Technical Expertise Support Services (TESS)

TO9 NOS Engine User's Manual

Submitted by:

TMC Technologies

January 20, 2017

Submitted to:

Independent Verification and Validation (IV&V)

Contract NNGG14SA05Z

Configuration Management Number TMC-NASA-TESS-15-019

Administrative Office
2050 Winners Drive
Fairmont, WV 26554
304.816.3600

Point of Contact
Scott Zemerick, PM
TMC Technologies
Phone: (681) 753-5230

Table of Contents

Table of Figures

Version History

Version #	Author	Revision Date	Peer Reviewer	Reason for Revision
0.2	Mark Pitts	08/08/2015		Initial document. For NOS Engine Phase 3 (Version 0.2)
1.0	Gary Carvell, Robert Brown, Tim Riley	01/07/2016		Updates for version 1.0 release
1.2	Tim Riley	05/03/2016		Updates for version 1.2 release
1.4	Tim Riley	01/18/2017		Updates for version 1.4 release

Reference Documents

Document Number	Document Name	Document Date	Source
	NOS Engine Design Document	January, 2017	Independent Test Capability Team, NASA IV&V, Fairmont, WV
	NOS Engine Requirements Document	January, 2017	Independent Test Capability Team, NASA IV&V, Fairmont, WV
	NOS Engine Road Map	January, 2017	Independent Test Capability Team, NASA IV&V, Fairmont, WV (Confluence)
	NOS Engine Test Plan	January, 2017	Independent Test Capability Team, NASA IV&V, Fairmont, WV
	NOS Engine Version Description Document	January, 2017	Independent Test Capability Team, NASA IV&V, Fairmont, WV

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 7 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, provides an overview of how to use NOS Engine in C code and provides multiple examples. Chapter 5, 6, 7, 8, and 9 are devoted to protocol layer implementations, MIL-STD-1553, SpaceWire, I2C, SPI, and UART, 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.

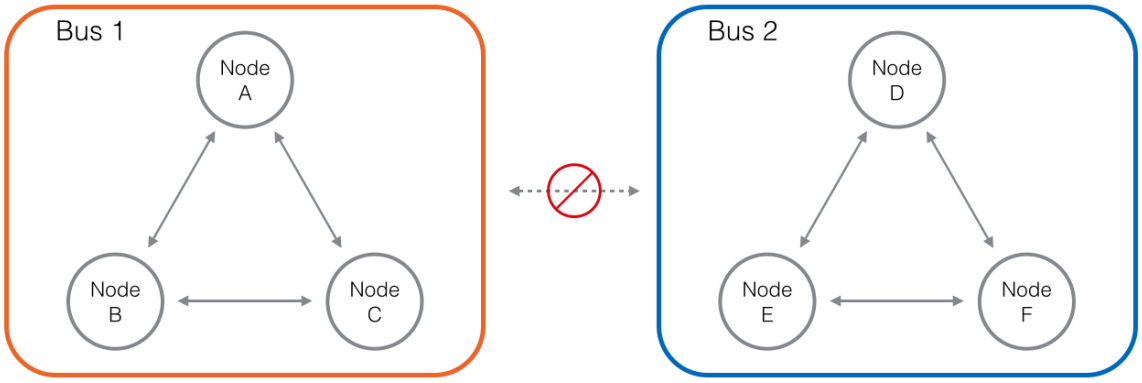

Figure 1: Bus & Node Topology

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.

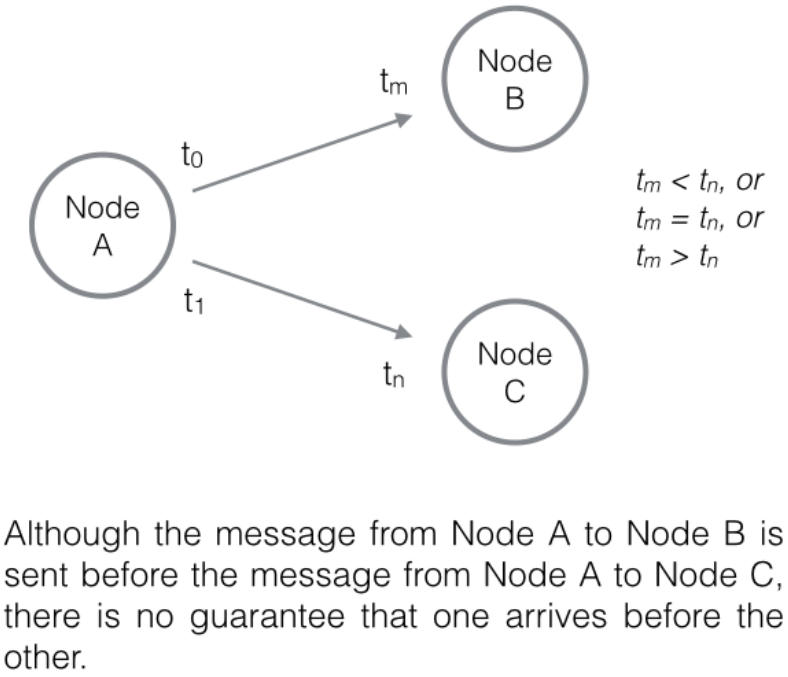

Figure 2: Asynchronous Messaging

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.

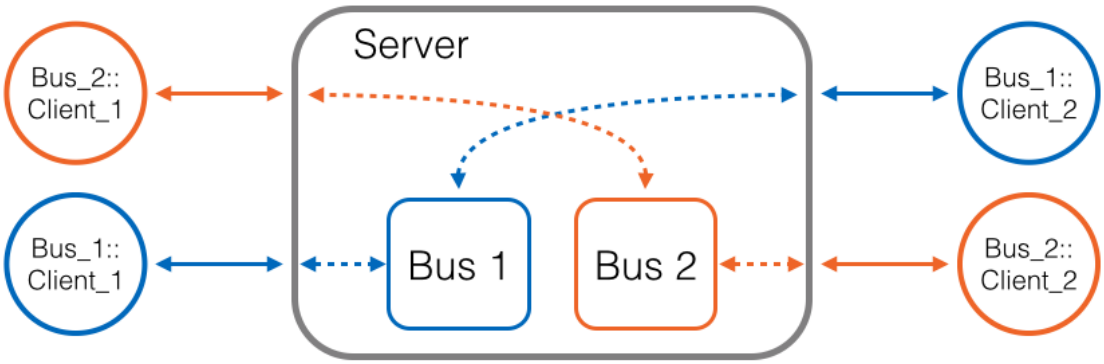

Figure 4: Server With Multiple Busses

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.X 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 3.1.3.6). 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("tcp://localhost:29556");

    // 3) keep server object around until time to shutdown
    while (true)
    {
        // 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, a simple example is shown in **Figure 6**.

```
#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("tcp://localhost:29556");

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

    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", "MyBus");

    // 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 MyBus. 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 can be different though). The server recognizes that multiple clients have connected and wish to participate on MyBus; 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 (asynchronously) for any transport. Once a client tries to connect (this happens internally when a Client::Bus object is created), the connection can be fully established and another listen performed.

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**.

3.1.3.4 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). The IPC transport can perform faster than the TCP transport, but it is limited to processes running on the same machine.

3.1.3.5 IPC Transport Connection String

```
ipc://<unique_name>:0
```

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.

Server ID

The server id, which is the last parameter, is always 0. The server id is used to differentiate between server and client URLs. Client URLs will always have a client id number greater than 0.

3.1.3.6 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!

3.1.3.7 Copy Transport Connection String

```
copy://<unique_name>:0
```

Transport Identifier

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

Unique Name

Each instance of the Copy transport requires a unique name.

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

Server ID

The server id, which is the last parameter, is always 0. The server id is used to differentiate between server and client URLs. Client URLs will always have a client id number greater than 0.

3.1.3.8 Transport hubs

Transport hubs are used to manage transport operations and objects. They provide a collection of worker threads which are used to perform server listens, client connects, and connection send/receive operations. NOTE: The underlying work hub (which manages the worker threads of the transport hub) can be retrieved from the transport hub and used to perform other types of work.

Transport hubs may be shared between server(s) and client object(s) (ie client-side objects that connect to a server). Sharing the hub will reduce the number of threads that are created by a process which uses NOS Engine. A source code example for sharing a transport hub is shown in **Figure 9**.

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

int main(int, char**)
{
    // 1) Create a transport hub with 5 threads
    NosEngine::Transport::TransportHub transport_hub(5);

    // 2) Create instance of server object
    NosEngine::Server::Server server(transport_hub);

    // 3) Add transport(s)
    server.add_transport("copy://example:0");

    // 4) application specific work (connecting clients to the server, any arbitrary work)
    NosEngine::Client::Bus bus_1(transport_hub, "copy://example:0", "Bus_1");
    NosEngine::Client::Bus bus_2(transport_hub, "copy://example:0", "Bus_2");
    // ...

    // create nodes with bus_X
    // ...

    // perform message exchange between nodes
    // ...

    return 0;
}
```

Figure 9: Shared TransportHub

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 10**. 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.

