Synapse
Reference Guide

tools@bsc.es

October 9, 2015

Contents

Contents **3**

1 Introduction **5**

2 Abstractions **7**

3 A simple example **9**
3.1 Front-End sample code 9
3.2 Back-End sample code 10
3.3 Front-End side of the "ping-pong" protocol sample code 10
3.4 Back-End side of the "ping-pong" protocol sample code 11

4 Synapse API **13**
4.1 Class FrontEnd 13
4.2 Class BackEnd 16
4.3 Class MRNetApp 18
4.4 Class Protocol 19
4.5 Class FrontProtocol 19
4.6 Class BackProtocol 20
4.7 Class PendingConnections 21

5 Synapse wrappers **23**

6 Synapse configuration tool **25**

Bibliography **27**

Chapter 1

Introduction

Synapse is a framework to facilitate the development of MRNet [2] applications. Synapse takes its name from the neurological structure in the nervous system, because following the analogy, MRNet applications resemble a neural network where inputs are passed from one neuron (processor) to the next, and each processor applies a function on the passing data.

As stated in the MRNet documentation [1], "MRNet is a customizable, high-throughput communication software system for parallel tools and applications with a master/slave architecture. MRNet reduces the cost of these tools' activities by incorporating a tree-based overlay network (TBON) of processes between the tool's front-end and back-ends. MRNet uses the TBON to distribute many important tool communication and computation activities, reducing analysis time and keeping tool front-end loads manageable.

MRNet-based tools send data between front-end and back-ends on logical flows of data called streams. MRNet internal processes use filters to synchronize and aggregate data sent to the tool's front-end. Using filters to manipulate data in parallel as it passes through the network, MRNet can efficiently compute averages, sums, and other more complex aggregations on back-end data".

Implementing and running an MRNet application requires to follow a series of steps that are common to most programs. These steps include: request resources for the TBON; map the available resources in a tree-like topology; start the front-end process; instantiate the MRNet internal nodes; and in the case where MRNet runs in the "back-end attach" mode, spawn the back-ends manually, pass the network connectivity information from the front-end to the back-ends externally (via the environment, shared filesystems or other information services), connect the back-ends; and load the filters into the network's internal processes and announce them.

MRNet provides a C++ API to facilitate each of these operations, yet the logical sequence of operations is the same for all programs and is rather long. Furthermore, MRNet support for BlueGene architectures requires of an entirely separate and slightly different API (known as *lightweight*), which is written in pure C. Synapse encapsulates all these common operations to all MRNet applications, and provides wrappers to hide the complexity of using two different API's depending on the architecture, making the development of a new program as easy as writing a few lines of code.

Sound understanding of the MRNet software is necessary to follow this manual, so please refer to [1] for a complete description of MRNet.

Chapter 2

Abstractions

Synapse provides a set of libraries and classes that can be extended to define the interactions between your front-end, filters and back-end processes. Synapse handles all the operations necessary to initiate and connect all the processes in the network, and just leaves to the user the task of implementing the logic of the algorithm that these processes will perform. To this end, Synapse provides the *Protocol* abstraction. A *Protocol* is the set of operations that determine how the MRNet processes interact. For example, imagine a trivial "ping-pong" protocol, where the back-ends send an ACK to the front-end, and the front-end replies with how many ACK's it has received. This and any other protocol has two parts: the part that is run at the back-ends, and the part that is run at the front-end. In this simple case, the back-ends have to send the ACK and receive the reply, while the front-end has to receive the ACK's and send the reply. In order to implement the logic of the front-end and the back-ends, Synapse provides two separate classes: FrontProtocol and BackProtocol. When the user wants to define its own protocol to run on top of the MRNet, he has to write the code that refers to the operations performed by the front-end extending the FrontProtocol class, and the complementary operations performed by the back-ends extending the BackProtocol class. More specifically, the user has to implement the Run() method, and write inside this method all the computations and communications that take place in the protocol.

