Gumby

Tribler Team

Oct 17, 2022

OVERVIEW:

CHAPTER

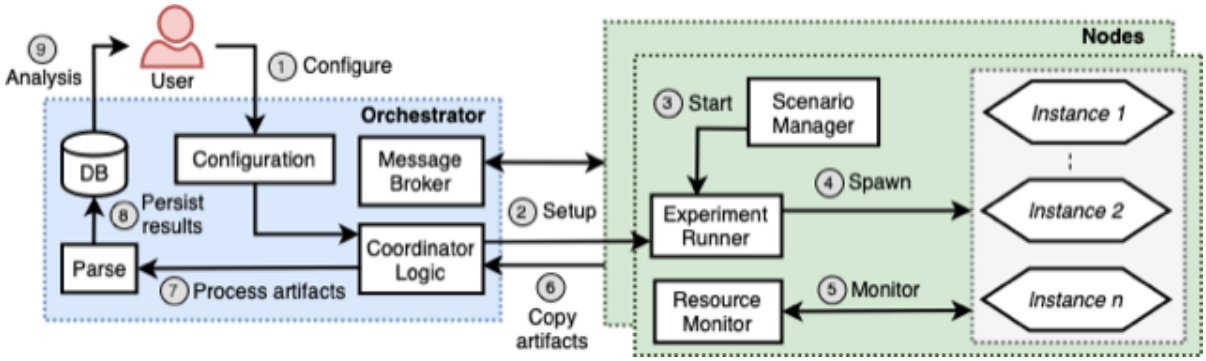
ONE

A QUICK OVERVIEW

Gumby is an experiment runner framework to run local and distributed experiments.

The current supported DAS environments are the DAS5 and DAS6 clusters. More information on the DAS environments can be found at: DAS5. DAS6.

The interaction between the researcher and the gumby framework can be presented als follows:



Installation

Prior to installing Gumby install the required dependencies for basic tests. Those are the software packages: python-psutil python-configobj r-base

The way to install packages is specific for your Linux installation. As an example on the Ubuntu or Debian based systems you may use:

```
$ sudo apt-get install python-psutil python-configobj r-base
```

These dependencies can also be installed using the *pip* utility. Please note that more elaborate experiments might require additional dependencies.

Next, clone the Gumby repository from GitHub by running the following command:

```
$ git clone https://github.com/tribler/gumby
```

In the Tutorials section you will find information on how to create your first Gumby experiment.

CHAPTER

TWO

ADVANCED SCENARIO LANGUAGE CONCEPTS

The document is meant to exemplify some of the more advanced features of the scenario language. Currently, these features refer to:

- Support for `variables`

They will be exemplified and presented in further detail in the sections to follow:

2.1 Variables

Variables have been introduced in the scenario language mainly for those cases when a value is repeatedly used during an experiment. Without variables this would force a programmer to frequently rewrite or copy-paste the literal, which may introduce human-made errors, and is at the very least cumbersome and tiring. Variables can help with this issue, by associating a value to a name / alias.

Variables are by default **strings**, and it is the responsibility of the programmer to ensure that they convert the string value to whatever type they require in the function code. This, however, does not differ to how normal literal parameters need to be processed within a function, as they are **strings** by default as well.

The syntax for declaring a variable is the following:

`@! set <variable_name> <variable_value>`

The scope of the variable begins immediately and persists until the end of the scenario file. The meaning of each component is explained in what follows:

- `@!` is a special timestamp which informs the parser that the command must be executed immediately, as opposed to using asyncio to schedule it in the future, after the scenario has been parsed and the experiment has begun.
- `set` is a special experiment callback, which is used to declare variables
- `<variable_name>` is the name of the variable
- `<variable_value>` is the value assigned to the variable

One can redefine a variable multiple times, and **the scope of the new value begins immediately, while the scope of the previous value will end immediately as well**.

In order to use the variable itself, one simply prepends the name of the variable with `$` and uses it as a parameter, as in the following example:

```
@0:10 emit_value $myVal
```

In the previous example, we assume that `myVal` is a variable which has been previously set using something like `!@ set myVal <variable_value>`.

