Implementing and deploying experiments using Dispersy and Gumby

M.A. de Vos Q. Stokkink

Distributed System group, EEMCS, TUDelft

November 28, 2017

This document describes how to implement and deploy a distributed experiment using Dispersy and Gumby. After finishing the assignments in this document, you will be able to to:

- Send messages to other users in a peer-to-peer network using Dispersy.
- Implement and modify a distributed experiment using Gumby. This includes writing your own scenarios and defining experiment modules.
- Deploy and execute your experiment on the DAS-5 supercomputer.

This tutorial assumes that the reader has a basic understanding of the Python programming language and basic familiarity with git.

1 Introduction

Experimentation lies at the heart of science. Experiments are important for verification of hypotheses or to analyse behaviour of algorithms. Particularly in the field of distributed systems, experiments are often executed at a large scale, involving hundreds of clients producing data and communicating with each other. Being able to define and run your own experiments is an essential skill.

At the distributed systems department and in particular the Blockchain Lab [1], we have created various tools to ease the design, implementation and deployment of experiments. In this workshop, we will explore two of these tools, Dispersy and Gumby, and show the practical value of these tools using a basic distributed algorithm that involves cryptographic secret sharing and network communication. Before turning our attention to the experiment, we will first introduce Dispersy, Gumby and the DAS-5 supercomputer.

1.1 Dispersy

Dispersy can be described as a scalable, distributed database, however, it also provides primitives for sending generic messages between users. Dispersy has the notion of *communities* which are best described as a group of users that share a common goal or objective. Some examples of available communities are:

- *SearchCommunity*: This community is responsible for keyword search of content within Tribler, our peer-to-peer filesharing software.

- *MarketCommunity*: This community enables peer-to-peer trading, trade matching and transaction processing.
- *TrustChainCommunity*: This community is the implementation of our scalable blockchain fabric.

Creating your own Dispersy community is straightforward and will be demonstrated in this tutorial. Dispersy is open-source software and can be downloaded from Github[1]. More information about Dispersy can be found in the technical report [4].

1.2 Gumby

Gumby is a framework specifically designed to run experiments. These experiments can either be executed locally, remotely on a server or on the DAS-5 supercomputer. Gumby can be downloaded from Github[2], however, local experiment execution is only possible on Linux-based environments due to the requirement of the *procfs* file system. This tutorial will explain the basic concepts in Gumby.

1.3 DAS-5 Supercomputer

As you might know, the DAS-5 supercomputer is not a single machine but shared amongst various universities in the Netherlands. Each university involved in the DAS-5 project hosts a cluster with a fixed amount of nodes. Every cluster has exactly one head node that can be accessed through SSH. This head node is basically the gateway to the other subnodes. The head node and subnodes share the same file system so all files available on the head node can be accessed by the subnodes and vice versa. Detailed information, including the number of nodes available in each cluster and hardware specifications, can be found on the DAS-5 website[3].

Since there are many researchers working on large-scale experiments, the DAS-5 has a reservation system where you can reserve some nodes in advance. This is in particular helpful to make sure that only you are running an experiment on a specific node, avoiding interference or resource usage from other researchers. However, the reservation system can be frustrating at times and it might be hard to claim a reservation when it's busy. Keep this in mind when you might have to run experiments multiple times before conferences or journal deadlines. Gumby takes care of automatically reserving and releasing nodes during experiments.

2 Create Your Experiment

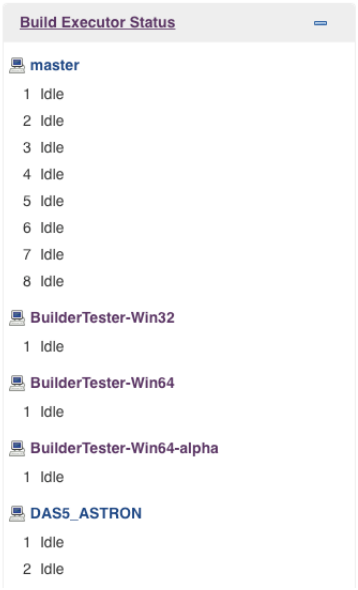
Now that we introduced Dispersy and Gumby, it is time to see what an experiment looks like and how it is being executed.

