OPFLEX

Release master

OpenDaylight Project

Apr 29, 2020

CONTENTS

1 OpFlex agent-ovs Developer Guide 1

2 OpFlex agent-ovs User Guide 3

3 OpFlex genie Developer Guide 11

4 OpFlex libopflex Developer Guide 13

CHAPTER

ONE

OPFLEX AGENT-OVS DEVELOPER GUIDE

1.1 Overview

agent-ovs is a policy agent that works with OVS to enforce a group-based policy networking model with locally attached virtual machines or containers. The policy agent is designed to work well with orchestration tools like OpenStack.

1.2 agent-ovs Architecture

agent-ovs uses libopflex to communicate with an OpFlex-based policy repository to enforce policy on network endpoints attached to OVS by an orchestration system.

The key components are:

- Agent - coordinates startup and configuration
- Renderers - Renderers are responsible for rendering policy. This is a very general mechanism but the currently-implemented renderer is the stitched-mode renderer that can work along with with hardware fabrics such as ACI that support policy enforcement.
- EndpointManager - Keep track of network endpoints and declare them to the endpoint repository
- PolicyManager - Keep track of and index policies
- IntFlowManager - render policies to OVS integration bridge
- AccessFlowManager - render policies to OVS access bridge

1.3 API Reference Documentation

Internal API documentation can be found by in doc/html/index.html in any build.

CHAPTER

TWO

OPFLEX AGENT-OVS USER GUIDE

2.1 Introduction

agent-ovs is a policy agent that works with OVS to enforce a group-based policy networking model with locally attached virtual machines or containers. The policy agent is designed to work well with orchestration tools like OpenStack.

2.2 Agent Configuration

The agent configuration is handled using its config file which is by default found at "/etc/opflex-agent-ovs/opflex-agent-ovs.conf"

Here is an example configuration file that documents the available options:

```
{
    // Logging configuration
    // "log": {
    //     // Set the log level.
    //     // Possible values in descending order of verbosity:
    //     // "debug5"-"debug0", "debug" (synonym for "debug0"),
    //     // "info", "warning", "error", "fatal"
    //     // Default: "info"
    //     "level": "info"
    // },

    // Configuration related to the OpFlex protocol
    "opflex": {
        // The policy domain for this agent.
        "domain": "openstack",

        // The unique name in the policy domain for this agent.
        "name": "example-agent",

        // a list of peers to connect to, by hostname and port.  One
        // peer, or an anycast pseudo-peer, is sufficient to bootstrap
        // the connection without needing an exhaustive list of all
        // peers.
        "peers": [
            // EXAMPLE:
            // {"hostname": "10.0.0.30", "port": 8009}
        ],
```

(continues on next page)

(continued from previous page)

```
        "ssl": {
            // SSL mode.  Possible values:
            // disabled: communicate without encryption (default)
            // encrypted: encrypt but do not verify peers
            // secure: encrypt and verify peer certificates
            "mode": "encrypted",

            // The path to a directory containing trusted certificate
            // authority public certificates, or a file containing a
            // specific CA certificate.
            // Default: "/etc/ssl/certs"
            "ca-store": "/etc/ssl/certs"
        },

        "inspector": {
            // Enable the MODB inspector service, which allows
            // inspecting the state of the managed object database.
            // Default: true
            "enabled": true,

            // Listen on the specified socket for the inspector
            // Default: "/var/run/opflex-agent-ovs-inspect.sock"
            "socket-name": "/var/run/opflex-agent-ovs-inspect.sock"
        },

        "notif": {
            // Enable the agent notification service, which sends
            // notifications to interested listeners over a UNIX
            // socket.
            // Default: true
            "enabled": true,

            // Listen on the specified socket for the inspector
            // Default: "/var/run/opflex-agent-ovs-notif.sock"
            "socket-name": "/var/run/opflex-agent-ovs-notif.sock",

            // Set the socket owner user after binding if the user
            // exists
            // Default: do not set the owner
            // "socket-owner": "root",

            // Set the socket group after binding if the group name
            // exists
            // Default: do not set the group
            "socket-group": "opflexep",

            // Set the socket permissions after binding to the
            // specified octal permissions mask
            // Default: do not set the permissions
            "socket-permissions": "770"
        }
    },

    // Endpoint sources provide metadata about local endpoints
    "endpoint-sources": {
        // Filesystem path to monitor for endpoint information
        // Default: no endpoint sources
```

