Component Based Tool Framework
Overview & Tutorial

The Component Based Tool Framework (hereafter "CBTF"), developed by the Krell Institute[1] under contract for the U.S. Department of Energy, is an open-source software framework for constructing debugging and performance tools (hereafter simply "tools") from a set of reusable components. This document describes the design goals for CBTF, its underlying concepts, and implementation details. Additionally, tutorials illustrating how to create and use CBTF components are provided.

Design Goals

Facilitating increased software componentization of tools was the principal design goal for CBTF. There are many benefits to such increased componentization:

- Less Duplicated Effort: Virtually all tools require certain key functionality such as data collection, symbol table processing, and view generation. Because existing tools tend to be monolithic, this functionality is usually designed and implemented anew for every new tool. When this functionality doesn't represent the developer's primary research interest, it represents a huge wasted effort.

- Increased Reconfigurability: Modern supercomputers - especially at petascale and above - vary widely in their architecture. This presents a challenge for tools. A particular configuration of data collection and processing that works well for one architecture may be completely infeasible for another. Componentizing tools allows improved reconfigurability for different architectures.

- Improved Scalability: During the last decade, the largest supercomputers have seen their node counts increase by several orders of magnitude. This trend is expected to continue for the exascale systems projected to be available by 2020. In order for tools to handle the quantity of data collected on such systems, multiple levels of data processing will be required. By componentizing tools, data processing can be more easily replicated and distributed.

Underlying Concepts

CBTF is based on the notion of dataflow programming[2]. Individual software components are modeled as black boxes that accept messages on zero or more inputs, internally perform some task, and then produce messages on zero or more outputs:

---

[1] http://www.krellinst.org

[2] http://en.wikipedia.org/wiki/Dataflow_programming

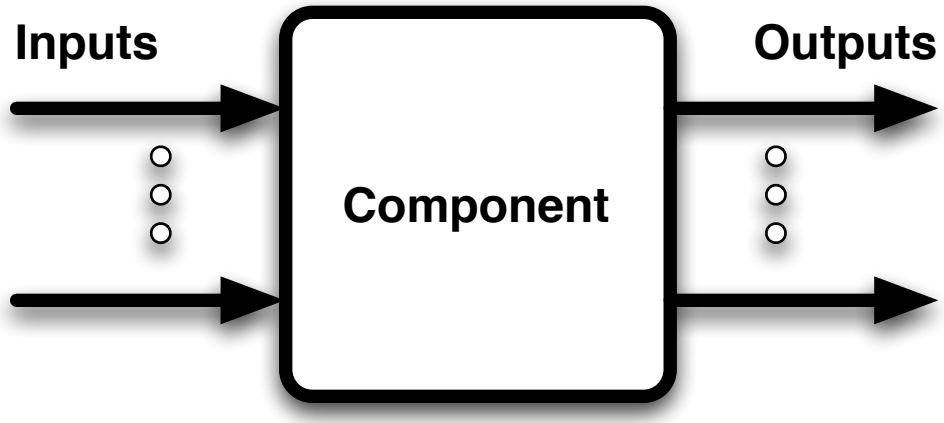

Figure 1: Abstract Model of a CBTF Component

Networks of these components are then constructed by connecting together the inputs and outputs of one or more component instances:

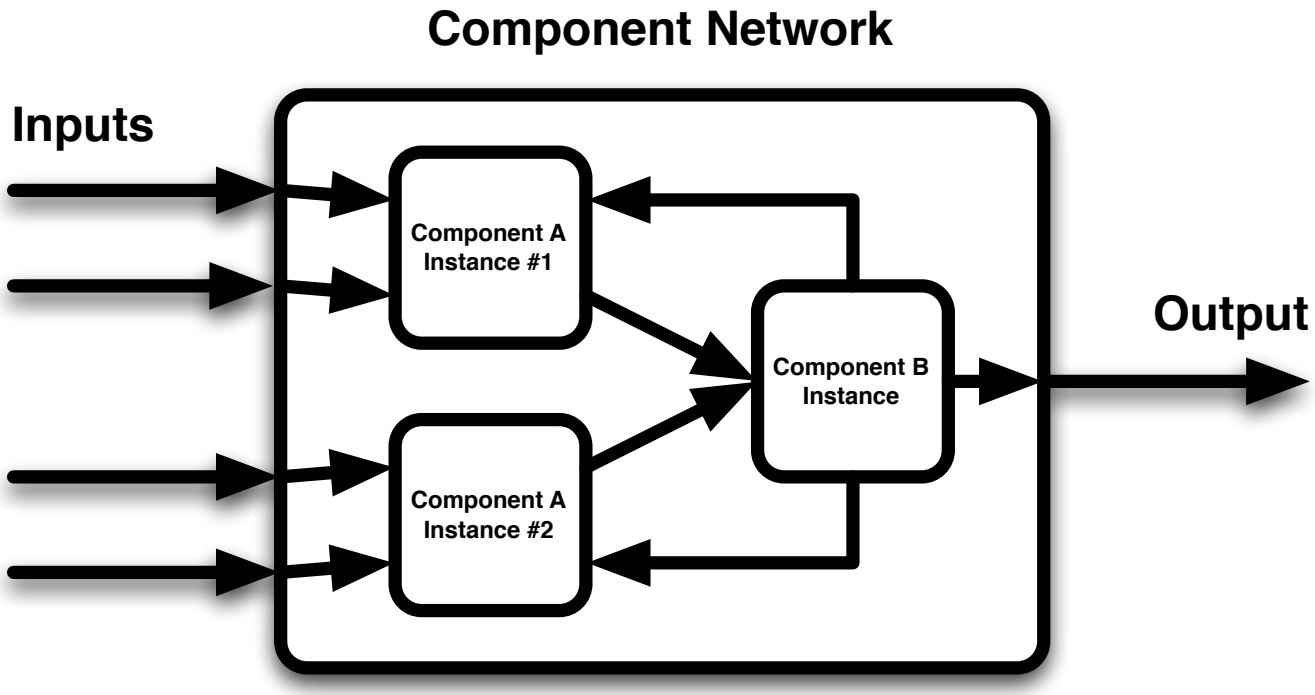

Figure 2: Example of a CBTF Component Network

Several important aspects of CBTF component networks are illustrated in the figure above:

- As with C++ where one can create multiple instances of a class, more than one instance of a given component type may be used within a single component network.
- Component networks need not be simple, straight, pipelines. Such configurations are perfectly legal, of course. But more complex arrangements - including cycles - are allowed as well.

