Tuscarora Developers Manual
Version 2.0

The Samraksh Company

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings contained in this article/presentation are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Contents

1 Introduction

Tuscarora is a framework for developing and deploying multiple networking patterns in heterogeneous MANETs. This manual explains how to install and use Samraksh's Tuscarora framework.

Definitions

API: Short for Application Programming Interface. In our case this refers to a single publicly accessible method defined in a class. When referring to multiple methods it will be used in plural as APIs or as an Interface.

Interface: A set of publicly accessible methods defined in a class. An interface is usually defined as a pure abstract class that needs to be implemented by some class before it can be instantiated.

Neighbor: Node k is a neighbor of node j if a request to send a datagram from j to k could be directly acted upon and is expected to succeed in a single hop. A neighbor is identified by an identifier assigned independently by Tuscarora to that neighbor. That identifier has type `NodeId_t`, which is a 16-bit unsigned integer; it is sometimes called the "node ID" of the neighbor. Note that the identifiers assigned to neighbors have strictly local scope. In particular, different neighbors of the same node may know it by different node IDs.

Neighborhood: The neighborhood of a node is the set of current neighbors of that node.

Waveform Name: An identifier that is assigned independently by the framework to uniquely identify each waveform on the device. Colloquially this might be referred to as the Waveform ID.

Link: The ordered pair consisting of the Neighbor ID and Waveform ID of a waveform that is expected to be valid for reaching that neighbor. Simply put, it is a combination of the node ID and the waveform ID. The same Node ID may occur in multiple links, if a neighbor is reachable via more than one waveform.

Link Metrics: A measure of the performance, quality, or other property of a link that is sufficiently standardized that it may be used by patterns to compare the link with other links over potentially different waveforms. Please see Section 6.4 for more details.

Target Audience

This document is intended for developers who use the Tuscarora Framework to write applications, new Networking Patterns or to develop Waveforms that would be compatible with the framework. If you would like to understand the design rationale of the framework, please read the "Tuscarora Framework Design" document. If you are a Waveform developer you should also consult the "Waveform Interface Definitions" document.

Deployment Targets

The current version of the framework can be built and run only on the NS3-DCE simulation system as well as on any standard POSIX/Linux systems. The architecture allows flexibility as it allow single-process and multi-process setups depending on the platform requirements. The details of how to configure different types of builds and the associated binding requirements will be elaborated in chapter 3.

Tuscarora Framework Overview

Tuscarora is Framework for building scalable heterogeneous MANETs. The framework is designed to support many simultaneous instances of what would traditionally be called routing protocols, which we refer to as 'Application-Specific Networking Patterns' (ASNPs) or 'Network Patterns' or even simply as 'Patterns'. The core service provided by the framework is a Standardized Neighborhood Abstraction. Tuscarora framework design is influenced by Samraksh's work on DARPA's Fixed Wireless At a Distance program.

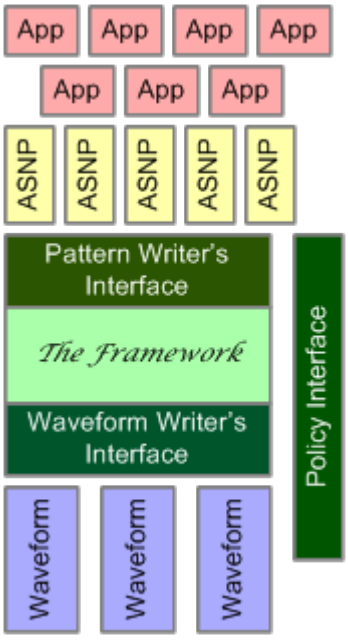


Figure 1.1: Framework Design Goal

Figure 1.1 shows the notional architecture of the framework. Tuscarora is designed to support these broad goals:

- Scaling of MANETs: By supporting multiple patterns and even multiple instances of a Pattern, an application will be able to choose a pattern that closely matches its data distribution needs, thereby decreasing overhead significantly versus the traditional any-to-any routing approach.
- Portability: A standardized abstraction for Patterns makes them portable across platforms and waveforms.
- Support Heterogeneity: Standardized abstractions for waveforms allow any waveform that implements the abstractions to be "plugged in" to the framework. Moreover, every Tuscarora node is a "gateway" node between all supported waveforms.
- Flexible Management: An open policy manager that operates on self-describing management interfaces of modules provides a very flexible way of managing the network. Further, it enables a human-in-the-loop management architecture to carry out network and mission management tasks.

To understand the design decisions better, please see the Tuscarora Design Rationale document.

Services

To accomplish the foregoing goals, the framework provides the following services:

- **Neighborhood service:** Discovers neighbors and evaluates the quality of their links; notifies Patterns (via callbacks) whenever there are changes in the neighborhood.
- **Data Flow:** Provides multi-destination send and broadcast services; also provides various forms of notification (acks) when messages are sent out and/or received at the destination.
- **Pattern Naming:** Provides a way of naming patterns. Multiple instances of the same pattern can be instantiated in the network.
- **Policy Manager:** Implements a number of policies for the framework. Also provides hooks for a network manager to various other modules which might support policy definitions.

The developers of the framework have also defined a number of optional services, that might be useful or even necessary on some platforms. These are not standardized and are not expected to be available on all platforms. Samraksh is not responsible for providing these services, however we might provide this service as a library or specific implementations for some platforms.

- **Time Stamping:** This service provides timestamps for events. The basic idea of this service is to translate a event that happened on a (direct) neighboring node to a corresponding time on a given node. That is provide a basis for synchronization of times when time synchronization is not available already. Two variations are defined a explicit sender based and implicit receiver based. In the explicit sender based version, a sending node marks a Event Time, in some time unit, puts it into a packet and sends it to a neighbor. The receiver node takes a timestamp when this packet is received on the lower levels. If the delays between the sending node taking the timestamp and the receiver node taking its timestamp can be accomplished then the event times across the two nodes can be synchronized. In implicit receiver based timestamping, the sending part is absent. The receiver is expected to know the rate or schedules at which an packet sending event is supposed to happen on a neighbor and simply expects packets according to that schedule. The implicit timestamping service simply takes a

timestamps of all incoming packets and sends them to the user of the service which can then parse these timestamps to arrive at time mapping between the two nodes.

- **Neighbor Time:** This services provides a given neighbor's time corresponding to local time. This is also called as network time synchronization service. This is expected to build on top of the timestamping service. However, specialized radios/waveforms may be able to implement this all by themselves.
- **Location:** Provides the geographical coordinates of a node. This service will need to interface with some underlying hardware such as GPS or location systems.
- **Mobility Control:** Provides the ability to move the current node to a given location or set the motions towards a given direction. Useful for nodes that can move or for mobile simulations.
- **Global Name**: Provides a globally unique name for a node. Patterns or Apps might query this service to get a the nodes global name and use it as the identifies in its protocols and algorithms.

List of Key Interfaces

List of APIs:

- Implemented by platform Shim Layer provider
  - PatternBase
  - FrameworkShim
- Implemented by Framework provider (Accessible by the Patterns)
  - Framework_I
  - PatternNaming_I
  - PatternNeighborTable_I (As a library)
- Implemented by the Waveforms
  - Waveform_I
  - WF_LinkEstimator_I

List of Service APIs:

- Implemented by Framework
  - LinkEstimation_I
  - PotentialNeighborRegistry_I
  - NetworkDiscovery_I (default)
  - PolicyManager_I
  - NeighborTable_I
  - PatternNaming_I
- Implemented by external providers

- GlobalNodeName_I
- LocationService_I
- NeighborTime_I
- NetworkDiscovery_I (Overrides the default if available)

2 Getting Started

Tuscarora is a cross platform software system and can be used on most desktop systems. Tuscarora comes pre-integrated with ns3 simulator that can be used to test the users code. The NS3 is designed for POSIX compliant systems and hence using the installation script for simulator will require POSIX systems. However, beginning Summer 2016 Windows is expected to support a POSIX/ubuntu environment. The following section is written for Ubuntu system. Howeve, it should work on most POSIX systems.

Supported Platforms

Ubuntu 16.04 64-bit and Ubuntu 14.04 64 bit are the recommended platforms. The instructions should work on most 64-bit Ubuntu platforms with the following restrictions. 32-bit distributions are not support.

- Ubuntu 16.04 is proffered developmental platform.
- Ubuntu 14.10 is not a long term release and reached EOL in July 2015. This version is not supported.
- Ubuntu 14.04: There are 6 sub-releases here, 14.04, 14.04.1, ..., 14.04.5. All of them should work with no modifications.
- Other 64-bit Ubuntu versions starting at 12.04 should work, although Samraksh does not officially support anything older than 14.04.
- Any other distribution is discouraged. If you are planning to use any other platform, first check out the requirements for ns3-dce [1].

Installation

Installation from Scratch

1. **Extract the source Files:** Move the Tuscarora package to a new directory, say 'C2E'. The release file is a password-protected rar-archive, so you will need the `rar` package.

```
$ cd
$ mkdir C2E
$ mv Tuscarora-release-0.x-y.rar C2E
$ sudo apt-get install rar # if not already installed
```

```
$ rar x Tuscarora-release-0.x-y.rar
```

2. **Launch installation:** The previous step will create a Tuscarora directory, this is the root of the framework. Launch the 'install.sh' script under Tuscarora root directory. Since the script adds some environment variables, it has to be called within the same shell. Please note the "." before the command. If not, a new shell should be started after the installation completes in order to set the environment variables.

```
$ cd TuscaroraFW
$ . ./install.sh
```

3. **Installation:** The installation script will ask for your 'sudo' password for installing some dependencies. Enter your password. The script will download and install the required ubuntu dependencies first. Next, it will proceed to download and install bake, ns3, and dce from the nsnam repositories. Ns3 and dce will be installed under a directory called 'dce' under your home directory; let's call this the $DCE_DIR directory. Next the script will make some symbolic links between the Tuscarora source directory and the $DCE_DIR. Finally, if all previous steps work fine, it will run some existing modules, and will return the status of execution.

Upgrading from Previous Release

To upgrade from an existing Tuscarora installation following the procedure below.

1. **Remove and Replace Tuscarora Directory**: Follow the path, where Tuscarora was installed, say C2E, and replace the Tuscarora directory with the contents of the new package.

```
$ cd C2E
$ rm -r Tuscarora
$ sudo apt-get install rar # [if not already installed]
$ rar x Tuscarora-release-x.y-z.rar
$ cd Tuscarora/TuscaroraFW
```

2. **Reinstallation**: Before we can install the new version, the ns3/dce patches from previous version should be removed first. To do this:

```
$ cd Tuscarora/TuscaroraFW
$ ./install.sh -u # [this will remove the patches]
$ ./install.sh
```

If at any stage you encounter error, please contact support.

Validating the installation

Finally, once the installation finishes, you can validate the installation by running the command "./validate.sh" from the TuscaroraFW directory followed by "-p platform" option. This would run a number of predefined

validation tests for the given platform and would print the result of the tests as FAILED or PASSED on the screen. If any of the tests fail, please contact support. Running the validate script with the '-h' option prints the list of validation tests available.

Executing Tuscarora Framework Modules

The 'runOrDebug.sh' in the root directory is the primary script to execute the modules and tests under framework. The script cleans, builds, runs and collects the output for the 'test-name' specified. Run the script without parameters for information on supported options. Run the script with the test-name and '-h' option to get test-specific help.

Example Executions

0. Getting help for the usage of runOrDebug.sh, to find the list of available tests.

    $ ./runOrDebug.sh

1. Getting help for Gossip test, to find the list of all test parameters with their default values.

    $ ./runOrDebug.sh -h Gossip

2. Running the Gossip pattern test for 60 secs on a 100 node network.

    $ ./runOrDebug.sh Gossip -- RunTime 60 Size 100

3. Running the Gossip pattern test inside the gdb to debug for 100 nodes (and a default duration of 6 secs).

    $ ./runOrDebug.sh -d Gossip – Size 100

4. Running the Framework Interface test 'FI' with Periodic Link Estimation with a dead neighbor period of 2 seconds and a Link Estimation Period of 1Hz using Global Network Discovery.

    $ ./runOrDebug.sh FI -- LinkEstimationType periodic LinkEstimationPeriod 1000000 DeadNeighborPeriod 2 DiscoveryType global

5. Running the Framework Interface test 'FI' with Schedule Aware Periodic Link Estimation, a Link Estimation Period of 10Hz, and 2-hop Long Link Network Discovery.

    $ ./runOrDebug.sh FI -- LinkEstimationType scheduleAwarePeriodic LinkEstimationPeriod 100000 DiscoveryType longlink2hop

6. Running the Gossip pattern test with 10 nodes with a waveform configuration file.

    $ ./runOrDebug.sh Gossip – Size 100 WFConfig ${TUS}/dceln/wf-config.cnf

7. Running the Gossip pattern test with a tracefile based mobility model. The tracefile needs to be in the ns-2 mobility trace file format.

```
$ ./runOrDebug.sh Gossip – Size 10 RunTime 10 Mobility TracefileMobilityModel Tracefile
${TUS}/TraceFiles/Simple-Size10-10secs.tr
```

For more information on running Tuscarora on various platforms, testing methodology and configuration options see Chapter 7.

Execution Output Files

The runOrDebug script creates outputs in the directory "dceln". This directory is a symlink to the main dce execution directory, which is located at "$HOME/dce/sources/ns-3-dce".

Under this directory you will find a number of outputs generated:

- time.output: This file summarizes the high level information about the test such seen from the operating system such as the command being run, its terminating condition, the time it takes to complete, the amount of system resources used, etc.
- simulation_description: This file details various features of the simulation either set by the command line arguments or used as the default values.
- exitprocs: This file stores information about the execution process.
- CourseChangeData.txt: This file stores information about the mobility. More specifically, for each change in the velocity of a node, a line of record indicating the time instances, node's ID, node's new velocity and node's location at that time instant is stored.
- elf-cache: The compiled program files are stored under this directory.
- file-x: Directories such as files-0, files-1, etc., store the file system of each simulated node. Each of these directories have a simulation_desciription file identical to the one located at "`\$HOME/dce/sources/ns-3-dce`". In addition to that, the generated data at each node is stored in these directories. "Configuration.bin" stores the configuration parameters in binary format. "linkestimation.bin" stores the link estimation state in binary format. The output for each node can be found under file-*/stdout. This file stores the outputs of the "Debug_Printf" macro or the "printf" to "stdout" of each node during the simulation.

