Dallinger Documentation

Release 2.7.2

Dallinger Development Team

Mar 09, 2017

User Documentation

Laboratory automation for the behavioral and social sciences.

CHAPTER 1

Installation

If you would like to contribute to Dallinger, please follow these *alternative install instructions*.

Install Python

Dallinger is written in the language Python. For it to work, you will need to have Python 2.7 installed. You can check what version of Python you have by running:

```
python --version
```

If you do not have Python 2.7 installed, you can install it from the Python website.

Install Postgres

Dallinger uses Postgres to create local databases. On OS X, install Postgres from postgresapp.com. This will require downloading a zip file, unzipping the file and installing the unzipped application.

You will then need to add Postgres to your PATH environmental variable. If you use the default location for installing applications on OS X (namely `/Applications`), you can adjust your path by running the following command:

```
export PATH="/Applications/Postgres.app/Contents/Versions/9.3/bin:$PATH"
```

NB: If you have installed a more recent version of Postgres (e.g., the the upcoming version 9.4), you may need to alter that command slightly to accommodate the more recent version number. To double check which version to include, then run:

```
ls /Applications/Postgres.app/Contents/Versions/
```

Whatever number that returns is the version number that you should place in the `export` command above. If it does not return a number, you have not installed Postgres correctly in your `/Applications` folder or something else is horribly wrong.

Create the Database

After installing Postgres, you will need to create a database for your experiments to use. Run the following command from the command line:

```
psql -c 'create database dallinger;' -U postgres
```

Install Dallinger

Install Dallinger from the terminal by running

```
pip install dallinger
```

Test that your installation works by running:

```
dallinger --version
```

If you use Anaconda, installing Dallinger probably failed. The problem is that you need to install bindings for the `psycopg2` package (it helps Python play nicely with Postgres) and you must use conda for conda to know where to look for the links. You do this with:

```
conda install psycopg2
```

Then, try the above installation commands. They should work now, meaning you can move on.

Next, you'll need *access keys for AWS, Heroku, etc.*.

CHAPTER 2

Installing Dallinger with Anaconda

If you are interested in Dallinger and use Anaconda, you'll need to adapt the standard instructions slightly.

Install psycopg2

In order to get the correct bindings, you need to install `psycopg2` before you use `requirements.txt`; otherwise, everything will fail and you will be endlessly frustrated.

```
conda install psycopg2
```

Install Dallinger

You'll follow all of the *Dallinger development installation instructions*, **with the exception of the virtual environment step**. Then return here.

Confirm Dallinger works

Now, we need to make sure that Dallinger and Anaconda play nice with one another. At this point, we'd check to make sure that Dallinger is properly installed by typing

```
dallinger --version
```

into the command line. For those of us with Anaconda, we'll get a long error message. Don't panic! Add the following to your `.bash_profile`:

```
export DYLD_FALLBACK_LIBRARY_PATH=$HOME/anaconda/lib/:$DYLD_FALLBACK_LIBRARY_PATH
```

If you installed anaconda using `Python 3`, you will need to change `anaconda` in that path to `anaconda3`.

After you `source` your `.bash_profile`, you can check your Dallinger version (using the same command that we used earlier), which should return the Dallinger version that you've installed.

Re-link Open SSL

Finally, you'll need to re-link `openssl`. Run the following:

```
brew install --upgrade openssl
brew unlink openssl && brew link openssl --force
```

CHAPTER 3

Setting Up AWS, psiTurk, and Heroku

Before you can use Dallinger, you will need accounts with Amazon Web Services, Amazon Mechanical Turk, Heroku, and psiTurk. You will then need to create a configuration file and set up your environment so that Dallinger can access your accounts.

Create the configuration file

The first step is to create the Dallinger configuration file in your home directory. You can do this using the Dallinger command-line utility through

```
dallinger setup
```

which will prepopulate a hidden file `.dallingerconfig` in your home directory. Alternatively, you can create this file yourself and fill it in like so:

```
[AWS Access]
aws_access_key_id = ???
aws_secret_access_key = ???
aws_region = us-east-1

[psiTurk Access]
psiturk_access_key_id = ???
psiturk_secret_access_id = ???

[Heroku Access]
heroku_email_address = ???
heroku_password = ???

[Email Access]
dallinger_email_address = ???
dallinger_email_password = ???

[Task Parameters]
```

```
experiment_code_version = 1.0
num_conds = 1
num_counters = 1

[Server Parameters]
port = 5000
cutoff_time = 30
logfile = -
loglevel = 0
debug = true
login_username = examplename
login_pw = examplepassword
threads = 1
clock_on = true
```

In the next steps, we'll fill in your config file with keys.

