FREEDM Documentation

Release 2.0.0

S&T DGI Team

March 31, 2015

Contents

CHAPTER 1

Intro

This tutorial covers the installation and use of the FREEDM DGI 2.0.0. You should acquire the DGI from this link. If you choose to checkout code from the git repository, you should understand that the code you are pulling is experimental and unsupported.

This package includes only the DGI. A separate program from connecting the DGI to PSCAD is available at the PSCAD Repository on github. A typical simulation environment will include multiple DGI, typically three or five, which may run on the same Linux machine or on different machines. The simulation itself runs on Windows, either in PSCAD, in which case the DGI will communicate with the PSCAD Interface on a Linux machine, or in RSCAD with an RTDS, in which case the DGI will communicate with FPGAs. The DGI can also interact with physical devices that implement its *Plug and Play Adapter* protocol.

CHAPTER 2

Obtaining Support

If there is an issue with the installation or use of the FREEDM DGI software, please contact the DGI developers at freedm-dgi-grp at <spamfree> mst dot edu.

CHAPTER 3

Getting Started With DGI:

3.1 DGI Features

Blah blah

3.2 System Requirements

This section lists system requirements for the FREEDM simulation.

3.2.1 DGI

The DGI is tested on recent versions of popular GNU/Linux distributions and require components that can generally be installed from your package manager:

- ISO-compliant C++98 compiler (such as recent versions of GCC or Clang)
- CMake 2.6 or higher
- Boost 1.47 or higher, including binaries
- Python 2.7.x (and not higher)
- Google Protocol Buffers 2.4.1 or higher (older versions may work)
- NTP daemon (not required if only running the PSCAD interface)

3.2.2 Boost

The recommended method to install Boost is through your system's package manager. If your distribution does not package a sufficiently recent version of Boost, refer to the *Boost documentation* for instructions on compiling a newer version of Boost and installing it to */usr/local/boost*. Some binary libraries are required: CMake will tell you which ones you are missing when you attempt to compile the DGI.

Additionally, set the BOOST_ROOT environment variable to include the Boost directory:

```
echo 'export BOOST_ROOT=/usr/local/boost/' >> ~/.bashrc
```

And restart your shell.

3.2.3 Python

FREEDM DGI uses Python 2.7 for the time being; newer versions will not work. You must have a binary in your path named *python2*; this is generally */usr/bin/python2*. If you don't have this binary then */usr/bin/python* is probably python2 and you can safely create a symlink:

```
ln -s /usr/bin/python /usr/bin/python2
```

3.2.4 NTP

All DGI nodes must run an NTP daemon (such as chrony or ntpd) to synchronize their clocks. The DGI runs its own fine-grained clock synchronizer designed to correct very small clock skews. However if two nodes' system clocks are off by a big amount, like one a minute or more, then there is no chance that the DGI's clock synchronizer will work and you will be unable to form groups.

3.2.5 PSCAD Simulation Requirements

- PSCAD v4.4 Educational Edition
- GFortran compiler
- sys/socket.h
- netdb.h

3.2.6 Network

Each computer that will run the DGI must have a unique hostname and each other computer that will run a DGI must be able to reach that machine by that hostname. You can check to see if the hostnames are properly configured with a simple ping test. On machine A:

```
$ hostname
raichu.freedm
```

On machine B:

```
$ ping raichu.freedm
PING raichu.freedm (216.229.90.108) 56(84) bytes of data.
64 bytes from ________________________________ (xxx.xxx.xxx.xxx): icmp_seq=1 ttl=64 time=0.348 ms
64 bytes from ________________________________ (xxx.xxx.xxx.xxx): icmp_seq=2 ttl=64 time=0.295 ms
64 bytes from ________________________________ (xxx.xxx.xxx.xxx): icmp_seq=3 ttl=64 time=0.264 ms
^C
--- raichu.freedm ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5007ms
rtt min/avg/max/mdev = 0.264/0.294/0.348/0.033 ms
```

The are many methods to achieve this. The easiest method is documented in *Network Configuration*

3.3 Building The DGI

This section gives an outline of the steps required to install the DGI. It is meant to be used as a reference once the remainder of the tutorial has been understood:

- navigate to the base directory (FREEDM/Broker)

- `cmake -DCMAKE_BUILD_TYPE=Release`
- `make`
- `cp config/samples/\* config/`
- Configure the DGI
- run the executable (./PosixBroker)

Make sure you have installed all the *System Requirements*.

The DGI builds its executable `PosixBroker` as part of the build process.

To start the build process, navigate to where you have extracted the DGI source code bundle. There are several folders in the extracted bundle, which include documentation for the DGI as well as the DGI build folder. First, change in to the DGI Broker directory which contains the bulk of the DGI code.:

```
$ pwd
/home/scj7t4/FREEDM/Broker
```

The makefile for the DGI is created by invoking `cmake` in the Broker folder:

```
cmake -DCMAKE_BUILD_TYPE=Release
```

CMake verifies that Boost is installed and properly configured. If everything goes well, there will now be a `makefile` in the current directory and the DGI can be built by invoking make:

```
make
```

At this point the DGI will build. If you ever make any changes to the DGI code, you can always include them by invoking `make` again. When the build process completes there should be a `PosixBroker` executable in the `Broker` folder. However, in order to use the DGI, we must first configure it.

Go on to *Network Configuration*

3.4 Network Configuration

In order for clients to correctly connect to each other, each client must have a *single, globally unique* name. To accomplish this in DGI, we use a combination of the hostname of the node and port the DGI instance listens on. It is critical that each DGI has only one name globally that all nodes refer to. *This requires correct configuration on all machines.*

3.4.1 Assigning A Hostname To A Computer

DGI loads the hostname automatically: there is no need to specify a hostname in `freedm.cfg`. The hostname DGI loads can be found by running `hostname`; here is an example from our testing system:

```
$ hostname
raichu.freedm
```

The hostname is specified in /etc/hostname as the only contents of that file. When you set the hostname you must restart the machine for it to take effect.

Each machine in the system must have a unique host name specified in `/etc/hostname`

You can confirm that the hostname has been loaded correctly by invoking *./PosixBroker -u* (again, on raichu.freedm):

```
$ ./PosixBroker -u
raichu.freedm:1870
```

3.4.2 Making Other Nodes Reachable

Each node must also know how to reach each other node by the name you assigned them in the `/etc/hostname` file. This is done using `/etc/hosts`. Be sure to take special care to ensure that each name in /etc/hosts is exactly as you specified in each of the `/etc/hostname` files.

Here is an example of a well done `/etc/hosts` file:

```
TS7800-4:~# cat /etc/hosts
127.0.0.1 localhost

192.168.100.11 MAMBA1
192.168.100.12 MAMBA2
192.168.100.13 MAMBA3
192.168.100.14 MAMBA4
192.168.100.15 MAMBA5
192.168.100.16 MAMBA6
192.168.100.17 TS7800-1
192.168.100.18 TS7800-2
192.168.100.19 TS7800-3
192.168.100.20 TS7800-4
192.168.100.21 TS7800-5
192.168.100.22 TS7800-6
```

- Each line is of the form "ipaddress hostname".
- The first entry in the file should be "127.0.0.1 localhost"
- The machine should appear in its own hosts file.
- Make sure that each machine you want to reach is defined in the hosts file.
- Make sure the hostname for each machine is **exactly** how you specified it in that machines `/etc/hostname`

Restart the machine to make sure the changes are applied correctly.

Once the network has been configured, you can go on to *Configuring The DGI*

3.5 Configuring The DGI

3.5.1 Configuration Files

You will need to copy some sample configuration files from Broker/config/samples/ into Broker/config/ and then edit them to your needs.

The following files are mandatory:

- `Broker/config/freedm.cfg`
- `Broker/config/logger.cfg`
- `Broker/config/timings.cfg`

Alternate locations for all configuration files can be specified in freedm.cfg, and an alternate location for freedm.cfg can be specified when running DGI: *./PosixBroker –config config/alternate-freedm.cfg*

3.5.2 freedm.cfg

`freedm.cfg` is the main configuration file for the DGI. The most important settings here specify the hostnames and ports of the other DGI to attempt to form groups with, and the port to use for outgoing communications with other DGI. Be very careful to ensure that each hostname you specify here is resolvable on your network. We recommend simply using the same port for every DGI in the network; the sample configuration uses port 51870 for every DGI and you probably will not need to change this. Start by copying the sample *freedm.cfg* from the samples directory down into the main config directory. A freedm.cfg should look something like this:

```
# Add some hosts to the peer list
add-host=raichu.freedm:1870
add-host=manectric.freedm:1870
add-host=zapdos.freedm:1870
add-host=thundurus.freedm:1870
add-host=raikou.freedm:1870
add-host=zapdos.freedm:1871
#add-host=raikou.freedm:1871

# Specify device name followed by ':' followed by device type.
# Device type must now be specified - it is no longer optional!
#add-device=fid77:Fid

# The address used by the DGIs for communication.
address=0.0.0.0
port=1870

# Global verbosity. Higher is more output.
# See logger.cfg for verbosity levels and per-file configuration.
verbose=4

#adapter-config=config/adapter.xml
# Host and port that the PSCAD/RTDS client will connect to.

# Filename of the FPGA message specification.
# Used only when compiled with DUSE_DEVICE_RTDS=ON

# Force this DGI's UUID to the specified value, instead of relying on the
# autogenerated UUID. Note that this option cannot be set by command line.
# Important: Each host's UUID must be unique.
#setuuid=36F3585E-F78C-4C52-AF8D-C6A78A27C831

#adapter-config=config/adapter.xml
#device-config=config/device.xml
#topology-config=config/topology.cfg
```

Lines with beginning with # are not parsed by the configuration file loader.

You should edit the *freedm.cfg*, adding one *add-host* directive for each DGI you wish to run, and making sure the port is set to the desired value. In the example, the DGI will coordinate with the other DGIs listed in *add-host*. An *add-host* directive for the current instance of the DGI will be ignored, so you can write the file once and copy it to all your machines, if you desire.

After building the DGI you can run *./PosixBroker -u* to print the ID of the current DGI and ensure it exactly matches an *add-host* directive of each peer DGI. For example:

```
[michael@victory-road Broker]$ ./PosixBroker -u
victory-road:51870
```

More information about the options for freedm.cfg can be found in *freedm.cfg options*.

3.5.3 logger.cfg

`logger.cfg` is useful for overriding the global verbosity level of the simulation (specified in freedm.cfg) for specific source files. This is primarily useful for developers at Missouri S&T, or those creating their own modules. You can leave this file unaltered.

More information about configuring the logger can be found in *Using The DGI Logger*.

3.5.4 timings.cfg

`timings.cfg` contains timings for the DGI; we recommend simply pointing to one of the provided timings samples in freedm.cfg; see *Configuring Timings*. Each host in the simulation must use the same timings which must be appropriate for the slowest DGI in the system, so if a system has two Core-2 computers and a TS-7800, all should use the slow timing set.

file	minimum CPU	Number of DGI	Cumulative time
timings-slow-5.cfg	TS7800	<=5	4950 ms
timings-slow-10.cfg	TS7800	<=10	7750 ms
timings-slow-30.cfg	TS7800	<=30	19150 ms
timings-fast-5.cfg	P4 2.4Ghz	<=5	1240 ms
timings-fast-10.cfg	P4 2.4Ghz	<=10	1990 ms
timings-fast-30.cfg	P4 2.4Ghz	<=30	4770 ms

Where minimum CPU is the performance of the slowest DGI in the group. Cumulative time is the time the DGI will take to configure a group, collect the global state and perform 10 migrations, before checking the system configuration again.

3.5.5 Test Your Configuration

At this point, you should be able to run the DGI. The DGI's should form a group, however, because devices have not been configured yet, they won't manage any devices or interact with simulations.

When the DGI is running, it will log various messages to the screen. To verify that the DGI is working correctly, watch for the Group Management or Load Balance status messages that list the current group. It make take up to a minute for the first groups to form when the DGIs are started. This is what the status message looks like from Group Management:

```
[raichu.freedm] out:    - SYSTEM STATE
[raichu.freedm] out: Me: raichu.freedm:30000, Group: 1804289384 Leader:raichu.freedm:30000
[raichu.freedm] out: SYSTEM NODES
[raichu.freedm] out: Node: galvantula.freedm:30000 State: Up (In Group)
[raichu.freedm] out: Node: manectric.freedm:30000 State: Unknown
[raichu.freedm] out: Node: raichu.freedm:30000 State: Up (Me, Coordinator)
[raichu.freedm] out: Node: zapdos.freedm:30000 State: Up (In Group)
```

The message lists all the processes in the system. Ideally, your message should say that all the hosts defined in your *freedm.cfg* should be listed as "In Group". A similar message is also logged by the Load Balancing module:

```
[galvantula.freedm] out:        ------- LOAD TABLE (Power Management) -------
[galvantula.freedm] out:        Net DRER (00):  0.00
[galvantula.freedm] out:        Net DESD (00):  0.00
[galvantula.freedm] out:        Net Load (00):  0.00
[galvantula.freedm] out:        -----------------------------------------
[galvantula.freedm] out:        SST Gateway:    0.00
[galvantula.freedm] out:        Net Generation: 0.00
[galvantula.freedm] out:        Predicted K:    0.00
```

```
[galvantula.freedm] out:        ------------------------------------------
[galvantula.freedm] out:        (NORMAL) galvantula.freedm:30000
[galvantula.freedm] out:        (NORMAL) raichu.freedm:30000
[galvantula.freedm] out:        (NORMAL) zapdos.freedm:30000
[galvantula.freedm] out:        ------------------------------------------
```

In this case, the list only includes processes that are correctly configured.

If processes are missing, verify your *freedm.cfg* files: it is a common issue that a machine's hostname hasn't be correctly specified. If the issue persists, or a DGI appears and disappears from the list, consider selecting a different timing configuration – the one you selected may not be appropriate for your configuration.

Once you have verified you have correctly configured the DGI and your DGI can form a group, you should move on to *DGI Device Framework*.

CHAPTER 4

Interacting With Simulations and Physical Devices:

4.1 DGI Device Framework

The DGI supports interaction with physical devices such as SSTs and DESDs through its device framework. This framework can communicate with both simulated and real devices. An important distinction is that this communication framework is independent of the protocol used to communicate with other DGI instances.

Note: This documentation covers how to connect the DGI to physical power hardware, not other instances of the DGI.

All devices used by the DGI must be defined in the *device.xml* configuration file. If a device is not specified in this file, then the DGI cannot communicate with it. A tutorial for introducing new device types to the DGI can be found at *Creating a Virtual Device Type*. The device types supported by the DGI by default, as well as their properties, are included in the following table. These devices can be extended to contain more states and commands than those listed, and are intended as placeholders for the DGI's load balancing algorithm.

Device Type	States (Readable Values)	Commands (Writable Values)
Sst	gateway	gateway
Desd	storage	storage
Drer	generation	
Fid	state	
Load	drain	
Logger	dgiEnable	groupStatus

The communication protocol the DGI uses to communicate with its physical devices depends on the type of physical adapter configured to run with the DGI. For most cases, configuration of the DGI physical adapters requires modification of another *adapter.xml* configuration file. The adapter types supported by the DGI are included in the following table.

Adapter Type	Communication Protocol	Communicates With	Documentation
rtds	TCP/IP	RTDS / PSCAD	*RTDS Adapter*
pnp	TCP/IP	Physical Hardware	*Plug and Play Adapter*
fake	none	nothing	undocumented

Users that plan on using a PSCAD or RTDS simulation should go on to *RTDS Adapter* to configure the DGI and their simulation.

Users that plan to interact with a physical device should consider using the DGI's *Plug and Play Adapter* to communicate with their device.

See *Other Methods For Connecting the DGI to Physical Devices* for documentation about the adapter interfaces.

Users creating modules devices can use the device manager to manipulate devices. An overview of how modules can use devices, as well as details on the device manager, can be found at *Using Devices in DGI Modules*.

4.2 RTDS Adapter

All communication between the DGI and physical devices is done through a set of classes called adapters. An adapter defines a communication protocol that the DGI uses to connect to real devices. The DGI can contain multiple adapters, and each adapter can use a different communication protocol. This allows, for instance, the DGI to talk to both a power simulation and real physical hardware at the same time. The DGI comes with multiple adapter types that can be used to interface with physical devices (either real or simulated). Configuration of the DGI depends on the type of adapter that is used. For almost all cases, we recommend use of the RTDS adapter. Despite its name, this adapter works with both RTDS and PSCAD, and has also been used to communicate with real hardware. Unless the user has extensive knowledge on the DGI, the RTDS adapter should be the default choice. The following sections document the two adapter types provided by the DGI team (RTDS and Plug and Play), as well as how to create a new adapter should neither option be viable.

The RTDS adapter was designed to allow the DGI to communicate with the FPGA connected to the RTDS rack at Florida State University. However, it has also become the default choice for connecting the DGI to a PSCAD simulation, and is a viable option when interfacing the DGI with real physical hardware. This is the adapter type recommended by the DGI development team. Throughout this section, the term device server will be used to refer to the endpoint the DGI communicates with while using the RTDS adapter. When using an RTDS simulation, the device server would be the FPGA server connected to the RTDS. When using PSCAD, the device server is a piece of code called the simulation server that runs on a linux computer. And when using real hardware, the device server refers to the controller attached to the physical device.

4.2.1 Configuration

The DGI can be configured to use one or more RTDS adapters through modification of the adapter specification file `Broker/config/adapter.xml`. This file contains the specifications for all adapters in the systems, and as such can contain multiple adapter specifications. Each adapter specification is located under the common **<root>** tag under its own **<adapter>** subtag. The following tutorial will cover how to create a new RTDS adapter specification for an RTDS simulation with the following devices:

Device Type	States (Readable Values)	Commands (Writable Values)
FID1	status	
FID2	status	
DESD7	I (current), V (voltage), T (temperature), SoC (state of charge)	RoC (rate of charge)

An important note is that this specification does not contain DESD1 through DESD6, which are presumed to exist. Each DGI instance has its own adapter specification file, and each file should contain the devices associated with its associated DGI. In this example, we can assume that the specification file is for DGI #7 which has control over DESD #7. Therefore, there must be similar (but not identical) configuration files for the other 6 DGI instances. Unlike SCADA systems, the DGI is distributed and each DGI instance only has knowledge of a subset of the devices in the system.

The DGI can communicate with physical devices which have been defined in its device configuration file. For a tutorial on how to define new virtual devices within the DGI, refer to the tutorial *Creating a Virtual Device Type*.

Warning: When running a power simulation, only a subset of the devices should be sent to each DGI!

The first step in our example is to create a new RTDS adapter that contains the devices in the simulation. Each adapter must have a type and a unique name, which must be specified within its associated **<adapter>** tag. For our example:

```
<root>
    <adapter name = "ExampleAdapter" type = "rtds">
        <!-- (comment) the adapter will be defined here -->
    </adapter>
</root>
```

The `name` and `type` properties for the **<adapter>** tag are not optional. In addition, the name must be unique (if there are multiple adapters running at once) and the type must be rtds (since we are defining an RTDS adapter). In the current version of the DGI, the name field is not used by RTDS adapters and can be set to any arbitrary, but unique, value.

Now the RTDS adapter has been defined, but the DGI has not been told the endpoint for the device server that contains the simulation data. Because the RTDS adapter communication protocol utilizes TCP/IP, the endpoint is specified using a hostname and port number. If the device server is located on the computer with hostname FPGA-Hostname listening for connections on port 52000, the endpoint can be specified using an **<info>** tag as follows:

```
<root>
    <adapter name = "ExampleAdapter" type = "rtds">
        <info>
            <host>FPGA-Hostname</host>
            <port>52000</port>
        </info>
        <!-- (comment) this specification is still incomplete! -->
    </adapter>
</root>
```

This enables the DGI to connect to the device server, but still does not tell the DGI the format of the state and command packets used during communication with the server. Both packet formats must be specified in the adapter configuration file. Specification of both packet formats is similar, but our tutorial will consider the state packet first.

The state packet format is defined under **<adapter>** using the **<state>** tag. Each state contained in this packet is defined as a separate **<entry>** subtag which contains all the information the DGI needs to parse the state packet. There are several require properties for each state entry:

1. The **index** of the state in the the packet. Indices range from 1 to the number of floating point values in the packet, and each index must be unique. If a state entry is given a value of i, then the DGI assumes that that state will be the i:sup:*th* floating point value in the state packet. Therefore, when the DGI needs to access the state, it will read the state packet starting from a byte offset of 4i.
2. The **signal** name for the state entry. For instance, a DESD device could have a current state which uses the signal identifier of I. This field tells the DGI that the state entry located at this index refers to a current value. All signal names must correspond to some **<state>** tag of a **<deviceType>** in the *device.xml* configuration file, see *Creating a Virtual Device Type*.
3. The **type** of physical device that owns the signal. For instance, a current value could refer to either the current at a generator or the current at a battery. The type field makes it explicit as to which sort of device the current is associated with. This allows the DGI to create an appropriate type of virtual device to handle storage of the state entry. All type identifiers must correspond to some **<id>** tag of a **<deviceType>** in the *device.xml* configuration file.
4. The **device** name of the device that owns the signal. In our example, there are two FID devices and so there will be two state signals with the value *status* that belong to a device of type *FID*. This field disambiguates which of the two FID devices the state belongs to. In addition, the DGI can access individual devices through use of this device name. For this reason, the name must be unique within the adapter specification file.

For our example, the state configuration would be:

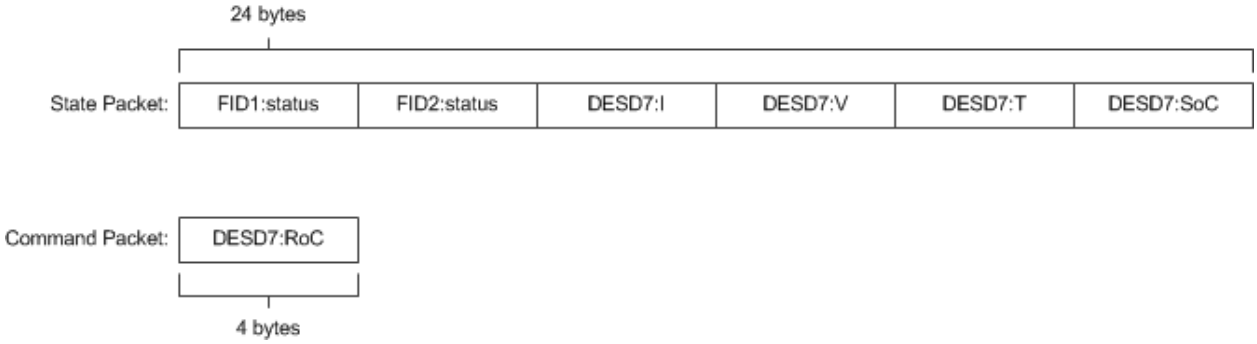
```
<root>
    <adapter name = "ExampleAdapter" type = "rtds">
        <info>
            <host>FPGA-Hostname</host>
            <port>52000</port>
        </info>
        <state>
            <entry index = 1>               <!-- The index must appear together with the entry tag --
                <type>Fid</type>            <!-- This defines the device type (from device.xml) -->
                <device>FID1</device>       <!-- This is the unique name / identifier -->
                <signal>status</signal>     <!-- This defines the state type (from device.xml) -->
            </entry>
            <entry index = 2>
                <type>Fid</type>
                <device>FID2</device>
                <signal>status</signal>
            </entry>
            <entry index = 3>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>I</signal>
            </entry>
            <entry index = 4>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>V</signal>
            </entry>
            <entry index = 5>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>T</signal>
            </entry>
            <entry index = 6>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>SoC</signal>
            </entry>
        </state>
        <!-- (comment) this specification is still incomplete! -->
    </adapter>
</root>
```

A similar specification must be done for the format of the command packet using the tag **<command>**. All of the required properties of states are also required for commands, and the XML format for both is identical. As such, the commands in our example lead to the final configuration file format:

```
<root>
    <adapter name = "ExampleAdapter" type = "rtds">
        <info>
            <host>FPGA-Hostname</host>
            <port>52000</port>
        </info>
        <state>
            <entry index = 1>
                <type>Fid</type>
                <device>FID1</device>
                <signal>status</signal>
            </entry>
```

```
            <entry index = 2>
                <type>Fid</type>
                <device>FID2</device>
                <signal>status</signal>
            </entry>
            <entry index = 3>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>I</signal>
            </entry>
            <entry index = 4>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>V</signal>
            </entry>
            <entry index = 5>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>T</signal>
            </entry>
            <entry index = 6>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>SoC</signal>
            </entry>
        </state>
        <command>
            <entry index = 1>               <!-- The index must appear together with the entry tag --
                <type>Desd</type>           <!-- This defines the device type (from device.xml) -->
                <device>DESD7</device>      <!-- This is the unique name / identifier -->
                <signal>RoC</signal>        <!-- This defines the command type (from device.xml) -->
            </entry>
        </command>
    </adapter>
</root>
```

This completes the RTDS adapter specification for our example. With this specification file, the command packet will be 4-bytes and contain a single command that corresponds to the rate of chargre for DESD7. The state packet will be 24-bytes and contain 6 separate floating point numbers. The following figure shows the exact format of both packets.



Both the **<state>** and **<command>** tags are required, even if there are no states or commands associated with a given adapter. For example, if this adapter did not contain the DESD device and instead contained the two FID devices, the sample configuration file would change to resemble:

```
<root>
    <adapter name = "ExampleAdapter" type = "rtds">
        <info>
```

```
            <host>FPGA-Hostname</host>
            <port>52000</port>
        </info>
        <state>
            <entry index = 1>
                <type>Fid</type>
                <device>FID1</device>
                <signal>status</signal>
            </entry>
            <entry index = 2>
                <type>Fid</type>
                <device>FID2</device>
                <signal>status</signal>
            </entry>
        </state>
        <command>
            <!-- The empty command tag must still be included -->
        </command>
    </adapter>
</root>
```

If the contents of the state tag are omitted, the DGI will never attempt to read from the TCP/IP socket it uses to communicate with the device server. Likewise, if the command tag is omitted, the DGI will never write a command packet to the device server. In both of these cases the communication protocol becomes unidirectional. However, in both cases, the **<state>** and **<command>** tags themselves must still be included as in above.