Directory Structure

- Config: Sample waveform configuration files for simulation
- Doc: Documentation
- Install: Scripts to automate installation of Tuscarora
- Include: Header files / API specifications

- Lib : Platform neutral library modules
- ns-3-dce: Simulation scripts to run under dce
- Patchs: Patches to DCE module to enable additional features
- Patchs.ns3: Patches to the ns-3 modules.
- Platform: Platform-specific library modules
- Scripts: Various utility scripts
- Src: Framework and Pattern source files
- Tests : Tests for modules, layers and patterns

Compilation, Linking and Execution Details

Tuscarora's framework links with DCE source tree through the $TUS/ns-3-dce/myscripts/TuscTest folder. A link for this folder is created in the "$DCE_DIR/sources/ns-3-dce/myscripts". This folder includes c++ programs defining ns3 simulation environment to be used in the simulation. The defaults setup of the runOrDebug.sh script uses the "tuscarora-test.cc" program that sets ns3 and DCE setup for the simulation depending on the run time arguments. It is compiled through waf system of the DCE and its binary is placed under "$DCE_DIR/build/bin".

While executing a program using the DCE framework, the test program written using Tuscarora is compiled as a dynamically linked shared library and loaded into the DCE environment. Notionally, DCE provides the "OS" and the tests written using Tuscarora are like "application" executed on that OS. DCE provides these "OS services" by utilizing either ns-3 modules or by using the native linux services.

One difference between DCE and an actual OS is that DCE "runs" multiple copies of the application, one per node, while an actual OS would run only one copy.

The actual test/simulation that is run on top of DCE can be controlled by the Test parameter provided to the runOrDebug.sh script. "TuscaroraFW/Tests" folder includes a number of programs intended to be used as test applications. Each such program has its own main() function, which is where a given node's execution begins.

Finally, the source code for the framework and the source code for the sample patterns provided reside in "TuscaroraFW/Src". As explained above, depending on the test program being run, these individual tuscarora modules are compiled and linked into DLL binary by the runOrDebug.sh script and an associated Makefile and are placed under "$DCE_DIR/sources/ns-3-dce/build/bin_dce". These binary DLLs are loaded into a ns3-dce binary executable which is usually called "tuscrora-test".

A typical simulation consists of following run-time threads:

- Initially, a single thread starts and runs the code in tuscarora-test. This thread act as the overall manager of the simulation, and does coordination between nodes, and also runs ns3 module code (simulating the waveform(s), the channel(s), nodes, events, etc.).
- With the start of the ns3 simulation, for each node a separate thread is created, which runs test program specified as the run time argument.

- The test program running on each node creates instance(s) of the pattern(s), an instance of the framework and instance(s) of the waveforms layer are created.

3 Inter Layer Messaging and Build Options

Tuscarora provides a flexible deployment procedure in which modules depicted in Fig. 1.1 can run in their own processes or several of these modules can be bundled to share a single process. In order to enable deployments on various platforms and to provide a common architecture for single process or multi-process setups, Tuscarora defines a communication module that translates calls between various layers. We colloquialy refer to this as the "communication shim layer" or just as "shim"; A shim layer is required between the applications and the patterns, between the patterns and the framework, and between the framework and waveforms as shown in Fig. 3.1.

Tuscarora APIs are designed with function syntax(aka method calls) and named with message passing semantics. The job of the shim layer is to translate a function call from within one layer to a function call on another layer, using asynchronous message passing between layers. For example, a pattern makes a call to a shim layer object and the shim layer object calls the framework, which might be part of the same process space or might be running as a different process. For multi-process environments, this shim layer should convert the function calls to messages or some other inter-process mechanism (such as pipes) and send them to the corresponding modules on the other layer. Similarly, the shim layer parses and converts received messages back into function calls and invokes the corresponding functions on the receiving layer.

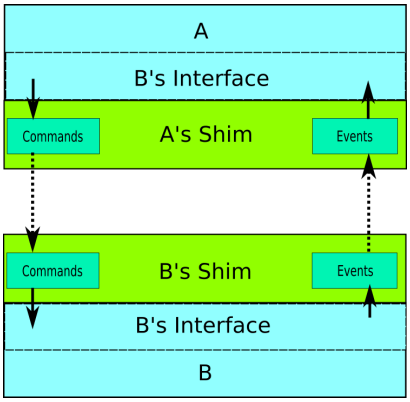


Figure 3.1: Generic example of Shims between Layer A and Layer B, where A is user of B

In a single-process environment, a shim layer can be implemented using one of the many different options available, such as synchronous blocking function call, asynchronous non-blocking function calls, messaging passing or signaling. In general, the implementation of these shim layers can be platform-specific and when porting Tuscarora to a new platform at least some of the shim layer modules may need to be reimplemented. Tuscarora interfaces are designed with 'Commands' that are received by a module and 'Events' that are invoked or sent from a module, i.e., Tuscarora interfaces are bidirectional. For example, consider two layers , A and B, as shown in Fig. 3.1. If layer A wants to invoke a command on layer B, then a communication

shim is need both on A's side to send out the Commands and to receive B's Events. On B's side we need a shim to receive the commands and to send out the Events. However, sometimes (mostly in single process case), the shim on either A or B can be skipped, if they have a common shared memory location. The Shim on A's side should implement B's interfaces, both Commands and Events. That is A's shim translates from B's interface to some common agreed upon communication standard between A and B, and B's shim translates back this communication standard back to B's interface specification and invokes B. Therefore A's shim looks like B to A.

Tuscarora package provides shim layers designed for the DCE environment as well as linux multi-process environments.

App-Pattern Shim Layer Modules

Applications can be written towards the interface definitions of one or more patterns. Depending on the platform, the object implementing the interface could be the pattern itself or a pair of shim layers that pass the call to the intended pattern. Fig. 3.2 depicts setups for multi-process and single process environments.

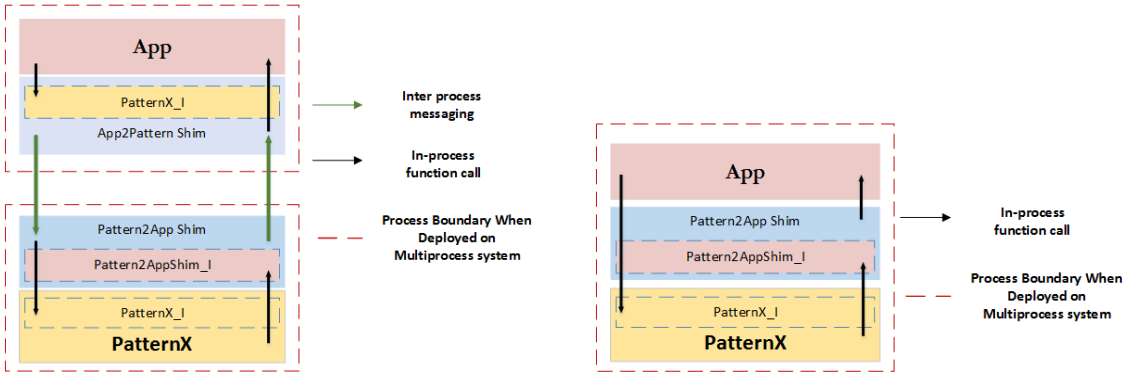


Figure 3.2: App-Pattern interface for multi process and single process environments

Multi process Shim Setup

In multi process environments, `App2PatternShim` and `Pattern2AppShim` `App2FWPShim` pair carry calls from each side of the process to the other side. Internally, these calls are converted to serialized messages on the sending side and carried over using a socket communication interface. Once received, the message is deserialized and the destination layer is called with the variables inside message.

Tuscarora provides `SocketCommunicatorClientBase` and the `SocketCommunicatorServerBase` classes to automate server-client connection process, and `GenericSerializer` and `GenericDeSerializer` templated classes that help serialization and deserialization process for the shim classes.

`SocketCommunicator` library simply provides functions to send serialized buffers to the other side and on receiving event it directs read contents to a virtual ***Deserialize*** function implemented by the corresponding shims.

A generic serializer object is simply constructed by listing the list of variable types and the corresponding variables and it creates a serialized buffer that contains copies of the variable values. Similarly, a generic

de-serializer deserializes a variable sized buffer by copying its content to individually created variables in the order specified in its constructor. They also accepts pointers in which case the value of the preceding variable is assumed to be the size of the object pointed by the pointer.

We'll elaborate this process with an example call to SendData from the "FanOutFWP" application to "FWP" pattern.

`App2FWPShim`'s ***SendData*** uses a GenericSerializer object to create a serialized buffer consisting of a calltype, and the function inputs, namely *AppId_t*, the variable sized message and a nonce that is used to identify the message between the app and the pattern. Then `App2FWPShim` calls the ***SendMsg2Server*** function implemented by the base class, `SocketCommunicatorClientBase`, to send the serialize message to the corresponding server.

Upon receiving the signal ,the message contents are read from the socket by `FWP2AppShim`'s base class `SocketCommunicatorClientBase` and ***Deserialize*** function of `FWP2AppShim` is invoked. ***Deserialize*** function, first reads the calltype from the received buffer. Depending on the calltype, the rest of the buffer is deserialized into individual variables. In the case of the calltype being *APP2FWP_Call_Send*, the rest of the buffer is deserialized into an *AppId_t*, a variable sized message and a nonce using a `GenericDeSerializer`. Using these variables, the "FWP" pattern's ***SendData*** function is invoked. Hence, the call from "FanOutFWP" to "FWP" is completed.

Single process Shim Setup

In single process setups, such as the ones DCE platform uses, "App2PatternShim" disappears and the function calls directly invokes implementations in the "PatternX" as depicted in Fig. 3.3. "Pattern2AppShim" only acts as a dispatcher or a multiplexer by invoking applications registered to the pattern.

We'll elaborate this process with an example call to ***ReceiveUpdatedGossipVariable*** from the "Gossip" pattern to "BasicGossipApp".

During the initialization, "BasicGossipApp" registers its delegate used for receiving updates by invoking ***RegisterGossipVariableUpdateDelegate*** method of the "Gossip" pattern. "Gossip" redirects delegates to the `Gossip2AppShim`, where they are stored for further use.

Whenever there is an update to the internal variable being, "Gossip" invokes ***ReceiveUpdatedGossipVariable*** in `Gossip2AppShim`, which in turn invokes the list of delegates stored.

Pattern-Framework Shim Layer Modules

In order to facilitate the Pattern development, Tuscarora package comes with a common base class library, namely `PatternBase`, that automates connections in the Pattern-Framework interface. For patterns deriving from it, `PatternBase` choses the shim classes required by the platform, initiates the communication medium with the Framework and provides patterns an abstracted pointer to the Framework through a member variable called *FRAMEWORK*. Thus, for convenience Patterns are expected to inherit from `PatternBase`, although it is not strictly necessary,

The implementation of `PatternShim` is platform-specific since implementing the Framework_I interface to communicate with the actual framework will depend on the types of communication mechanisms available

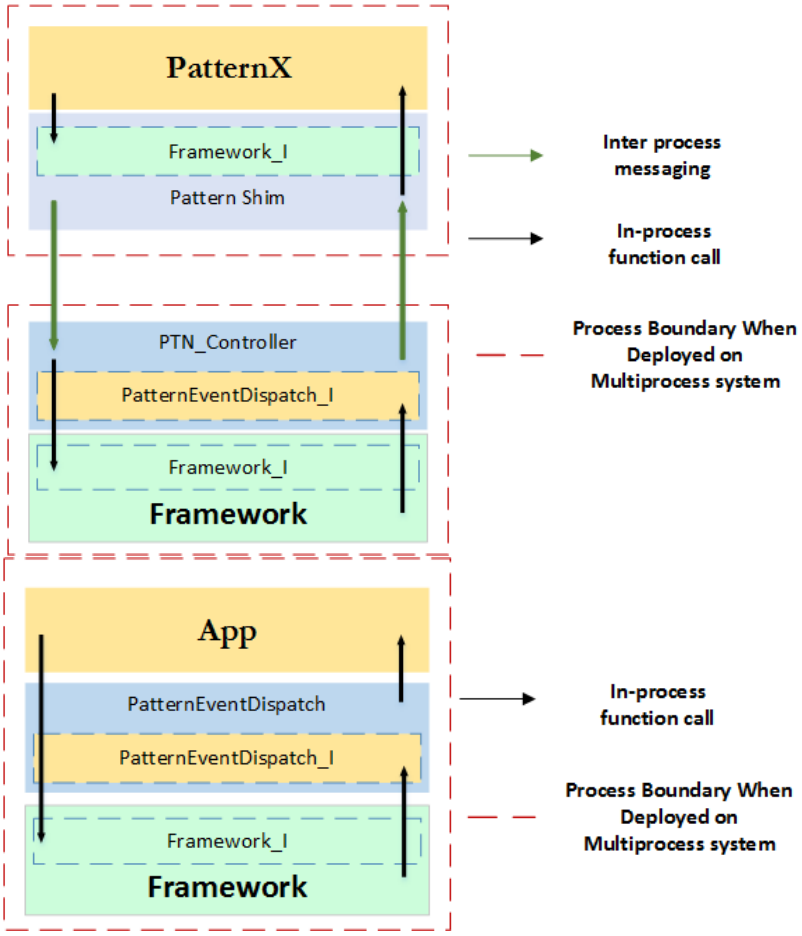


Figure 3.3: Pattern-Framework interface for multi process and single process environments

on a given platform. PatternBase also receive the `Framework_I` Events and convert them back into delegate calls for the Patterns. The corresponding Shim layer on the framework side is implemented by two classes– one for Commands by the `FrameworkShim` and one for Events by the `PatternEventDispatch`. Fig. 3.3 shows the flow of messages and bindings for this scenario..

In the DCE, the shim layers are implemented using synchronous function calls, since they are part of the same process space as shown in Fig. 3.3. PatternBase gets a pointer directly to the FrameworkShim class (which is the shim class on the framework side) and invokes commands on it. Similarly the PatternEvent-Dispatch module gets access to the Delegate objects in PatternBase and directly invoke them to implement the Events.

Framework-Waveform Shim Layer Modules

For the shim layer between the Framework and Waveform, the `WF_Controller` implements the commands in the WF_I interface and "Waveform Event Receiver" module implements Events in WF_I interface.

On the waveform side the commands are directly provided by the modules that implement the WF_I (i.e no shim in the forward direction in necessary) and the Events part of the WaveformBase shim are implemented `AsyncEvent_Special` module.

Once again, since in DCE we are in a single process address space, the shim modules directly get access to the objects on the other side and invoke them. The multi process and single process case interfaces are depicted in Fig. 3.4.