Once this work is done, that protocol can be easily loaded into any MRNet application through the FrontEnd::LoadProtocol and BackEnd::LoadProtocol methods, and triggered anytime through FrontEnd::Dispatch. Synapse will start the protocol in all the end-points of the MRNet application, and ensure they meet a synchronization point once the protocol is over so that the next one can start.

Chapter 3

A simple example

A detailed description of the Synapse API is in Section 4. This section shows a brief example of a MRNet application that implements a "ping-pong" protocol. In this example, the back-ends send an ACK to the front-end, and the front-end replies with how many ACK's were received.

3.1 Front-End sample code

This piece of code creates the front-end end-point of the MRNet application, loads the front-end side of the "ping-pong" protocol, and then starts the protocol.

```
#include <iostream>
#include "FrontEnd.h"
#include "Ping_FE.h"

int main(int argc, char *argv[])
{
   /* Start the front-end side of the network */
   FrontEnd *FE = new FrontEnd();
   FE->Init("topology_1x4.txt", "./test_BE", NULL);

   /* Load the protocols the network understand */
   FrontProtocol *prot = new Ping();
   FE->LoadProtocol( prot )  ;

   /* Execute protocol "PING" */
   int status;
   FE->Dispatch("PING", status);

   /* Shutdown the network */
   FE->Shutdown();

   return 0;
}
```

3.2 Back-End sample code

This piece of code creates the back-end end-point process of the MRNet application, loads the back-end side of the "ping-pong" protocol, and waits for the front-end to orchestrate the start of the protocol.

```
#include "BackEnd.h"
#include "Ping_BE.h"

int main(int argc, char *argv[])
{
   /* Start the back-end side of the network */
   BackEnd *BE = new BackEnd();
   BE->Init(argc, argv);

   /* Load the protocols the network understand */
   BackProtocol *prot = new Ping();
   BE->LoadProtocol(prot);

   /* The back-end enters the main analysis loop,
      waiting for commands from the front-end */
   BE->Loop();

   return 0;
}
```

3.3 Front-End side of the "ping-pong" protocol sample code

The following code defines the front-end side of the protocol. The user has to extend the Setup() method to initialize all the streams that are going to be used during the execution of the protocol, and also the Run() method that implements the protocol. In this case, the front-end first receives all the ACK's (through the aggregation stream stAdd), and then broadcasts to the back-ends the total number of ACK's received.

```
#include <iostream>
#include "Ping_FE.h"

/**
 * In the Setup function we have to register all the streams we want
 * to use for this protocol. When the function returns, all streams
 * pushed to the registeredStreams queue are automatically published
 * to the back-ends. The Register_Stream is a wrapper to
 * net->new_Stream() that creates a new stream in the FE and pushes
 * it to the queue.
 */
void Ping::Setup()
```

```
{
    stAdd = Register_Stream(TFILTER_SUM, SFILTER_WAITFORALL);
    cout << "[FE] Created new stream " << stAdd->get_Id() << endl;
}

/**
 * Implement the front-end side of the protocol.
 * It is expected to return 0 on success; -1 otherwise.
 */
int Ping::Run()
{
    int tag, countPongs = 0;
    PacketPtr p;

    cout << "[FE] Sending PING to " << stAdd->size()
         << " back-ends through stream " << stAdd->get_Id() << endl;
    MRN_STREAM_SEND(stAdd, TAG_PING, "");
    cout << "[FE] Waiting for PONG from " << stAdd->size()
         << " back-ends..." << endl;
    MRN_STREAM_RECV(stAdd, &tag, p, TAG_PONG);
    p->unpack("%d", &countPongs);
    cout << "[FE] " << countPongs << " PONGs received!" << endl;

    if (countPongs == stAdd->size())
    {
        cout << "[FE] Addition filter ran successfully! :)" << endl;
        return 0;
    }
    else
    {
        cout << "[FE] Addition filter FAILED! :(" << endl;
        return -1;
    }
}
```