- Connections between components need not be one-to-one. It is possible for a single component output to be attached to multiple inputs. And, alternatively, multiple component outputs may be attached to a single input.

- Component networks themselves can be viewed as components. This allows component networks to be built out of other component networks.

Finally, the component network concept can be extended across more than one node, resulting in a distributed component network:

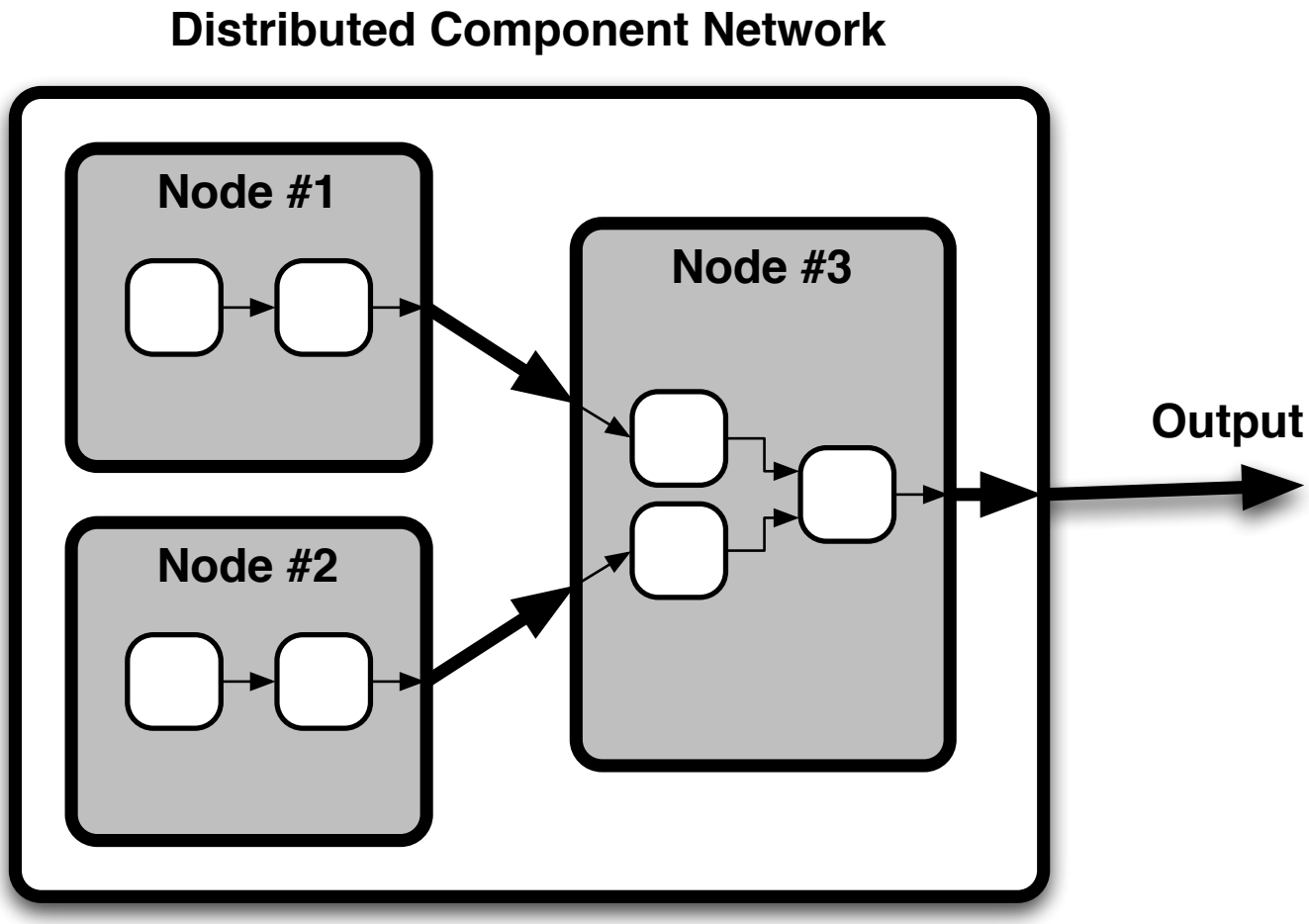

Figure 3: Example of a CBTF Distributed Component Network

Note that none of these concepts are specific to tools. I.e. CBTF has in no way dictated the scope of what each component does or the content of the messages exchanged between those components. While originally envisioned as a framework for constructing tools, CBTF can actually be used to construct virtually any software system for which the dataflow programming paradigm is a sensible choice.