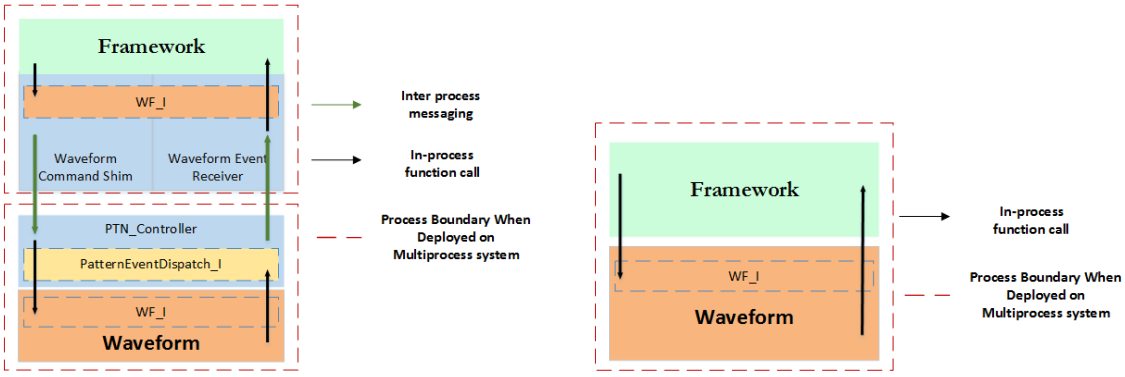


Figure 3.4: Framework-Waveform interface for multi process and single process environments

NS3-DCE

For portability reasons, this release uses the Direct Code Execution (DCE) [1] environment for running simulations. DCE is a module that enables ns-3 simulator to run existing source code that is targeted towards POSIX environments. The executable for each node runs on a separate process and these processes are coordinated through another process running ns3 simulator. For more information see the DCE documentation [1].

4 Application Development

Rather than building up applications around the idea of peer to peer communications, Tuscarora adopts the concept of using information distribution strategies provided by ASNPs. As shown in Fig. 1.1, applications are positioned on top of the ASNP's in Tuscarora system architecture. Applications should use one or more of these underlying ASNPs for it information distribution and gathering needs. Hence, it is critical for an application developer to develop her applications around information flows rather than P2P communication. This isolates the information flow needs from the rest of the application. If this need cannot be fulfilled with the existing ASNPs, the information flow can be implemented as a new ASNP.

In this chapter, we present the available ASNPs as well as envisioned ASNPs for application developers and provide examples of how applications can interact with ASNPs. The Application Writters Interface (AWI) is the namespace that exposes the services of the patterns to the applications. The core API for the patterns are defined in their respective interface definitions PatternX_I. For full API documentation please refer to the companion document "Tuscaroro API documentation".

Patterns Available in Tuscarora Package

Out of the box, Tuscarora package comes with 2 ASNPs,

- Gossip: Gossip pattern stores some common information and distributes that information by infrequent random updates in the network. Internally stored Gossip information is updated with the received information only if the received data is larger than the existing one based on a customizable comparator. For slowly changing data, the last state of the data is eventually distributed to all connected nodes.
- FWP: The Flooding with Pruning (FwP) pattern distributes a dataset generated (or aggregated) at a node to all nodes in some region. The addition of pruning ensures efficient distribution of the data by limiting retransmission on the nodes that does not improve the nodes that do not have the information.

Gossip

Gossip is a templated class that allows distribution of various data types in the network. An application writer has to specify a data type of his choice, `GOSSIPVARIABLE`, and can optionally specify a corresponding comparator, `GOSSIPCOMPARATOR`, which implements a ***LessThan*** and a ***EqualTo*** functions. In the case of receiving updates from other nodes or from local applications, this comparator is used. Comparator specification can be skipped in which specified gossip variable's ***operator==*** and ***operator<*** are used.

`Gossip_I` interface defines the following:

- typedef Delegate<void, GOSSIPVARIABLE&> GossipVariableUpdateDelegate_t : is a delegate storing the callback function pointer that is invoked when there is a change in the GOSSIPVARIABLE. This callback function should have void return type and should accept a single input of type `GOSSIPVARIABLE`.
- void RegisterGossipVariableUpdateDelegate (GossipVariableUpdateDelegate_t* _gvu_del) : is the method to receive the delegates sent to the Gossip pattern. Applications that desire to receive updates on the gossip variable use this method to register to the incoming information flow using this method. The Gossip pattern stores delegates it receives with this method and invokes them whenever there is an update on *gossip_variable*.
- void UpdateGossipVariable(GOSSIPVARIABLE& newgossipVariable)) : is a method used by the applications to update the underlying `GOSSIPVARIABLE` of the pattern. This method lets a application to request changing the variable being gossiped with a newVariable. The newgossipVariable is compared with the existing one.
  - if the existing variable is larger(based on the comparator), no further action is taken.
  - if the new variable is larger(based on the comparator), the gossip variable is updated, and all registered delegates are invoked with the new `GOSSIPVARIABLE`.

Flooding with Pruning

The interface for FWP pattern is defined by `Fwp_I`, which declares the following:

- typedef Delegate<void, void* , uint16_t > AppRecvMessageDelegate_t : is a delegate storing the callback function pointer that is invoked when data is received. This callback function should have void return type and 2 inputs, namely a void* pointer and a uint16_t type specifying the size of the data.
- void RegisterAppReceiveDelegate(AppId_t _app_id , AppRecvMessageDelegate_t* _gvu_del) : is the method to get the delegates for the `FWP` pattern. The delegate is stored by the pattern and is invoked with the incoming data from the network.
- void Send (void *data , uint16_t size) : is a method used by the applications to start a network-wide flood with the data specified by the pointer *data* and *size*.

Patterns Envisioned

The following ASNP's are envisioned and formally defined by Samraksh however, they are not yet available in this release. Here we present short descriptions. For details, please refer to the Framework Rationale document.

- COP: COP pattern is designed to provide a common operating picture of the entire network by distributing and collecting information about all the nodes in the network such as their location. In order to satisfy scalability, Samraksh envisions this pattern to maintain faster updates for nearby nodes and slower updates for farther away nodes.
- Census: The objective of Census is to collect information about resources of some type deployed in the network, or to aggregate sensor values, by querying each node in the network, ideally without

missing out on any node and without double counting any response. The pattern is designed to work even when the underlying network topology is dynamic and may also be subject to temporary partitioning. Census examples in a military mobile ad-hoc network include counting the available artillery units, ammunition, food, fuel, etc. A Census query can be sent into the network from any node in the network, to collect some aggregate statistic (such as count, max, sum) of the network nodes.

- Exfiltration: Exfiltration patterns maintain a spanning tree across the network, which can be traversed to the root in order to accomplish exfiltration. When a link-state changes, the spanning tree is updated quickly and ideally with minimal traffic. Samraksh proposes Inverse-Wave Based Exfiltration (IWBE) over tree based techniques due to the dynamics of maintaining the spanning wave in the presence of link state changes are significantly better.

Application Example

In this section we will work through an example of creating and testing an application to be used with existing Patterns in the Tuscarora system. Our example, "BasicGossipApp", is an application that distributes and maintains a common gossip variable throughout the network. The variable is incremented on the root network of the network at periodic intervals and through the use of Gossip pattern the variable updates are distributed to the network.

For the sake of generality, we want to keep the Gossip variable to be templated. The source file for this application is available at "`TuscaroraFW/Apps/Gossip/BasicGossipApp.h`".

Initialization

The constructor

- initializes *gossip_variable*

```
gossip_variable = 0;
```

- creates a one-shot timer that is set to start execution when triggered,

```
timerDel = new TimerDelegate (this, &BasicGossipApp<GOSSIPVARIABLE,
    GOSSIPCOMPARATOR>::InitiateApp);
startTestTimer = new Timer(2000000, ONE_SHOT, *timerDel);
```

- if the node is the root node, creates a periodic timer that is set to call ***IncrementGossipVariable*** function when triggered,

```
if(MY_NODE_ID == 0){
  updatevarDel = new TimerDelegate (this, &BasicGossipApp<GOSSIPVARIABLE,
      GOSSIPCOMPARATOR>::IncrementGossipVariable);
  updateVariableTimer = new Timer(2000000, PERIODIC, *updatevarDel);
}
```

- gets a pointer to the gossip interface

```
gossip = GetApp2GossipShimPtr<GOSSIPVARIABLE, GOSSIPCOMPARATOR>();
```

- creates the delegate that is used to receive updates from the Gossip Pattern

```
    recvUpdateVarDelegate = new (GossipVariableUpdateDelegate_t_1)(this, &
        BasicGossipApp<GOSSIPVARIABLE,GOSSIPCOMPARATOR>::
        ReceiveGossipVariableUpdate);
  };
```

Starting

The application gets the start signal when its ***Execute*** function is called. This method starts the timer that triggers application's initiation.

```
void Execute(RuntimeOpts *opts){
  //Delay starting the actual application, to let the network stabilize
  //So start the timer now and when timer expires start the pattern
  Debug_Printf(DBG_PATTERN,"About to start timer on node 1\n");
  startTestTimer->Start();
};
```

Initiation

The application initiates its operation by registering its reception delegate. The root node also starts its timer that triggers updating the *gossip_variable*.

```
void InitiateApp(uint32_t event){
  Debug_Printf(DBG_PATTERN, "GossipTEST: Starting the Gossip Pattern...\n");
  gossip->RegisterGossipVariableUpdateDelegate(recvUpdateVarDelegate);
  if(MY_NODE_ID == 0){
    updateVariableTimer->Start();
  }
};
```

Updating Gossip Variable by the Root Node

Updating the gossip variable is simply done by incrementing the *gossip_variable* and sending this update to the pattern.

```
void IncrementGossipVariable(uint32_t event){
  Debug_Printf(DBG_PATTERN, "GossipTEST: Manually incrementing gossip variable!
      Previous Value = %d...\n", gossip_variable);
  gossip->UpdateGossipVariable(++gossip_variable);
};
```

Receiving Updated Gossip Variable

The application simply stores the received *gossip_variable*.

```
void ReceiveGossipVariableUpdate(GOSSIPVARIABLE& updated_gossip_variable){
  Debug_Printf(DBG_PATTERN, "GossipTEST: ReceiveGossipVariableUpdate
      gossip_variable = %d, updated_gossip_variable = %d ...\n", gossip_variable,
      updated_gossip_variable);
  gossip_variable = updated_gossip_variable;
}
```

5 Pattern Development

The Pattern Writters Interface (PWI) is the namespace that exposes the services of the framework to the pattern.

The core API for the framework is defined in Framework_I. Additionally, the PatternNeighborTable_I exposes the customized neighborhood to the patterns. A Pattern can utilize the all the methods of the Framework_I and PatternNeighborTable_I to implement the primitives of the pattern.

For full framework API documentation please refer to the companion document "Tuscaroro API documentation".

Pattern Libraries

In order to make certain "plumbing" functions easier, functions such as registering the pattern with the framework, bootstrapping the communication between the pattern and the framework (depending on the platform and type of deployment), we have defined a `PatternBase` helper class. `PatternBase` provides a member variable called FRAMEWORK which provides the functions available on the framework side and defines virtual functions that must be implemented by a Pattern, to handle the Events generated by framework. That is, the PatternBase class defines the shim layer for the Patterns and its implementation would vary from platform to platform.

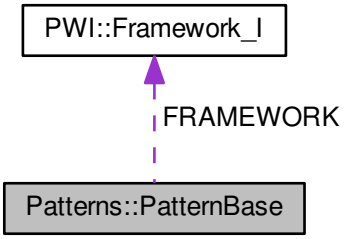
The chapter describes the `PatternBase` class and then explains how to drive from the `PatternBase` class to implement a full-fledged Pattern use case.

Patterns::PatternBase Class Reference

Defines the abstract base class for patterns.

```
#include <PatternBase.h>
```

Collaboration diagram for Patterns::PatternBase:



Public Member Functions

- **PatternBase** (PatternTypeE type, char _uniqueName[128])
- virtual bool Start ()=0

  Starts the pattern, must be implemented by each pattern.
- virtual bool **Stop** ()=0
- virtual ~PatternBase ()

  Virtual destructor.

Public Attributes

- RecvMessageDelegate_t * **recvDelegate**
- NeighborDelegate * **nbrDelegate**
- DataflowDelegate_t * **dataflowDelegate**
- ControlResponseDelegate_t * **controlDelegate**

Protected Member Functions

- virtual void **ReceiveMessageEvent** (FMessage_t &msg)=0
- virtual void **NeighborUpdateEvent** (NeighborUpdateParam nbrUpdate)=0
- virtual void **DataStatusEvent** (DataStatusParam notification)=0
- virtual void **ControlResponseEvent** (ControlResponseParam response)=0
- void **RegisterPatternDelegates** (PatternId_t pid, PatternTypeE _type)
- void **Handle_RegisterResponse** (ControlResponseParam response)

- void **RandomLocalSpray** (PatternId_t pid, FMessage_t &msg, SprayTypeE spraytype, bool israndomselection, Patterns::PatternNeighborTableI &ptnNbrTable, uint16_t nonce)
- bool **RandomlySelectNeighbor** (NeigborContainerType &selectedNeighborList, Patterns::PatternNeighborTableI &ptnNbrTable, UniformRandomInt *rand=NULL)
- void **Send2SelectedNeighbors** (PatternId_t pid, NeigborContainerType &selectedNeighborList, FMessage_t &msg, uint16_t nonce)

Protected Attributes

- char **uniqueName** [128]
- PatternId_t **PID**
- Framework_I * **FRAMEWORK**
- bool **registered**
- PatternRequestStateE **requestState**
- PatternStateE **patternState**
- uint32_t **n_ExpectedFrameworkResponses**

Detailed Description

Defines the abstract base class for patterns.

The documentation for this class was generated from the following file:

- TuscaroraFW/Include/Interfaces/Pattern/PatternBase.h

Pattern Example

In this section we will work through an example of creating and testing a pattern in the framework. Our example will be a gossip-like protocol. Gossiping is a class of protocols where a particular network state is updated in the network using periodic/random continuous messaging. Gossip-like protocols can be implemented using either broadcast mechanisms or unicast mechanisms. In this example we will implement a unicast-based Gossip (which is closer to Tuscarora's design philosophy). The state being kept up to date is simply an integer number that increases monotonically. The nodes in the network "gossip" with each other to make sure they have the highest/latest number. The code for this example can be found in the release in the directories "`TuscaroraFW/Src/Pattern/Gossip`" and "`TuscaroraFW/Tests/Patterns/Gossip`".

Let's define some basic primitives for the Gossip pattern; then we show how to implement these primitives.

1. The pattern sends a status message to selected neighbors at random intervals, with some average period—say, 200ms. It uses a uniform random number generator and the events module to achieve this.
2. New discovered neighbors are selected to be notified with the next status message.

3. If the status info of a received message is older than the stored information, its sender is selected to be notified with the next status message.
4. If no new neighbors were selected to be notified, one neighbor is selected randomly from the set of neighbors.
5. The list of selected neighbors is cleared after sending a status update.
6. If the packet send to a neighbor fails, that neighbor is selected again to be notified with the next status message.

Pattern Overview and Definition

Operation of the framework and the patterns are asynchronous. Pattern issues "Commands" to the Framework, through the PatternBase *FRAMEWORK* variable. For the responses of these "Commands" as well as changes in the pattern neighborhood, changes in the status of the previously send packets, and incoming packets to the pattern, the framework generates "Events" and sends them back to the Patterns.