To better exemplify the usefulness of a variable within a scenario file let us look at a more extensive use case:

```
&module gumby.modules.tribler_module.TriblerModule
&module experiments.dht.dht_module.DHTModule
@0:0 isolate_ipv8_overlay DHTDiscoveryCommunity
@0:1 start_session
@0:1 annotate start-experiment
@0:5 introduce_peers
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {2}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {3}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {4}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {5}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {6}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {7}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {8}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {9}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {10}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {11}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {12}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {13}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {14}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {15}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {16}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {17}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {18}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {19}
@0:10 start_queries 10d00d55231921911991e2f7fd538ec989a2df00 {20}
@0:10 store 10d00d55231921911991e2f7fd538ec989a2df00 value_to_store {1}
@0:50 annotate end-experiment
@0:50 stop_session
@0:55 stop
```

In this example, we will have 20 active peers, 19 of which are querying each other for a particular datum, identified by a key (`d00d55231921911991e2f7fd538ec989a2df00` in this case), while the 20th is publishing the aforementioned datum. One can easily see that copy-pasting the key throughout the scenario is a cumbersome task, and may introduce errors. To this extent, one can simplify the scenario by introducing variables:

```
&module gumby.modules.tribler_module.TriblerModule
&module experiments.dht.dht_module.DHTModule
@0:0 isolate_ipv8_overlay DHTDiscoveryCommunity
@0:1 start_session
@0:1 annotate start-experiment
@0:5 introduce_peers
@! set key 10d00d55231921911991e2f7fd538ec989a2df00
@0:10 start_queries $key {2}
@0:10 start_queries $key {3}
@0:10 start_queries $key {4}
@0:10 start_queries $key {5}
@0:10 start_queries $key {6}
@0:10 start_queries $key {7}
@0:10 start_queries $key {8}
@0:10 start_queries $key {9}
@0:10 start_queries $key {10}
@0:10 start_queries $key {11}
@0:10 start_queries $key {12}
```

(continues on next page)

(continued from previous page)

```
@0:10 start_queries $key {13}
@0:10 start_queries $key {14}
@0:10 start_queries $key {15}
@0:10 start_queries $key {16}
@0:10 start_queries $key {17}
@0:10 start_queries $key {18}
@0:10 start_queries $key {19}
@0:10 start_queries $key {20}
@0:10 store 10$key value_to_store {1}
@0:50 annotate end-experiment
@0:50 stop_session
@0:55 stop
```

The variable introduces a cleaner scenario. Moreover, if further changes are required to the key, one can simply change the value once, when the `key` is set. Previously, if the key needed to be changed, one would have to manually go through each of its occurrences and make the required modification.

CHAPTER

THREE

ISOLATING IPV8 OVERLAYS

The purpose of this document is to show a means of isolating IPv8 overlays from outside interference in Gumby experiments. This document assumes the reader has a basic understanding of running Gumby experiments, creating IPv8 overlays and running them.

As you may have noticed, some of the overlays loaded through Gumby are the actual overlays that are also re-used in production. In some cases this may be desirable functionality, in other cases one would like to isolate these overlays as such that they do not communicate with third parties.

How have we solved this in the past? As you may know, part of the unique identification of a IPv8 overlay is its community ID. Previously, one was required to create a subclass of the overlay under test in Gumby which had a different overlay ID. Even though this is still possible, a system has been implemented in Gumby which allows you to easily isolate and/or replace these existing overlays or add your own.

3.1 Isolation

Isolating a particular community is as simple as adding the following line to a scenario file (before the IPv8 session starts):

```
@0:0 isolate_ipv8_overlay PingPongCommunity
```

In this case, Gumby will automatically search for the launcher associated with the `PingPongCommunity` and replace the existing launcher with an instance of `IsolatedIPv8LauncherWrapper`. All information in the existing launcher will be copied to the instance of `IsolatedIPv8LauncherWrapper`. The community ID will be randomized such that it becomes more difficult for the Gumby experiment to interfere with deployed communities.