Configuration Errors

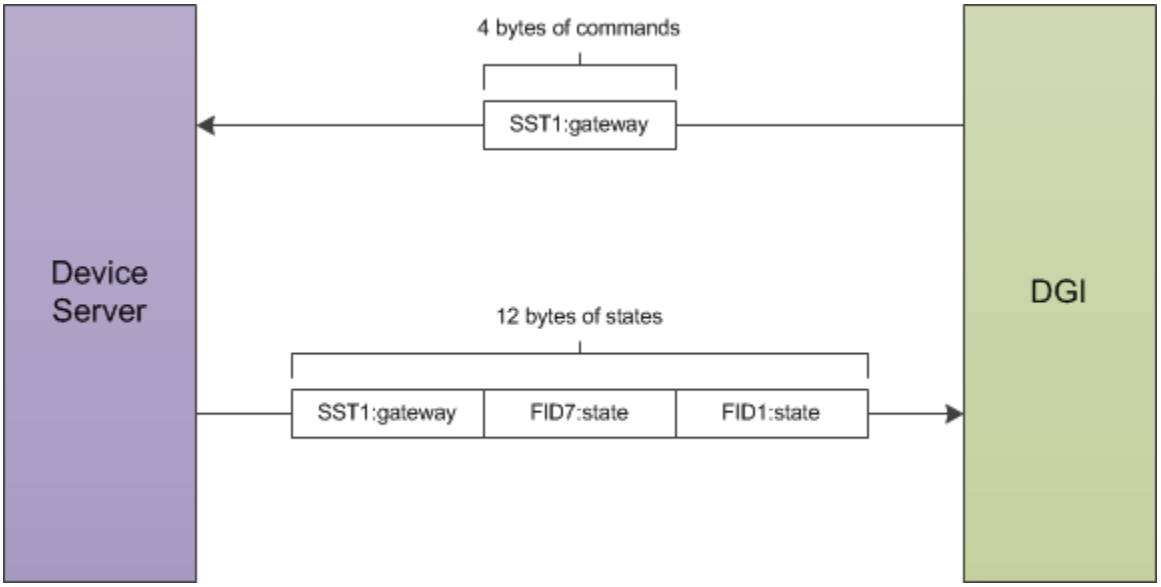
1. The name field for each adapter must be unique.
2. Each **<type>** specified during the state and command packet configuration must refer to the **<id>** of a **<deviceType>** found in the *device.xml* configuration file.
3. Each **<signal>** specified during the state or command packet configuration must refer to some **<state>** or **<command>** of its associated **<type>** in the *device.xml* configuration file.
4. When a device of a specific **<type>** is specified, all of its **<state>** and **<command>** values from the *device.xml* configuration file must appear in the adapter configuration file. It is impossible to use a subset of the states or commands of a device when using an RTDS adapter.
5. The complete state and command specification of each device must be contained within a single **<adapter>**. If a device spans multiple adapters, it will result in tremendously undefined behavior.
6. Indices for state and command entries must be unique, consecutive, and start counting from an initial index of 1.
7. All adapters must have a **<state>** and **<command>** subtag, even if the contents of the tags are empty.

4.2.2 Communication Protocol

The RTDS adapter uses TCP/IP to connect to a server with access to some set of physical devices. When the DGI runs using an RTDS adapter, it attempts to create a client socket connection to an endpoint specified in the adapter configuration file during startup. Once connected, it sends a periodic command packet to the server, and expects to receive a device state packet in return. The device server must be running and prepared to receive connections before the DGI starts when using an RTDS adapter. In addition, the DGI will always send its command packet before the device server responds with its state packet.

This communication protocol is very brittle. If the DGI loses connection to the server, it will not attempt to reconnect and after some time the DGI process will terminate. In addition, if the DGI receives a malformed or unexpected packet from the server, it will terminate with an exception. Therefore, this protocol should only be used on a stable network.

The following diagram shows on round of message exchanges in the communication protocol. It assumes that there are three states produced by the devices, and one command produced by the DGI.



Both the command packet from the DGI and the state packet from the device server contain a stream of 4-byte floating point numbers. Other data formats such as boolean or string values cannot be used with the RTDS adapter; all the device states must be represented as floats. The command packet must contain every command for the devices attached to the server, while the state packet must contain every device state. It is not possible to send a subset of the commands, or to send the values for different commands at different times. If the DGI has not calculated the value of a particular command, it will send a special value of 10^8 to indicate a NULL command. The device server must recognize and ignore the value of 10^8 when parsing the command packet received from the DGI. Likewise, the device server can use the value of 10^8 for device states which are not yet available when communicating with the DGI.

4.2.3 Running the Device Server

The steps discussed so far configure the DGI to connect to a device server utilizing a particular packet format. However, the device server must also be configured to expect the same packet format as the DGI. Because the RTDS adapter can be used for multiple power simulations, the configuration of the device server depends on the type of simulation being run. The details for configuration are thus delegated to the individual tutorials on how to run specific simulations.

If you are simulating with PSCAD, see *Running a PSCAD Simulation*.

If you are simulating with RTDS, see *Configuring RTDS*.

4.3 Running a PSCAD Simulation

When running a PSCAD simulation, three separate processes are required:

1. The PSCAD simulation running on a windows machine.
2. A simulation server connected to PSCAD running on a linux machine.
3. The DGI configured with an RTDS adapter and connected to the simulation server.

This tutorial will describe how to setup both PSCAD and the simulation server. Details on configuration of the DGI can be found in the section *RTDS Adapter*.

4.3.1 Required Files

The code required to run both PSCAD and the simulation server is stored in a repository separate from the rest of the DGI. This repository can be found on github and must be checked out or downloaded in order to proceed with this tutorial. In addition, it may be helpful to download an existing simulation from this list of files as a reference.

4.3.2 PSCAD Simulation

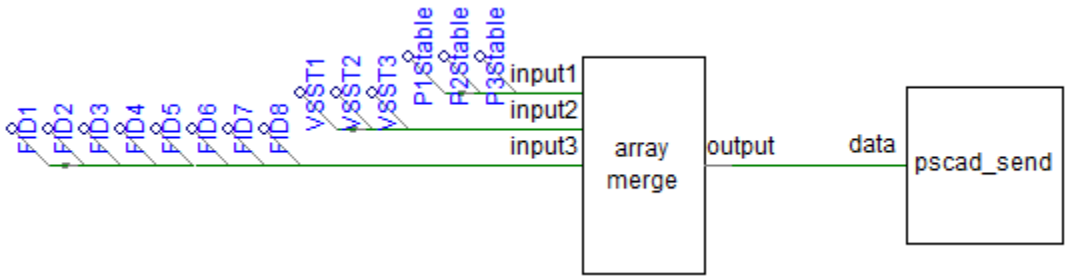
The files related to the PSCAD simulation are found in the `pscad` directory of the repository. The `.cmp` files are new components that must be imported into the PSCAD simulation. A component can be imported into PSCAD by right-clicking the Definitions tab in the project workspace and selecting the import definitions option. An instance of each component should also be added to the simulation by right-clicking on its definition, clicking create instance, and then pasting the instance inside the simulation schematic. For this tutorial, one instance of both *pscad_send* and *pscad_recv* is sufficient, but you will need two instances of the *array_merge* component.

These components contain Fortran scripts that require the code found in `psocket.c`. This C file should be placed in the same directory as your simulation `.psc` file. Both the scripts and C file were designed to work with the free GFortran compiler provided with PSCAD, and alternative compilers are unlikely to work. Specifically, it is unlikely that a compiler that does not translate the Fortran scripts into C during the compilation process will work.

Warning: PSCAD must be set to use the GFortran compiler or it will not be able to call the C code.

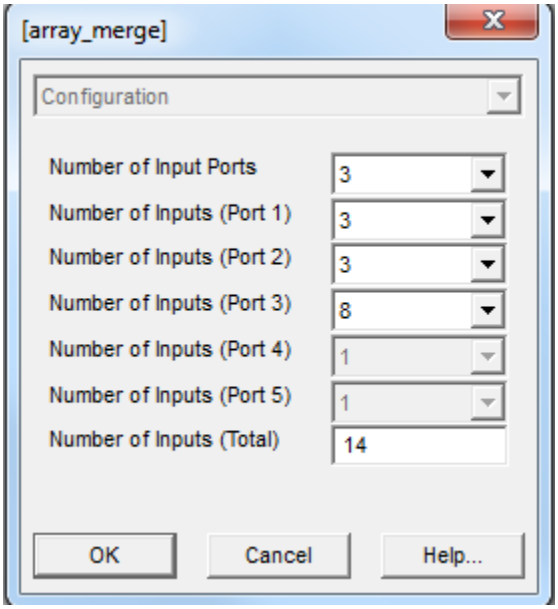
Sending Data using PSCAD_SEND

Create an instance of both the *pscad_send* and *array_merge* components and connect them as shown in the following diagram:



All of the data that will be sent out of PSCAD into the DGI has been assigned a signal name. These signals have been merged together using the default PSCAD `data merge` component found in the master library. Multiple data merge components had to be used to handle all of the data because each handles at most 12 inputs due to the limitations of PSCAD. These separate components were then combined into a single array through use of the `array merge` component that was imported from the git repository. The final output of the array merge component, which is a single large array that contains all the data that will be sent out from PSCAD, is used as an input for the `pscad send` component.

The first step to connecting the components as shown in the example is to configure the array merge component to have the correct number of input pins for the amount of data that will be sent to DGI. Determine how many signals will be sent from PSCAD, and then access the properties of the `array merge` component:

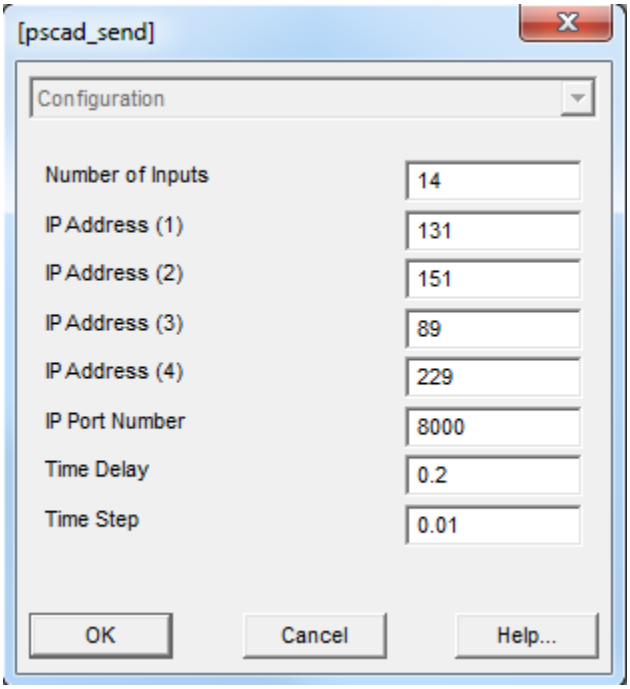


The structure of the array merge component is that it supports up to 5 input pins, and each input pin receives the output of a single data merge component. As such, the maximum amount of data that can be sent from PSCAD is 60. The first property, the number of input ports, should be set to the number of data merge components that are required for your data set. In the example, this value is three, and there are a total of three input pins (input1, input2, input3) available for use. If this property is changed, the number of pins on the component will also change when the properties window is saved and closed.

Each input port must then be configured to know the size of the data merge component connected to it. This is done through the next five number of input (port #) properties. For each one of these properties, the value should be set to the number of data points be merged on the associated input pin. For example, the number of inputs for port 3 in the example is set to 8 because there are a total of 8 signals put through the data merge component on the input3 port.

The final number of inputs total property will be the total number of signals put through the array merge component, or the sum of the preceeding five properties.

The *pscad_send* component must also be configured:



The number of inputs property should be set to the same value as that given for the *array_merge* component, and should be the total amount of data points being sent out of PSCAD.

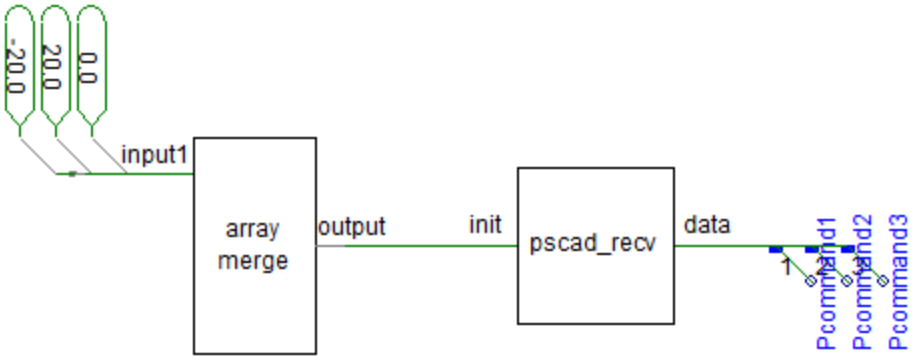
Each of the IP address fields refers to the IPv4 address of the simulation server associated with PSCAD. All simulations must have an associated linux machine running the simulation server code from the git repository. That server will store the current simulation state for the DGI to easily access. The IP and port fields from the *pscad_send* component should be set for the linux machine that will run the server. In the example, PSCAD will attempt to connect to the server located at 131.151.89.229:8000 during runtime.

The time delay property is used to delay the sending of data until after the simulation reaches its steady state. It should be set to the simulation time at which data should first be sent to the DGI. The time step property then refers to the frequency at which data is sent after this delay. At each multiple of the time step value, PSCAD will send the input signals to the simulation server.

Note: PSCAD will not send data to the simulation unless the simulation time is greater than the time delay specified in the pscad_send component.

Receiving Data with PSCAD_RECV

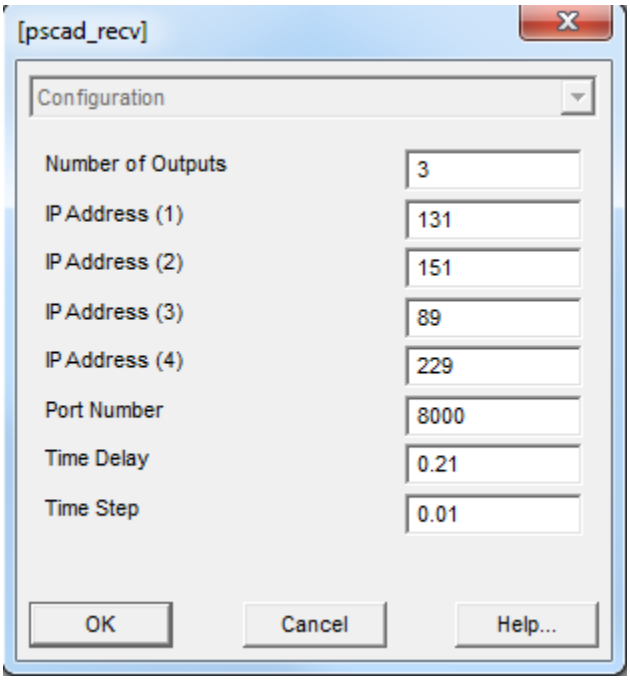
Create an instance of both the *pscad_recv* and *array_merge* components and connect them as shown:



All the data that will be received by PSCAD must be assigned an initial value. When the simulation begins, there is a configurable delay before PSCAD receives the first set of values from the simulation server. There will always be at least one simulation step where PSCAD has yet to receive these values from the server. In this case, the initial values defined in PSCAD will be used until the first bit of data from the simulation server arrives. These initial values are the inputs that go through the array merge component into *pscad_send*.

The output of *pscad_recv* is a large array that contains all of the data sent to PSCAD. Each element of the array must be individually accessed using the `data tap` component from the default PSCAD master library. Data tap components must be individually modified to access separate elements of the output array.

The properties of the array merge component are discussed in the previous section on sending data and will be skipped. For the *pscad_recv* properties:



The first property defines the number of elements in the output array, and should be set to the number of data points that will be received from the simulation server.

The next set of properties defines the IPv4 address of the simulation server in the same manner as the *pscad_send* component. Although this component is set to receive data, due to the nature of PSCAD it is impossible to maintain a stable socket over multiple simulation steps, and so the receive component connects as a client to the server and requests the next set of simulation commands. As such, the server endpoint must be specified even when receiving data. In the example, the *pscad_recv* component is configured to retrieve data from the simulation server located at

131.151.89.229:8000.

both the time delay and time step are the same as with the *pscad_send* component. The time delay prevents PSCAD from receiving data until after a certain simulation time, while the time step specifies how many simulation seconds are between two successive data reads. However, the time delay for the receive component should be set to a larger value than the one assigned to the send component to ensure that PSCAD always sends at least one value to the simulation server before it tries to read a command.

Note: Prior to the specified time delay, the output of the pscad_recv component is equal to the initial values provided as an input.

4.3.3 Simulation Server

The simulation server must run a compiled version of the code from the repository downloaded earlier. The repository can be compiled using the sequences of commands *cmake .* and *make* from the main repository directory. This will produce the simulation server executable which by default has the filename `driver`. To configure the simulation server, move all files from `config/samples/` into `config/` and then open `config/rtds.xml` to change its settings to match your simulation. The rest of this section will describe how to modify this XML file, as the other two configuration files will work with their default values.

The XML configuration file has a strict format illustrated in the following example:

```
<root>
    <adapter type="TYPE" port="PORT">
        <state>
            <entry index="INDEX">
                <device>DEVICE</device>
                <signal>SIGNAL</signal>
                <value>VALUE</value>
            </entry>
        </state>
        <command>
            <entry index="INDEX">
                <device>DEVICE</device>
                <signal>SIGNAL</signal>
                <value>VALUE</value>
            </entry>
        </command>
    </adapter>
</root>
```

There is a main **<root>** tag that contains the complete configuration of the server. Under this, there will be one **<adapter>** tag for each client (simulation and DGI instances) connected to the server. Each **<adapter>** must be specified under **<root>** and assigned both a type and a port number. The port number must be unique and defines which port number that client will connect to when communicating with the simulation server. For instance, if the PSCAD simulation has been configured to connect to port 8000 as in the example, then the adapter with `port=8000` define the configuration for talking with PSCAD. The type must be either *simulation* or *rtds* and refers to the adapter type the simulation server uses to communicate with the client. *Simulation* refers to the PSCAD simulation, while *RTDS* refers to an instance of the DGI (which uses its RTDS adapter to communicate with PSCAD).

There is no hard limit on the number of adapters that can be specified. There is also no limit on how many instances of a specific adapter type can be specified. It is possible, for example, to create twelve different simulation adapters that communicate with twelve concurrent PSCAD power simulations. The variables from all simulations will be stored together in the simulation server and be accessible to all of the DGI.

Each adapter also follows a strict format:

```
<state>
    <entry index="INDEX">
        <device>DEVICE</device>
        <signal>SIGNAL</signal>
        <value>VALUE</value>
    </entry>
</state>
<command>
    <entry index="INDEX">
        <device>DEVICE</device>
        <signal>SIGNAL</signal>
        <value>VALUE</value>
    </entry>
</command>
```

The **<state>** tag refers to a value that originates from PSCAD, while the **<command>** tag refers to a value that originates from some DGI instance. It is not possible to omit either tag, and neither tag can be empty. This means that you cannot run a simulation where PSCAD receives no commands, or one of the DGI instances does not receive a state. All clients connected to the simulation server must both send and receive data.

Note: If you run a simulation where one client does not send or receive data, it should send a fake dummy value that will not be used by the other end.

Each entry for a state or command has the same format:

```
<entry index="INDEX">
    <device>DEVICE</device>
    <signal>SIGNAL</signal>
    <value>VALUE</value>
</entry>
```

The index refers to the index of the data if it were sent as a byte stream of data. Even if the adapter does not send and receive byte streams, a valid index must be specified. The index must begin with 1 and contain unique, consecutive integers.

The "device" and "signal" tags are both required and generate a unique identifier for an entry in the device table. If two entries are in the same table (state or command) and have the same "device" and "signal" pair, even if they are in different adapters, they refer to the same memory location.

The "value" tag is optional and specifies an initial value for the "device" and "signal" pair. This value will be set in the device table when memory is allocated for the device signal. If the same device signal is specified in multiple entries, the value only needs to be specified once. It does not matter where the value is specified. A value can also be specified multiple times without error, so long as all of the tags contain the same numeric value. If a device signal does not have an initial value, the tag can be omitted.

Note that a device signal must be specified in each adapter that uses it. This means there will be a large number of duplicate "device" "signal" pairs in the specification. Spend time when writing the device specification file to make sure all of these duplicate entries have the same spelling, as otherwise the device table will have an inconsistent state.

4.3.4 Running the Simulation

1. Start the simulation server with the command ./driver.
2. Run each instance of the DGI that connects to the simserv
3. Run the PSCAD simulation with the green arrow on the top toolbar.
4. Select Yes if the simulation warns that it will use a large amount of memory.

5. Allow the firewall exception if windows complains about the processes' internet usage

Note: You will need administrative access on the windows machine running PSCAD to allow the firewall exception for PSCAD.

A turning gear icon will appear in the lower-right corner to indicate the simulation is running. After some time, the current simulation time will appear in this corner below the gear icon. If the simulation time never appears, and the message log does not indicate a compilation error, then the simulation has been misconfigured and cannot connect to the simulation server. If the time advances, the connection has been formed.

4.3.5 Common Errors

User Source File does not exist

psocket.c is not in the same directory as simulation.psc - verify the files have been kept together, or obtain a new version of psocket.c from the repository.

psocket.c:7:19: error: netdb.h: No such file or directory

netdb.h is not in the GFortran/version/include folder. You must obtain a version of this file and place it in this folder.

psocket.c:8:24: error: sys/socket.h: No such file or directory

socket.h is not in the GFortran/version/include/src folder. You must obtain a version of this file and place it in this folder.

The simulation stalls or stops responding

The TCP Sockets are set to block until a connection is made to the Interface. If the simulation stalls, either the Interface code is not running or the pscad_send and pscad_recv components have not been configured to use the correct Interface IPv4 address. Run the Interface, or correct the IP Address and Port Number fields.

4.4 Configuring RTDS

4.5 Physical Topology

DGI can react to changes in the physical topology if the PhysicalTopology code is enabled. Currently, this code is only available in the physical topology side-branch. It should be a part of master (and a release) soon.

Physical Topology is based on vertices (SSTs) and edges. Edges can have 0 or more FIDs. These FIDs determine the availability of the edge: all FIDs on the edge must be closed (Unless the edge does not have an FID). If there is no series of edges which connect two SSTs, they should not be in a group together.

4.5.1 Physical Topology Configuration

Topology is configured in `config/topology.cfg`. A topology config file looks like this:

```
edge a b
edge b c
edge c a
sst a raichu.freedm:1870
sst b manectric.freedm:1870
sst c galvantula.freedm:1870
fid a b FID1
fid a b FID4
fid b c FID2
fid c a FID3
```

Each line of the file is composed of a statement type, and then a series of keywords that are necessary to construct that object.

- **edge** - A physical connection between two SSTs. An edge indicates there is a direct physical connection between two SSTs through a power line or similar object. An edge is followed by strings that represent the two verticies they connect. For convenience, the DGI controlling the vertex is set by the _sst_ statement. Only one edge for each vertex pair needs to be named, and all edges are bidirectional by default (that is, `edge a b` also gives you `edge b a`)
- **sst** - A vertex. This statement is followed by two strings: the first is the name of the vertex. This is the name used in the `edge` and `fid` statements. The second string is the UUID of the DGI that controls that vertex.
- **fid** - A control for an edge. This statement is controlled by three strings: first two strings are the edge that this FID controls, which is named in the same was as the edge above, which two vertex names. The third string is the name of the device which controls this edge. A device can control multiple edges and multiple devices can control one edge.

The topology configuration file is specified by adding the `topology-config` option to ``freedm.cfg`. For example, this line in a `freedm.cfg` enables physical topology:

```
topology-config=config/topology.cfg
```

The topology configuration file should be the same on all DGI peers.

4.5.2 Expected Group Management Behavior

If there is no topology configuration file specified in *freedm.cfg* then the physical topology feature is disabled and DGI will group with all available peers.

If a topology configuration file is specified then the DGI will only group with nodes that it deems to be reachable. If the FID state changes so that a node is no longer reachable then DGI will remove that peer from the group. The peer does not immediately receive the notification it has been removed so it will appear to remain in the group for an additional round; however, no other DGI will respond to its requests (so no migrations will occur) and it will leave during the next Group Management phase.

4.5.3 Physical Topology Implementation

The FID state is passed to peers using the "Are You Coordinator" (AYC) response message: The message will be received by the coordinator from any node that wants to merge groups and any node that wants to remain in a group with that coordinator. The coordinator combines all the reported FID states with its own and then runs a breadth first search (BFS) on the specified topology. Any edge where a controlling FID is marked as open is not used to determining physical reachability. The BFS returns a set of peers it has determined to be connected to the coordinator. The coordinator then interacts with the reachable nodes, ignoring any peers that are not reachable. Each round a peer may provide new FID state to show that they may now be reachable. The held state is wiped each time the BFS is run: an edge is only held open if the FIDs controlling it are consistently reported to coordinator.

4.6 Creating a Virtual Device Type

All physical device types (SST, DESD, FID, etc) must have a corresponding virtual device type defined in the DGI. This virtual device class tells the DGI modules how they can interact with the physical device. Virtual device types are defined in a single XML file located at *config/devices.xml*. This file must either be created, or moved from the samples folder, when the DGI is first installed on a new computer.

Warning: The DGI cannot communicate with devices whose type has not been defined in the device.xml configuration file.

4.6.1 Example Device Definition

When a new device type is introduced to the system (such as a new generation of SST), a new virtual device must be defined in this XML configuration file. This tutorial will describe how to modify the device.xml file to introduce a new virtual device type to the DGI. A DESD device with the following properties will be used as an example:

Device Type	States (Readable Values)	Commands (Writable Values)
DESD	Current, Voltage, Temperature, State of Charge	Charge Rate

This sample device meters its internal current, voltage, temperature, and amount of charge. A DGI module can also issue a command to change the charge rate to make the battery charge or discharge. All physical devices should have specifications similar to this sample DESD device, as the DGI's interaction with devices is limited to reading states and issuing commands.