3.4 Back-End side of the "ping-pong" protocol sample code

The following code defines the back-end side of the protocol. The user has to extend the Setup() method to initialize all the streams that are going to be used during the execution of the protocol, and also the Run() method that implements the protocol. In this case, the back-end first sends an ACK's to the front-end (through the aggregation stream stAdd), and then receives the reply with the total number of ACK's that the front-end received.

```
#include <iostream>
#include "Ping_BE.h"
```

```
/**
 * The streams created in the front-end are received here,
 * in the same order that were created.
 */
void Ping::Setup()
{
   Register_Stream(stAdd);
}

/**
 * Implement the back-end side of the protocol.
 * It is expected to return 0 on success; -1 otherwise.
 */
int Ping::Run()
{
   int tag;
   PACKET_new(p);

   MRN_STREAM_RECV(stAdd, &tag, p, TAG_PING);
   MRN_STREAM_SEND(stAdd, TAG_PONG, "\%d", 1);

   PACKET_delete(p);

   return 0;
}
```

Chapter 4

Synapse API

Synapse provides several classes to help building front-end and back-end MRNet processes and the protocols they execute. These are bundled into two libraries (libsynapse_frontend and libsynapse_backend) for the respective end-point processes.

4.1 Class FrontEnd

An instance of this class is needed to make your front-end process. This class provides basic methods to start the network.

Init (normal mode)

Synopsis

```
int Init(const char *TopologyFile, const char *BackendExe,
         const char **BackendArgs);

int Init(const char *BackendExe, const char **BackendArgs);
```

Description The basic method that is used to create the network and spawn the back-ends (normal instantiation mode). *TopologyFile* is the path to a configuration file that defines the desired process tree topology. The specification format is the same that the one produced by the mrnet_topgen tool. *BackendExe* is the path to the binary that will act as the application back-end process. *BackendArgs* is a null terminated list of arguments to pass to the back-end application upon creation.

If *TopologyFile* is not given, this information is read from the environment variable MRNAPP_TOPOLOGY.

Return value Returns 0 if the MRNet starts successfully; -1 otherwise.

Init (back-end attach mode)

Synopsis

```
int Init(const char *TopologyFile, unsigned int numBackends,
         const char *ConnectionsFile, bool wait_for_BEs=true);

int Init(bool wait_for_BEs=true);
```

Description Starts the MRNet except the back-ends (back-end attach mode), and waits for these to connect. *TopologyFile* is the path to a configuration file that defines the desired process tree topology (not including the back-ends). *numBackends* is the number of back-ends that will be created outside the MRNet application. *ConnectionsFile* is the path to a file where the necessary information for the back-ends to connect to the network (hosts and ports of their parent processes) will be written to. *wait_for_BEs* is an optional argument (set by default). When set, the front-end will wait for all the back-ends to connect and then complete all the initializations. Otherwise, the user will have to call to Connect() later to complete the initialization.

If *TopologyFile*, *numBackends* and *ConnectionsFile* are not given, this information is read from the environment variables MRNAPP_TOPOLOGY, MRNAPP_NUM_BE and MRNAPP_BE_CONNECTIONS.

Return value Returns 0 if the MRNet starts successfully; -1 otherwise.

Connect

Synopsis

```
int Connect();
```

Description In the back-end attach instantiation mode, this is the second part of the Init() function. Init() can call this function automatically if specified, otherwise you have to call it manually. This is implemented to support the use-case where the front-end and the back-ends are threads of the same MPI process and you need to get the control back after the front-end initialization to distribute the pending connection information among the MPI tasks to start the back-ends.

Return value Returns 0 on success; -1 otherwise;

isConnectionsFileWritten

Synopsis

```
bool isConnectionsFileWritten();
```

Description This method exists to control in multi-threaded programs not to start parsing the file before all the necessary information has been totally dumped.

Return value Returns true if the necessary information for the back-ends to connect to the network (hosts and ports of their parent processes) has been written completely.