CHAPTER

FOUR

RUNNING A SIMPLE EXPERIMENT WITH GUMBY

In this first tutorial, we will run a very simple experiment using the functionality that Gumby provides. This experiment is executed locally and spawns two independent CPU-intensive instances (processes) that execute the `yes` command for ten seconds.

The entry point for all Gumby experiments is the configuration (`.conf`) file. As the name implies, configuration files specify how an experiment is configured. This configuration is a file containing key and values, and includes an experiment name, the number of instances being spawned and which command is being executed. For our first experiment, this configuration file looks as follows:

```
# The name of the experiment.
experiment_name = LocalProcessGuard

# The number of instances that we spawn.
instances_to_run = 2

# The command that each instance runs. This command will spawn the process guard that in␣
↪turns spawns two instances that run the yes command for ten seconds.
local_instance_cmd = process_guard.py -c "(yes CPU > /dev/null & find /etc /usr > /dev/
↪null ; wait)" -n $INSTANCES_TO_RUN -t 10 -m $OUTPUT_DIR -o $OUTPUT_DIR --network

# The command to run after the experiment is finished. The graph_process_guard_data bash␣
↪script plots various resource statistics (e.g., CPU, memory, I/O etc.).
post_process_cmd = graph_process_guard_data.sh

# In this simple experiment, we do not use a synchronization server and spawn␣
↪independent processes.
experiment_server_cmd = ""
```

We annotate each configuration option with a small explanation. Of particular interest is the `local_instance_cmd` option. In this experiment, we start the process guard which is a small module that can spawn and monitor subprocesses. We provide the command that we want each instance to run as argument to the process guard script. the `-t` flag specifies the experiment timeout. When this timeout expires, the spawned instances are terminated. For more information on the process guard, we refer the reader to the Gumby documentation.

When the experiment ends, the script provided to the `post_process_cmd` configuration option will run, if provided. The `graph_process_guard_data.sh` will create graphs from the data collected by the process guard using the R plotting library.

Note that we set the `experiment_server_cmd` option to an empty value. The default way of running Gumby is to spawn instances that are time-synchronized with each other and can communicate with each other. However, for this simple tutorial we sidestep this and spawn two independent instances instead.

To run this basic experiment, execute the following command in the directory that contains the `gumby` directory:

```
$ gumby/run.py gumby/docs/tutorials/local_processguard.conf
```

After around ten seconds, the experiment should be done. All experiment artifacts are written to the `output` directory. The standard output and standard error streams are written to the `.out` and `.err` files, respectively. You should also see the raw monitoring statistics collected by the process guard, and the plotted graphs (the `.png` files). The `utimes.png` file shows the CPU usage (in user mode) for each spawned instance. Note that process guard uses the `procfs` file system to gather statistics and therefore resource statistics will not be available when the experiment runs on Mac and/or Windows computers.

That's all for now! By modifying the `local_instance_cmd` configuration option, you can run custom commands with as many instances your experiment requires. In the next tutorial, we will use more advanced concepts of Gumby and show how to spawn time-synchronized instances and how to work with scenario files.

CHAPTER

FIVE

ADVANCED GUMBY EXPERIMENTS AND SCENARIOS

In the previous tutorial, we have shown how to run a simple experiment with Gumby. Gumby spawned two instances that run independently from each other. Most of the time, however, you want more control over the actions performed during your experiment. In this tutorial, we will setup a more advanced experiment. Our experiment will spawn five instances that are time-synchronized with each other. After five seconds, each instance will write its ID to a file. When the experiment ends, we will sum up all these instance IDs and write it to a file.