Amazon Web Services API Keys

You can get API keys for Amazon Web Services by following these instructions.

Then fill in the following lines of `.dallingerconfig`, replacing `???` with your keys:

```
[AWS Access]
aws_access_key_id = ???
aws_secret_access_key = ???
```

N.B. One feature of AWS API keys is that they are only displayed once, and though they can be regenerated, doing so will render invalid previously generated keys. If you are running experiments using a laboratory account (or any other kind of group-owned account), regenerating keys will stop other users who have previously generated keys from being able to use the AWS account. Unless you are sure that you will not be interrupting others' workflows, it is advised that you do **not** generate new API keys. If you are not the primary user of the account, see if you can obtain these keys from others who have successfully used AWS.

Amazon Mechanical Turk

It's worth signing up for Amazon Mechanical Turk (perhaps using your AWS account from above), both as a requester and as a worker. You'll use this to test and monitor experiments. You should also sign in to each sandbox, requester and worker using the same account. Store this account and password somewhere, but you don't need to tell it to Dallinger.

psiTurk

Next, create an account on psiTurk, which will require a valid email address. Once you confirm your account, click on **API Keys**, which will allow you to access your API keys as seen in the image below:

Fig. 3.1: Don't even try to use these API Keys, they've been reissued!

Place these credential in the `.dallingerconfig` file:

Then fill in the following lines of `.dallingerconfig`, replacing `???` with your keys:

```
[psiTurk Access]
psiturk_access_key_id = ???
psiturk_secret_access_id = ???
```

Heroku

Next, sign up for Heroku and install the Heroku toolbelt.

You should see an interface that looks something like the following:

Fig. 3.2: This is the interface with the Heroku app

Then, log in from the command line:

```
heroku login
```

And fill in the appropriate section of `.dallingerconfig`:

```
[Heroku Access]
heroku_email_address = ???
heroku_password = ???
```

Done?

Done. You're now all set up with the tools you need to work with Dallinger.

Next, we'll *test Dallinger to make sure it's working on your system*.

CHAPTER 4

Demoing Dallinger

First, make sure you have Dallinger installed:

- *Installation*
- *Developer Installation*

To test out Dallinger, we'll run a demo experiment in debug mode. First download the Bartlett (1932) demo and unzip it. Then run Dallinger in debug mode from within that demo directory:

```
dallinger debug
```

You will see some output as Dallinger loads. When it is finished, you will see something that looks like:

```
Now serving on http://0.0.0.0:5000
[psiTurk server:on mode:sdbx #HITs:4]$
```

This is the psiTurk prompt. Into that prompt type:

```
debug
```

This will cause the experiment to open in a new window in your browser. Alternatively, type

```
debug --print-only
```

to get the URL of the experiment so that you can view it on a different machine than the one you are serving it on.

Once you have finished running through the experiment as a participant, you can type `debug` again to play as the next participant.

Help, the experiment page is blank! This may happen if you are using an ad-blocker. Try disabling your ad-blocker and refresh the page.

CHAPTER 5

Learning to Use Dallinger

Beginner

Key concepts in Dallinger

- *Database API*
- *The Experiment Class*

Dallinger as a web app

- *Communicating With the Server*
- *Web API*

Experimental design

- *Required Experimental Files*
- config.txt
- Dallinger.js

Example walkthroughs

- Bartlett1932 walkthrough

Intermediate

Experimental design

- Networks
- Nodes
- Infos
- Transformations
- Using properties 1 through 5
- Processes
- Failing

Running experiments

- *Command-Line Utility*
- Debugging

Advanced

Experimental design

- Changing route behavior and making new routes
- Sending requests from within Dallinger

Running experiments

- Writing automated tests
- Compensating workers
- Monitoring a live experiment
- Recruiters

CHAPTER 6

Monitoring a Live Experiment

There are a number of ways that you can monitor a live experiment:

Command line tools

`dallinger summary --app {#id}`, where `{#id}` is the id (`w...`) of the application.

