Instrumentation Facility

The Ice run time can be instrumented using observer interfaces that monitor many aspects of the run time's internal objects and activities, including connections, threads, servant dispatching, proxy invocations, endpoint lookups and connection establishment. We refer to these internal objects and activities as "instrumented objects" in the discussion below. The application is responsible for implementing the instrumentation observer interfaces. Note however that an implementation of these interfaces is provided by the Metrics Facet, so most applications do not need to implement them and can instead collect metrics through the `MetricsAdmin` facet. The definition of the instrumentation interfaces can be found in the `Ice/Instrumentation.ice` Slice file.

The Ice run time uses the `Ice::Instrumentation::CommunicatorObserver` interface to obtain observers for instrumented objects created by the run time:

Slice

```
module Ice {
    module Instrumentation {
        local interface CommunicatorObserver {
            Observer getConnectionEstablishmentObserver(Endpoint endpt, string connector);
            Observer getEndpointLookupObserver(Endpoint endpt);
            ConnectionObserver getConnectionObserver(ConnectionInfo c, Endpoint e,
                                                     ConnectionState s, ConnectionObserver o);
            ThreadObserver getThreadObserver(string parent, string id, ThreadState s,
                                             ThreadObserver o);
            InvocationObserver getInvocationObserver(Object* prx, string operation, Context ctx);
            DispatchObserver getDispatchObserver(Current c);
            void setObserverUpdater(ObserverUpdater updater);
        };
    };
    local interface Communicator {
        Ice::Instrumentation::CommunicatorObserver getObserver();
        // ...
    };
};
```

The Ice run time calls the appropriate `get...Observer` operation each time a new instrumented object is created. The implementation of these methods should return an observer, or nil if the implementation does not want to monitor the instrumented object. This observer is associated with the instrumented object and receives notifications of any changes to its attributes or state. All observer interfaces derive from the `Ice::Instrumentation::Observer` base interface:

Slice

```
local interface Observer {
    void attach();
    void detach();
    void failed(string exceptionName);
};
```

The `attach` operation is called upon association of the observer with the new instrumented object. The `detach` operation is called when the object is destroyed. The `failed` operation is called to report any failures that might occur during the lifetime of the instrumented object. Observer specializations provide additional operations for monitoring other attributes. For example, here is the `Ice::Instrumentation::ConnectionObserver` interface:

Slice

```
local interface ConnectionObserver extends Observer {
    void sentBytes(int num);
    void receivedBytes(int num);
};
```

The `sentBytes` and `receivedBytes` methods are called by the Ice connection when new bytes are received or sent over the connection.

Shown below is an implementation of the communicator and connection observer interfaces to record sent and received bytes on a per-connection basis. The observer dumps how many bytes were received and sent for the connection when it is detached:

C++

```
class ConnectionObserverImpl : public Ice::Instrumentation::ConnectionObserver {
public:
    ConnectionObserverImpl(const Ice::ConnectionInfoPtr& connInfo) :
        info(connInfo), sentBytes(0), receivedBytes(0)
    {
    }

    void attach() {}

    void detach()
    {
        cerr << info->remoteHost << ":" << info->remotePort << ": sent bytes = "
             << sentBytes << ", received bytes = " << receivedBytes << endl;
    }

    void sentBytes(int num)
    {
        sentBytes += num;
    }

    void receivedBytes(int num)
    {
        receivedBytes += num;
    }

    void failed(const std::string&)
    {
    }

private:

    Ice::ConnectionInfoPtr info;
    long sentBytes;
    long receivedBytes;
};

class CommunicatorObserverImpl : public Ice::Instrumentation::CommunicatorObserver {
public:

    Ice::Instrumentation::ConnectionObserverPtr
    getConnectionObserver(const Ice::ConnectionInfoPtr& c, const Ice::EndpointPtr& e,
                          Ice::Instrumentation::ConnectionState s,
                          const Ice::Instrumentation::ConnectionObserverPtr& previous)
    {
        return new ConnectionObserverImpl(c);
    }
};
```

For brevity we have omitted the implementation of the other `get...Observer` methods; they all return 0 as we are only interested in instrumenting connections.

To register your implementation, you must pass it in an `InitializationData` parameter when you initialize a communicator:

C++

```
Ice::InitializationData id;
id.observer = new CommunicatorObserverImpl();
Ice::CommunicatorPtr ic = Ice::initialize(id);
```

You can install a `CommunicatorObserver` object on either the client or the server side (or both). Here is some example output produced by installing our `CommunicatorObserver` and `ConnectionObserver` object implementations in a simple server:

```
127.0.0.1:3487: sent bytes = 14, received bytes = 32
127.0.0.1:3487: sent bytes = 33, received bytes = 14
127.0.0.1:3490: sent bytes = 14, received bytes = 14
...
```

In addition to the operations for retrieving observers, the `CommunicatorObserver` interface also defines a `setObserverUpdater` operation that is called by the Ice run time on initialization to provide an updater object to the `CommunicatorObserver` implementation. This updater object can be used to "refresh" some of the created observers. The updater object provided by the Ice run time implements the following interface:

Slice

```
local interface ObserverUpdater {
    void updateConnectionObservers();
    void updateThreadObservers();
};
```

The `CommunicatorObserver` implementation can call these operations to update the observers associated with Ice connections or threads. When one of these operations is called, the Ice run time calls the matching `get...Observer` method on the `CommunicatorObserver` interface for each of the instrumented objects. For example, if you call `updateConnectionObservers`, your implementation of `getConnectionObserver` will be called again for each Ice connection in the communicator. The `previous` parameter to `getConnectionObserver` represents the observer that is currently associated with the connection.

This mechanism can be used to re-configure the observers associated with instrumented objects. For instance, the application might not wish to instrument connections all the time but only when needed. It can use the observer updater to enable or disable the instrumentation. Here is the example above modified to provide this functionality:

C++

```
class CommunicatorObserverImpl : public Ice::Instrumentation::CommunicatorObserver,
                                 private IceUtil::Mutex {
public:

    Ice::Instrumentation::ConnectionObserverPtr
    getConnectionObserver(const Ice::ConnectionInfoPtr& c, const Ice::EndpointPtr& e,
                          Ice::Instrumentation::ConnectionState s,
                          const Ice::Instrumentation::ConnectionObserverPtr& previous)
    {
        Lock sync(*this);
        return enabled ? new ConnectionObserverImpl(c) : 0;
    }

    void setEnabled(bool enabled)
    {
        {
            Lock sync(*this);
            if(this->enabled == enabled)
                return;
            this->enabled = enabled;
        }
        updater->updateConnections();
    }

    void setObserverUpdater(const Ice::Instrumentation::ObserverUpdaterPtr& updater)
    {
        this->updater = updater;
    }

    const Ice::Instrumentation::ObserverUpdaterPtr updater;
    bool enabled;
};
```

As you can see in the example above, special care needs to be taken with respect to synchronization. The Ice run time can call observers with Ice internal locks held to guarantee consistency of the information passed to the `get...Observer` methods. It is therefore important that the implementation of your observers performs quickly and does not create deadlocks. Your observers should not make remote invocations or call Ice APIs that require acquiring locks on instrumented objects.

See Also

- Communicator Initialization
- The Metrics Facet