ConnectedBackEnds

Synopsis

```
int ConnectedBackEnds(void);
```

Return value Returns the number of back-ends that are connected to the network.

LoadProtocol

Synopsis

```
int LoadProtocol(Protocol *prot);
```

Description Loads the user-defined protocol *prot* for the front-end side of the MRNet.

Return value Returns 0 on success; -1 otherwise;

LoadFilter

Synopsis

```
int LoadFilter (string filter_name);
```

Description Looks for the filter shared object specified by *filter_name* (appending .so) in the paths specified with the environment variable MRNAPP_FILTER_PATH. If the filter is found, it is loaded into the network.

Return value Returns the filter identifier; or -1 if can not be found or loaded.

Dispatch

Synopsis

```
int Dispatch(string protID, int &status, Protocol *& prot);
int Dispatch(string protID, int &status);
```

Description Tells the back-ends the next protocol the network is going to execute and runs it. *prot_id* is the protocol identifier. *status* is a parameter by reference that captures the return code of the protocol. *prot* is an optional parameter that returns the instance of the protocol that was executed, that the user can use to retrieve results from the execution of the protocol.

Return value Returns 0 on success; -1 otherwise.

Shutdown

Synopsis

```
void Shutdown(void);
```

Description Notifies the back-ends to exit and shutdowns the MRNet.

isUp

Synopsis

```
bool isUp();
```

Return value Returns true if the network has started correctly and the front-end is ready to issue protocols.

4.2 Class BackEnd

Instances of this class are needed to make your back-end processes. This class provides basic methods to connect the leaves of the network.

Init (normal mode)

Synopsis

```
int Init(int argc, char *argv[]);
```

Description Starts the network back-end. argc and argv are the number of arguments and a NULL-terminating list of arguments that the back-end binary receives, respectively.

Return value Returns 0 on success; -1 otherwise.

Init (back-end attach mode)

Synopsis

```
int Init(int wRank, const char *connectionsFile);

int Init(int wRank, char *parHostname, int parPort, int parRank);

int Init(int wRank);
```

Description Starts the network attaching the pending back-ends (BE attach mode). *wRank* is the back-end rank identifier.

connectionsFile is the path to a file that contains all the parent's hosts and ports where each back-end has to connect.

Alternatively, you can manually provide the host, port and rank by setting *parHostname*, *parPort* and *parRank*. This variant was implemented to avoid stressing the filesystem by reading the connections file simultaneously from many back-ends.

If none of these arguments is provided, the path to the connections file is read from the environment variable MRNAPP_BE_CONNECTIONS.

Return value Returns 0 on success; -1 otherwise.

LoadProtocol

Synopsis

```
int LoadProtocol(Protocol *prot);
```

Description Loads a user-protocol for the back-end side of the MRNet.

Return value Returns 0 on success; -1 otherwise.

Loop

Synopsis

```
void Loop();

void Loop(callback_function preProtocol, callback_function postProtocol);
```

Description The back-end enters a loop waiting for requests from the front-end. When the front-end dispatches protocol, the back-end executes the counterpart back-end side of the same protocol. The loop exits when the front-end dispatches a message with tag TAG_EXIT (when calling Shutdown()).

If *preProtocol* and *postProtocol* are given, these callbacks are executed before and after the protocol is executed. This was introduced to prepare input data and retrieve results.

Shutdown

Synopsis

```
void Shutdown();
```

Description Gracefully disconnects the back-end end-point from the MRNet.

4.3 Class MRNetApp

Common class inherited both from FrontEnd and BackEnd. Provides common methods to identify the processes.

NumBackEnds

Synopsis

```
unsigned int NumBackEnds   (void);
```

Description Queries the MRNet Network Topology for the total number of back-ends.

Return value Returns how many back-ends are in the network.