This will print a summary showing the number of participants with each status code, as well as the overall yield:

```
status  | count
----------------
1   | 26
101 | 80
103 | 43
104 | 2

Yield: 64.00%
```

Papertrail

You can use Papertrail to view and search the live logs of your experiment. You can access the logs either through the Heroku dashboard's Resources panel (https://dashboard.heroku.com/apps/{#id}/resources), where {#id} is the id of your experiment, or directly through Papertrail.com (https://papertrailapp.com/systems/{#id}/events).

Setting up alerts

You can set up Papertrail to send error notifications to Slack or another communications platform.

0. Take a deep breath.
1. Open the Papertrail logs.

2. Search for the term `error`.
3. To the right of the search bar, you will see a button titled "+ Save Search". Click it. Name the search "Errors". Then click "Save & Setup an Alert", which is to the right of "Save Search".
4. You will be directed to a page with a list of services that you can use to set up an alert.
5. Click, e.g., Slack.
6. Choose the desired frequency of alert. We recommend the minimum, 1 minute.
7. Under the heading "Slack details", open (*in a new tab or window*) the link new Papertrail integration.
8. This will bring you to a Slack page where you will choose a channel to post to. You may need to log in.
9. Select the desired channel.
10. Click "Add Papertrail Integration".
11. You will be brought to a page with more information about the integration.
12. Scroll down to Step 3 to get the Webhook URL. It should look something like `https://hooks.slack.com/services/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`.
13. Copy this link to your clipboard.
14. Change anything else you want and then scroll to the bottom and click "Save integration".
15. Go back to Papertrail page that you left in Step 7.
16. Paste the copied URL into the input text box labeled "Integration's Webhook URL" under the "Slack Details" heading.
17. Click "Create Alert" on the same page.
18. Victory.

CHAPTER 7

Viewing the PostgreSQL Database

Postico is a nice tool for examining Postgres databases on OS X. We use it to connect to live experiment databases. Here are the steps needed to do this:

1. Download Postico and place it in your Applications folder.
2. Open Postico.
3. Press the "New Favorite" button in the bottom left corner to access a new database.
4. Get the database credentials from the Heroku dashboard:
    - Go to https://dashboard.heroku.com/apps/{app_id}/resources
    - Under the **Add-ons** subheading, go to "Heroku Postgres :: Database"
    - Note the database credentials under the subheading "Connection Settings". You'll use these in step 5.
5. Fill in the database settings in Postico. You'll need to include the:
    - Host
    - Port
    - User
    - Password
    - Database
6. Connect to the database.
    - You may see a dialog box pop up saying that Postico cannot verify the identity of the server. Click "Connect" to proceed.

CHAPTER 8

Command-Line Utility

Dallinger is executed from the command line within the experiment directory with the following commands:

verify

Verify that a directory is a Dallinger-compatible app.

debug

Run the experiment locally. An optional `--verbose` flag prints more detailed logs to the command line.

sandbox

Runs the experiment on MTurk's sandbox using Heroku as a server. An optional `--verbose` flag prints more detailed logs to the command line.

deploy

Runs the experiment live on MTurk using Heroku as a server. An optional `--verbose` flag prints more detailed logs to the command line.

logs

Open the app's logs in Papertrail. A required `--app <app>` flag specifies the experiment by its id.

summary

Return a summary of an experiment. A required `--app <app>` flag specifies the experiment by its id.

export

Download the database and partial server logs to a zipped folder within the data directory of the experimental folder. Databases are stored in CSV format. A required `--app <app>` flag specifies the experiment by its id.

summary

Print a summary of the participant table to the command line. A required `--app <app>` flag specifies the experiment by its id.

qualify

Assign qualification to a worker. Requires a qualification id `qualification_id`, value `value`, and worker id `worker_id`. This is useful when compensating workers if something goes wrong with the experiment.

hibernate

Temporarily scales down the specified app to save money. All dynos are removed and so are many of the add-ons. Hibernating apps are non-functional. It is likely that the app will not be entirely free while hibernating. To restore the app use `awaken`. A required `--app <app>` flag specifies the experiment by its id.

awaken

Restore a hibernating app. A required `--app <app>` flag specifies the experiment by its id.

destroy

Tear down an experiment server. A required `--app <app>` flag specifies the experiment by its id.

CHAPTER 9

2048

2048 is a sliding-block puzzle game by the Italian web developer Gabriele Cirulli. The goal is to slide numbered tiles on a grid, combining them to create a tile with a value of 2048.