Note: The DGI does not support non-numeric values for devices. For instance, the DESD could not have a manufacturer state as the name of a manufacturer is non-numeric.

First examine the structure of the sample configuration file *config/samples/device.xml*.

There is a **<root>** tag which contains several **<deviceType>** subtags. This **<root>** tag is required for all device.xml files, and each device type must be defined under **<root>** in its own **<deviceType>** subtag. To define a new virtual device, the first step is to append an additional **<deviceType>** subtag under **<root>**. If no other devices are defined, then for our tutorial the content of the *device.xml* file should resemble:

```
<root>
    <deviceType>
        <!-- (comment) our virtual DESD will be defined here -->
    </deviceType>
</root>
```

All the properties of the physical device must be defined under its associated **<deviceType>** subtag. The only required property for a physical device is a unique identifier to differentiate it from other devices. In our case, we are defining a generic DESD device, and so the unique identifier will simply be the string *DESD*. When the DGI needs to access a set of physical devices, it will use this unique identifier in the code. The unique identifier is defined using an **<id>** tag as follows:

```
<root>
    <deviceType>
        <id>DESD</id>
    </deviceType>
</root>
```

At this point the device has been defined and can be used within the DGI, as although the definition is incomplete for our sample DESD device, all properties of a virtual device other than its unique identifier are optional. However, the sample DESD device has a large number of readable states. Each one of these states must be defined using a separate

<state> tag. All of the states must be listed in separate **<state>** tags, so **<state>** will appear four times for our DESD device with four unique states:

```
<root>
    <deviceType>
        <id>DESD</id>
        <state>current</state>
        <state>voltage</state>
        <state>temperature</state>
        <state>charge</state>
    </deviceType>
</root>
```

Again, these string identifiers will be used by the DGI when it attempts to read the current internal state of our new DESD device. The last requirement to finish the definition of our virtual device is to list all of its commands. Commands are specified using a **<command>** tag, and each command must appear within its own tag in the same manner as the states:

```
<root>
    <deviceType>
        <id>DESD</id>
        <state>current</state>
        <state>voltage</state>
        <state>temperature</state>
        <state>charge</state>
        <command>chargeRate</state>
    </deviceType>
</root>
```

When a state or command consists of multiple words, the recommended approach for its unique identifier is to remove the spaces and capitalize the first letter of each word as in the case of chargeRate. This will reduce the number of potential errors that can be generated by the BOOST XML parser that reads the device.xml configuration file. With this, the device specification for the virtual DESD is complete. It would now be possible to connect the DGI to an actual DESD device using the tutorial on connecting the DGI to physical devices, *Other Methods For Connecting the DGI to Physical Devices*.

4.6.2 Devices without States or Commands

Not all devices have both states and commands. A second brief example of an FID will illustrate how to define a device that doesn't have any commands. This device can still be used by DGI modules to read the state of the physical system, but the DGI is unable to control the behavior of the device. Consider the following sample device:

Device Type	States (Readable Values)	Commands (Writable Values)
FID	status (open, closed)	none

An FID has no commands as it cannot be controlled. Instead, the status of the FID (whether it is opened or closed) is used by the DGI to determine the current topology of the physical system. When a device contains no commands, the **<command>** tag should be omitted entirely from the device specification. As such, the *device.xml* configuration for this device would be:

```
<root>
    <deviceType>
        <id>FID</id>
        <state>status</state>
    </deviceType>
</root>
```

In the same manner, a device with no states can also be defined through omission of all the **<state>** tags.

4.6.3 (Advanced) Virtual Device Inheritance

This section is primarily intended for computer scientists with a background in programming. Virtual devices support inheritance, and one device definition can inherit from any number of other devices. This can be useful to allow for more powerful queries over devices in DGI modules.

For example, a PVArray (solar panel) is a more specific form of a DRER (generator). A DGI module might want to make a query about the total amount of generation in the system, in which case it would request all instances of the DRER device. However, another module might want to determine the current amount of solar generation, in which case it would request all instances of a PVArray. Because a PVArray must be selected for both of these queries, it must recognize both the DRER and PVArray identifiers. We have chosen to use inheritance to support this functionality. Consider the following device specifications:

Device Type	States (Readable Values)	Commands (Writable Values)
DRER	real power output	none
PVArray	real power output	on / off

An **<extends>** tag can be used to allow one device type to inherit from another. For our example, the easiest way to define both devices would be:

```
<root>
    <deviceType>
        <id>DRER</id>
        <state>realPower</state>
    </deviceType>
    <deviceType>
        <id>PVArray</id>
        <extends>DRER</extends>
        <command>onOff</command>
    </deviceType>
</root>
```

In this case, the PVArray type inherits all the states and commands of the DRER type. When a PVArray device is created in the DGI, modules will be able to access its realPower state inherited from the DRER. In addition, the PVArray will respond to both the DRER and PVArray types when the DGI queries for devices. Note that the order of the type definitions is irrelevant in the *device.xml* configuration file; the PVArray could be defined before the DRER device without error so long as the type it inherits from is eventually defined.

There is no limit to the depth of the inheritance, or the number of types that can be inherited from. In addition, virtual devices do not have the diamond inheritance problem. Consider the following definitions:

```
<root>
    <deviceType>
        <id>A</id>
        <state>appearsOnce</state>
    </deviceType>
    <deviceType>
        <id>B</id>
        <extends>A</extends>
    </deviceType>
    <deviceType>
        <id>C</id>
        <extends>A</extends>
    </deviceType>
    <deviceType>
        <id>D</id>
        <extends>B</extends>
        <extends>C</extends>
    </deviceType>
```

```
</root>
```

This configuration file would create four virtual device types, with each device type having a single appearsOnce state. This example demonstrates three important points:

1. One device can inherit from multiple others (D extends both B and C).
2. There is no limit on the depth of inheritance (D extends A through B and C).
3. There is no diamond inheritance problem (D doesn't have two appearsOnce states).

For further information on how the DGI supports inheritance in virtual devices, refer to the code at `Broker/src/device/CDeviceBuilder.cpp` to see how the *device.xml* file is parsed.

4.7 Other Methods For Connecting the DGI to Physical Devices

The DGI can also communicate with devices using its Plug-N-Play module. Users can also create their own adapters.

4.7.1 RTDS Adapter

All communication between the DGI and physical devices is done through a set of classes called adapters. An adapter defines a communication protocol that the DGI uses to connect to real devices. The DGI can contain multiple adapters, and each adapter can use a different communication protocol. This allows, for instance, the DGI to talk to both a power simulation and real physical hardware at the same time. The DGI comes with multiple adapter types that can be used to interface with physical devices (either real or simulated). Configuration of the DGI depends on the type of adapter that is used. For almost all cases, we recommend use of the RTDS adapter. Despite its name, this adapter works with both RTDS and PSCAD, and has also been used to communicate with real hardware. Unless the user has extensive knowledge on the DGI, the RTDS adapter should be the default choice. The following sections document the two adapter types provided by the DGI team (RTDS and Plug and Play), as well as how to create a new adapter should neither option be viable.

The RTDS adapter was designed to allow the DGI to communicate with the FPGA connected to the RTDS rack at Florida State University. However, it has also become the default choice for connecting the DGI to a PSCAD simulation, and is a viable option when interfacing the DGI with real physical hardware. This is the adapter type recommended by the DGI development team. Throughout this section, the term device server will be used to refer to the endpoint the DGI communicates with while using the RTDS adapter. When using an RTDS simulation, the device server would be the FPGA server connected to the RTDS. When using PSCAD, the device server is a piece of code called the simulation server that runs on a linux computer. And when using real hardware, the device server refers to the controller attached to the physical device.

Configuration

The DGI can be configured to use one or more RTDS adapters through modification of the adapter specification file `Broker/config/adapter.xml`. This file contains the specifications for all adapters in the systems, and as such can contain multiple adapter specifications. Each adapter specification is located under the common **<root>** tag under its own **<adapter>** subtag. The following tutorial will cover how to create a new RTDS adapter specification for an RTDS simulation with the following devices:

Device Type	States (Readable Values)	Commands (Writable Values)
FID1	status	
FID2	status	
DESD7	I (current), V (voltage), T (temperature), SoC (state of charge)	RoC (rate of charge)

An important note is that this specification does not contain DESD1 through DESD6, which are presumed to exist. Each DGI instance has its own adapter specification file, and each file should contain the devices associated with its associated DGI. In this example, we can assume that the specification file is for DGI #7 which has control over DESD #7. Therefore, there must be similar (but not identical) configuration files for the other 6 DGI instances. Unlike SCADA systems, the DGI is distributed and each DGI instance only has knowledge of a subset of the devices in the system.

The DGI can communicate with physical devices which have been defined in its device configuration file. For a tutorial on how to define new virtual devices within the DGI, refer to the tutorial *Creating a Virtual Device Type*.

Warning: When running a power simulation, only a subset of the devices should be sent to each DGI!

The first step in our example is to create a new RTDS adapter that contains the devices in the simulation. Each adapter must have a type and a unique name, which must be specified within its associated **<adapter>** tag. For our example:

```
<root>
    <adapter name = "ExampleAdapter" type = "rtds">
        <!-- (comment) the adapter will be defined here -->
    </adapter>
</root>
```

The `name` and `type` properties for the **<adapter>** tag are not optional. In addition, the name must be unique (if there are multiple adapters running at once) and the type must be rtds (since we are defining an RTDS adapter). In the current version of the DGI, the name field is not used by RTDS adapters and can be set to any arbitrary, but unique, value.

Now the RTDS adapter has been defined, but the DGI has not been told the endpoint for the device server that contains the simulation data. Because the RTDS adapter communication protocol utilizes TCP/IP, the endpoint is specified using a hostname and port number. If the device server is located on the computer with hostname FPGA-Hostname listening for connections on port 52000, the endpoint can be specified using an **<info>** tag as follows:

```
<root>
    <adapter name = "ExampleAdapter" type = "rtds">
        <info>
            <host>FPGA-Hostname</host>
            <port>52000</port>
        </info>
        <!-- (comment) this specification is still incomplete! -->
    </adapter>
</root>
```

This enables the DGI to connect to the device server, but still does not tell the DGI the format of the state and command packets used during communication with the server. Both packet formats must be specified in the adapter configuration file. Specification of both packet formats is similar, but our tutorial will consider the state packet first.

The state packet format is defined under **<adapter>** using the **<state>** tag. Each state contained in this packet is defined as a separate **<entry>** subtag which contains all the information the DGI needs to parse the state packet. There are several require properties for each state entry:

1. The **index** of the state in the the packet. Indices range from 1 to the number of floating point values in the packet, and each index must be unique. If a state entry is given a value of i, then the DGI assumes that that state will be the i:sup:*th* floating point value in the state packet. Therefore, when the DGI needs to access the state, it will read the state packet starting from a byte offset of 4i.
2. The **signal** name for the state entry. For instance, a DESD device could have a current state which uses the signal identifier of I. This field tells the DGI that the state entry located at this index refers to a current value. All signal names must correspond to some **<state>** tag of a **<deviceType>** in the *device.xml* configuration file, see *Creating a Virtual Device Type*.

3. The **type** of physical device that owns the signal. For instance, a current value could refer to either the current at a generator or the current at a battery. The type field makes it explicit as to which sort of device the current is associated with. This allows the DGI to create an appropriate type of virtual device to handle storage of the state entry. All type identifiers must correspond to some **<id>** tag of a **<deviceType>** in the *device.xml* configuration file.
4. The **device** name of the device that owns the signal. In our example, there are two FID devices and so there will be two state signals with the value *status* that belong to a device of type *FID*. This field disambiguates which of the two FID devices the state belongs to. In addition, the DGI can access individual devices through use of this device name. For this reason, the name must be unique within the adapter specification file.

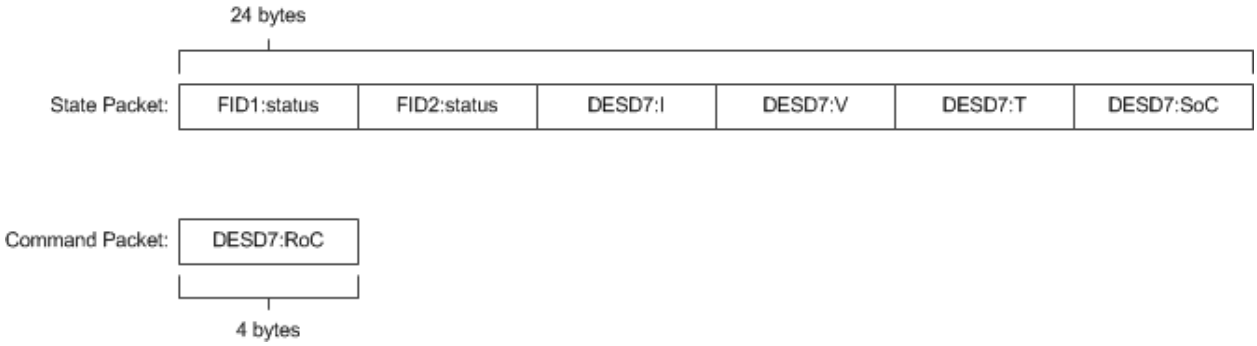
For our example, the state configuration would be:

```
<root>
    <adapter name = "ExampleAdapter" type = "rtds">
        <info>
            <host>FPGA-Hostname</host>
            <port>52000</port>
        </info>
        <state>
            <entry index = 1>               <!-- The index must appear together with the entry tag --
                <type>Fid</type>            <!-- This defines the device type (from device.xml) -->
                <device>FID1</device>       <!-- This is the unique name / identifier -->
                <signal>status</signal>     <!-- This defines the state type (from device.xml) -->
            </entry>
            <entry index = 2>
                <type>Fid</type>
                <device>FID2</device>
                <signal>status</signal>
            </entry>
            <entry index = 3>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>I</signal>
            </entry>
            <entry index = 4>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>V</signal>
            </entry>
            <entry index = 5>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>T</signal>
            </entry>
            <entry index = 6>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>SoC</signal>
            </entry>
        </state>
        <!-- (comment) this specification is still incomplete! -->
    </adapter>
</root>
```

A similar specification must be done for the format of the command packet using the tag **<command>**. All of the required properties of states are also required for commands, and the XML format for both is identical. As such, the commands in our example lead to the final configuration file format:

```
<root>
    <adapter name = "ExampleAdapter" type = "rtds">
        <info>
            <host>FPGA-Hostname</host>
            <port>52000</port>
        </info>
        <state>
            <entry index = 1>
                <type>Fid</type>
                <device>FID1</device>
                <signal>status</signal>
            </entry>
            <entry index = 2>
                <type>Fid</type>
                <device>FID2</device>
                <signal>status</signal>
            </entry>
            <entry index = 3>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>I</signal>
            </entry>
            <entry index = 4>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>V</signal>
            </entry>
            <entry index = 5>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>T</signal>
            </entry>
            <entry index = 6>
                <type>Desd</type>
                <device>DESD7</device>
                <signal>SoC</signal>
            </entry>
        </state>
        <command>
            <entry index = 1>               <!-- The index must appear together with the entry tag --
                <type>Desd</type>           <!-- This defines the device type (from device.xml) -->
                <device>DESD7</device>      <!-- This is the unique name / identifier -->
                <signal>RoC</signal>        <!-- This defines the command type (from device.xml) -->
            </entry>
        </command>
    </adapter>
</root>
```

This completes the RTDS adapter specification for our example. With this specification file, the command packet will be 4-bytes and contain a single command that corresponds to the rate of chargre for DESD7. The state packet will be 24-bytes and contain 6 separate floating point numbers. The following figure shows the exact format of both packets.



Both the **<state>** and **<command>** tags are required, even if there are no states or commands associated with a given adapter. For example, if this adapter did not contain the DESD device and instead contained the two FID devices, the sample configuration file would change to resemble:

```
<root>
    <adapter name = "ExampleAdapter" type = "rtds">
        <info>
            <host>FPGA-Hostname</host>
            <port>52000</port>
        </info>
        <state>
            <entry index = 1>
                <type>Fid</type>
                <device>FID1</device>
                <signal>status</signal>
            </entry>
            <entry index = 2>
                <type>Fid</type>
                <device>FID2</device>
                <signal>status</signal>
            </entry>
        </state>
        <command>
            <!-- The empty command tag must still be included -->
        </command>
    </adapter>
</root>
```

If the contents of the state tag are omitted, the DGI will never attempt to read from the TCP/IP socket it uses to communicate with the device server. Likewise, if the command tag is omitted, the DGI will never write a command packet to the device server. In both of these cases the communication protocol becomes unidirectional. However, in both cases, the **<state>** and **<command>** tags themselves must still be included as in above.

Configuration Errors

1. The name field for each adapter must be unique.
2. Each **<type>** specified during the state and command packet configuration must refer to the **<id>** of a **<deviceType>** found in the *device.xml* configuration file.
3. Each **<signal>** specified during the state or command packet configuration must refer to some **<state>** or **<command>** of its associated **<type>** in the *device.xml* configuration file.
4. When a device of a specific **<type>** is specified, all of its **<state>** and **<command>** values from the *device.xml* configuration file must appear in the adapter configuration file. It is impossible to use a subset of the states or commands of a device when using an RTDS adapter.

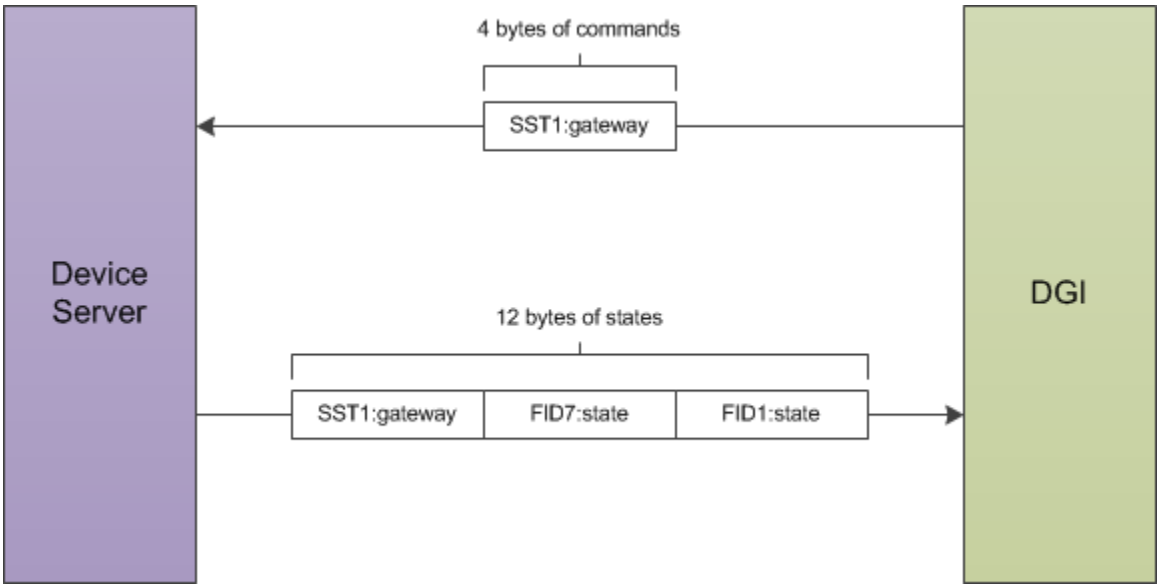
5. The complete state and command specification of each device must be contained within a single **<adapter>**. If a device spans multiple adapters, it will result in tremendously undefined behavior.
6. Indices for state and command entries must be unique, consecutive, and start counting from an initial index of 1.
7. All adapters must have a **<state>** and **<command>** subtag, even if the contents of the tags are empty.

Communication Protocol

The RTDS adapter uses TCP/IP to connect to a server with access to some set of physical devices. When the DGI runs using an RTDS adapter, it attempts to create a client socket connection to an endpoint specified in the adapter configuration file during startup. Once connected, it sends a periodic command packet to the server, and expects to receive a device state packet in return. The device server must be running and prepared to receive connections before the DGI starts when using an RTDS adapter. In addition, the DGI will always send its command packet before the device server responds with its state packet.

This communication protocol is very brittle. If the DGI loses connection to the server, it will not attempt to reconnect and after some time the DGI process will terminate. In addition, if the DGI receives a malformed or unexpected packet from the server, it will terminate with an exception. Therefore, this protocol should only be used on a stable network.

The following diagram shows on round of message exchanges in the communication protocol. It assumes that there are three states produced by the devices, and one command produced by the DGI.



Both the command packet from the DGI and the state packet from the device server contain a stream of 4-byte floating point numbers. Other data formats such as boolean or string values cannot be used with the RTDS adapter; all the device states must be represented as floats. The command packet must contain every command for the devices attached to the server, while the state packet must contain every device state. It is not possible to send a subset of the commands, or to send the values for different commands at different times. If the DGI has not calculated the value of a particular command, it will send a special value of 10^8 to indicate a NULL command. The device server must recognize and ignore the value of 10^8 when parsing the command packet received from the DGI. Likewise, the device server can use the value of 10^8 for device states which are not yet available when communicating with the DGI.

Running the Device Server

The steps discussed so far configure the DGI to connect to a device server utilizing a particular packet format. However, the device server must also be configured to expect the same packet format as the DGI. Because the RTDS adapter can be used for multiple power simulations, the configuration of the device server depends on the type of simulation being run. The details for configuration are thus delegated to the individual tutorials on how to run specific simulations.

If you are simulating with PSCAD, see *Running a PSCAD Simulation*.

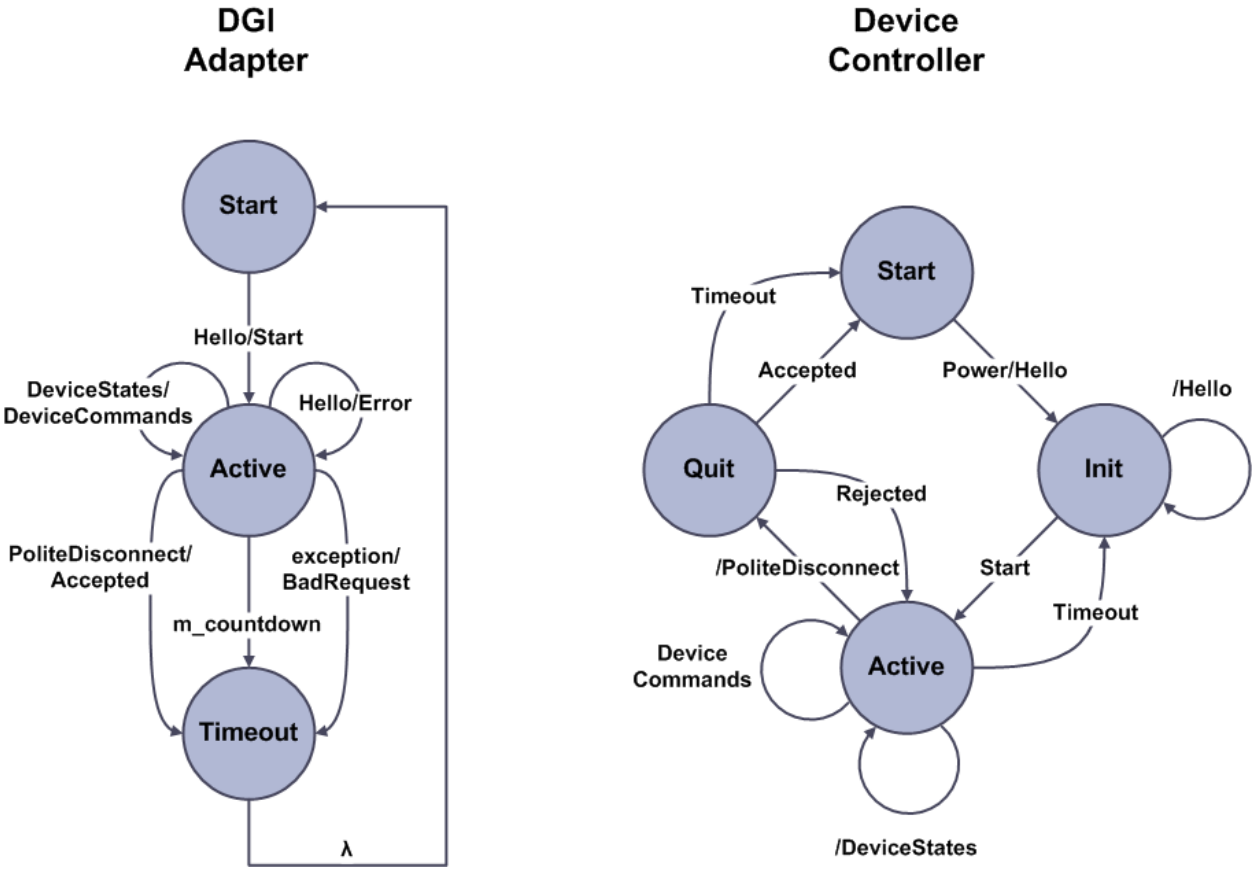
If you are simulating with RTDS, see *Configuring RTDS*.

4.7.2 Plug and Play Adapter

The PNP Adapter allows the DGI to communicate with plug and play devices. Unlike the other adapter types, PNP adapters are created automatically and do not need to be specified in an adapter configuration file. However, by default, the plug and play behavior of the DGI is disabled.

Communication Protocol

The plug and play protocol uses TCP/IP with the DGI listening for client connections from physical devices on a configurable port number. All packets in the protocol are written in ASCII and converted to floating point numbers within the DGI. An overview of both sides of the protocol is shown in the following state machine.



The protocol begins with both the DGI and the device controller in their respective start states. For the DGI, this state corresponds to some time after the DGI has created its TCP/IP socket that listens for connections from devices. For the device, the start state represents when the device is powered off or disconnected. First, the device powers on and

its controller sends the DGI a Hello message. The contents of this message tell the DGI which devices are connected to the controller, and the DGI uses the Hello message to construct a new plug and play adapter. If the new adapter is created without error, the DGI responds with a Start message and maintains the client socket for the duration of the protocol. A separate socket will be maintained for each concurrent plug and play connection to the DGI.

