NOS Engine User's Manual

January 7, 2016

Submitted to:

Independent Verification and Validation (IV&V)

Administrative Office
2050 Winners Drive
Fairmont, WV 26554
304.816.3600

Point of Contact
Randy Hefner, PM
TMC Technologies
Phone: (304) 816-3605
Cell: (304) 672-2530

Table of Contents

Table of Figures

1 Introduction

NOS Engine is a message passing middleware designed specifically for use in simulation. With a modular design, the library provides a powerful core layer that can be extended to simulate specific communication protocols. With advanced features like time synchronization, data manipulation, and fault injection, NOS Engine provides a fast, flexible, and reusable system for connecting and testing the pieces of a simulation.

1.1 Document Overview

After the introduction, the remainder of the document is divided into 4 more chapters. Chapter 2, Concepts, explains the design of NOS Engine at a conceptual level. Chapter 3, Basics, walks through everything you need to know to use the core capabilities of NOS Engine. Chapter 4 and 5 are devoted to protocol layer implementations, MIL-STD-1553 and SpaceWire, respectively.

2 Concepts

NOS Engine is built on a conceptual model based on two fundamental types of objects: nodes and buses. A node is any type of endpoint in the system, capable of sending and/or receiving messages. Any node in the system has to belong to a group, formally referred to as a bus. A bus can have an arbitrary number of nodes, and each node within the bus must have a name that is unique to all other member nodes. The nodes of a bus operate in a sandbox; a node can communicate with another node on the same bus, but cannot talk to nodes that are members of a different bus. **Figure 1** depicts the bus and node relationship.

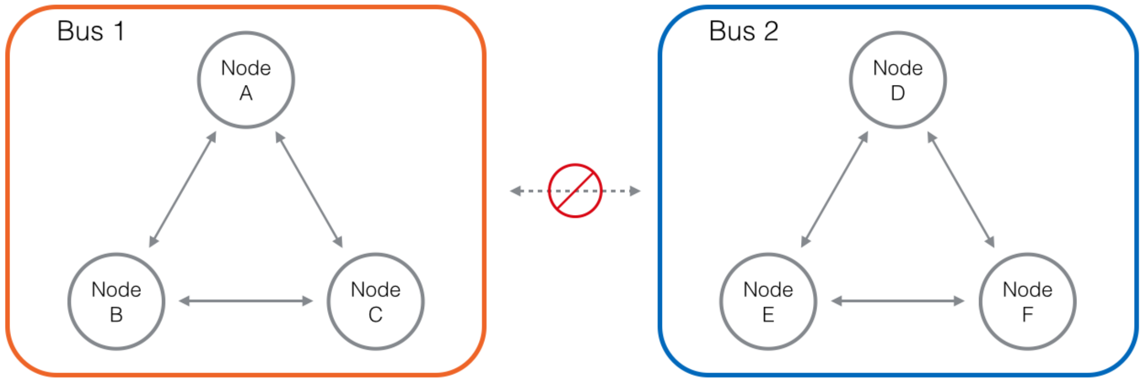

Figure 1: Bus & Node Topology

2.1 Messaging

Nodes live a simple life, with their only real function being the exchange of messages. Messages are any set of arbitrary data - there are no restrictions on the contents of a message, in either size or format. This allows maximum flexibility, but of course requires the nodes participating in the communication to be aware of how to handle the payload.

Messages are transferred as an atomic unit – they will arrive at the destination the same way they are sent from the source. This is in direct contrast to streaming protocols, such as TCP, where the receiving end can read partial messages or multiple messages at a time. With NOS Engine, it is not necessary to use any start-of-message or end-of-message markers; the system will guarantee that separate messages are delivered as separate messages.

Another key concept is that all message communication happens asynchronously within the bus. This means that multiple conversations can be happening simultaneously on the bus without any blocking or interference on each other. This allows for maximum performance on the bus, but as a result does not guarantee deterministic delivery of messages. **Figure 2** shows a representation of the behavior. If deterministic delivery is required, synchronization will have to be implemented on top.

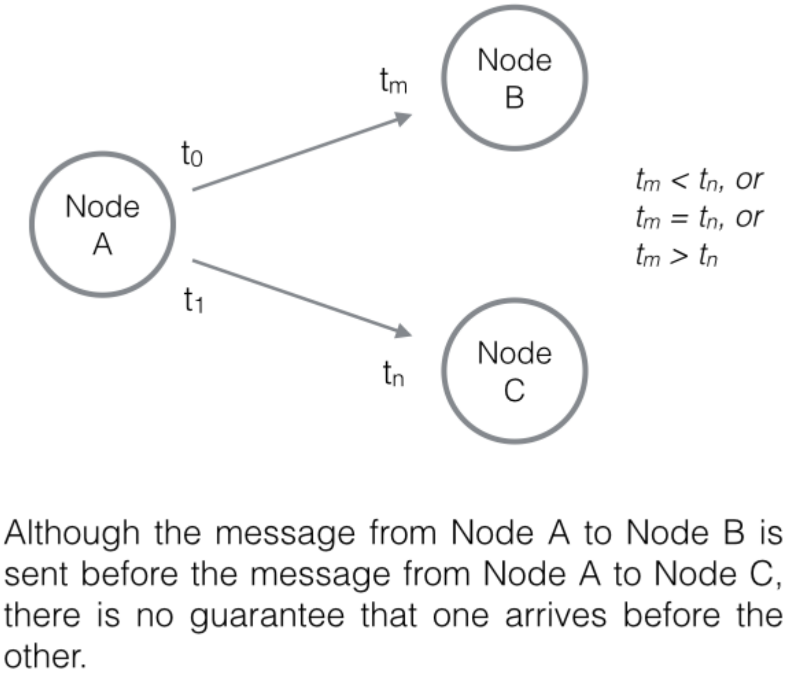
Node A
Node B
Node C
t_0
t_m
t_1
t_n
$t_m < t_n$, or
$t_m = t_n$, or
$t_m > t_n$
Although the message from Node A to Node B is sent before the message from Node A to Node C, there is no guarantee that one arrives before the other.

Figure 2: Asynchronous Messaging

2.2 Interception

Interception is a special feature that is baked into the gooey center of NOS Engine. Interception is the capability to view and manipulate messages mid-transfer – either discarding messages or modifying the contents. This is specifically useful in simulations for generating fault scenarios dynamically. For example, a communication link could be made to appear as disconnect (blocking messages), or sensor values in a telemetry packet could be modified at will to test system response.

Interception is possible through the use of a specialized node known as an interceptor. An interceptor is akin to a man-in-the-middle; communication between two endpoints is relayed through the interceptor. The interceptor can view the message and choose a course of action – pass it along as normal, modify the contents, or drop the message entirely, as shown in **Figure 3**.

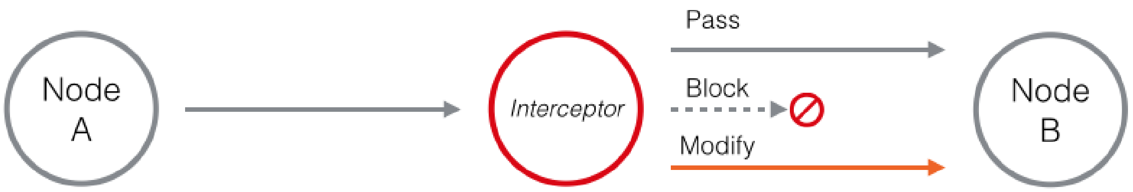

Figure 3: Interceptor Capabilities

3 Basics

3.1 Client-Server Setup

NOS Engine operates in a client-server configuration. In this setup, a central server acts as the gateway for all communication. Instead of any clients communicating directly with each other, the server facilitates all message transfers. This architecture provides the simplest solution for coordinating a distributed system and providing features like interception. The downside is that the server could become a bottleneck and impact performance if resources for the server application are insufficient.

A single server is capable of supporting an arbitrary number of busses simultaneously. Busses managed under a single server are isolated and operate independently. Typically, a single server instance will be enough to support simulation activity across multiple busses. **Figure 4** depicts a simple setup with a server supporting multiple busses. In the event a server instance becomes saturated, additional server instances could be brought up on other machine(s) to alleviate the bottleneck. The server instances do not perform any automated load balancing between each other; they act independently and will have to be configured manually to provide the desired performance.

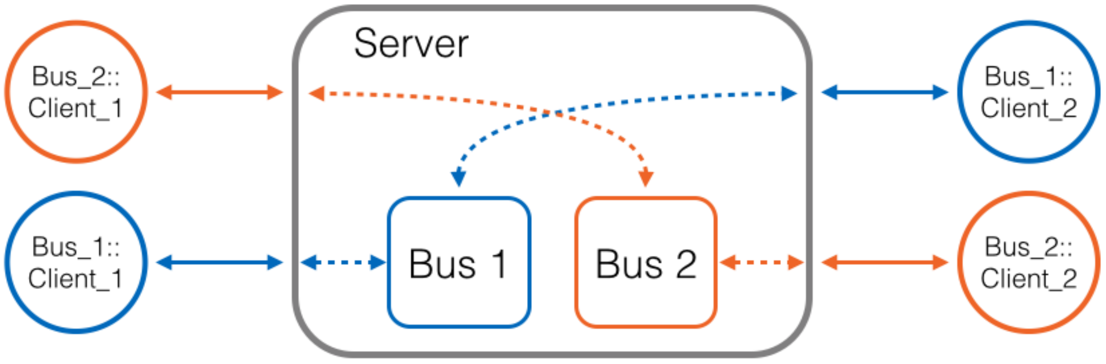

Figure 4: Server With Multiple Busses

Setting up a NOS Engine system is a two-step process:

1. Startup a server instance which will wait for clients to connect
2. Create clients and connect to the server

Subsections 3.1.1 – 3.1.5 will break down the process and provide some basic examples.