(continues on next page)

(continued from previous page)

```
        "filesystem": ["/var/lib/opflex-agent-ovs/endpoints"]
    },

    // Service sources provide metadata about services that can
    // provide functionality for local endpoints
    "service-sources": {
        // Filesystem path to monitor for service information
        // Default: no service sources
        "filesystem": ["/var/lib/opflex-agent-ovs/services"]
    },

    // Renderers enforce policy obtained via OpFlex.
    // Default: no renderers
    "renderers": {
        // Stitched-mode renderer for interoperating with a
        // hardware fabric such as ACI
        // EXAMPLE:
        "stitched-mode": {
            // "Integration" bridge used to enforce contracts and forward
            // packets
            "int-bridge-name": "br-int",

            // "Access" bridge used to enforce access control and enforce
            // security groups.
            "access-bridge-name": "br-access",

            // Set encapsulation type.  Must set either vxlan or vlan.
            "encap": {
                // Encapsulate traffic with VXLAN.
                "vxlan" : {
                    // The name of the tunnel interface in OVS
                    "encap-iface": "br0_vxlan0",

                    // The name of the interface whose IP should be used
                    // as the source IP in encapsulated traffic.
                    "uplink-iface": "team0.4093",

                    // The vlan tag, if any, used on the uplink interface.
                    // Set to zero or omit if the uplink is untagged.
                    "uplink-vlan": 4093,

                    // The IP address used for the destination IP in
                    // the encapsulated traffic.  This should be an
                    // anycast IP address understood by the upstream
                    // stiched-mode fabric.
                    "remote-ip": "10.0.0.32",

                    // UDP port number of the encapsulated traffic.
                    "remote-port": 8472
                }

                // Encapsulate traffic with a locally-significant VLAN
                // tag
                // EXAMPLE:
                // "vlan" : {
                //     // The name of the uplink interface in OVS
                //     "encap-iface": "team0"
```

(continues on next page)

(continued from previous page)

```
            // }
        },

        // Configure forwarding policy
        "forwarding": {
            // Configure the virtual distributed router
            "virtual-router": {
                // Enable virtual distributed router.  Set to true
                // to enable or false to disable.
                // Default: true.
                "enabled": true,

                // Override MAC address for virtual router.
                // Default: "00:22:bd:f8:19:ff"
                "mac": "00:22:bd:f8:19:ff",

                // Configure IPv6-related settings for the virtual
                // router
                "ipv6" : {
                    // Send router advertisement messages in
                    // response to router solicitation requests as
                    // well as unsolicited advertisements.  This
                    // is not required in stitched mode since the
                    // hardware router will send them.
                    "router-advertisement": false
                }
            },

            // Configure virtual distributed DHCP server
            "virtual-dhcp": {
                // Enable virtual distributed DHCP server.  Set to
                // true to enable or false to disable.
                // Default: true
                "enabled": true,

                // Override MAC address for virtual dhcp server.
                // Default: "00:22:bd:f8:19:ff"
                "mac": "00:22:bd:f8:19:ff"
            },

            "endpoint-advertisements": {
                // Set mode for generation of periodic ARP/NDP
                // advertisements for endpoints.  Possible values:
                // disabled: Do not send advertisements
                // gratuitous-unicast: Send gratuitous endpoint
                //   advertisements as unicast packets to the router
                //   mac.
                // gratuitous-broadcast: Send gratuitous endpoint
                //   advertisements as broadcast packets.
                // router-request: Unicast a spoofed request/solicitation
                //   for the subnet's gateway router.
                // Default: router-request.
                "mode": "gratuitous-broadcast"
            }
        },

        // Location to store cached IDs for managing flow state
```