Download the demo.

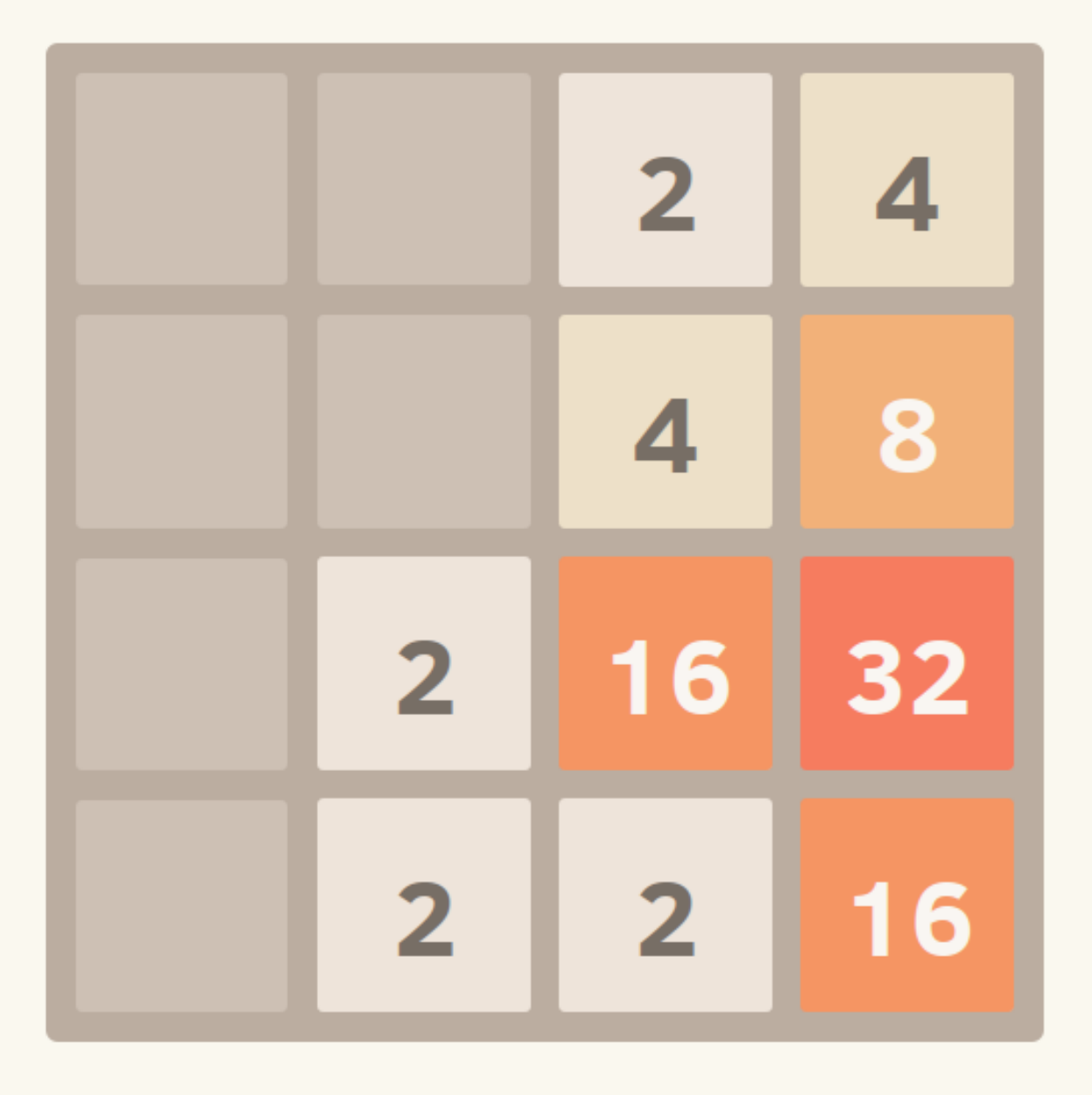


Fig. 9.1: Screenshot of an in-progress 2048 game

CHAPTER 10

Bartlett (1932), stories

Frederic Bartlett's 1932 book *Remembering* documents early experiments that explore how using and transmitting a memory can affect the memory's contents. Bartlett wanted to understand how culture shapes memory. Inspired by Philippe (1897), he performed a series of experiments that asked participants to repeatedly recall a memory or to pass it down a chain of people, from one to the next. Bartlett showed that the process of reproduction alters memories over time, causing them to take on features from an individual's culture. More generally, the methods he developed expose cumulative effects of the forces that reshape and degrade memories and how they impact the structure and veracity of what we remember.