3.1.1 The Server

A NOS Engine server instance can be brought up in one of two ways: as a standalone application (separate executable), or built-in to another application. A standalone server must be coordinated to be available before client applications start, but provides independence from the lifetime of any client application (i.e., a client application that exits will not shutdown the server).

In contrast, a built-in server can easily and programmatically be made to startup before any client-side activity. Another benefit is the availability of the Copy Transport (details in section 3.1.3.4). The caveat of running the server within another application is that the server will go down with the ship, so to speak (which could be a pro or con depending on your configuration).

With either choice, the process is nearly identical and only differs in boilerplate.

3.1.1.1 Standalone Application

To run a server in a standalone application, the following needs to happen:

1. Create an instance of the Server object
2. Add transport(s)
3. Maintain the lifetime of the Server object until it is time to shutdown

Enough with the dialogue, let's jump into some code!

```
#include <Server/Server.hpp>

int main(int, char**)
{
  // 1) Create instance of server object
  NosEngine::Server::Server server;

  // 2) Add transport(s)
  server.add_transport("Transport1", "tcp://localhost:29556");

  // 3) keep server object around until time to shutdown
  while(keep_running)
  {
    // poll for exit condition (signal, keyboard, etc.)
    .
    .
    .
  }

  return 0;
}
```

Figure 5: The Simplest Standalone Server

Figure 5 demonstrates how simple it is to write a standalone server application. The first two steps are each a single line (specifics on transports will be laid out in section 3.1.3). After that, all that is necessary is to keep the server object around and the executable running until your desired exit condition occurs. Easy as pie – or perhaps cake!

3.1.1.2 Built-In to Another Application

The procedure for running the server is nearly identical to a standalone application, except that the add_transport calls should be formed in a background thread. The reason is that the add_transport calls will block while the transport connection is being established. Since your application will likely be one of the clients attempting to connect, it will be necessary to have the operations in separate threads to avoid deadlock. A simple approach that uses C++11 thread support is shown in **Figure 6**.

During the add_transport call, the connection is established and worker threads are spawned automatically to handle the operations of the server. Piece of cake!

```
#include <Server/Server.hpp>
#include <thread>

int main(int, char**)
{
  // 1) Create instance of server object
  NosEngine::Server::Server server;
  // 2) Add transport(s) (in a background thread)
  std::thread server_connectors([&server]() {
    server.add_transport("Transport1", "tcp://localhost:29556");
  });

  // 3) application specific work (connecting clients to the server, any arbitrary
work)
    .
    .
    .

  // clean-up thread
  if (server_connectors.joinable())
  {
    server_connectors.join();
  }

  return 0;
}
```

Figure 6: Simple Built-In Server

3.1.2 Connecting Clients - Bus Registration

Once the server is running, the next step is to get your application connected to the server. A connection is established through the client-side Bus object. A bus object has only two parameters for instantiation: the transport connection string, and the name of the bus. The end result is a single line of code to get connected – see **Figure 7** below.

```
#include <Client/Bus.hpp>

int main(int, char**)
{
  // connect to server – specify transport string & bus name
  NosEngine::Client::Bus my_bus("tcp://localhost:29556", "Short_Bus");

  // create nodes with my_bus
  .
  .
  .

  // perform message exchange between nodes
  .
  .
  .

  return 0;
}
```

Figure 7: Connecting a Client to the Server

In the example, a new bus object is created for a bus called Short_Bus. The server will automatically create an internal representation of this bus, if it doesn't already exist. Any other client applications that wish to have nodes on this bus will create a client bus object by using the same bus name (the connection string will be different though). The server recognizes that multiple clients have connected and wish to participate on Short_Bus; it handles the dirty work of message distribution between each of the clients and their nodes, without the clients having to know any specifics about each other. Neat!

3.1.3 Transports

Up to this point, the details of the transport model have been delicately avoided. But no longer – let's go!

NOS Engine is able to support multiple types of transports. The currently supported transports are TCP, Inter-Process Communication (IPC), and a special Copy transport. Each transport is implemented with the same generic interface, which closely emulates a stream socket, providing methods to listen and connect. Within NOS Engine, the server will always perform the listen operation for any transport; this is the operation that causes the Server::add_transport call to block. Once a client tries to connect (this happens internally when a Client::Bus object is created), the connection can be fully established and the listen call can return.

3.1.3.1 Transport Connection String

To identify a transport, a transport connection string is used. This simple string provides a unified way to specify the type of transport and the configuration parameters – similar to a Uniform Resource Locator (URL).

The connection string is divided into two parts:

- Transport Type Identifier
- Configuration parameters, delimited with a colon

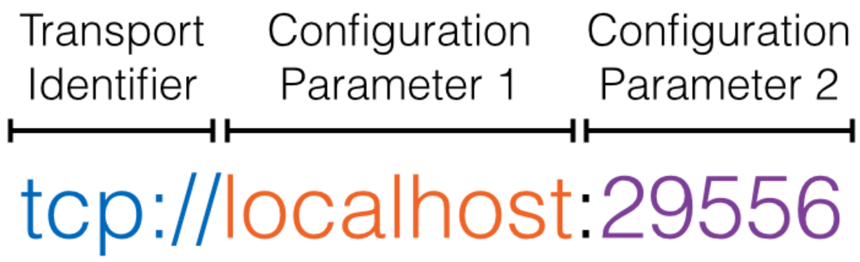

Figure 8: A TCP Connection String

Let's breakdown the transport connection string shown in **Figure 8**. The first part, tcp://, indicates the type of transport – this time referring to the TCP transport. The remainder of the string is for configuration parameters. Each parameter is delimited with a colon. The first parameter, localhost, specifies the hostname of the target machine (this could also be an IP address). The third part of this string is the last configuration parameter; in this case it is simply the port number used to connect/listen.

The configuration parameters for each transport connection string will vary based on each transport; see the section specific to each transport for more details.

3.1.3.2 TCP Transport

The TCP transport is, predictably, implemented with TCP sockets. TCP provides good performance and is the most versatile of the available transports. The TCP transport can support connections within a process, between processes on the same machine, or between processes on different machines.

If you are not sure which transport to use or you don't want to think about it, use this one – it will work in any configuration.

3.1.3.3 TCP Transport Connection String

```
tcp://<ip_address/host_name>:<port number>
```

Transport Identifier

The identifier for the TCP transport, *tcp://*, is placed at the beginning of the string.

IP Address or Hostname

The first parameter is the IP address (or hostname).

For the client side bus, this value should be the IP address/hostname of the machine running the server. The loopback address (127.0.0.1) or *localhost* can be used if the client and server are running on the same machine.

For the server, this value will be ignored and should be set to the default value of *localhost*.

Port Number

This parameter is a positive integer in the range 0-65535, written as plain text.

Ports 0-1023 should not be used as these are well-known ports for other services.

An example of TCP transport connection string was shown previously in **Figure 8**. IPC Transport

The IPC (Inter-Process Communication) transport is implemented with Unix domain sockets (or named pipes if you're stuck with a Windows system – though a Window's release is not currently supported in Release Version 1.0.). The IPC transport can perform faster than the TCP transport, but it is limited to processes running on the same machine.

3.1.3.3.1 IPC Transport Connection String

```
ipc://<unique_name>:<max_bytes>
```

Transport Identifier

The identifier for the IPC transport, *ipc://*, is placed at the beginning of the string.

Unique Name

Each instance of the IPC transport requires a unique name. Try to keep names brief as there may be platform-specific limits on the length of the name.

When connecting a client to the server, the unique name in the connection string for either should be the same.

Max Bytes

The last parameter specifies the maximum number of bytes for any one message (multiple message can be queued though). This value should be set to the size of the largest message that may be sent **plus** some additional bytes for the internal message header info. The message header size can vary (especially if bus/node names are long), but a relatively safe value for the additional space is 256 bytes.

When connecting a client to the server, the max bytes value in the connection string for either should be the same.

3.1.3.4 Copy Transport

The copy transport is a special transport that can only be used within a process – i.e., when the server and client(s) are running in the same application, as in **Figure 6**. The copy transport is the fastest transport available – it does not require the networking stack, and does not use any system calls (other than mutex calls for critical section operations). And, despite its name, the copy transport does the least amount of copying of any of the transports – it does the least amount of work possible to shuffle a message from the sender to the receiver.

If you are using a local server in the same application as your client busses, you should definitely use the copy transport! Or if you defaulted to the TCP transport and didn't read this guide up until now, switch to the copy transport now and brag about the performance improvements you made!

3.1.3.5 Copy Transport Connection String

```
copy://
```

Transport Identifier

The identifier for the copy transport, *copy://*, is placed at the beginning of the string.

The copy transport connection string does not have any parameters and it is the same for the client bus or the server. One caveat – if you are going to connect multiple in-app client busses to the local server, you must call *Server::add_transport* once for each client bus.

3.1.4 Example #1: Connecting All the Pieces

Let's go through an example that demonstrates connecting the server and clients with all of the different transports. The source code for this example is based on the topology shown in **Figure 9**. Machine 1 has two applications – one with a server and a client, and another application with only a client. Machine 2 has a single application running another client. Sample source code is shown below (one file for each application), demonstrating how everything can be connected together.