Once the plug and play adapter has been created, the DGI will remain in its active state until the device powers off, loses communication with the DGI, or causes an exception during the protocol. While in this state, the DGI expects the device to send it periodic DeviceStates messages which the it will respond to with a corresponding DeviceCommands message. Unlike the RTDS adapter protocol, the DGI expects the device to send it the states before it issues commands. The plug and play connection is maintained by the DGI as long as it receives periodic DeviceStates messages from the device.

A device can gracefully disconnect from the DGI by sending a PoliteDisconnect message. However, if a device fails to send a DeviceStates message for a configurable timeout period, the DGI will close the socket connection and delete the plug and player adapter under the assumption the device has crashed or gone offline. As shown in the state machine, the DGI does not notify the device when it choses to terminate the connection. If a device comes back online after the connection has been terminated, it must restart the protocol from the Hello message.

Messages from the Device

Hello Message:

```
Hello\r\n
UniqueID\r\n
DeviceType1 DeviceName1\r\n
DeviceType2 DeviceName2\r\n
...
DeviceTypeN DeviceNameN\r\n
\r\n
```

The hello message tells the DGI the number and type of devices that are associated with the device controller. Each device controller must have a unique name stored in non-volatile memory that it reuses each time it connects to the DGI. This name replaces the `UniqueID` placeholder string in the message format, and tells the DGI which controller has initiated the connection. It is imperative that this identifier be the same every time the same controller connects to the DGI to prevent the DGI from creating multiple plug and play adapters for a single device controller. In addition, the `DeviceType#` fields must be replaced with the unique identifiers for devices registered in the *device.xml* file. The unique identifier corresponds to the **<id>** tag as discussed in the virtual device tutorial, *Creating a Virtual Device Type*.

DeviceStates Message:

```
DeviceStates\r\n
DeviceName1 State1 Value\r\n
DeviceName1 State2 Value\r\n
...
DeviceNameN StateM Value\r\n
\r\n
```

The device state message gives the current values for all states of the devices listed in the hello message. If a device from the hello packet has a state, then its device name must appear in this message. In addition, this message cannot contain partial device states. If a device has three states listed in the *device.xml* configuration file, then all three states for that device must be included in this message. When the device states message is received. the DGI will convert each of the value fields into a floating point number. If a device from the hello packet is missing, or at least one state is missing, or at least one value is not numeric, then DGI will reject the message. The device controller can use the special null value of 10^8 if it cannot give the DGI a state to indicate the value should be ignored.

PoliteDisconnect Message:

```
PoliteDisconnect\r\n
\r\n
```

This message indicates that a device is about to turn off and wishes to terminate the connection. The device controller should wait for a response from the DGI before closing its TCP socket.

Error Message:

```
Error\r\n
Message\r\n
\r\n
```

Both the DGI and device controller can send this message, and it indicates that some error has happened during execution of the protocol. This error might not be fatal, and often the DGI sends it to indicate that a received packet did not have the expected format and was dropped.

Messages from the DGI

Start Message:

```
Start\r\n
\r\n
```

The start message indicates that the DGI has created a plug and play adapter for the device controller and the main protocol can begin. Once this message has been sent, the DGI expects to receive periodic DeviceStates messages from the device or it will terminate the TCP connection without warning. The device controller should start sending DeviceStates as soon as the Start message is received.

DeviceCommands Message:

```
DeviceCommands\r\n
DeviceName1 Command1 Value\r\n
DeviceName1 Command2 Value\r\n
...
DeviceNameN CommandM    Value\r\n
\r\n
```

The device command packet is sent by the DGI in response to a state packet. All commands for all devices are included in this packet, even if the DGI does not have a command to issue or the command has not changed since the last packet. If the DGI does not have a command for a device, then the value for that command will be set to the special null value of 10^8 to indicate the value should be ignored. The value for the device name field will be identical to the names provided by the controller in the hello message, and the value for the command fields will be pulled from the *device.xml* configuration file.

PoliteDisconnectAccepted Message:

```
PoliteDisconnect\r\n
Accepted\r\n
\r\n
```

This message acknowledges a polite disconnect request from a device controller and indicates that the DGI will terminate the TCP connection to the device as soon as the message is delivered.

PoliteDisconnectRejected Message:

```
PoliteDisconnect\r\n
Rejected\r\n
\r\n
```

This message tells the device controller that the DGI has received a disconnect request, but it cannot yet terminate the TCP connection. In the current version of the DGI, this message is never sent as all disconnect requests are accepted.

Error Message:

```
Error\r\n
Message\r\n
\r\n
```

Both the DGI and device controller can send this message, and it indicates that some error has happened during execution of the protocol. This error might not be fatal, and often the DGI sends it to indicate that a received packet did not have the expected format and was dropped.

Configuration

The plug and play protocol must be enabled through the main DGI configuration file `Broker/config/freedm.cfg`. If a port number is provided for the TCP server that listens for device connections, then the plug and play protocol will be initialized after running DGI. Otherwise, the plug and play protocol will be disabled. The port number can be set using the command `factory-port=X` anywhere on its own line in the *freedm.cfg* file. Once this port number has been specified, the plug and play protocol has been enabled. All hello messages sent from device controllers should be sent to this port to initiate the plug and play protocol.

Sample Device Controller

A sample implementation of the device controller side of the plug and play protocol is available on the FREEDM-DGI git repository. This sample implementation requires both a configuration file that tells the controller how to communicate with the DGI, as well as a script that controls the behavior of the device controller over time. As the controller is not connected to real physical hardware, its behavior changes only in response to this script.

A sample configuration file can be found at `config/samples/controller.cfg`. The important configurable options in this file are the name, host, and port entries. The name specifies the unique identifier for the device controller that will be included in the hello packet, and must be unique if multiple controllers are used at the same time. The host and port fields must be set to the location of the DGI plug and play server, which will be the hostname of the linux machine that runs the DGI and the port number specified for the factory-port option in the DGI configuration file. The remaining fields do not need to be changed from their default values.

A sample script file can be found at `config/samples/dsp-script.txt`. A script must be provided for the controller in order for it to function after connecting to the DGI. Without a script, the controller will send a PoliteDisconnect packet to the DGI as soon as it receives the Start message from the DGI. Each script contains a sequence of commands follower by a special work command. After all of the commands in the script are processed, the controller will disconnect from the DGI.

Available Script Commands

```
enable DeviceType DeviceName State1 InitialValue1 ... StateN InitialValueN
```

The enable command adds a new device to the controller which will be included in the next Hello message. Each state of the device must be specified and given an initial value, but a value can be set to 10^8 to force the DGI to ignore it. After this command is used in the script, the controller will disconnect from the DGI and restart the protocol with a fresh hello message.

```
disable DeviceName
```

The disable command deletes a device that was added using a prior enable command. This command also causes the controller to disconnect from the DGI and restart the protocol using a fresh hello message.

```
change DeviceName State NewValue
```

The change command updates the value of a device state. DeviceName must have been added using a prior enable command, and the state must refer to one of the states that was initialized when the enable command was used. Like all device states, the value must be a floating point number or it will be rejected by the DGI. This command will change the values sent in the device states message.

```
dieHorribly Duration
```

The dieHorribly command causes the controller to be unresponsive to the DGI for a given amount of time (stop sending state messages). This can be used to simulate network traffic or slow processing speed, but does not simulate connection failure as controller socket is still maintained for the duration of the command.

```
work Duration
```

The work command causes the protocol to continue in its active state for the specified duration. During this time, the controller will send the DGI device state messages and receive command messages. However, the internal state of the device will not change for the duration of the work command.

```
work forever
```

Often the last command in a script, this will cause the controller to stay in the active state until the process is terminated by the user. No commands after this command will be read from the script, and the controller can never change its internal state once this command has been processed.

4.7.3 Creating a New Adapter

If none of the adapter types provided by the DGI are sufficient for communication with a particular device, a new adapter can be implemented in the DGI without much code modification. However, this requires extreme expertise in both C++ and the BOOST libraries and is not recommended for most users. This tutorial will cover the basics on how to create a new adapter class.

Required Functions

All adapters must inherit from the base adapter class `IAdapter` located at `Broker/src/device/IAdapter.hpp`. IAdapter has a required set of functions that all adapter classes must implement. These functions are:

void `IAdapter::Start()`

The start function is called after a new instance of the adapter has been created and the DGI is ready to use the new adapter to communicate with physical devices. This function is guarenteed to be called exactly once by the DGI, and the adapter is expected to do no work until after this function has been called. This function should be implemented to start the protocol that sends and receives device data.

void `IAdapter::Stop()`

The stop function is called when the DGI terminates to allow the adapter a chance to cleanly terminate its connection with its associated physical devices. This function will be called once during the DGI teardown procedure. At the end of the function call, the DGI should be disconnected from the device.

float `IAdapter::GetState`(*const std::string*, *const std::string*) `const`

The GetState function is called each time a virtual device within the DGI attempts to access the current state of its real device communicating with the adapter. Each adapter is required to ensure that the GetState function returns the most recent value of a requested state each time it is called. However, there is no guarentee that this function will ever be called.

void IAdapter::**SetCommand**(*const std::string*, *const std::string*, *const float*)

The SetCommand function is called each time a virtual device within the DGI attempts to send a command to its real device communication with the adapter. Each adapter must ensure that the value sent with this function call eventually reaches the physical device unless a later SetCommand call changes the commanded value. There is no guarentee that this function will ever be called, and as such the initial default values for each command must be set by the adapter itself during construction.

static IAdapter::Pointer YourAdapter::**Create**(...)

The create function is not required in the sense it will not cause a compile error if omitted. However, a create function makes integration of new adapters with the DGI much easier and is thus strongly encouraged. The parameters of the create function will depend on which variables the adapter needs to be initialized, but all create functions should be static. This function will be called when a new instance of the adapter is first created.

Sample Header File

The following is a sample header file that provides a template most adapters should utilize. Both the run function and the deadline timer are optional features that should be utilized if your adapter has to periodically reschedule itself to maintain communication with the physical device. The remaining functions are mostly required as stated above.

```
#include "IAdapter.hpp"

#include <boost/enable_shared_from_this.hpp>
#include <boost/shared_ptr.hpp>
#include <boost/asio/deadline_timer.hpp>
#include <boost/asio/io_service.hpp>

namespace freedm {
namespace broker {
namespace device {

class CYourAdapter
     : public IAdapter, public boost::enable_shared_from_this<CYourAdapter>
{
public:
    /// Typedef for ease of use.
    typedef boost::shared_ptr<CYourAdapter> Pointer;

    /// Called once when adapter is first created (recommended).
    static IAdapter::Pointer Create(boost::asio::io_service & service);

    /// Called once afted the adapter is initialized (required).
    void Start();

    /// Called once before the DGI terminates (required).
    void Stop();

    /// Called each time a DGI module tries to send a command (required).
    void SetCommand(const std::string device, const std::string signal, const SignalValue value);

    /// Called each time a DGI module tries to read a state (required).
    SignalValue GetState(const std::string device, const std::string signal) const;

    /// Destructor.
    ~CYourAdapter();
private:
    /// Constructor.
```

```
    CYourAdapter(boost::asio::io_service & service);

    /// Called periodically to maintain communication with devices (optional).
    void Run(const boost::system::error_code & e);

    /// Used to schedule the Run function (optional).
    boost::asio::deadline_timer m_timer;
};

} //namespace device
} //namespace broker
} //namespace freedm
```

This template assumes that the adapter maintains constant communication with its associated physical devices. The `void CYourAdapter::Run(const boost::system::error_code &)` function will be scheduled at periodic intervals to send and receive data. This data will have to be stored in a member variable inside of the adapter (not in the template), and both the SetCommand and GetState functions will use the member variables instead of directly communicating with the device. This format is identical to the RTDS adapter which can be used as an additional example at `Broker/src/device/CRtdsAdapter.hpp`.

Another possible implementation of adapters would be for the SetCommand and GetState functions to send messages to the device on demand as they are called. In this case, the Run function might be unnecessary as all the communication happens through the required function implementations. In this case, only the required functions would be necessary, and the Run function and its associated `boost::asio::deadline_timer` could be omitted.

Sample Implementation File

The following is a sample implementation file that shows how the various functions interact with each other. It is mostly intended to illustrate how to set up a reoccuring Run function to maintain communication with physical devices. If the Run function is not relevant to your adapter type, this example can likely be ignored.

```
#include "CYourAdapter.hpp"
#include "CLogger.hpp"

namespace freedm {
namespace broker {
namespace device {

IAdapter::Pointer CYourAdapter::Create(boost::asio::io_service & service)
{
  return CYourAdapter::Pointer(new CYourAdapter(service));
}

CYourAdapter::CYourAdapter(boost::asio::io_service & service)
  : m_timer(service)
{
    // initialize your adapter here (change arguments as needed)
}

void CYourAdapter::Start()
{
    // do post-initialization processing here

    // schedule Run 1000 milliseconds from now (change time as needed)
    m_timer.expires_from_now(boost::posix_time::milliseconds(1000));
    m_timer.async_wait(boost::bind(&CYourAdapter::Run, shared_from_this(), boost::asio::placeholders
}
```

```
void CYourAdapter::Stop()
{
    // do pre-termination processing here
}

CYourAdapter::~CYourAdapter()
{
    // deconstruct your adapter here
}

void CYourAdapter::Run(const boost::system::error_code & e)
{
    if(!e)
    {
        // do periodic communication with your devices here
        // this should be the main portion (if not all) of your code

        // reschedule Run 1000 milliseconds from now (change time as needed)
        m_timer.expires_from_now(boost::posix_time::milliseconds(1000));
        m_timer.async_wait(boost::bind(&CYourAdapter::Run, shared_from_this(), boost::asio::placeholo
    }
    else if(e == boost::asio::error::operation_aborted)
    {
        // happens if DGI is terminating ; do nothing special
    }
    else
    {
        // error condition! something in the device framework is broken!
    }
}

void CYourAdapter::SetCommand(const std::string device, const std::string signal, const SignalValue v
{
    // send or prepare to send a command to your devices here
}

SignalValue CYourAdapter::GetState(const std::string device, const std::string signal) const
{
    // read a state from your devices here
}

} //namespace device
} //namespace broker
} //namespace freedm
```

Assuming that the communication code has a sequential block of code that sends a block of data to a device and receives a block of data in return, this sequential code should all be placed into the run function with the data to be sent and receive declared as member variables. Then the SetCommand and GetState functions would write to and read from these member variables rather than interacting with the physical device. This is the approach of the RTDS adapter, which can be used as a sample implementation at `Broker/src/device/CRtdsAdapter`.

Integration with the DGI

Your new adapter class must be integrated with the DGI once its implementation has been completed. This involves three separate steps.

First, the adapter must be included in the DDGI compilation process. We assume that your adapter is located at

Broker/src/device/CYourAdapter.cpp. Open the file `Broker/src/device/CMakeLists.txt` and locate the `set(DEVICE_FILES` command near the very top. Include `CYourAdapter.cpp` after *DEVICE_FILES* and before the closing parenthesis, following the example of the other device files. After this change, when `make` is run from the *Broker* directory, *CYourAdapter.cpp* will be included in the DGI compilation process.

Second, you must decide what configurable options are required for your adapter type. All standard adapters (plug and play being the exception) are created through the adapter configuration file `Broker/config/adapter.xml`. When the DGI starts with an adapter configuration file specified in `Broker/config/freedm.cfg`, it parses the contents of the file to determine which adapters it needs to create. Your adapter will also be configured in *adapter.xml*. Consider the first line of each adapter configuration:

```
<adapter name = "simulation" type = "rtds">
```

You must define a new `type` identifier for your adapter which will go in the type field when a user wants to create a new instance of your adapter type. You cannot remove the **<state>** and **<command>** subtags for your adapter specification, as they are required by the DGI to create virtual devices that modules will use to interact with your adapter. However, you can change the contents of the **<info>** subtag which is intended to contain all the configurable settings unique to your adapter. If you have any user-defined settings that are required when a new instance of your adapter is created, you should determine how best to incorporate them into this **<info>** tag in *adapter.xml*.

Third, you must modify the behemoth of a file that is CAdapterFactory.cpp. This file handles the creation and maintenance of all types of adapters, including the parsing of the adapter configuraton file mentioned above. It can be found at `Broker/src/device/CAdapterFactory.cpp`. The most relevant functions for creation of a new adapter are the `void CAdapterFactory::CreateAdapter(const boost::property_tree::ptree &)` function and the `void CAdapterFactory::InitializeAdapter(IAdapter::Pointer, const boost::property_tree::ptree &)` function. The CreateAdapter function is called each time a new **<adapter>** tag is parsed, and the `boost::property_tree::ptree` stores the contents of the XML specification for the new adapter. The InitializeAdapter function is called once for each adapter, and parses the contents of the **<state>** and **<command>** specifications.

For the DGI to create your adapter, it must create an instance of your adapter's type when it parses the XML in the CreateAdapter function. Locate the following line of code:

```
if( type == "rtds" )
{
    adapter = CRtdsAdapter::Create(m_ios, subtree);
}
else if( type == "pnp" )
{
    adapter = CPnpAdapter::Create(m_ios, subtree, m_server->GetClient());
}
else if( type == "fake" )
{
    adapter = CFakeAdapter::Create();
}
else
{
    throw EDgiConfigError("Unregistered adapter type: " + type);
}
```

This conditional determines which type of adapter the **<adapter>** tag specifies and creates the appropriate adapter class in the DGI. You must extend this conditional to support your new adapter type:

```
else if( type == "YourAdapter" )
{
    adapter = CYourAdapter::Create(m_ios);
}
```

This will call the `static IAdapter::Pointer CYourAdapter::Create(...)` function you defined

in your adapter implementation, and should be passed all the parameters that are required for your implementation's constructor. For the other adapter types, not that several of them are passed the variable *subtree* which is a `boost::property_tree::ptree`. This stores the contents of the **<info>** tag from the *adapter.xml* file. If your adapter uses the **<info>** tag, you should also pass this subtree variable. For using the *subtree* variable, you will have to refer to the BOOST Property Tree Documentation as well as the `Broker/src/device/CRtdsAdapter.cpp` adapter implementation for an example.

With this, your adapter has been compiled into the DGI, constructed when the DGI parses the adapter configuration file, and perhaps initialized with the contents of its **<info>** tag. However, at no point does your adapter learn of the devices that have been attached to it. The **<state>** and **<command>** tags from the adapter configuration file are never seen by your adapter instance. These tags are instead parsed in the function `void CAdapterFactory::InitializeAdapter(IAdapter::Pointer, const boost::property_tree::ptree &)`. If your adapter needs to know the type and number of devices the DGI has associated with it, you must modify this InitializeAdapter function.

First, search for the following line:

```
IBufferAdapter::Pointer buffer = boost::dynamic_pointer_cast<IBufferAdapter>(adapter);
```

You will want to add an additional line after this one with a similar format for your adapter:

```
CYourAdapter::Pointer youradapter = boost::dynamic_pointer_cast<CYourAdapter>(adapter);
```

This will attempt to convert the `IAdapter::Pointer` passed to the function into a `CYourAdapter::Pointer`. If the conversion fails, the variable youradapter will store a null pointer. This conversion is required because any modifications you make to this function are unique to your adapter. They should not be executed if the adapter passed to this function does not have your type, and this dynamic pointer cast provides a very easy way to determine if the adapter has your type in conditionals.

Next search for the line:

```
// create the device when first seen
if( devtype.count(name) == 0 )
{
    CreateDevice(name, type, adapter);
    adapter->RegisterDevice(name);
    devtype[name]  = type;
    states[name]   = 0;
    commands[name] = 0;
}
```

This block of code executes when a device is seen for the first time in the *adapter.xml* configuration file, and creates a virtual device for use in the DGI. The name of this device is already associated with your adapter through the `void IAdapter::RegisterDevice(std::string)` function call, and if you only need to know the number of devices you can overwrite this function in your implementation file. However, by default, the type is not sent to your adapter. Change the code to the following:

```
// create the device when first seen
if( devtype.count(name) == 0 )
{
    CreateDevice(name, type, adapter);
    adapter->RegisterDevice(name);
    devtype[name]  = type;
    states[name]   = 0;
    commands[name] = 0;

    // use the dynamic pointer cast from before
    if( youradapter )
    {
```

```
        // if the code executes this, the adapter is of type CYourAdapter
        youradapter->RegisterType(type);
    }
}
```

With this modification, `void CYourAdapter::RegisterType(std::string)` will be called once for each device associated with your adapter in the *adapter.xml* specification file. However, you will have to implement the RegisterType function as it is not a standard adapter function.

This tutorial only provides a brief overview of creation of a new adapter type, as well as several possibilities for integrating the new type with the DGI. The creation of a new adapter type is complicated and requires extensive knowledge of the device architecture. If you need to create a new adapter type, we strongly recommend you contact the DGI development team and keep in close contact with us. However, it should not be necessary to look at any part of the code other than the two CAdapterFactory functions mentioned in this section.

4.8 Using Devices in DGI Modules

Once a virtual device type has been defined, and a real physical device has been connected to the DGI, modules can use the devices to read the state of the physical system and send commands to the physical hardware. This tutorial will cover how to use the physical device architecture in DGI modules.

A typical module has the following execution:

1. Retrieve a subset of the physical devices
2. Read the state of the retrieved devices
3. Perform some computation using the state
4. Send commands to a devices based on the computation

This execution pattern corresponds to the three main functions a DGI module can perform using devices:

1. Retrieve a virtual device from the device framework
2. Read the state of a virtual device
3. Send a command to a virtual device

4.8.1 Retrieve a Virtual Device

An object called the device manager is a singleton available to all DGI modules. It stores all of the virtual devices in the system, and provides several functions that enable modules to retrieve a subset of the physical devices. In order to retrieve a device, a module must use the interface provided by the device manager. It is important to recognize that the device manager only stores local devices. Each DGI has a subset of the physical devices in the system, and can not access the devices that do not belong to it. Therefore, no DGI can access the entire system state using its own device manager. In order to read values from devices that belong to other DGI processes, refer to the documentation on state collection.

In order to use the device manager, its header file must be included in your module:

```
#include "CDeviceManager.hpp"
```

The device manager can then be retrieved, and stored if necessary, using its static Instance() function:

```
device::CDeviceManager & manager = device::CDeviceManager::Instance();
```

From the device manager instance, a device can be retrieved through using either its unique identifier or its device type. If a module needs to collect a set of devices of the same type, such as the set of generators in the system, it should use the device type. However, if a module only needs a specific device, such as the one SST associated with the DGI, it should use the device's unique identifier.

Retrieve a Device using its Identifier

Suppose a module needs to access a device it knows exists with the unique identifier SST5. The following call will store a pointer to that device:

```
device::CDevice::Pointer dev;
dev = device::CDeviceManager::Instance().GetDevice("SST5");
```

All device pointers must be stored in a `device::CDevice::Pointer`. The `device::CDevice::Pointer device::CDeviceManager::GetDevice(std::string)` function of the device manager can be used to get a pointer to a device with a specific unique identifier, which in this case is SST5. This can be a dangerous function call as there is no guarentee that a device exists with that specific name. If the device manager does not store a device with the given identifier, then it does not throw an exception, but instead returns a null pointer. The pointer can be treated like a boolean truth value to determine whether the call was successful:

```
if(!dev)
    // the device was not found! do some recovery action!
```

Retrieve Devices using their Type

Suppose a module needs to access all the devices associated with the type DRER. The following call will return a set of the matching devices:

```
std::set<device::CDevice::Pointer> devices;
devices = device::CDeviceManager::Instance().GetDevicesOfType("DRER");
```

The `std::set<device::CDevice::Pointer> device::CDeviceManager::GetDevicesOfType(std::string)` function returns all the devices that associate with a specific type. This function will always return a set of CDevice pointers. If no devices of the specified type are stored in the device manager, then an empty set will be returned. The empty function can be used to determine whether the call was successful at returning any devices:

```
if(devices.empty())
    // no devices were found! do some recovery action!
```

BOOST can be utilized to easily iterate over each device in the resulting set. This requires an additional header to be included in the implementation file:

```
#include <boost/foreach.hpp>
```

And the code to iterate over the result would resemble:

```
std::set<device::CDevice::Pointer> devices;
devices = device::CDeviceManager::Instance().GetDevicesOfType("DRER");
BOOST_FOREACH(device::CDevice::Pointer dev, devices)
{
    // dev now stores a pointer to a single DRER device!
}
```

Retrieve a Device with an Unknown Identifier

There are some cases where a module might not know the name of a specific device, but does know that only a single instance of that device should exist. For example, a DGI should only have a single associated SST device, but a module might not make any assumptions on what the unique identifier for that device could be. In this case, the best solution is to use the `std::set<device::CDevice::Pointer> device::CDeviceManager::GetDevicesOfType(std::string)` function with additional error-checking:

```
device::CDevice::Pointer dev;
std::set<device::CDevice::Pointer> devices;
devices = device::CDeviceManager::Instance().GetDevicesOfType("SST");
if(devices.size() != 1)
    // unexpected number of devices (should have been 1)! recover!
dev = *devices.begin();
```

This code retrieves all of the SST devices, of which there should only be one, and then stores the first SST device in the dev pointer. Be careful with this solution as the dereferencing of the devices set could be disastrous if the set is empty.

4.8.2 Read a Device State

Once a device has been retrieved and stored in a `device::CDevice::Pointer` object (assumed at this point to be named dev), the device pointer can be used to read a state. This is done through the `float CDevice::GetState(std::string)` function, which returns a floating point number that corresponds to the current value of the state known to the DGI:

```
float voltage = dev->GetState("voltage");
```

In this example, if the device did not have a voltage state, the function call would throw an exception. A catch block is required to prevent this exception from causing the DGI to terminate:

```
try
{
    float voltage = dev->GetState("voltage");
}
catch(std::exception & e)
{
    // device does not have a voltage state! recover!
}
```

The list of states that are recognized be each device can be found in the *device.xml* configuration file. For each device type, the string identifiers that will not cause exceptions with the GetState call are those specified with the **<state>** tag. To be safe, all uses of the GetState function should be done inside of a try block with a corresponding catch statement.