Implementation Details

CBTF is currently split into three layered C++ software libraries, with each layer building upon the capabilities of the lower layers:

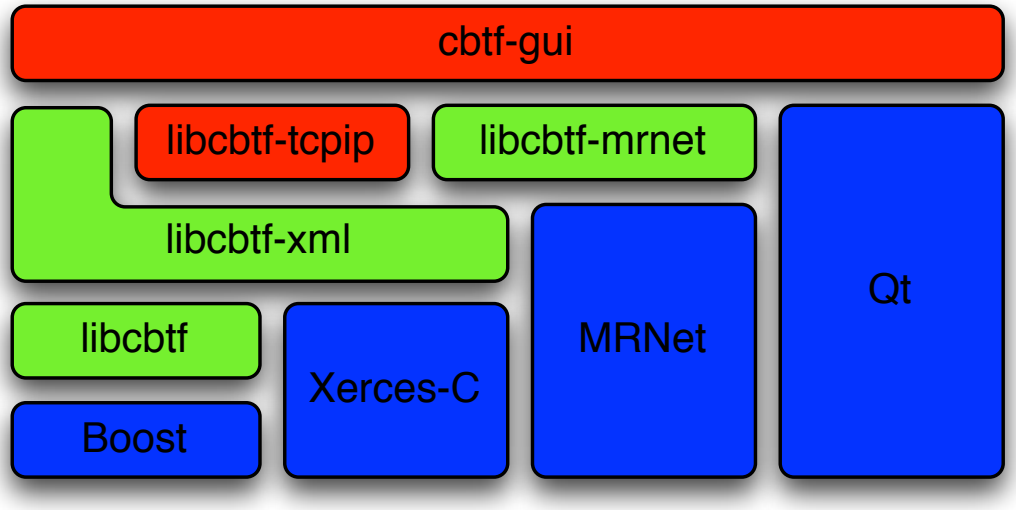

Figure 4: CBTF Software Stack

Blue in the above diagram denotes the following existing open source software packages:

Boost	http://www.boost.org/
Xerces-C	http://xerces.apache.org/xerces-c/
MRNet	http://www.paradyn.org/mrnet/
Qt	http://qt.nokia.com/

Green denotes CBTF software libraries that have been completed to date. Red denotes planned future CBTF functionality. One can utilize a lower-level library (e.g. libcbtf) without requiring the higher-level library (e.g. libcbtf-mrnet) but not vice-versa.

The CBTF source code is heavily documented using Doxygen (http://www.doxygen.org/) and this documentation should be considered the definitive reference for the APIs of the CBTF libraries. Some of the details of each library are discussed below, however, to provide the reader with further context for understanding the tutorials.

libcbtf

This library provides the basic mechanisms to perform the following tasks:

- Defining Components: All CBTF components are implemented as C++ classes that inherit from the `Component` base class defined in libcbtf. Such component classes provide a default constructor that declares the component's type, version, inputs, and outputs via the `Component` class's

protected methods. Additionally, components must also provide a factory function for creating component instances. Beyond these requirements component classes are free to define any additional methods, data members, etc. that are required to accomplish their functionality.

- Registering Plugins: One or more CBTF component classes can be bundled together into a single shared library. This shared library can then by dynamically loaded as a plugin by a tool using the `Component` class' `registerPlugin` static method. The components bundled in the plugin are then available for instantiation or metadata queries via the APIs described below. Note that CBTF does not provide a specific plugin discovery policy - only the mechanism by which plugins are loaded.

- Query Metadata: Once plugins containing CBTF component classes have been registered, a tool can dynamically determine the available components, their versions, inputs, and outputs via static methods defined by the `Component` class. This allows tools to utilize components in a generic manner - at least on a limited basis - without prior knowledge of what those components are.

- Instantiate Components: New instances of a CBTF component are created using the `Component` class' `instantiate` static method. The caller may optionally specify a specific component version to be instantiate, with the default being to instantiate the most recent available version. Component instances are only returned as shared pointers. Thus all component instances are automatically destroyed when their last shared pointer is released.

- Connect Components: Finally, multiple CBTF components can be connected into a component network using the `Component` class' `connect` static method. Existing connections may also be removed via the `disconnect` static method.

Alone, with minimal additional dependencies (i.e. only Boost), this library can be used to define components. And while this library can, in theory, be used to construct tools from components, it is fairly tedious to build large component networks using individual calls to the `connect` method. This tedium is addressed by libcbtf-xml, described below.

Versioning

Each CBTF component has a version number associated with it and multiple versions of the same component can coexist within a single tool. The standard GNU version numbering scheme has been adopted for CBTF, with version numbers consisting of a major.minor.maintenance triplet. Specific meanings have been adopted for each element in the triplet:

- Major: This number, initially zero, is incremented every time a change is made to a component's interface. E.g. if a new required input is added to component.

- Minor: This number, initially zero, is incremented every time a change is made to a component's semantics without changing its interface. E.g. a data aggregation component changes aggregation algorithms.

- Revision: This number, initially zero, is incremented every time a change is made to a component without changing its interface or semantics. E.g. a bug fix is made.

The metadata interface provided by libcbtf allows a tool to dynamically - at runtime - discover the available versions of a given component. And its instantiation interfaces allow a specific component version to be requested.

Threading Model

Multiple threads may call libcbtf (or libcbtf-xml and libcbtf-mrnet for that matter) concurrently without corrupting the internal state of those libraries. I.e. the CBTF libraries are “thread safe”. At the same time, CBTF itself does not impose any specific threading model upon components or tools.

When a thread provides an input to a CBTF component, that thread executes the handler associated with that input. If that handler causes a value to be emitted on one of the component’s outputs, that same thread will execute the handler of the second component’s input. The thread will continue to execute handlers in a sort of “depth first” manner until all chained handlers have been exhausted. Of course if the component network contains cycles (as in figure 2 above), this could result in an endless recursion - at least in theory. In practice, the process would be killed after the thread exhausted its stack.

However, various other threading models are possible which would address the issue of potential endless recursion due to cycles in the component network. A component can, for example, create within its constructor a thread that implements the component’s functionality. Inputs, then, might simply push values onto a thread-safe queue, with the implementation pulling them off as they become available. If component A and B in figure 2 were implemented in this way, there would be no danger of an endless recursion causing stack overflow and a crash.

This lack of a specific thread model was intentionally designed into CBTF in order to given the user maximum flexibility in designing their components and tools.

libcbtf-xml

The CBTF XML library simplifies the construction of complex component networks by allowing the instantiated components, and the connections between them, to be entirely specified within a single XML file. A schema for these files (“Network.xsd”) can be found in the libcbtf-xml source code, but the structure of an XML component network description can be broken down into the following sections:

- Type & Version: All CBTF component networks are themselves abstracted as a single component with inputs and outputs. Thus we must specify the network’s type and version just as if it were a component written directly in C++. The `<Type>` and `<Version>` tags are used to provide this information.