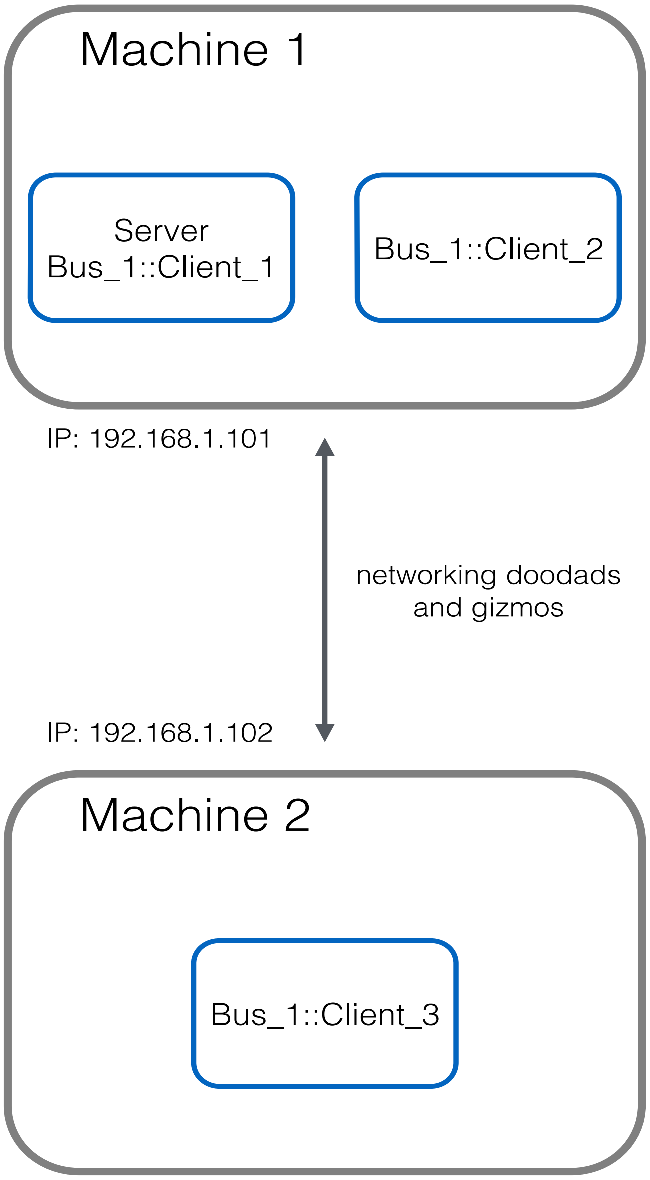

Figure 9: Example #1 Topology

```
FILE: Machine_1_Server_and_Client_1.cpp

#include <Server/Server.hpp>
#include <Client/Bus.hpp>
#include <thread>

int main(int, char**)
{
  // 1) Create instance of server object
  NosEngine::Server::Server server;

  // 2) Add transport(s) (in a background thread)
  std::thread server_connectors([&server]() {
    server.add_transport("Client_1", "copy://");
    server.add_transport("Client_2", "ipc://Client_2:1000");
    server.add_transport("Client_3", "tcp://localhost:29556");
  });

  // 3) application specific work (connecting clients to the server, any arbitrary
work)

  NosEngine::Client::Bus bus_1("copy://", "Bus_1");

  // create nodes with bus_1
  .
  .
  .

  // perform message exchange between nodes
  .
  .
  .

  // clean-up thread
  if (server_connectors.joinable())
  {
    server_connectors.join();
  }

  return 0;
}
```

```
FILE: Machine_1_Client_2.cpp

#include <Client/Bus.hpp>

int main(int, char**)
{
  // connect to server – specify transport string & bus name
  NosEngine::Client::Bus bus_1("ipc://Client_2:1000", "Bus_1");

  // create nodes with bus_1
  .
  .
  .

  // perform message exchange between nodes
  .
  .
  .

  return 0;
}
```

```
FILE: Machine_2_Client_3.cpp

#include <Client/Bus.hpp>

int main(int, char**)
{
  // connect to server – specify transport string & bus name
  NosEngine::Client::Bus bus_1("tcp://192.168.1.101:29556", "Bus_1");
  // create nodes with bus_1
  .
  .
  .

  // perform message exchange between nodes
  .
  .
  .

  return 0;
}
```

3.2 Registering Data Nodes

If you've been following from the beginning, you are now able to run a server and create a bus with connected clients. To do anything worthwhile, you'll need to add some nodes to the bus. Registering nodes is very easy, and there is really only one requirement – each node must have a name that is unique from any other node (independent of type) on the bus. In this section we will be specifically talking about registering *data nodes*. Data nodes are a type of node that is capable of directly sending and receiving messages to other data nodes.

The *Client::Bus* object provides 3 methods for registration-type operations on data nodes:

```
Client::DataNode* get_data_node(std::string name)
Client::DataNode* get_or_create_data_node(std::string name)
void        remove_data_node(const std::string& name)
```

get_data_node

This method is a convenience method for retrieving a node that has already been successfully registered with *get_or_create_data_node***.** This means that your application code doesn't have to hold on to the *DataNode* pointer; it can be retrieved by name whenever needed. Alternatively, you could just keep track of the *DataNode* pointer and wouldn't need this method.

Do not delete the *DataNode* pointer that is returned. The *Client::Bus* object maintains the lifetime and will handle deletion when appropriate.

Note that this method can only retrieve a *DataNode* that was created with the same particular instance of a *Client::Bus*.

get_or_create_data_node

This method is double-duty – if the node has already been created on this instance of *Client::Bus*, it will behave the same as *get_data_node*. If the node has not been created, a request to register the node with the Bus will be sent to the server. The server will verify that the name of the *DataNode* is not already in use on the Bus and will accept or reject the response accordingly. If accepted, the client bus will create and return the new *DataNode*.

remove_data_node

This method is for removing a data node from the *Client::Bus*. This will de-register the node from the bus and free up any related resources. Any *DataNode* pointers for the node will become invalid as soon as the call returns.

Once deregistered, the name for the node would become available for any *Client::Bus* instance to use.

The sample code in **Figure 10** shows and explains the dos and don'ts of the registration methods in a variety of situations.

```
#include <Client/Bus.hpp>

// 3 Client::Bus instances: two for Bus_1, 1 for Bus_2
NosEngine::Client::Bus bus_1_client_1(“copy://”, “Bus_1”);
NosEngine::Client::Bus bus_1_client_2(“copy://”, “Bus_1”);
NosEngine::Client::Bus bus_2_client(“copy://”, “Bus_2”);

NosEngine::Client::DataNode *bus1_node1;

bus_1_client_1.get_data_node(“Node_1”);       //ERROR: node does not exist yet
bus_1_client_1_.get_or_create_data_node(“Node_1”);  //Success
bus1_node1 = bus_1_client_1.get_data_node(“Node_1”); //Success

bus_1_client_2.get_data_node(“Node_1”);          //ERROR:  node  not  created  with  this
instance
bus_1_client_2.get_or_create_data_node(“Node_1”);  //ERROR: node already registered on
the bus

bus_2_client.get_or_create_data_node(“Node_1”);   //Success – different bus

bus_1_client_2.remove_data_node(“Node_1”);        //ERROR:  node  not  created  with  this
instance
bus_1_client_1.remove_data_node(“Node_1”):       //Success; bus1_node1 pointer is now
invalid
```

Figure 10: Data Node Registration

3.3 Sending and Receiving Messages

In the previous section, you learned how to create data nodes on the bus. The next big piece of the puzzle is getting the nodes to exchange messages between each other. Exchanging messages is performed with send and receive operations on a *DataNode* object. Let's take a quick look at the basics for sending and receiving.

3.3.1 Sending

There are multiple send operations available, but they all have a common method signature with three parameters: the destination, length of data to send, and a pointer to the data to send.

Destination

The destination argument is just the name of the node that should receive the message.

Data Length

Specify the length of data you wish to send.

Data Pointer

A pointer to your data. Obviously this must have enough space to satisfy the value in the data length parameter (unless you enjoy debugging peculiar bugs and seg. faults).

All send methods are simple function calls on the *DataNode* – it is not an event- or callback-based model. While a send method is running, your data must be left undisturbed. Once the send call returns, you are free to do as you wish with your data.

3.3.2 Receiving

Receiving messages is a quite a bit different compared to sending messages. Instead of calling a receive method, you will register a receive callback. Whenever a message arrives for the node, the registered callback will be run.

The receive callback has a very simple signature:

```
void receive_callback(NosEngine::Common::Message);
```

Simple. You just write a receive_callback with whatever logic in the body you need to read and interpret the message!

This is the first mention of the *Common::Message* object (hereinafter referred to as *Message)*, so let's explore. The *Message* class contains the payload data (the data sent from another node), and some contextual information about the message. This contextual information is needed by NOS Engine for routing and processing, but is also relevant to the receiver. More on that later!

3.3.3 Example #2: Basic Send-Receive

The following code demonstrates a message being sent and received between two nodes.

```
#include <Client/Bus.hpp>
#include <Client/DataNode.hpp>
#include <iostream>

NosEngine::Client::Bus bus_1 (“copy://”, “Bus_1”);

NosEngine::Client::DataNode *sender, *receiver;

sender  = bus_1.get_or_create_data_node(“Sender”);
receiver = bus_1.get_or_create_data_node(“Receiver”);

receiver->set_message_recevied_callback([&receiver](Common::Message)
     {
       std::cout << “Received a message!” << std::endl;
     });

sender->send_message(“Receiver”, 5, “Hello”);
```

3.3.4 Extracting Data from a Message

Payload data is stored within the buffer part of a Message. The buffer is a simple class that provides some convenience operations for handling a data array. In addition to the payload data, the buffer may contain additional header information that is used for logical processing in NOS Engine. To retrieve the payload data, a constant value, *USER_DATA_START*, is defined in *Protocol.hpp*. This value is simply the offset to the user data within the buffer. The buffer object has a property *len* that indicates how many characters are in the buffer. Therefore, the length of the payload data can be calculated as *len – USER_DATA_START*.

Here is a simple illustration of accessing the user data:

```
#include <Client/Bus.hpp>
#include <Client/DataNode.hpp>
#include <Common/Protocol.hpp>

using namespace NosEngine;
Client::Bus bus_1 (“copy://”, “Bus_1”);
Client::DataNode *sender, *receiver;
```

```
sender  = bus_1.get_or_create_data_node("Sender");
receiver = bus_1.get_or_create_data_node("Receiver");

receiver->set_message_recevied_callback([&receiver](Common::Message msg)
     {
       int length;
       length = msg.buffer.len – Common::Protocol::USER_DATA_START; // length=1
       // msg.buffer.data[USER_DATA_START] == 'A'
     });

sender->send_message("Receiver", 1, "A"); // sending the single letter, A
```

Another option to access the payload data is to use the utility class from *DataBufferOverlay*. This class is a simply wrapper that takes the buffer from a message, hides any header data in the buffer, and only presents the payload data.