5.1 Configuration file

Our configuration file will look as follows:

```
experiment_name = synchronized_instances
experiment_time = 20
instances_to_run = 5
local_instance_cmd = "process_guard.py -c launch_scenario.py -n $INSTANCES_TO_RUN -t
↪$EXPERIMENT_TIME -m $OUTPUT_DIR -o $OUTPUT_DIR "
post_process_cmd = post_process_write_ids.sh

# The scenario file to run after an instance has spawned.
scenario_file = write_ids.scenario

# The host and port of the synchronization server.
sync_host = localhost
sync_port = __unique_port__
```

Most of the configuration options should be familiar at this point. We annotated two new configuration options, namely `sync_port` and `scenario_file`. We further explain the effect of these configuration options.

5.2 Scenario Files

Note that we specify a scenario file in our configuration file. A scenario file is a file that contains all events during our experiment. Each event includes a time when the event should fire, and a reference to a Python method. The scenario file is a very convenient way of building an experiment, and can quickly be reused. For example, a scenario file can contain a particular workload that the system should process. The scenario file used in our experiment looks as follows:

```
&module docs.tutorials.simple_module.SimpleModule

@0:5 write_peer_id
@0:9 stop
```

The first two lines specify which modules to include. We will explain experiment modules later in this tutorial. For now, we will focus on the two events that are specified in this scenario file:

```
@0:5 write_peer_id
@0:9 stop
```

The above two events are scheduled to fire after five and nine seconds after experiment start, respectively. The first event calls the `write_peer_id` method that simply writes the ID of the instance, or peer, to a file. Specifically, each Gumby instance is assigned a unique ID, starting from 1. The `write_peer_id` method is defined in the `SimpleModule` class in the `simple_module.py` file. This Python code is imported in the first line of our scenario file. The `stop` method is implemented in the `ExperimentModule` class, which is the superclass of `SimpleModule`.

By default, an event will be executed by all peers. One can restrict which instances run a particular event. For example, the line below specifies that only peer 2 writes its peer ID:

```
@0:5 write_peer_id {2}
```

5.3 Experiment Modules

Gumby enables experiment designers to provide their functionality in separate modules. In the experiment described in this tutorial, we import the `SimpleModule` class in our experiment, which has the following content:

```
from gumby.experiment import experiment_callback
from gumby.modules.experiment_module import ExperimentModule


class SimpleModule(ExperimentModule):
    """
    A very simple experiment module that has a single callback.
    """

    @experiment_callback
    def write_peer_id(self):
        """
        Simply write my peer ID to a file.
        """
        with open("id.txt", "w") as id_file:
            id_file.write("%d" % self.my_id)
```

This file contains the definition of the `SimpleModule` class. The class contains a single method, namely `write_peer_id`. This method simply writes the ID of the peer to a file named `id.txt`. Note that the name of this method corresponds to the event specified in our scenario file, and this method is invoked when the event fires. The method is annotated with a `experiment_callback` decorator. This is required to correctly connect the event in the scenario file and the logic in the module.

An experiment can also import multiple modules. Gumby automatically imports these module on runtime and registers the events to the available callbacks. This allows re-use of experiment logic across different experiments.

5.4 Instance Synchronization

Each spawned instance independently executes the scenario file. This makes it important that each instance starts roughly at the same time. Gumby includes a synchronization server that prepares peers for the experiment, assigns IDs, and makes sure that peers start execution of the scenario file roughly at the same time. This synchronization process is coordinated by Gumby automatically. Recall that, in contrast to the previous experiment, we do not set the `experiment_server_cmd` configuration option to blank. This ensures that Gumby will spawn a synchronization server. The `sync_host` option specifies the IP address or host name of the machine running the synchronization server. In this experiment we set it to localhost. The `sync_port` option in the configuration file specifies the port of the synchronization server, and indicates to which port the spawned clients should connect. A value of `__unique_port__` indicates that Gumby will pick a random free port on runtime.