- Search Paths & Plugins: CBTF must be told where to find the plugins containing the components instantiated by the component network. The `<Plugin>` tag is used to specify the plugins, and the `<SearchPath>` tag can be used to specify paths to be searched for those plugins. One important thing to note is that a plugin can be a shared library containing components as described in the previous section. But it can also name a XML file describing a network since these are components too. I.e. component networks can be combined together to build other component networks.

- Components: This is a list of the component network's instances. One or more occurrences of the `<Component>` tag are used to specify these instances. In addition to providing the component type and, optionally, the minimum and/or maximum version to be instantiated, each instance has a symbolic name. That name is used within the following two sections to describe the instance's connections to other instances, and its' connections outside this component network.

- Connections: Each `<Connection>` tag specifies a single connection between two component instances within the component network.

- Inputs & Outputs: Each `<Input>` or `<Output>` tag has similar purpose to the `<Connection>` tags above. The difference is the these tags connect a single component instance to the exterior of the component network. I.e. the XML description of figure 2 above would contain four `<Input>` tags and a single `<Output>` tag.

These XML files are registered with CBTF via this library's one and only externally visible API function - `registerXML`. Since component networks are, themselves, components, they can be instantiated and connected using the mechanisms described above once their XML file has been registered.

libcbtf-mrnet

The libcbtf-mrnet library builds upon the concepts and design of libcbtf-xml, but extends them to a distributed component network built upon the University of Wisconsin's MRNet (Multicast Reduction Network) software. Like libcbtf-xml, this library uses XML to describe distributed component networks. And a schema for this library's XML files ("MRNet.xsd") can be found in the libcbtf-mrnet source code.

Before delving into the XML for libcbtf-mrnet distributed component networks, a couple additional concepts must be introduced:

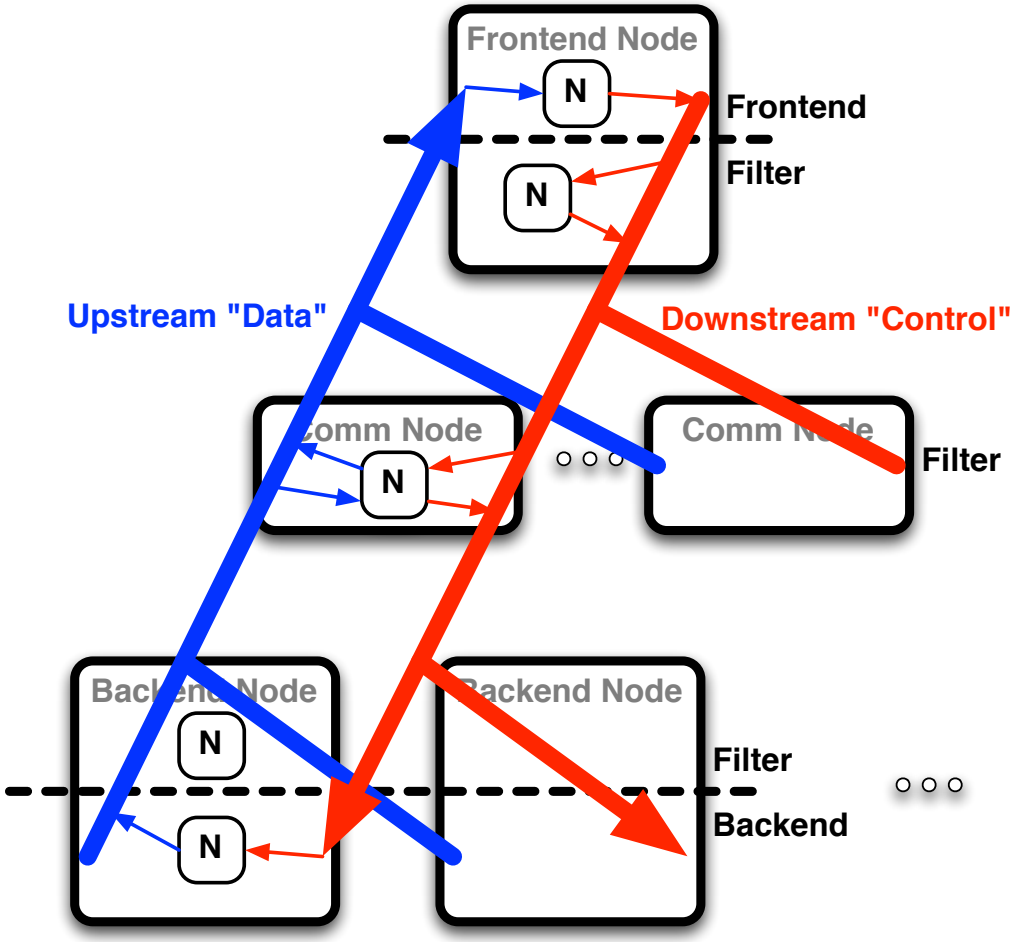

Figure 5: Example MRNet Distributed Component Network

Figure 5 above illustrates an example distributed component network as supported by libcbtf-mrnet. Each “N” box represents a non-distributed component network. I.e. each frontend, filter, and backend can contain an arbitrarily complex network of components. Within a single level of the MRNet tree, all nodes must have identical networks. It is not allowed, for example, to place network A on half the backend nodes and network B on the other half. It is allowed, however, to have one level of filters to have a different network than the other levels. And in fact networks need not be present at all levels of the MRNet tree.

In this figure, there are two uniquely named streams. The first, shown in red, is a downward traveling stream of data named (fairly generically) “Control”. The second, shown in blue, is a upward traveling stream of data named (again, fairly generically) “Data”. Although the above figure shows one stream flowing in each direction, in reality the number of streams is not fixed. A distributed component network can defined as many - or as few as zero - streams in each direction.