It is advised that Patterns derive from `PatternBase` class, since it helps automating registration and bootstrapping procedures of patterns. However, a pattern writer could choose to implement his Pattern from scratch. `PatternBase` also provides standard methods that help registering the pattern with the framework. In addition to that, it defines virtual functions, one corresponding to each type of "Event" generated by the Framework, namely

- ***ReceiveMessageEvent*** for handling message receptions,
- ***NeighborUpdateEvent*** for neighbor notifications,
- ***DataStatusEvent*** for handling data notifications, and
- ***ControlResponseEvent*** for handling control responses

These functions should be implemented by patterns derived from the `PatternBase` class.

The code below shows a simple implementation for our `Gossip` pattern.

```
1  /*
2   * Gossip.h
3   *
4   *  Created on:  March 3, 2015
5   *      Authors: Mukundan Sridharan, Bora Karaoglu
6   */
7
8  #ifndef GOSSIP_H_
9  #define GOSSIP_H_
10
11 #include <Types/BasicTypes.h>
12 #include <Interfaces/Pattern/PatternBase.h>
13 #include <Lib/Math/Rand.h>
14 #include "Lib/PAL/PAL_Lib.h"//include the PAL layer
15 #include "Lib/DS/AVLBinarySearchTreeT.h"
16
17 using namespace PWI;
18
```

```
19 namespace Patterns {
20
21 typedef AVLBSTElement<NodeId_t> GossipNeigborContainerTypeElement;
22 typedef AVLBST_T<NodeId_t> GossipNeigborContainerType;
23
24 class GossipLinkComparator: public LinkComparatorI {
25 public:
26   //GossipLinkComparator();
27   bool BetterThan (Core::LinkMetrics& A, Core::LinkMetrics& B){
28    return (A.quality > B.quality);
29   }
30 };
31
32 class Gossip : public PatternBase{
33   uint32_t currentStatusId; //Status of the gossip protocol
34   FrameworkAttributes fwAttributes;
35   uint32_t nonce;
36   //Pattern Specific
37   UniformRandomInt *rand; //A random number generator
38   Event *randEvent_SendUpdate;  //A pointer for the event class used to send random
       updates
39   Event *randEvent_UpdateVar; //A pointer for the event class used to update stored
       information
40   EventDelegate *eventDel_SendUpdate;//delegate for the random data send event.
41   EventDelegate *eventDel_UpdateVar;//delegate for the random data send event.
42
43   void RandomSendHandler(EventDelegateParam param); //Handler for the event
       generator used to send random updates
44   void UpdateVariableHandler(EventDelegateParam param);  //Handler for the event
       generator used to update stored information
45
46   PatternNeighborTableI *myNbrHood;
47   //GossipLinkComparator *myLinkComparator;
48
49   GossipNeigborContainerType SelectedNeighborList; //Set of selected neighbors to
       send a status update
50
51   FMessage_t& PrepareStatusUpdate();  //Broadcast your status
52   void SendMessage();       //Sends a message to the list of selected neighbors
53
54   bool SelectNeighbors(NodeId_t nbr); //Add a neighbor to the set of selected
       neighbors
55   void AdjustSelectedNeighbors(); //Make sure at least one neighbor is selected
56   void ClearSelectedNeighborList(); //Clear the list of selected neighbors
57
58   NodeId_t IterateThroughNeighbors(uint16_t table_index);
59   bool InitiateProtocol();
60   void Handle_AttributeResponse(ControlResponseParam response);
61   void Handle_LinkThresholdResponse(ControlResponseParam response);
62   void Handle_SelectDataNofiticationResponse(ControlResponseParam response);
63 public:
64   //Common to most patterns
65
66   Gossip(); //constructor
67   bool Start();  //Lets the test code or network admin start the pattern
68   bool Stop(); //Lets the test code or network admin stop the pattern
69
```

```
70    void NeighborUpdateEvent (NeighborUpdateParam nbrUpdate);
71    void ControlResponseEvent (ControlResponseParam response);
72    void DataNotificationEvent (DataNotifierParam notification);
73    void ReceiveMessageEvent (FMessage_t& msg);
74  };
75
76  } //end of namespace
77
78  #endif //GOSSIP_H_
```

The code below shows the definitions provided by the `PatternBase` class.

```
79  /*
80   * PatternI.h
81   *
82   *  Author: Mukundan Sridharan
83   *  This is an abstract base class for patterns to derive and implement, to make it
       easy for registrating with framework and to start them.
84   */
85
86  #ifndef PATTERN_BASE_H_
87  #define PATTERN_BASE_H_
88
89  #include <Types/BasicTypes.h>
90  #include <Types/FrameworkTypes.h>
91  #include <PAL/Delegate.h>
92  #include <Interfaces/PWI/Framework_I.h>
93  #include <Interfaces/Core/PatternNamingI.h>
94
95
96  using namespace PAL;
97  using namespace PWI;
98  using namespace PWI::Neighborhood;
99
100 namespace Patterns {
101   //A enum to keep track of Pattern Framework interaction state;
102   enum PatternStateE {
103     NO_PID,
104     GOT_PID,
105     REGISTERED,
106     EXECUTING,
107     ERROR,
108   };
109
110   enum PatternRequestStateE {
111     NONE_PENDING,
112     WAITING_FOR_CONTROL_RESPONSE,
113     WAITING_FOR_DATA_RESPONSE
114   };
115 ///Defines the abstract base class for patterns
116 class PatternBase {
117 protected:
118   /*
119    * @brief Returns a reference to the Pattern's custom neighbor table
120    *
121    * @param patternId Pattern's instance ID.
122    * @return PWI::Neighborhood::PatternNeighborTableI&. Reference to the patterns
```

```
      neighborhood table.
123
124   virtual PatternNeighborTableI& GetNeighborTable(PatternId_t patternId)  = 0;
125   */
126
127
128 protected:
129   PatternId_t PID;
130   Framework_I* FRAMEWORK;
131   bool registered;
132   PatternRequestStateE requestState;
133   PatternStateE patternState;
134
135   virtual void ReceiveMessageEvent (FMessage_t &msg)=0;
136   virtual void NeighborUpdateEvent (NeighborUpdateParam nbrUpdate) =0;
137   virtual void DataNotificationEvent (DataNotifierParam notification) =0;
138   virtual void ControlResponseEvent (ControlResponseParam response) = 0;
139
140   void RegisterPattern(PatternId_t pid, PatternTypeE _type);
141   void Handle_PatternIDResponse(ControlResponseParam response);
142   void Handle_RegisterResponse(ControlResponseParam response);
143
144
145 public:
146   RecvMessageDelegate_t *recvDelegate;
147   NeighborDelegate *nbrDelegate;
148   DataflowDelegate_t *dataflowDelegate;
149   ControlResponseDelegate_t *controlDelegate;
150   ////////////////// Registration and Instantiation Service ////////////////////
151   //PatternBase(PatternTypeE type);
152   PatternBase(PatternTypeE type, char uniqueName[128]);
153   ///Starts the pattern, must be implemented by each pattern.
154   virtual bool Start()=0;
155   virtual bool Stop()=0;
156
157   ///Virtual destructor
158   virtual ~PatternBase(){}
159 };
160
161 }//End of namespace
162
163 #endif /* PATTERN_BASE_H_ */
```

Pattern Flow

Before going into the implementation details, lets look at the flowchart in Fig. 5.1 that show how the pattern is initialized and is integrated with the framework. Fig. 5.1 is divided into 3 sections corresponding the state of the pattern; *UNREGISTERED*, *REGISTERED*, and *EXECUTING*.

First, the pattern object is constructed. This constructor initializes various parameters of a pattern. Next, a shim layer implementing platform specific functions is created. Our simple pattern is deriving from `PatternBase` and call `PatternBase`'s constructor explicitly that constructs and initializes the shim layer.

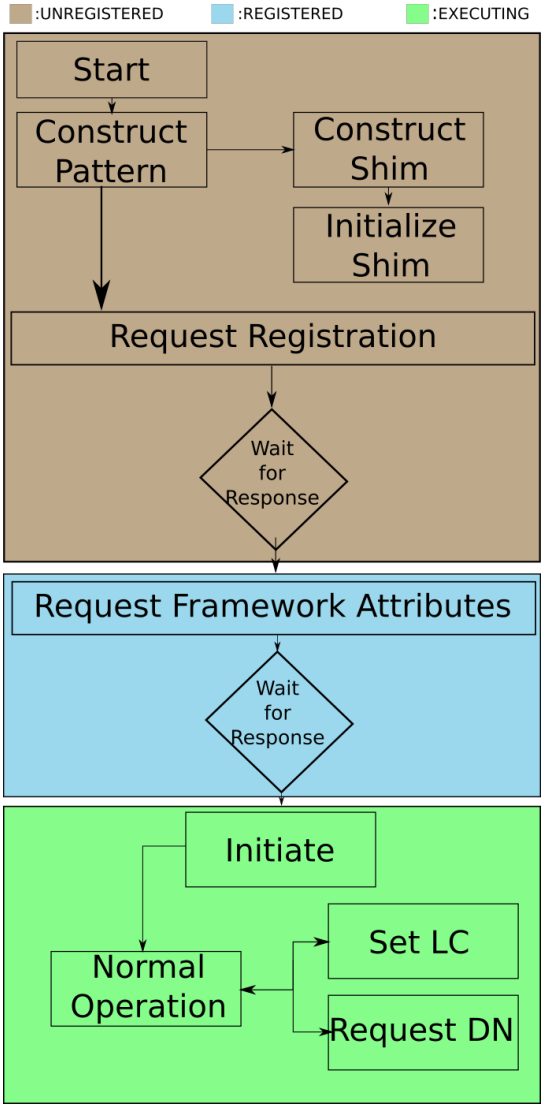


Figure 5.1: Flow chart of pattern initialization and registration

Initialization of the shim layer initializates the internal variables and stores the delegates used for callbacks back to the pattern, namely ***ReceiveMessageEvent***, ***NeighborUpdateEvent***, ***DataNotificationEvent***, and ***ControlResponseEvent***. However, it does not automatically initiate the registration of the pattern with the framework. "Patterns" explicitly request registration with the framework and handle framework's response through ***ControlResponseEvent*** function.

Gossip's PID is assigned by the framework. This PID is included in the positive response to the pattern's registration request, Receiving a positive response to the pattern's registration request, pattern changes state from *UNREGISTERED* to *REGISTERED*. At this stage, framework is queried for its attributes.

By receiving the response to framework attributes request, pattern's state changes to *EXECUTING*. In this state, the pattern initiates its operation and starts operating normally. In the beginning of its operation, a pattern can set its link comparator and subscribe to the data notifications of interest. However, these operations can be repeated anytime during the normal operation to change the link comparator or to change pattern's subscription to data notifications.

In the following sections, we will investigate how `Gossip` implements various functions for its operations.

Initialization

Patterns deriving from `PatternBase` initialize their shim layer through `PatternBase`'s constructor. `Gossip`'s constructor further initializes

- two event delegates that are used to schedule future events,
- a timer delegate that is used to resend messages in case a message gets lost before getting received by the framework
- a random number generator,
- a table that stores neighbors,
- a nonce and a boolean variables facilitating message passing between the pattern and the framework, and
- internal gossip variables for keeping the status of the gossiped information,

```
164 Gossip::Gossip() : PatternBase(Core::Naming::GOSSIP_PTN, (char*)"Gossip_1"){
165   Debug_Printf(DBG_PATTERN, "Gossip::Initializing\n");
166
167
168   //delegate handles to the events
169   eventDel_SendUpdate = new EventDelegate(this, &Gossip::RandomSendHandler);
170   eventDel_UpdateVar = new EventDelegate(this, &Gossip::UpdateVariableHandler);
171   eventDel_ReSendUpdate = new TimerDelegate(this, &Gossip::ReSendHandler);
172
173
174   //Creates a uniform random number with mean 200,000 and with range +/-100,000
175   //That is the values vary between 100,000 to 300,000
176   UniformRNGState _state;
177   _state.cmrgState.stream = MY_NODE_ID;
178   _state.mean = MeanUpdateInterval;
179   _state.range = RangeUpdateInterval;
180   rand = new UniformRandomInt(_state);
181
182   //Initialize pattern neighbor table
183   myNbrHood = new PatternNeighborTable(QUALITY_LC);
184
185   //Initialize variables facilitating message passing
186   nonce=1;
187   hasFWRejectedPacket = false;
188
189   //Initialize Gossip status
190   currentStatusId = 0;
191 }
```

Starting

`Gossip` starts registering itself with the framework with its ***Start*** method. In this method, `Gossip` initiates pattern registration.

The framework supports usage of both external and internal naming services for obtaining unique pattern IDs (PID). Patterns using framework supported naming service get their IDs from the framework with the registration response while if there is an external service, Pattens can get a PID using this external service prior to registration and include the obtained PID in their registration request.

Gossip does not have an external pattern naming service and its pattern ID (PID) assigned by the framework during registration. In its registration request, Gossip uses a null *PID* of 0 together with a unique string identifier of the pattern. The string identifier is used by the framework in generating a *PID*.

```
192 bool Gossip::Start(){
193   Debug_Printf(DBG_PATTERN, "Gossip::Start Starting gossiping \n");
194   Debug_Printf(DBG_PATTERN, "Gossip:: Sending the RegisterPattern Request to
          framework...\n");
195
196   FRAMEWORK->RegisterPatternRequest (PID, uniqueName, Core::Naming::GOSSIP_PTN);
197   ++n_ExpectedFrameworkResponses;
198   return true;
199 }
```

Control Response Event Handling

The Events related to the control plane are handled by the ***ControlResponseEvent*** function, which is declared as virtual by the `PatternBase` class and must be implemented by all patterns deriving from it.

`PatternBase` has two internal variables, *n_ExpectedFrameworkResponses* and *patternState*, which help keeping the state of the messaging interface between the patterns and the framework.

patternState can take values of *NO_PID*, *GOT_PID*, *REGISTERED*, and *EXECUTING*. It is updated by a pattern at each stage of the registration process.

Patterns expect different responses from the framework depending on their association status. In general, the ***ControlResponseEvent*** function implementation should filter out unexpected responses[1] and implement error handling for unexpected events from Framework.