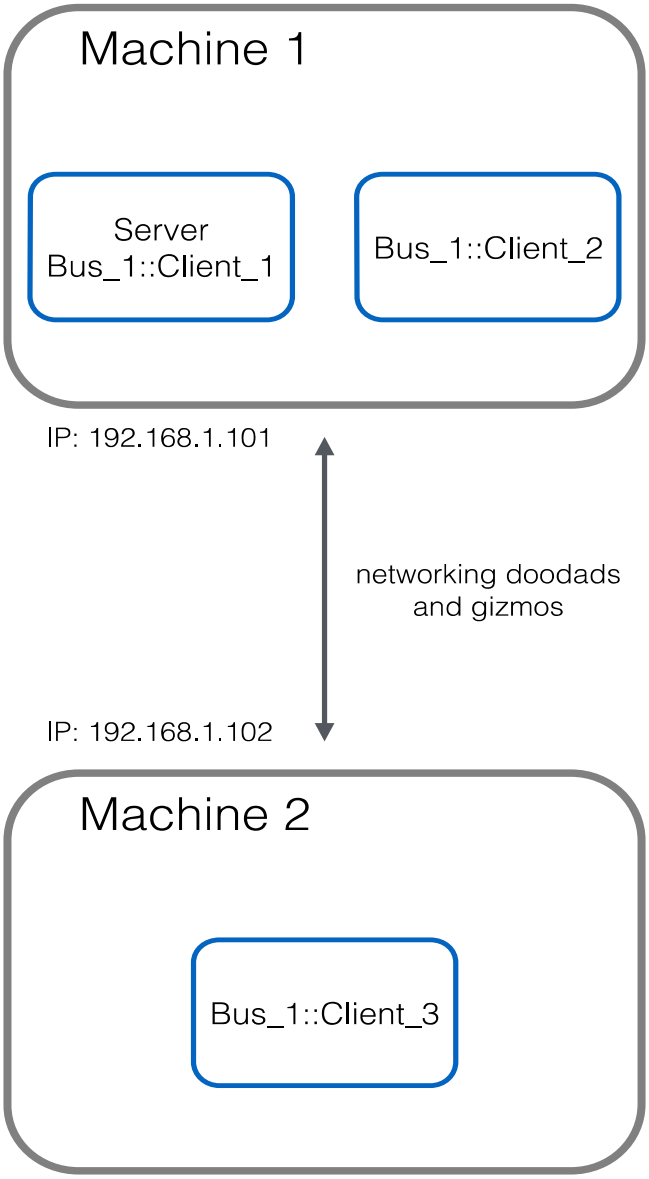

Figure 10: Example #1 Topology

```
// Machine 1 - Server and Client1

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

int main(int, char**)
{
    // 1) Create a transport hub with the default number of threads
    NosEngine::Transport::TransportHub transport_hub;

    // 2) Create instance of server object
    NosEngine::Server::Server server(transport_hub);

    // 3) Add transport(s)
    server.add_transport("copy://example:0");
```

```
    server.add_transport("ipc://example:0");
    server.add_transport("tcp://localhost:29556");

    // 4) application specific work (connecting clients to the server, any arbitrary work)
    NosEngine::Client::Bus bus_1(transport_hub, "copy://example:0", "Bus_1");
    // ...

    // create nodes with bus_1
    // ...

    // perform message exchange between nodes
    // ...

    return 0;
}
```

```
// Machine 1 - Client2

#include <Client/Bus.hpp>

int main(int, char**)
{
    // connect to server – specify transport string & bus name
    NosEngine::Client::Bus bus_1("ipc://example:0", "Bus_1");

    // create nodes with bus_1
    // ...

    // perform message exchange between nodes
    // ...

    return 0;
}
```

```
// Machine 2 - Client3

#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:

```
DataNode* get_data_node(std::string name)
DataNode* get_or_create_data_node(std::string name, const size_t &timeout)
void remove_data_node(const std::string& name, const size_t &timeout)
```

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 not use 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*.

Additionally, if a timeout is specified and the server does not respond within the desired time; then the registration request will fail and an error will be thrown.

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.

Additionally, if a timeout is specified and the server does not respond within the desired time; then the removal request will fail and an error will be thrown.

The sample code in **Figure 11** shows and explains the dos and don'ts of the registration methods in a variety of situations.

```
#include <Server/Server.hpp>
#include <Client/Bus.hpp>
#include <Utility/Error.hpp>

int main(int, char**)
```

```
{
    NosEngine::Server::Server server;
    server.add_transport("copy://example:0");

    // Create 3 Client::Bus instances: two for Bus_1, 1 for Bus_2
    NosEngine::Client::Bus bus_1_client_1("copy://example:0", "Bus_1");
    NosEngine::Client::Bus bus_1_client_2("copy://example:0", "Bus_1");
    NosEngine::Client::Bus bus_2_client("copy://example:0", "Bus_2");

    NosEngine::Client::DataNode *bus1_node1;

    //-------------------------------------------------------

    //ERROR: node does not exist yet (returns null)
    bus_1_client_1.get_data_node("Node_1");

    //Success
    bus_1_client_1.get_or_create_data_node("Node_1");

    //Success
    bus1_node1 = bus_1_client_1.get_data_node("Node_1");

    //-------------------------------------------------------

    //ERROR: node not created with this instance (returns null)
    bus_1_client_2.get_data_node("Node_1");

    //ERROR: node already registered on the bus (throws exception)
    try
    {
        bus_1_client_2.get_or_create_data_node("Node_1");
    }
    catch (NosEngine::Utility::Error::Error &e)
    {
        std::cout << e << std::endl;
    }

    //-------------------------------------------------------

    //Success - different bus
    bus_2_client.get_or_create_data_node("Node_1");

    //-------------------------------------------------------

    //ERROR: node not created with this instance
    bus_1_client_2.remove_data_node("Node_1");

    //Success; bus1_node1 pointer is now invalid
    bus_1_client_1.remove_data_node("Node_1");
}
```

Figure 11: 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 segmentation faults).

All of the 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 Asynchronous Receiving

Asynchronous receiving of messages is 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 #1: Basic Send-Receive (Non-Blocking)

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

```
#include <iostream>

#include <Server/Server.hpp>
#include <Client/Bus.hpp>
#include <Client/DataNode.hpp>
#include <Common/Message.hpp>

int main(int, char**)
{
    NosEngine::Server::Server server;
    server.add_transport("copy://example:0");

    NosEngine::Client::Bus bus_1("copy://example:0", "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_received_callback([&receiver](NosEngine::Common::Message)
    {
        std::cout << "Received a message!" << std::endl;
    });

    sender->send_message("Receiver", 6, "Hello");

    std::cin.get();
}
```

3.3.4 Blocking Receiving

Messages may also be received by calling the following blocking method:

```
NosEngine::Common::Message receive_message(const size_t &timeout);
```

This method blocks until a message is available to be returned and must be repeatedly called to continuously receive messages. Unless there is a good reason to use this method, non-blocking (asynchronous) receipt of messages is recommended.

NOTE: While non-blocking and blocking receive can be used simultaneously, it is recommended that one and only one method be used (for a given node).

Additionally, if a timeout is specified and a message is not received within the desired time; then the blocking receive will fail and an error will be thrown.

3.3.5 Example #2: Basic Send-Receive (Blocking)

```
#include <iostream>

#include <Server/Server.hpp>
#include <Client/Bus.hpp>
#include <Client/DataNode.hpp>
#include <Common/Message.hpp>

int main(int, char**)
{
    NosEngine::Server::Server server;
    server.add_transport("copy://example:0");

    NosEngine::Client::Bus bus_1("copy://example:0", "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");

    sender->send_message("Receiver", 6, "Hello");

    NosEngine::Common::Message message = receiver->receive_message();
    std::cout << "Received a message!" << std::endl;
}
```

3.3.6 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 <iostream>

#include <Server/Server.hpp>
#include <Client/Bus.hpp>
#include <Client/DataNode.hpp>
#include <Common/Message.hpp>

using namespace NosEngine;

int main(int, char**)
{
    Server::Server server;
    server.add_transport("copy://example:0");

    Client::Bus bus_1("copy://example:0", "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_received_callback([&receiver](Common::Message message)
    {
        int length;
        length = message.buffer.len - Common::Protocol::USER_DATA_START;
        // length == 1
        // msg.buffer.data[USER_DATA_START] == 'A'
        std::cout << "Received a message: "
            << "length=" << length << " "
            << "data=" << message.buffer.data[Common::Protocol::USER_DATA_START]
            << std::endl;
    });

    sender->send_message("Receiver", 2, "A"); // sending the single letter, A

    std::cin.get();
}
```

Another option to access the payload data is to use the utility class from *DataBufferOverlay*. This class is a simple 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 <iostream>

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

```
#include <Client/DataNode.hpp>
#include <Common/Message.hpp>
#include <Common/DataBufferOverlay.hpp>

using namespace NosEngine;

int main(int, char**)
{
    Server::Server server;
    server.add_transport("copy://example:0");

    Client::Bus bus_1("copy://example:0", "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_received_callback([&receiver](Common::Message message)
    {
        Common::DataBufferOverlay overlay(message.buffer);
        // overlay.len == 1
        // overlay.data[0] == 'A'
        std::cout << "Received a message: "
            << "length=" << overlay.len << " "
            << "data=" << overlay.data[0]
            << std::endl;
    });

    sender->send_message("Receiver", 2, "A"); // sending the single letter, A

    std::cin.get();
}
```