Here is the previous example shown with the *DataBufferOverlay* class:

```
#include <Client/Bus.hpp>
#include <Client/DataNode.hpp>
#include <Common/Protocol.hpp>
#include <Common/BufferOverlay.hpp>

using namespace NosEngine;
Client::Bus bus_1 ("copy://", "Bus_1");
Client::DataNode *sender, *receiver;

sender  = bus_1.get_or_create_data_node("Sender");
receiver = bus_1.get_or_create_data_node("Receiver");

receiver->set_message_recevied_callback([&receiver](Common::Message msg)
     {
       Common::DataBufferOverlay overlay(msg.buffer);
       // overlay.len == 1
       // overlay.data[0] == 'A'
     });

sender->send_message("Receiver", 1, "A"); // sending the single letter, A
```

3.4 Messaging Patterns

NOS Engine includes support for common message patterns to save some effort on behalf of the user. The next few sections will describe each of the available patterns in detail.

3.4.1 Non-Confirmed Messaging

Non-confirmed messaging is the capability to send a message to another node with best-effort delivery - the sending node does not receive acknowledgment of successful delivery from the receiving end. This method of delivery provides the highest throughput but does not guarantee delivery.

It should be noted that although NOS Engine does not send any acknowledgements, the transports used underneath can have their own reliable-delivery protocols (e.g., TCP). This may provide comfort but should not be taken as a guarantee that a non-confirmed message will arrive successfully.

Figure 11 shows the data flow of a non-confirmed message moving through NOS Engine.

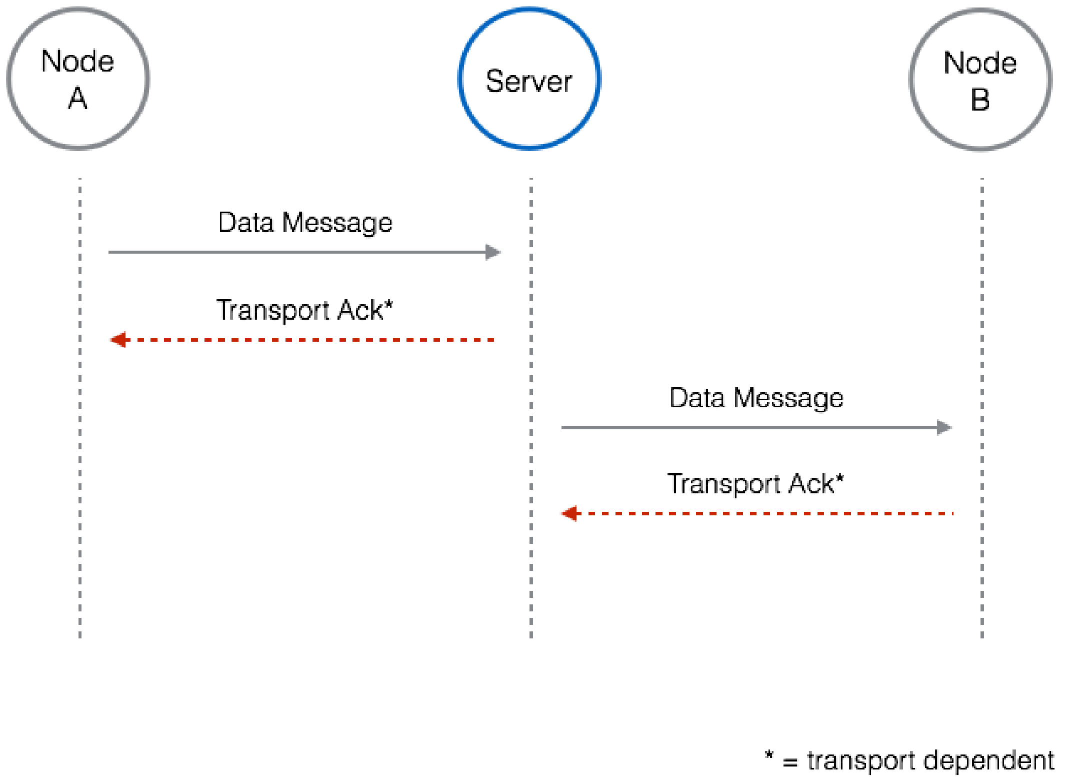

Figure 11: Non-Confirmed Data Flow

Non-confirmed messaging is performed using the synchronous *DataNode::send_message(…)* method.

3.4.2 Confirmed Messaging

Confirmed messaging is the complement to non-confirmed – the destination will send an acknowledgement after it has received the message. This should be used for delivery of critical messages that do not require a response.

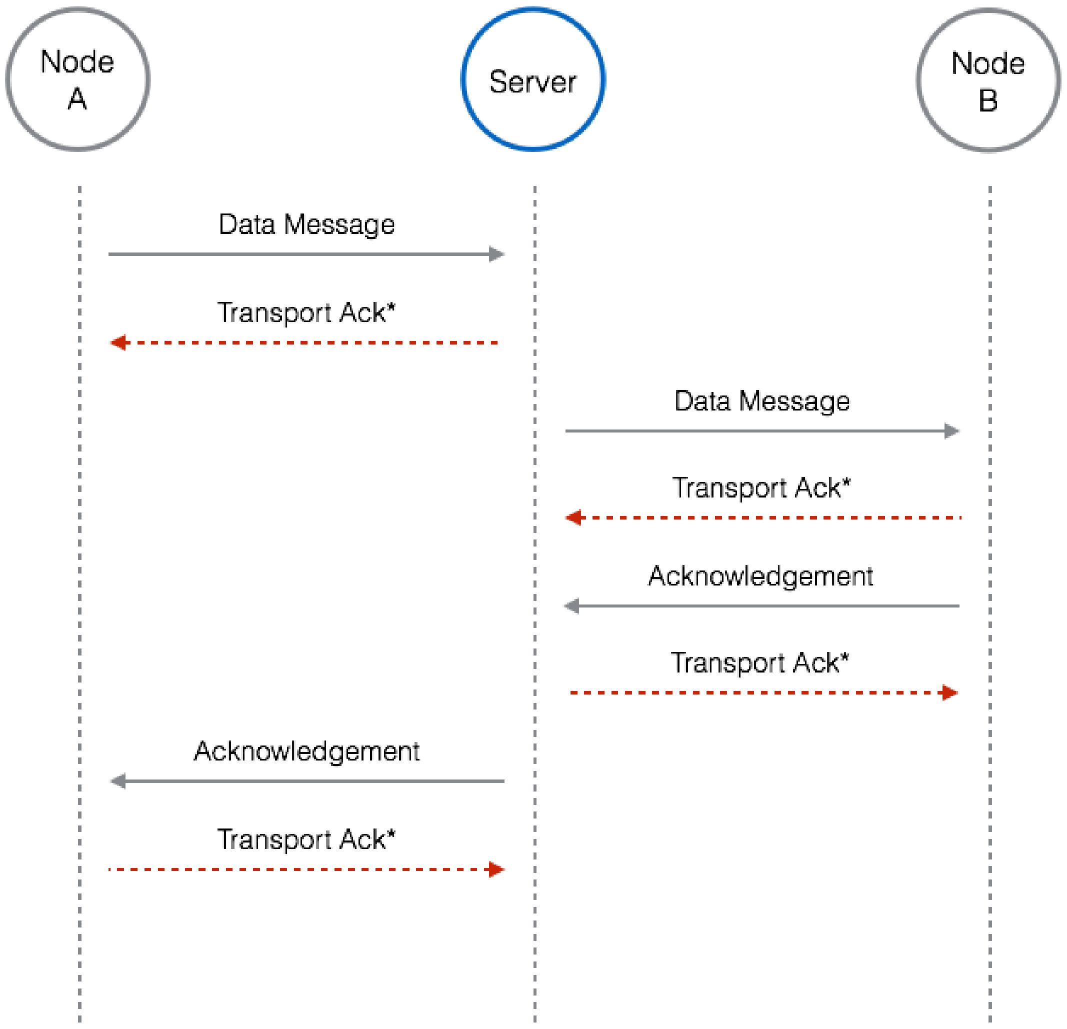

Figure 12: Confirmed Data Flow

Confirmed messaging is performed using the synchronous or asynchronous *DataNode:: send_confirmed_message(…)* method. The synchronous method will block until the acknowledgement is received; the asynchronous method requires a callback parameter that will be run when the acknowledgement is received.

3.4.3 Send-Reply Messaging

Send-reply functions similarly to confirmed messaging, except that the reply from the destination is not an empty acknowledgement, but instead some meaningful data in response the original message.

Starting a send-reply operation is done with the *DataNode::send_request_message(…)* method. The receiving end has special responsibility to send back a reply. First, the receiving end will check the message to see if the *duplex* flag has been set to true. This indicates that the source is expecting a reply. Once the receiving end has generated its response, it uses the *DataNode::send_reply_message*(…) call to respond.

