Writing Borealis Applications

September 24, 2004

1 Intro

2 Running an Application

1. Start the central catalog:
   
   CentralLookupServer [-d cls_ip:cls_port] [-t duration]

   The first option sets the ip and port number on which the CentralLookupServer will listen for RPC requests. The default is: 127.0.0.1:30000. The second parameter serves to turn the server off after a predefined time period (in milliseconds). It can be handy when running experiments.

2. Start one or more borealis nodes:
   
   export BOREALIS_CLS = cls_ip:cls_port
   borealis [-d borealis_ip:borealis_port] [-t duration] [-f medusa_config.xml]

   Borealis uses the BOREALIS_CLS environment variable to find the CentralLookupServer. The -d option is the ip and port on which the borealis node should listen for TCP-RPC requests. Borealis will also listen for equivalent XML-RPC requests on port: borealis_port+1. The -f option indicates the name of the configuration file for this borealis node. The configuration file specifies HA and load management parameters. For HA, the file indicates the static assignments of secondary and primary pairs. For load management, the file indicates the contracts between nodes. This file is not necessary if you don’t care about HA nor contract-based load management.

3. Finally, you can start an application. The simplest test application is:
   
   utility/test/simple/mytest -d borealis_ip:borealis_port

3 Writing a Client Application for Borealis

A typical Borealis application performs one or more of the following tasks:

1. Define the name and schemas of the system-wide input streams
2. Submit queries that operate on the pre-defined input streams or on streams produced by previously defined queries.
3. Subscribe to receive output streams
4. Push data on input streams.

Borealis clients communicate with Borealis nodes either with TCP-RPC or XML-RPC. An example of applications that uses TCP-RPC is in: borealis/utility/test/simple/rpctest.cc. An example of application that uses XML-RPC is in: utility/statsView/src/monitor. It uses the library provided under utility/medusaXmlRPC. This library is a bit out of date and not quite at the right level of abstraction but it’s a starting point. Applications can send only control messages using XML-RPC.
To make this communication easier, however, we also provide an API. To use this API, a client application instantiate `MedusaClient` object.

```cpp
// Client for Borealis node
ptr<MedusaClient> client(new MedusaClient(InetAddress(borealis_ip,borealis_port)));
```
or
```cpp
MedusaClient client(InetAddress(borealis_ip,borealis_port));
```
where `borealis_ip` and `borealis_port` are the IP and port number of the Borealis node to which the client should send its requests. The code for the `MedusaClient` class is under: `borealis/src/modules/common`.

In the below examples, we use the `MedusaClient` API to communicate with Borealis nodes. Section 4 enumerates the methods available through the `MedusaClient` API.

### 3.1 Defining Input Streams

To define an input stream, we must first define its schema. Schemas are defined with XML as in the below example:

```cpp
int PROT_SIZE = 4;
string schema_xml = string() +
"<schema name="medusa://nms.lcs.mit.edu/ids/packet_header_schema">
  <field name="start_time_sec" type="int"/>
  <field name="protocol" type="string" size="" + PROT_SIZE + ">/n" +
"</schema>"
;
```

From the XML description, we can instantiate a `Schema` object and send it to borealis:

```cpp
Schema schema;
Status s = schema.from_xml(schema_xml);
if ( !s )
  FATAL << "Failed parsing schema from xml" << s;
RPC<void> r = client->create_schema(schema);
if (!r.stat()) {
  FATAL << "Failed creating a schema: " << r;
}
```

To define a stream, we simply define a name and associate it with a schema:

```cpp
string input_stream_name = "medusa://nms.lcs.mit.edu/ids/packet_header/";
StreamDef input_stream_def(input_stream_name, schema);
r = client->create_stream(input_stream_def);
if (!r.stat()) {
  FATAL << "Failed creating a stream: " << r;
}
```

### 3.2 Submitting Queries

A query is a chunk of the complete query network. A query lists a set of operators, their parameters, and the streams that should serve as input. When a client submits a query to a Borealis node, the input streams must already exist. They must have either been created as described above or they must be the output streams of a previously defined query. Clients can submit queries to any Borealis node. A query is created and started as illustrated below:

```cpp
string output_stream_name = "medusa://nms.lcs.mit.edu/ids/suspicious/";
string query_name = "medusa://nms.lcs.mit.edu/ids/simple_query/";
string box_name = query_name + "/box";
```
string query_xml = string() +
"<query name="" + query_name + "">n" +

    // For every 1 seconds count the number of packets
"<box name="" + box_name + "" type="aggregate">n" +
"<input port="1" stream="" + input_stream_name + ""/>n" +
"<output port="1" stream="" + output_stream_name + ""/>n" +
"<param name="aggregate-function.0" value="count()">n" +
"<param name="window-size-by" value="VALUES">n" +
"<param name="window-size" value="1">n" +
"<param name="advance" value="1">n" +
"<param name="order-by" value="FIELD">n" +
"<param name="order-on-field" value="start_time_sec">n" +
"</box>n" +
"</query>n";

r = client->create_query_xml(query_xml);
if (!r.stat()) {
    FATAL << "Failed creating a query: " << r;
}

r = client->start_query(query_name);
if (!r.stat()) {
    FATAL << "Failed starting a query: " << r;
}