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

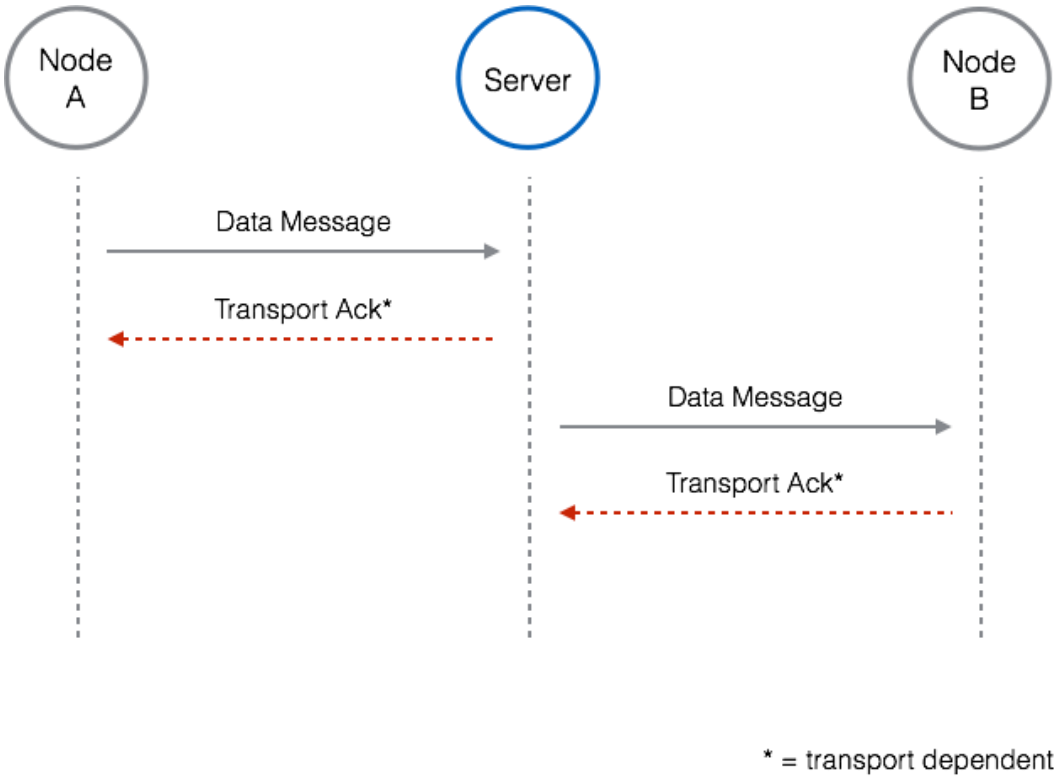

Figure 12: Non-Confirmed Data Flow

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.

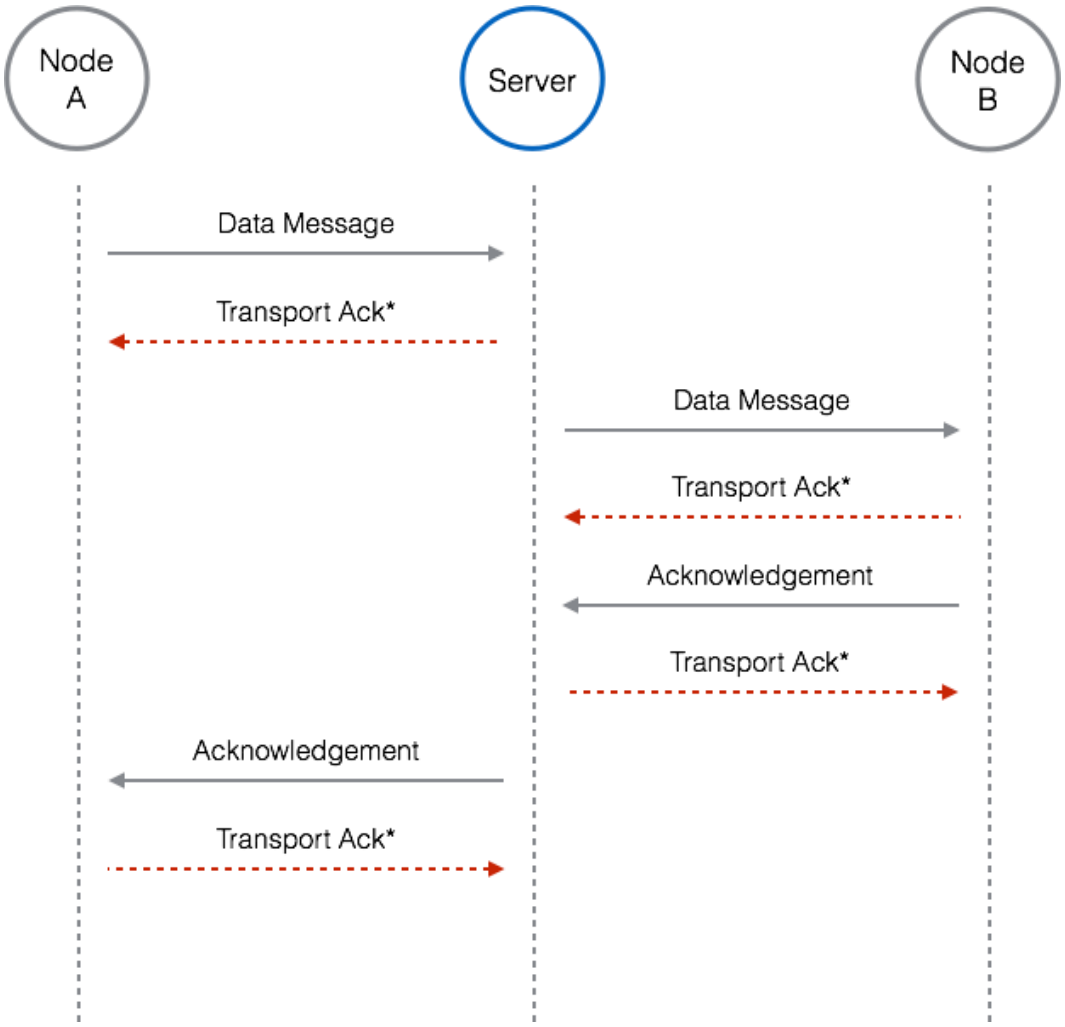

Figure 13: Confirmed Data Flow

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 14 shows source code a basic send-reply operation.

```
#include <iostream>

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

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

int main(int, char**)
{
    NosEngine::Server::Server server;
    server.add_transport("copy://example:0");

    NosEngine::Client::Bus bus_1("copy://example:0", "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_received_callback([&player_two](NosEngine::Common::Message message) {
        // normally should verify the message (the source, is duplex, contents)
        player_two->send_reply_message(message, 5, "Pong");
    });

    // this call will block until the response is received
    NosEngine::Common::Message response_message =
        player_one->send_request_message("Player_2", 5, "Ping");

    // the response_message contains "Pong"
    NosEngine::Common::DataBufferOverlay overlay(response_message.buffer);
    std::string response(overlay.data, overlay.len);
    std::cout << response << std::endl;
}
```

Figure 14: 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 <Server/Server.hpp>
#include <Client/Bus.hpp>
#include <Client/DataNode.hpp>
#include <Client/InterceptorNode.hpp>

int main(int, char**)
{
    NosEngine::Server::Server server;
    server.add_transport("copy://example:0");

    NosEngine::Client::Bus bus_1("copy://example:0", "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",
        NosEngine::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.

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

int main(int, char**)
{
    NosEngine::Server::Server server;
    server.add_transport("copy://example:0");

    NosEngine::Client::Bus bus_1("copy://example:0", "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",
        NosEngine::Common::BusProtocol::InterceptorDirection::Incoming);

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

```
        //call an interceptor action (pass, block, modify, mimic)
        //interceptor->send_X_response(...)
    });
}
```

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 operates exclusively with pass operations may be used like a "real-time" traffic monitor.

Figure 15 shows the message flow when passing a message through the send side. **Figure 16** shows the message flow when the interception occurs on the reply side (in case of a send-reply exchange).