Bartlett, F. C. (1932). Remembering. Cambridge: Cambridge University Press.

In this demo, a story is passed down a chain.

Download the demo.

CHAPTER 11

Networked chatroom-based coordination game

This is a networked coordination game where players broadcast messages to each other and try to make the same decision as others.

Download the demo.

CHAPTER 12

Concentration

The objective of Concentration is to flip and match all the turned-down cards in as few moves as possible.

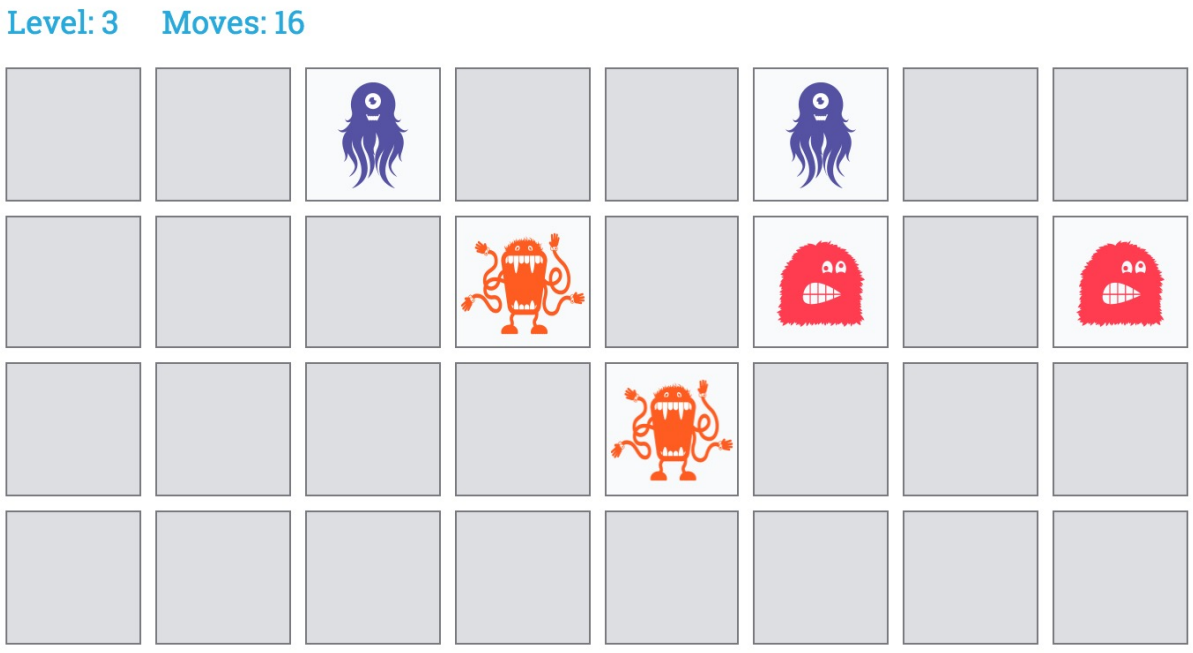


Fig. 12.1: Screenshot of an in-progress Concentration game

Download the demo.

CHAPTER 13

Transmitting functions

Culturally transmitted knowledge changes as it is transmitted from person to person. Some of the most striking instances of this process come from cases of language acquisition. For example, in Nicaragua, a community of deaf children transformed a fragmentary pidgin into a language with rich grammatical structure by learning from each other (Kegl and Iwata, 1989; Senghas and Coppola, 2001). Languages, legends, and social norms are all shaped by the processes of cultural transmission (Cavalli-Sforza, 1981; Boyd and Richerson, 1988; Kirby, 1999, 2001; Briscoe, 2002).

Laboratory studies of cultural transmission often use the method of "iterated learning", which has roots in Bartlett's experiments. In the iterated learning paradigm, information is passed along a chain of individuals, from one to the next, much like in the children's game Telephone. Iterated learning paradigms for the transmission of language and other forms of knowledge have been developed, too (Kalish et al., 2007; Griffiths and Kalish, 2007; Griffiths et al., 2008a). For example, in one study, participants learned the relationship between two continuous variables ("function learning") and were tested on what they had discovered (Kalish et al., 2007). Responses on the test were then used to train the next participant in the chain. Kalish et al. (2007) found that, over time, knowledge transmitted through the chain reverts to the prior beliefs of the individual learners.