Figure 13 shows source code a basic send-reply operation.

```
#include <Client/Bus.hpp>
#include <Client/DataNode.hpp>
#include <iostream>
```

```
NosEngine::Client::Bus bus_1 (“copy://”, “Bus_1”);

NosEngine::Client::DataNode *player_one, *player_two;

player_one = bus_1.get_or_create_data_node(“Player_1”);
player_two = bus_1.get_or_create_data_node(“Player_2”);

player_two->set_message_recevied_callback([&player_two](Common::Message message) {
  // normally should verify the message (the source, is duplex, contents)
  player_two->send_reply_message(message, “Player_1”, 4, “Pong”);
});

//this call will block until the response is received
NosEngine::Common::Message response_message =
  std::move(player_one->send_request_message(“Player_2”, 4, “Ping”));

// the response_message contains “Pong”
```

Figure 13: Example Send-Reply Messaging

3.4.4 Multicast

All messaging so far has been a 1-to-1 operation, but NOS Engine also supports multicast – sending a message to multiple destinations. Currently, multicast support is limited to broadcast operations (sending a message to all *DataNodes* on the bus).

Multicast support is available for both non-confirmed and confirmed messaging (no such luck for send-reply).

To send a broadcast message, use the *DataNode::send_multicast_message(…)* or *DataNode::send_multicast_confirmed_message(…)* methods with “*” as the destinations string.

3.5 Interceptors

As mentioned previously, interception allows one to manipulate messages as they move through the system. This is accomplished through the use of a specialized node known as an *InterceptorNode.* Each *InterceptorNode* is made to target communication either incoming or outgoing from a specific *DataNode*. This means that an interceptor can either manipulate messages that are to be received by a node (independent of the source), or any messages sent from a specific node (independent of the destination).

3.5.1 Registration

InterceptorNode registration is done similarly to *DataNode* registration, but now with three parameters: the name of the interceptor, the name of the target node, and the direction of interest (incoming or outgoing).

The name of the interceptor must be unique from any other node on the bus.

An *InterceptorNode* can target any *DataNode* on the same bus – it needs to only know the name of the target *DataNode*, and that node must exist.

Here's a simple example registering an interceptor for incoming messages:

```
#include <Client/Bus.hpp>
#include <Client/DataNode.hpp>
#include <Client/InterceptorNode.hpp>

NosEngine::Client::Bus bus_1 (“copy://”, “Bus_1”);
```

```
NosEngine::Client::DataNode *node_1;
NosEngine::Client::InterceptorNode *interceptor;

node_1 = bus_1.get_or_create_data_node("Node_1");
interceptor = bus_1.get_or_create_interceptor("Interceptor", "Node_1",
         Common::BusProtocol::InterceptorDirection::Incoming);
```

3.5.2 *Interceptor Actions*

To perform any action, you'll first need to register a callback for when a message is intercepted. This is done using the *InterceptorNode::set_interceptor_function(…)* method. Without a method set, the interceptor will pass every message unaltered by default.

The callback method has the same signature as a *DataNode* receive message callback. Within your callback, you can view the message as needed.

```
.
.
.
interceptor = bus_1.get_or_create_interceptor("Interceptor", "Node_1",
         Common::BusProtocol::InterceptorDirection::Incoming);

interceptor->set_interceptor_function([&interceptor](NosEngine::Common::Message
message) {
  //inspect the message

  //call an interceptor action (pass, block, modify, mimic)

});
```

The last step is to choose the interceptor action to perform:

- **Pass**: Allow the message to propagate as normal
- **Block**: Discard the message from the system
- **Modify**: allow the message to propagate but with altered data
- **Mimic:** impersonate the destination node (only for send-reply messaging)

3.5.2.1 Pass

Passing messages does not alter the communication in any way. An interceptor that operators exclusively with pass operations may be used like a "real-time" traffic monitor.

Figure 14 shows the message flow when passing a message through the send side. **Figure 15** shows the message flow when the interception occurs on the reply side (in case of a send-reply exchange).

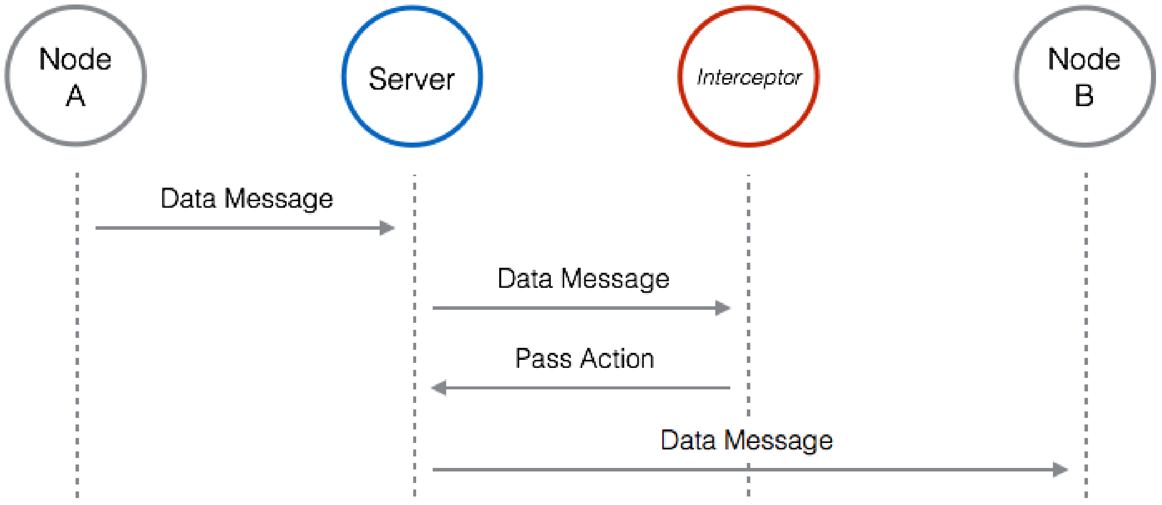

Figure 14: Send Chain Interceptor Pass

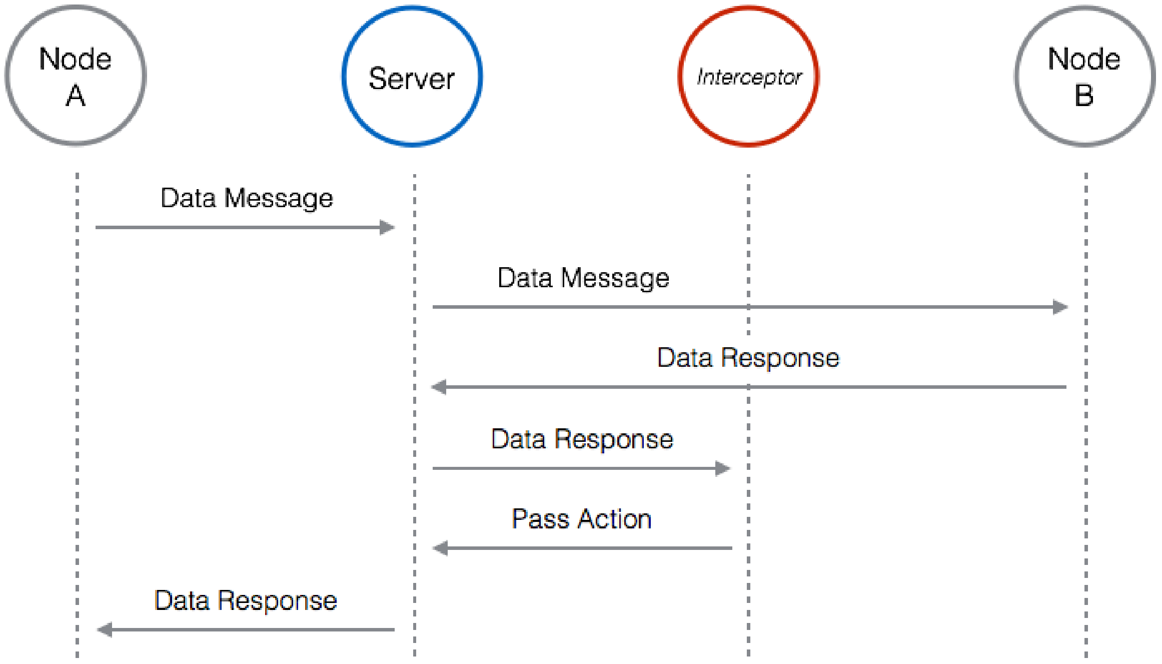

Figure 15: Receive Chain Interceptor Pass

3.5.2.2 Block

A block operation allows an interceptor to prevent delivery of a message. In the case of a confirmed message or send-reply exchange, the initial sender will receive an error as if the destination did not exist on the bus. This is necessary to close out the transaction and prevent any 'zombie' transactions, which could lead to deadlock.

Figure 16 shows the message flow when blocking a message during the send side. **Figure 17** shows the message flow when the interception occurs on the reply side (in case of a send-reply exchange).