WhoAmI

Synopsis

```
unsigned int WhoAmI(bool return_network_id=false);
```

Description If *return_network_id* is set, returns the real ID for the backends in the network topology (range from 1000000 to 1000000+N). Otherwise, returns the logical ID for the remote back-ends (range from 0 to N).

Return value Returns the rank of the current MRNet process.

isFE

Synopsis

```
bool isFE (void);
```

Return value Returns true if the calling process is the network front-end.

isBE

Synopsis

```
bool isBE (void);
```

Return value Returns true if the calling process is a network back-end.

4.4 Class Protocol

All user-defined protocols extend this class through the inheritance of FrontProtocol and BackProtocol, that are necessary to define the logic executed at the different end-points of the network. When writing your FrontProtocol and BackProtocol objects, you must provide an implementation for the following methods.

ID

Synopsis

```
string ID (void);
```

Description Sets a textual identifier for the protocol (e.g. "PINGPONG"). The identifier must coincide in the front-end side of the protocol (FrontProtocol object) and the back-end side of the protocol (BackProtocol object).

Setup

Synopsis

```
void Setup (void);
```

Description When a given protocol starts executing, its Setup() method will be executed first. In the Setup() method you have to register all the streams (and filters) that are going to be used during the execution of the protocol. In order to register streams, it is necessary to make calls to FrontProtocol::Register_Stream (in the front-end side of the protocol) and BackProtocol::Register_Stream (in the back-end side of the protocol), **in the same order**.

Run

Synopsis

```
int Run (void);
```

Description The protocol to execute has to be defined inside this method.

Return value Returns the return code of the protocol.

4.5 Class FrontProtocol

The front-end side of a protocol has to inherit this class, and implement the generic methods `ID`, `Setup` and `Run` (see 4.4).

Barrier

```
int Barrier (void);
```

Description Blocks the front-end protocol until `BackProtocol::Barrier` is called by all back-ends.

Return value Returns 0 on success; -1 otherwise.

Register_Stream

Synopsis

```
STREAM * Register_Stream(int up_transfilter_id, int up_syncfilter_id);

STREAM * Register_Stream(string filter_name, int up_syncfilter_id);
```

Description Front-end wrapper for MRNet::Network::new_Stream that stores the newly created stream in a registration queue. When a protocol is loaded, all registered streams are published automatically to the back-ends. *up_transfilter_id* is the transformation filter to apply to data flowing upstream (the default is not to apply any filter, TFILTER_NULL). *up_syncfilter_id* is the synchronization filter to apply to upstream packets (the default is to wait for all children, SFILTER_WAITFORALL).

If *filter_name* is specified, the filter identified by *filter_name* is loaded into the network and linked to the new stream.

Return value Returns the new stream.

4.6 Class BackProtocol

The back-end side of a protocol has to inherit this class, and implement the generic methods ID(), Setup() and Run() (see 4.4).

Barrier

```
int Barrier (void);
```

Description Blocks the calling back-end waiting for the remaining back-ends to call BackProtocol::Barrier() and the front-end to call FrontProtocol::Barrier().

Return value Returns 0 on success; -1 otherwise.

Register_Stream

Synopsis

```
void Register_Stream(STREAM *& new_stream);
```

Description Retrieves a new stream that was registered in the front-end. The streams have to be registered in the same order than in the front-end!

Return value Returns the stream that was registered in the front-end.

4.7 Class PendingConnections

This clas provides generic methods to exchange the connections information from the front-end to the back-ends outside of the MRNet application. Currently it supports distribution of this information through shared filesystems and through an MPI network.

PendingConnections

Synopsis

```
void PendingConnections (string ConnectionsFile);
```

Description Constructor that receives the file where the connections information has to be written (front-end), or from where has to be read (back-ends).

Write

Synopsis

```
int Write(NETWORK *net, unsigned int numBackends);
```

Description This front-end method queries the connection information from the network and dumps it into a file to share with the back-ends via a shared filesystem.

Return value Returns 0 on success; -1 otherwise.