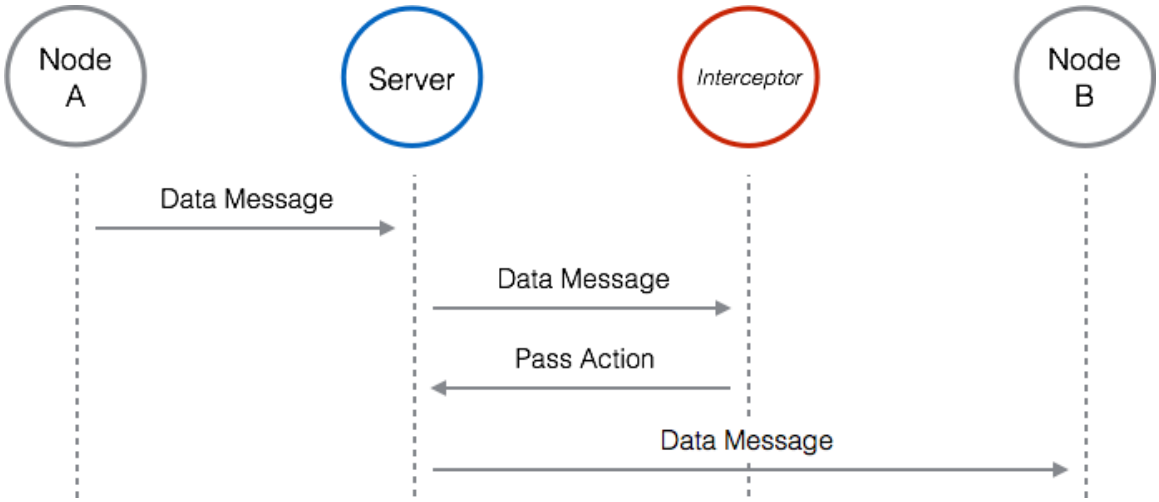

Figure 15: Send Chain Interceptor Pass

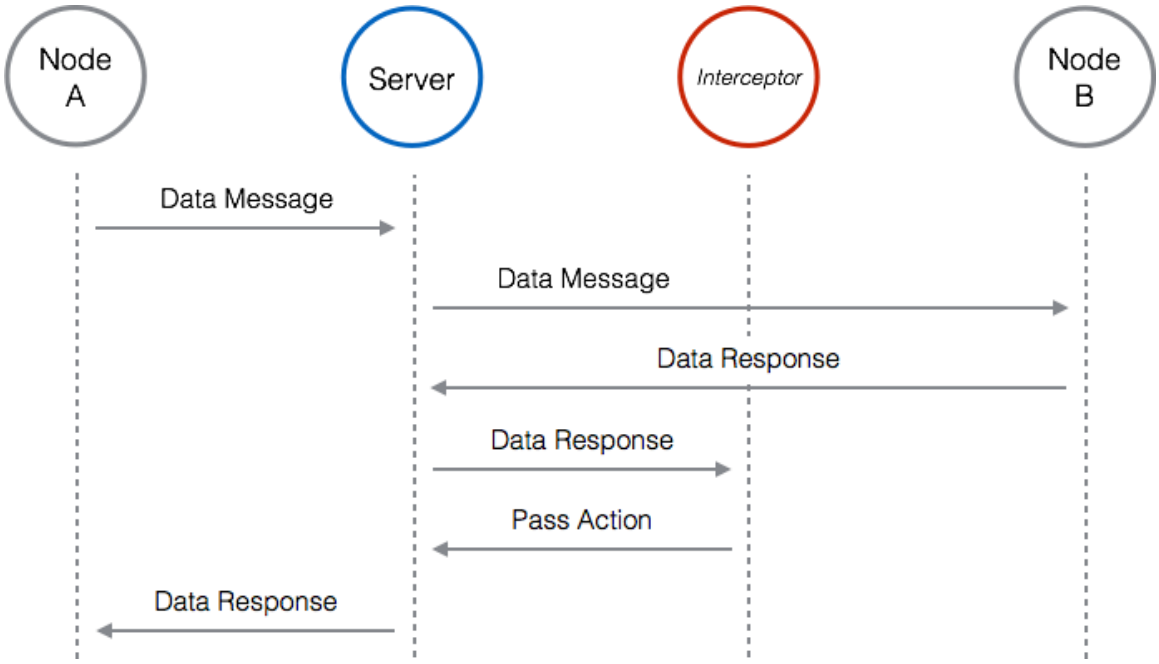

Figure 16: Receive Chain Interceptor Pass

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 17 shows the message flow when blocking a message during the send side. **Figure 18** shows the message flow when the interception occurs on the reply side (in case of a send-reply exchange).

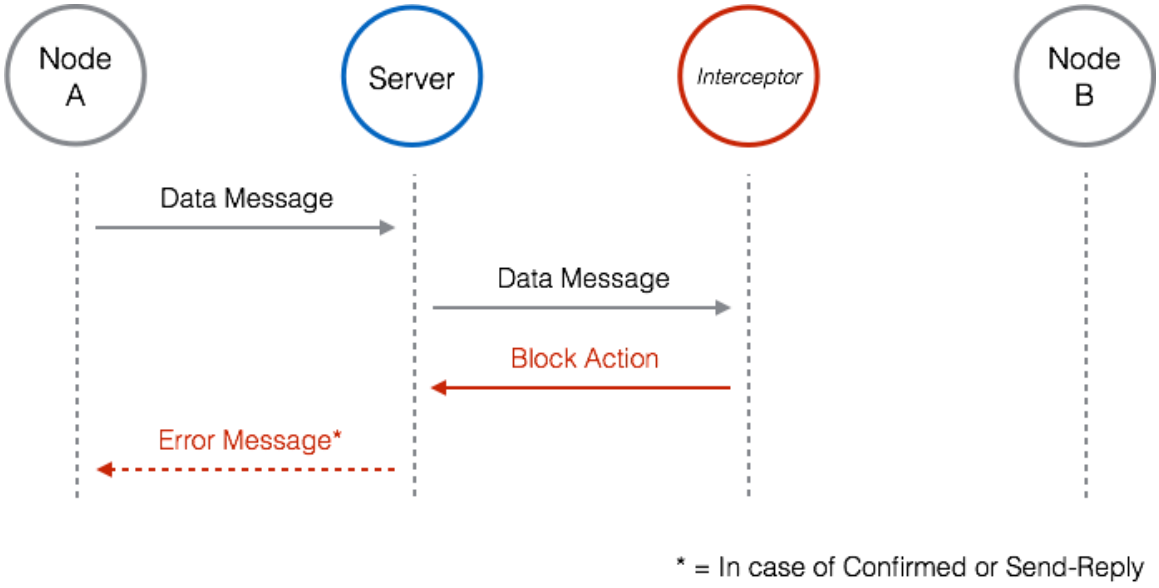

Figure 17: Send Chain Interceptor Block

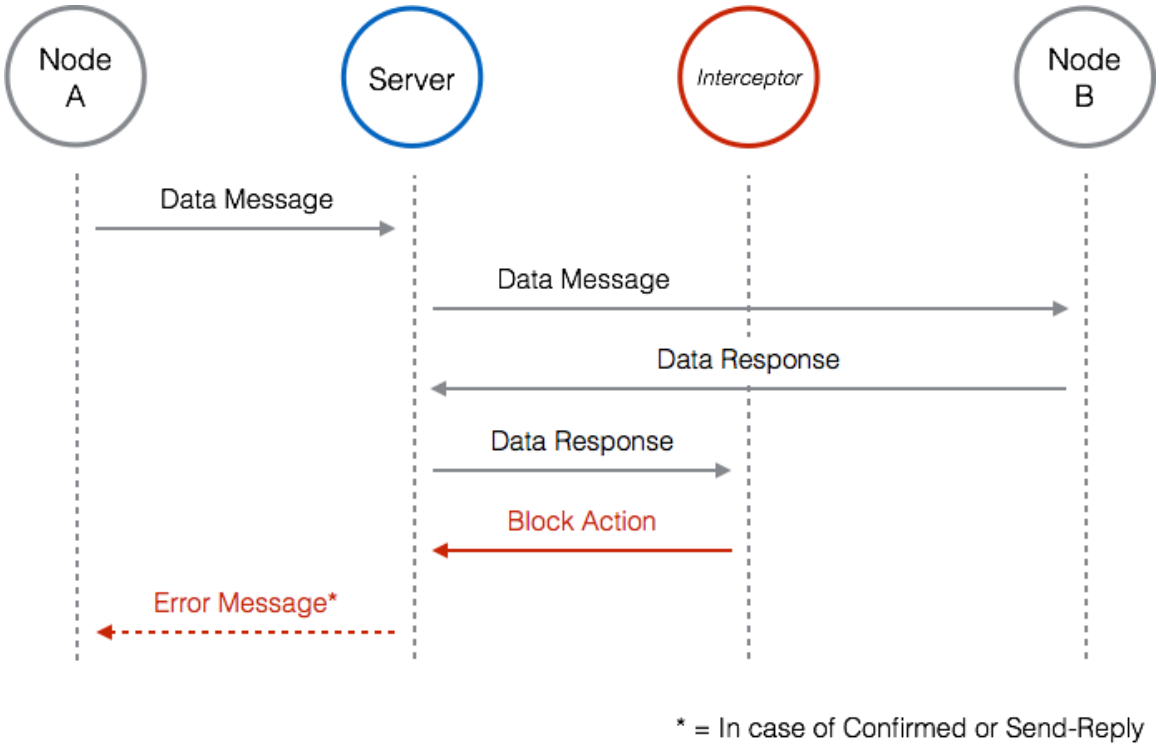

Figure 18: Reply Chain Interceptor Block

3.5.2.3 Modify

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

Figure 19 shows the modify operation on the send-side. **Figure 20** shows the modify operation on the reply-side.