3.3 Pushing Data
To send data to Borealis, the client must open a data path to a specific Borealis node identified with
borealis_ip:borealis_port:

if (!client->set_data_path(max_buffer,borealis_ip,borealis_port)) {
    FATAL << "Failed setting data path";
}

where max_buffer is the maximum size for output buffers. When a client sends data, the data is copied
to the buffer. If the client sends too fast, the buffer simply grows. We limit this growth and return
error messages when clients try to send too fast.

Once the data path exists, the client can send tuples in batches wrapped in StreamEvent objects:

ptr<StreamEvent> event(new StreamEvent(input_stream_name));
event->m_inject = true;
event->insert_bin(string((const char *)&buffer_with_tuples,buffer_size));

Status stat = client->fast_post_event(event); // Non blocking
if (stat) {
    // All ok
} else if (stat.as_string() == DataHandler::NO_SPACE) {
    // The client is trying to send too fast... slow down
} else {
    ERROR << "Connection closed... stopping event stream";
    return;
}
3.4 Subscribing to Receive a Stream

Before subscribing to receive streams, a client must set up a handler for incoming tuples. The method that performs the setup takes two parameters. The first parameter defines the IP and port on which the client should listen for data. As a second parameter, the client indicates the callback method that should be invoked every time an StreamEvent arrives.

```cpp
// Subscribing to receive the output on fast datapath
Status stat = client->set_data_handler(InetAddress(my_ip,my_port),wrap(&print_event));
```

The callback method takes `ptr<StreamEvent>` as argument. For example, the signature of the `print_event` method is: `Status print_event(ptr<StreamEvent> event)`.

Once a handler exists, the client can subscribe to streams as follows:

```cpp
Status stat = client->fast_subscribe(output_stream_name);
Status stat = client->fast_subscribe(some_other_stream_name);
```

The client must also start its event loop to continuously listen for events. The call below does not return:

```cpp
client->run();
```

4 Client API

The table below summarizes the MedusaClient API. Streams are defined with StreamDef objects. Tuples are sent to and from Borealis nodes encapsulated in StreamEvent objects.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MedusaClient(InetAddress remote)</td>
<td>Constructor</td>
</tr>
<tr>
<td>RPC&lt;void&gt; create_schema(Schema schema)</td>
<td>Defines a new schema</td>
</tr>
<tr>
<td>RPC&lt;void&gt; create_stream(StreamDef streamdef)</td>
<td>Defines a new stream</td>
</tr>
<tr>
<td>RPC&lt;void&gt; create_cp(CPViewDescription view_desc, StreamDef streamdef);</td>
<td>Creates a connection point view on a stream</td>
</tr>
<tr>
<td>RPC&lt;void&gt; create_query(Query query)</td>
<td>Submits a new query</td>
</tr>
<tr>
<td>RPC&lt;void&gt; create_query_xml(string query)</td>
<td>Submits a new query in XML format</td>
</tr>
<tr>
<td>RPC&lt;void&gt; set_query_status(Name name, QueryStatus status)</td>
<td>Starts, stops, or deletes a previously defined query</td>
</tr>
<tr>
<td>RPC&lt;void&gt; start_query(Name name)</td>
<td>Convenience method that changes the status of the name query to RUNNING</td>
</tr>
<tr>
<td>RPC&lt;vector&lt;Stats&gt;&gt; &gt; get_stats()</td>
<td>Returns load statistics for the contacted node</td>
</tr>
<tr>
<td>Status set_data_path(int max_buffer, string ip, int port, bool blocking = true)</td>
<td>Opens a data path to Borealis node at ip:port</td>
</tr>
<tr>
<td>Status fast_post_event(ptr&lt;StreamEvent&gt; evt)</td>
<td>Sends tuples on previously opened datapath</td>
</tr>
<tr>
<td>Status set_data_handler(InetAddress listen, DataHandler::DHCallback cb)</td>
<td>Starts listening for incoming tuples at given address. When tuples arrive, invokes the callback given as the second argument.</td>
</tr>
<tr>
<td>Status fast_subscribe(StreamDef stream_def)</td>
<td>Subscribes the client to the given stream</td>
</tr>
<tr>
<td>IOEventLoop&amp; get_loop()</td>
<td>Returns a reference to the client’s event loop. Useful when scheduling periodic events</td>
</tr>
<tr>
<td>void run()</td>
<td>Start the client’s event loop</td>
</tr>
<tr>
<td>void terminate()</td>
<td>Terminates the client’s event loop</td>
</tr>
</tbody>
</table>