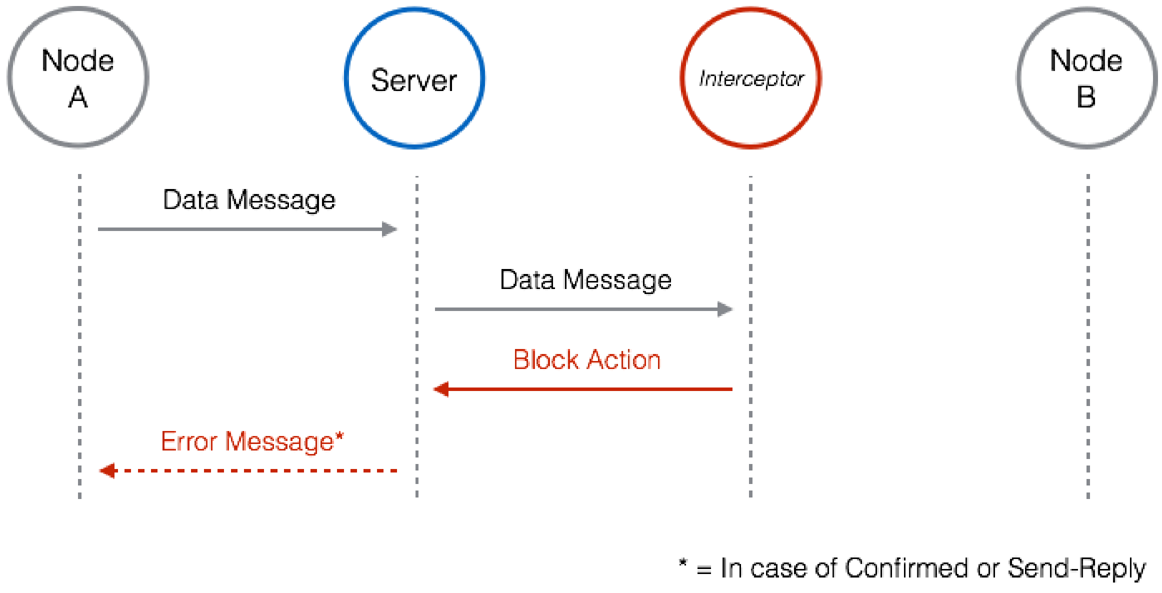

Figure 16: Send Chain Interceptor Block

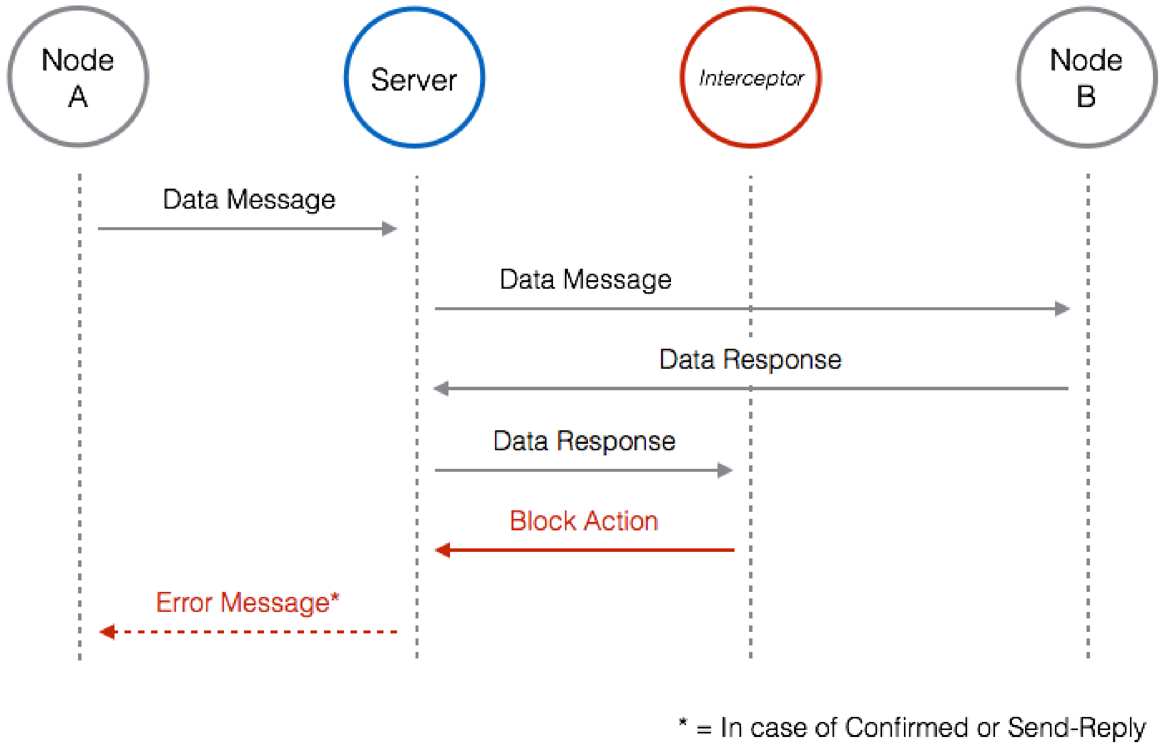

Figure 17: Reply Chain Interceptor Block

3.5.2.3 Modify

A modify operation allows the interceptor to replace/alter the data before it arrives at the intended target.

Figure 18 shows the modify operation on the send-side. **Error! Reference source not found.Figure 19** shows the modify operation on the reply-side.

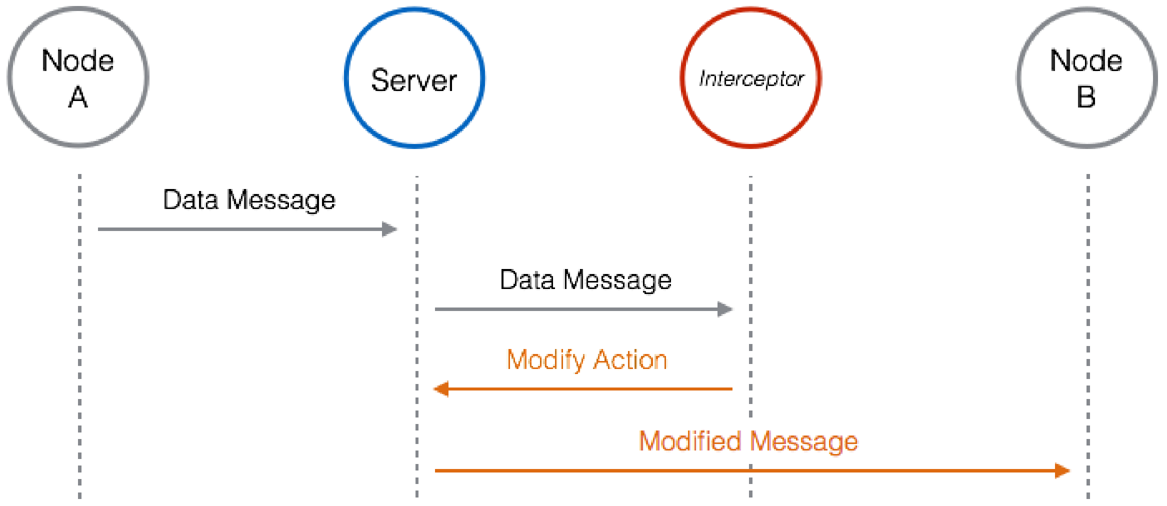

Figure 18: Send Chain Interceptor Modify

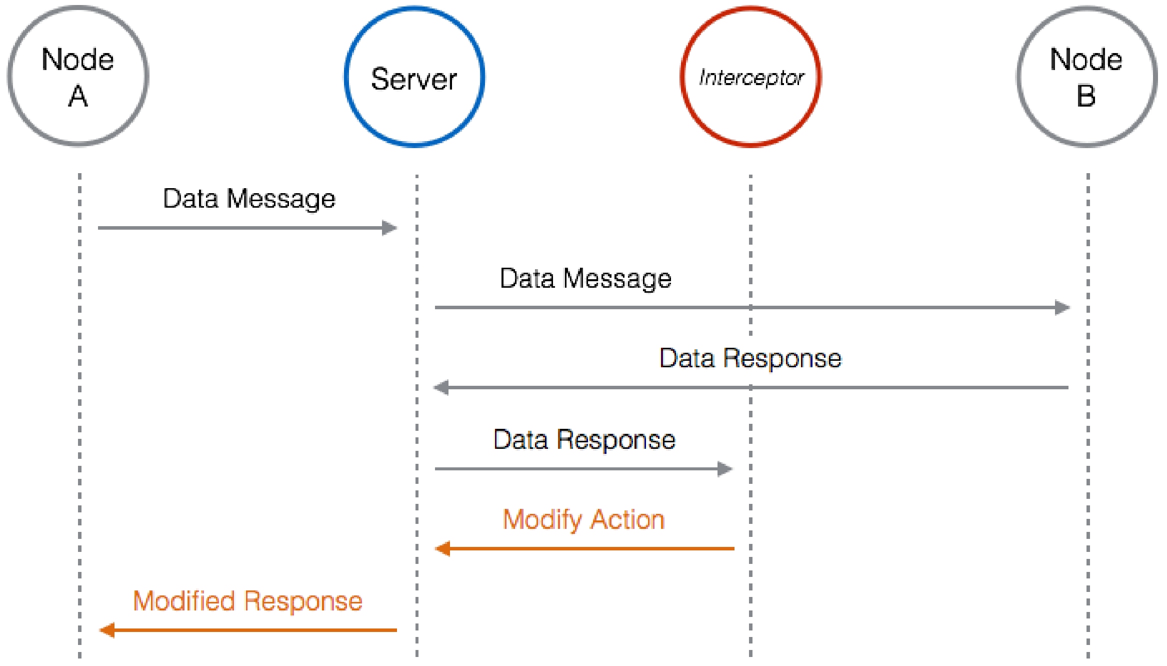

Figure 19: Reply Chain Interceptor Modify

3.5.2.4 Mimic

The mimic operation is a special operation that allows an interceptor to masquerade as the intended receiver. The mimic operation can only be performed on request/reply messages when the send-side message is captured. The result is essentially a combination of block and modify operations – the receiver does not receive the message, and the interceptor then sends its own reply back to the sender. **Figure 20** shows the flow of a mimic operation.