- In *NO_PID* state, `Gossip` is only expecting a registration response. Registration is completed by calling `PatternBase`'s ***Handle_RegisterResponse***. Next, `Gossip` queries for the attributes of the framework.
- In *REGISTERED* state, `Gossip` expects a response to its query for the attributes of the framework. The response for this query is handled in ***Handle_AttributeResponse***, which stores the framework attributes and updates *patternState*.
- In *EXECUTING* state, the registration is completed and the pattern is operational. In this state the pattern receives responses to various directives that it passes to the framework.
  - *PTN_SelectDataNotificationResponse*: Framework responds to the pattern's selection of data notifications with this response. `Gossip` handles this response with ***Handle_SelectDataNofiticationResponse***, which checks the success or the failure of the request

[1] An example situation in which an unexpected response might be received from the framework occurs when a pattern crashes and restarts without the crash being detected by the Framework. In this case, framework can potentially continue sending responses to earlier requests by the crashed pattern instance.

and generate debug messages. In the case of a failure, ***The SelectDataNotificationsRequest()*** function is called to issue a new request to the framework to set the appropriate data notification masks.

- *PTN_SetLinkThresholdResponse*: Framework responds to the pattern's selection of link threshold with this response. `Gossip` handles this response with ***Handle_LinkThresholdResponse***, which checks the success or the failure of the request and generate debug messages. In the case of a failure, ***SetNeighborhoodandLinkComparator()*** function is recalled issuing a new request to the framework for link comparator and threshold.
- *PTN_AttributeResponse*: This is a notification indicates some change in framework's attributes. `Gossip` handles this response with ***Handle_AttributeResponse*** function similar to the *REGISTERED* state.
- *PTN_AddDestinationResponse*: This is the framework's response to a request to add a destination(s) to the list of destinations of a previous packet before it is sent out of the waveform. Not used by `Gossip`.
- *PTN_ReplacePayloadResponse*: This is the framework's response to a request to replace the payload of a previous packet before it is sent out of the waveform. Not used by `Gossip`.
- *PTN_CancelDataResponse*: This is the framework's response to a request to stop one of its previous packets before it is sent out of the waveform. Not used by `Gossip`.

```
200 void Gossip::ControlResponseEvent(ControlResponseParam response)
201 {
202   Debug_Printf(DBG_PATTERN, "Gossip:: Got a control response of type %d\n", response
        .type);
203   switch (patternState){
204   case NO_PID:
205   case GOT_PID:
206     if(response.type == PWI::PTN_RegisterResponse){
207         --n_ExpectedFrameworkResponses;
208       Handle_RegisterResponse(response);
209       FRAMEWORK->FrameworkAttributesRequest(PID);
210       ++n_ExpectedFrameworkResponses;
211     }else {
212       Debug_Printf(DBG_PATTERN, "Gossip:: ControlResponseEvent: My state is %d I got
            wrong response type %d\n", patternState, response.type);
213     }
214     break;
215   case REGISTERED:
216     if(response.type == PWI::PTN_AttributeResponse){
217         --n_ExpectedFrameworkResponses;
218         Handle_AttributeResponse(response);
219         InitiateProtocol();
220
221     }else {
222       Debug_Printf(DBG_PATTERN, "Gossip:: ControlResponseEvent: My state is %d, I
          got wrong response type %d\n", patternState,response.type);
223     }
224     break;
225   case EXECUTING:
226     switch(response.type){
227     case PTN_SelectDataNotificationResponse:
228           Handle_SelectDataNofiticationResponse(response);
```

```
229             --n_ExpectedFrameworkResponses;
230         break;
231       case PTN_SetLinkThresholdResponse:
232         Handle_LinkThresholdResponse(response);
233         --n_ExpectedFrameworkResponses;
234         break;
235       case PTN_AttributeResponse:
236           --n_ExpectedFrameworkResponses;
237           Handle_AttributeResponse(response);
238           InitiateProtocol();
239           break;
240       case PTN_AddDestinationResponse:
241       case PTN_ReplacePayloadResponse:
242       case PTN_CancelDataResponse:
243       default:
244         Debug_Printf(DBG_PATTERN, "Gossip:: ControlResponseEvent: My state is %d, I
              got wrong response type %d\n", patternState,response.type);
245         break;
246       }
247       break;
248       default:
249         break;
250   }
251 }
252
253
254 void Gossip::Handle_AttributeResponse(ControlResponseParam response){
255   FrameworkAttributes *atr = (FrameworkAttributes*) response.data;
256   fwAttributes = *atr;
257
258   if( patternState == REGISTERED) {
259     patternState = EXECUTING;
260   }
261   else {
262      Debug_Printf(DBG_PATTERN, "PatternBase::Pattern getting initiated without
           registration.\n");
263   }
264
265   Debug_Printf(DBG_PATTERN,"Gossip:: Handle_AttributeResponse: Framework supports %d
        waveforms, with max pkt size is %d .\n", fwAttributes.numberOfWaveforms,
       fwAttributes.maxFrameworkPacketSize);
266   for(uint i=0; i<fwAttributes.numberOfWaveforms; i++){
267     Debug_Printf(DBG_PATTERN, "Gossip::Handle_AttributeResponse: Waveform %d Id is %
         d\n", i, fwAttributes.waveformIds[i]);
268   }
269 }
```

Initiation

With the reception of a positive response to its registration request, `Gossip` initiates its operation with ***InitiateProtocol()*** function.

In this function, `Gossip` calls ***SetNeighborhoodandLinkComparator()*** to select (i) a link comparator type

defining the criteria for differentiating between links, and (ii) a threshold defining minimum acceptable limits for a link to be accepted in pattern's neighborhood. `Gossip` selects the link quality as the criterion to differentiate between links and sets its threshold to a quality level of 0.1.

Next, by calling ***RequestDataNotifications()*** function, `Gossip` notifies the framework about the type of acknowledgements that it is interested in, namely destination-oriented acknowledgments such as a successful (or unsuccessful) reception of the packet (indicated by *PDN_RECV_DEST_MASK*).

Finally, `Gossip` initiates its operation by starting two random event generators used for (i) periodically sending messages and updating the information (see the next section). These random event generators are stopped in `Gossip`'s stop routine, which stops the operation of the pattern.

```
270 bool Gossip::InitiateProtocol(){
271   //Initialize neighborhood and set link comparator.
272   SetNeighborhoodandLinkComparator();
273
274   //tell framework which notifications you are interested in
275   RequestDataNotifications();
276
277   Debug_Printf(DBG_PATTERN, "Gossip::Have configured everything succeessfully. Good
          to go. \n");
278
279   uint64_t eventDelay = rand->GetNext();
280   //The event callback will have null parameter
281   randEvent_SendUpdate = new Event(eventDelay, *eventDel_SendUpdate, (void *) 0);
282   randEvent_UpdateVar = new Event(eventDelay, *eventDel_UpdateVar, (void *) 0);
283
284   randEvent_ReSendUpdate = new Timer(ReSendTimeOutInterval, ONE_SHOT, *
          eventDel_ReSendUpdate);
285   randEvent_ReSendUpdate->Suspend();
286   return true;
287 }
288
289 void Gossip::SetNeighborhoodandLinkComparator(){
290   FRAMEWORK->SelectLinkComparatorRequest(PID,PWI::Neighborhood::QUALITY_LC);
291   ++n_ExpectedFrameworkResponses;
292   //FRAMEWORK->SelectLinkComparatorRequest (PID, QUALITY_LC);
293   LinkMetrics myThreshold;
294   myThreshold.quality = 0.10;
295
296
297   FRAMEWORK->SetLinkThresholdRequest(PID,myThreshold);
298   requestState = WAITING_FOR_CONTROL_RESPONSE;
299   ++n_ExpectedFrameworkResponses;
300 }
301
302 void Gossip::RequestDataNotifications(){
303   uint8_t mask = PDN_RECV_DEST_MASK;
304   FRAMEWORK->SelectDataNotificationRequest(PID, mask);
305   requestState = WAITING_FOR_CONTROL_RESPONSE;
306   ++n_ExpectedFrameworkResponses;
307 }
```

Data Status Handling

Packet related notifications are handled with ***DataStatusEvent(DataStatusParam ntfy)***. The patterns notify the framework about the type of notifications that they are interested in. They can change this request anytime throughout the course of operation. For a full list of notifications please refer to the TuscaroraFramework_API documentation.

In its start routine, `Gossip` passes its interest on two types of acknowledgements:

- *PDN_RECV_DEST_MASK*: Since `Gossip` registers for this type, it will be notified about whether a packet was successfully received via a `DataNotifierParam` with a *ackType* of *PDN_DST_RECV*. *ntfy.status* indicates the success or the failure of the reception by the *ntfy.noOfDest* destinations listed in the array, *ntfy.dest[]*. For negative acknowledgements, `Gossip` adds the list of negatively acknowledged destinations into its list of neighbors to be updated in the next cycle.
- *PDN_FW_RECV_MASK*: This notification type is on by default for all patterns and is generated without requiring to ask for it. Framework notifies `Gossip` whether it accepts the previous packet sent by the pattern via a `DataNotifierParam` with a *ackType* of *PDN_FW_RECV*. *ntfy.status* indicates whether the packet is accepted or rejected. With the reception of the response, `Gossip` cancels the timer that resends packets after a timeout period after which no response is received. For accepted packets, `Gossip` further clears the list of selected neighbors, and increments *nonce* variable used to identify packets not yet assigned a packet id. For rejected packets, `Gossip` further checks *ntfy.readyToReceive* parameter that indicate whether the framework is accepting new packets at that moment and resends a status update with the same *nonce* id if it does. On the other hand, if the framework indicates that it is not ready to accept packets, `Gossip` sets the *hasFWRejectedPacket* variable preserving the state. In that state, `Gossip` tries sending a new status update[2] with the reception of a future data notification (of any *ackType*) that indicates the availability of the framework to receive packets.

```
308 void Gossip::ReceiveDataNotifications(DataNotifierParam ntfy) {
309   ///You have received ack for previous message.
310   //check what happened to the previous message
311   Debug_Printf(DBG_PATTERN,"Gossip:: Received a data notification from framework\n")
          ;
312   switch (ntfy.type){
313     case ACK_DST_RECV:
314       if(!ntfy.status ){
315         Debug_Printf(DBG_PATTERN,"Gossip:: Message id %d not received by destination
              \n", ntfy.messageId);
316         SelectNeighbors(ntfy.dest);
317       }
318     break;
319     case READY_TO_RECV:
320       SendMessage();
321     break;
322     default:
323     break;
324   }
325 }
```

[2]but with the same nonce id since it is not matched to any messages yet

Neighbor Updates

Neighbor related notifications are handled by ***NeighborUpdateEvent(NeighborUpdateParam nbrUpdate)***, which is defined as virtual by the `PatternBase` and must be implemented by all patterns deriving from it.

For each change in the pattern neighborhood, the framework notifies the pattern with a *NeighborUpdateParam* that has a *changeType* of

- *NBR_NEW* for detected links,
- *NBR_DEAD* for links that are detected as broken or not satisfying the threshold conditions specified,
- *NBR_UPDATE* for links that remain in neighborhood but change some properties.

`Gossip` uses the ***UpdateTable(NeighborUpdateParam _param)*** of the `PatternNeighborTable` to make the corresponding changes in the *myNbrHood*.

```
326 void Gossip::NeighborUpdateEvent(NeighborUpdateParam nbrUpdate)
327 {
328   myNbrHood->UpdateTable(nbrUpdate);
329 }
330 void Patterns::PatternNeighborTable::UpdateTable(NeighborUpdateParam _param)
331 {
332   //bool signalPattern=false;
333   LinkMap *newLink;Link *ptnLink;
334   switch(_param.changeType)
335   {
336   case NBR_NEW:
337     newLink = new LinkMap(_param.link.linkId, _param.link);
338     nbrhood->Insert(newLink);
339     //signalPattern =true;
340     break;
341   case NBR_DEAD:
342     if(nbrhood->DeleteLink(_param.link.linkId)){
343       //signalPattern =true;
344     }
345     break;
346   case NBR_UPDATE:
347     Debug_Printf(DBG_PATTERN, "PatternNeighborTable:: updating neighbor %d on
          waveform %d\n", _param.nodeId, _param.link.linkId.waveformId);fflush (stdout
          );
348     ptnLink = nbrhood->GetLink(_param.link.linkId);
349     if(ptnLink){
350       if(ptnLink->linkId.waveformId){
351         ptnLink->metrics = _param.link.metrics;
352       }
353     }else {
354       printf("Mismatch in neighbor table status, I (Pattern %d) dont have link (%d,
            %d) in table, but received an update event \n Created new link\n",
355           this->patternId,_param.nodeId, _param.link.linkId.waveformId);
356       newLink = new LinkMap(_param.link.linkId, _param.link);
357       nbrhood->Insert(newLink);
358     }
359
360     break;
361   default:
```

```
362       printf("PatternNeighborTable:: Error: wrong neighbor update signal\n");
363       break;
364     }
365 }
```

Receiving Messages

Received messages are handled by ***ReceiveMessage(FMessage_t& msg)***. In this function, `Gossip` compares the status ID reported in the packet with the internal state ID. If the internal state variable (*currentStatusId*)

- is smaller than the one reported in the packet, the internal state variable is updated.
- is larger than the one reported in the packet, the source node of the the packet is added to the list of nodes that will be reported in the following period.
- is equal to the one reported in the packet, the packet is ignored.

```
366 void Gossip::ReceiveMessageEvent(FMessage_t& msg)
367 {
368   GossipMsg* gMsg = (GossipMsg*) msg.GetPayload();
369  Debug_Printf(DBG_PATTERN, "Gossip:: Received msg from %u with status seq %u \n",msg
       .GetSource(), gMsg->currentStatusId);
370   //Process the recevied message
371   if(currentStatusId < gMsg->currentStatusId) {
372     currentStatusId = gMsg->currentStatusId;
373   }
374   else if(currentStatusId > gMsg->currentStatusId) {
375     SelectNeighbors(msg.GetSource());
376   }
377 }
```