4.8.3 Set a Device Command

A command can be issued to a device pointer using the `void CDevice::SetCommand(std::string, float)` function. If the specified command cannot be found, then this function call will throw an exception. The correct usage of this command should resemble:

```
try
{
    dev->SetCommand("rateOfCharge", -0.25);
}
catch(std::exception & e)
```

```
{
    // device does not have a rateOfCharge command! recover!
}
```

4.8.4 Example Usage

The following example code will show how the device framework will be integrated into most modules. In this example, the net generation at a DGI instance is calculated and used to set the charge rate of a battery. As this is an example, the actual calculations involved in the code are nonsensical.

```
#include "CDeviceManager.hpp"
#include <boost/foreach.hpp>
#include <iostream>
#include <set>

void YourModule::PerformCalculation()
{
    std::set<device::CDevice::Pointer> drerSet;
    device::CDevice::Pointer desd;
    float netGeneration;
    float rateOfCharge;

    // retrieve the set of DRER devices
    drerSet = device::CDeviceManager::Instance().GetDevicesOfType("DRER");
    if(drerSet.empty())
    {
        std::cout << "Error! No generators!" << std::endl;
        return;
    }

    // calculate the net DRER generation
    netGeneration = 0;
    try
    {
        BOOST_FOREACH(device::CDevice::Pointer drer, drerSet)
        {
            netGeneration += drer->GetState("generation");
        }
    }
    catch(std::exception & e)
    {
        std::cout << "Error! Generators did not recognize OUTPUT state!" << std::endl;
        return;
    }

    // determine the appropriate battery charge rate (nonsensical)
    rateOfCharge = 0;
    if(netGeneration > 0)
        rateOfCharge = netGeneration;

    // retrieve the DESD device
    desd = device::CDeviceManager::Instance().GetDevice("MyDesd");
    if(!desd)
    {
        std::cout << "Error! MyDesd device not found!" << std::endl;
        return;
    }
```

```
    // set the DESD command
    try
    {
        desd->SetCommand("charge", rateOfCharge);
    }
    catch(std::exception & e)
    {
        std::cout << "Error! Could not set battery CHARGE command!" << std::endl;
    }
}
```

These functions should be sufficient for all modules that need to use physical devices. However, additional functions are provided by the device manager. A list of these functions can be obtained from the device manager header file in the DGI code located at `Broker/src/device/CDeviceManager.hpp`.

CHAPTER 5

Creating Modules

5.1 Starting Your Module

5.1.1 IDGIModule Reference

DGI modules are always based on the abstract interface IDGIModule.

IDGIModule is found in IDGIModule.hpp

class `freedm::broker::IDGIModule`
An interface for an object which can handle recieving incoming messages.

Public Functions

IDGIModule()
Constructor, initializes the reference to self.

IDGIModule

Description:

Constructor for an *IDGIModule*. Gets the uuid from CGlobalConfiguration and makes a CPeerNode referencing it.

Precondition:

CGlobalConfiguration is loaded.

Postcondition:

m_me is created.

virtual **~IDGIModule**()
Virtual destructor for inhertiance.

virtual void **HandleIncomingMessage**(boost::shared_ptr< const ModuleMessage > *msg*, CPeerNode *peer*) = 0
Handles received messages.

Protected Functions

CPeerNode **GetMe**()
Gets a CPeerNode representing this process.

GetMe

Description:

Gets a CPeerNode that refers to this process.

Return

A CPeerNode referring to this process.

std::string **GetUUID**() const
Gets the UUID of this process.

GetUUID

Description:

Gets this process's UUID.

Return

This process's UUID

Additionally, you will want to create a Run() method that will kick-off the actions your module peforms.

5.1.2 Module Creation

To create your module, first create a new directory for it in the *Broker/src* directory. You should select a short name for your module (typically 3 characters or less) use that as the name of your folder. For example, if you are creating a Volt-Var control module you might start your module like so:

```
$ cd Broker/src
$ mkdir vv
$ cd vv
$ touch VoltVar.cpp
$ touch VoltVar.hpp
```

This will create a folder for your module and start you out with two blank C++ files where you will create your module.

5.1.3 Module .hpp

The .hpp should contain your Module's class declaration. Modules inherit from IDGIModule. In our example where we are creating *VoltVar.hpp*, this will get us started:

```
#ifndef VOLTVAR_HPP_
#define VOLTVAR_HPP_

#include "IDGIModule.hpp"
#include "CPeerNode.hpp"
#include "PeerSets.hpp"

#include "messages/ModuleMessage.pb.h"

namespace freedm {
namespace broker {
namespace vv {

/// Declaration of Garcia-Molina Invitation Leader Election algorithm.
class VVAgent
  : public IDGIModule
```

```
{
  public:
    /// Constructor for using this object as a module.
    VVAgent();
    /// Module destructor
    ~VVAgent();
    /// Called to start the system
    int     Run();
  private:
    /// Handles received messages
    void HandleIncomingMessage(boost::shared_ptr<const ModuleMessage> msg, CPeerNode peer);
};

} // namespace vv
} // namespace broker
} // namespace freedm
```

We've created a **constructor**, **destructor**, **Run()** method, and a method for handling messages (**HandleIncomingMessages()**). Let's implement these methods in *VoltVar.cpp*.

5.1.4 Module .cpp

Here are the implementations for the methods we've defined so far:

```
#include "VoltVar.hpp"
#include "CBroker.hpp"
#include "CLogger.hpp"

namespace freedm {
namespace broker {
namespace vv {

namespace {
    /// This file's logger.
    CLocalLogger Logger(__FILE__);
}

VVAgent::VVAgent()
{
}

VVAgent::~VVAgent()
{
}

int VVAgent::Run()
{
    Logger.Warn<<"Volt Var Control Sure Is Neat!"<<std::endl;
}
void HandleIncomingMessage(boost::shared_ptr<const ModuleMessage> msg, CPeerNode peer)
{
    Logger.Warn<<"Dropped message of unexpected type:\n" << msg->DebugString();
}

} // namespace vv
} // namespace broker
} // namespace freedm
```

What's going on here? We've created an instance of **CLocalLogger** called `Logger`. This allows us to log messages from this module. When creating your module you may find it handy to familiarize yourself with *Using The DGI Logger*.

Next, we need to register our module with the scheduler and message delivery system. In *Broker/src/PosixMain.cpp* locate the initialize modules section and add your new module:

```
// Initialize modules
boost::shared_ptr<IDGIModule> GM = boost::make_shared<gm::GMAgent>();
boost::shared_ptr<IDGIModule> SC = boost::make_shared<sc::SCAgent>();
boost::shared_ptr<IDGIModule> LB = boost::make_shared<lb::LBAgent>();

// My new module!!
boost::shared_ptr<IDGIModule> VV = boost::make_shared<lb::VVAgent>();
```

Just below that you'll register your module with the dispatcher, which is responsible for delivering messages to your module:

```
// Instantiate and register the group management module
CBroker::Instance().RegisterModule("gm",
    boost::posix_time::milliseconds(CTimings.Get("GM_PHASE_TIME")));
CDispatcher::Instance().RegisterReadHandler(GM, "gm");
// Instantiate and register the state collection module
CBroker::Instance().RegisterModule("sc",
    boost::posix_time::milliseconds(CTimings.Get("SC_PHASE_TIME")));
CDispatcher::Instance().RegisterReadHandler(SC, "sc");
// StateCollection wants to receive Accept messages addressed to lb.
CDispatcher::Instance().RegisterReadHandler(SC, "lb");
// Instantiate and register the power management module
CBroker::Instance().RegisterModule("lb",
    boost::posix_time::milliseconds(CTimings.Get("LB_PHASE_TIME")));
CDispatcher::Instance().RegisterReadHandler(LB, "lb");

// REGISTER YOUR NEW MODULE
CBroker::Instance().RegisterModule("vv", boost::posix_time::milliseconds(2000));
CDispatcher::Instance().RegisterReadHandler(VV, "vv");
```

What did we do here? I've registered our module with the **Broker**, which will allocate it 2000 milliseconds of execution time in the real time scheduler. Later, when we start working with the schedule in our module, we'll cover adding entries to the timing configuration file, so that users can adjust the timing of your module for their system. Next, we will need to invoke a call to our **Run()** method to get our module going:

```
Logger.Debug << "Starting thread of Modules" << std::endl;
CBroker::Instance().Schedule(
    "gm",
    boost::bind(&gm::GMAgent::Run, boost::dynamic_pointer_cast<gm::GMAgent>(GM)),
    false);
CBroker::Instance().Schedule(
    "lb",
    boost::bind(&lb::LBAgent::Run, boost::dynamic_pointer_cast<lb::LBAgent>(LB)),
    false);

// New Module!
CBroker::Instance().Schedule(
    "vv",
    boost::bind(&lb::VVAgent::Run, boost::dynamic_pointer_cast<lb::VVAgent>(VV))
    false);
```

When the broker starts, the Volt Var module's **Run()** method will be called. However, before we run DGI with our

new module, we need to add our new module to the CMake configuration. Edit *Broker/src/CMakeLists.txt* and add your new module:

```
...
CClockSynchronizer.cpp
CTimings.cpp
CPhysicalTopology.cpp
gm/GroupManagement.cpp
lb/LoadBalance.cpp
sc/StateCollection.cpp
vv/VoltVar.cpp
)
```

Then to build, you will invoke `cmake` and then `make`:

```
$ pwd
/home/scj7t4/FREEDM/Broker
$ cmake
$ make
```

If everything goes well, you can run *PosixBroker*. With careful observation you should be able to catch the message we log in the **Run()** method of our module:

```
2015-Feb-17 13:10:50.014181 : VoltVar.cpp : Warn(3):
    Volt Var Control Sure Is Neat!
```

Next, let's make our module do something go to *Scheduling DGI Modules*

5.2 Scheduling DGI Modules

The CBroker (the broker) is in CBroker.hpp

DGI modules are scheduled through the Broker. The broker provides a real-time round robin scheduler that manages the execution time of modules. Modules can request timers, special objects that request a task be performed at some point in the future. The Broker allows each module a certain amount of execution time each round. A modules execution time is usually refered to as a *phase*. Each module has it's own phase. This line of code from *PosixMain.cpp* registers the length of a module's phase with the Broker:

```
CBroker::Instance().RegisterModule("vv", boost::posix_time::milliseconds(2000));
```

In this case, our Volt Var module is allotted 2000 milliseconds of execution time each round. We also invoked a call to **Run()** in *PosixMain.cpp* that is the entry point for our module's execution. In order for anything else to happen though, we need to learn how to schedule tasks for our module to perform.

There are two options for scheduling tasks. Tasks can be scheduled for immediate execution, they will be executed as soon as the active phase is your module. If they are scheduled while the module is active, they will be immediately executed. Tasks can be scheduled fo the future by using a timer. They will execute when timer expires and the active phase is your module.

5.2.1 Scheduling For Immediate Execution

boost::bind is a part of <boost/bind.hpp>

The first approach to scheduling tasks is to schedule them for immediate execution. These tasks will be run immediately while the active phase is your module.

Tasks are scheduled as functors. The functor that is scheduled to execute must take no arguements and return void. To do this, you will use **boost::bind** to create functors for the member functions of your module:

```
// New Method
void VVAgent::MyScheduledMethod()
{
    Logger.Error<<"Schedule!"<<std::endl;
    CBroker::Instance().Schedule("vv",
        boost::bind(&VVAgent::MyScheduledMethod, this));
}

// Modified Existing method
void VVAgent::Run()
{
    CBroker::Instance().Schedule("vv",
        boost::bind(&VVAgent::MyScheduledMethod, this));
}
```

Now, if we run the DGI, when the Volt Var module is active, it will repeatedly print "Schedule!" to the screen. One thing to consider when programming for the DGI, is that modules are responsible for ensuring they do not exceede their alotted execution time. The scheduler does not actively enforce any timing requirements – it is your job as the programmer to ensure your module does not exceed its alotted execution time. Scheduling for immediate execution has the potential to do so if the time to complete the scheduled task exceeds the amount of time remainin in the module. Exceeding the phase time for your module can cause faults in other modules, resulting in unpredictable behavior.

5.2.2 Scheduling Tasks To Run In The Future

boost::posix_time::ptime is a part of <boost/date_time/posix_time/posix_time.hpp> and <boost/date_time/posix_time/posix_time_types.hpp>

boost::asio::error::operation_aborted and boost::system::error_code are a part of <boost/asio.hpp>

Our second approach to scheduling tasks is to use timers to schedule them to be executed in the future. In order to use timers, you must first ask the **Broker** to allocate a timer for you. A timer can be used multiple times, but can only be used for one pending task at a time. A good place to ask the Broker to allocate a timer is in your constructor:

```
VVAgent::VVAgent()
{
    m_timer = CBroker::Instance().AllocateTimer("vv");
}
```

Where `m_timer` is a member variable of **VVAgent** of type **CBroker::TimerHandle**. The arguement to the AllocateTimer method is the same short name that you used to register your module with the Broker.

Important! Make sure the short name you use to allocate the timer matches the one used to register the module, or the task will never be executed.

You should schedule anything in the constructor, instead you'll schedule your first tasks in the **Run()** method. Here is a modification of the first example that runs our task every 300 milliseconds:

```
void VVAgent::MyScheduledMethod(const boost::system::error_code& err)
{
    if(!err)
    {
        Logger.Error<<"Schedule!"<<std::endl;
        CBroker::Instance().Schedule(m_timer, boost::posix_time::milliseconds(300),
            boost::bind(&VVAgent::MyScheduledMethod, this, boost::asio::placeholders::error));
    }
    else
    {
        /* An error occurred or timer was canceled */
```

```
        Logger.Error << err << std::endl;
    }

}

void VVAgent::Run()
{
    CBroker::Instance().Schedule("vv",
        boost::bind(&VVAgent::MyScheduledMethod, this, boost::system::error_code()));
}
```

Methods that are scheduled with a timer must expect one argument of type **boost::system::error_code**. This argument will be populated with the reason why the timer expired. If the value is zero, the timer expired because the specified amount of time had passed. If the value is non-zero, the timer expired either because it was cancelled or some other system error has occured. A timer can be cancelled if the timer is used to schedule another task before it expires, or if the DGI is stopping. If a timer is cancelled, err will have the value `boost::asio::error::operation_aborted` It is a good practice to check err and only schedule new tasks if has expired for a reason you expect. In this example we only schedule our task again if the timer was not cancelled.

Inside **VVAgent::MyScheduledMethod** we call the **Broker::Schedule** function to schedule our method to run again. The first parameter `m_timer` is the timer we allocated in the constructor. The second parameter is the amount of time, as a **boost::posix_time::time_duration** that should pass before the task is executed. The third parameter is a functor that will be executed when the timer expires. This functor expects one argument, denoted by **boost::asio::placeholders::error** that will be filled with a **boost::system::error_code** when the timer expires.

The **Run()** method demonstrates how a method that is normally called by a timer can be scheduled to run immediately. In the example, the call to **boost::bind** that creates the functor, now takes another argument **boost::system::error_code** that binds a zero error code to the eventual method call.

If a timer expires while the module is not active (that is, it is another module's phase), the execution of the method will be delayed until that module is active.

Scheduling Tasks To Run Next Time

boost::posix_time::not_a_date_time is a part of <boost/date_time/posix_time/posix_time.hpp>

Tasks can also be scheduled to run the next time a module is active. Tasks scheduled this way still need a timer, but the timer will expire as soon the module is no longer active. Here is a modification of our previous example that will execute **MyScheduledMethod** once each round:

```
void VVAgent::MyScheduledMethod(const boost::system::error_code& err)
{
    if(!err)
    {
        Logger.Error<<"Schedule!"<<std::endl;
        CBroker::Instance().Schedule(m_timer, boost::posix_time::not_a_date_time,
            boost::bind(&VVAgent::MyScheduledMethod, this, boost::asio::placeholders::error));
    }
    else
    {
        /* An error occurred or timer was canceled */
        Logger.Error << err << std::endl;
    }

}

void VVAgent::Run()
```

```
{
    CBroker::Instance().Schedule("vv",
        boost::bind(&VVAgent::MyScheduledMethod, this, boost::system::error_code()));
}
```

From the previous example, we have replaced the 300 millisecond **boost::posix_time::time_duration** with a `boost::posix_time::not_a_date_time`. Now when the **VVAgent**'s phase ends, `m_timer` will expire, and when it is the module's phase again, **MyScheduledMethod** will be executed.

From Here, you can read more about the scheduler: *CBroker Reference*

Or you can go on to message passing: *Receiving Messages*

5.3 Receiving Messages

GMAgent::ProcessPeerList is available by including gm/GroupManagement.hpp

Messages arrive at your module as "ModuleMessages." When a message is received, your module's HandleIncomingMessage method is invoked. This method, which you must implement, should take the ModuleMessage, extract the contents you are interested in, and operate on those contents. We recommend writing a Handler for each type of message you expect to receive. Since our VoltVar example doesn't do anything yet, let's just set up the module to process GroupManagement's PeerList message. This message is sent by the GroupManagement module occasionally to announce when other DGIs are started or stopped. Let's expand our HandleIncomingMessage function to handle the PeerList message:

```
void HandleIncomingMessage(boost::shared_ptr<const ModuleMessage> msg, CPeerNode peer)
{
    if(m->has_group_management_message())
    {
        gm::GroupManagementMessage gmm = m->group_management_message();
        if(gmm.has_peer_list_message())
        {
            HandlePeerList(gmm.peer_list_message(), peer);
        }
        else
        {
            Logger.Warn << "Dropped unexpected group management message:\n" << m->DebugString();
        }
    }
    else
    {
        Logger.Warn<<"Dropped message of unexpected type:\n" << msg->DebugString();
    }
}
```

When we receive a message, we check to see if it contains a group management message. If it does, we see if that message is a peer list. If it is, then we invoke a HandlePeerList method (that we are about to write) to handle that message:

```
//Class definition
class VVAgent
{
    ...
    private:
    PeerSet m_peers;
    std::string m_leader;
};
```

```
// HandlePeerList Implementation
void VVAgent::HandlePeerList(const gm::PeerListMessage & m, CPeerNode peer)
{
    Logger.Trace << __PRETTY_FUNCTION__ << std::endl;
    Logger.Notice << "Updated peer list received from: " << peer.GetUUID() << std::endl;

    m_peers.clear();

    PeerSet m_peers = gm::GMAgent::ProcessPeerList(m);
    m_leader = peer.GetUUID();
}
```

When the peerlist message is received, the HandlePeerList message will be invoked. This method will update m_peers with the new list of running DGI. It will also set m_leader, which contains the UUID of the process that is currently managing the active process list. If needed, you can initialize m_leader in the constructor to GetUUID() as that is a safe startup value.

At this point you can run your module and see that it is receiving peer lists. It will only receive a peer list when the list of active DGIs change.

Now that you've seen how receiving message works, you can move on creating and handling your own messages: *Message Passing*

5.4 Message Passing

The message passing allows messages to be exchanged between DGI. Messages are creating using Google's Protocol Buffers and are sent with a few simple commands inside your DGI module. Messages are defined in .proto files which are kept in the *Broker/src/messages* directory. Each module has it's own .proto file.

5.4.1 Creating Your Protocol File

First create a new .proto file in the *Broker/src/messages* folder. For our VoltVar example, I'll be making *VoltVar.proto*. The first thing to do in our proto file is to define what package the messages belong to:

```
package freedm.broker.vv;
```

Once again, we are using "vv" as the short name for our module. It's best to try and keep your short name consistent across the system. Next, we can start creating messages. Here's a couple:

```
message VoltageDeltaMessage {
    required uint32 control_dactor = 1;
    required float phase_measurement = 2;
    optional string reading_location = 3;
}

message LineReadingsMessage
{
    repeated float measurement = 1;
    required string capture_time = 2;
}
```

We've created two messages, VoltageDeltaMessage and LineReadingsMessage and listed what these messages will carry. The first message, VoltageDeltaMessage carries and integer "controlFactor" and float "phaseMeasurement" which are required parameters. The VoltageDeltaMessage can't be sent without those parameters. It also has an optional third parameter "readingLocation" that contains a string, and is not needed to send the message. The second

message, LineReadingsMessage, contains any number of measurement's (including zero!) as well as the time the measurements were captured.

Note the = <somenumber> item we have on each line. This is to help the protocol buffers library pack each message. When creating simple message for the DGI, the best practice is to simply use increasing numbers in each message.

You will also need to create a message type for your module that can hold any of the other messages you create. We will refer to this as a module container message. This will help the message delivery system get your message to your module, and make it easier to prepare your messages for sending:

```
message VoltVarMessage
{
    optional VoltageDeltaMessage voltage_delta_message = 1;
    optional LineReadingsMessage line_readings_message = 2;
}
```

Each message type I created previously is now an optional value in my *VoltVarMessage*. My complete *VoltVar.proto* is below:

```
message VoltageDeltaMessage {
    required uint32 control_factor = 1;
    required float phase_measurement = 2;
    optional string reading_location = 3;
}

message LineReadingsMessage
{
    repeated float measurement = 1;
    required string capture_time = 2;
}

message VoltVarMessage
{
    optional VoltageDeltaMessage voltage_delta_message = 1;
    optional LineReadingsMessage line_readings_message = 2;
}
```

The last thing to do is to register your VoltVar messages with the DGI. Doing so is simple: open ModuleMessage.proto and append your module container message (VoltVarMessage) to the existing list. You will need to use your short name prefix to access your message:

```
message ModuleMessage
{
    required string recipient_module = 1;
    optional gm.GroupManagementMessage group_management_message = 2;
    optional sc.StateCollectionMessage state_collection_message = 3;
    optional lb.LoadBalancingMessage load_balancing_message = 4;
    optional ClockSynchronizerMessage clock_synchronizer_message = 5;

    /// My New Message!
    optional vv.VoltVarMessage volt_var_message = 6;
}
```

Make sure the number you select is not repeated anywhere else.

Protocol Buffers are a powerful library. This covered basic creation of protocol buffers messages, which should be sufficient to create any module, however, additional documentation for protocol buffers can be found in their official manual. LIIIIIIIIINK

5.4.2 Preparing Messages

In order to send a message, you must first create your module container message (VoltVarMessage), then add the values to the specific message you are sending, and lastly pack the module container message in a ModuleMessage. The DGI team recommends creating methods for your module for each message type you wish to send as well as a method that packs the module container message into the ModuleMessage. Let's make some methods for the messages we created previously:

```
ModuleMessage VVAgent::VoltageDelta(unsigned int cf, float pm, std::string loc) {
    VoltVarMessage vvm;
    VoltageDeltaMessage *vdm = vvm.mutable_voltage_delta_message();
    vdm->set_control_factor(cf);
    vdm->set_phase_measurement(pm);
    vdm->set_reading_location(loc);
    return PrepareForSending(vvm,"vv");
}

ModuleMessage VVAgent::LineReadings(std::vector<float> vals)
{
    VoltVarMessage vvm;
    std::vector<float>::iterator it;
    LineReadingsMessage *lrm = vvm.mutable_line_readings_message();
    for(it = vals.begin(); it != vals.end(); it++)
    {
        lrm->add_measurement(*it);
    }
    lrm->set_capture_time = boost::posix_time::to_simple_string(boost::posix_time::microsec_clock::ur
    return PrepareForSending(vvm,"vv");
}
```

These methods prepare our two messages and return them as a ModuleMessage. The messages are first created as our module container message, VoltVarMessage. We access the specific child message we want to populate and fill in it's contents. Send functions expect to receive ModuleMessages, so we have to do a little legwork to convert our VoltVarMessages to a ModuleMessage. PrepareForSending is a method you'll need to add to your module. Fortunately, it's pretty simple to make:

```
ModuleMessage VVAgent::PrepareForSending(
    const VoltVarMessage& message, std::string recipient)
{
    Logger.Trace << __PRETTY_FUNCTION__ << std::endl;
    ModuleMessage mm;
    mm.mutable_volt_var_message()->CopyFrom(message);
    mm.set_recipient_module(recipient);
    return mm;
}
```

To make this for your own module, you'll need to change the type of the message parameter, and change the "mutable_volt_var_message" to be the correct message type. PrepareForSending expects two arguments, the original message you wish to convert and a recipient, which is the shortname of the module you wish to receive the message.

5.4.3 Sending Messages

Messages can be sent to other DGI using the Send method of a CPeerNode. This method expects one parameter of type ModuleMessage and sends that method the peer the object represents. The best way to get some CPeerNode's is to use the peers from the PeerList message from group managemnt. Assuming you have access to *m_peers* from that example, you can sent a message to each peer in the PeerSet like so:

```
void VVAgent::MyScheduledMethod(const boost::system::error_code& err)
{
    if(!err)
    {
        BOOST_FOREACH(CPeerNode peer, m_peers | boost::adaptors::map_values)
        {
            ModuleMessage mm = VoltageDelta(2, 3.0, "S&T");
            peer->Send(mm);
        }
        Logger.Error<<"Schedule!"<<std::endl;
        CBroker::Instance().Schedule(m_timer, boost::posix_time::not_a_date_time,
            boost::bind(&VVAgent::MyScheduledMethod, this, boost::asio::placeholders::error));
    }
    else
    {
        /* An error occurred or timer was canceled */
        Logger.Error << err << std::endl;
    }

}
```

And that's it! PeerSet's have several methods that can help you select individual peers from the set, but in general, sending messages to the entire list, or to the leader is sufficient.

5.4.4 Processing Messages

To handle the messages you create, you'll need to add them to your module's HandleIncomingMessage method as well as create handlers for each type of message you'll receive. Let's do that:

```
void HandleIncomingMessage(boost::shared_ptr<const ModuleMessage> msg, CPeerNode peer)
{
    if(m->has_volt_var_message())
    {
        VoltVarMessage vvm = m->volt_var_message();
        if(vvm.has_voltage_delta_message())
        {
            HandleVoltageDelta(m, peer);
        }
        else if(vvm.has_line_readings_message())
        {
            HandleLineReadings(m, peer);
        }
        else
        {
            Logger.Warn << "Dropped unexpected volt var message: \n" << m->DebugString();
        }
    }
    else if(m->has_group_management_message())
    {
        gm::GroupManagementMessage gmm = m->group_management_message();
        if(gmm.has_peer_list_message())
        {
            HandlePeerList(gmm.peer_list_message(), peer);
        }
        else
        {
            Logger.Warn << "Dropped unexpected group management message:\n" << m->DebugString();
```

```
        }
    }
    else
    {
        Logger.Warn<<"Dropped message of unexpected type:\n" << msg->DebugString();
    }
}

void VVAgent::HandleVoltageDelta(const gm::PeerListMessage & m, CPeerNode peer)
{
    Logger.Trace << __PRETTY_FUNCTION__ << std::endl;
    Logger.Notice << "Got VoltageDelta from: " << peer.GetUUID() << std::endl;
    Logger.Notice << "CF "<<m.control_factor()<<" Phase "<<m.phase_measurement()<<std::endl;
}

void VVAgent::HandleLineReadings(const gm::PeerListMessage & m, CPeerNode peer)
{
    Logger.Trace << __PRETTY_FUNCTION__ << std::endl;
    Logger.Notice << "Got Line Readings from "<< peer.GetUUID() << std::endl;
}
```

That's it!