5.5 Post-experiment Data Aggregation

We now focus on the post-experiment script. This post-experiment script is executed after the scenario file is finished and reads all written peer IDs and sums them. Note that in the configuration file, we specify to run the `post_process_write_ids.sh` bash script, which looks as follows:

```
#!/usr/bin/env bash
gumby/docs/tutorials/post_process_write_ids.py .

graph_process_guard_data.sh
```

This simple script first executes the `post_process_write_ids.py` file and then calls the `graph_process_guard_data.sh` script to plot the graphs. The content of the `post_process_write_ids.py` file is as follows:

```
#!/usr/bin/env python3
import os
import sys

from gumby.statsparser import StatisticsParser


class IDStatisticsParser(StatisticsParser):
    """
    Simply read all the id.txt files created by instances and sum up the numbers inside␣
↪them.
    """

    def aggregate_peer_ids(self):
        peer_id_sum = 0
        for _, filename, _ in self.yield_files('id.txt'):
            with open(filename, "r") as peer_id_file:
                read_peer_id = int(peer_id_file.read())
                peer_id_sum += read_peer_id

        with open("sum_id.txt", "w") as sum_id_file:
            sum_id_file.write("%d" % peer_id_sum)

    def run(self):
```

(continues on next page)

(continued from previous page)

```
        self.aggregate_peer_ids()


# cd to the output directory
os.chdir(os.environ['OUTPUT_DIR'])

parser = IDStatisticsParser(sys.argv[1])
parser.run()
```

This Python file defines the `IDStatisticsParser` class which is a subclass of `StatisticsParser`. The latter class provides basic functionality to quickly aggregate data generated by peers. Of particular interest is the `yield_files` method that returns an iterator with files created by experiment peers that match a particular pattern. In the `aggregate_peer_ids` method we iterate through all the files named `id.txt`, read them, and aggregate the integer value included in these files. Then the result is written to the `sum_id.txt` file.

5.6 Running the Experiment

You can run the experiment with the following command:

```
$ gumby/run.py gumby/docs/tutorials/synchronized_instances.conf
```

This will execute the experiment described above. You should see something similar to the log lines below:

```
INFO:ProcessRunner:[] ERR: 2021-07-17 17:55:23,644:INFO:1 of 5 expected subscribers␣
↪connected.
INFO:ProcessRunner:[] ERR: 2021-07-17 17:55:26,639:INFO:2 of 5 expected subscribers␣
↪connected.
INFO:ProcessRunner:[] ERR: 2021-07-17 17:55:27,645:INFO:4 of 5 expected subscribers␣
↪connected.
INFO:ProcessRunner:[] ERR: 2021-07-17 17:55:27,646:INFO:All subscribers connected!
INFO:ProcessRunner:[] ERR: 2021-07-17 17:55:27,646:INFO:1 of 5 expected subscribers␣
↪ready.
INFO:ProcessRunner:[] ERR: 2021-07-17 17:55:27,649:INFO:All subscribers are ready,␣
↪pushing data!
INFO:ProcessRunner:[] ERR: 2021-07-17 17:55:27,649:INFO:Pushing a 359 bytes long json␣
↪doc.
INFO:ProcessRunner:[] ERR: 2021-07-17 17:55:27,649:INFO:1 of 5 expected subscribers␣
↪received the data.
INFO:ProcessRunner:[] ERR: 2021-07-17 17:55:27,649:INFO:Data sent to all subscribers,␣
↪giving the go signal in 1.0 secs.
INFO:ProcessRunner:[] ERR: 2021-07-17 17:55:27,650:INFO:Starting the experiment!
```

These log lines indicate that the spawned instances are connecting with the synchronization server and that the experiment starts only after all instances have an ID assigned and are synchronized.

A file named `sum_id.txt` should have been created in the experiment output directory. This file should contain the value 15, which corresponds to the sum of the IDs of all participating peers (1+2+3+4+5). Note that Gumby also creates sub-directories to store particular files created by individual peers. The directory name corresponds with the peer ID. These sub-directories should contain the `id.txt` file, created by the `write_peer_id` method.

This tutorial covere more advanced concepts of Gumby, and shows how one can setup advanced experiments using scenario files and experiment modules. In the next tutorial, we show how to deploy and execute the above experiment on the DAS5 supercomputer.