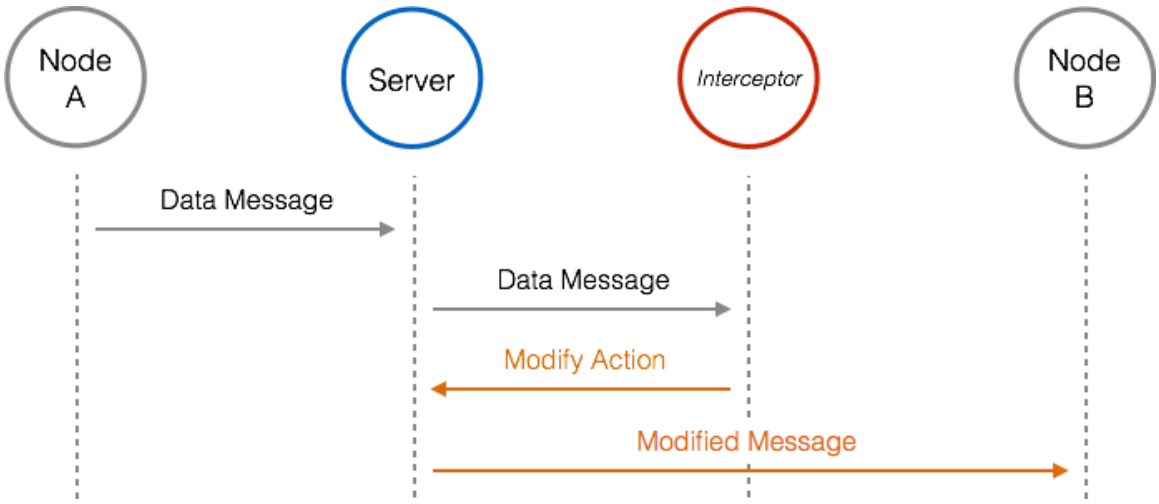

Figure 19: Send Chain Interceptor Modify

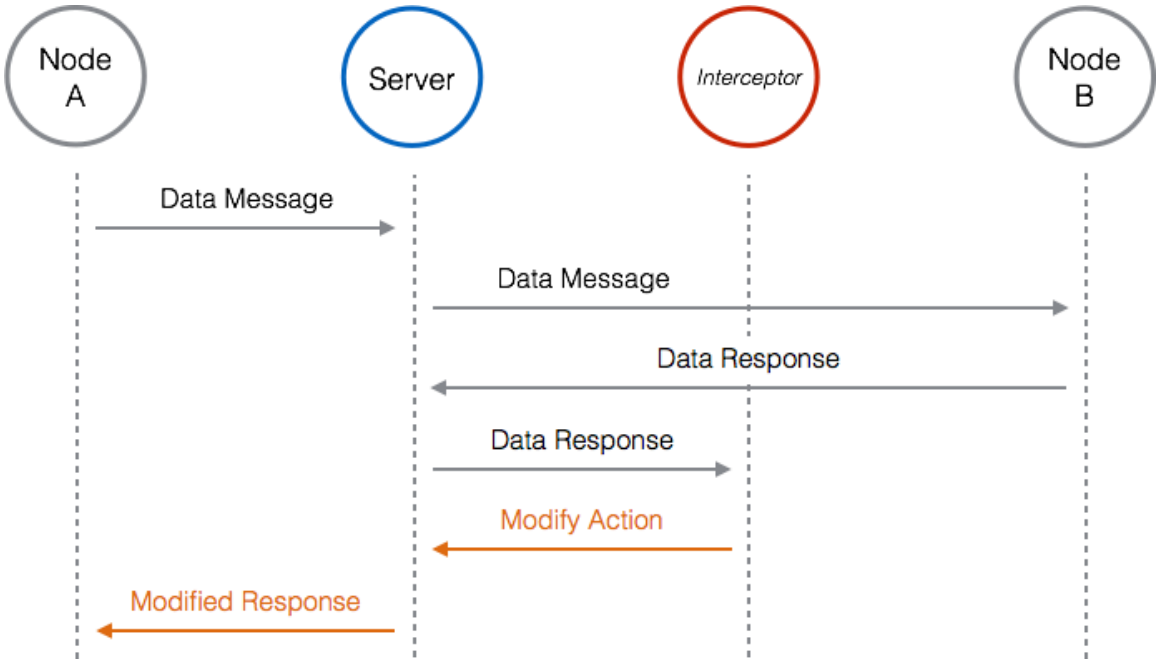

Figure 20: Reply Chain Interceptor Modify

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 21** shows the flow of a mimic operation.

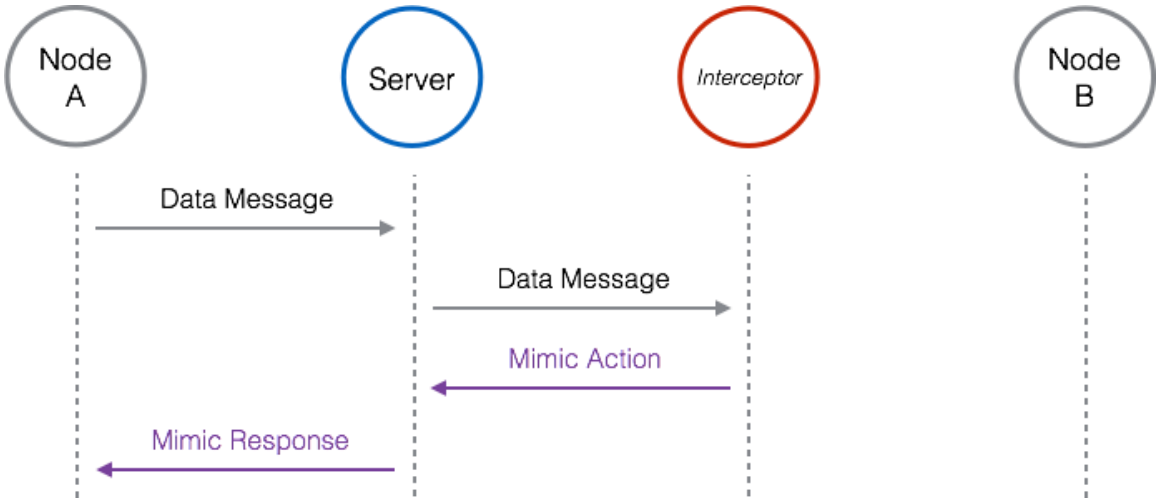

Figure 21: Interceptor Mimic

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 by two different node types: *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 22** illustrates the behavior between the *TimeSender* and *TimeClients* when a time value is sent.

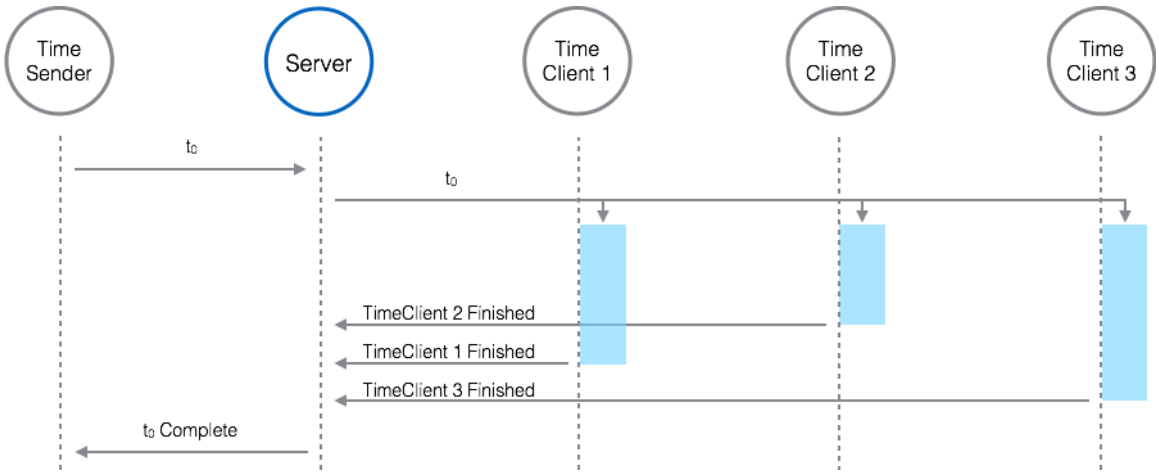

Figure 22: Time Distribution

Note that *TimeSender* and *TimeClient* nodes are not registered and used directly in the same way that a *DataNode* is. Instead sending and receiving of time is baked directly into the *Client::Bus* class.

3.6.1 Setting Bus Time

TimeSender nodes are not directly registered or used; instead time sending is baked directly into *Client::Bus*. Time sending is enabled by calling *Bus::enable_set_time*; and of course, there can only be one *Bus* instance which is enabled for setting time.

Once enabled, the *Bus* object is used to distribute time using the *Bus::set_time(…)* call. The first parameter is *SimTime (*currently defined as a 64-bit signed integer). Sorry, there is no support for floating-precision numbers; get creative!

Note that both *enable_set_time* and *set_time* have a timeout parameter. If a timeout is specified and a response isn't received from the server within the desired time; then the request fails and an error is thrown.

```
void enable_set_time(const size_t &timeout);
void set_time(NosEngine::Common::SimTime time, const size_t &timeout);
```

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

```
#include <Server/Server.hpp>
#include <Client/Bus.hpp>
#include <Utility/Error/Error.hpp>

int main(int, char**)
{
    NosEngine::Server::Server server;
    server.add_transport("copy://example:0");
```

```
    NosEngine::Client::Bus bus_1("copy://example:0", "Bus_1");

    // ERROR: set time is not enabled
    try
    {
        bus_1.set_time(0);
    }
    catch (NosEngine::Utility::Error::Error &e)
    {
        std::cout << e << std::endl;
    }

    // Success
    bus_1.enable_set_time();

    // Send time 0 to all clients, blocks until all clients are done
    bus_1.set_time(0);
}
```

3.6.2 Receiving Bus Time

TimeClient nodes are also not directly registered or used; receiving of time is also baked directly into *Client::Bus*. To receive time a 'tick' callback must be added to the *Bus*. 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 <iostream>

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

using namespace NosEngine;

int main(int, char**)
{
    Server::Server server;
    server.add_transport("copy://example:0");

    Client::Bus bus_1("copy://example:0", "Bus_1");
    bus_1.enable_set_time();
    bus_1.set_time(0);

    bus_1.add_time_tick_callback([](Common::SimTime time) {
        //perform any work as necessary
        std::cout << time << std::endl;
        //...

        //return when work is complete
    });

    // send time tick for every 10 units of time
    for (Common::SimTime time = 10; time < 100; time += 10)
    {
        bus_1.set_time(time);
    }
}
```

3.6.3 Retrieving Last Bus Time Received

The time for the last tick received can also be retrieved directly from a *Client::Bus* or *Client::Node* instance via the *get_time* method. This method can be called from inside or outside of a 'tick' callback, in fact this method will work even if no 'tick' callbacks have been added to the *Bus*.

```
#include <iostream>

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

using namespace NosEngine;

int main(int, char**)
{
    Server::Server server;
    server.add_transport("copy://example:0");

    Client::Bus bus_1("copy://example:0", "Bus_1");
    bus_1.enable_set_time();

    NosEngine::Client::DataNode *node_1 = bus_1.get_or_create_data_node("Node_1");

    bus_1.set_time(0);
    std::cout << "Time: "
        << "bus=" << bus_1.get_time() << " "
        << "node=" << node_1->get_time()
        << std::endl;

    bus_1.set_time(100);
    std::cout << "Time: "
        << "bus=" << bus_1.get_time() << " "
        << "node=" << node_1->get_time()
        << std::endl;
}
```

3.6.4 Timers

Sometimes you might not want to have your 'tick' callback executed for every time update which is received; this is where timers come in. The *Client::Bus* class has the following two methods which allow for adding a timer which is executed once (and only once).