GetParentInfo

Synopsis

```
int GetParentInfo(int rank, char *phost, char *pport, char *prank);
```

Description This back-end method retrieves the *host*, *port* and *rank* where a given backend has to connect from the connections file.

Return value Returns 0 on success; -1 otherwise.

ParseForMPIDistribution

Synopsis

```
int ParseForMPIDistribution(int world_size, char *&sendbuf,
                            int *&sendcnts, int *&displs);
```

Description This back-end method parses the connections file and serializes the data into arrays to be distributed through MPI scatter. This method is meant for the use-case where the back-ends are spawned through MPI, and just one process reads the connection information and distributes the data to the rest through MPI. This was implemented to avoid stressing the filesystem because of many back-ends reading the same file simultaneously.

Parameter *world_size* is the total number of back-ends. *sendbuf* is the address of send buffer to store the data to send to each process. *sendcnt* is the address of the integer array to specify in entry i the number of elements to send to processor i. *displs* is the address of the integer array to specify in entry i the displacement (relative to sendbuf) from which to take the outgoing data to process i.

Return value Returns 0 on success; -1 otherwise.

Chapter 5

Synapse wrappers

Synapse provides several wrappers to unify the lightweight API for BlueGene architectures and the standard C++ API. These wrappers are defined as macros in upper case. There are wrappers available for the basic MRNet objects Network, Stream and Packet:

- NETWORK
- NETWORK_PTR
- STREAM
- STREAM_PTR
- PACKET
- PACKET_PTR

There are also wrappers for the basic methods of these objects. The wrappers take the same arguments as the MRNet routines:

- STREAM_recv(stream, tag, data, block)
- STREAM_get_Id(stream)
- STREAM_send(stream, tag, format, args...)
- STREAM_flush(stream)
- STREAM_is_Closed(stream)
- STREAM_delete(stream)
- PACKET_unpack(p, fmt, args...)
- PACKET_new(p)
- PACKET_delete(p)
- NETWORK_recv(net, tag, data, stream, block)

- NETWORK_CreateNetworkBE(argc, argv)
- NETWORK_get_LocalRank(net)
- NETWORK_waitfor_ShutDown(net)
- NETWORK_delete(net)
- NETWORK_get_NumBackEnds(net, num_be)

And additionally provides a few communication primitives:

- `MRN_STREAM_SEND(stream, tag, format, args...)`

Broadcast to all back-ends.

- `MRN_STREAM_SEND_P2P(stream, be_list, tag, format, args...)`

Sends a message to the subset of back-ends in the stream specified in *be_list*.

- `MRN_STREAM_RECV(stream, tag, data, expected_tag)`

Receive from a specific stream (blocking receive).

- `MRN_STREAM_RECV_NONBLOCKING(stream, tag, data, expected_tag)`

Receive from a specific stream (non-blocking receive).

- `MRN_NETWORK_RECV(net, tag, data, expected_tag, stream, blocking)`

Receive from any stream.

Chapter 6

Synapse configuration tool

Synapse provides a configuration tool that helps compiling and linking a MRNet-based application. It can be queried (for example, from a Makefile or building system) for the following information:

Synopsis

```
synapse-config <option>
```

Options

- –prefix: print Synapse installation directory
- –fe-cflags: prints pre-processor and compiler flags for the front-end
- –fe-libs: prints library linking information for the front-end
- –be-cflags: prints pre-processor and compiler flags for the back-ends
- –be-libs: prints library linking information for the back-ends
- –cp-cflags: prints pre-processor and compiler flags for the filters
- –libdir: prints the library directory
- –rpath: prints run-time search path flags for the shared libraries
- –libtool-rpath: prints the rpath flags used by libtool
- –mrnet: prints MRNet installation directory
- –version: prints version information

Bibliography

[1] U. of Wisconsin. MRNet API Programmer's Guide. http://www.paradyn.org/mrnet.

[2] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 21, Washington, DC, USA, 2003. IEEE Computer Society.