CHAPTER

SIX

RUNNING DISTRIBUTED EXPERIMENTS ON THE DAS

In the previous tutorials we have devised experiments that spawn multiple instances on a single computer. In this tutorial, we will show how to run an experiment on the DAS compute cluster. The DAS is a Dutch nation-wide compute infrastructure that consists of multiple clusters, managed by different universities. This tutorial assumes that the reader has access to the DAS head nodes.

The experiment we will run on the DAS is the same experiment that we described in the previous tutorial. In this experiment, we will spawn multiple (synchronized) instances and each instance will write its ID to a file five seconds after the experiment starts. The experiment is started on the DAS head node and before the instances spawn, Gumby automatically reserves a certain number of compute nodes. Each compute node then spawns a certain number of instances, depending on the experiment configuration. When the experiment ends, all data generated by instances is collected by the head node.

The configuration file for this DAS experiment looks as follows:

```
experiment_name = synchronized_instances_das5
instances_to_run = 16
local_instance_cmd = das_reserve_and_run.sh
post_process_cmd = post_process_write_ids.sh
scenario_file = write_ids.scenario
sync_port = __unique_port__

# The command that is executed prior to starting the experiment. This script prepares␣
↪the DAS environment.
local_setup_cmd = das_setup.sh

# We use a venv on the DAS since installing packages might lead to conflicts with other␣
↪experiments.
use_local_venv = TRUE

# The number of DAS compute nodes to use.
node_amount = 2

# The experiment timeout after which the connection with the compute node is closed.
node_timeout = 20

# What command do we want to run?
das_node_command = launch_scenario.py
```

The new configuration options are annotated with some explanation. It includes a `local_setup_cmd` configuration option that is executed before the experiment starts. The `das_setup.sh` script checks the user quote on the DAS and invokes the `build_virtualenv.sh` script that prepares a virtual environment with various Python packages. To use this virtual environment, the `use_local_venv` option is set.

Additionally, there are a few configuration options that are specific to DAS related experiments. The `node_amount` configuration option indicates how many DAS compute nodes are used. The maximum number of compute nodes in each cluster can be found at the DAS sites listed in the Quick Overview. In our experiment, we spawn 16 instances and use 2 compute nodes. Gumby automatically balances instances over compute nodes and in our experiment, each compute node hosts 8 instances. The `node_timeout` configuration option indicates the timeout of the experiment. To prevent premature termination of an experiment, we recommend to set this value a bit higher than the time of the latest event in the scenario file.

To run this experiment, execute the following command on one of the DAS system head nodes:

```
$ gumby/run.py gumby/docs/tutorials/synchronized_instances_das.conf
```

CHAPTER

SEVEN

RUNNING AN EXPERIMENT WITH IPV8 OVERLAYS

In the previous tutorials, we have discussed how to run both experiments on a single computer and on the DAS5 cluster. We now show how to combine Gumby experiments with IPv8 overlays and show how we can use Gumby to quickly test decentralized overlays. We assume that the reader is familiar with IPv8 overlays. Otherwise, we refer to the IPv8 documentation for more information.

In the following experiment, we spawn two instances and each instance joins a simple IPv8 community. After five seconds, instance 1 sends a `ping` message to instance 2 and instance 2 responds with a `pong` message. Our configuration file looks as follows:

```
experiment_name = simple_ipv8
experiment_time = 30
instances_to_run = 2
local_instance_cmd = "process_guard.py -c launch_scenario.py -n $INSTANCES_TO_RUN -t
↪$EXPERIMENT_TIME -m $OUTPUT_DIR -o $OUTPUT_DIR "
post_process_cmd = graph_process_guard_data.sh
scenario_file = simple_ipv8.scenario
sync_host = localhost
sync_port = __unique_port__
```

All these configuration options have been discussed in previous tutorials. Our scenario file looks as follows:

```
&module gumby.modules.base_ipv8_module.BaseIPv8Module
&module docs.tutorials.ping_pong_module.PingPongModule

@0:0 isolate_ipv8_overlay PingPongCommunity
@0:1 start_session
@0:1 annotate start-experiment
@0:5 introduce_peers
@0:10 send_ping {1}
@0:15 stop_session
@0:20 stop
```

There are several things going on here. First, we call the `isolate_ipv8_overlay` method, which ensures that the overlay loaded in our experiments do not communicate with peers external to our experiments. After 1 second, we call the `start_session` method that starts the IPv8 service. After 5 seconds, we introduce the peers to each other so peers know about each other and can send messages. After 10 seconds, we call the `send_ping` message (defined in the `PingPongModule` class) which sends a `ping` message to all known peers. We then stop the IPv8 service after 15 seconds and stop the entire experiment after 20 seconds.