```
void set_timer_absolute(Common::SimTime time, BusTimerCallback callback);
void set_timer_relative(Common::SimTime time, BusTimerCallback callback);
```

The *BusTimerCallback* has the following signature.

```
void timer_callback(Common::SimTime time, Common::SimTime requested_time)
```

Note that for both absolute and relative timers, if the time provided is not a multiple of the time sender's period; then the callback will be run for the tick which occurs immediately after the provided time. If the period is 1000 ms (simulated time), a wait for 150 ms (delta) or 1150 ms (absolute) will return about 850 ms (simulated time) later than expected. It is recommended to wait for multiples of the period.

An example of how to create timers is shown in **Figure 23**.

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

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

using namespace NosEngine;
typedef Common::SimTime SimTime;

int main(int, char**)
{
    Server::Server server;
    server.add_transport("copy://example:0");

    Client::Bus bus_1("copy://example:0", "Bus_1");
    bus_1.enable_set_time();
    bus_1.set_time(0);

    // add a timer to be executed 10 units of time from current time
    bus_1.set_timer_relative(10, [&bus_1](SimTime time, SimTime requested_time) {
        std::cout << time << " " << requested_time << std::endl;

        // add a timer to be executed 5 units of time from current time
        bus_1.set_timer_relative(5, [&bus_1](SimTime time, SimTime requested_time) {
            std::cout << time << " " << requested_time << std::endl;
        });
    });

    // add a timer to be executed at time 50
    bus_1.set_timer_absolute(50, [](SimTime time, SimTime requested_time) {
        std::cout << time << " " << requested_time << std::endl;
    });

    // add a timer to be executed at time 52
    bus_1.set_timer_absolute(52, [](SimTime time, SimTime requested_time) {
        std::cout << time << " " << requested_time << std::endl;
    });

    // add a timer to be executed at time 55
    bus_1.set_timer_absolute(55, [](SimTime time, SimTime requested_time) {
        std::cout << time << " " << requested_time << std::endl;
    });

    // add a timer to be executed at time 58
    bus_1.set_timer_absolute(58, [](SimTime time, SimTime requested_time) {
        std::cout << time << " " << requested_time << std::endl;
    });

    // send time tick for every 10 units of time
    for (SimTime time = 10; time < 100; time += 10)
    {
        bus_1.set_time(time);
    }
}
```

Figure 23 Timer Example

The output from this example is:

```
10 10
20 15
50 50
60 52
60 55
60 58
```

3.6.5 Bus Groups

A *Client::BusGroup* is a container of *Client::Bus* instances which send time at the same rate. When a *Bus* is added to a *BusGroup* the *enable_set_time* method for the bus is called; therefore the *BusGroup::add* method will throw any errors which *enable_set_time* would. An example is shown in **Figure 24** below.

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

using namespace NosEngine;

int main(int, char**)
{
    Server::Server server;
    server.add_transport("copy://example:0");

    Client::Bus bus_1("copy://example:0", "Bus_1");
    Client::Bus bus_2("copy://example:0", "Bus_2");

    bus_1.add_time_tick_callback([](Common::SimTime time) {
        std::cout << "Bus_1 time: " << time << std::endl;
    });

    bus_2.add_time_tick_callback([](Common::SimTime time) {
        std::cout << "Bus_2 time: " << time << std::endl;
    });

    Client::BusGroup bus_group;
    bus_group.add(&bus_1);
    bus_group.add(&bus_2);

    // Set the time to 10 for all buses in the group
    bus_group.set_time(10);
}
```

Figure 24 BusGroup Example

4 C Interface

While NOS Engine is written in C++, it can be used in C code by utilizing the included C Interface headers. The C interface header for a NOS Engine library will be named CInterface.h. For example, to include the C interface header for the client library, the following line would need to be added to your C file:

```
#include <Client/CInterface.h>
```

NOTE: Version 1.4 of NOS Engine does not provide a C Interface for the server-side library(s); however, it will be added in a future release.

NOTE: Version 1.4 of NOS Engine does not support interception via the C Interface; however, it will be added in a future release.

4.1 C Interface Examples

4.1.1 Example #1 Client Connect

The example in **Figure 25** shows how to create a client-side bus.

```
#include <stdio.h>
#include <Client/CInterface.h>
```

```
int main()
{
  NE_Bus *bus_1 = NULL;

  // connect to server – specify transport string & bus name
  bus_1 = NE_create_bus2("Bus_1", "tcp://localhost:26555");
  if(NE_failed())
  {
      printf("Failed to create bus!\n");
  }

  // create nodes with bus_1...

  // perform message exchange between nodes...

  // perform clean up
  NE_destroy_bus(&bus_1);

  return 0;
}
```

Figure 25: C Interface - Client Connect

4.1.2 Example #2 Client Connect with Shared Transport Hub

The example in **Figure 26** shows how to create a two client-side buses that share the same transport.

```
#include <stdio.h>
#include <Client/CInterface.h>

int main()
{
  NE_TransportHub *transport_hub = NULL;
  NE_Bus *bus_1 = NULL;
  NE_Bus *bus_2 = NULL;

  // create transport hub with default number of threads
  transport_hub = NE_create_transport_hub(0);
  if(NE_failed())
  {
      printf("Failed to create transport hub!\n");
  }

  // connect to server – specify transport string & bus name
  bus_1 = NE_create_bus(transport_hub, "Bus_1", "tcp://localhost:26555");
  if(NE_failed())
  {
      printf("Failed to create bus 1!\n");
  }

  bus_2 = NE_create_bus(transport_hub, "Bus_2", "tcp://localhost:26555");
  if(NE_failed())
  {
      printf("Failed to create bus 2!\n");
  }

  // create nodes with bus_X...

  // perform message exchange between nodes...
```

```
  // perform clean up
  NE_destroy_bus(&bus_1);

  return 0;
}
```

Figure 26: C Interface - Client Connect with Shared Transport Hub

4.1.3 Example #3 Client Send and Receive

The example in **Figure 27** shows how to create a client and send and receive a message.

```
#include <stdio.h>
#include <Client/CInterface.h>

int main()
{
    NE_Bus *bus_1 = NULL;
    NE_DataNode *bus_1_node_1 = NULL;
    NE_DataNode *bus_1_node_2 = NULL;
    NE_Message *msg = NULL;

    // connect to server – specify transport string & bus name
    bus_1 = NE_create_bus2("Bus_1", "tcp://localhost:26555");
    if (NE_failed())
    {
        printf("Failed to create bus!\n");
    }

    // create nodes with bus_1
    bus_1_node_1 = NE_create_data_node(bus_1, "Node_1");
    if (NE_failed())
    {
        printf("Failed to create node 1!\n");
    }

    bus_1_node_2 = NE_create_data_node(bus_1, "Node_2");
    if (NE_failed())
    {
        printf("Failed to create node 2!\n");
    }

    // perform message exchange between nodes
    NE_data_node_send_message_sync(bus_1_node_1, "Node_2", 6, (const uint8_t*)"Hello");
    if (NE_failed())
    {
        printf("Failed to send a message!\n");
    }

    msg = NE_data_node_receive_message_sync(bus_1_node_2);
    if (NE_ok())
    {
        printf("Received a message: %s\n", NE_message_get_user_data(msg));
        NE_destroy_message(&msg);
    }
    else
    {
        printf("Failed to receive a message!\n");
    }

    // perform clean up
```

```
    NE_destroy_data_node(bus_1, &bus_1_node_1);
    NE_destroy_data_node(bus_1, &bus_1_node_2);
    NE_destroy_bus(&bus_1);

    return 0;
}
```

Figure 27: C Interface - Client Send and Receive

4.1.4 Example #4 Client Request and Reply

The example in **Figure 28** shows how to create a client and send a request. The receiving node receives the request message asynchronously. The *NE_data_node_send_request_message_sync* call blocks until a reply is received from the receiving node.

```
#include <stdio.h>
#include <Client/CInterface.h>

void message_received(NE_DataNode *receive_data_node, NE_Message *message)
{
    printf("Received a request: %s\n", NE_message_get_user_data(message));

    // send reply message to node 1
    NE_data_node_send_reply_message_sync(
        receive_data_node, message, 8, (const uint8_t*)"Goodbye");

    // NOTE: do NOT destroy the received message
}

int main()
{
    NE_Bus *bus_1 = NULL;
    NE_DataNode *bus_1_node_1 = NULL;
    NE_DataNode *bus_1_node_2 = NULL;
    NE_Message *reply_msg = NULL;

    // connect to server – specify transport string & bus name
    bus_1 = NE_create_bus2("Bus_1", "tcp://localhost:26555");
    if (NE_failed())
    {
        printf("Failed to create bus!\n");
    }

    // create nodes with bus_X
    bus_1_node_1 = NE_create_data_node(bus_1, "Node_1");
    if (NE_failed())
    {
        printf("Failed to create node 1!\n");
    }

    bus_1_node_2 = NE_create_data_node(bus_1, "Node_2");
    if (NE_failed())
    {
        printf("Failed to create node 2!\n");
    }

    // set callback for node 2 to receive messages asynchronously
    NE_data_node_set_message_received_callback(bus_1_node_2, message_received);
    if (NE_failed())
    {
```

```
        printf("Failed to set message received callback!\n");
    }

    // perform message exchange between nodes
    NE_data_node_send_request_message_sync(
        bus_1_node_1, "Node_2", 6, (const uint8_t*)"Hello", &reply_msg);
    if (NE_failed())
    {
        printf("Failed to send request!\n");
    }
    else
    {
        printf("Received a reply: %s\n", NE_message_get_user_data(reply_msg));
        NE_destroy_message(&reply_msg);
    }

    // perform clean up
    NE_destroy_data_node(bus_1, &bus_1_node_1);
    NE_destroy_data_node(bus_1, &bus_1_node_2);
    NE_destroy_bus(&bus_1);

    return 0;
}
```

Figure 28: C Interface - Client Request and Reply

4.1.5 Example #5 Sending and Receiving Time

The example in **Figure 29** shows how to enable setting of time and how to set the time. A 'tick' callback is added to receive the time ticks asynchronously. The *NE_bus_set_time* call blocks until all 'tick' callbacks have finished receiving and processing the current time tick.