(continues on next page)

(continued from previous page)

```
            // Default: "/var/lib/opflex-agent-ovs/ids"
            "flowid-cache-dir": "/var/lib/opflex-agent-ovs/ids",

            // Location to write multicast groups for the mcast-daemon
            // Default: "/var/lib/opflex-agent-ovs/mcast/opflex-groups.json"
            "mcast-group-file": "/var/lib/opflex-agent-ovs/mcast/opflex-groups.json"
        }
    }
}
```

2.3 Endpoint Registration

The agent learns about endpoints using endpoint metadata files located by default in "/var/lib/opflex-agent-ovs/endpoints".

These are JSON-format files such as the (unusually complex) example below:

```
{
    "uuid": "83f18f0b-80f7-46e2-b06c-4d9487b0c754",
    "policy-space-name": "test",
    "endpoint-group-name": "group1",
    "interface-name": "veth0",
    "ip": [
        "10.0.0.1", "fd8f:69d8:c12c:ca62::1"
    ],
    "dhcp4": {
        "ip": "10.200.44.2",
        "prefix-len": 24,
        "routers": ["10.200.44.1"],
        "dns-servers": ["8.8.8.8", "8.8.4.4"],
        "domain": "example.com",
        "static-routes": [
            {
                "dest": "169.254.169.0",
                "dest-prefix": 24,
                "next-hop": "10.0.0.1"
            }
        ]
    },
    "dhcp6": {
        "dns-servers": ["2001:4860:4860::8888", "2001:4860:4860::8844"],
        "search-list": ["test1.example.com", "example.com"]
    },
    "ip-address-mapping": [
        {
           "uuid": "91c5b217-d244-432c-922d-533c6c036ab4",
           "floating-ip": "5.5.5.1",
           "mapped-ip": "10.0.0.1",
           "policy-space-name": "common",
           "endpoint-group-name": "nat-epg"
        },
        {
           "uuid": "22bfdc01-a390-4b6f-9b10-624d4ccb957b",
           "floating-ip": "fdf1:9f86:d1af:6cc9::1",
```

(continues on next page)

(continued from previous page)

```
            "mapped-ip": "fd8f:69d8:c12c:ca62::1",
            "policy-space-name": "common",
            "endpoint-group-name": "nat-epg"
        }
    ],
    "mac": "00:00:00:00:00:01",
    "promiscuous-mode": false
}
```

The possible parameters for these files are:

uuid A globally unique ID for the endpoint

endpoint-group-name The name of the endpoint group for the endpoint

policy-space-name The name of the policy space for the endpoint group.

interface-name The name of the OVS interface to which the endpoint is attached

ip A list of strings contains either IPv4 or IPv6 addresses that the endpoint is allowed to use

mac The MAC address for the endpoint's interface.

promiscuous-mode Allow traffic from this VM to bypass default port security

dhcp4 A distributed DHCPv4 configuration block (see below)

dhcp6 A distributed DHCPv6 configuration block (see below)

ip-address-mapping A list of IP address mapping configuration blocks (see below)

DHCPv4 configuration blocks can contain the following parameters:

ip the IP address to return with DHCP. Must be one of the configured IPv4 addresses.

prefix the subnet prefix length

routers a list of default gateways for the endpoint

dns a list of DNS server addresses

domain The domain name parameter to send in the DHCP reply

static-routes A list of static route configuration blocks, which contains a "dest", "dest-prefix", and "next-hop" parameters to send as static routes to the end host

DHCPv6 configuration blocks can contain the following parameters:

dns A list of DNS servers for the endpoint

search-patch The DNS search path for the endpoint

IP address mapping configuration blocks can contain the following parameters:

uuid a globally unique ID for the virtual endpoint created by the mapping.

floating-ip Map using DNAT to this floating IPv4 or IPv6 address

mapped-ip the source IPv4 or IPv6 address; must be one of the IPs assigned to the endpoint.

endpoint-group-name The name of the endpoint group for the NATed IP

policy-space-name The name of the policy space for the NATed IP

2.4 Inspector

The Opflex inspector is a useful command-line tool that will allow you to inspect the state of the managed object database for the agent for debugging and diagnosis purposes.