Note that the scenario file imports the `ping_pong_module.py` file. The content of this file is as follows:

```python
from binascii import unhexlify

from ipv8.community import Community
from ipv8.lazy_community import lazy_wrapper
from ipv8.loader import overlay
from ipv8.messaging.lazy_payload import VariablePayload, vp_compile
from ipv8.types import Peer

from gumby.experiment import experiment_callback
from gumby.modules.community_experiment_module import IPv8OverlayExperimentModule
from gumby.modules.ipv8_community_launchers import IPv8CommunityLauncher


# ---- messages ---- #
@vp_compile
class PingPayload(VariablePayload):
    msg_id = 1


@vp_compile
class PongPayload(VariablePayload):
    msg_id = 2


# ---- community ---- #
class PingPongCommunity(Community):
    community_id = unhexlify("d37c847b628e1414cffb6a4626b7fa0999fba888")

    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.add_message_handler(PingPayload, self.received_ping)
        self.add_message_handler(PongPayload, self.received_pong)

    def send_ping(self):
        for peer in self.network.verified_peers:
            self.logger.info("Sending ping to peer %s", peer)
            self.ez_send(peer, PingPayload())

    @lazy_wrapper(PingPayload)
    def received_ping(self, peer: Peer, _: PingPayload):
        self.logger.info("Received ping from peer %s", peer)
        self.ez_send(peer, PongPayload())

    @lazy_wrapper(PongPayload)
    def received_pong(self, peer: Peer, _: PongPayload):
        self.logger.info("Received pong from peer %s", peer)


# ---- launcher ---- #
@overlay(PingPongCommunity)
class PingPongCommunityLauncher(IPv8CommunityLauncher):
    pass
```

(continues on next page)

(continued from previous page)

```
# ---- module ---- #
class PingPongModule(IPv8OverlayExperimentModule):
    """
    This module contains code to manage experiments with the Basalt community.
    """
    def __init__(self, experiment):
        super().__init__(experiment, PingPongCommunity)

    def on_id_received(self):
        super().on_id_received()
        self.ipv8_provider.custom_ipv8_community_loader.set_
↪launcher(PingPongCommunityLauncher())

    @experiment_callback
    def send_ping(self):
        self.overlay.send_ping()
```

This file implements the following:

- We define two IPv8 payloads, namely `PingPayload` and `PongPayload`.
- We define the `PingPongCommunity` class which contains some logic when receiving ping and pong messages.
- We implement a `PingPongCommunityLauncher` which is used by Gumby to correctly load the community on experiment startup.
- Finally, we implement the `PingPongModule` which contains a single experiment callback.

You can run the experiment with the following command:

```
$ IPV8_DIR=<PATH TO IPV8> gumby/run.py gumby/docs/tutorials/simple_ipv8.conf
```

Note the `IPV8_DIR` environment variables which tells Gumby where to find the IPv8 source code. You should change this to point to your IPv8 installation. The experiment ends after around 20 seconds. Inspecting the instance output should reveal log lines like:

```
2021-07-18 12:52:06,699:INFO:Sending ping to peer Peer<127.0.0.1:12002,␣
↪s4WWx6gchewSqE27LpP+xBt8QsE=>
2021-07-18 12:52:06,700:INFO:Received pong from peer Peer<127.0.0.1:12002,␣
↪s4WWx6gchewSqE27LpP+xBt8QsE=>
```

This shows that the peers have successfully communicated with each other using IPv8 and Gumby.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search