Kalish, M. L., Griffiths, T. L., & Lewandowsky, S. (2007). Iterated learning: Intergenerational knowledge transmission reveals inductive biases. Psychonomic Bulletin and Review, 14, 288-294.

Download the demo.

CHAPTER 14

Bartlett (1932), drawings

Frederic Bartlett's 1932 book *Remembering* documents early experiments that explore how using and transmitting a memory can affect the memory's contents. Bartlett wanted to understand how culture shapes memory. Inspired by Philippe (1897), he performed a series of experiments that asked participants to repeatedly recall a memory or to pass it down a chain of people, from one to the next. Bartlett showed that the process of reproduction alters memories over time, causing them to take on features from an individual's culture. More generally, the methods he developed expose cumulative effects of the forces that reshape and degrade memories and how they impact the structure and veracity of what we remember.

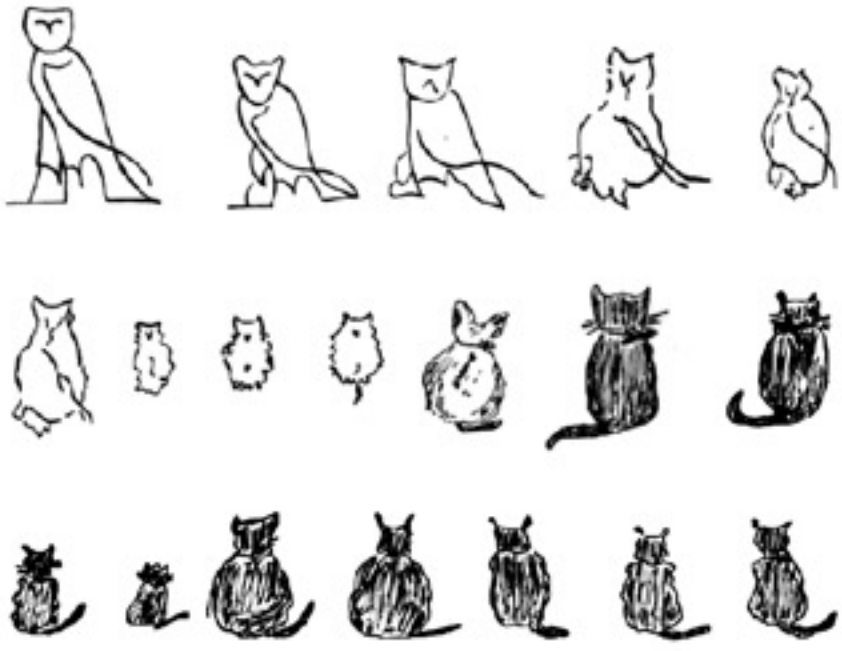


Fig. 14.1: Bartlett's drawing experiment

Bartlett, F. C. (1932). Remembering. Cambridge: Cambridge University Press.

In this demo, a drawing is passed down a chain.

Download the demo.

CHAPTER 15

Markov Chain Monte Carlo with People

Markov Chain Monte Carlo with People (MCMCP) is a method for uncovering mental representations that exploits an equivalence between a model of human choice behavior and an element of an MCMC algorithm. This demo replicates Experiment 3 of Sanborn, Griffiths, & Shiffrin (2010), which applies MCMCP to four natural categories, providing estimates of the distributions over animal shapes that people associate with giraffes, horses, cats, and dogs.

Sanborn, A. N., Griffiths, T. L., & Shiffrin, R. M. (2010). Uncovering mental representations with Markov chain Monte Carlo. *Cognitive Psychology*, 60(2), 63-106.

Download the demo.

CHAPTER 16

Rogers' Paradox

This experiment, which demonstrates Rogers paradox, explores the evolution of asocial learning and unguided social learning in the context of a numerical discrimination task.

Download the demo.

CHAPTER 17

The Sheep Market

"The Sheep Market is a collection of 10,000 sheep created by workers on Amazon's Mechanical Turk. Each worker was paid $.02 (US) to "draw a sheep facing left."

http://www.aaronkoblin.com/project/the-sheep-market/

Download the demo.