Handling Scheduled Events

Next, we implement the handler methods of scheduled events. In `Gossip`, we have two such methods: a method that periodically sends messages, ***RandomSendHandler(EventDelegateParam param)***; and a method that increments the state of the pattern, *UpdateVariableHandler(EventDelegateParam param)*. Both of these methods are scheduled randomly and reschedule their triggering events when executed. When sending an update, if no nodes were selected before, we randomly select a node using *AdjustSelectedNeighbors()*. We make sure that the selected node is a neighbor and was not already selected using *SelectNeighbors(NodeId_t nbr)*.

```
378 //Handler for the randEvent_UpdateVar event
379 void Gossip::UpdateVariableHandler(EventDelegateParam param){
380   uint64_t eventDelay = rand->GetNext();
381   uint64_t curTime = Debug::GetTimeMicro();
382   param.eventPtr->ReSchedule(eventDelay*100,(void *) (0));
383   Debug_Printf(DBG_PATTERN, "UpdateVariable:: Rescheduled event to fire at %lu\n",
       curTime+eventDelay);
```

```
384   ++currentStatusId;
385 }
386 //Handler for the RandomSend event
387 void Gossip::RandomSendHandler(EventDelegateParam param){
388   //Lets us first reshedule the event
389   uint64_t eventDelay = rand->GetNext();
390   uint64_t curTime = Debug::GetTimeMicro();
391   param.eventPtr->ReSchedule(eventDelay,(void *) (0));
392  Debug_Printf(DBG_PATTERN,"Gossip::RandomSend:: Rescheduled event to fire at %lu\n",
       curTime+eventDelay);fflush(stdout);
393   //Now lets broadcast our status
394   AdjustSelectedNeighbors();
395   if(SelectedNeighborList.Size() > 0) SendMessage(); //If I have at least one
        destination send this message
396 }
397
398 void Gossip::AdjustSelectedNeighbors() {
399   if(SelectedNeighborList.Size() > 0) return; //Already selected neighbors based on
        other criteria
400   uint16_t nbrCount = myNbrHood->GetNumberOfNeighbors();
401   Debug_Printf(DBG_PATTERN,"Gossip:: Got %d neighbors  \n", nbrCount);fflush(stdout)
        ;
402   if (nbrCount == 0) return;//Return if there are no neighbors to choose from
403   NodeId_t sNeighbor;
404   uint16_t i=0;
405   while(SelectedNeighborList.Size() == 0) {
406     uint64_t pickrandom = rand->GetNext();
407     pickrandom = pickrandom % nbrCount;
408     Debug_Printf(DBG_PATTERN,"Gossip:: Got %d neighbors and picking %lu neighbor
          from tabel  \n", nbrCount, pickrandom);fflush(stdout);
409     PatternNeighborIterator it = myNbrHood->Begin();
410     for(uint16_t ii=0; ii< nbrCount; ii++) {
411       if(pickrandom == 0) {
412         sNeighbor = (it->linkId.nodeId);
413       }
414       --pickrandom;
415       Debug_Printf(DBG_PATTERN,"Gossip:: Intering neighbortable inde %d neighbor is
            %d \n",ii, it->linkId.nodeId);fflush(stdout);
416       it=it.GetNext();
417     }
418     /*if(pickrandom == 0) {
419       sNeighbor = (it->linkId.nodeId);
420     }*/
421
422
423     Debug_Printf(DBG_PATTERN,"Gossip:: Selected %d neighbor \n", sNeighbor);fflush(
          stdout);
424     if( SelectNeighbors(sNeighbor) ) {
425       Debug_Printf(DBG_PATTERN,"Gossip:: PickRandomNeighbor %d \n", sNeighbor);
            fflush(stdout);
426     }
427     i++;
428     Debug_Printf(DBG_PATTERN,"Gossip:: in while %d times  \n", i);fflush(stdout);
429   }
430 }
431
432 bool Gossip::SelectNeighbors(NodeId_t nbr){
```

```
433   if( myNbrHood->GetNeighborLink(nbr) && !(SelectedNeighborList.Search(nbr)) ) { //
          If the node is my neighbor and not selected before
434     Debug_Printf(DBG_PATTERN,"Gossip:: SelectNeighbor: Ading neighbor %d neighbors
            to SelectedNeighborList \n", nbr);fflush(stdout);
435     return SelectedNeighborList.Insert(nbr);
436   }
437   return false;
438 }
439 }
```

Sending Messages

Sending status update messages is implemented in ***SendMessage()***. This method creates a packet containing the current status information of the node using ***PrepareStatusUpdate()***, and passes the constructed packet to the framework along with

- an array including the set of previously selected destinations in *SelectedNeighborList*,
- the size of this array,
- a temporary pattern generated message ID, namely *nonce* and
- a boolen variable indicating whether ACKs are requested.

While sending a message to the framework, pattern should provide a temporary ID to the message called a *nonce*, which is used to identify a packet until a unique Message ID is generated for the message by the Framework and returned to the Pattern. When the framework receives the packet, it generates a *DataNotifierParam* with an *ackType* of *PDN_WF_RECV*. This *DataNotifierParam* also includes the *nonce* generated with the pattern along with the *messageId* that can be used to uniquely identify the message from that point on.

The nonce is also useful for error detection and recovery, if either the Command from pattern to framework gets lost or if the Data Notification Event with the message ID from the framework gets lost. A pattern in general should start a timer when sending a message to the framework and if a Data Notification for the message is not received before the timer expires, it should resend the same message with the same nonce. If the framework receives two consecutive messages with the same nonce, it would recognize it as a duplicate and resend the data notification for that packet.

SendMessage() function also schedules an event resending a new packet if no framework response is received in *ReSendTimeOutInterval*. Finally, the *hasFWRejectedPacket* state variable is cleared since the packet has not been rejected at that time instant. This state variable can be set via a data notification as discussed in section 5.10.

```
440 // This function broadcast status
441 FMessage_t& Gossip::PrepareStatusUpdate(){
442   Debug_Printf(DBG_PATTERN, "Gossip:: PrepareStatusUpdate\n");
443
444   //Construct packet and send
445   FMessage_t *sendMsg;
446   //Make sure to use new() operator to create packet on heap.
447   sendMsg = new FMessage_t(sizeof(struct GossipMsg));
448   sendMsg->SetSource(MY_NODE_ID);
```

```
449   sendMsg->SetInstance(PID);
450
451   GossipMsg* gMsg = (GossipMsg*) sendMsg->GetPayload();
452   gMsg->currentStatusId=currentStatusId;
453   return (*sendMsg);
454 }
455 // This function sends a message to the list of selected neighbors
456 void Gossip::SendMessage() {
457   NodeId_t selnodes[SelectedNeighborList.Size()];
458   GossipNeigborContainerTypeElement* elptr = SelectedNeighborList.Begin();
459   int i=0;
460   while(elptr){
461     selnodes[i] = elptr->data;
462     elptr = SelectedNeighborList.Next(elptr);
463     ++i;
464   }
465   Debug_Printf(DBG_PATTERN, "Gossip::SendMessage multicasting to %d destinations \n"
        , SelectedNeighborList.Size()); fflush(stdout);
466   MessageId_t msgID = FRAMEWORK->Send(selnodes, (uint16_t) SelectedNeighborList.Size
        (), PrepareStatusUpdate(), (U64NanoTime*) NULL, false);
467
468   if (msgID == 0){
469     Debug_Printf(DBG_PATTERN, "Gossip::SendMessage framework is busy. Rejected the
          packet. Keeping list of neighbors.\n"); fflush(stdout);
470   }
471   else{
472     Debug_Printf(DBG_PATTERN, "Gossip::SendMessage framework accepted message (%d).
          \n", msgID);fflush(stdout);
473     ClearSelectedNeighborList();
474   }
475 }
476
477 void Gossip::ClearSelectedNeighborList(){
478   GossipNeigborContainerTypeElement *elptr,*elptr2;
479   elptr = SelectedNeighborList.Begin();
480   while(SelectedNeighborList.Size()>0){
481     elptr2 = SelectedNeighborList.Next(elptr);
482     SelectedNeighborList.DeleteElement(elptr);
483     elptr = elptr2;
484   }
485 }
```

Broadcasting a Message

While `Gossip` primarily uses generic ***SendData*** API, sending a broadcast message is quite similar. The primary difference is that, while generic ***SendData*** primitive addresses to one or more destinations, a broadcast is addressed to a particular waveform. The framework uses the waveform's broadcast API (if it has one) to send the message. *Please note that how the actual broadcasting is implemented is up to the waveform provider.* The other difference is that a broadcast message will not generate any *PDN_DST_RECV* data notifications, but will generate the *PDN_FW_RECV* when the message is accepted by the framework, and *PDN_WF_SENT* when the waveform has sent the message out. The number of waveforms available on a node and their IDs are obtained by the Pattern using the ***FrameworkAttributeRequest()*** Command issued

while the pattern initialized. Once their IDs are known, the waveform IDs can be used to send broadcasts. The code listing belows shows an example of sending a broadcast message.

```
///These lines show how to handle AttributeResponse to parse and get the waveform IDs
void Gossip::Handle_AttributeResponse(ControlResponseParam response){
  FrameworkAttributes *atr = (FrameworkAttributes*) response.data;
  fwAttributes = *atr;
  requestState = NONE_PENDING;
  patternState = EXECUTING;
  active=true;
  Debug_Printf(DBG_PATTERN,"FWP:: Handle_AttributeResponse: Framework supports %d waveforms, with max pkt size is %d .\n", fwAttributes.numberOfWaveforms, fwAttributes.maxFrameworkPacketSize);
  for(uint i=0; i<fwAttributes.numberOfWaveforms; i++){
    Debug_Printf(DBG_PATTERN, "FWP::Handle_AttributeResponse: Waveform %d Id is %d\n", i, fwAttributes.waveformIds[i]);
  }
}


///Creating and sending broadcast
FMessage_t *sendMsg = new FMessage_t();
sendMsg->SetType(Types::Pattern_Type);
sendMsg->SetInstance(PID);
sendMsg->SetPayload(&dataPtr[i*maxPayload]); //Set some payload
sendMsg->SetPayloadSize(maxPayload); //Set payload size

///Send a broadcast on the first waveform
FRAMEWORK->BroadcastData(PID, *sendMsg, fwAttributes.waveformIds[0] , nonce);
```

Stopping

The `Gossip` stops its operation when its ***Stop*** method is called. If there are previously scheduled events for sending updates, updating internal gossip value, and resending an update after a predefined value, this method cancels them. This in turn also stops rescheduling of these events.

```
bool Gossip::Stop()
{
  randEvent_UpdateVar->Cancel();
  randEvent_SendUpdate->Cancel();
  randEvent_ReSendUpdate->Suspend();
  return true;
}
```

Testing the Pattern

In order to test the pattern, a separate test needs to be written. The test code is separate so that the pattern code can be used for simulation, deployment, and across different hardware platforms. The tests reside under the "`Tuscarora/Tests/`" directory. Pattern tests reside under "`Tuscarora/Tests/`

Patterns/<pattern-name>" directory. In our case the test files can be found under "Tuscarora/Tests/Patterns/Gossip".

The runOrDebug script in the Tuscarora root directory finds all the tests under the "Tuscarora/Tests" directory and compile them. The test files have the format "run<TestModuleName>.cpp". To run a test, you need to simply provide the "<TestModuleName>" corresponding to the test file, which in our case is "Gossip". Each test has a "main" method, which first sets the node id, instantiates the framework modules, and finally creates an instance of the specific test and executes the test.

```
516 //main for the test
517 int main(int argc, char* argv[]) {
518   //Make sure RuntimeOpts construction is the very first line of main
519   RuntimeOpts opts ( argc-1, argv+1 );
520
521   MY_NODE_ID = atoi(getenv("NODEID"));
522  // NETWORK_SIZE = opts.nodes;
523    //Turn on debugging for Gossip pattern
524 //    DBG_TEST = true;
525  // set in Lib/Misc/Debug.cpp
526 //    DBG_PATTERN = true;
527  // set in Lib/Misc/Debug.cpp
528 //    //DBG_WAVEFORM = true;
529  // set in Lib/Misc/Debug.cpp
530 //    //DBG_CORE_DATAFLOW = true;
531  // set in Lib/Misc/Debug.cpp
532
533 //    DBG_CORE = true;
534  // set in Lib/Misc/Debug.cpp
535
536   FW_Init fwInit;
537   fwInit.InitFI();
538   fwInit.Execute(&opts);
539
540   GossipTest *gossipTest = new GossipTest();
541   gossipTest->Execute(&opts);
542
543   while(1){sleep(1);}
544   return 0;
545 }
```

The *GossipTest* class itself is quite simple and exposes an Execute method, which is used to start the test. The constructor and the Execute method of this class are used to setup and start the test respectively. The *GossipTest* starts a 2-second timer to let the network stabilize. (This example also shows how to initialize and use a timer module). The timer is started within the Execute method and when the timer fires, the Gossip pattern is started.

```
546 GossipTest::GossipTest(){
547   //Turn on debugging for Gossip pattern
548 //   DBG_PATTERN = true;
549  // set in Lib/Misc/Debug.cpp
550 //   DBG_WAVEFORM = false;
551  // set in Lib/Misc/Debug.cpp
552 //   DBG_CORE = false;
553  // set in Lib/Misc/Debug.cpp
554
555   PatternId_t pid = GetNewPatternInstanceId(Gossip_P);
```

```
556    gossip = new Gossip(pid);
557    timerDel = new TimerDelegate (this, &GossipTest::TimerHandler);
558    startTestTimer = new Timer(2000000, ONE_SHOT, *timerDel);
559    }
560
561    void GossipTest::TimerHandler(uint32_t event){
562    Debug_Printf(DBG_PATTERN, "GossipTEST: Starting the Gossip Pattern...\n");
563    gossip->Start();
564    }
565
566    void GossipTest::Execute(RuntimeOpts *opts){
567    //Delay starting the actual pattern, to let the network stabilize
568    //So start the timer now and when timer expires start the pattern
569    Debug_Printf(DBG_PATTERN,"About to start timer on node 1\n");
570    startTestTimer->Start();
571 }
```

6 Waveform Development

This chapter explains some of the features of the waveform interface. The interface is defined in a flexible way to accommodate a diverse set of waveforms.

Overview

The waveform interface serves as the lower interface of the Tuscarora Framework. Any waveform that satisfies the interface definition may be plugged into the Framework for use alongside other waveforms, in order to support a heterogeneous MANET.

To satisfy the waveform interface, a waveform provider needs to implement a single interface, called `Waveform_I`. The interface, `Waveform_I` is a template class that is instantiated by passing the waveforms address type as a template parameter. In total, there are 10 methods in each instantiated class, of which 3 are optional, that is, a waveform provider can choose to not implement any or all of the optional methods. Fig. 6.1 shows the waveform interfaces as related to the framework interfaces.

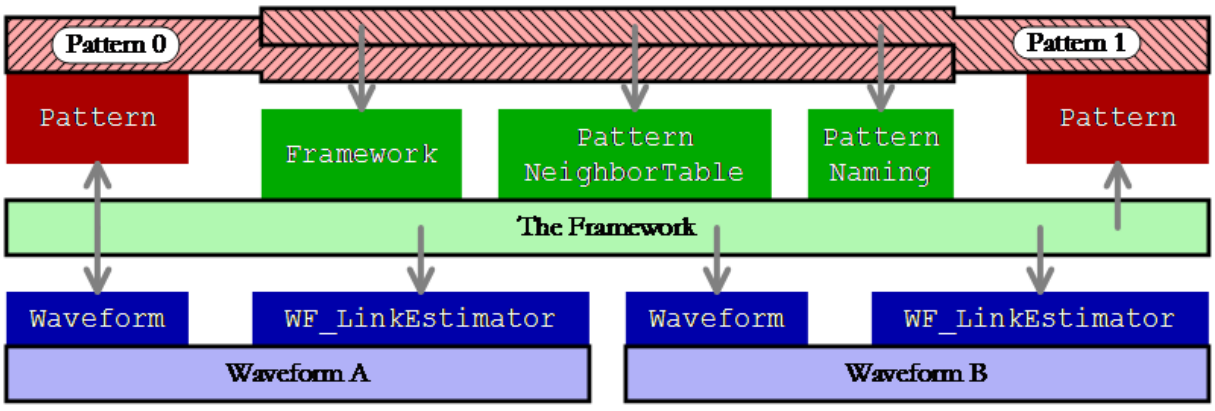


Figure 6.1: Waveform Interfaces

Please refer to the Framework API documentation for full detail about the `Waveform_I` interface.