5.5 Using Devices in DGI Modules

Once a virtual device type has been defined, and a real physical device has been connected to the DGI, modules can use the devices to read the state of the physical system and send commands to the physical hardware. This tutorial will cover how to use the physical device architecture in DGI modules.

A typical module has the following execution:

1. Retrieve a subset of the physical devices
2. Read the state of the retrieved devices
3. Perform some computation using the state
4. Send commands to a devices based on the computation

This execution pattern corresponds to the three main functions a DGI module can perform using devices:

1. Retrieve a virtual device from the device framework
2. Read the state of a virtual device
3. Send a command to a virtual device

5.5.1 Retrieve a Virtual Device

An object called the device manager is a singleton available to all DGI modules. It stores all of the virtual devices in the system, and provides several functions that enable modules to retrieve a subset of the physical devices. In order to retrieve a device, a module must use the interface provided by the device manager. It is important to recognize that the device manager only stores local devices. Each DGI has a subset of the physical devices in the system, and can not access the devices that do not belong to it. Therefore, no DGI can access the entire system state using its own device manager. In order to read values from devices that belong to other DGI processes, refer to the documentation on state collection.

In order to use the device manager, its header file must be included in your module:

```
#include "CDeviceManager.hpp"
```

The device manager can then be retrieved, and stored if necessary, using its static Instance() function:

```
device::CDeviceManager & manager = device::CDeviceManager::Instance();
```

From the device manager instance, a device can be retrieved through using either its unique identifier or its device type. If a module needs to collect a set of devices of the same type, such as the set of generators in the system, it should use the device type. However, if a module only needs a specific device, such as the one SST associated with the DGI, it should use the device's unique identifier.

Retrieve a Device using its Identifier

Suppose a module needs to access a device it knows exists with the unique identifier SST5. The following call will store a pointer to that device:

```
device::CDevice::Pointer dev;
dev = device::CDeviceManager::Instance().GetDevice("SST5");
```

All device pointers must be stored in a `device::CDevice::Pointer`. The `device::CDevice::Pointer device::CDeviceManager::GetDevice(std::string)` function of the device manager can be used to get a pointer to a device with a specific unique identifier, which in this case is SST5. This can be a dangerous function call as there is no guarentee that a device exists with that specific name. If the device manager does not store a device with the given identifier, then it does not throw an exception, but instead returns a null pointer. The pointer can be treated like a boolean truth value to determine whether the call was successful:

```
if(!dev)
    // the device was not found! do some recovery action!
```

Retrieve Devices using their Type

Suppose a module needs to access all the devices associated with the type DRER. The following call will return a set of the matching devices:

```
std::set<device::CDevice::Pointer> devices;
devices = device::CDeviceManager::Instance().GetDevicesOfType("DRER");
```

The `std::set<device::CDevice::Pointer> device::CDeviceManager::GetDevicesOfType(std::string)` function returns all the devices that associate with a specific type. This function will always return a set of CDevice pointers. If no devices of the specified type are stored in the device manager, then an empty set will be returned. The empty function can be used to determine whether the call was successful at returning any devices:

```
if(devices.empty())
    // no devices were found! do some recovery action!
```

BOOST can be utilized to easily iterate over each device in the resulting set. This requires an additional header to be included in the implementation file:

```
#include <boost/foreach.hpp>
```

And the code to iterate over the result would resemble:

```
std::set<device::CDevice::Pointer> devices;
devices = device::CDeviceManager::Instance().GetDevicesOfType("DRER");
BOOST_FOREACH(device::CDevice::Pointer dev, devices)
{
```

```
    // dev now stores a pointer to a single DRER device!
}
```

Retrieve a Device with an Unknown Identifier

There are some cases where a module might not know the name of a specific device, but does know that only a single instance of that device should exist. For example, a DGI should only have a single associated SST device, but a module might not make any assumptions on what the unique identifier for that device could be. In this case, the best solution is to use the `std::set<device::CDevice::Pointer> device::CDeviceManager::GetDevicesOfType(std::string)` function with additional error-checking:

```
device::CDevice::Pointer dev;
std::set<device::CDevice::Pointer> devices;
devices = device::CDeviceManager::Instance().GetDevicesOfType("SST");
if(devices.size() != 1)
    // unexpected number of devices (should have been 1)! recover!
dev = *devices.begin();
```

This code retrieves all of the SST devices, of which there should only be one, and then stores the first SST device in the dev pointer. Be careful with this solution as the dereferencing of the devices set could be disastrous if the set is empty.

5.5.2 Read a Device State

Once a device has been retrieved and stored in a `device::CDevice::Pointer` object (assumed at this point to be named dev), the device pointer can be used to read a state. This is done through the `float CDevice::GetState(std::string)` function, which returns a floating point number that corresponds to the current value of the state known to the DGI:

```
float voltage = dev->GetState("voltage");
```

In this example, if the device did not have a voltage state, the function call would throw an exception. A catch block is required to prevent this exception from causing the DGI to terminate:

```
try
{
    float voltage = dev->GetState("voltage");
}
catch(std::exception & e)
{
    // device does not have a voltage state! recover!
}
```

The list of states that are recognized be each device can be found in the *device.xml* configuration file. For each device type, the string identifiers that will not cause exceptions with the GetState call are those specified with the **<state>** tag. To be safe, all uses of the GetState function should be done inside of a try block with a corresponding catch statement.

5.5.3 Set a Device Command

A command can be issued to a device pointer using the `void CDevice::SetCommand(std::string, float)` function. If the specified command cannot be found, then this function call will throw an exception. The correct usage of this command should resemble:

```
try
{
    dev->SetCommand("rateOfCharge", -0.25);
}
catch(std::exception & e)
{
    // device does not have a rateOfCharge command! recover!
}
```

5.5.4 Example Usage

The following example code will show how the device framework will be integrated into most modules. In this example, the net generation at a DGI instance is calculated and used to set the charge rate of a battery. As this is an example, the actual calculations involved in the code are nonsensical.

```
#include "CDeviceManager.hpp"
#include <boost/foreach.hpp>
#include <iostream>
#include <set>

void YourModule::PerformCalculation()
{
    std::set<device::CDevice::Pointer> drerSet;
    device::CDevice::Pointer desd;
    float netGeneration;
    float rateOfCharge;

    // retrieve the set of DRER devices
    drerSet = device::CDeviceManager::Instance().GetDevicesOfType("DRER");
    if(drerSet.empty())
    {
        std::cout << "Error! No generators!" << std::endl;
        return;
    }

    // calculate the net DRER generation
    netGeneration = 0;
    try
    {
        BOOST_FOREACH(device::CDevice::Pointer drer, drerSet)
        {
            netGeneration += drer->GetState("generation");
        }
    }
    catch(std::exception & e)
    {
        std::cout << "Error! Generators did not recognize OUTPUT state!" << std::endl;
        return;
    }

    // determine the appropriate battery charge rate (nonsensical)
    rateOfCharge = 0;
    if(netGeneration > 0)
        rateOfCharge = netGeneration;

    // retrieve the DESD device
    desd = device::CDeviceManager::Instance().GetDevice("MyDesd");
```

```
    if(!desd)
    {
        std::cout << "Error! MyDesd device not found!" << std::endl;
        return;
    }

    // set the DESD command
    try
    {
        desd->SetCommand("charge", rateOfCharge);
    }
    catch(std::exception & e)
    {
        std::cout << "Error! Could not set battery CHARGE command!" << std::endl;
    }
}
```

These functions should be sufficient for all modules that need to use physical devices. However, additional functions are provided by the device manager. A list of these functions can be obtained from the device manager header file in the DGI code located at `Broker/src/device/CDeviceManager.hpp`.

CHAPTER 6

Advanced Configuration

6.1 freedm.cfg options

The following options may be specified in a freedm.cfg file. Additionally, many of these values may be set at runtime using command line switches. Use `PosixBroker -h` for a full list of options.

6.1.1 add-host

Adds a peer to the DGI system. This DGI will attempt to communicate with the specified peer. DGI will ignore references to itself in this directive.

Example: `add-host=raichu.freedm:51870`

6.1.2 address

Specifies an interface to bind the DGI to. Defaults to 0.0.0.0 which listens on all interfaces.

Example: `address=192.168.1.120`

6.1.3 port

Specifies the port the DGI should listen to. `add-host` directives on other peers should refer to his port.

Example: `port=51780`

6.1.4 factory-port

Specifies the port for the plug and play session protocol. If omitted, the protocol is not activated.

See *Plug and Play Adapter*

Example: `factory-port=60000`

6.1.5 device-config

Specifies the configuration file for the device types. This file defines the types of devices available and the signals for those devices. If this file is not specified the device framework will not be available.

See *Creating a Virtual Device Type*

Example: `device-config=./config/device.xml`

6.1.6 adapter-config

Specifies the configuration file for the PSCAD/RSCAD interface. If this file is not specified the PSCAD/RSCAD interface will not be available.

See *Other Methods For Connecting the DGI to Physical Devices*

Example: `adapter-config=./config/adapter.xml`

6.1.7 logger-config

Specifies the configuration file used to control the verbosity of the loggers in the DGI. If not specified, the value defaults to `./config/logger.cfg`

See *Using The DGI Logger*

Example `logger-config=./config/logger.cfg`

6.1.8 timings-config

Specifies the configuration file used to set the timings of the DGI. If not specified, the value defaults to `./config/timings.cfg`

See *Configuring Timings*

Example `timings-config=./config/timings.cfg`

6.1.9 topology-config

Specifies the topology file to use if you want FIDs to control the connectivity of DGI. If not specified, the physical topology feature is disabled.

See *Physical Topology*

Example `topology-config=./config/physical.cfg`

6.1.10 migration-step

Specifies the size of quantum of power to use during migrations. This value should be scaled as appropriate for the design of your physical system. If not specified, this value defaults to 1.

Example `migration-step=3`

6.1.11 malicious-behavior

Specifies if the DGI should act "maliciously." This switch will cause this DGI, when it is in a demand state, to ignore accept messages. This will cause a power imbalance which may drive a system to instability. Defaults to 0 which disables the behavior. Setting this value to 1 enables the behavior.

Example `malicious-behavior=1`

6.1.12 check-invariant

Enables the evaluation of a physical invariant that should protect the physical system from DGIs using the malicious-behavior flag. Defaults to 0 which disables the invariant check. Setting this value to 1 enables the check.

Example `check-invariant=1`

6.1.13 verbose

Sets the logger level of all loggers in the system. Individual loggers can be overriden by values in a `logger.cfg` file. If omitted, this value will be set to 5. Zero is the lowest value, 8 is the highest (most verbose) setting.

See *Using The DGI Logger*

Example `verbose=0`

6.1.14 devices-endpoint

Specify an interface that the devices framework will use to communicate. If not specified, the devices will use any available interface to communicate.

Example `devices-endpoint=192.168.1.150`

6.2 Multiple DGI Per Host

6.2.1 Performance Consideration

Running multiple DGI per machine will affect timings. The provided timings.cfg files are designed with one DGI per host in mind. If you've got a TS-7800 in a group with Intel Core 2 machines, the TS-7800 is not the one that should be running two DGI at once.

6.2.2 Configuring DGI For Multiple Hosts

To get multiple DGI on the same machine, copy your `PosixBroker` executable and the `/config` directory to a new location on that machine, then set up the second DGI to use a different port than the first and make sure all of its peers know about the different port. Here are example configuration files for a three node group, with two DGI on one computer and the third on another:

Zapos DGI #1:

```
# Portion of freedm.cfg for zapdos.freedm DGI #1

add-host=zapdos.freedm:50001
add-host=raikou.freedm:50000

address=0.0.0.0
port=50000
```

Zapdos DGI #2:

```
# Portion of freedm.cfg for zapdos.freedm DGI #2

add-host=zapdos.freedm:50000
add-host=raikou.freedm:50000

address=0.0.0.0
port=50001
```

Raikou DGI:

```
# Portion of freedm.cfg for DGI on raikou.freedm

add-host=zapdos.freedm:50000
add-host=zapdos.freedm:50001

address=0.0.0.0
port=50000
```

Since the hostname you choose is the unique identifier of the DGI in its group, it has to be specified exactly the same in each DGI's configuration file and each DGI in the group must be able to resolve the hostname to the same host. This implies that localhost is NEVER a valid hostname in an add-host directive. It won't work for groups on multiple machines, and it won't even work for groups where each DGI is on the same machine since you don't get to specify the hostname that the DGI uses for itself.

6.3 Troubleshooting Hostnames

6.3.1 "Are Your Hostnames Configured Correctly Error"

If you observe this error:

```
2013-Mar-19 11:39:37.207802 : PosixMain.cpp : Error(2):
    Exception caught in Broker: Could not resolve the endpoint victory-road:1870 (victory-road:1870)
```

By default, most systems cannot resolve their own hostnames, unless, of course, you get your hostname from DHCP, in which case it could (potentially) be resolvable via DNS. But local hostname resolution is mandatory for FREEDM (and many other programs) to work. You have two options:

- Specify your own hostname in */etc/hosts*. Some distros (e.g. Debian-based) do this for you automatically. This will suffice for static hostnames, but it will not work if you get your transient hostname from DHCP unless you add your transient hostname to */etc/hosts* AND know that it will never change. (Alternatively, older versions of NetworkManager modify */etc/hosts* for you.)
- Enable local hostname resolution using nss-myhostname. nss-myhostname is available on all modern distros, but not necessarily installed or enabled by default. To enable, you will have to add the module "myhostname" to the end of the hosts line in */etc/nsswitch.conf*. (nss-myhostname has been merged into systemd 197, so on newer distros you might not need a separate package for this to work.)

Using nss-myhostname will allow you to seamlessly switch networks without losing resolvability, so we strongly recommend this approach.

6.3.2 All of my DGI are on the same machine, and I can't form groups

First, a bit of background info. You actually have a few different types of hostnames. If you have systemd, run *hostnamectl* to see them all.

- static hostname, stored in */etc/hostname* by systemd and also on Debian-based systems.
- pretty hostname, which doesn't matter. (If you have GNOME 3.6 or higher, this is what you see on the Details panel in System Settings.)
- transient/dynamic/kernel hostname, may be assigned by your network configuration (e.g. DHCP) but will otherwise be the same as your static hostname (in which case *hostnamectl* will not display it). This is the most important hostname since it is what can be resolved by Linux system calls. You can also check this with the *hostname* command or *cat /proc/sys/kernel/hostname*

Other computers on your network may be able to resolve your transient hostname via DNS provided it is distinct from your static hostname, but they cannot resolve your static hostname unless you purposefully set it to something they can resolve via DNS or else manually modify */etc/hosts* on those machines. If you're reading this page, then you can already form groups of DGI with one DGI per host, so you've already figured out how this works and know that your transient hostname is what goes in the add-host directive in *freedm.cfg*, so that's not a problem for you. And since you read the page up to this point, you've also made sure that this hostname is resolvable locally.

So having considered all of that, you're using a static hostname (by far the most common out-of-the-box configuration) that's locally resolvable, but you can't form groups with two DGI on the same machine. This is a known bug, but we're not quite sure what's wrong. It's most likely a DGI bug, but it could very well be a Boost bug or even an operating system bug. Try manually adding your hostname to /etc/hosts.

6.4 Configuring Timings

6.4.1 Timing Settings

The real time scheduler uses a round robin approach. A module's individual processing time as its "Phase". A round is composed of a GM Phase, an SC Phase and a LB Phase.

Every DGI instance must have the same timing profile.

6.4.2 Using The Excel Sheet To Create New Timings

There are three parameters which dictate how the sheet produces values. One (TProc) is computed using experimental data and the others (Nodes and TPTM) are adjustable by the user. To generate a set of timing parameters one must first collect the requisite data. Here are the parameters that must be collected and the techniques that can be used to find them. The excel sheet can be found as *FREEDM-Timings.xlsx* in the `OtherDocs` folder.

Blue Box

In the blue shaded area you can change the user-tuneable parameters for the DGI.

No. of DGI – This value is the size of the largest group you would be able to form.

In Channel Time (aka TPTM) – You can collect this value by performing a ping, preferably with packets of at least the maximum size you intend to send. Messages tend to stay at less than 1 MTU so approximately 1400 bytes is sufficient.

You can also use this to describe the maximum amount of latency the DGI will experience. A higher latency means a higher in channel time.

No. of Migrations – This value is how many migrations should occur each load-balancing phase. Depending on your circumstances, you may want this to be small (so only one migration happens) or big (so that lots of migrations happen)

Premerge Intervals – The number of intervals used to handle the race condition in the leader election algorithm. The idea is that the node with the lowest identifier should be the highest priority for becoming the leader. If that node is slow, the next lowest identifier should have priority and so on. The premerge slots them into an ordering for accomplishing this. Tau is the smallest premerge time of the nodes participating in the election. This parameter changes the number of premerge slots available.

Pink Box

Values in the pink box are collected in a run of the system. To capture the requisite data, run a small slice of the system with ONLY two (More nodes will mislead the calculations because of how the processing queue works) nodes and capture a log with the Group Management log level set to 6. From there, grep out the lines containing the phrase "after query sent" and copy the timings from these lines into the excel sheet. There's an example of what you are looking for below. These values can go directly into the "AYC Resp. Time" column (it doesn't matter if they are AYC or AYT; the formula is the same). The Processing column computes how much time is spent by the DGI packing and unpacking the message. The more data you collect, the more accurate your timings may be:

```
$ ./PosixBroker -v 6 &> timing.dat
$ cat timing.dat | grep -i "after query sent"
AYC response received 00:00:00.038365 after query sent
AYT response received 00:00:00.054224 after query sent
AYT response received 00:00:00.055469 after query sent
AYT response received 00:00:00.056330 after query sent
AYT response received 00:00:00.056489 after query sent
AYT response received 00:00:00.056032 after query sent
AYT response received 00:00:00.055191 after query sent
AYT response received 00:00:00.055169 after query sent
```

Green Box

The green box shows how the timings from the pink box are distributed. Q0 is the minimum observed time and Q4 is the maximum. Q2 is the median. If you use timings from Q4, then the processing time allotted is the maximum observed processing time from the values you put in the pink box.

Orange Box

The orange box shows the individual events in the system. These events are composed together to create the timings parameters in the purple box.

Purple Box

The purple box has each of the timing parameters. Select the desired quartile and transfer the values to the specified parameters.

6.4.3 Parameter Descriptions

Group Management Settings

- GM_PHASE_TIME - The time allotted to Group Management as a whole should be greater than max(GM_AYC_RESPONSE_TIMEOUT + GM_PREMERGE_MAX_TIMEOUT + GM_INVITE_RESPONSE_TIMEOUT,GM_AYT_RESPONSE_TIMEOUT)
- GM_PREMERGE_MIN_TIMEOUT - The minimum amount of time that a node should wait before deciding the node with highest priority has crashed.
- GM_PREMERGE_GRANULARITY - The step size for the premerge timeout.
- GM_PREMERGE_MAX_TIMEOUT - The maximum amount of time that a node should wait before deciding all other nodes with higher priority have crashed.
- GM_AYC_RESPONSE_TIMEOUT - How long nodes have to respond to the coordinator before being removed from the group.
- GM_INVITE_RESPONSE_TIMEOUT - How long nodes have to respond to invitations from a coordinator If you have trouble establishing groups this is a good parameter to adjust.
- GM_AYT_RESPONSE_TIMEOUT - How long the coordinator has to respond to keep alive messages from the member nodes. If groups break often, try increasing this parameter.

Load Balancing Settings

- LB_PHASE_TIME - Length of the LB phase. Recommended to be set to (LB_GLOBAL_TIMER * Desired number of migrations). It is also recommended to keep the number of migrations per phase to be low.
- LB_ROUND_TIME - Length of time in between individual migrations. Should be set to be longer than the time required to do an individual migration.
- LB_REQUEST_TIMEOUT - The amount of time another process has to respond to a load balancing message.

State Collection Settings

- SC_PHASE_TIME - The amount of time needed for state collection to run. It should be set to a value large enough that the state can be collected each time.

Special Parameters

- CSRC_RESEND_TIME - The amount of time that the send and wait protocol should wait before considering a packet lost (No ACK). Should be greater than 1 RTT. If this is set too low it will appear that devices are not receiving messages.
- CSRC_DEFAULT_TIMEOUT - The time that a message should be considered to be worth sending if the module does not specify a TTL.

6.4.4 Some More Tips For Working Out Timings

Setting the CSRC_RESEND_TIME too low can make it look like all the modules are broken, when in fact, the receiver is being flooded by resent messages. A good diagnostic is to squelch all output except GM's and watch for the AYC queries and replies to be passed back and forth.

Group formation is highly dependent on the GM_INVITE_RESPONSE_TIMEOUT parameter. Setting this value too low will not allow for a sufficient amount of time to collect responses from coordinators and invites. If you see a lot of "Unexpected AYC responses" then this parameter should be increased.

Group stability is dependent on the GM_AYT_RESPONSE_TIMEOUT parameter. If this is set to low the member nodes won't give the coordinator enough time to respond to all their requests, and will leave the group. If you see the group membership changing frequently, or the group id constantly changing, try increasing this parameter.

There is no hard boundary between modules. If your groups are stable, but state collections are not finishing, then consider reducing the number of LB migrations per phase or increasing state collection's time. Watch the output of the Broker module. It can report how long it is scheduling modules for. If you frequently see modules being schedule for less than their phase time then you should increase the time for other modules; they are plundering another module's time.

CHAPTER 7

DGI Modules Reference

7.1 State Collection

DGI state collection provides a method for collecting a casually consistent state of devices in a group of DGI.

7.1.1 Using State Collection

State collections are requested using the DGI message passing interface. Modules submit requests to state collection, state collection runs those collections during it's phase, and returns the collected state a message to the requesting module.

Supported Signal Types

The following signal and device types are supported. To support more signals, you'll need to modify the StateCollection code. See *Adding New Signal Types*

Device Type	Supported Signal(s)
Sst	gateway
Drer	generation
Desd	storage
Load	drain
Fid	state

Requesting State Collection

State collection requests are initiated with a `sc::RequestMessage`. The message must contain:

- *module* - The module that requests the state collection (in order to return the result to the requesting module. Set with **set_module()**
- One or more `DeviceSignalRequestMessages` which contain the device type and signal you wish to collect. Each `DeviceSignalRequestMessages` must have their *type* and *signal* values set. *type* is the type of device to collect the value from (For example, "Sst") and the specific signal from that device (For example, "gateway"). State collection will collect the values from the first device of that type attached to the DGI.

Example: Collecting Single Device/Value

This example requests values from one device (**SST**) per DGI in the group:

```
sc::StateCollectionMessage msg;
sc::RequestMessage * submsg = msg.mutable_request_message();
submsg->set_module("YOURMODULE");

sc::DeviceSignalRequestMessage * subsubmsg = submsg->add_device_signal_request_message();
subsubmsg->set_type("Sst");
subsubmsg->set_signal("gateway");

ModuleMessage m;
m.mutable_state_collection_message()->CopyFrom(msg);
m.set_recipient_module("sc");
```

The module prepares an `sc::StateCollectionMessage` then accesses its child `request_message`, and adds the signals that need to be collected (In this case, SST's gateway values). It then packs the message into a module message and addresses it to State Collection.

Example: Multiple Devices

This example requests values from multiple devices (**SST, DESD, DRER**) per DGI in the group:

```
sc::StateCollectionMessage msg;
sc::RequestMessage * state_request = msg.mutable_request_message();
state_request->set_module("YOURMODULE");

sc::DeviceSignalRequestMessage * device_state;
device_state = state_request->add_device_signal_request_message();
device_state->set_type("Sst");
device_state->set_signal("gateway");

device_state = state_request->add_device_signal_request_message();
device_state->set_type("Desd");
device_state->set_signal("storage");

device_state = state_request->add_device_signal_request_message();
device_state->set_type("Drer");
device_state->set_signal("generation");

ModuleMessage m;
m.mutable_state_collection_message()->CopyFrom(msg);
m.set_recipient_module("sc");
```

Collected State Response

State collection will send back the following state response, which contains the collected state:

```
message CollectedStateMessage
{
    repeated double gateway = 1;
    repeated double generation = 2;
    repeated double storage = 3;
    repeated double drain = 4;
    repeated double state = 5;
```

```
    required int32 num_intransit_accepts = 6;
}
```

Values can be accessed by iterating over the values in the `sc::CollectedStateMessage` fields. Each of the fields in the above message can be accessed through a function call to that field's name. For example, accessing the gateway values can be done with the gateway() method of the `sc::CollectedStateMessage`.

Here is an example **HandleCollectedState** method:

```
HandleCollectedState(const sc::CollectedStateMessage &m, const CPeerNode& peer)
{
    ...
    //for SST device
    BOOST_FOREACH(float v, m.gateway())
    {
        //print out each gateway value from collected message
        Logger.Info << "Gateway value is " << v << std::endl;
    }

    //for DRER device
    BOOST_FOREACH(float v, m.generation())
    {
        //print out each generation value from collected message
        Logger.Info << "Generation value is " << v << std::endl;
    }

    //for DESD device
    BOOST_FOREACH(float v, m.storage())
    {
        //print out each storage value from collected message
        Logger.Info << "Storage value is " << v << std::endl;
    }
}
```

Adding New Signal Types

To add a new signal or device to state collection, first add a new entry to the CollectedState message in `src/messages/StateCollection.proto`. For example, to add a new frequency signal to a new or existing device one line related to signal type frequency should be added as follows:

```
message CollectedStateMessage
{
    repeated double gateway = 1;
    repeated double generation = 2;
    repeated double storage = 3;
    repeated double drain = 4;
    repeated double state = 5;
    repeated double frequency = 6; // New line for the new signal.
    required int32 num_intransit_accepts = 7;
}
```

Make sure you adjust the assigned numbers for the fields accordingly.