2.1 Jenkins

Start by opening a web browser and navigate to our Jenkins server, located at https://jenkins.tribler.org. Jenkins is open-source software to automate running jobs like tests or experiments. It is often used during the software development cycle to make sure that software is still stable. Jenkins jobs can be

[1] https://github.com/tribler/dispersy
[2] https://github.com/tribler/gumby
[3] http://www.cs.vu.nl/das5/clusters.shtml

executed on machines, which are displayed under *Build Executor Status*. Here you will see an overview of all the machines that are capable of executing Jenkins jobs (these machines are also called *slaves*). Take a look at the different machines: the slaves that are prefixed with *DAS5_* are clusters of the DAS-5 supercomputer. Additionally, some machines running Windows or macOS are available.



Navigate to the *pers* directory on the Jenkins website and locate the job named *workshop_benaloh_test*. We will copy this job so we can modify it accordingly. To create a copy of an existing job, you should have a valid Jenkins account; if you don't have one, please ask one of the workshop instructors for one.

Start by clicking on the *New Item* in the menu in the upper-left corner of the Jenkins website. You are now prompted to enter the name of the new job. Name it something like *workshop_benaloh_test_username* where *username* should be replaced by your Jenkins username or your real name. Since we copy an existing job, enter the name of this job under the *Copy from* input, at the bottom of the page (which should be *workshop_benaloh_test*) and press the *OK* button to create the job.

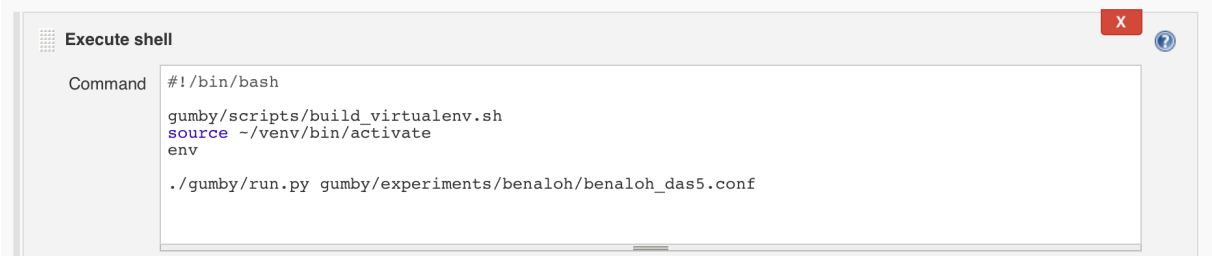
You will be redirected to the configuration options of the newly created Jenkins job. Here you can change various settings that are related to your experiment. Some of the most important experiments will be elaborated now:

- *Restrict where this project can be run*: here you specify on which Jenkins slave you wish to run your job. It should be filled with *DAS5_TUD*, which is the TU Delft cluster of the DAS-5 supercomputer. You can also select other DAS5 clusters here but the default option should work fine.

- *Source Code Management*: this section specifies whether the experiment needs some code from external sources like a Version Control System (Git, SVN etc) to operate. Our first option specifies that we are using the workspace of another Jenkins job here (*Clone_tribler_devel*). This makes

sure that Tribler is downloaded into the Jenkins workspace since Tribler is a dependency of Gumby. Next, we specify that we want to clone a Git repository, namely the Gumby tutorial. The repository URL is specified, including the git branch that we want to checkout. These options together make sure that we have a copy of Tribler and Gumby ready before the experiment starts.

- *Build*: this section indicates the performed actions during the experiment execution. In this case, we run a simple shell script where activate a virtual Python environment (this makes sure that all required Python libraries can be found but it's not very interesting for now). The final line of the bash script starts the Gumby experiment.