Programming Style: Syntax and Semantics

The interface design uses asynchronous message passing semantics, with a functional syntax, to define interactions between the waveform and Tuscarora. It consists of two sorts of methods: 1) incoming

commands, exercised by the user of the interface (namely, the Framework, via a communication shim) on the waveform, and 2) outgoing events, sent by the waveform to the user of the interface.

In terms of syntax, both commands and events are specified using function calls. This simple representation yields ease of programming and, equally importantly, avoids restriction to a particular implementation of method invocation, such as socket communications between processes, which may be appropriate for some use cases (e.g., on large hardware platforms) but not others (e.g., for ns-3 simulation platforms where the overhead seems to matter).

In terms of semantics, all interface methods are explicitly associated with a message passing semantics. The message passing semantics is reflected by the fact that all incoming commands have a return type of void. Moreover, for any command that expects some sort of response from the waveform, an associated event method exists in the waveform interface, to convey the result of the computation.

The design assumes that a communication library, which we colloquially refer to as a shim-layer, is provided that implements the invocation of command and event methods using asynchronous message passing. The slightly different shim layer might be required for each waveform-platform pair, although generally far fewer shim layers would be required.

Data Plane and Control Plane

The waveform interface functionality spans a Data Plane and a Control Plane. The Data Plan interface consists of the functions SendData, BroadcastData, Data Notification Event and the Received Message Event. The remaining commands and events, as detailed below, which deal with the control status of the waveform, link estimation and schedules comprise the Control Plane. Figure 1 shows the Summary of the Commands and the Events that are expected by the framework in response to them, split across the Data and Control planes. The dotted lines show the dependency between a command and the event.

Interface Interaction Pattern using Asynchronous Message Passing

In general, the data and control plane functions follow a common interaction pattern. A command originates from Tuscarora to the waveform and, subsequently, one or more corresponding events are sent from the waveform back to the Framework. Events are mapped and routed to particular modules and/or function within the Framework; it is convenient to think of each event as named pipe between the waveform and the Framework. Each event is specified as a class with a single Invoke method that has a specific type of parameter. The Invoke method returns a bool (as against the void return type of most commands). The return type of Invoke indicates the status of queuing the Event on to the communication mechanism. The return type does not indicate the success or failure of sending the event to the framework. For example, if an Invoke returned a true it means that the shim layer responsible for the communication has accepted the event and has queued it up for sending at a latter time. If it returns a false, it means either the communication mechanism is either broken or is busy processing existing events and cannot accept any new events to be sent out. Five types of events are defined for the waveform:

1. Received Message Event (WF_RcvMessageEvent_t): This Event is sent whenever a data message is received on the waveform, with WF_RecvMsgParam as the parameter.
2. Data Notification Event (WF_DataNotifierEvent_t): This Event is sent to convey the status of the SendData and BroadcastData calls and the WF_DataNotifierParam is used as the event parameter.

This Event might be generated up to three times for each data message, once to acknowledge the message has been received by the waveform, once when it is sent out and once when the message is received at the destination.

3. Control Response Event (WF_ControlResponseEvent_t): This Event is sent in response to one of the request commands in the control plane with WF_ControlRssponseParam as the parameter. The parameter contains a response type, response status (success/failure), any additional data and its size.
4. Link Estimation Event (WF_LinkEstimateEvent_t): This Event is sent to provide to the Framework link estimates updates about a neighbor. The WF_LinkEventParam is used as the parameter for this event.
5. Schedule Update Event (WF_ScheduleUpdateEvent_t): This Event is sent by the waveform to provide updates about its sending or receiving schedules. WF_ScheduleUpdateParam is used as the event parameter.

Every command originating from the Framework has a sequence number and the events use the same sequence number to identify their response. The Framework tracks the sequence number it uses for each command and if it fails to receive a response for some command, it may resend that command with the same sequence number. The waveform using the same sequence number to send multiple response to the same command or to update/change the response to the command is acceptable. For the control plane commands the sequence number is called as the RequestId, for the data plane it is called as the MessageId. A couple of special cases are allowed: The Link Estimation Event and the Schedule Update Event do not use sequence numbers in their events and error recovery is not provided for those events. For the Received Message Event the sequence number is generated by the Waveform and sent to the Framework.

Control Plane State

As far as the Frameworks awareness of the control state of the waveform is concerned, it knows only that the state of the waveform can be one of the following states which are defined by the enum WF_ControlP_-StatusE:

- WF_NORMAL: Waveform has been initialized and is operating within normal bounds.
- WF_BUSY: Waveform has been initialized, but is busy with non-data plane functions. The Framework should desist from sending messages to waveform, it is likely that the waveform might not accept the messages.
- WF_ERROR: Waveform is in an error state. The Framework should not send messages to waveform, until the status changes.
- WF_BUFFER_LOW: The waveform is operating normally, but is at risk of running out of messages to send out. The Framework may respond by attempting to increase the data output to the waveform.
- WF_BUFFER_FULL: The waveform is operating normally, but is at risk of being overrun by data packets. The Framework should respond by slow down its rate of invoking data packet sends on the waveform.

Flow Control

The Waveform accomplishes flow control between the Waveform and Framework in one of three ways:

1. Waveform can drop the packet being sent to it and return a negative status when sending a data notification message back to the Framework. The Framework will interpret this as the waveform not being ready to receive data messages.
2. Waveform can send a WF_BUFFER_LOW or WF_BUFFER_FULL response messages to the Framework, to request it to speed up or slow down the data message rate.
3. Waveform can send a WF_ERROR status message to completely stop receiving any data messages from the Framework if it has to clear a backlog. It could subsequently send a WF_NORMAL status message when it is ready to resume. The remaining sections describe the API methods (first commands, then events) in detail.

Parameter Ownership

In Tuscarora whenever a function call happens across layers the ownership of the data structures passed as parameters is transferred to the callee. In this particular case, this applies to the commands the Waveform_I interface receives and the event calls the waveform makes. We call the Give model of ownership where the ownership is given when to the callee. The Give model is used to avoid deep copying of packets and other parameters at the callee so that efficiency of processing can be improved. However, there are some implications of this Give model.

- Pass by reference: Any large parameter objects should be passed by reference in a function to make optimum use of the Give model. Passing by value does not utilize the Give model. However, for small objects this might not matter much.
- Allocation on heap: The caller must also be explicitly aware of which parameters are passed as references, since these objects must be allocated on the heap (and not on the local stack). Creating a local object and passing this as reference to another method will result in run time errors. Also, once a parameter has been passed as reference and the call has been made, the same object cannot be modified by the caller anymore. This would also result in run time errors.
- Deallocation: It is the callees responsibility to deallocate any parameter that is passed by reference, since these were originally allocated on the heap by the caller. Failing to deallocate (or reuse) such objects will result in a memory leak.

Packet Metadata

Waveforms should also provide metadata for each packet received. The metadata provided is standardized across all waveforms. Currently 3 types of metadata have been defined, (i) Received signal strength indicator (RSSI), (ii) Signal to noise and interference ratio (SINR) and (iii) Packet reception time stamp. The metadata is used by the framework to create and/or improve the estimates for the properties of links.

Link Estimation and Network Discovery

Link Metrics

A *Link Metric* is a measure of the performance or other property of a link that is sufficiently generic that it may be used by patterns to compare links over potentially different waveforms. **Please note** that while this is expected to be standardized, it is not yet finalized, and could change between versions of the framework. The current set of link metrics are:

Quality: Abstract quality metric expressed as a real number between [0,1]

Data rate: Link data rate expressed as log_2 (bps)

Average Latency: Average latency in sending a packet to the destination, expressed as log_2 (seconds)

Energy: Average energy to transmit a packet expressed as log_2 (picojoules)

Cost: Abstract average cost of sending a byte

Expected Transmit Latency: Expected delay to start transmitting the packet (Need not be implemented by waveform)

```
572 ///Structure to be used by Waveforms to provide link metrics to the framework
573 struct WF_LinkMetrics {
574 ///A quality metric expressed as a real number between [0,1]
575 UFixed_7_8_t quality;
576 ///Datarate expressed as log_2(bps)
577 UFixed_2_8_t dataRate;
578 ///Average latency expressed as log_2(seconds)
579 Fixed_2_8_t avgLatency;
580 ///Average energy to transmit a packet expressed as log_2(pica-joules)
581 UFixed_2_8_t energy;
582 };
583
584 ///Structure used by the framwork to expose link metrics to the pattern
585 struct LinkMetrics: public WF_LinkMetrics {
586 ///Average cost of sending a packet
587 UFixed_7_8_t cost;
588 ///Tuscarora's estimate of the latency for the next transmit
589 U64NanoTime expLatencyToXmit;
590 };
591
592 struct LinkId {
593 NodeId_t nodeId;
594 WaveformId_t waveformId;
595 };
596
597
598 ///Structure to store information about a link in the framework and pattern layers
    including its metrics
599 struct Link {
600 LinkId linkId;
601 //WaveformId_t waveformId;
602 LinkTypeE type;
603 LinkMetrics metrics;
```

```
604 };
```

Listing section 6.4 shows the structures that define the Link Metrics.

Link Estimation

Tuscarora supports three types of link estimation protocols, one of which can be chosen at run time. The link estimation protocols currently support only the quality metric, other metrics are not supported in the current version of the framework . Each of the estimation protocols assigns a quality metric to each of the neighbors in a slightly different way. The requirements for the quality metric are:

- Its value is real number between 0 and 1.
- It should reflect the stability, as well as the gross availability of a particular link
- Its evaluation should be independent of the beaconing frequency or information exchange rate.

The three link estimation protocols differ mainly in how they interpret whether or not a particular estimation is received correctly. They all use the same 'beaconing' mechanism. (Beaconing is a term usually used to indicate a periodic local broadcast messaging mechanism.)

Periodic: A basic protocol that beacons periodically using a pseudo-random schedule at a given rate and has no knowledge of its neighbors' beaconing schedules. Links are removed after a period of time (called the Inactivity Period) goes by without receiving a beacon. This parameter is specified by a positive integer > 0, which is interpreted as a multiple of the beaconing period; the default value is 3.

Schedule-Aware: This protocol beacons periodically and maintains a record of neighbor's beaconing schedules. Links are removed when a beacon should have been received, but has not been.

Conflict-Aware: This protocol also beacons periodically and maintains a record of neighbor's beaconing schedules. In addition it considers potential conflicts in the beaconing schedules of its neighbors. Links are removed when a) an expected beacon is not received, and b) there is no conflict with the schedules for the other nodes in the Neighbor Table Table. If there is a potential conflict, the link is not removed, but link quality does go down.

The default Link Estimation is the Periodic estimation protocol with message beaconing frequency of 5hz (200 ms period). To set the estimation period, the parameter LinkEstimationPeriod (specified in microseconds) should be set when running Tuscarora in the simulator. The Link Estimation protocol is chosen by setting the LinkEstimationType parameter.

Updating the quality of the links

Link quality estimates are updated at intervals roughly corresponding to the beaconing frequency—either when a beacon is received or when it was expected but was not received. The quality updating method is the same, regardless of the beaconing protocol used. The quality of a link reflects its behavior over a certain time interval in the past called the 'Estimation Window'. This window is divided into 'Activity Periods' when a link was alive or dead. Each such period gets a certain number of points and the quality is the normalized average of points for the Estimation Window. When a link is dead it gets 0 points. When a

link is alive its received points based on the number of consecutive estimation beacons it has received. The first time it receives a message it gets one point. When the second message is received, the points for the period is updated to 1 + 1.5. The points at the end of x^{th} consecutive message is given by:

$$Points(x) = \sum_{0}^{x} 1.5^{y},\ where\ y = max(x,5) \tag{6.1}$$

The number of points in each Activity Period is multiplied by the size (time) of the Activity Period and added up, and then normalized by the maximum possible points for the Estimation Window to arrive at a value between 0 and 1.

Network Discovery

The Network Discovery protocol is chosen by setting the DiscoveryType parameter when running Tuscarora. There are three Network Discovery protocols, identified by the following values for the DiscoveryType parameter:

Global: This protocol adds all nodes in the network to the Potential Neighbor Table at the beginning of the simulation.

Oracle-2hop: This protocol adds and removes neighbors from the Potential Neighbor Table on a periodic basis currently set to 1Hz. It uses the exact and current positions of all nodes to determine the contents of the Potential Neighbor Table. The contents of the Potential Neighbor Table are all nodes within 2 communication range.

Long Link: This protocol adds and removes neighbors from the Potential Neighbor Table on a period defined by the parameter LongLinkPeriod. The contents of the Potential Neighbor table are all nodes within LongLinkHops communication ranges.

Long Link Network Discovery has two extra parameters, LongLinkPeriod and LongLinkHops. LongLinkPeriod is measured in microseconds between Long Link Beacons. LongLinkPeriod must be greater than $(Size + Size/25) \times 1000$. LongLinkHops defines the diameter of the Potential Neighbor Region, a circle with a radius of $LongLinkHops \times Range$

Examples

Using Periodic Link Estimation with a dead neighbor period of 2 seconds and a Link Estimation Period of 1Hz using Global Network Discovery.

> $ ./runOrDebug.sh FI - - LinkEstimationType periodic LinkEstimationPeriod 1000000 DeadNeighborPeriod 2 DiscoveryType global

Using Schedule Aware Periodic Link Estimation with a Link Estimation Period of 10Hz and 2 hop Long Link Network Discovery.

```
$ ./runOrDebug.sh FI -- LinkEstimationType scheduleAwarePeriodic LinkEstimationPeriod
100000 DiscoveryType longlink2hop
```

7 Customizing Build, Testing and Execution

Customizing the build of Tuscarora

Tuscarora has been designed with the unique ability to customize the build to a given platform by turning on or off certain features even while keeping the key interfaces and therefore the Patterns and Applications portable across platforms. In general the 'Platform' directory contains files and implementations that are specific to a platform. The Tuscarora build system looks for a *PlatformConfig.h* file under a given platform directory. Every platform should have this file. A number of customizations can be done by defining (#define) certain contants in this file. Currently the following customizations are supported.

