore internal state:

```
std::tr1::unordered_map<uint32_t, uint32_t> _myState;
void checkpoint(Checkpoint & ckpt) {
    ckpt << _myState;
}
void reset(Checkpoint & ckpt) {
    ckpt >> _myState;
}
void resetToInitialState() {
    _myState.clear();
}
```

Persisting and Restoring Window

If your operator supports windowing, you will also need to handle saving and restoring 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 cover automatically and no extra code is required.

Example code to save window into a checkpoint:

```
<%if ($isInConsistentRegion) {%>
void MY_OPERATOR::checkpoint(Checkpoint & ckpt)
{
    SPLAPPTRC(L_DEBUG, "Before checkpoint window is: " << _window.toString(), SPL_OPER_DBG);
    _window.checkpoint(ckpt);
}
```

Example code to restore window on reset:

```
void MY_OPERATOR::reset(Checkpoint & ckpt)
{
    _window.reset(ckpt);
    SPLAPPTRC(L_DEBUG, "After reset window is: " << _window.toString(), SPL_OPER_DBG);
}
```

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:

```
while(!pe.getShutdownRequested() {
{
    {
        ConsistentRegionPermit crm(_crContext);
        …
        // Change state
        _numTuples++;
        // Construct tuple
        …
        // Submit tuples
        submit(tuple, 0);
    }
}
```

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 `void MY_OPERATOR::process(Tuple const & tuple, uint32_t port)` 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