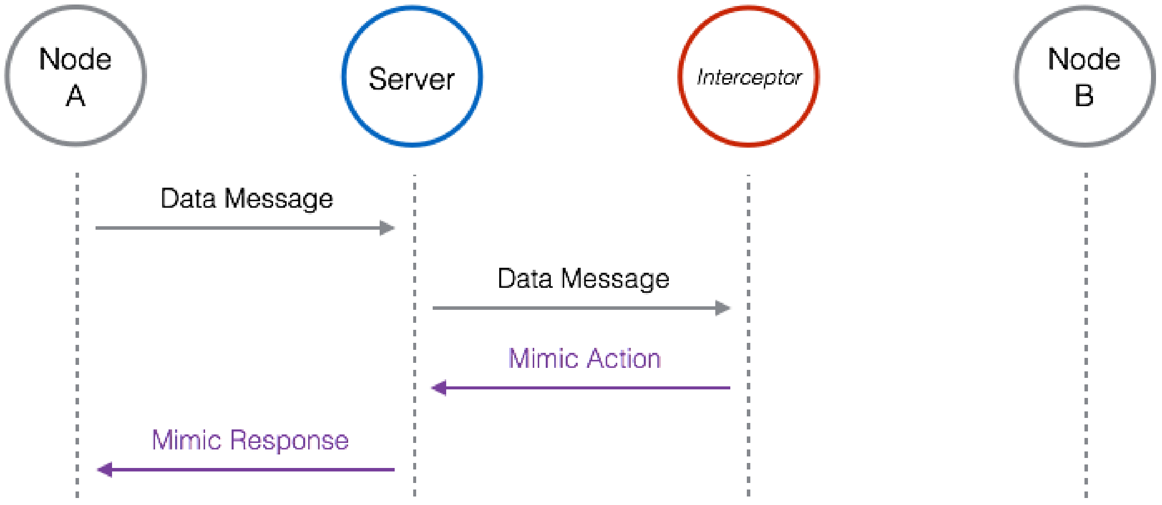

Figure 20: Interceptor Mimic

3.6 Time Distribution

NOS Engine provides a simple time distribution feature. The true power of time distribution is not just broadcasting a time value, but rather the ability to synchronize activity!

Time distribution is handled using two different objects: *TimeSender* and *TimeClient*. On any given bus, there can be a single *TimeSender* and unlimited *TimeClients*. The *TimeSender* has the sole responsibility of sending a time value. Each time a new time value is sent, it is distributed to each and every *TimeClient* on the same bus. Synchronization is possible since the *TimeSender* will wait until each *TimeClient* has received the tick and performed any activity. This allows activity to be synchronized between multiple clients when each may take a different amount of "wall-time" to complete. **Figure 21** illustrates the behavior between the *TimeSender* and *TimeClients* when a time value is sent.

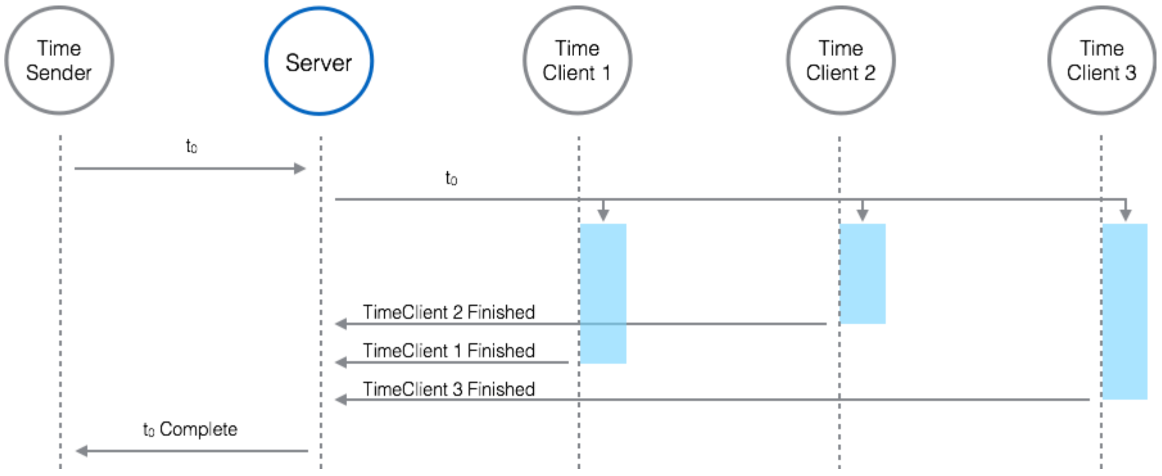

Figure 21: Time Distribution

3.6.1 Time Sender

TimeSender registration is done similarly to *DataNode* registration. The only parameter needed is a name, which must be unique from any other node on the bus. And of course, there can only be one *TimeSender* at a time.

Once registered, the *TimeSender* object is used to distribute time using the *TimeSender::send_time(…)* call. This call has a single parameter, *SimTime (*currently defined as a 64-bit signed integer). Sorry, there is no support for floating-precision numbers; get creative!

The following example shows how to register a time sender and send a time message.

```
#include <Client/Bus.hpp>
#include <Client/TimeSender.hpp>

NosEngine::Client::Bus bus_1 (“copy://”, “Bus_1”);

NosEngine::Client::TimeSender *time_sender;

time_sender = bus_1.get_or_create_time_sender(“Time_Sender”);
time_sender    =    bus_1.get_or_create_time_sender(“Time_Sender2”);    //ERROR:    sender
registered already


time_sender->send_time(0); // send time 0 to all clients, blocks til all clients done
```

3.6.2 Time Clients

TimeClient registration is just as easy as the rest – the only parameter is a name, which as always must be unique from any other node on the bus. Once a *TimeClient* is registered, a ‘tick’ callback has to be assigned. This callback will be run each time a time value is received. Any work that should be performed during the ‘frame’ should be completed during the callback (either directly within or using condition variables to control other parts of the application). Once the callback returns, the *TimeSender* will have permission to send the next tick.

And this is how it’s done:

```
#include <Client/Bus.hpp>
#include <Client/TimeClient.hpp>

NosEngine::Client::Bus bus_1 (“copy://”, “Bus_1”);

NosEngine::Client::TimeClient *time_client;

time_client = bus_1.get_or_create_time_client(“Time_Client”);

time_client->set_tick_callback([](NosEngine::Common::SimTime time)
{
  //perform any work as necessary

  //return when work is complete
});
```

4 MIL-STD-1553

4.1 Overview

The purpose of this section is to familiarize users and developers with the inner workings and usage of the MS1553 plug-in for NOS Engine. The plug-in provides both client and server side components inside of a single library. It is designed to provide a clean, easy to use interface that mimics hardware interactions without being too verbose.

4.2 Terms

Bus Controller	Primary data node on the bus. Initiates all traffic
Bus Monitor	Special node on a MS 1553 bus that is a passive observer of all traffic
Remote Terminal	An endpoint node on the bus capable of sending and receiving data via subaddresses. Identified as a numerical value 0-30 (hardware address) or 1 -31 (logical address)
Subaddress	A unique address within a Remote Terminal for the purpose of sending or receiving data.
Command Word	16 bit data word that describes the intentions of the current transaction.
Status Word	Response from Remote Terminals for a given Command Word
Receive Request	Receive data request from the Remote Terminal perspective for a subaddress.
Transmit Request	Request that an RT transmit a given amount of data for a subaddress

4.3 Usage

The MS 1553 plug in is separated into two distinct pieces. The client side is designed to be the user facing portion of code. The server side is designed to be loaded into the base engine server using the plug in system.

4.3.1 Client

When using the client portion of the MS 1553 plug-in it is required that the server side plug-in is loaded. This is due to the fact that the plugin makes certain assumptions of the server's knowledge of how to process messages that are above the basic capabilities of a plain server side bus. Most interactions within the server are simply extensions of the base layer with the notable exceptions of interception creation and Remote terminal registration.

Currently if an application wishes to contain both a Bus Controller and Remote Terminal hub, two unique connections must be made to the server. This is due to the fact that currently transport sharing is not supported in the foundation layer and no factory method currently exists to generate the objects.

4.3.1.1 Bus Controller

The Bus Controller ("BC") is the object on the network used to initiate all traffic on the bus. Currently, the creation of the BC object on the bus will result in the creation of a data node called "bc". At current time, this node name is hardcoded into the system.

The BC may interact with the system using either synchronous or asynchronous methods.

4.3.1.2 Remote Terminals

Remote Terminals ("RT"s) are not directly created by a user on the bus. RTs are instead attached to a user created Remote Terminal Hub. The hub is responsible for distributing MS1553 messages to user designated callbacks as needed. Each hub may represent multiple RT's at any given time (See **Figure 22**). They may be added and removed from the system at runtime as the user needs. Users are then provided with a representation of an RT to use in order to respond to the message.

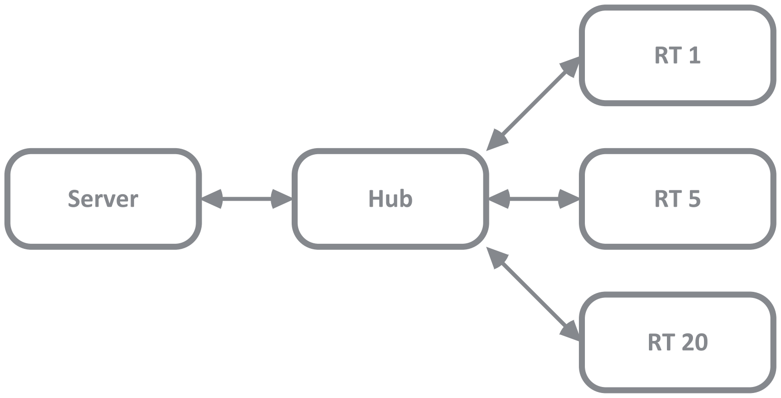

Figure 22: RT communication through the Hub

All messages are received for a given RT asynchronously through the provided callback during RT registration. The callback is used to represent the entirety of an RT, meaning that both the receiving and transmission of all data for all subaddresses is expected to be handled.

In the event of a broadcast message on the bus, all provided callbacks will be called. NOTE: Each callback is required to acknowledge the broadcast message before bus communication can proceed.

4.3.2 Protocol

The plugin library provides users and developers with several convenience methods for inserting or extracting protocol information from a given message. NOTE: Currently RT to RT transmissions are not supported by the protocol.

"User space" protocol elements of data messages are rather simple. The following table describes the components of the data protocol.