Next, in the `StateResponse()` method of `sc/StateCollection.cpp` add the new device or signal. In this example, we have added both a new device (Omega) and a new signal to that device (frequency):

```
if (dssm.type() == "Sst")
{
```

```
    if(dssm.count()>0)
    {
        csm->add_gateway(dssm.value());
    }
    else
    {
        csm->clear_gateway();
    }
}
else if (dssm.type() == "Drer")
{
    if(dssm.count()>0)
    {
        csm->add_generation(dssm.value());
    }
    else
    {
        csm->clear_generation();
    }
}
else if (dssm.type() == "Desd")
{
    if(dssm.count()>0)
    {
       csm->add_storage(dssm.value());
    }
    else
    {
       csm->clear_storage();
    }
}
else if (dssm.type() == "Omega")
{
    if(dssm.count()>0)
    {
        csm->add_frequency(dssm.value());
    }
    else
    {
        csm->clear_frequency();
    }
}
```

When LB requests the state of the OMEGA device with SST, DESD, DRER, requested message will need to add following code related with the OMEGA device in LoadBalance.cpp file:

```
sc::StateCollectionMessage msg;
sc::RequestMessage * state_request = msg.mutable_request_message();
state_request->set_module("lb");

sc::DeviceSignalRequestMessage * device_state;
device_state = state_request->add_device_signal_request_message();
device_state->set_type("Sst");
device_state->set_signal("gateway");

device_state = state_request->add_device_signal_request_message();
device_state->set_type("Desd");
device_state->set_signal("storage");
```

```
device_state = state_request->add_device_signal_request_message();
device_state->set_type("Drer");
device_state->set_signal("generation");

// New device and signal
device_state = state_request->add_device_signal_request_message();
device_state->set_type("Omega");
device_state->set_signal("frequency");
```

When LB handles received states back in LoadBalance.cpp file, the following code related with OMEGA with signal type frequency should be added:

```
HandleCollectedState(const sc::CollectedStateMessage &m)
{
    ...
    //for SST device
    BOOST_FOREACH(float v, m.gateway())
    {
        //print out each gateway value from collected message
        Logger.Info << "Gateway value is " << v << std::endl;
    }

    //for DRER device
    BOOST_FOREACH(float v, m.generation())
    {
        //print out each generation value from collected message
        Logger.Info << "Generation value is " << v << std::endl;
    }

    //for DESD device
    BOOST_FOREACH(float v, m.storage())
    {
        //print out each storage value from collected message
        Logger.Info << "Storage value is " << v << std::endl;
    }

    // New device and signal
    //for OMEGA device
    BOOST_FOREACH(float v, m.frequency())
    {
        //print out each frequency value from collected message
        Logger.Info << "Frequency value is " << v << std::endl;
    }
}
```

7.1.2 Implementation Details

Theory: Algorithm in State Collection

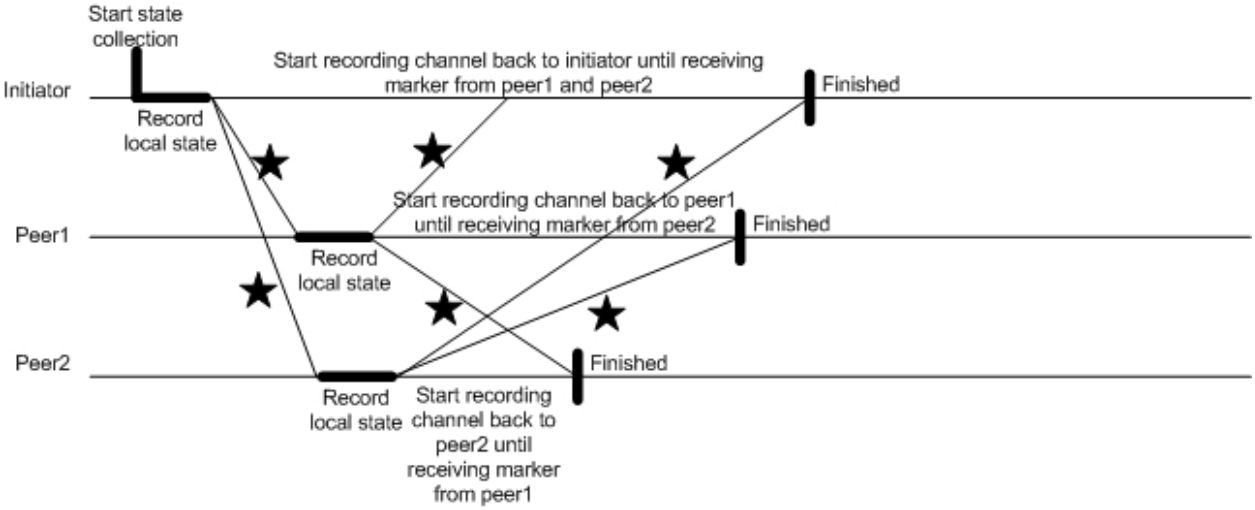
The DGI State Collection module is implemented based on the Chandy-Lamport algorithm [1], which is used to collect consistent states of all participants in a distributed system. A consistent global state is one corresponding to a consistent cut. A consistent cut is left closed under the causal precedence relation. In another words, if one event is belongs to a cut, and all events happened before this event also belong to the cut, then the cut is considered to be consistent. The algorithm works as follows:

- The initiator starts state collection by recording its own states and broadcasting a marker out to other peers. At

the same time, the initiator starts recording messages from other peers until it receives the marker back from other peers.

- Upon receiving the marker for the first time, the peer records its own state, forwards the marker to others (include the initiator) and starts recording messages from other peers until it receives the marker back from other peers.

The following diagram illustrates the Chandy-Lamport algorithm working on three nodes. The initiator is the leader node chosen by Group Management module in DGI.



Message Passing

State collection defines the following message types

- **MarkerMessage**
- **DeviceSingalStateMessage**
- **StateMessage**
- **DeviceSingalRequestMessage**
- **RequestMessage**
- **CollectedStateMessage**
- **StateCollectionMessage**

SCAgent Reference

State Collection Functions

- HandleIncomingMessage: "Downcasts" incoming messages into a specific message type, and passes the message to an appropriate handler.
- HandleRequest: Handle RequestMessage from other modules. Extract type and value of devices and insert into a list with certain format.
- Initiate: Initiator records its local state and broadcasts marker to the peer node.
- TakeSnapshot: Record its local states according to the device list.
- HandleMarker: Handle MarkerMessage.
- SaveForward: Save its local state and send marker out.

- SendStateBack: Send its collected state back to the initiator.
- HandleState: Handle StateMessage return back from the peer node.
- StateResponse: Handle collected states and prepare CollectedStateMessage back to the requested module. Following are signal types that has been defined by the current protocol buffers for CollectedStateMessage.

class `freedm::broker::sc::`**`SCAgent`** **Description:**

Declaration of Chandy-Lamport Algorithm Each node that wants to initiate the state collection records its local state and sends a marker message to all other peer nodes. Upon receiving a marker for the first time, peer nodes record their local states and start recording any message from incoming channel until receive marker from other nodes (these messages belong to the channel between the nodes).

Public Functions

`SCAgent``()`
Constructor.

SCAgent

Description:

: Constructor for the state collection module.

Precondition:

PoxisMain prepares parameters and invokes module.

Postcondition:

Object initialized and ready to enter run state.

Limitations:

: None

Protected Functions

CPeerNode **`GetMe`**`()`
Gets a CPeerNode representing this process.

GetMe

Description:

Gets a CPeerNode that refers to this process.

Return

A CPeerNode referring to this process.

std::string **`GetUUID`**`()` const
Gets the UUID of this process.

GetUUID

Description:

Gets this process's UUID.

Return

This process's UUID

Private Functions

CPeerNode **AddPeer**(CPeerNode *peer*)

Add a peer to peer set from a pointer to a peer node object.

AddPeer

Description:

Add a peer to peer set from a pointer to a peer node object m_AllPeers is a specific peer set for SC module.

Precondition:

m_AllPeers

Postcondition:

Add a peer to m_AllPeers

Return

a pointer to a peer node

Parameters

- `peer` -

CPeerNode **GetPeer**(std::string *uuid*)

Get a pointer to a peer from UUID.

GetPeer

Description:

Get a pointer to a peer from UUID. m_AllPeers is a specific peer set for SC module.

Precondition:

m_AllPeers

Postcondition:

Add a peer to m_AllPeers

Return

a pointer to the peer

Parameters

- `uuid` - string

void **HandleAccept**(CPeerNode *peer*)

Handle receiving messages.

This function will be called to handle Accept messages from LoadBalancing. Normally, state collection can safely ignore these messages, but if they arrive during state collection's own phase, then there is a problem and they need to be added to the collected state.

Parameters

- `peer` - the DGI that sent the message

virtual void **HandleIncomingMessage**(boost::shared_ptr< const ModuleMessage > *msg*, CPeerNode *peer*)

Handles received messages.

"Downcasts" incoming messages into a specific message type, and passes the message to an appropriate handler.

Parameters

- `msg` - the incoming message
- `peer` - the node that sent this message (could be this DGI)

void **HandleMarker**(const MarkerMessage & *msg*, CPeerNode *peer*)

SCAgent::HandleMarker

Description:

This function will be called to handle marker message.

Key:

sc.marker

Precondition:

Messages are obtained.

Postcondition:

parsing marker messages based on different conditions.

Interaction Peers:

Invoked by dispatcher, other SC

Parameters

- `msg` - the received message
- `peer` - the node

void **HandlePeerList**(const gm::PeerListMessage & *msg*, CPeerNode *peer*)

SCAgent::HandlePeerList

Description:

This function will be called to handle PeerList message.

Key:

any.PeerList

Precondition:

Messages are obtained.

Postcondition:

parsing messages, reset to default state if receiving PeerList from different leader.

Interaction Peers:

Invoked by dispatcher, other SC

Parameters

- `msg` - the received message

- `peer` - the node

void **`HandleRequest`**(const RequestMessage & *msg*, CPeerNode *peer*)
SCAgent::HandleRequest

Description:

This function will be called to handle state collect request message.

Key:

sc.request

Precondition:

Messages are obtained.

Postcondition:

start state collection by calling *Initiate()*.

Parameters

- `msgpeer` -

void **`HandleState`**(const StateMessage & *msg*, CPeerNode *peer*)
SCAgent::HandleState

Description:

This function will be called to handle state message.

Key:

sc.state

Precondition:

Messages are obtained.

Postcondition:

parsing messages based on state or in-transit channel message.

Interaction Peers:

Invoked by dispatcher, other SC

Parameters

- `msg` - the received message
- `peer` - the node

void **`Initiate`**()
Initiator starts state collection.

Initiate

Description:

Initiator redcords its local state and broadcasts marker.

Precondition:

Receiving state collection request from other module.

Postcondition:

The node (initiator) starts collecting state by saving its own states and broadcasting a marker out.

Device I/O:

TakeSnapshot()

Return

Send a marker out to all known peers

Citation:

Distributed Snapshots: Determining Global States of Distributed Systems, ACM Transactions on Computer Systems, Vol. 3, No. 1, 1985, pp. 63-75

void **SaveForward**(StateVersion *latest*, const MarkerMessage & *msg*)
Peer save local state and forward maker.

SaveForward

Description:

SaveForward is used by the node to save its local state and send marker out.

Precondition:

Marker message is received.

Postcondition:

The node saves its local state and sends marker out.

Parameters

- `latest` - the current marker's version
- `msg` - the message tp semd

void **SendStateBack**()
Peer sends collected states back to the initiator.

SendStateBack

Description:

SendStateBack is used by the peer to send collect states back to initiator.

Precondition:

Peer has completed its collecting states in local side.

Postcondition:

Peer sends its states back to the initiator.

Limitation:

Currently, only sending back gateway value and channel transit messages.

void **StateResponse**()
Initiator sends collected states back to the request module.

StateResponse

Description:

This function deals with the collectstate and prepare states sending back.

Precondition:

The initiator has collected all states.

Postcondition:

Collected states are sent back to the request module.

Interaction Peers:

other SC processes

Return

Send message which contains gateway values and channel transit messages

Limitation:

Currently, only gateway values and channel transit messages are collected and sent back.

void **TakeSnapshot** (const std::vector< std::string > & *devicelist*)
Save local state.

TakeSnapshot

Description:

TakeSnapshot is used to collect local states.

Precondition:

The initiator starts state collection or the peer receives marker at first time.

Postcondition:

Save local state in container m_curstate

Limitation:

Currently, it is used to collect only the gateway values for LB module

Private Members

std::multimap< StateVersion, StateMessage > **collectstate**
collect states container and its iterator

PeerSet **m_AllPeers**
all known peers

unsigned int **m_countdone**
count number of "Done" messages

unsigned int **m_countmarker**
count number of marker

unsigned int **m_countstate**
count number of states

StateMessage **m_curstate**
current state

StateVersion **m_curversion**
current version of marker

std::string **m_module**
module that request state collection

bool **m_NotifyToSave**
flag to indicate save channel message

std::string **m_scleader**
save leader

Private Static Functions

ModuleMessage **PrepareForSending**(const StateCollectionMessage & *message*, std::string *recipient* = "sc")
Wraps a StateCollectionMessage in a ModuleMessage.

Wraps a StateCollectionMessage in a ModuleMessage.

Return

a ModuleMessage containing a copy of the StateCollectionMessage

Parameters

- `message` - the message to prepare. If any required field is unset, the DGI will abort.
- `recipient` - the module (sc/lb/gm/clk etc.) the message should be delivered to

7.1.3 References

[1] Distributed Snapshots: Determining Global States of Distributed Systems, ACM Transactions on Computer Systems, Vol. 3, No. 1, 1985, pp. 63-75.

7.2 Group Management Reference

Group management provides consistent list of active DGI processes as well as identifies a primary process for initiating distributed algorithms.

7.2.1 Using Group Management

GMAgent::ProcessPeerList is available by including gm/GroupManagement.hpp

Group management will send all modules in the system a PeerList message when the list of active processes in the system changes. This message will also contain the identity of the leader process. Group management provides a static method for to aid processing this message.

PeerSet **ProcessPeerList**(const PeerListMessage & *msg*)
Handles Processing a PeerList.

GMAgent::ProcessPeerList

Description:

Provides a utility function for correctly handling incoming peer lists.

Return

A PeerSet with all nodes in the group.

Parameters

- `msg` - The message to parse

This method returns a *PeerList* of the processes in the group. The sender of the message will always be the group leader.

An example of processing the PeerList message is in *Receiving Messages*

7.2.2 Physical Topology

Group management can be configured to respect the topology of the physical network: it will not group two DGI unless a physical path exists between the two processes (as opposed to a cyber path).

Directions on configuring physical topology can be found in *Physical Topology*

7.2.3 Embedding Group State In Simulation

The group state is by manipulating a "Logger" device on the FREEDM system. The value of device is updated each time the check or timeout procedure is called, that is, once at the beginning of the Group Management phase.

Data is stored in the device as a bit field. Because of the way data is passed to and from the RTDS and PSCAD the data is transported as a float, although the individual bits are unaffected by this process, it may be necessary to convert the float to an unsigned integer to be able to preform the bit fiddling needed to access the information.

This setup assumes that all DGI are aware of all other DGI in the system and that all the Group Management status tables are the same. That is, there is not a DGI some DGI is aware of that some other DGI isn't aware of. This is reasonable because of the experiments we are currently running, and the fact that the container for the other DGIs in the system is a map so when iterated the items are returned in alpha numeric order.

UUIDS ARE ENTERED IN THE BITFIELD IN ASCENDING ALPHANUMERIC ORDER.

Bit Field Structure

Note that 2^0 is set based on the endianess of an x86_64 machine.

1. (2^0) - Set to 1 if the DGI editing the device is the coordinator.
1. (2^1) - Set to 1 if the DGI with the first UUID is in the same group as the DGI editing the bitfield
1. (2^2) - Set to 1 if the DGI with the second UUID is in the same group as the DGI editing the bitfield.
1. (2^n) - Set to 1 if the DGI with the nth UUID is in the same group as the DGI editing the bitfield.

Bits of non-members will be set to 0. The SGI will always set the bit relating to its own UUID to 1.

In order to use this feature, you need to define a *Logger* device with a "groupStatus" field. The DGI will write its group state to the first *Logger* device on the system.

7.2.4 Implementation Details

Algorithm

Group Management is an implementation of the Garcia-Molina Invitation Election Algorithm found in "Elections in a Distributed System" published 1982 in IEEE Transactions on Computers.

GMAgent Reference

class `freedm::broker::gm::`**`GMAgent`**
Declaration of Garcia-Molina Invitation Leader Election algorithm.

Public Types

enum **[anonymous]**
Module states.

Values:

NORMAL

DOWN

RECOVERY

REORGANIZATION

ELECTION

Public Functions

GMAgent()
Constructor for using this object as a module.

GMAgent::GMAgent

Description:

Constructor for the group management module.

Limitations:

None

Precondition:

None

Postcondition:

Object initialized and ready to enter run state.

~GMAgent()
Module destructor.

GMAgent::~GMAgent

Description:

Class desctructor

Precondition:

None

Postcondition:

The object is ready to be destroyed.

int **Run**()
Called to start the system.

GMAgent::Run

Description:

Main function which initiates the algorithm

Precondition:

connections to peers should be instantiated

Postcondition:

execution of group management algorithm

Public Static Functions

PeerSet **ProcessPeerList**(const PeerListMessage & *msg*)
Handles Processing a PeerList.

GMAgent::ProcessPeerList

Description:

Provides a utility function for correctly handling incoming peer lists.

Return

A PeerSet with all nodes in the group.

Parameters

- `msg` - The message to parse

Protected Functions

CPeerNode **GetMe**()
Gets a CPeerNode representing this process.

GetMe

Description:

Gets a CPeerNode that refers to this process.

Return

A CPeerNode referring to this process.

std::string **GetUUID**() const
Gets the UUID of this process.

GetUUID

Description:

Gets this process's UUID.

Return

This process's UUID

Private Functions

ModuleMessage **Accept**()
Creates an Accept Message.

GMAgent::Accept

Description:

Creates a new accept message from this node

Precondition:

This node is in a group.

Postcondition:

No change.

Return

A GroupManagementMessage with the contents of an Accept message

CPeerNode **AddPeer**(std::string *uuid*)
Adds a peer to the peer set from UUID.

GMAgent::AddPeer

Description:

Adds a peer to allpeers by uuid.

Precondition:

the UUID is registered in the connection manager

Postcondition:

A new peer object is created and inserted in CGlobalPeerList::instance().

Return

A pointer to the new peer

Parameters

- `uuid` - of the peer to add.

CPeerNode **AddPeer**(CPeerNode *peer*)
Adds a peer from a pointer to a peer node object.

GMAgent::AddPeer

Description:

Adds a peer to all peers by pointer.

Precondition:

the pointer is valid

Postcondition:

The peer object is inserted in CGlobalPeerList::instance().

Return

A pointer that is the same as the input pointer

Parameters

- `peer` - a pointer to a peer.

ModuleMessage **AreYouCoordinator**()
Creates AYC Message.

GMAgent::AreYouCoordinator

Description:

Creates a new Are You Coordinator message, from this object

Precondition:

The UUID is set

Postcondition:

No change

Return

A GroupManagementMessage with the contents of an Are You Coordinator Message.

Limitations:

: Can only author messages from this node.

ModuleMessage **`AreYouCoordinatorResponse`**(std::string *payload*, int *seq*)
Creates A Response message.

GMAgent::AreYouCoordinatorResponse

Description:

Creates a response message (Yes/No) message from this node

Precondition:

This node has a UUID.

Postcondition:

No change.

Return

A GroupManagementMessage with the contents of a Response message

Parameters

- `payload` - Response message (typically yes or no)
- `seq` - sequence number? (?)

ModuleMessage **`AreYouThere`**()
Creates a AYT, used for Timeout.

GMAgent::AreYouThere

Description:

Creates a new AreYouThere message from this node

Precondition:

This node is in a group.

Postcondition:

No Change.

Return

A GroupManagementMessage with the contents of an AreYouThere message

ModuleMessage **AreYouThereResponse**(std::string *payload*, int *seq*)
Creates A Response message.

GMAgent::AreYouThereResponse

Description:

Creates a response message (Yes/No) message from this node

Precondition:

This node has a UUID.

Postcondition:

No change.

Return

A GroupManagementMessage with the contents of a Response message

Parameters

- `payload` - Response message (typically yes or no)
- `seq` - sequence number? (?)

void **Check**(const boost::system::error_code & *err*)
Checks for other up leaders.

GMAgent::Check

Description:

This method queries all nodes to Check and see if any of them consider themselves to be Coordinators.

Precondition:

This node is in the normal state.

Postcondition:

Output of a system state and sent messages to all nodes to Check for Coordinators.

Citation:

GroupManagement (Check).

Parameters

- `err` - Error associated with calling timer.

std::string **Coordinator**() const
Returns the coordinators uuid.

CPeerNode **GetPeer**(const std::string & *uuid*)
Gets a pointer to a peer from UUID.

GMAgent::GetPeer

Description:

Gets a peer from All Peers by uuid

Precondition:

None

Postcondition:

None

Return

A pointer to the requested peer, or a null pointer if the peer could not be found.

Parameters

- `uuid` - The uuid of the peer to fetch

int **`GetStatus`**`()` const
Gets the status of a node.

Description:

returns the status stored in the node as an an integer. This means that in an inherited class, you may either define integer constants or an enumeration to generate status values.

void **`HandleAccept`**(const AcceptMessage & *msg*, CPeerNode *peer*)
Handles recieving accept messsages.

GMAgent::HandleAccept

Description:

Handles receiveing the invite accept

Key:

gm.Accept

Precondition:

In an election, and invites have been sent out.

Postcondition:

The sender is added to the tenative list of accepted peers for the group, if the accept messag is for the correct group identifier.

Interaction Peers:

Any node which has received an invite. (This can be any selection of the global peerlist, not just the ones this specific node sent the message to.)

void **`HandleAreYouCoordinator`**(const AreYouCoordinatorMessage & *msg*, CPeerNode *peer*)
Handles recieving are you coordinator messages.

GMAgent::HandleAreYouCoordinator

Description:

Handles receiveing the AYC message

Key:

gm.AreYouCoordinator

Precondition:

None

Postcondition:

The node responds yes or no to the request.

Interaction Peers:

Any node in the system is eligible to receive this at any time.

void **HandleAreYouThere**(const AreYouThereMessage & *msg*, CPeerNode *peer*)
Handles recieving are you there messsages.

GMAgent::HandleAreYouThere

Description:

Handles recieving the AYT message

Key:

gm.AreYouThere

Precondition:

None

Postcondition:

The node responds yes or no to the request.

Interaction Peers:

Any node in the system is eligible to receive this message at any time.

virtual void **HandleIncomingMessage**(boost::shared_ptr< const ModuleMessage > *msg*, CPeerNode *peer*)
Handles received messages.

"Downcasts" incoming messages into a specific message type, and passes the message to an appropriate handler.

Parameters

- `msg` - the incoming message
- `peer` - the node that sent this message (could be this DGI)

void **HandleInvite**(const InviteMessage & *msg*, CPeerNode *peer*)
Handles recieving invite messages.

GMAgent::HandleInvite

Description:

Handles recieving the invite

Key:

gm.Invite

Precondition:

The system is in the NORMAL state

Postcondition:

The system updates its group to accept the invite and switches to reorganization mode; it will wait for a timeout. If the Ready/Peerlist has not arrived, it will enter recovery. Otherwise, it will resume work as part of the group it was invited to.

Interaction Peers:

Any node could send an invite at any time.

void **HandlePeerList**(const PeerListMessage & *msg*, CPeerNode *peer*)
Hadles recieving peerlists.

GMAgent::HandlePeerList

Description:

Handles receiveing the peerlist.

Key:

any.PeerList

Precondition:

The node is in the reorganization or normal state

Postcondition:

The node's peerlist is updated to match that of the incoming message if the message has come from his coordinator.

Interaction Peers:

Coordinator only.

void **HandlePeerListQuery**(const PeerListQueryMessage & *msg*, CPeerNode *peer*)
Handles recieving peerlist requests.

GMAgent::HandlePeerListQuery

Description:

Handles responding to peerlist queries.

Key:

gm.PeerListQuery

Precondition:

None

Postcondition:

Dispatched a message to the requester with a peerlist.

Interaction Peers:

Any. (Local or remote, doesn't have to be a member of group)

void **HandleResponseAYC**(const AreYouCoordinatorResponseMessage & *msg*, CPeerNode *peer*)
Handles recieving AYC responses.

GMAgent::HandleResponseAYC

Description:

Handles recieving the Response

Key:

gm.Response.AreYouCoordinator

Precondition:

The timer for the AYC request batch the original request was a part of has not expired

Postcondition:

The response from the sender is noted, if yes, the groups will attempt to merge in the future.

Interaction Peers:

A node the AYC request was sent to.

void **HandleResponseAYT**(const AreYouThereResponseMessage & *msg*, CPeerNode *peere*)
Handles recieving AYT responses.

GMAgent::HandleResponseAYT

Description:

Handles recieving the Response

Key:

gm.Response.AreYouThere

Precondition:

The timer for the AYT request batch (typically 1) the original request was a part of has not expired

Postcondition:

The response from the sender is noted. If the answer is no, then this node will enter recovery, if the recovery timer has not already been set, otherwise we will allow it to expire.

ModuleMessage **Invitation**()
Creates Group Invitation Message.

GMAgent::Invitation

Description:

Creates a new invation message from the leader of this node's current leader, to join this group.

Precondition:

The node is currently in a group.

Postcondition:

No change

Return

A GroupManagementMessage with the contents of a Invitation message.

void **InviteGroupNodes**(const boost::system::error_code & *err*, PeerSet *p_tempSet*)
Sends invitations to all group members.