```
#include <stdio.h>
#include <Client/CInterface.h>

#ifdef WIN32
#include <windows.h>
#define SLEEP(X) Sleep(X)
#else
#include <unistd.h>
#define SLEEP(X) usleep(X * 1000)
#endif

void time_tick_received(NE_SimTime time)
{
    printf("Received a time tick: %li\n", time);
    SLEEP(1000);
}

int main()
{
    NE_Bus *bus_1 = NULL;
    NE_SimTime time = 0;

    // connect to server – specify transport string & bus name
    bus_1 = NE_create_bus2("Bus_1", "tcp://localhost:26555");
    if (NE_failed())
    {
        printf("Failed to create bus!\n");
    }
```

```
    // enable set time with bus_X
    NE_bus_enable_set_time(bus_1);

    // set callback for time client to receive time ticks asynchronously
    NE_bus_add_time_tick_callback(bus_1, time_tick_received);
    if (NE_failed())
    {
        printf("Failed to set time tick received callback!\n");
    }

    // send time ticks to time client
    while (time < 5)
    {
        printf("Sending time tick: %li\n", time);
        NE_bus_set_time(bus_1, time);
        if (NE_failed())
        {
            printf("Failed to send time!\n");
            break;
        }

        time++;
    }

    // perform clean up
    NE_destroy_bus(&bus_1);

    return 0;
}
```

Figure 29: C Interface - Sending and Receiving Time

5 MIL-STD-1553

5.1 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.

5.2 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

5.3 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.

5.3.1 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.

5.3.1.1 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.

5.3.1.2 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 30**). 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.

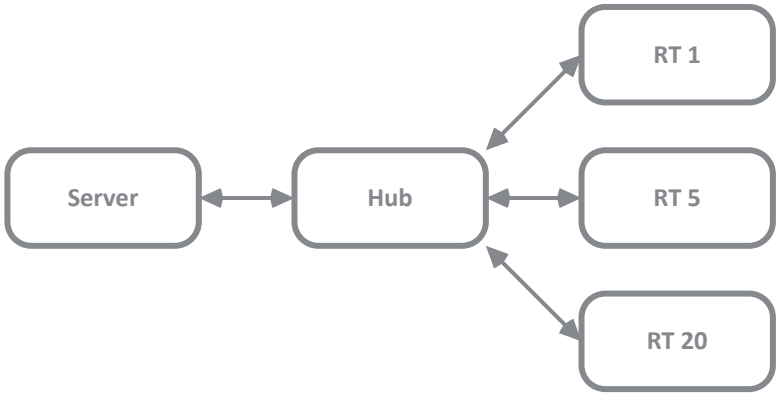

Figure 30: RT communication through the Hub

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.

5.3.2 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.

5.3.3 Server

The requirements for MS1553 communications within the server are very minimal. This is mostly due to the fact that 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.

5.3.4 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.

5.4 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.

6 SpaceWire

6.1 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.

6.2 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.

6.3 Usage

6.3.1 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.

6.3.1.1 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 31**).

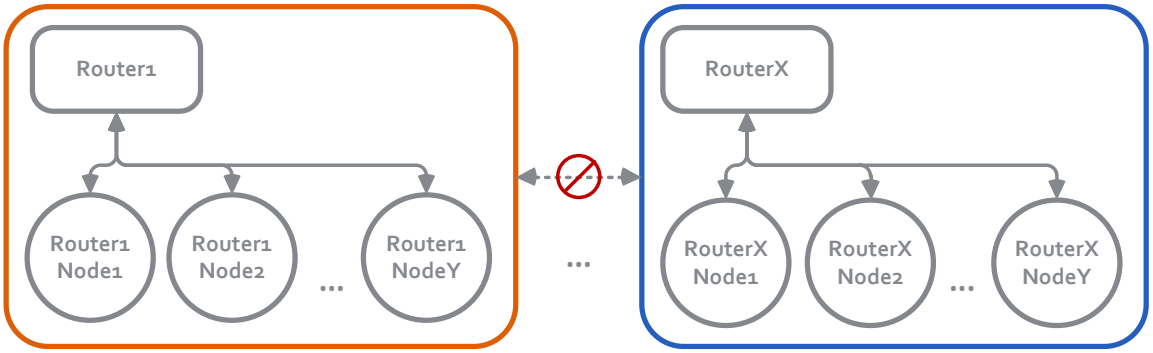

Figure 31: SpaceWire Network without routing

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.

6.3.1.2 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.

6.3.2 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.

6.3.3 Server

6.3.3.1 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.).

6.3.4 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.

6.4 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

7 I2C

7.1 Overview

The purpose of this section is to familiarize users and developers with the inner workings and usage of the NOS Engine I2C protocol layer, which is a client side NOS Engine component. The I2C API is designed to provide a clean, easy to use interface that mimics common I2C hardware APIs, without being too verbose. This hardware protocol layer hides as much of the NOS Engine concepts and objects as possible, such as buses and nodes, so that it is familiar to developers experienced at working with various hardware APIs. The I2C protocol layer essentially wraps an I2C style API around base NOS Engine messaging capabilities, hiding all NOS Engine implementation details.

All I2C addresses, which identify all I2C devices, are defined as the base address, meaning the read/write bit is not included. The transaction direction is inferred from the read/write method calls, so adding an address bit is unnecessary at this level of abstraction. The result is that the base I2C address is used for all calls, so it is not necessary for the user to perform any bit manipulation on the address.

Although the I2C protocol API appears similar to a real I2C hardware API, the messaging differs at the hardware procotol level for performance reasons. For example, rather than sending and requiring slave devices to ACK/NACK individual bytes, as indicated in the I2C specification, the entire transaction is handled in a single transport round trip. The slave device returns the actual number of bytes read/written, providing a mechanism to model transaction errors at a higher level, such as a busy condition or a pre-mature NACK. This level of abstraction was chosen to balance performance with increased fidelity of a protocol that would exactly model the hardware bus.

7.2 Terms

I2C Master	Device that starts/stops all communications with slave devices
I2C Slave	Device that listens to the bus to be addressed by the master and responds to transactions
I2C Address	Address used to identify device, typically 7-bits, but I2C extensions allow 10-bit addressing mode, although not widely supported
TWI	Two wire interface, another name used for I2C due to previous trademark issues
Multi-Master	Operating mode in which I2C bus has multiple master devices that each can send commands, handled by arbitration

7.3 Usage

The client side I2C protocol layer consists of both a C++ API and a C interface, which wraps the C++ API. This section describes using the API and provides quick code examples. All NOS Engine objects are created/connected during construction, so a data node on the specified bus will exist with the node name equal to the I2C address. This happens automatically to hide NOS Engine constructs from the user to provide a more hardware API look and feel. This section assumes the user is familiar with configuring and starting an instance of the NOS Engine server, which is the central server that coordinates all message transactions. The bus name parameter defaults to "i2c", but can easily be specified to rename or if multiple I2C buses exist within a system.

7.3.1 I2C Master

An I2C master device can be created by simply instantiating an object of the I2CMaster class and passing in the appropriate parameters.

```
#include <string>
#include <cstdint>

#include <I2C/Client/I2CMaster.hpp>

const std::string SERVER_URI = "tcp://127.0.0.1:12000";
const NosEngine::I2C::I2CAddress MASTER_ADDRESS = 20;
const NosEngine::I2C::I2CAddress SLAVE_ADDRESS = 30;

int main()
{
    // create i2c master device
    NosEngine::I2C::I2CMaster master(MASTER_ADDRESS, SERVER_URI);

    // do work.....

    // i2c write (transmit data to slave device)
    uint8_t wbuf[] = { 0xde, 0xad };
    master.i2c_write(SLAVE_ADDRESS, wbuf, sizeof(wbuf));

    // i2c read (receive data from slave device)
    uint8_t rbuf[10];
    master.i2c_read(SLAVE_ADDRESS, rbuf, sizeof(rbuf));

    // i2c transaction (combined write/read operation)
    master.i2c_transaction(SLAVE_ADDRESS, wbuf, sizeof(wbuf), rbuf, sizeof(rbuf));

    return 0;
}
```

7.3.2 I2C Slave

An I2C slave device can be created by subclassing the abstract I2CSlave class, which has pure virtual read/write methods. These methods defined in the derived class are automatically called when the I2C master device starts a transaction. The I2C slave device returns the actual number of bytes read/written. As discussed earlier for performance, rather than an ACK/NACK of individual bytes, the slave device can use the return value to indicate various error conditions. For example, if the I2C master attempts to write 2 bytes, the I2C slave can return 1 byte, to indicate that only one byte was written, which could represent a premature NACK at the lowest hardware level.