Scenario	BC -> RT Message	RT -> BC Response	Notes
BC send data to RT (Receive Request)	[Command Word] [Word Count] [Data]	[Error Code] [Status Word]	
BC request data from RT (Transmit Request)	[Command Word]	[Error Code] [Status Word] [Word Count] [Data]	
RT Ignores transaction (Either receive or transmit)	[Command Word] [Optional Data]	[Error Code] [Status Word]	An automatic Status Word is sent for the server to interpret the message. It is ignored by the BC
BC Broadcast Data	[Command Word] [Optional Data]	[Error Code] [Status Word]	The Error Code portion of the message is always returned as "No Error". Each Hub will only respond after all RTs have acknowledged the message. Only a single response sent to the BC.

4.3.3 Server

The requirements for MS1553 communications within the server are very minimal. This is mostly due to the fact that data the majority of work performed on a 1553 bus is simply data passing. Interactions between nodes are not complex and therefore the foundation layer of engine can largely be relied upon to perform the work.

The server side implementation of the plugin provides a specialized 1553 bus class and router for use. The work performed by these classes is minimal and only performed in a small number of cases. The most common of these special operations is the association of a RT address with an existing data node. A special protocol command is processed to report the association.

The server side work is mostly directing messages to a given RT on an associated node. This allows the 1553 layer to be more dynamic than the traditional foundation layer. For example, any time a RT is successfully associated with a data node all interceptors for that RT are added to the data node.

4.3.4 Interceptors

Currently no special implementation exists for interceptors for 1553 communication. All interception can be performed utilizing the core interceptor logic and the protocol helpers for 1553. In order to enhance this capability and to reduce the burden of clients, interceptors may request to intercept specific RTs if the "target" of interception is generated with the "create_rt_name" protocol utility function. This also allows interceptors to dynamically change targets in the event that the hub representing an RT changes during runtime.

Server side interceptors utilize a message filtering mechanism when intercepting specific RTs on a hub. The message filter is designed to be generic and only allow messages which match the requested RT or broadcast address to be sent to the interceptor.

4.4 Unsupported Features

Currently the MS 1553 plug in does not support all features of the MS 1553 protocol. This is either due to time constraints or lack of necessity within the system.

- RT to RT transmission - Currently RT to RT transmissions are have no planned implementation path. The feature is not widely (if ever) used on most systems.
- Bus Monitor - No official bus monitor exists within the plug in. Users may utilize incoming and outgoing interceptors on the "bc" node to achieve the same outcome. This feature may be added as needed.
- Dynamic Bus Control - The MS 1553 protocol allows for the BC to give up control to another node. This feature is not supported due to the lack of use within most systems.
- MS 1553 queries - The plug in does not support any protocol specific queries. Planned queries include registered RTs and current RT associations.
- Single transport connection for RTs and BC - Currently if an application wishes to be both a BC and RT on the bus, two separate transports must be connected to the server.
- Server side data logging - No current specialized logging exists within the server. This will be added at a future date.

5 SpaceWire

5.1 Overview

The purpose of this document is to familiarize users and developers with the inner workings and usage of the SpaceWire plug-in for NOS Engine. The plug-in provides both client and server side components inside of a single library. It is designed to provide a clean, easy to use interface that mimics hardware interactions without being too verbose.

5.2 Terms

SpaceWire Node	Represents the interface between an application and a SpaceWire Network and is the source or destination of SpaceWire Packets.
SpaceWire Router	Represents a collection of SpaceWire Nodes which are linked to a routing switch via input/output ports.
SpaceWire Link	The connection between two Nodes or a Node and a Router.
SpaceWire Network	Represents the links between a collection of SpaceWire Nodes and Routers.
Addresses	A numeric value, between 0 and 254, that represents a port or logical address.
Configuration Port Addresses	Internal configuration port, of a SpaceWire Router, that is used to access configuration and status information. (Address range: 0)
Physical Port Addresses	Physical output ports of a router. (Address range: 1-31)
Logical Addresses	Mapped onto physical output ports. (Address range: 32-254)
SpaceWire Packet	Contains a destination address, Cargo, and an End of Packet Marker.
Addressing Type	The type of addressing (Path or Logical) used for the destination address of a SpaceWire Packet.
Path Addressing	A type of addressing that consists of a series of physical output ports that are used to route a packet to its destination.
Logical Addressing	A type of addressing that maps a logical address to a physical output port of a router. The mapping is defined in the routing tables stored in each router.
Cargo	The data contained in a packet that is to be transferred from source to destination.
End of Packet (EOP) Marker	Marks the end of a packet. There are two possible markers: EOP and EEP. The EOP marker indicates the normal end of a packet, while the EEP marker indicates that an error occurred within the packet.

5.3 Usage

5.3.1 Client

When using the client portion of the SpaceWire plug-in it is required that the server side plug-in is loaded. This is due to the fact that the plugin makes certain assumptions of the server's knowledge of how to process messages that are above the basic capabilities of a plain server side bus. Most interactions within the server are simply extensions of the base layer.

5.3.1.1 SpaceWire Network

A SpaceWire Network currently may contain one or more unconnected SpaceWire Routers, where each router may have one or more SpaceWire Nodes connected to their input/output ports (See **Figure 23**).

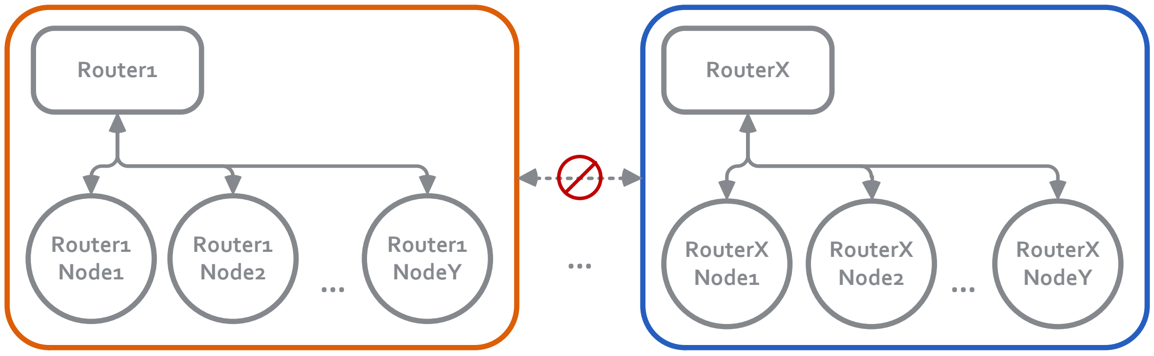

Figure 23: SpaceWire Network without routing

NOTE: Currently point-to-point links are not supported, so the network must contain at least one router.

NOTE: Routing of packets is not currently supported; therefore, routers cannot be connected and a node can only send a packet to another node that is connected to the same router.

5.3.1.2 SpaceWire Node

SpaceWire Nodes are added to a user created SpaceWire Network. To add a node, to an existing network, a router name and port must be provided.

All packets are sent or received from/to a node asynchronously. Packets are received through a user provided callback function. Request/reply packets are not currently supported, but could be added in a future update, if desired.

5.3.2 Protocol

The plug-in library provides users and developers with several convenience methods for inserting or extracting protocol information from a given message.

"User space" protocol elements of data messages are rather simple. The following table describes the components of the data protocol.

Protocol Header		
Addressing Type	EOP Marker	Cargo

NOTE: The destination address will be stored in the base layer message's primary header (once routing is implemented), so it's not included in the protocol header.

NOTE: Due to how messages are handled in NOS the EOP marker does not need to actually follow the packet cargo and is therefore included in the protocol header for simplicity.

5.3.3 Server

5.3.3.1 SpaceWire Network Bus (Protocol Plug-in)

The current protocol plugin simply provides the same functionality that is provided by the base layer's server-side Bus. It is a placeholder until additional SpaceWire functionality is implemented (ie routing, etc.).

5.3.4 Interceptors

There is currently no specialized interceptor implementation for SpaceWire. Interception can be done by using base layer interceptors and the protocol helpers for SpaceWire.

5.4 Unsupported Features

Currently the SpaceWire plug-in does not support all features of the SpaceWire protocol. This is either due to time constraints or lack of necessity within the system.

- SpaceWire Links (Physical level, Signal level, Character level, Exchange level)
- Fault tolerant links
- Error recovery
- Point-to-Point links (ie two directly connected SpaceWire Nodes)
- Cascading
- Multiple interconnected SpaceWire Routers
- Routing and routing tables
- Arbitration
- Group adaptive routing
- Wormhole routing
- Header deletion
- Virtual channels
- Router configuration/status via configuration port
- Path Addressing with a destination address that contains more than one Physical Port Address
- Logical Addressing
- Regional Logical Addressing
- Interval labeling
- Time-codes
- Packet distribution (broadcast and multicast)
- SpaceWire Protocols

6 I2C

6.1 Overview

The purpose of this section is to familiarize users and developers with the inner workings and usage of the I2C plug-in for NOS Engine. The plug-in is designed to provide a clean, easy to use interface that mimics hardware interactions without being too verbose. The I2C client API will simply provide a means of initialization and registering a callback function to handle I2C master requests.

6.2 Usage

• i2c_init	–	NOS engine specific initialization
• i2c_read	–	Read I2C data from bus
• i2c_write	–	Write I2C data to the bus
• i2c_transaction	–	Simplification of I2C write/read transactions

7 UART

7.1 Overview

The purpose of this section is to familiarize users and developers with the inner workings and usage of the UART plug-in for NOS Engine. It is designed to provide a clean, easy to use interface that mimics hardware interactions without being too verbose. The UART plugin API will internally wrap existing asynchronous communication functionality provided by NOS engine. It will also provide a means of point-to-point communications, throwing the appropriate errors if two endpoints are already connected (UART is in use). The plugin API will also allow the UART to be opened/closed using an integer port number, rather than NOS engine data nodes.

7.2 Usage

• uart_init	–	NOS engine specific initialization
• uart_open	–	Open specific UART port number
• uart_close	–	Close UART port number
• uart_read	–	Read data from the opened UART port, or via callback mechanism
• uart_write	–	Write data to the opened UART port