1. **ENABLE_FW_BROADCAST:** By defining this flag to either 0 or 1, the BroadcastData API in the Framework_I can either be removed or added. This flag is the only one that actually changes an interface module. Samraksh believes that the BroadcastData method should be removed from the framework interface, but this could be too drastic a change for a community that has grown up equating wireless network with broadcast. Also the versions of framework before 2.0 had this APi. Hence the API is kept for backward compatibility. However it is possible, that this option might disappear in future versions and NOT having the BroadcastData might become the mainstream version. To be sure, this has no connection to how the message is actually sent out by the waveform, since that is completely left to the waveform implementation. Also, if a given waveform offers BroadcastData api, the framework itself might continue to use that API for purposes such as Link Estimation or Discovery. This flag is only about the interface provided to the Pattern and not about what the framework uses with a Waveform.
2. **ENABLE_PIGGYBACKING:** By defining this flag to either 0 or 1 piggybacking can be turned On or Off. Piggybacking is the capability of framework to add a small message to an already scheduled packet on a particular waveform. This capability can now be used only with time stamping service.
3. **ENABLE_EXPLICIT_SENDER_TIMESTAMPING:** Tuscarora supports two kinds of timestamping services, a explicit sender based one and an implicit receiver based one. Either the explicit or the implicit will be turned on. Both or them cant be turned on simultaneously. The explicit service super seeds the implicit service. Setting this tag to one will enable the Explicit Time Stamping Service. Please see section 1.4.1 for more details about timestamping service.
4. **ENABLE_IMPLICIT_SYNC_TIMESTAMPING:** Setting this flag to 0 or 1 will turn Off or turn On the implicit timestamping service. If explicit timestamping service is turn on, that will super seed this flag. Please see section 1.4.1 for more details about timestamping service.

Deploying Tuscarora Binaries

Cross Compilation

Tuscarora and associated sofware is written mostly in C++, with minimal C functions. The build system uses CMake [2] to configure the build for a particular platform. CMake is a cross platform open source tool, that can generate configuration scripts (such as Makefiles) for build tools (such as make) and also setup the compiler to be used and its associated toolchain. This gives the framework the ability to run the exact same code on many different platforms such as from embedded microcontrollers to Ubuntu Desktops to Windows PCs. However, some platform specific implementations will be needed when porting to a new system. These are mostly the PAL layer modules (where it might be possible to copy large parts of the code) and the shim layer modules that enable binding between the layers according to the needs of the deployment.

Currently, the ./runOrDebug.sh script recognizes three types of platforms, a native x64_linux platform, a 'dce' simulator platform and an 'arm-linux' platform. Depending on platform specified, the script invokes the CMake with the corresponding toolchain file. All of the platform specific setup for a CMake given platform goes to into a Toolchain-platform-arch.cmake file. These files are found under the 'Platform' directory. For example for dce this file is called, 'Toolchain-ns3-dce.cmake'

To add a new platform, first a toolchain.cmake file needs to be created that will specify the location of the compiler and toolchains, compiler options, etc. Then the runOrDebug.sh needs to be modified minimally to call the platform as an option.

Installing Binaries Remotely

Deployment of binaries to target system depends on the types of file transfers supported by the target system. There are no special requirements for cross compiled Tuscarora binaries. Transfer modes such as using a USB cable or SSH or ftp is generally used. A tool to deploy the compiled binaries to remote systems that support SSH is expected to be available soon.

Executing Tuscarora Framework Modules in a Simulator

The 'runOrDebug.sh' in the root directory is the primary script to execute the modules and tests under framework. The script cleans, builds, runs and collects the output for the 'test-name' specified for a given platform. For simulations, the platform is called 'dce'. Run the script without parameters for information on supported options. Run the script with the test-name and '-h' option to get test-specific help.

Example Executions

0. Getting help for the usage of runOrDebug.sh, to find the list of available tests.

    $ ./runOrDebug.sh

1. Getting help for Gossip test, to find the list of all test parameters with their default values.

$ ./runOrDebug.sh -h Gossip

2. Running the Gossip pattern test for 60 secs on a 100 node network.

$ ./runOrDebug.sh -p dce Gossip -- RunTime 60 Size 100

3. Running the Gossip pattern test inside the gdb to debug for 100 nodes (and a default duration of 6 secs).

$ ./runOrDebug.sh -p dce -d Gossip – Size 100

4. Running the Framework Interface test 'FI' with Periodic Link Estimation with a dead neighbor period of 2 seconds and a Link Estimation Period of 1Hz using Global Network Discovery.

$ ./runOrDebug.sh -p dce I -- LinkEstimationType periodic LinkEstimationPeriod 1000000 DeadNeighborPeriod 2 DiscoveryType global

5. Running the Framework Interface test 'FI' with Schedule Aware Periodic Link Estimation, a Link Estimation Period of 10Hz, and 2-hop Long Link Network Discovery.

$ ./runOrDebug.sh FI -- LinkEstimationType scheduleAwarePeriodic LinkEstimationPeriod 100000 DiscoveryType longlink2hop

6. Running the Gossip pattern test with 10 nodes with a waveform configuration file.

$ ./runOrDebug.sh Gossip – Size 100 WFConfig ${TUS}/dceln/wf-config.cnf

7. Running the Gossip pattern test with a tracefile based mobility model. The tracefile needs to be in the ns-2 mobility trace file format.

$ ./runOrDebug.sh Gossip – Size 10 RunTime 10 Mobility TracefileMobilityModel Tracefile ${TUS}/TraceFiles/Simple-Size10-10secs.tr

Waveform Configuration File

Each line in the the waveform configuration file has 6 columns separated by spaces specifying the waveform ID, the implementation of the waveform within DCE, the underlying network device in ns-3, cost metric associated with using this waveform, energy metric associated with this waveform, and the list of nodes that this network device will be available on. The list of nodes is a comma separated list that is a combination of node IDs and/or node ID ranges.

An example configuration of 20 nodes is given in $TUS/Config/hybrid-20nodes.cnf file that has the following contents:

2 WF_AlwaysOn_DCE_Ack WifiNetDevice 4 3 0-9,15
3 WF_AlwaysOn_DCE_Ack SimpleWirelessNetDevice 2 4 5-19

The first line in this configuration file specifies a Wifi ns3 radio module, the WF_AlwaysOn_DCE_Ack waveform in Tuscarora with an ID of 2 on nodes 0,1,..,9 and on node 15. Similarly, the second line specifies a SimpleWireless radio with an ID of 3 on nodes 5,6,..., 20.

Testing Methodology

Tuscarora provides a testing methodology where the user has complete control over what tests are run and they are run for how long. More over the testing itself is written in such a way that it is portable and platform independent. Test code is cleanly seperated from framework and ASNP code to enhance portability. All tests must reside under the 'Tests' directory. The runOrDebug.sh will find all the tests unders the 'Tests' directory, that are named using a particular naming convention (explained below) and make them available to be executed on any platform. The validate.sh script does the same for validations (or stage 2).

There are two steps to a test (1) to orchestrate and run a test, (2) to validate the output of the test. The two steps can be separated or integrated into a single test file/script. Step 1 needs to be done in C/C++ and usually are names as 'runTestName.cpp', for example, a file that runs a time test is named 'runTimer.cpp'. Step 1 should contain a main file, which is where the test logic will start and the test writer can instantiate the various modules of the framework that are needed for the test and run them. If step 1 is separate from step 2, then step 1 should generate the output files / data that is necessary of step 2. Step 2 files can be implemented in any language/script, but should be named as 'validateTestName.*'. For example, a python validation script to verify timer's correct behavior is called as 'validateTimer.py'.

There are several reasons to adopt this two step process;

- When running in the simulation mode inside the ns3-dce, every node has access only to its own state and therefore end-to-end validation tests or network-level behaviors need to be run outside of the simulator using output files collected from every node. Thus necessitating the two step process.
- The first state essentially just setup and runs a given test, therefore in many cases this code can double up as an deployment app. Or in otherwords an app is not very different from a test and this enhances the tight coupling between the testing-to-deployment cycle
- The second step can be more flexible and versatile. It can use state from across nodes, either in simulation mode or otherwise. Also it can be written in any language, that a platform can support. Many a times the actual validation involves data analysis and running the second stage using Python or even Matlab would be more appropriate.

8 Porting and Platform Abstraction Layer

Platform Abstraction Layer

Tuscarora's dependency on the platform that it runs on is abstracted through a set of classes forming platform abstraction layer, "PAL". All other Tuscarora modules interact with this layer for platform layer functionality such as:

- time and timer related operations,
- inter process messaging mechanisms,
- interactions with the radio hardware,
- random number generation,
- logging and file I/O.

The interfaces for these classes are listed by the header files in "`Include/PAL/`"

When porting the framework to a different platform, it is expected to check the compatibility of these classes with the new platform and re-implement them as necessary.

Modules necessary to port framework

Time

`U64NanoTime` defined in "`Include/Interfaces/PAL/Time\_I.h`" provides time related services such as time retrieval, resetting, and comparisons for the Tuscarora system. It uses nano seconds as the smallest time unit and represents time with `uint64_t`.

The interface for the particular queries to get the system time is defined by `SysTime` structure in "`Include/Interfaces/PAL/SysTime.h`"

Timer

`Timer` defined in "`Include/Interfaces/PAL/Timer\_I.h`" provides a timer service for Tuscarora system. The objects of this class is created by providing 4 inputs: a period defined in microseconds, a timer type defining *ONE_SHOT* or *PERIODIC* operation, a `TimerDelegate` that is invoked at the time of firing,

and an optional name for the timer. Timer objects start in inactive state. They start their operation when their ***Start()*** method is called. A timer in active state call the function specified by its delegate at its due time. Timers specified as *PERIODIC* automatically reschedule themselves and fire up periodically while *ONE_SHOT* timers goes to an inactive state at the end of their operation. Users can suspend or modify the period and the type of a timer by ***Suspend()*** and ***Change(uint32_t period, uint8_t type)*** methods.

Random Number Generator

Tuscarora system internally uses `UniformRandomInt` that provides random variables of integer type and the `UniformRandomValue` class that provide random values of double type from real numbers domain. Platform independent definitions for these classes are available "`Include/PAL/PseudoRandom/UniformRandomInt.h`" and in "`Include/PAL/PseudoRandom/UniformRandomValue.h`". These random number generators use a state definitions specialized from the common `RNGState` that has a platform independent definition available in platform-independent definition available in "`Include/PAL/PseudoRandom/rng-state.h`".

RNGState

These random number generators use a state definitions specialized from the common `RNGState`. The platform-independent definition of `RNGState` is available in "`Include/PAL/PseudoRandom/rng-state.h`".

The numbers obtained from a random number generator engine forms a stream of numbers. `RNGState` object represents a state identifying the position of the next random number to be generated within that stream. The stream is identified by platform dependent `RNStreamID`. `RNGState` also holds a private templated `DistributionParametersType` that define the distribution of the random numbers generated from the random number generator. The distribution parameters can be retrieved/set using ***GetDistributionParameters/SetDistributionParameters*** methods.

Other than the default and the copy constructors, `RNGState` also provides a constructor that initializes state based on a random number stream definition, `RNStreamID`. `RNStreamID` can be retrieved/set using ***GetRNStreamID/SetRNStreamID*** methods. However, setting `RNStreamID` reinitializes the state to the beginning of a random stream.

Finally, ***SetEngineStateBuffer/GetEngineState*** can retrieve/set the state from a given random number generator engine.

UniformRandomInt

Produces uniform random values of type `uint_64_t`. The distribution parameters for this class is `UniformIntRVDistributionParametersType`, which defines the minimum and maximum of the range of the random values. The limits are included in this range. The internal state of this RNG is derived from `RNGState`, which encapsulates `UniformIntRVDistributionParametersType`.

UniformRandomValue

Produces uniform random values of type `double`. The distribution parameters for this class is `UniformDoubleRVDistributionParametersType`, which defines the minimum and maximum of the range of the random values. The limits are included in this range. The internal state of this RNG is derived from `RNGState`, which encapsulates `UniformDoubleRVDistributionParametersType`.

Logging

Creating logs of is critical for effective debugging as well as getting statistics from the system. At the early development stages, simple text based logs may be preferred thanks to their ease of use. However, as the system complexity grows, the size of these logs grew too large to be useful. In that case, using memory mapped I/O is the preferred method for many developers.

Tuscarora merges these two concepts by providing a common logging element that can easily switch between text based logs and memory mapping based logs through the templated class `InfoLogger<Obj>`.

`InfoLogger<Obj>` can create a memory mapped output file, a text based logging file or both based on the constructor parameters.

```
InfoLogger(std::string filename, bool _record_in_txt = true, bool _record_in_memmap
    = false )
```

The template parameter `Obj` defines the logging element. The logging element should be derived from `GenericLogElement` class and should internally implement the following methods:

- std :: string PrintHeader(): This method is executed at the beginning of a txt based log. It prints textual representation of the variables being written in order they are defined in the logging element.
- std :: ofstream& operator<< (std::ofstream &out, const GenericLogElement &nUpLogEl) : This method directs the member variables to the ofstream. The order used for this operation must match the order defined in PrintHeader().

An example use for this library is available in "`Patterns/Pplsc/Pplsc.h`". `PplscRecvMsgLogElement` and `PplscSentMsgLogElement` are defined as the logging elements. `PplscSentMsgLogElement` defines recording (i) the time of the log, (ii) the destination node ID, (iii) the link selection criteria ID, and (iv) a message ID. `PplscRecvMsgLogElement` defines (i) the time of the log, (ii) the source node ID, (iii) the waveform ID from which the message is received, and (iv) a message ID.

`Pplsc`'s constructor creates information loggers to record receiving and sending messages and selects only txt based logging.

```
  msg_logger_ptr_rx = new InfoLogger<PplscRecvMsgLogElement>("PplscReceivedMessages.
      bin", true, false);
  msg_logger_ptr_tx = new InfoLogger<PplscSentMsgLogElement>("PplscSentMessages.bin"
      , true, false);
```

`Pplsc`'s constructor creates information loggers to record receiving and sending messages and selects only txt based logging.

```
msg_logger_ptr_rx->addRecord(PplscRecvMsgLogElement(msg.GetSource(), msg.GetWaveform
    (),gMsg->numofMessagesSend));
```

For each received message or sent message, the corresponding logging element is created and *addRecord* method of the corresponding logger is invoked with this element. For example, a log for the received messages is created with the following call to *msg_logger_ptr_rx*.

```
msg_logger_ptr_rx->addRecord(PplscRecvMsgLogElement(msg.GetSource(), msg.GetWaveform
    (),gMsg->numofMessagesSend));
```

9 Support

Support for Tuscarora is provide mainly (at this point) through email. Please email tuscarora-support@samraksh.com

List of References

[1] ns3-dce: Direct code execution. http://www.nsnam.org/overview/projects/direct-code-execution/.

[2] Cmake: An open-source cross-platform family of tools. https://cmake.org/.