CHAPTER 18

Snake

This is the video game Snake, in which the player maneuvers a line which grows in length within the bounds of a box, with the line itself being a primary obstacle.

Download the demo.

CHAPTER 19

Vox Populi (Wisdom of the crowd)

https://en.wikipedia.org/wiki/Wisdom_of_the_crowd

Download the demo.

CHAPTER 20

Developer Installation

We recommend installing Dallinger on Mac OS X. It's also possible to use Ubuntu.

Install Python 2.7

You will need Python 2.7. You can check what version of Python you have by running:

```
python --version
```

If you do not have Python 2.7 installed, you can install it from the Python website.

Or, if you use Homebrew:

```
brew install python
```

Or, if you use Anaconda, install using `conda`, not Homebrew.

If you have Python 3.*x* installed and and symlinked to the command `python`, you will need to create a `virtualenv` that interprets the code as `python2.7` (for compatibility with the `psiturk` module). Fortunately, we will be creating a virtual environment anyway, so as long as you run `brew install python` and you don't run into any errors because of your symlinks, then you can proceed with the instructions. If you do run into any errors, good luck, we're rooting for you.

Install Postgres

On OS X, we recommend installing Postgres.app to start and stop a Postgres server. You'll also want to set up the Postgres command-line utilities by following the instructions here.

You will then need to add Postgres to your PATH environmental variable. If you use the default location for installing applications on OS X (namely `/Applications`), you can adjust your path by running the following command:

```
export PATH="$PATH:/Applications/Postgres.app/Contents/Versions/latest/bin"
```

NB: If you have installed an older version of Postgres (e.g., < 9.5), you may need to alter that command to accommodate the more recent version number. To double check which version to include, run:

```
ls /Applications/Postgres.app/Contents/Versions/
```

Whatever values that returns are the versions that you should place in the `export` command above in the place of `latest`.

If it does not return a number, you have not installed Postgres correctly in your `/Applications` folder or something else is horribly wrong.

On Ubuntu, follow the instructions under the heading "Installation" here.

Create the Database

After installing Postgres, you will need to create a database for your experiments to use. First, open the Postgres.app. Then, run the following command from the command line:

```
psql -c 'create database dallinger;' -U postgres
```

If you get the following error...

```
psql: could not connect to server: No such file or directory
    Is the server running locally and accepting
    connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

...then you probably did not start the app.

If you get the following error...

```
dyld: Library not loaded: /usr/local/opt/readline/lib/libreadline.6.dylib
    Referenced from: /usr/local/bin/psql
    Reason: image not found
Abort trap: 6
```

... then type the following command into the command line:

```
createdb -U postgres dallinger
```

(This error may arise if you are running Python 2.7 with Anaconda within a virtual environment.)

Set up a virtual environment

Note: if you are using Anaconda, ignore this `virtualenv` section; use `conda` to create your virtual environment. Or, see the special *Anaconda installation instructions*.

Set up a virtual environment by running the following commands:

```
pip install virtualenv
pip install virtualenvwrapper
export WORKON_HOME=$HOME/.virtualenvs
mkdir -p $WORKON_HOME
```

```
source $(which virtualenvwrapper.sh)
mkvirtualenv dallinger --python /usr/local/bin/python2.7
```

These commands use `pip`, the Python package manager, to install two packages `virtualenv` and `virtualenvwrapper`. They set up an environmental variable named `WORKON_HOME` with a string that gives a path to a subfolder of your home directory (`~`) called `Envs`, which the next command (`mkdir`) then makes according to the path described in `$WORKON_HOME` (recursively, due to the `-p` flag). That is where your environments will be stored. The `source` command will run the command that follows, which in this case locates the `virtualenvwrapper.sh` shell script, the contents of which are beyond the scope of this setup tutorial. If you want to know what it does, a more in depth description can be found on the documentation site for virtualenvwrapper.

Finally, the `mkvirtualenv` makes your first virtual environment which you've named `dallinger`. We have explicitly passed it the location of `python2.7` so that even if your `python` command has been remapped to `python3`, it will create the environment with `python2.7` as its interpreter.

In the future, you can work on your virtual environment by running:

```
source $(which virtualenvwrapper.sh)
workon dallinger
```

NB: To stop working on the virtual environment, run `deactivate`. To list all available virtual environments, run `workon` with no arguments.

Install prerequisites for building documentation

To be able to build the documentation, you will need:

- pandoc. Please follow the instructions here to install it.
- the Enchant library. Please follow the instructions here to install it.

Install Dallinger

Next, navigate to the directory where you want to house your development work on Dallinger. Once there, clone the Git repository using:

```
git clone https://github.com/Dallinger/Dallinger
```

This will create a directory called `Dallinger` in your current directory.

Change into your the new directory and make sure you are still in your virtual environment before installing the dependencies. If you want to be extra careful, run the command `workon dallinger`, which will ensure that you are in the right virtual environment.

Note: if you are using Anaconda – as of August 10, 2016 – you will need to follow special *Anaconda installation instructions*. This should be fixed in future versions.