GMAgent::InviteGroupNodes

Description:

This function will invite all the members of a node's old group to join it's new group.

Precondition:

None

Postcondition:

Invitations have been sent to members of this node's old group. if this node is a Coordinator, this node sets a timer for Reorganize

Citation:

Group Management (Merge)

Parameters

- `err` - The error message associated with the calling thimer
- `p_tempSet` - The set of nodes that were members of the old group.

bool **`IsCoordinator`**`()` const
Returns true if this node considers itself a coordinator.

void **`Merge`**(const boost::system::error_code & *err*)
Sends invitations to all known nodes.

GMAgent::Merge

Description:

If this node is a Coordinator, this method sends invites to join this node's group to all Coordinators, and then makes a call to second function to invite all current members of this node's old group to the new group.

Precondition:

This node has waited for a Premerge.

Postcondition:

If this node was a Coordinator, this node has been placed in an election state. Additionally, invitations have been sent to all Coordinators this node is aware of. Lastly, this function has called a function or set a timer to invite its old group nodes.

Citation:

Group Management (Merge)

Parameters

- `err` - A error associated with the calling timer.

ModuleMessage **`PeerList`**(std::string *requester* = "all")
Generates a peer list.

GMAgent::PeerList

Description:

Packs the group list (Up_Nodes) in GroupManagementMessage

Precondition:

This node is a leader.

Postcondition:

No Change.

Return

A GroupManagementMessage with the contents of group membership

void **`Premerge`**(const boost::system::error_code & *err*)
Waits a time period determined by UUID for merge.

GMAgent::Premerge

Description:

Handles a proportional wait prior to calling Merge

Precondition:

Check has been called and responses have been collected from other nodes. This node is a Coordinator.

Postcondition:

A timer has been set based on this node's UUID to break up ties before merging.

Citation:

GroupManagement (Merge)

Parameters

- `err` - An error associated with the clling timer.

void **PushPeerList**()

Sends the peer list to all group members.

GMAgent::PushPeerList

Description:

Sends the membership list to other modules of this node and other nodes

Precondition:

This node is new group leader

Postcondition:

A peer list is pushed to the group members

Return

Nothing

void **Recovery**()

Resets the algorithm to the default startup state.

GMAgent::Recovery

Description:

The method used to set or reset a node into a "solo" state where it is its own leader. To do this, it forms an empty group, then enters a normal state.

Precondition:

None

Postcondition:

The node enters a NORMAL state.

Citation:

Group Management Algorithmn (Recovery).

void **Recovery**(const boost::system::error_code & *err*)

Handles no response from timeout message.

GMAgent::Recovery

Description:

Recovery function extension for handling timer expirations.

Precondition:

Some timer leading to this function has expired or been canceled.

Postcondition:

If the timer has expired, Recovery begins, if the timer was canceled and this node is NOT a Coordinator, this will cause a Timeout Check using Timeout()

Parameters

- `err` - The error code associated with the calling timer.

void **Reorganize**(const boost::system::error_code & *err*)
Puts the system into the working state.

GMAgent::Reorganize

Description:

Organizes the members of the group and prepares them to do their much needed work!

Precondition:

The node is a leader of group.

Postcondition:

The group has received work assignments, ready messages have been sent out, and this node enters the NORMAL state.

Citation:

Group Management Reorganize

void **SetStatus**(int *status*)
Sets the status of the node.

Description:

Sets the internal status variable, m_status based on the paramter status. See GetStatus for tips on establishing the status code numbers.

Precondition:

None

Postcondition:

The modules status code has been set to "status"

Parameters

- `status` - The status code to set

void **StartMonitor**(const boost::system::error_code & *err*)
Start the monitor after transient is over.

void **SystemState**()
Outputs information about the current state to the logger.

GMAgent::SystemState

Description:

Puts the system state to the logger.

Precondition:

None

Postcondition:

None

void **Timeout** (const boost::system::error_code & *err*)
Checks that the leader is still alive and working.

GMAgent::Timeout

Description:

Sends an AreYouThere message to the coordinator and sets a timer if the timer expires, this node goes into recovery.

Precondition:

The node is a member of a group, but not the leader

Postcondition:

A timer for recovery is set.

Citation:

Group Managment Timeout

Private Members

boost::posix_time::time_duration **AYC_RESPONSE_TIMEOUT**
How long to wait for responses from other nodes.

boost::posix_time::time_duration **AYT_RESPONSE_TIMEOUT**
How long to wait for responses from other nodes.

boost::posix_time::time_duration **CHECK_TIMEOUT**
How long between AYC checks.

boost::posix_time::time_duration **FID_TIMEOUT**
How long to wait before checking attached FIDs.

boost::posix_time::time_duration **INVITE_RESPONSE_TIMEOUT**
How long to wait for responses from other nodes.

TimedPeerSet **m_AYCResponse**
Nodes expecting AYC response from.

TimedPeerSet **m_AYTResponse**
Nodes expecting AYT response from.

PeerSet **m_Coordinators**
Known Coordinators.

std::map< std::string, bool > **m_fidstate**
A store for the state of attached FIDs.

CBroker::TimerHandle **m_fidtimer**
Timer for checking FIDs.

google::protobuf::uint32 **m_GroupID**
The ID number of the current group (Never initialized for fun)

std::string **m_GroupLeader**
The uuid of the group leader.

int **m_groupsbroken**
Number of groups broken.

int **m_groupselection**
Number of elections started.

int **m_groupsformed**
Number of groups formed.

int **m_groupsjoined**
Number of accepts sent.

unsigned int **m_GrpCounter**
The number of groups being formed.

boost::asio::io_service **m_localservice**
The io_service used.

int **m_membership**
Total size of groups after all checks.

int **m_membershipchecks**
Number of membership checks.

int **m_status**
A store for the status of this node.

CBroker::TimerHandle **m_timer**
A timer for stepping through the election process.

PeerSet **m_UpNodes**
Nodes In My Group.

boost::posix_time::time_duration **TIMEOUT_TIMEOUT**
How long beteween AYT checks.

Private Static Functions

ModuleMessage **PeerListQuery** (std::string *requester*)
Generates a CMessage that can be used to query for the group.

GMAgent::PeerListQuery

Description:

Generates a GroupManagementMessage that can be used to query the peerlist of a node.

Precondition:

: None

Postcondition:

: No change

Return

A GroupManagementMessage which can be used to query for

Parameters

- `requester` - The module who the response should be addressed to.

ModuleMessage **`PrepareForSending`**(const GroupManagementMessage & *message*, std::string *recipient* = "gm")

Wraps a GroupManagementMessage in a ModuleMessage.

Wraps a GroupManagementMessage in a ModuleMessage.

Return

a ModuleMessage containing a copy of the GroupManagementMessage

Parameters

- `message` - the message to prepare. If any required field is unset, the DGI will abort.
- `recipient` - the module (sc/lb/gm/clk etc.) the message should be delivered to

7.3 Load Balancing Module

7.3.1 Algorithm

Give an overview of how the module works and link to the paper

7.3.2 Invariant Checking

Which invariant is implemented. Which models can you use with that invariant? How do you get the models?

Document the malicious flag Document the invariant flag.

7.3.3 Migration Size

Document the migration size flag.

CHAPTER 8

DGI Framework Reference

8.1 CBroker Reference

CBroker is a part of *CBroker.hpp*

class `freedm::broker::CBroker`
Scheduler for the DGI modules.

Public Functions

~CBroker()
De-allocates the timers when the *CBroker* is destroyed.

Description:

Cleans up all the timers from this module since the timers are stored as pointers.

Precondition:

None

Postcondition:

All the timers are destroyed and their handles no longer point at valid resources.

TimerHandle **AllocateTimer**(ModuleIdent *module*)
Allocate a timer to a specified module.

Description:

Returns a handle to a timer to use for scheduling tasks. Timer handles are used in *Schedule()* calls.

Precondition:

The module is registered

Postcondition:

A handle to a timer is returned.

Return

A CBroker::TimerHandle that can be used to schedule tasks.

Parameters

- `module` - the module the timer should be allocated to

CClockSynchronizer & **`GetClockSynchronizer`**()
Returns the synchronizer.

Description:

Returns a reference to the ClockSynchronizer object.

Precondition:

None

Postcondition:

Any changes to the ClockSynchronizer will affect the object owned by the broker.

Return

A reference to the Broker's ClockSynchronizer object.

boost::asio::io_service & **`GetIOService`**()
Return a reference to the boost::ioservice.

Description:

returns a refernce to the ioservice used by the broker.

Return

The ioservice used by this broker.

void **`HandleSignal`**(const boost::system::error_code & *error*, int *parameter*)
Handle signals from the operating system (ie, Control-C)

Description:

Handle signals that terminate the program.

Precondition:

None

Postcondition:

The broker is scheduled to be stopped.

void **`HandleStop`**(unsigned int *signum* = 0)
Handles the stop signal from the operating System.

Description:

Handles closing all the sockets connection managers and Services. Should probably only be called by *CBroker::Stop()*.

Precondition:

The ioservice is running but all agents have been stopped.

Postcondition:

The Broker has been cleanly shut down.

Postcondition:

The devices subsystem has been cleanly shut down.

Parameters

- `signum` - positive if called from a signal handler, or 0 otherwise

bool **IsModuleRegistered**(ModuleIdent *m*)
Checks to see if a module is registered with the scheduler.

Description:

Checks to see if a module is registered with the scheduler.

Precondition:

None

Postcondition:

None

Return

true if the module is registered with the broker.

Parameters

- `m` - the identifier for the module.

void **RegisterModule**(ModuleIdent *m*, boost::posix_time::time_duration *phase*)
Registers a module for the scheduler.

Description:

Places the module in to the list of schedulable phases. The scheduler cycles through registered modules to do real-time round robin scheduling.

Precondition:

None

Postcondition:

The module is registered with a phase duration specified by the phase parameter.

Parameters

- `m` - the identifier for the module.
- `phase` - the duration of the phase.

void **Run**()
Starts the DGI Broker scheduler.

Description:

Starts the adapter factory. Runs the ioservice until it is out of work. Runs the clock synchronizer.

Precondition:

The ioservice has some schedule of jobs waiting to be performed (so it doesn't exit immediately).

Postcondition:

The ioservice has stopped.

Error Handling:

Could raise arbitrary exceptions from anywhere in the DGI.

int **Schedule**(TimerHandle *h*, boost::posix_time::time_duration *wait*, Scheduleable *x*)
Schedules a task that will run after a timer expires.

Description:

Given a scheduleable task that should be run in the future. The task will be scheduled to run by the Broker after the timer expires and during the module that owns the timer's phase. The attempt to schedule may be rejected if the Broker is stopping, indicated by the return value.

Precondition:

The module is registered

Postcondition:

If the Broker is not stopping, the function is scheduled to be called in the future. If a next time function is scheduled, its timer will expire as soon as its round ends. If the Broker is stopping the task will not be scheduled.

Return

0 on success, -1 if rejected

Parameters

- `h` - The handle to the timer being set.
- `wait` - the amount of the time to wait. If this value is "not_a_date_time" The wait is converted to positive infinity and the time will expire as soon as it is not the module that owns the timer's phase.
- `x` - A schedulable, a functor, that expects a single boost::system::error_code parameter and returns void, created via boost::bind()

int **Schedule**(ModuleIdent *m*, BoundScheduleable *x*, bool *start_worker* = true)
Schedule a task to be run as soon as the module is active.

Description:

Given a module and a task, put that task into that modules job queue. The attempt to schedule will be rejected if the Broker is stopping.

Precondition:

The module is registered.

Postcondition:

The task is placed in the work queue for the module m. If the start_worker parameter is set to true, the module's worker will be activated if it isn't already.

Return

0 on success, -1 if rejected because the Broker is stopping.

Parameters

- `m` - The module the schedulable should be run as.
- `x` - The method that will be run. A functor that expects no parameters and returns void. Created via boost::bind()
- `start_worker` - tells the worker to begin processing again, if it is currently idle. The worker may be idle if the work queue is currently empty

void **Stop**(unsigned int *signum* = 0)
Requests that the Broker stops execution to exit the DGI.

Description:

Registers a stop command into the io_service's job queue. when scheduled, the stop operation will terminate all running modules and cause the ioservice.run() call to exit.

Precondition:

The ioservice is running and processing tasks.

Postcondition:

The command to stop the ioservice has been placed in the service's task queue.

Parameters

- `signum` - A signal identifier if the call came from a signal, or 0 otherwise

boost::posix_time::time_duration **TimeRemaining**()
Returns how much time the current module has left in its phase.

Description:

Returns how much time is remaining in the active module's phase. The result can be negative if the module has exceeded its allotted execution time.

Precondition:

The Change Phase function has been called at least once. This should have occured by the time the first module is ready to look at the remaining time.

Postcondition:

None

Return

A time_duration describing the amount of time remaining in the phase.

Public Static Functions

CBroker & **Instance**()
Get the singleton instance of this class.

Access the singleton Broker instance.

8.1.1 See Also

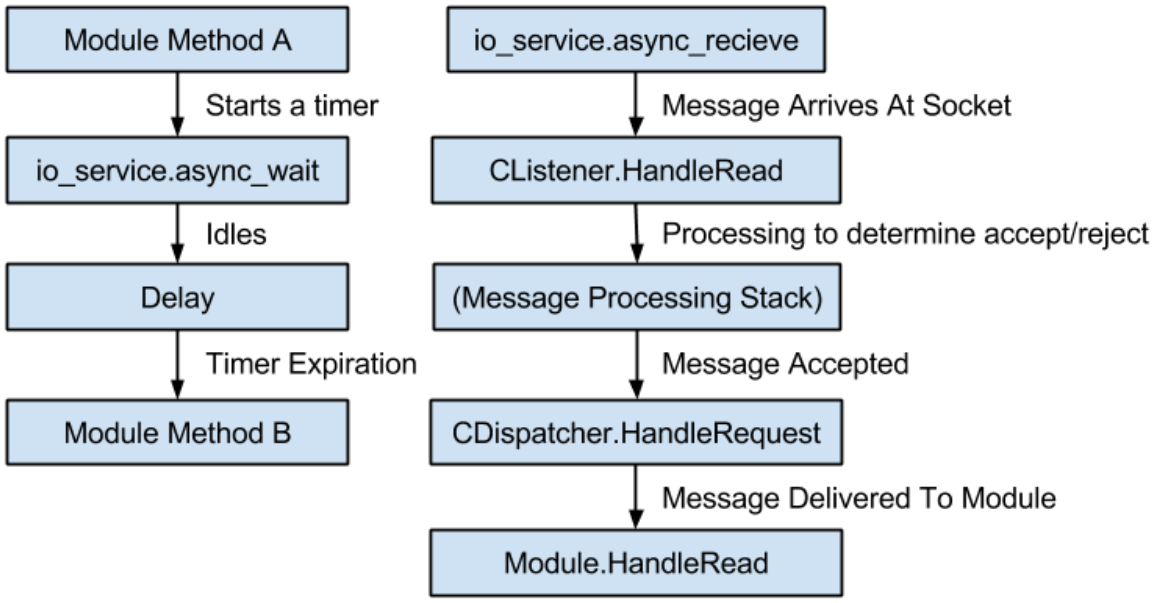
It may be useful to start with *Scheduling DGI Modules*.

8.1.2 The Scheduler

The non-RT DGI code made heavy use of the boost provided io_service. This library allows for easy asynchronous calls to methods and very easy socket handling. Tasks are inserted into the io_services run list and executed in FIFO order. The RT scheduler takes advantage of the io_service, while trying to apply constraints on the ordering of execution.

Review: Non-RT Processing

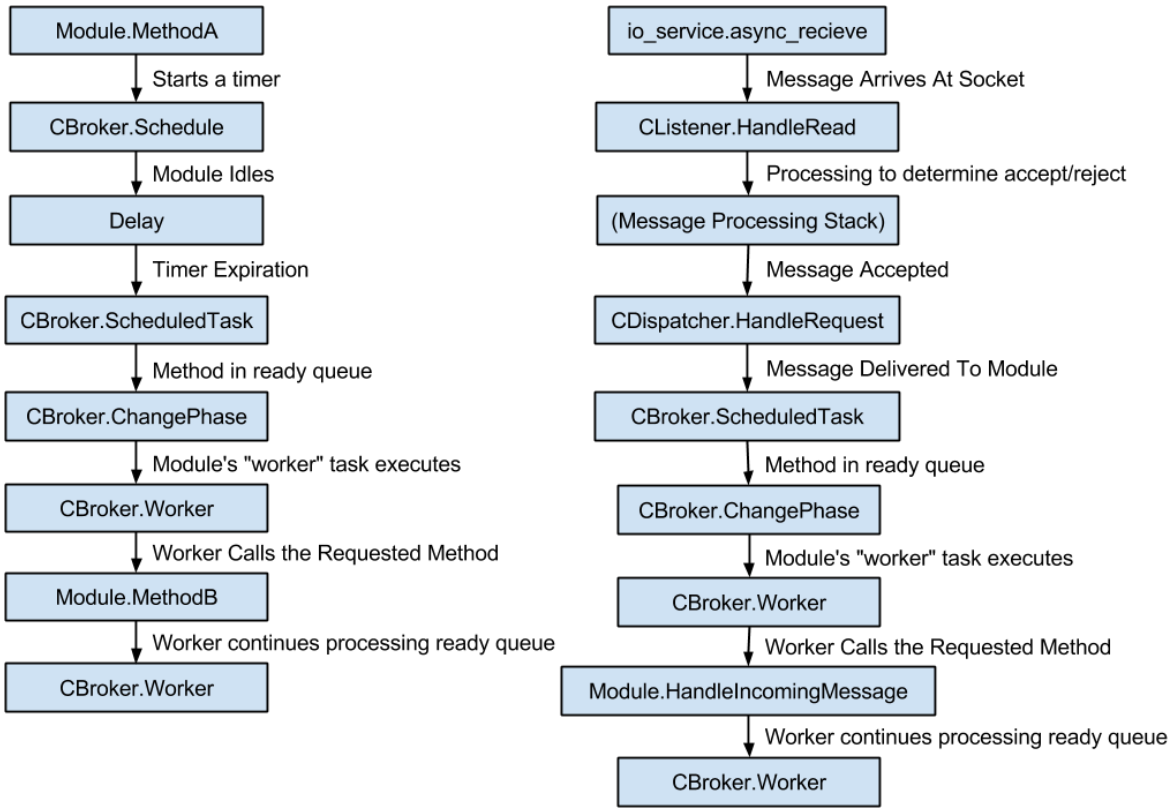
To review, the modules and message processing (which are the two major components of the system) relied on a structure very similar to this:



- **Sleeps and Thread Relinquishment** - Since most of the processing for the FREEDM system currently runs on the single io_service thread used by the broker, multi-threading is emulated through the use of the io_service's async wait routines which allow a module to sleep while other modules (or the broker itself) continues doing processing on a single thread. When a task requires a break for I/O or other similar services, it will call async_wait and wait for the io_service to execute the callback it requests. There are many examples of this in the codebase.
- **Message Processing** - When a message is received for processing, it will be taken into the CListener module where it is processed to determine if it should be accepted or rejected. If accepted, it is passed to the dispatcher which examines the contents to determine which modules if any should be given the message. The module responsible for receiving the message is immediately called and allowed to act on the message.

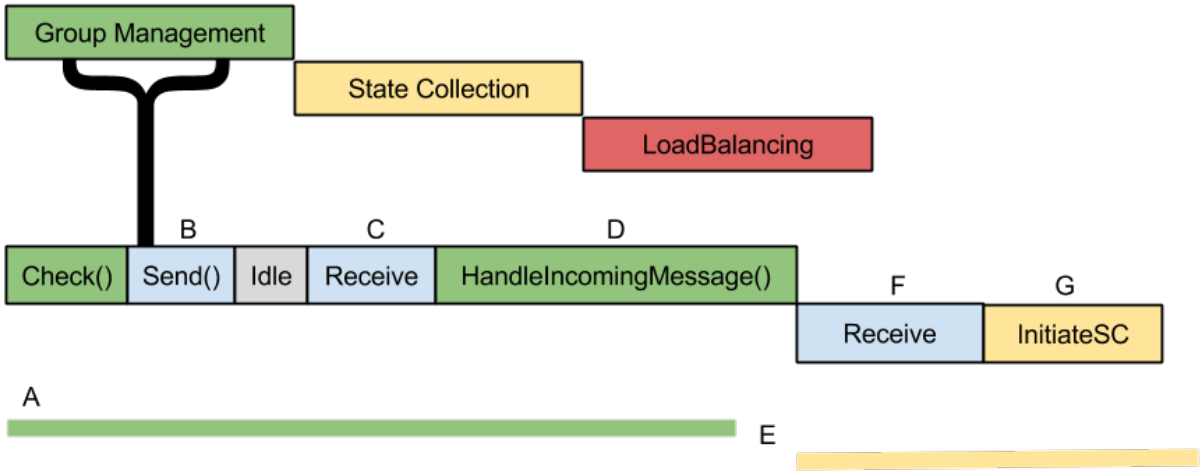
Real-time Processing

To create a real-time scheduler, I've added a layer between the modules and the io_service in the broker. This layer defers the expired async_waits until it is a modules "phase." To do this, the broker now distributes timers (or more specifically, handles to the timers) and provides a function "Schedule" which is used instead of async_wait. The result is a scheduling system like this:



- **Sleeps and Thread Relinquishment** - The behavior remains similar to how it did before, except the async_wait call is replaced by a schedule call. When the timer expires, it calls the ScheduledTask method, rather than directly calling the scheduled task. This method enters the readied task into a per-module "ready queue." Once the scheduler enters that modules phase (or it is already in it) a worker method will process the ready queue in FIFO order, calling the scheduled methods.
- **Message Processing** - This also remains similar, except now instead of the dispatcher directly calling the module who should receive the incoming message the call to the module is placed in that module's ready queue. Then, as before, when the scheduler enters that module's phase, a worker method handles the actual call to the receive method.

Phase Behavior



Phases proceed in a round robin fashion. However, there are some observations to make a for this implementation. Consider the diagram above.

- **A** The group management phase begins. It spends some time authoring messages for the check function, requests them to be sent and then idles. Send, which is handled by the broker operates outside of the module scheduler and begins work immediately.
- **B** Since group management has no work to do while waiting for replies, the system is idle.
- **C** Message(s) arrive. Since one of them is addressed to group management and it is currently GroupManagement's phase, the worker immediately fulfills the request.
- **D** Processing the message makes GroupManagement over-run its phase.
- **E** There is a "No-man's" land where the group management task is completing work outside of its phase, but the scheduler is ready to switch phases to state collection. Because phases are aligned in order to form groups, state collection is penalized for by the overflow.
- **F** The scheduler has changed phases, but the broker does work on a received message before calling state collection's readied method. Note that the message is put into the ready queue for its intended process, so the message processing time is only spent by the Broker and communication stack
- **G** Finally, the state collection module begins.

8.2 Using The DGI Logger

Note: The DGI is a real-time system and its correctness depends on its performance. However, the amount of data generated at high verbosity reduces the performance of the system. Were it to be run at full verbosity, it would not operate correctly. We generally recommend running the DGI with a global verbosity level of 4 (Status) or 5 (Notice).

The DGI provides nine levels of logging

- 0 - Fatal
- 1 - Alert
- 2 - Error

- 3 - Warn
- 4 - Status
- 5 - Notice
- 6 - Info
- 7 - Debug
- 8 - Trace

There are three methods for adjusting the logging level:

- Set verbosity=n in freedm.cfg to specify the global verbosity
- Call PosixBroker with –verbose=n to override the global verbosity specified in freedm.cfg
- Override the global verbosity for a particular source file in logger.cfg

As greater verbosity levels are desired, a sacrifice in performance must be made (by increasing the timings as explained in *Configuring Timings*.

Sometimes you will want more data from a particular source file or module. In this case you should utilize logger.cfg. For example, you may want to run Group Management at greater verbosity in order to pick up AYT and AYC response delays, which are printed at level 6 (Info).

Editing logger.cfg

The logger.cfg in the sample folder looks like this:

```
# Example of how to set a file's verbosity:
#
# StateCollection.cpp=2
#
# Note that the path to the file is not included.
# Any files not added here are set to the verbosity specified in freedm.cfg
#
# Valid Verbosity Levels:
# 8 - Trace
# 7 - Debug
# 6 - Info
# 5 - Notice
# 4 - Status
# 3 - Warn
# 2 - Error
# 1 - Alert
# 0 - Fatal
#
# If a file does not have a logger, you will receive an "Unknown Option" error!!
```

The location of logger config is specified in `freedm.cfg` the default value is set to `./config/logger.cfg`.

Logger levels are specified by setting a specific logger to a specific level. Most files in DGI have their own logger dedicated for their output. Therefore, most `.cpp` files have a logger attached to them that can be used for output. Each module has its own logger.

- `StateCollection.cpp` - The logger for State Collection
- `GroupManagement.cpp` - The logger for Group Management
- `LoadBalance.cpp` - The logger for load balancing.

To set a logger for a file, add a line that looks like:

```
StateCollection.cpp=2
```

8.2.1 Archiving DGI Runs

The DGI generates a significant amount of data when it is run. This data is saved by DGIProcess, Distributed Timestamp, and Event into a cloud archiver, one file for each DGI. By default the data is sent to Unix stderr as the DGI runs. To redirect the data into the cloud archiver over the communications network, redirect stderr when starting the DGI. The following command runs the DGI and redirects stderr to stdout and pipes it to tee, which will save the data to a file named cloud.DGIx while also printing it to the screen:

```
`./PosixBroker 2>&1 | tee cloud.DGIx`
```

where DGIx is the DGI process on a particular SST, x.

The data is stored in the cloud in a text format database, allowing it to be processed by a wide range of existing text processing facilities. We expect the Unix "grep" command to be extremely useful for querying the database given its advanced regular expression capabilities and adjustable levels of output context. If you're interested in grabbing AYT response delays, grep cloud.DGIx for "AYT response received" and then format the result to your liking with gawk or copy it directly into a spreadsheet. As there is really no limit to the data processing capabilities already available for text-based data, you can use whatever tools you want to do whatever you need to do.

CHAPTER 9

Other

- Doxygen Documentation
- *search*

F

I

Y