```
#include <string>
#include <cstdint>

#include <I2C/Client/I2CSlave.hpp>

const std::string SERVER_URI = "tcp://127.0.0.1:12000";
const NosEngine::I2C::I2CAddress SLAVE_ADDRESS = 30;

// my custom i2c device
class MyDevice : public NosEngine::I2C::I2CSlave
{
```

```
public:
    MyDevice() : NosEngine::I2C::I2CSlave(SLAVE_ADDRESS, SERVER_URI) {}

    size_t i2c_read(uint8_t* rbuf, size_t rlen) {
        // return data
        return rlen;
    }

    size_t i2c_write(const uint8_t *wbuf, size_t wlen) {
        // do something with data
        return wlen;
    }
};

int main()
{
    // create i2c master device
    MyDevice slave;

    // loop, work, etc....

    return 0;
}
```

7.3.3 C Interface

The I2C C interface wraps all API features in C function calls. The main difference when using the C interface involves creating an I2C slave device. In order to handle transactions, the user must register a callback function that gets called in place of the pure virtual read/write methods that exist in the C++ API abstract base class. Both transaction directions, read and write, are handled via this single callback. The transaction direction is passed as the first parameter to the callback function, allowing the user to handle appropriately.

7.4 Unsupported Features

Currently the I2C protocol layer does not support all features of the I2C protocol. This is mainly due to lack of necessity in existing systems or uncommon usage in real hardware.

- Multi-Master mode arbitration
  - Multiple master devices can be created, but nothing prevents simultaneous bus transactions
- Reserved I2C addresses (general call address, CBUS, etc.)
  - General exception thrown if reserved address used
- Timing and bus speeds are not currently modeled
  - Each transaction is handled by a NOS Engine request message, which is a synchronous transaction that occurs in zero 'NOS' time

8 SPI

8.1 Overview

The purpose of this section is to familiarize users and developers with the inner workings and usage of the NOS Engine serial peripheral interface (SPI) protocol layer, which is a client side NOS Engine component. The SPI API

is designed to provide a clean, easy to use interface that mimics common SPI hardware APIs, without being too verbose. This hardware protocol layer hides as much of the NOS Engine concepts and objects as possible, such as buses and nodes, so that it is familiar to developers experienced at working with various hardware APIs. The SPI protocol layer essentially wraps an SPI style API around base NOS Engine messaging capabilities, hiding all NOS Engine implementation details.

The SPI protocol layer is similar to the I2C protocol layer, differing only in how slave devices are addressed. SPI devices use a separate chip-select line. In the virtual SPI bus, there is no notion of hardware connections, such as from the SPI master device to the slave device chip-selects, so an arbitrary chip select value is chosen when instantiating master/slave devices to represent the connections. Internally, this simply means the registered chip-select value is used as the base NOS Engine message destination node.

8.2 Terms

SPI Master	Device that starts/stops all communications with slave devices
SPI Slave	Device that listens to the bus when selected by the master and responds to transactions
Chip-Select (CS) / Slave-Select (SS)	Separate slave select line used to active slave device to start transactions

8.3 Usage

The client side SPI protocol layer consists of both a C++ API and a C interface, which wraps the C++ API. This section describes using the API and provides quick code examples. All NOS Engine objects are created/connected during construction, so a data node on the specified bus will exist with the node name equal to the chip-select value for slave devices and hardcoded to "master" for the master device. This happens automatically to hide NOS Engine constructs from the user to provide a more hardware API look and feel. This section assumes the user is familiar with configuring and starting an instance of the NOS Engine server, which is the central server that coordinates all message transactions. The bus name parameter defaults to "spi", but can easily be specified to rename or if multiple SPI buses exist within a system.

8.3.1 SPI Master

An SPI master device can be created by simply instantiating an object of the SpiMaster class and passing in the appropriate parameters.

```
#include <string>
#include <cstdint>

#include <Spi/Client/SpiMaster.hpp>

const std::string SERVER_URI = "tcp://127.0.0.1:12000";
const int SLAVE_CHIP_SELECT = 1;

int main()
{
    // create spi master device
    NosEngine::Spi::SpiMaster master(SERVER_URI);

    // do work.....

    // select slave device
```

```
    master.select_chip(SLAVE_CHIP_SELECT);

    // spi write (transmit data to slave device)
    uint8_t wbuf[] = { 0xde, 0xad };
    master.spi_write(wbuf, sizeof(wbuf));

    // spi read (receive data from slave device)
    uint8_t rbuf[10];
    master.spi_read(rbuf, sizeof(rbuf));

    // spi transaction (combined write/read operation)
    master.spi_transaction(wbuf, sizeof(wbuf), rbuf, sizeof(rbuf));

    // unselect slave device
    master.unselect_chip();

    return 0;
}
```

8.3.2 SPI Slave

An SPI slave device can be created by subclassing the abstract SpiSlave class, which has pure virtual read/write methods. These methods defined in the derived class are automatically called when the SPI master device starts a transaction. The SPI slave device returns the actual number of bytes read/written. The slave device can use the return value to indicate various error conditions. For example, if the SPI master attempts to write 2 bytes, the SPI slave can return 1 byte, to indicate that only one byte was written, which could represent a premature transaction termination at the lowest hardware level.

```
#include <string>
#include <cstdint>

#include <Spi/Client/SpiSlave.hpp>

const std::string SERVER_URI = "tcp://127.0.0.1:12000";
const int SLAVE_CHIP_SELECT = 1;

class MyDevice : public NosEngine::Spi::SpiSlave
{
public:
    MyDevice() : NosEngine::Spi::SpiSlave(SLAVE_CHIP_SELECT, SERVER_URI) {}

    size_t spi_read(uint8_t* rbuf, size_t rlen) {
        // return data
        return rlen;
    }

    size_t spi_write(const uint8_t *wbuf, size_t wlen) {
        // do something with data
        return wlen;
    }
};

int main()
{
    // create spi master device
    MyDevice slave;

    // loop, work, etc...
```

```
    return 0;
}
```

8.3.3 C Interface

The SPI C interface wraps all API features in C function calls. The main difference when using the C interface involves creating an SPI slave device. In order to handle transactions, the user must register a callback function that gets called in place of the pure virtual read/write methods that exist in the C++ API abstract base class. Both transaction directions, read and write, are handled via this single callback. The transaction direction is passed as the first parameter to the callback function, allowing the user to handle appropriately.

8.4 Unsupported Features

Due to the lack of a formal SPI standard, only basic messaging related to SPI communications is supported. The following features used by some SPI implementations are unsupported:

- Multi-Master mode
  - The SPI protocol layer prevents multiple SPI master devices from being created on a single bus
- Timing and bus speeds are not currently modeled
  - Each transaction is handled by a NOS Engine request message, which is a synchronous transaction that occurs in zero 'NOS' time

9 UART

9.1 Overview

The purpose of this section is to familiarize users and developers with the inner workings and usage of the NOS Engine UART protocol layer, which consists of a client side component and a server side plugin.

The client side UART API is designed to provide a clean, easy to use interface that mimics common UART APIs, without being too verbose. This protocol layer hides as much of the NOS Engine concepts and objects as possible, such as buses and nodes, so that it is familiar to developers experienced at communicating with UART devices. The UART protocol layer essentially wraps a UART style API around base NOS Engine asynchronous messaging capabilities, hiding all NOS Engine implementation details.

The UART server side plugin provides a means of point-to-point communications, throwing the appropriate errors if two endpoints are already connected (UART is in use). The plugin also maintains current UART connections, mapping integer port numbers to destination NOS Engine data nodes.

9.2 Terms

UART	Universal Asynchronous Receiver Transmitter

9.3 Usage

The UART protocol layer is separated into two distinct pieces. The client side API is designed to be the user facing portion of the code, while the server side is designed to be loaded into the base NOS Engine server as a plugin.

9.3.1 Client

The client side UART protocol layer consists of both a C++ API and a C interface, which wraps the C++ API. This section describes using the API and provides quick code examples. All NOS Engine objects are created/connected during construction, so a data node on the specified bus will exist with the specified name. This section assumes the user is familiar with configuring and starting an instance of the NOS Engine server, which is the central server that coordinates all message transactions. The bus name parameter defaults to "uart", but can easily be specified to rename or if multiple UARTs exist within a system.

A UART device can be created by simply instantiating an object of the Uart class and passing in the appropriate parameters. In order to facilitate communications, a specific UART port must be opened on both ends. The port number is a simple unsigned integer that can be arbitrarily chosen to represent a real UART port number (COM0, tty1, etc.). The key is that the same port must be opened by multiple UART devices to create the virtual connection. An error is thrown if a port is in use, meaning that two endpoints are already connected.

The code example only illustrates basic UART device creation and communication. Numerous additional methods exist to allow setting a callback function, specifying queue size, flushing the queue, peeking at the number of bytes available to read, etc. See the header file for a complete listing of available methods.

```
#include <string>
#include <cstdint>

#include <Uart/Client/Uart.hpp>

const std::string SERVER_URI = "tcp://127.0.0.1:12000";
const int PORT = 2;

int main()
{
    // create uart device 1 and open port
    NosEngine::Uart::Uart dev1("dev1", SERVER_URI);
    dev1.open(PORT);

    // create uart device 2 and open port
    NosEngine::Uart::Uart dev2("dev2", SERVER_URI);
    dev2.open(PORT);

    // send data
    uint8_t wbuf[] = { 0xde, 0xad, 0xbe, 0xef };
    dev1.write(wbuf, sizeof(wbuf));

    // receive data
    size_t num_bytes = dev2.available(); // should equal 4
    uint8_t rbuf[8];
    dev2.read(rbuf, num_bytes);

    // close connection
    dev1.close();

    return 0;
}
```

9.3.1.1 C Interface

The UART C interface wraps all API features in C function calls.

9.3.2 Server

The server side UART plugin implementation provides a specialized bus and router to facilitate connections. The majority of work involves maintaining connection states and mapping data node names to port numbers. When the UART device attempts to communicate over a port, the server side plugin determines the destination node, if any, and fills in the message header appropriately. The server side plugin also restricts point-to-point communications.

9.4 Unsupported Features

Many intricacies of UART communications that do not apply to a virtual software bus have been omitted from the API (BAUD rates, TERMIOS structure, etc.) in the current implementation.