- *Post-build Actions*: here we specify actions that should be performed after the experiment has finished. In this experiment, we store files in the *output* directory for later usage (this is called archiving). Finally, we create an image gallery from *.png* files.

By pressing the *Save* button at the bottom of the webpage, you save the configuration and return to the overview of the specific job. You can always return to the configuration overview by clicking the *Configure* menu entry.

When executing a job, a new *build* is created. Each build has a number of outcomes, including *succeeded* (everything went well), *failed* (something went wrong during the execution of the build) or *aborted* (the job got manually or automatically aborted). The history of builds is presented under the *Build History* tab on the left. Here you will see the time, outcome and amount of produced data for each build.

2.2 Running The Experiment

We are now ready to run your newly created job! Executing jobs can be done by pressing the *Build Now* button in the upper-left menu, please do so now. Refresh the web page shortly after pressing the button and you will notice that a new entry appears under the *Build History* overview on the left. You will see a progress bar that indicates the progress of the build. Click on the build number to navigate to the page with specific details of each build. We can monitor the progress of each build in detail by clicking on the *Console Output* menu entry on the left. The experiment should take several minutes to finish and when it does, you will see something similar to the following lines in the console log:

```
1 Creating image galleries.Creating archived images gallery.No files found for
2 image gallery.Started calculate disk usage of build
3 Finished Calculation of disk usage of build in 0 seconds
4 Started calculate disk usage of workspace
5 Finished Calculation of disk usage of workspace in 0 seconds
6 Notifying upstream projects of job completion
7 Finished: SUCCESS
```

This indicates that the build has succeeded. We can now investigate the output files of the selected build by going to the *Build Artifacts*. This opens a basic file explorer where you can navigate through all artifacts generated by the build. The experiment uses five nodes on the DAS-5 supercomputer and each node writes its output to a directory in *output/localhost*[4]. If you navigate to *output/localhost*, you will see five directories. Click on any of them, for instance, *node301*, to view files produced by this node. Browse the content of the file named *00000.out* which contains the standard out (stdout) output of the processes running on this DAS-5 node. You should see the following lines:

```
1 My computed sum is 656
2 My secret value is 722
```

The computed sum should be the same for all nodes but the secret values are (probably) different.

[4]In actuality the head node of the cluster collects the results of each of the nodes and writes them there.

3 The Benaloh Experiment

The output produced in the last section comes from an implementation of a simplified version of the Benaloh secret sharing scheme [2]. You can find the complete code of this exercise on GitHub[5]. In particular we will focus on the different ways to share data and not so much on the security or efficiency of the algorithm. The protocol is implemented as follows:

1. Every node starts with a globally known node count n and modulus M and a private share s. The nodes then generate $n-2$ random numbers R and calculates $s - \Sigma R$ individually.
2. Every node shares all values from step 1 **directly** with all other nodes, in a point-to-point fashion (using a `local-share` message).
3. Every node sums all the received values from step 2 and **broadcasts** this value to all other nodes (using a `broadcast-share` message).
4. All nodes print the sum of all received values from step 3.

We will now explain all components that together form this experiment.

3.1 The Dispersy Community

Recall that a Dispersy community is a group of users that share a common goal or objective. In this experiment, our objective is secret sharing. The community provides message definitions to do so. The source code of this community is given here:

https://github.com/qstokkink/gumby/blob/workshop_code/experiments/benaloh/benaloh_community.py

Globally this community needs the following components to function:

- A `Payload` object, representing packets going over the network.
- A `Conversion` object, which can transform `Payload` objects into binary data.
- An object that is a subclass of `Community`, handling incoming `Payload` objects and providing functionality to send `Payload` objects to other users in the community.

The finer details of creating these objects will remain out of the scope of this document, but we refer the interested reader to our extensive Dispersy tutorial[6]. For some general clarification, we would now like to draw the attention of the reader to the different types of messages defined in the `BenalohCommunity.initiate_meta_messages()` method.