Streams can be bound to inputs or outputs of the network at each level of the tree in configurations that depend on the level of the tree:

- Frontends: Outgoing downstreams may be bound to one or more outputs of the local component network. And incoming upstreams may be bound to one or more inputs of the network. If a given upstream isn’t bound, the data received on that upstream is simply discarded.

- Filters: Both incoming and outgoing, downstreams and upstreams, may be bound to one or more inputs or outputs (as appropriate) of the local component network. Data arriving on a stream whose incoming side is not bound is simply passed thru to the next MRNet level unmolested. Data arriving on a stream whose incoming side is bound is redirect as appropriate and not passed thru to the next MRNet level unless the network re-emits an output onto that same stream. Note that MRNet places filters on the frontend and backend nodes of the tree.

- Backends: Outgoing upstreams may be bound to one or more outputs of the local component network. And incoming downstreams may be bound to one or more inputs of the network. If a given downstream isn't bound, the data received on that downstream is simply discarded.

Now that the concepts of a frontend, filter, backend, and stream have been introduced, the structure of the XML distributed (via MRNet) component network description can be broken down into the following sections:

- Type & Version: One point not well illustrated by figure 5 is that all CBTF distributed component networks are also themselves abstracted as a single component with inputs and outputs. Thus as with non-distributed networks, we must specify the distributed network's type and version. The `<Type>` and `<Version>` tags are used to provide this information just as with a non-distributed network.

- Inputs & Outputs: Ditto. Just as with a non-distributed component network, the `<Input>` and `<Output>` tags are used for this purpose. Note that the inputs and outputs of the network can only be connected to component instances in the frontend's local component network.

- Streams: Each stream must be declared via a `<Stream>` tag. The (unique) name of the stream must always be provided. Optionally, the MRNet message tag associated with this stream may also be specified. A unique tag is chosen automatically for each stream when a specific value isn't given. Don't specify a tag explicitly unless you have a specific reason for doing so.

- Frontend, Filter, and Backend: The `<Frontend>`, `<Filter>`, and `<Backend>` tags are used to specify the local component network at each level of the MRNet tree, and the bindings of streams to inputs and/or outputs of that network. The network is specified in the `<Network>` subtag, and its syntax is identical to the non-distributed component network syntax discussed in the libcbtf-xml section. Bindings are created via that `<IncomingDownstream>`, `<OutgoingDownstream>`, `<IncomingUpstream>`, and `<OutgoingUpstream>` tags. In addition, the `<Filter>` tag contains a special `<Depth>` subtag that is used to specify at which level of the tree the filter should be placed. Unlike `<Frontend>` and `<Backend>`, which may only be specified once, multiple `<Filter>` tags may be used to place different networks at different levels of the tree. Depths may currently be one of:

  - `<AllOther>`: This network is placed at all filter levels for which there isn't another filter specified with a `<LeafRelative>` or `<RootRelative>` tag.

  - `<LeafRelative>`: This network is placed at the specified offset, relative to the leaves of the tree. In figure 5, a network could be restricted to only the filter on the backend nodes by using `<Depth><LeafRelative offset="0"/></Depth>`.

- `<RootRelative>`: This network is placed at the specified offset, relative to the root of the tree. In figure 5, a network could be restricted to only the filter on the frontend node by using `<Depth><RootRelative offset="0"/></Depth>`.

The same `registerXML` API function provided by libcbtf-xml is used to register these networks. A tool need only link against the libcbtf-mrnet library in order to add support for these networks.

Tutorials

The following tutorials present the construction of a tool that utilizes a CBTF distributed component network to execute an arbitrary command (such as “top” or “ps”) on multiple nodes. Output from the executed command is gathered up and displayed by the tool.

Creating Components

Multiple components are used by this tool. All of those components will be contained within a single source file that begins by including header files and using namespaces.

```
#include <boost/bind.hpp>
#include <mrnet/MRNet.h>
#include <stdio.h>
#include <string>
#include <sys/param.h>
#include <typeinfo>

#include <KrellInstitute/CBTF/Component.hpp>
#include <KrellInstitute/CBTF/Type.hpp>
#include <KrellInstitute/CBTF/Version.hpp>

using namespace KrellInstitute::CBTF;
using namespace std;
```

Most of the above includes are for standard Linux, C++, and Boost header files. MRNet is referenced as well. Note that all of CBTF’s header files are found in the above-referenced directory.

The class for the tool’s main component - the one that executes the command - is defined next.

```
class __attribute__ ((visibility ("hidden"))) ExecuteCommand :
    public Component
{
```

As already noted previously, all CBTF components must inherit from the `Component` class. They should also be marked with hidden visibility. Hiding the class declaration causes its symbols to not be exported by the linker. This is necessary in order to allow multiple definitions of the same class, but with different versions, to peacefully co-exist in CBTF.

All CBTF components must have a public factory function with the name `factoryFunction`. This is the means by which CBTF dynamically discovers the components type, version, etc. and by which it creates instances of the component.

```
public:

    static Component::Instance factoryFunction()
    {
        return Component::Instance(
            reinterpret_cast<Component*>(new ExecuteCommand())
            );
    }
```

All CBTF components should also have a private default constructor. This constructor defines the type of the component, the version of the component, and the input/output names and their types. A binding of inputs to the function that will handle those inputs is also specified at this time.

```
private:

    ExecuteCommand() :
        Component(Type(typeid(ExecuteCommand)), Version(0, 0, 1))
    {
        declareInput<string>(
            "command",
            boost::bind(&ExecuteCommand::commandHandler, this, _1)
            );
        declareOutput<vector<string> >("output");
    }
```

A handler for the `command` input must be defined. It is invoked every time someone provides a value for that input. In this case, the requested command is immediately executed, its output is gathered, and emitted on the component's output.

Note that components don't have to follow this precise model. Inputs can be queued or even ignored. Components may also emit outputs in a completely asynchronous manner. For example, the default constructor might create a thread which monitors, say, /proc, and emits an output every time a particular process' CPU usage exceeds 50%.

```
    void commandHandler(const string& command)
    {
        vector<string> output;

        char hostname[MAXHOSTNAMELEN];
        gethostname(hostname, MAXHOSTNAMELEN);

        output.push_back(string(
            "Output of \"" + command + "\" from host \"" +
            hostname + "\"."
```

```
            ));

        FILE *fd = popen(command.c_str(), "r");

        if (fd != NULL)
        {
            char buffer[BUFSIZ];
            memset(&buffer, 0, sizeof(buffer));

            while (fgets(buffer, sizeof(buffer), fd))
            {
                string line(buffer);

                if (!line.empty())
                {
                    if (line[line.length() - 1] == '\n')
                    {
                        line.erase(line.length() - 1);
                    }
                    output.push_back(line);
                }
            }
            pclose(fd);
        }
        else
        {
            output.push_back(string("Error executing this command!"));
        }

        emitOutput<vector<string> >("output", output);
    }
```

Finally, to complete this component, its class is closed and registered with CBTF. A macro provided by CBTF, which generates a statically initialized C++ structure, is used to perform the registration.

```
}; // class ExecuteCommand

KRELL_INSTITUTE_CBTF_REGISTER_FACTORY_FUNCTION(ExecuteCommand)
```

The remainder of the components used by this tool are conversion components that translate several datatypes to and from a MRNet packet. These are used in order to bind various component inputs and outputs to the upward and downward streams that MRNet provides. The implementation of these components are very similar in nature to the above component, so the complete definition of only one is shown below.

```
class __attribute__ ((visibility ("hidden"))) ConvertStringToPacket :
    public Component
{
```

```
public:

    static Component::Instance factoryFunction()
    {
        return Component::Instance(
            reinterpret_cast<Component*>(new ConvertStringToPacket())
            );
    }

private:

    ConvertStringToPacket() :
        Component(Type(typeid(ConvertStringToPacket)), Version(0, 0, 1))
    {
        declareInput<string>(
            "in", boost::bind(&ConvertStringToPacket::inHandler, this, _1)
            );
        declareOutput<MRN::PacketPtr>("out");
    }

    void inHandler(const string& in)
    {
        emitOutput<MRN::PacketPtr>(
            "out", MRN::PacketPtr(new MRN::Packet(0, 0, "%s", in.c_str()))
            );
    }

}; // class ConvertStringToPacket

KRELL_INSTITUTE_CBTF_REGISTER_FACTORY_FUNCTION(ConvertStringToPacket)

class __attribute__ ((visibility ("hidden"))) ConvertStringListToPacket :
    public Component
{
    ...
}; // class ConvertStringListToPacket

KRELL_INSTITUTE_CBTF_REGISTER_FACTORY_FUNCTION(ConvertStringListToPacket)

class __attribute__ ((visibility ("hidden"))) ConvertPacketToString :
    public Component
{
    ...
}; // class ConvertPacketToString

KRELL_INSTITUTE_CBTF_REGISTER_FACTORY_FUNCTION(ConvertPacketToString)

class __attribute__ ((visibility ("hidden"))) ConvertPacketToStringList :
    public Component
```

```
{
    ...
}; // class ConvertPacketToStringList

KRELL_INSTITUTE_CBTF_REGISTER_FACTORY_FUNCTION(ConvertPacketToStringList)
```

All that remains to complete the plugin for this tool is to compile the above source file and link it against libcbtf into a shared library.

Writing XML

A distributed component network must now be defined which combines the above components into a useful configuration. As described above, XML is the language used for this purpose. The XML begins with the `<MRNet>` root element:

```
<?xml version="1.0" encoding="utf-8"?>
<MRNet xmlns="http://www.krellinst.org/CBTF/MRNet.xsd">
```

Every CBTF distributed component network is encapsulated as if it were a single component with inputs and outputs. I.e. component networks - distributed or not - are themselves components. Thus we must specify the CBTF type, version, inputs, and outputs of this distributed component. The inputs and outputs of the component must be connected to inputs and outputs of the frontend component network that will be defined shortly.

```
  <Type>TutorialTool</Type>
  <Version>1.0.0</Version>

  <Input>
    <Name>command</Name>
    <To><Input>command_to_frontend</Input></To>
  </Input>

  <Output>
    <Name>output</Name>
    <From><Output>output_from_frontend</Output></From>
  </Output>
```

The next section specifies the component network to be placed on the frontend of the MRNet tree. A single non-distributed, local, component network is located on the frontend. Like every other component network, it is itself a component and thus must specify its CBTF type, version, inputs, and outputs.

```
  <Frontend>
    <Network>
      <Type>TutorialTool_Frontend</Type>
      <Version>1.0.0</Version>
```

All component networks must specify the plugins which contain the components they use. Note that a plugin in this case may refer to a shared library containing components like the one described in the previous section. But it may also refer to an XML file that defines one or more component networks which, once again, are themselves components. Search paths can be specified in order to avoid repeatedly specifying a full path for the plugin. In the following, the plugin defined in the previous section is assumed to be named `TutorialToolPlugin`, and resides in the `/opt/TutorialTool` directory.

```
<SearchPath>/opt/TutorialTool</SearchPath>

<Plugin>TutorialToolPlugin</Plugin>

<Component>
  <Name>convert_command</Name>
  <Type>ConvertStringToPacket</Type>
</Component>

<Component>
  <Name>convert_output</Name>
  <Type>ConvertPacketToStringList</Type>
</Component>

<Input>
  <Name>command_to_frontend</Name>
   <To>
     <Name>convert_command</Name>
     <Input>in</Input>
   </To>
</Input>

<Input>
  <Name>output_from_network</Name>
  <To>
    <Name>convert_output</Name>
    <Input>in</Input>
  </To>
</Input>

<Output>
  <Name>command_to_network</Name>
  <From>
    <Name>convert_command</Name>
    <Output>out</Output>
  </From>
</Output>

<Output>
  <Name>output_from_frontend</Name>
  <From>
```

```
          <Name>convert_output</Name>
          <Output>out</Output>
        </From>
      </Output>

    </Network>
```

For CBTF's purposes, MRNet's communication mechanism is abstracted as zero or more symbolically-named upward or downward data streams of a single type. The frontend has incoming upward streams and outgoing downward streams. The backends have incoming downward streams and outgoing upward streams. Filters have all four stream types.

A particular input or output from a non-distributed, local, component network (such as the one defined above for the frontend) can be bound to an appropriate incoming/outgoing upward/downward stream using the notation shown below. In this example, there is a single incoming upward stream named `output_stream` bound to the `output_from_network` input of the above network, as well as a single outgoing downward stream to carry the command to all the backends.

If a given named stream is not bound at a particular filter level of the MRNet tree, messages on it simply continue upward or downward in their original direction of travel. I.e. in this example, since there are no filters defined below, values sent downward on `command_stream` by the frontend will simply continue along their way until they reach the backends.

If a given upward named stream is not bound within the frontend, values received on that stream are simply discarded. By the same token, if a given downward named stream is not bound within the backends, values received on that stream are simply discarded as well.

The component network input or output to which a stream is bound must (currently) always be of the `MRN::PacketPtr` type. Hence the plugin described above contained numerous conversion components that can translate data to/from `MRN::PacketPtr`. And this XML file contains numerous instantiations of these components.

```
    <IncomingUpstream>
      <Name>output_stream</Name>
      <To><Input>output_from_network</Input></To>
    </IncomingUpstream>

    <OutgoingDownstream>
      <Name>command_stream</Name>
      <From><Output>command_to_network</Output></From>
    </OutgoingDownstream>

  </Frontend>
```

This tutorial does not place component networks within the filters running on the MRNet frontend, communication processes, or backends. But if it did, this is where they would be specified. Instead, the following section specifies the component network to be placed on every backend (daemon) in the MRNet tree.

```
  <Backend>
```

Just as with the frontend, a complete non-distributed component network must be specified for execution on the backends. Note that if lightweight MRNet is being used with the MRNet backend attach mode, this section should be omitted because the lightweight MRNet backend - being C-only code - will not be capable of instantiating a CBTF component network.

```
    <Network>

      <Type>TutorialTool_Backend</Type>
      <Version>1.0.0</Version>

      <SearchPath>/opt/TutorialTool</SearchPath>

      <Plugin>TutorialToolPlugin</Plugin>

      <Component>
        <Name>convert_command</Name>
        <Type>ConvertPacketToString</Type>
      </Component>

      <Component>
        <Name>execute_command</Name>
        <Type>ExecuteCommand</Type>
      </Component>

      <Component>
        <Name>convert_output</Name>
        <Type>ConvertStringListToPacket</Type>
      </Component>

      <Input>
        <Name>command_from_network</Name>
        <To>
          <Name>convert_command</Name>
          <Input>in</Input>
        </To>
      </Input>

      <Connection>
        <From>
          <Name>convert_command</Name>
          <Output>out</Output>
        </From>
        <To>
          <Name>execute_command</Name>
          <Input>command</Input>
        </To>
```

```
      </Connection>

      <Connection>
        <From>
          <Name>execute_command</Name>
          <Output>output</Output>
        </From>
        <To>
          <Name>convert_output</Name>
          <Input>in</Input>
        </To>
      </Connection>

      <Output>
        <Name>output_to_network</Name>
        <From>
          <Name>convert_output</Name>
          <Output>out</Output>
        </From>
      </Output>

    </Network>

    <IncomingDownstream>
      <Name>command_stream</Name>
      <To><Input>command_from_network</Input></To>
    </IncomingDownstream>

    <OutgoingUpstream>
      <Name>output_stream</Name>
      <From><Output>output_to_network</Output></From>
    </OutgoingUpstream>

  </Backend>
</MRNet>
```

This XML is placed into a file named `TutorialTool.xml` which is used in the next tutorial.

Creating Tools

Now that all the necessary components have been created, and a XML description of the tool's distributed component network has been written, it is time to construct the tool itself. The tool, as with the component plugin, is contained within a single source file that begins by including header files and using namespaces.

```
#include <iostream>
#include <boost/program_options.hpp>
```

```
#include <boost/shared_ptr.hpp>
#include <boost/thread.hpp>
#include <KrellInstitute/CBTF/BoostExts.hpp>
#include <KrellInstitute/CBTF/Component.hpp>
#include <KrellInstitute/CBTF/Type.hpp>
#include <KrellInstitute/CBTF/ValueSink.hpp>
#include <KrellInstitute/CBTF/ValueSource.hpp>
#include <KrellInstitute/CBTF/XML.hpp>
#include <string>

using namespace boost;
using namespace KrellInstitute::CBTF;
using namespace std;
```

Most of the tool's implementation is wrapped in a class that uses the Boost.Thread library to execute the tool within a separate thread. The following is boilerplate for this class that is not specific to CBTF.

```
class TutorialTool
{

public:

    TutorialTool()
    {
    }

    void start(const string& topology,
               const string& cmd,
               const unsigned int& numBE)
    {
        dm_thread = thread(
            &TutorialTool::run, this, topology, cmd, numBE
            );
    }

    void join()
    {
        dm_thread.join();
    }

    void run(const string& topology,
             const string& cmd,
             const unsigned int& numBE)
    {
```

In order to make use of the CBTF distributed component network `TutorialTool` defined in `TutorialTool.xml`, that file must first be registered with CBTF.

```
        registerXML("TutorialTool.xml");
```

The plugin `BasicMRNetLaunchers` contains, as one might expect, two MRNet launchers - one that uses MRNet's backend create mode, and one that uses MRNet's backend attach mode. Since the distributed component network `TutorialTool` should be usable regardless of the type of launcher employed, `TutorialTool` does not specify a launcher. Instead, one will be created explicitly shortly. This means, however, that `TutorialTool` doesn't list `BasicMRNetLaunchers` in its list of plugins. Thus it must be registered manually here.

```
Component::registerPlugin("BasicMRNetLaunchers");
```

Create an instance of the `TutorialTool` distributed component network. This constructs the component network that resides on the frontend, but does not yet attempt to construct the filter and backend component networks. This full network is not instantiated until a MRNet `Network` object is received on the `Network` input. This inputs is not defined in `TutorialTool.xml` - it is automatically added by CBTF to all MRNet-based CBTF distributed component networks.

```
Component::Instance network = Component::instantiate(
    Type("TutorialTool")
    );
```

Now create an instance of the basic MRNet launcher that uses the backend create mode. This component takes a MRNet topology file as input and has a `Network` output that is of the MRNet `Network` type.

```
Component::Instance launcher = Component::instantiate(
    Type("BasicMRNetLauncherUsingBackendCreate")
    );
```

Instruct CBTF to connect the `Network` output of the launcher component to the `Network` input of the `TutorialTool` instance.

```
Component::connect(launcher, "Network", network, "Network");
```

In order to pass values between regular C++ code and a CBTF component network, bridge objects are used. The CBTF `ValueSource` and `ValueSink` template classes function as these bridges. In this example there are two inputs to, and one output from, the `TutorialTool` + launcher combination:

`topology_file`: The name of the topology file describing the MRNet network to be constructed. This will be passed into the launcher.

`command`: The command to be executed. This is passed into the `TutorialTool` component.

`outputs`: The outputs of the command. This is generated by the `TutorialTool` component.

Note that unlike most other CBTF components, the `ValueSource` and `ValueSink` components are explicitly instantiated via their `instantiate` method. In order for CBTF to connect these bridge components into the network, they must be cast to a `Component::Instance`.

```
shared_ptr<ValueSource<filesystem::path> > topology_file =
    ValueSource<filesystem::path>::instantiate();

shared_ptr<ValueSource<string> > command =
    ValueSource<string>::instantiate();

shared_ptr<ValueSink<vector<string> > > outputs =
    ValueSink<vector<string> >::instantiate();

Component::Instance topology_file_component =
    reinterpret_pointer_cast<Component>(topology_file);

Component::Instance command_component =
    reinterpret_pointer_cast<Component>(command);

Component::Instance outputs_component =
    reinterpret_pointer_cast<Component>(outputs);
```

Connect the bridges to the appropriate inputs and output of the launcher and `TutorialTool` components.

```
Component::connect(
    topology_file_component, "value", launcher, "TopologyFile"
    );

Component::connect(
    command_component, "value", network, "command"
    );

Component::connect(
    network, "output", outputs_component, "value"
    );
```

Now that all the components have been connected, it is time to get things rolling. The `BasicMRNetLauncherUsingBackendCreate` launcher needs a single input - the path of the MRNet topology file. Once this value is passed in, the launcher calls MRNet to construct the MRNet network and then emits the `Network` object on its output, where it travels to the TutorialTool component and causes the complete CBTF distributed component network to be created.

```
*topology_file = topology;
```

Send a command down the MRNet network to the backends for execution.

```
*command = cmd;
```

Loop until command outputs have been received from all backends.

```
for (int num_received = 0; num_received < numBE; ++num_received)
```

```
        {
```

The following line, which calls `ValueSink<...>::operator()`, will block until a value is available on the component output to which it is attached.

```
            vector<string> output = *outputs;
```

Display the received command output on the standard output stream and finish this method and the class.

```
            for (vector<string>::const_iterator
                     i = output.begin(); i != output.end(); ++i)
            {
                cout << *i << endl;
            }
        }
    } // run

private:
     thread dm_thread;

}; // class TutorialTool
```

The main function for the tutorial tool can now be written.

```
int main(int argc, char** argv)
{
```

Specify the default topology file location and command that will be used if they aren't specified in the command-line arguments.

```
    char const* home = getenv("HOME");
    string default_topology(home);
    default_topology += "/.cbtf/cbtf_topology";
    string default_cmd = "ps -ef";
```

Parse the command-line arguments using the Boost.Program_options library.

```
    unsigned int numBE;
    string topology;
    string cmd;

    program_options::options_description desc("daemonToolDemo options");

    desc.add_options()
        ("help,h", "Produce this help message.")

        ("numBE",
         program_options::value<unsigned int>(&numBE)->default_value(1),
```

```
         "Number of expected mrnet backends. Default is 1, This should "
         "match the number of nodes in your topology and job allocation.")

        ("topology",
         program_options::value<string>(&topology)->
         default_value(default_topology),
         "Path name to a valid mrnet topology file. (i.e. from
mrnet_topgen),")

        ("cmd",
         program_options::value<string>(&cmd)->default_value(default_cmd),
         "Command to execute on backend tool daemon. Must be in quotes "
         "if command has arguments.")
        ;

    program_options::variables_map vm;

    program_options::store(
        program_options::parse_command_line(argc, argv, desc), vm
        );
    program_options::notify(vm);

    program_options::positional_options_description p;
    program_options::store(
        program_options::command_line_parser(argc, argv).
        options(desc).positional(p).run(), vm
        );
    program_options::notify(vm);

    if (vm.count("help"))
    {
        cout << desc << endl;
        return 1;
    }

    cout << "Running command " << cmd
         << "\nNumber of mrnet backends: "  << numBE
         << endl;
```

Construct an instance of TutorialTool, initiate a separate thread to run the command, and then wait for that thread to complete.

```
    TutorialTool tutorial_tool;
    tutorial_tool.start(topology, cmd, numBE);
    tutorial_tool.join();
}
```

To complete the tool, this source file is compiled and linked against libcbtf, libcbtf-xml, and libcbtf-mrnet. It does not need to be linked against `TutorialToolPlugin` since that plugin is loaded dynamically by the above code.