The command is called "gbp_inspect" and takes the following arguments:

```
# gbp_inspect -h
Usage: gbp_inspect [options]
Allowed options:
  -h [ --help ]                         Print this help message
  --log arg                             Log to the specified file (default
                                        standard out)
  --level arg (=warning)                Use the specified log level (default
                                        warning)
  --syslog                              Log to syslog instead of file or
                                        standard out
  --socket arg (=/usr/var/run/opflex-agent-ovs-inspect.sock)
                                        Connect to the specified UNIX domain
                                        socket (default /usr/var/run/opfl
                                        ex-agent-ovs-inspect.sock)
  -q [ --query ] arg                    Query for a specific object with
                                        subjectname,uri or all objects of a
                                        specific type with subjectname
  -r [ --recursive ]                    Retrieve the whole subtree for each
                                        returned object
  -f [ --follow-refs ]                  Follow references in returned objects
  --load arg                            Load managed objects from the specified
                                        file into the MODB view
  -o [ --output ] arg                   Output the results to the specified
                                        file (default standard out)
  -t [ --type ] arg (=tree)             Specify the output format: tree,
                                        asciitree, list, or dump (default tree)
  -p [ --props ]                        Include object properties in output
```

Here are some examples of the ways to use this tool.

You can get information about the running system using one or more queries, which consist of an object model class name and optionally the URI of a specific object. The simplest query is to get a single object, nonrecursively:

```
# gbp_inspect -q DmtreeRoot
—— DmtreeRoot,/
# gbp_inspect -q GbpEpGroup
—— GbpEpGroup,/PolicyUniverse/PolicySpace/test/GbpEpGroup/group1/
—— GbpEpGroup,/PolicyUniverse/PolicySpace/common/GbpEpGroup/nat-epg/
# gbp_inspect -q GbpEpGroup,/PolicyUniverse/PolicySpace/common/GbpEpGroup/nat-epg/
—— GbpEpGroup,/PolicyUniverse/PolicySpace/common/GbpEpGroup/nat-epg/
```

You can also display all the properties for each object:

```
# gbp_inspect -p -q GbpeL24Classifier
—— GbpeL24Classifier,/PolicyUniverse/PolicySpace/test/GbpeL24Classifier/classifier4/
      {
        connectionTracking : 1 (reflexive)
        dFromPort          : 80
        dToPort            : 80
        etherT             : 2048 (ipv4)
        name               : classifier4
```

(continues on next page)

(continued from previous page)

```
        prot               : 6
      }
—— GbpeL24Classifier,/PolicyUniverse/PolicySpace/test/GbpeL24Classifier/classifier3/
      {
        etherT : 34525 (ipv6)
        name   : classifier3
        order  : 100
        prot   : 58
      }
—— GbpeL24Classifier,/PolicyUniverse/PolicySpace/test/GbpeL24Classifier/classifier1/
      {
        etherT : 2054 (arp)
        name   : classifier1
        order  : 102
      }
—— GbpeL24Classifier,/PolicyUniverse/PolicySpace/test/GbpeL24Classifier/classifier2/
      {
        etherT : 2048 (ipv4)
        name   : classifier2
        order  : 101
        prot   : 1
      }
```

You can also request to get the all the children of an object you query for:

```
# gbp_inspect -r -q GbpEpGroup,/PolicyUniverse/PolicySpace/common/GbpEpGroup/nat-epg/
—— GbpEpGroup,/PolicyUniverse/PolicySpace/common/GbpEpGroup/nat-epg/
  ├—— GbpeInstContext,/PolicyUniverse/PolicySpace/common/GbpEpGroup/nat-epg/
↪GbpeInstContext/
  —— GbpEpGroupToNetworkRSrc,/PolicyUniverse/PolicySpace/common/GbpEpGroup/nat-epg/
↪GbpEpGroupToNetworkRSrc/
```

You can also follow references found in any object downloads:

```
# gbp_inspect -fr -q GbpEpGroup,/PolicyUniverse/PolicySpace/common/GbpEpGroup/nat-epg/
—— GbpEpGroup,/PolicyUniverse/PolicySpace/common/GbpEpGroup/nat-epg/
  ├—— GbpeInstContext,/PolicyUniverse/PolicySpace/common/GbpEpGroup/nat-epg/
↪GbpeInstContext/
  —— GbpEpGroupToNetworkRSrc,/PolicyUniverse/PolicySpace/common/GbpEpGroup/nat-epg/
↪GbpEpGroupToNetworkRSrc/
—— GbpBridgeDomain,/PolicyUniverse/PolicySpace/common/GbpBridgeDomain/bd_ext/
  —— GbpBridgeDomainToNetworkRSrc,/PolicyUniverse/PolicySpace/common/GbpBridgeDomain/
↪bd_ext/GbpBridgeDomainToNetworkRSrc/
—— GbpFloodDomain,/PolicyUniverse/PolicySpace/common/GbpFloodDomain/fd_ext/
  —— GbpFloodDomainToNetworkRSrc,/PolicyUniverse/PolicySpace/common/GbpFloodDomain/fd_
↪ext/GbpFloodDomainToNetworkRSrc/
—— GbpRoutingDomain,/PolicyUniverse/PolicySpace/common/GbpRoutingDomain/rd_ext/
  ├—— GbpRoutingDomainToIntSubnetsRSrc,/PolicyUniverse/PolicySpace/common/
↪GbpRoutingDomain/rd_ext/GbpRoutingDomainToIntSubnetsRSrc/152/%2fPolicyUniverse
↪%2fPolicySpace%2fcommon%2fGbpSubnets%2fsubnets_ext%2f/
  —— GbpForwardingBehavioralGroupToSubnetsRSrc,/PolicyUniverse/PolicySpace/common/
↪GbpRoutingDomain/rd_ext/GbpForwardingBehavioralGroupToSubnetsRSrc/
—— GbpSubnets,/PolicyUniverse/PolicySpace/common/GbpSubnets/subnets_ext/
  ├—— GbpSubnet,/PolicyUniverse/PolicySpace/common/GbpSubnets/subnets_ext/GbpSubnet/
↪subnet_ext4/
  —— GbpSubnet,/PolicyUniverse/PolicySpace/common/GbpSubnets/subnets_ext/GbpSubnet/
↪subnet_ext6/
```

CHAPTER

THREE

OPFLEX GENIE DEVELOPER GUIDE

3.1 Overview

Genie is a tool for code generation from a model. It supports generating C++ and Java code. C++ can be generated suitable for use with libopflex. C++ and Java can be generated as a plain set of objects.

3.2 Group-based Policy Model

The group-based policy model is included with the genie tool and can be found under the MODEL directory. By running mvn exec:java, libmodelgbp will be generated as a library project that, when built and installed, will work with libopflex. This model is used by the OVS agent.

3.3 API Reference Documentation

Complete API documentation for the generated libmodelgbp can be found in doc/html/index.html in any build

CHAPTER

FOUR

OPFLEX LIBOPFLEX DEVELOPER GUIDE

4.1 Overview

The OpFlex framework allows you to develop agents that can communicate using the OpFlex protocol and act as a policy element in an OpFlex-based distributed control system. The OpFlex architecture provides a distributed control system based on a declarative policy information model. The policies are defined at a logically centralized policy repository and enforced within a set of distributed policy elements. The policy repository communicates with the subordinate policy elements using the OpFlex control protocol. This protocol allows for bidirectional communication of policy, events, statistics, and faults.

Rather than simply providing access to the OpFlex protocol, this framework allows you to directly manipulate a management information tree containing a hierarchy of managed objects. This tree is kept in sync as needed with other policy elements in the system, and you are automatically notified when important changes to the model occur. Additionally, we can ensure that only those managed objects that are important to the local policy element are synchronized locally.

4.1.1 Object Model

Interactions with the OpFlex framework happen through the management information tree. This is a tree of managed objects defined by an object model specific to your application. There are a few important major category of objects that can appear in the model.

- First, there is the policy object. A policy object represents some data related to a policy that describes a user intent for how the system should behave. A policy object is stored in the policy repository which is the source of "truth" for this object.
- Second, there is an endpoint object. A endpoint represents an entity in the system to which we want to apply policy, which could be a network interface, a storage array, or other relevent policy endpoint. Endpoints are discovered and reported by policy elements locally, and are synchronized into the endpoint repository. The originating policy element is the source of truth for the endpoints it discovers. Policy elements can retrieve information about endpoints discovered by other policy elements by resolving endpoints from the endpoint repository.
- Third, there is the observable object. An observable object represents some state related to the operational status or health of the policy element. Observable objects will be reported to the observer.
- Finally, there is the local-only object. This is the simplest object because it exists only local to a particular policy element. These objects can be used to store state specific to that policy element, or as helpers to resolve other objects. Read on to learn more.

You can use the genie tool that is included with the framework to produce your application model along with a set of generated accessor classes that can work with this framework library. You should refer to the documentation that

accompanies the genie tool for information on how to use to to generate your object model. Later in this guide, we'll go through examples of how to use the generated managed object accessor classes.

4.1.2 Programming by Side Effect

When developing software on the OpFlex framework, you'll need to think in a slightly different way. Rather than calling an API function that would perform some specific action, you'll need to write a managed object to the managed object database. Writing that object to the store will trigger the side effect of performing the action that you want.

For example, a policy element will need to have a component responsible for discovering policy endpoints. When it discovers a policy endpoint, it would write an endpoint object into the managed object database. That endpoint object will contain a reference to policy that is relevant to the endpoint object. This will trigger a whole cascade of events. First, the framework will notice that an endpoint object has been created and it will write it to the endpoint repository. Second, the framework to will attempt to resolve the unresolved reference to the relevent policy object. There might be a whole chain of policy resolutions that will be automatically performed to download all the relevent policy until there are no longer any dangling references.

As long as there is a locally-created object in the system with a reference to that policy, the framework will continually ensure that the policy and any transitive policies are kept up to date. The policy element can subscribe to updates to these policy classes that will be invoked either the first time the policy is resolved or any time the policy changes.

A consequence of this design is that the managed object database can be temporarily in an inconsistent state with unresolved dangling references. Eventually, however, the inconsistency will be fully resolved. The policy element must be able to cleanly handle partially-resolved or inconsistent state and eventually reach the correct state as it receives these update notifications. Note that, in the OpFlex architecture, when there is no policy that specifically allows a particular action, that action must be prevented.

Let's cover one slightly more complex example. If a policy element needs to discover information about an endpoint that is not local to that policy element, it will need to retrieve that information from the endpoint repository. However, just as there is no API call to retrieve a policy object from the policy repository, there is no API call to retrieve an endpoint from the endpoint repository.

To get this information, the policy element needs to create a local-only object that references the endpoint. Once it creates this local-only object, if the endpoint is not already resolved, the framework will notice the dangling reference and automatically resolve the endpoint from the endpoint respository. When the endpoint resolution completes, the framework deliver an update notification to the policy element. The policy element will continue to receive any updates related to that endpoint until the policy element remove the local-only reference to the object. Once this occurs, the framework can garbage-collect any unreferenced objects.

4.1.3 Threading and Ownership

The OpFlex framework uses a somewhat unique threading model. Each managed object in the system belongs to a particular owner. An owner would typically be a single thread that is reponsible for all updates to objects with that owner. Though anything can read the state of a managed object, only the owner of a managed object is permitted to write to it. Though this is not strictly required for correctness, the performance of the system wil be best if you ensure that only one thread at a time is writing to objects with a particular owner.

Change notifications are delivered in a serialized fashion by a single thread. Blocking this thread from a notification callback will stall delivery of all notifications. It is therefore best practice to ensure that you do not block or perform long-running operations from a notification callback.

4.2 Key APIs and Interfaces

4.2.1 Basic Usage and Initialization

The primary interface point into the framework is opflex::ofcore::OFFramework. You can choose to instantiate your own copy of the framework, or you can use the static default instance.

Before you can use the framework, you must initialize it by installing your model metadata. The model metadata is accessible through the generated model library. In this case, it assumes your model is called "mymodel":

```
#include <opflex/ofcore/OFFramework.h>
#include <mymodel/metadata/metadata.hpp>
// ...
using opflex::ofcore::OFFramework;
OFFramework framework;
framework.setModel(mymodel::getMetadata());
```

The other critical piece of information required for initialization is the OpFlex identity information. The identity information is required in order to successfully connect to OpFlex peers. In OpFlex, each component has a unique name within its policy domain, and each policy domain is identified by a globally unique domain name. You can set this identity information by calling:

```
framework
    .setOpflexIdentity("[component name]", "[unique domain]");
```

You can then start the framework simply by calling:

```
framework.start();
```

Finally, you can add peers after the framework is started by calling the `opflex::ofcore::OFFramework::addPeer` method:

```
framework.addPeer("192.168.1.5", 1234);
```

When connecting to the peer, that peer may provide an additional list of peers to connect to, which will be automatically added as peers. If the peer does not include itself in the list, then the framework will disconnect from that peer and add the peers in the list. In this way, it is possible to automatically bootstrap the correct set of peers using a known hostname or IP address or a known, fixed anycast IP address.

To cleanly shut down, you can call:

```
framework.stop();
```

4.2.2 Working with Data in the Tree

Reading from the Tree

You can access data in the managed tree using the generated accessor classes. The details of exactly which classes you'll use will depend on the model you're using, but let's assume that we have a simple model called "simple" with the following classes:

- root - The root node. The URI for the root node is "/"
- foo - A policy object, and a child of root, with a scalar string property called "bar", and a unsigned 64-bit integer property called baz. The bar property is the naming property for foo. The URI for a foo object would be "/foo/[value of bar]/"

- fooref - A local-only child of root, with a reference to a foo, and a scalar string property called "bar". The bar property is the naming property for foo. The URI for a fooref object would be "/fooref/[value of bar]/"

In this example, we'll have a generated class for each of the objects. There are two main ways to get access to an object in the tree.

First, we can get instantiate an accessor class to any node in the tree by calling one of its static resolve functions. The resolve functions can take either an already-built URI that represents the object, or you can call the version that will locate the object by its naming properties.

Second, we can access the object also from its parent object using the appropriate child resolver member functions.

However we read it, the object we get back is an immutable view into the object it references. The properties set locally on that object will not change even though the underlying object may have been updated in the store. Note, however, that its children can change between when you first retrieve the object and when you resolve any children.

Another thing that is critical to note again is that when you attempt to resolve an object, you can get back nothing, even if the object actually does exist on another OpFlex node. You must ensure that some object in the managed object database references the remote managed object you want before it will be visible to you.

To get access to the root node using the default framework instance, we can simply call:

```
using boost::shared_ptr;
using boost::optional;
optional<shared_ptr<simple::root> > r(simple::root::resolve());
```

Note that whenever we can a resolve function, we get back our data in the form of an optional shared pointer to the object instance. If the node does not exist, then the optional will be set to boost::none. Note that if you dereference an optional that has not been set, you'll trigger an assert, so you must check the return as follows:

```
if (!r) {
   // handle missing object
}
```

Now let's get a child node of the root in three different ways:

```
// Get foo1 by constructing its URI from the root
optional<shared_ptr<simple::foo> > foo1(simple::foo::resolve("test"));
// get foo1 by constructing its URI relative to its parent
foo1 = r.get()->resolveFoo("test");
// get foo1 by manually building its URI
foo1 = simple::foo::resolve(opflex::modb::URIBuilder()
                            .addElement("foo")
                            .addElement("test")
                            .build());
```

All three of these calls will give us the same object, which is the "foo" object located at "/foo/test/".

The foo class has a single string property called "bar". We can easily access it as follows:

```
const std::string& barv = foo1.getBar();
```

Writing to the Tree

Writing to the tree is nearly as easy as reading from it. The key concept to understand is the mutator object. If you want to make changes to the tree, you must allocate a mutator object. The mutator will register itself in some thread-local storage in the framework instance you're using. The mutator is specific to a single "owner" for the data, so you can only make changes to data associated with that owner.

Whenever you modify one of the accessor classes, the change is actually forwarded to the currently-active mutator. You won't see any of the changes you make until you call the commit member function on the mutator. When you do that, all the changes you made are written into the store.

Once the changes are written into the store, you will need to call the appropriate resolve function again to see the changes.

Allocating a mutator is simple. To create a mutator for the default framework instance associated with the owner "owner1", just allocate the mutator on the stack. Be sure to call commit() before it goes out of scope or you'll lose your changes.

```
{
    opflex::modb::Mutator mutator("owner1");
    // make changes here
    mutator.commit();
}
```

Note that if an exception is thrown while making changes but before committing, the mutator will go out of scope and the changes will be discarded.

To create a new node, you must call the appropriate add[Child] member function on its parent. This function takes parameters for each of the naming properties for the object:

```
shared_ptr<simple::foo> newfoo(root->addFoo("test"));
```

This will return a shared pointer to a new foo object that has been registered in the active mutator but not yet committed. The "bar" naming property will be set automatically, but if you want to set the "baz" property now, you can do so by calling:

```
newfoo->setBaz(42);
```

Note that creating the root node requires a call to the special static class method createRootElement:

```
shared_ptr<simple::root> newroot(simple::root::createRootElement());
```

Here's a complete example that ties this all together:

```
{
    opflex::modb::Mutator mutator("owner1");
    shared_ptr<simple::root> newroot(simple::root::createRootElement());
    shared_ptr<simple::root> newfoo(newroot->addFoo("test"));
    newfoo->setBaz(42);

    mutator.commit();
}
```

4.2.3 Update Notifications

When using the OpFlex framework, you're likely to find that most of your time is spend responding to changes in the managed object database. To get these notifications, you're going to need to register some number of listeners.

You can register an object listener to see all changes related to a particular class by calling a static function for that class. You'll then get notifications whenever any object in that class is added, updated, or deleted. The listener should queue a task to read the new state and perform appropriate processing. If this function blocks or peforms a long-running operation, then the dispatching of update notifications will be stalled, but there will not be any other deleterious effects.

If multiple changes happen to the same URI, then at least one notification will be delivered but some events may be consolidated.

The update you get will tell you the URI and the Class ID of the changed object. The class ID is a unique ID for each class. When you get the update, you'll need to call the appropriate resolve function to retrieve the new value.

You'll need to create your own object listener derived from opflex::modb::ObjectListener:

```
class MyListener : public ObjectListener {
public:
    MyListener() { }
    virtual void objectUpdated(class_id_t class_id, const URI& uri) {
        // Your handler here
    }
};
```

To register your listener with the default framework instance, just call the appropriate class static method:

```
MyListener listener;
simple::foo::registerListener(&listener);
// main loop
simple::foo::unregisterListener(&listener);
```

The listener will now recieve notifications whenever any foo or any children of any foo object changes.

Note that you must ensure that you unregister your listeners before deallocating them.

4.3 API Reference Documentation

Complete API documentation can be found by in doc/html/index.html in any build.