The first type of message you will find defined in the community file is a **broadcast** message using Dispersy's gossiping: `u"broadcast-share"`. This message uses `MemberAuthentication`, which means that every message will be signed and verified when received. The alternative to `MemberAuthentication` is `NoAuthentication`, where messages won't be signed and verified. This option is helpful when you want to save CPU cycles since verification of a cryptographic signature is a costly and relatively time-consuming operation. Next, you will see `FullSyncDistribution(u"ASC", 128, False)`; this takes care

[5] https://github.com/qstokkink/gumby/tree/workshop_code/experiments/benaloh
[6] https://dispersy.readthedocs.org

of stamping your message with a global time (a Lamport clock timestamp to be precise [3]), avoiding processing of duplicate messages. Besides this, the `FullSyncDistribution` definition also serves as a flag to tell Dispersy that your message needs to be gossiped throughout the complete network. The `CommunityDestination(10)` directive then tells Dispersy that your message will be gossiped to 10 connected nodes. Finally, `self.on_broadcast_share` tells Dispersy which method handles incoming messages of the
`u"broadcast-share"` persuasion.

The other message in this file is the `u"local-share"` message. This message can be sent directly to other known peers. You will notice that the difference between the two messages lies in the declaration of the `DirectDistribution` and `CandidateDestination`. These two flags let Dispersy know that you want to directly send this message to others and you will be the one in charge of supplying the peers/candidates to send it to.

One observation you will make is that it is much faster to send messages to other candidates directly instead of gossiping it (using the broadcast message). Still, it would not be wise, even erroneous, to use the direct candidate sending, as given by the supplied code, outside of Gumby.

3.2 The Gumby Experiment

Now that you know how a basic Dispersy community works and that you understand messaging between nodes, we should focus on the Gumby experiment that uses the Dispersy community.

3.2.1 Gumby Configuration File

Each experiment has a configuration file, which specifies generic settings of each experiment. This file can be found here:

https://github.com/qstokkink/gumby/blob/workshop_code/experiments/benaloh/benaloh_das5.conf

This file has the following content:

```
1  experiment_name = "benaloh"
2  scenario_file = 'benaloh.scenario'
3
4  das4_node_amount = 5
5  das4_instances_to_run = 5
6  das4_node_timeout = 165
7
8  local_setup_cmd = 'das4_setup.sh'
9  local_instance_cmd = 'das4_reserve_and_run.sh'
10 das4_node_command = "launch_scenario.py"
11 tracker_cmd = 'run_tracker.sh'
12 tracker_port = __unique_port__
13 experiment_server_cmd = 'experiment_server.py'
14 sync_port = __unique_port__
15 sync_experiment_start_delay = 1
16 use_local_venv = TRUE
```

```
17 with_systemtap = false
```

We will now elaborate some important configuration variables:

1. *scenario_file*: this option specifies the scenario file that the experiment should use. They should be located in the same directory as the configuration file. We will explain scenario files in the next section.

2. *das4_node_amount*: specifies the total number of nodes that should be used for the experiment.

3. *das4_instances_to_run*: the number of instances to run **in total**. Each DAS-5 node can run multiple instances. Gumby automatically load-balances the instances over the nodes. In this scenario, Gumby will schedule one instance on each node.

4. *das4_node_timeout*: the timeout of the DAS-5 node in seconds. After this duration, your DAS-5 reservation will expire and your experiment will be aborted. It is important to make sure that your experiment finishes within the specified DAS-5 timeout period to avoid unexpected results during your experiments.

The other configuration options are often not modified and are badly documented, however, if you have any questions about them, you can ask the instructors for more information. For your convenience, a configuration file for running the experiment locally has also been provided (*benaloh.conf*).