```
cd Dallinger
```

Now we need to install the dependencies using pip:

```
pip install -r dev-requirements.txt
```

Next run `setup.py` with the argument `develop`:

```
python setup.py develop
```

Test that your installation works by running:

```
dallinger --version
```

Note: if you are using Anaconda and get a long traceback here, please see the special *Installing Dallinger with Anaconda*.

Next, you'll need *access keys for AWS, Heroku, etc.*.

CHAPTER 21

Running the tests

If you push a commit to a branch in the Dallinger organization on GitHub, or open a pull request from your own fork, Dallinger's automated code tests will be run on Travis.

Current build status:

The tests include:

- Making sure that a source distribution of the Python package can be created.
- Running flake8 to make sure Python code conforms to the PEP 8 style guide.
- Running the tests for the Python code using nose and making sure they pass in Python 2.7.
- Making sure that code coverage for the Python code is above the desired threshold.
- Making sure the docs build without error.

You can also run all these tests locally, simply by running:

```
tox
```

To run just the Python tests:

```
nosetests
```

To build documentation:

```
tox -e docs
```

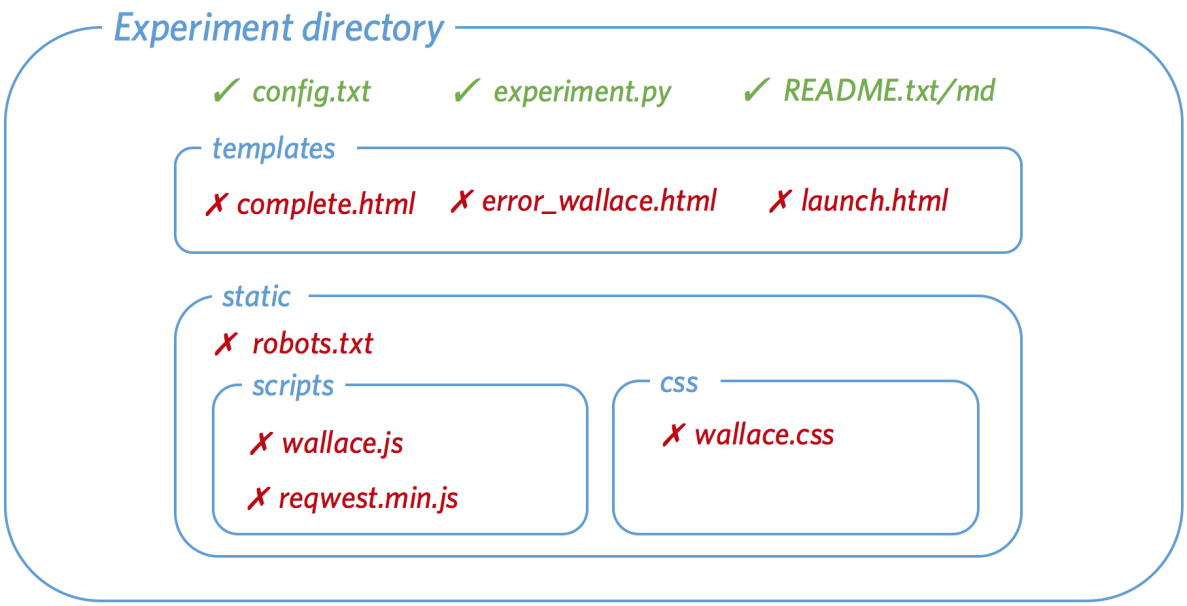
To run flake8:

```
flake8
```

CHAPTER 22

Required Experimental Files

Dallinger is flexible with regards to the form the front end takes. However, there are a number of required or forbidden files. You can verify that a directory is compatible by running the *verify* command from a terminal within the directory. