CPN Library Tutorial

John Bridgman

May 27, 2011

1 Introduction

This tutorial provides an introduction to using the Computational Process Network (CPN) library. For most of this document, the same example Fibonacci number generator will be used. This way you may see several different ways of implementing the same algorithm in CPN. The Fibonacci sequence can be described as the following recursive equation and initial values.

$$F_n = F_{n-1} + F_{n-2} \tag{1}$$
$$F_0 = 1 \tag{2}$$
$$F_1 = 1 \tag{3}$$

This Fibonacci number generator can be represented by a very simple process network (PN), and requires only two types of nodes. The first type of node is a summing node that takes two input queues and one output queue. It repeatedly takes one token from each input queue, sums them together and then puts the result on the output queue. The second type of node is a *cons* node. The cons node has an initial value, an input queue, and two (identical) output queues. It first sends its initial value to the output queues, then repeatedly copies individual tokens from the input queue to the output queues. The topology of this Fibonacci example is in Figure 1. Note that this example uses 64-bit unsigned integer arithmetic, and the largest Fibonacci number that can be represented is about $1.22 * 10^{19}$. Requesting a maximum number larger than this will result in overflow and incorrect results.

2 CPN Kernel

The primary class used to construct a CPN program is the `CPN::Kernel`. The kernel is responsible for creating nodes and queues, maintaining the state of the network, and cleaning up resources. The kernel creates nodes from a node type name by using a node loader (detailed in section 3.3). The kernel provides two ways to create a new node. It also provides an interface for accessing queue endpoints from outside the network via the *external endpoint* interface. (The Fibonacci examples use the external endpoint interface to gather the PN output.) The kernel contains a *Context* (section 2.2) to store and query the configuration of the CPN program, and to talk to other kernels.

2.1 Construction

The kernel takes a `KernelAttr` parameter object in the constructor. The only required parameter is the name, which is taken in the `KernelAttr` constructor. The optional setter methods are:

SetName The name for the kernel.

SetHostName The hostname to bind to for making remote queue connections.

SetServName The service name or port number to bind to for making remote queue connections.

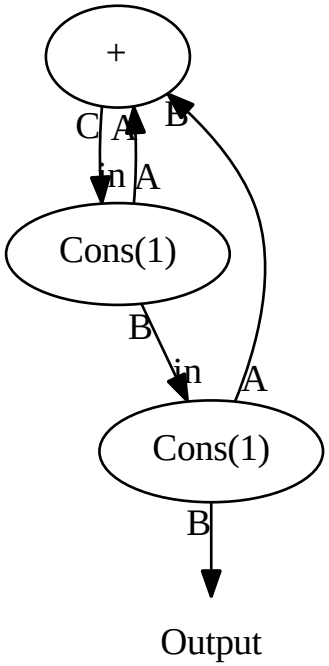

Figure 1: Process network to generate Fibonacci numbers

UseD4R Option to turn D4R off or on. Default is on.

SetContext The context for execution. If not given a local context is generated.

SetRemoteEnabled Force enable remote operations even when the context is not remote enabled and no host name or service name given.

SwallowBrokenQueueExceptions Whether to handle broken queue exceptions that propagate out of the node. By default the broken queue exceptions are left unhandled and result in a call to `std::terminate` (and application termination).

GrowQueueMaxThreshold Whether to grow the queue max threshold if a request for more than the max threshold is made.

AddSharedLib Add a shared library to the list of libraries to preload in the node loader for node lookup.

AddNodeList Add a node list to the node loader to parse for node lookup.

2.2 Context

The context provides the mechanism through which the kernel maintains state about the CPN program. The context also provides shared state and communication between kernels so that the program can be distributed across multiple computers. Normally, a user need only access a context directly when a CPN program is distributed. The CPN library provides two implementations of a context. The first implementation is `LocalContext` which provides the context interface to one or more kernels in the same process. The second implementation is `RemoteContext` which (when combined with the remote context server) gives a context object that provides state across multiple processes. The `RemoteContext` and server is built on top of two abstract base classes which provide everything except the transportation medium.

3 Nodes

There are two different ways to define or create a node. The first way is to create a node class which is derived from `CPN::NodeBase`. This method is more verbose but allows one to load the node through the `VariantCPNLoader` (section 3.3). The second way is by asking the kernel to create a function node, where a node is created and calls a function with a particular signature.

3.1 Derived Node Example

In order to define a new derived node type, one must create a class that inherits from `CPN::NodeBase`. The declaration for the summing node is in Listing 1. The full source code for this example is in section 6.1.

Listing 1: Declaration for a derived summing node

```
class Summer : public NodeBase {
public:
    Summer(Kernel &ker, const NodeAttr &attr)
    : NodeBase(ker, attr) {}
private:
    void Process();
};
```

When a node starts, the `Process()` method is called in a new, separate thread. The code for the summing node's `Process()` method is in Listing 2.

Listing 2: Summing node process declaration

```
void Summer::Process() {
    IQueue<uint64_t> in_a = GetIQueue("A");
    IQueue<uint64_t> in_b = GetIQueue("B");
    OQueue<uint64_t> out = GetOQueue("C");
    while (true) {
        uint64_t val_a, val_b, sum;
        if (!in_a.Dequeue(&val_a, 1)) break;
        if (!in_b.Dequeue(&val_b, 1)) break;
        sum = val_a + val_b;
        out.Enqueue(&sum, 1);
    }
}
```

First, the summer node gets all its input endpoints and places them in `IQueue`s. Next, the node gets the output port and places it in an `OQueue`. Then it repeatedly reads from all its inputs, sums them and enqueues the result. Note that the `IQueue<T>::Dequeue(T *, unsigned)` function will return `false` if the other end of the queue disconnects.

Note that the node does not officially exist until the constructor returns, and a node must exist before a queue can be created for it. Therefore, `GetIQueue` or `GetOQueue` cannot be called in the constructor.

Also note that the raw underlying queue is typeless and simply does a blind bit-wise copy of the data. This means that only plain old data (POD) types can be sent over the queues in CPN. The `CPN::IQueue<T>` and `CPN::OQueue<T>` provide type checking and type coercion for the raw queue interface. Each queue has an associated type string for data that it carries. `IQueue` and `OQueue` check this string and throw an exception on assignment if the types do not match.

If a node returns from its `Process()` method, all queue endpoints will be disconnected. An endpoint can also be disconnected by calling `Release` on the queue endpoint. Trying to enqueue to a disconnected queue will result in a `CPN::BrokenQueueException` being thrown which will result in a call to `std::terminate` if not caught (unless SwallowBrokenQueueExceptions is set as a kernel parameter).

To instantiate a derived node, one must create a `CPN::NodeAttr` that specifies the desired node name and node type name, and then pass that `NodeAttr` object to the kernel method `CreateNode`. A `NodeAttr`

object can also contain node parameters as a string-to-string map. The method `NodeAttr::SetParam` can be used to set node parameters, parameters can be read with the `NodeBase::GetParam` family of methods. For a distributed CPN program (see section 4), `NodeAttr::SetKernel` is used to tell on which kernel the node should be created. A node loader and a node factory for each node type is also required, and will be discussed later in this section.

Full code for the main function to run this Fibonacci example is in Listing 5. First we create a `CPN::Kernel` on the stack, which will be used to build the CPN program. The Kernel constructor takes a `CPN::KernelAttr` object with some parameters set (as described in section 2.1)

```
Kernel kernel(KernelAttr("kernel")
    .UseD4R(false)
    .SwallowBrokenQueueExceptions(true));
```

Next, the nodes need to be created. It is important to notice that both (endpoint) nodes for a queue must exist before the queue can be created. Attribute objects can be reused as their contents are copied in the function. Also note that an external reader is created (and will be discussed later in this section).

```
kernel.CreateNode("summer", "Summer");
NodeAttr nattr("Cons␣1", "Cons");
nattr.SetParam("initial", 1).SetParam("num␣outputs", 2);
kernel.CreateNode(nattr);
nattr.SetName("Cons␣2");
kernel.CreateNode(nattr);
kernel.CreateExternalReader("result");
```

Then the queues can be specified (with a `CPN::QueueAttr` object) and created by the kernel. Again, the same `QueueAttr` is used several times. The final queue connects not to a node, but to the external reader.

```
QueueAttr qattr(2*sizeof(uint64_t), sizeof(uint64_t));
qattr.SetDatatype<uint64_t>();
qattr.SetWriter("Cons␣1", "out0").SetReader("summer", "A");
kernel.CreateQueue(qattr);
qattr.SetWriter("Cons␣2", "out0").SetReader("summer", "B");
kernel.CreateQueue(qattr);
qattr.SetWriter("summer", "C").SetReader("Cons␣1", "in");
kernel.CreateQueue(qattr);
qattr.SetWriter("Cons␣1", "out1").SetReader("Cons␣2", "in");
kernel.CreateQueue(qattr);
qattr.SetWriter("Cons␣2", "out1").SetExternalReader("result");
kernel.CreateQueue(qattr);
```

An external reader can read from a queue in a CPN program despite not being a node in the program. In this case, it is used to gather the results of the Fibonacci number generator. The remainder of the main program is shown below. It reads from the result queue until the specified maximum is reached. It then releases the result queue, which causes the node at the other end to shut down. The flag to swallow broken queue exceptions is set so that the program does not immediately terminate. Instead, the disconnecting of queues cascades through the network until eventually all of the nodes exit. Notice that we must explicitly destroy the external reader.

```
IQueue<uint64_t> result = kernel.GetExternalIQueue("result");
uint64_t value;
do {
    result.Dequeue(&value, 1);
    std::cout << "-␣" << value << std::endl;
} while (value < max_fib);
result.Release();
kernel.DestroyExternalEndpoint("result");
kernel.WaitForAllNodes();
```

This gathering of results gives a simple example of how to use the external endpoints. The external endpoints were added for the explicit reason of having outside code that needed to act as a source or sink into the PN. For example if you were using CPN as processing glue between two independent subsystems that both used callbacks, You could supply external endpoints to each callback and the two systems could then work as sources and sinks in the PN.

For the kernel to dynamically instantiate a node from a node type name (e.g. instantiate a Summer from the string "Summer"), there must be a factory for that node type. The kernel's node loader (section 3.3) uses a factory for the node to find the class and construct an instance.

Because creating a node factory class is boilerplate code, the CPN library provides a simple macro: `CPN_DECLARE_NODE_FACTORY(class, typename)`. This macro defines a node factory and then the special symbol that the node loader uses to look up the node type at run time. Note that the type name must be a valid symbol (limited to alphanumeric and underscore), and is usually the same as the node class name. Listing 3 provides an example of its use.

For node types that require no additional members or methods (true for both Summer and Cons), there is an additional macro: `CPN_DECLARE_NODE_AND_FACTORY(class, typename)` which declares both the node and the factory. The Cons node uses this macro in Listing 4. (The Summer could have used this macro, but did not for illustrative purposes.) In this case, only the one method with the `void Process()` needs to be written by the user.

3.2 Function Node Example

The second type of CPN node is a function node. Function nodes cannot be created by the node loader, but are somewhat simpler. They must be created manually with a call to the kernel, and can make a simple CPN program easier to write.

To create the Fibonacci example, two node types are needed. This time we will use functions for the nodes. The first argument of the functions must be a pointer to a `CPN::NodeBase`. The Summer and Cons functions are defined as:

```
static void Summer(NodeBase *node) {
    IQueue<uint64_t> in_a = node->GetIQueue("A");
    IQueue<uint64_t> in_b = node->GetIQueue("B");
    OQueue<uint64_t> out = node->GetOQueue("C");
    while (true) {
        uint64_t val_a, val_b, sum;
        if (!in_a.Dequeue(&val_a, 1)) break;
        if (!in_b.Dequeue(&val_b, 1)) break;
        sum = val_a + val_b;
        out.Enqueue(&sum, 1);
    }
}

static void Cons(NodeBase *node, uint64_t initial) {
    IQueue<uint64_t> in = node->GetIQueue("in");
    OQueue<uint64_t> out_a = node->GetOQueue("out0");
    OQueue<uint64_t> out_b = node->GetOQueue("out1");
    uint64_t current = initial;
    while (true) {
        out_a.Enqueue(&current, 1);
        out_b.Enqueue(&current, 1);
        if (!in.Dequeue(&current, 1)) break;
    }
}
```

To create CPN nodes from the functions, we use the kernel:

```
kernel.CreateFunctionNode("summer", Summer);
kernel.CreateFunctionNode("Cons 1", Cons, 1);
kernel.CreateFunctionNode("Cons 2", Cons, 1);
kernel.CreateExternalReader("result");
```

The remainder of the main program is the same as in the derived node example. The full example is in Listing 6.

3.3 Node Loader

When you ask the kernel to create a node by type name, it asks the node loader for a node factory that can create a node of that type. If the node loader has a factory for that type it will return it. You can explicitly add a factory for a type name to the node loader using `RegisterNodeFactory`. If no factory is found then there are several places the node loader will look to see if it can find one. The first thing the node loader will do is search the executable for a symbol which is the type name added to an identifier. If a symbol with that name is found it is treated as a function that will return a factory that can create nodes of that type. If such a symbol is not found then the node loader will look if there is any node list that have information about that type. If so, it will look through it to see if there is a shared library that can be loaded to get a function that can supply a factory for that node type. If all of these fail then the node loader throws an exception indicating that no factory could be found for that node type. Each kernel possesses its own node loader.

A node list is a file that has a very simple format. The format is a directive followed by parameters then a semicolon. Newlines, spaces, tabs, etc. are all considered white space and are collapsed unless they are inside of quotations or escaped. The supported directives are "lib" and "include". "lib" has two parameters, the first being the type symbol name, and the second being a shared library object that can be loaded to get that symbol name. The "include" directive treats each parameter as a file name that should be recursively loaded for more definitions. Filenames with spaces and other special characters may be quoted. The format also supports backslashes to escape any special meaning of the next character in unquoted strings and the # character for single line comments. See the file `libraries/CPN/NodeLoader.cc` for details.

One consequence of this loading scheme is that, in general, nodes will not have any referenced symbols in them. This is not normally a problem except when one tries to put a node in a library and use a tool like `jmake`. `jmake` collects objects into archives and then uses the linker to load the archive together. The problem is the linker will prune out object files in archives with no references. This causes objects in which we want to call symbols dynamically to not be linked into the executable. So, when using tools like `jmake`, a manual effort to ensure that there is a referenced symbol in the appropriate object for all the nodes in archives needs to be made. Also, if no nodes are statically linked efforts needs to be made to ensure that the linker does not prune out `typeinfo` symbols that the nodes need in the CPN library. On systems with the GNU linker this can be done by ensuring your tool uses the whole archive flag around the CPN library archives and the node archive files. Even more linkers support the '-u' option which tells them to consider the given symbol name to be undefined and search through and load objects that define that symbol. But using '-u' requires knowledge of the symbols that must be loaded and this is compiler and platform specific.

3.4 VariantCPNLoader

The `VariantCPNLoader` is a class and a set of static methods that can create CPN nodes and queues as described by a Variant object containing data in a specific format.

The Variant class is a simple container that can represent a set of data structures. What Variant can represent is directly mappable into the JavaScript Object Notation (JSON) specification and back again. (More details on the JSON format can be found at `www.json.org`.) The CPN library contains a set of classes and functions for parsing JSON to a Variant and back again. There is also a set of classes and functions for parsing to an XML format and back again. An application that uses Variant can therefore accept both JSON and XML input. The path `apps/XML-JSON` contains an example which reads in JSON to Variant and then outputs in our XML format or reads in or XML format and outputs JSON.

The `VariantCPNLoader` takes a Variant object and uses the contained data to call methods on a kernel object. The `VariantCPNLoader` can also construct a KernelAttr object from the Variant. The `VariantCPNLoader` has a validator that goes through the configuration and ensures that all required fields are present and that there are no obvious errors (e.g. two nodes with the same name). An example of the loader's usage is given in section 4.

The format of the Variant that the loader expects is a key value map at the base. There are nine keys that specify options for the kernel and three that specify how to load the CPN program. The specification for the the `VariantCPNLoader` is in section 7.

The `VariantCPNLoader` cannot create function nodes. Only nodes with a factory can be created. (Queues could still be created `VariantCPNLoader` to connect manually created function nodes.)

The CPNKernel application is an example of an application that will load up any set of derived nodes from a node library using configuration files. It is locate on the path `apps/CPNKernel/main.cc`.

4 Distributing a CPN Program

To distribute a CPN program, you first need to execute a remote context daemon, which is located in `apps/RemoteContext`. Each of the kernels that are created must have a unique name, and must be instructed (with a kernel attribute) to connect to the remote context. When creating nodes in the CPN program, the kernel which should instantiate that node is specified with `NodeAttr::SetKernel`. The approach used in the examples is to have a single program that runs on multiple hosts, and have the the differences between them specified by command line arguments and configuration files.

Now we return to our Fibonacci number generator example, while using the distributed features of the CPN library. The Summer and Cons node are exactly as shown for the derived nodes (section 3.1). The `VariantCPNLoader` is used to load and execute the example program specification. (The listings in this section omit some error checking for clarity, but full error checking is included in Listing 7.)

The configuration files that describe the example are included in Listings 8, 9 and 10. The first file (Listing 8) provides some definitions that are global for the example. The second file (Listing 9) describes the Fibonacci example by listing the nodes and queues, their parameters, and how they interconnect. The third file (Listing 10) describes how the different nodes are mapped onto different kernels (and therefore potentially different hosts) in the distributed system. Because the configuration is split between multiple files, parts of the configuration can be reused and combined in different ways. For example, it may be interesting to map the same CPN program onto the same distributed system a number of different ways, which can be accomplished by changing only the mapping file. Frequently, only a small part of the configuration needs to be different for each kernel. Once the distributed example is built there is a option '-C' which causes the program to load all configuration data then print that data out in JSON format and exit. This can give a good idea of what the loader is doing and you can see how the command line options change what goes into the loader.

The main program of the Fibonacci example initializes some default values and instantiates a VariantCPNLoader.

```
uint64_t max_fib = 100;
bool load_config = true;
bool print_config = false;
VariantCPNLoader loader;
```

The command line options are then parsed (but not shown here). Several command line options are required, such as `--name` that specifies the kernel name. Included in the directory `tutorial/code/distributed` is a shell script `run` that will run this example. It will run the remote context daemon and then run each of the kernels with the correct parameters. The configuration files are also specified on the command line. They are loaded in order, and any duplicate entries override previous entries.

The `loader` creates a `KernelAttr`, which is then used to construct a kernel with the proper parameters. The loader knows to use a remote context because a context host and port are specified.

```
Kernel kernel(loader.GetKernelAttr());
```

Only one of the executed programs will load the configuration and read out the results. It must also create an external reader.

```
if (load_config) {
    kernel.CreateExternalReader("result");
}
loader.Setup(&kernel);
```

Once the configuration is done, the loading kernel will read from the external reader and print out the results until termination is reached. The other kernels simply wait for the network to terminate.

```
if (load_config) {
    QueueAttr qattr(2*sizeof(uint64_t), sizeof(uint64_t));
    qattr.SetWriter("Cons␣2", "out1");
    qattr.SetExternalReader("result");
    kernel.CreateQueue(qattr);

    IQueue<uint64_t> result = kernel.GetExternalIQueue("result");
    uint64_t value;
    do {
        result.Dequeue(&value, 1);
        std::cout << "-␣" << value << std::endl;
    } while (value < max_fib);
    kernel.DestroyExternalEndpoint("result");
    kernel.WaitForAllNodes();
} else {
    kernel.Wait();
}
```

5 Kahn Example

Also included with the code for this tutorial is an example usage of CPN to create the process network that Kahn described in his original paper. This code is located on the path `tutorial/code/kahn`.

6 Appendix A

6.1 Derived Node

Listing 3: The Summing Node

```
#include "NodeBase.h"
#include "IQueue.h"
#include "OQueue.h"
#include <stdint.h>

using namespace CPN;

class Summer : public NodeBase {
public:
    Summer(Kernel &ker, const NodeAttr &attr)
    : NodeBase(ker, attr) {}
private:
    void Process();
```

```
};


void Summer::Process() {
    IQueue<uint64_t> in_a = GetIQueue("A");
    IQueue<uint64_t> in_b = GetIQueue("B");
    OQueue<uint64_t> out = GetOQueue("C");
    while (true) {
        uint64_t val_a, val_b, sum;
        if (!in_a.Dequeue(&val_a, 1)) break;
        if (!in_b.Dequeue(&val_b, 1)) break;
        sum = val_a + val_b;
        out.Enqueue(&sum, 1);
    }
}

CPN_DECLARE_NODE_FACTORY(Summer, Summer);
```

Listing 4: The Cons Node

```
#include "NodeBase.h"
#include "IQueue.h"
#include "OQueue.h"
#include <stdint.h>

using namespace CPN;

CPN_DECLARE_NODE_AND_FACTORY(Cons, Cons);

void Cons::Process() {
    IQueue<uint64_t> in = GetIQueue("in");
    OQueue<uint64_t> out_a = GetOQueue("out0");
    OQueue<uint64_t> out_b = GetOQueue("out1");
    uint64_t current = GetParam<uint64_t>("initial");
    while (true) {
        out_a.Enqueue(&current, 1);
        out_b.Enqueue(&current, 1);
        if (!in.Dequeue(&current, 1)) break;
    }
}
```

Listing 5: Instantiating the declared nodes and running them

```
#include "Kernel.h"
#include "IQueue.h"
#include <iostream>
#include <stdlib.h>
#include <stdint.h>

using namespace CPN;

int main(int argc, char **argv) {
    uint64_t max_fib = 100;
    while (true) {
        int c = getopt(argc, argv, "m:");
        if (c == -1) break;
        switch (c) {
```

```
        case 'm':
            max_fib = strtoull(optarg, 0, 10);
            break;
        default:
            break;
        }
    }

    Kernel kernel(KernelAttr("kernel")
            .UseD4R(false)
            .SwallowBrokenQueueExceptions(true));

    // Create the three nodes use the same parameters for both the cons nodes
    kernel.CreateNode("summer", "Summer");
    NodeAttr nattr("Cons 1", "Cons");
    nattr.SetParam("initial", 1).SetParam("num outputs", 2);
    kernel.CreateNode(nattr);
    nattr.SetName("Cons 2");
    kernel.CreateNode(nattr);
    kernel.CreateExternalReader("result");

    QueueAttr qattr(2*sizeof(uint64_t), sizeof(uint64_t));
    qattr.SetDatatype<uint64_t>();
    qattr.SetWriter("Cons 1", "out0").SetReader("summer", "A");
    kernel.CreateQueue(qattr);
    qattr.SetWriter("Cons 2", "out0").SetReader("summer", "B");
    kernel.CreateQueue(qattr);
    qattr.SetWriter("summer", "C").SetReader("Cons 1", "in");
    kernel.CreateQueue(qattr);
    qattr.SetWriter("Cons 1", "out1").SetReader("Cons 2", "in");
    kernel.CreateQueue(qattr);
    qattr.SetWriter("Cons 2", "out1").SetExternalReader("result");
    kernel.CreateQueue(qattr);

    IQueue<uint64_t> result = kernel.GetExternalIQueue("result");
    uint64_t value;
    do {
        result.Dequeue(&value, 1);
        std::cout << "- " << value << std::endl;
    } while (value < max_fib);
    result.Release();

    kernel.DestroyExternalEndpoint("result");
    kernel.WaitForAllNodes();

    return 0;
}
```

6.2 Function Node

Listing 6: Example of function nodes

```
#include "Kernel.h"
#include "NodeBase.h"
#include "IQueue.h"
#include "OQueue.h"
```

```
#include <stdlib.h>
#include <iostream>

using namespace CPN;
using std::string;

static void Summer(NodeBase *node) {
    IQueue<uint64_t> in_a = node->GetIQueue("A");
    IQueue<uint64_t> in_b = node->GetIQueue("B");
    OQueue<uint64_t> out = node->GetOQueue("C");
    while (true) {
        uint64_t val_a, val_b, sum;
        if (!in_a.Dequeue(&val_a, 1)) break;
        if (!in_b.Dequeue(&val_b, 1)) break;
        sum = val_a + val_b;
        out.Enqueue(&sum, 1);
    }
}

static void Cons(NodeBase *node, uint64_t initial) {
    IQueue<uint64_t> in = node->GetIQueue("in");
    OQueue<uint64_t> out_a = node->GetOQueue("out0");
    OQueue<uint64_t> out_b = node->GetOQueue("out1");
    uint64_t current = initial;
    while (true) {
        out_a.Enqueue(&current, 1);
        out_b.Enqueue(&current, 1);
        if (!in.Dequeue(&current, 1)) break;
    }
}


int main(int argc, char **argv) {
    uint64_t max_fib = 100;
    while(true) {
        int c = getopt(argc, argv, "m:");
        if (c == -1) break;
        switch (c) {
        case 'm':
            max_fib = strtoull(optarg, 0, 10);
            break;
        default:
            break;
        }
    }

    Kernel kernel(KernelAttr("kernel")
        .UseD4R(false)
        .SwallowBrokenQueueExceptions(true));

    kernel.CreateFunctionNode("summer", Summer);
    kernel.CreateFunctionNode("Cons 1", Cons, 1);
    kernel.CreateFunctionNode("Cons 2", Cons, 1);
    kernel.CreateExternalReader("result");

    QueueAttr qattr(2*sizeof(uint64_t), sizeof(uint64_t));
```

```
    qattr.SetDatatype<uint64_t>();
    qattr.SetEndpoints("summer", "A", "Cons 1", "out0");
    kernel.CreateQueue(qattr);
    qattr.SetWriter("Cons 2", "out0").SetReader("summer", "B");
    kernel.CreateQueue(qattr);
    qattr.SetWriter("summer", "C").SetReader("Cons 1", "in");
    kernel.CreateQueue(qattr);
    qattr.SetWriter("Cons 1", "out1").SetReader("Cons 2", "in");
    kernel.CreateQueue(qattr);

    qattr.SetWriter("Cons 2", "out1").SetExternalReader("result");
    kernel.CreateQueue(qattr);

    // This is our result reader.
    {
        IQueue<uint64_t> result = kernel.GetExternalIQueue("result");
        uint64_t value;
        do {
            result.Dequeue(&value, 1);
            std::cout << "- " << value << std::endl;
        } while (value < max_fib);
        result.Release();
    }

    kernel.DestroyExternalEndpoint("result");
    kernel.WaitForAllNodes();
    return 0;
}
```

6.3 Distributed

Listing 7: Example of the main routine for distributing the PN

```
#include "Kernel.h"
#include "IQueue.h"
#include "VariantToJSON.h"
#include "JSONToVariant.h"
#include "XMLToVariant.h"
#include "VariantCPNLoader.h"
#include "ParseBool.h"
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <stdint.h>
#include <getopt.h>

using namespace CPN;

static bool LoadJSONConfig(VariantCPNLoader &loader,
        const std::string &filename) {
    std::ifstream def_file(filename.c_str());
    if (def_file) {
        JSONToVariant parser;
        parser.ParseStream(def_file);
        if (!parser.Done()) {
            std::cerr << "Parse error in " << filename << " at line "
                << parser.GetLine() << " column " << parser.GetColumn()
```

```
                << std::endl;
            return false;
        } else {
            loader.MergeConfig(parser.Get());
        }
    } else {
        std::cerr << "Could not open the process network definition file\n";
        return false;
    }
    return true;
}

static bool LoadXMLConfig(VariantCPNLoader &loader,
        const std::string &filename) {
    std::ifstream def_file(filename.c_str());
    if (def_file) {
        XMLToVariant parser;
        parser.ParseStream(def_file);
        if (!parser.Done()) {
            std::cerr << "Parse error in " << filename << ": "
                << parser.GetMessage() << std::endl;
            return false;
        } else {
            loader.MergeConfig(parser.Get());
        }
    } else {
        std::cerr << "Could not open the process network definition file\n";
        return false;
    }
    return true;
}

static void PrintUsage(const char *progname) {
    using std::cerr;
    cerr << "Usage: " << progname << " [options]\n"
    "\t-m val\n\t--max=val\n\t\tSpecify the value to stop at.\n"
    "\t-c yes|no\n\t--config=yes|no\n\t\tSpecify if the config"
       " file should be loaded.\n"
    "\t-C\n\t\tPrint the internal config and exit.\n"
    "\t--ctx-port port\n\t\tSpecify the port the context daemon is"
       " listening on.\n"
    "\t--ctx-host host\n\t\tSpecify the host the context daemon is on.\n"
    "\t--port port\n\t\tSpecify the port the kernel should listen on.\n"
    "\t--host host\n\t\tSpecify the host the kernel should listen on.\n"
    "\t--name name\n\t\tSpecify the name the kernel should have,"
       " note that this must match the names in the nodemap file.\n"
    ;
}

enum Option_t {
    OP_CTX_PORT = 256,
    OP_CTX_HOST,
    OP_KERNEL_PORT,
    OP_KERNEL_HOST,
    OP_NAME
};
```

```
static const option long_options[] = {
    { "config", 1, 0, 'c'},
    { "max", 1, 0, 'm'},
    { "ctx-port", 1, 0, OP_CTX_PORT},
    { "ctx-host", 1, 0, OP_CTX_HOST},
    { "port", 1, 0, OP_KERNEL_PORT},
    { "host", 1, 0, OP_KERNEL_HOST},
    { "name", 1, 0, OP_NAME},
    { "help", 0, 0, 'h'},
    {0,0,0,0}
};
int main(int argc, char **argv) {
    uint64_t max_fib = 100;
    bool load_config = true;
    bool print_config = false;
    VariantCPNLoader loader;
    while (true) {
        int c = getopt_long(argc, argv, "m:c:Chj:x:", long_options, 0);
        if (c == -1) break;
        switch (c) {
        case 'm':
            max_fib = strtoull(optarg, 0, 10);
            break;
        case 'c':
            load_config = ParseBool(optarg);
            break;
        case 'C':
            print_config = true;
            break;
        case OP_CTX_PORT:
            loader.ContextPort(optarg);
            break;
        case OP_CTX_HOST:
            loader.ContextHost(optarg);
            break;
        case OP_KERNEL_PORT:
            loader.KernelPort(optarg);
            break;
        case OP_KERNEL_HOST:
            loader.KernelHost(optarg);
            break;
        case OP_NAME:
            loader.KernelName(optarg);
            break;
        case 'j':
            if (!LoadJSONConfig(loader, optarg)) {
                return 1;
            }
            break;
        case 'x':
            if (!LoadXMLConfig(loader, optarg)) {
                return 1;
            }
            break;
        case 'h':
```

```
        default:
            PrintUsage(*argv);
            return 1;
        }
    }

    if (print_config) {
        std::cout << PrettyJSON(loader.GetConfig(), true) << std::endl;
        return 0;
    }

    std::pair<bool, std::string> validate_result = loader.Validate();
    if (!validate_result.first) {
        std::cout << "Configuration did not validate:\n\t"
            << validate_result.second << std::endl;
        return 1;
    }

    Kernel kernel(loader.GetKernelAttr());

    if (load_config) {
        kernel.CreateExternalReader("result");
    }
    loader.Setup(&kernel);

    if (load_config) {
        QueueAttr qattr(2*sizeof(uint64_t), sizeof(uint64_t));
        qattr.SetWriter("Cons 2", "out1");
        qattr.SetExternalReader("result");
        kernel.CreateQueue(qattr);

        IQueue<uint64_t> result = kernel.GetExternalIQueue("result");
        uint64_t value;
        do {
            result.Dequeue(&value, 1);
            std::cout << "- " << value << std::endl;
        } while (value < max_fib);
        kernel.DestroyExternalEndpoint("result");
        kernel.WaitForAllNodes();
    } else {
        kernel.Wait();
    }
    return 0;
}
```

6.4 Distributed Configuration Files

Listing 8: Common definitions between kernels

```
{
    "d4r": false,
    "swallow-broken-queue-exceptions": true
}
```

Listing 9: Process Network Definition

```
{
```

```
"nodes": [
    {
        "name": "summer",
        "type": "Summer"
    },
    {
        "name": "Cons␣1",
        "type": "Cons",
        "param": {
            "num␣outputs": 2,
            "initial": 1
        }
    },
    {
        "name": "Cons␣2",
        "type": "Cons",
        "param": {
            "num␣outputs": 2,
            "initial": 1
        }
    }
],

"queues": [
    {
        "size": 16,
        "threshold": 8,
        "type": "default",
        "datatype": "uint64_t",
        "readernode": "summer",
        "readerport": "A",
        "writernode": "Cons␣1",
        "writerport": "out0"
    },
    {
        "size": 16,
        "threshold": 8,
        "type": "default",
        "datatype": "uint64_t",
        "readernode": "summer",
        "readerport": "B",
        "writernode": "Cons␣2",
        "writerport": "out0"
    },
    {
        "size": 16,
        "threshold": 8,
        "type": "default",
        "datatype": "uint64_t",
        "readernode": "Cons␣1",
        "readerport": "in",
        "writernode": "summer",
        "writerport": "C"
    },
    {
        "size": 16,
```

```
            "threshold": 8,
            "type": "default",
            "datatype": "uint64_t",
            "readernode": "Cons␣2",
            "readerport": "in",
            "writernode": "Cons␣1",
            "writerport": "out1"
        }
    ]

}
```

Listing 10: Node mapping

```
{
    "nodemap": {
        "summer": "kernel_1",
        "Cons␣1": "kernel_2",
        "Cons␣2": "kernel_3"
    }
}
```

7 Appendix B

Kernel loader data formats.

The nine options for the kernel are:

"name" The name that the kernel will use (required).

"host" The host name that the kernel will use to listen on (CPN uses POSIX `getaddrinfo` for address lookup and as such supports any host name or port name format that `getaddrinfo` can turn into a `sockaddr` struct) (optional).

"port" The port number or name that the kernel will use to listen on (optional).

"d4r" Whether to use the D4R algorithm or not (optional, default true).

"swallow-broken-queue-exceptions" Whether to swallow the BrokenQueueException that can be emitted from an enqueue (optional, default false).

"grow-queue-max-threshold" Whether to automatically grow the queue if a larger threshold is asked for than the queue can handle (optional, default true)

"libs" A list of shared libraries that contain node definitions that the node loader should load up immediately.

"liblist" A list of node list files that contain information about where to find the shared libraries for nodes that are not statically linked in or manually loaded.

"context" A sub-map that describes options for what kind of context to load and what options on the context to set. Those options are:

- **"host"** What host name the context daemon is listening on (optional, if present "port" must also be specified, else the local context implementation will be used).
- **"port"** What port number or name the context daemon is listening on.
- **"loglevel"** Specify the logging level for the internal CPN logger. This is mainly used for debugging the internals of the CPN library.

The three keys that specify how to load the PN are:

“nodes” A list of node definitions.

“queues” A list of queue definitions.

“nodemap” A key value map with node names as keys and the kernel name which that node should be run on as values.

A node definition is a map with the following fields:

“name” The unique name for the node (required).

“type” The type name of this node (required).

“param” Parameters to pass to this node (optional). If parameters is a Variant then it will be serialized as JSON.

“kernel” The name of the kernel that this node should be loaded on (optional). The nodemap overrides this option.

The queue definition is a map with the following fields:

“size” The size in bytes of this queue (required).

“threshold” The size in bytes of the threshold (required).

“readernode” The name of the node that will read from this queue (required).

“readerport” The port name that the reader node will use (required).

“writernode” The name of the node that will write to this queue (required).

“writerport” The port name that the writer node will use (required).

“type” The type of queue, either “threshold” or “default” (optional, default is “default”).

“datatype” A string that specifies the data type for the queue (optional, default “void”). Some valid types are “void”, and the various types names defined in “stdint.h” like “int32_t” as well as “float”, “double”, and “complex<T>” where “T” is any of the previous types.

“numchannels” The number of channels for this queue (optional, default 1).

“alpha” A parameter used only by the remote queue allowing one to specify how much space to split between the two sides (optional, default 0.5).