3.2.2 Gumby Scenario File

Each experiment has a scenario file attached that specifies exactly what should happen during the experiment. This file can be found here:

https://github.com/qstokkink/gumby/blob/workshop_code/experiments/benaloh/benaloh.scenario

This file has the following content:

```
1 &module gumby.modules.dispersy_module.DispersyModule
2 &module experiments.benaloh.benaloh_module.BenalohModule
3
4 @0 isolate_community BenalohCommunity
5 @1 start_session
6 @2 introduce_peers
7 @15 reset_dispersy_statistics
8 @15 annotate start-experiment
9
10 @30 share_local
11 @45 share_subset_sum
12
13 @1:30 print_local_value
14 @1:30 print_total_sum
15
16 @2:0 stop_session
17 @2:15 stop
```

The first two lines indicate the required Gumby modules for this experiment. The exact specification of a Gumby module will be discussed in the next section. Note that we are using two different modules, namely `DispersyModule` and `BenalohModule`. `DispersyModule` provides support for using Dispersy during our experiment while `BenalohModule` implements functionality specifically for our experiment. The full path to these modules must be provided.

After line 4, our scenario starts. Each line starting with a @ symbol specifies a command. Let's consider the command on line 5. The number after the @ symbol specifies the time into the experiment when this command should be executed, in this case, one second after the experiment starts. Next, the command itself is specified which is *start_session*. This command makes sure that Dispersy is started and is listening for network traffic on the right interfaces. Optionally, we can provide some arguments. For instance, the string `BenalohCommunity` is passed as argument to the `isolate_community` command on line 4. We now elaborate some of the commands found in the scenario file:

1. *isolate_community*: when we define and start a community during an experiment, it will connect to the "live" network with real users. This is often not desired since we want to run our experiments in a controlled environment. This command isolates the community that is passed as argument to prevent any communication with users outside the experiment.

2. *introduce_peers*: this command makes sure that all experiment instances know about the existence of each other. Note that this does not resemble real-world behaviour where users in a network do not discover all other users immediately, but it might be helpful during experimentation.

3. *reset_dispersy_statistics*: during the experiment, Dispersy keeps track of various statistics like the number of incoming and outgoing connections and data transferred. This command resets all statistics.

4. *annotate*: often, an experiment is divided in several phases. For instance, there might a phase where users compute a value and a phase where they share this value with others. Developers are able to annotate important timestamps during the experiment with the *annotate* command. These annotations will be visible in the final plots when the experiment is finished.

5. *share_local*: this command will share the locally generated random values with other users in the network and will send a `local-share` message. This happens exactly 30 seconds after the experiment has started. The implementation of this command is present in the `BenalohModule` file which will be explained in the next section.

6. *share_subset_sum*: this command will share the subset sum and broadcasts it to all users using a `broadcast-share` message. This is also custom functionality provided by the `BenalohModule`.

7. *print_local_value* and *print_total_sum*: these commands will simply print the local value and total sum to the standard output. These commands are responsible for the lines in the *.out* file, explored in Section 2. Note that we are able to specify duration in $[[hh:]mm:]ss$ format so these commands will be executed 90 seconds into the experiment (1:30 minutes).

8. *stop_session*: this command will stop the running Dispersy session after two minutes.

9. *stop*: this will terminate the running process.

Scenario files are a powerful way to specify a sequence of commands at specific times. When the experiment involves hundreds of nodes, it is not uncommon for the scenario to become large. Note that scenario files **do not** have the ability to chain/order events.

3.2.3 Gumby Module File

Implementation of commands present in the scenario file can be found in the Gumby module files. The `BenalohModule` file can be found here:

https://github.com/qstokkink/gumby/blob/workshop_code/experiments/benaloh/benaloh_module.py

This file contains three classes, the `BenalohCommunityLoader`, the `BenalohCommunityLauncher` and the `BenalohModule`. The `BenalohCommunityLauncher` specifies the launch parameters which Dispersy will use to launch our custom community and optionally any load order requirements in respect to other communities. The `BenalohCommunityLoader` is in charge of managing all of the specified `CommunityLaunchers`, of which we have 1. Finally the `BenalohModule` implements our scenario callbacks and manages the loaded community at runtime.

Let's take a closer look at the implementation of the `share_local` command:

```
1     @experiment_callback
2     def share_local(self):
3         self.community.share_local()
```

The implementation of each scenario file command should be decorated by an `@experiment_callback`. The name of this method must match the command in the scenario file. In this case, we simply call the `share_local` method of the `BenalohCommunity`.

4 Questions and Exercises

Now that you have a basic understanding of Dispersy, Gumby and Jenkins, you can try to answer the following questions and do the exercises.

4.1 Questions

Consider the `benaloh_community.py` file explained in Section 3.1:

1. *Name one reason why the supplied code is not scalable. I.e. what are the bottlenecks if we increase the number of users in the network?*
2. *Find two race conditions in the message delivery which can cause an incorrect algorithm execution.*
3. *Do the two messages defined in the Benaloh community (`local-share` and `broadcast-share`) have a different format when sent over the Internet?*

Consider the `benaloh.scenario` file explained in Section 3.2.2.

1. *Recall that the scenario file used two different modules: `DispersyModule` and `BenalohModule`. Which callbacks present in the scenario file are defined by which module?*
2. *What is the difference between the scenario's* stop *callback and the node timeout defined in the experiment configuration file?*

Consider the `benaloh_module.py` file explained in Section 3.2.3.

1. *How could you load another BenalohCommunity on each node?*
2. *Which node(s) execute this module file?*

4.2 Exercises

This tutorial was built upon an existing experiment but in the end, you should be able to adapt experiments to your needs. For instance, you might want to add more messages or use more resources on the DAS-5 supercomputer. To test your prowess, you will be tasked with performing two modifications of the existing experiment. To perform these tasks you will need the `git` software on your system. For Ubuntu users, this can be done as follows: `sudo apt-get install git`).
To get your own copy of the Gumby framework, go to GitHub and click the "fork" button in the upper-right corner. The Gumby repository can be found here: https://github.com/Tribler/gumby. Note that you should have an account on Github to do so.

If you click the "Clone or download" button on your newly created repository, it will give you the link which you can use to fetch your repository copy. Now you can open a terminal, browse to a folder where you want Gumby to exist on your computer and run the following command:

```
1 git clone <your repository URL>
2 cd gumby
```

This will fetch the repository from Github and store it on your computer. Git will keep track of all the changes you make in this directory (`git status` is a helpful command for that). If you are done editing your folder, add all of your modified files to git using `git add .`. Then think of a good message for your change and commit these added files using `git commit -m "This is my great change"`. Finally you can send your commits to the GitHub server by running `git push`.

Now we should instruct Jenkins to fetch a copy of your repository instead. This can be changed on the configuration page of the job you created in Section 2.1. Change the settings under *Source Code Management* so it points to your repository. Try building the experiment to check whether everything works correctly and the experiment finishes.

4.2.1 Exercise 1: Modify the scenario

Add your own experiment callback `exercise_1` which:

1. adds 1 to the input defined by the scenario file
2. prints the resulting value

4.2.2 Exercise 2: Modify the configuration

Have the experiment use an additional node (note that you should not only change the Gumby configuration file but also make a change in the Dispersy community code).

4.2.3 Bonus: Modify the community

Have the `BenalohCommunity` auto-detect when all other nodes have performed a `local-share` and start the `broadcast-share` automatically. You can then get rid of the `share_subset_sum` callback.

References

[1] Blockchain lab. `http://blockchain-lab.org`, 2017.

[2] Josh Benaloh and Jerry Leichter. Generalized secret sharing and monotone functions. In *Proceedings on Advances in cryptology*, pages 27–35. Springer-Verlag New York, Inc., 1990.

[3] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[4] Niels Zeilemaker, Boudewijn Schoon, and Johan Pouwelse. Dispersy bundle synchronization. *TU Delft, Parallel and Distributed Systems*, 2013.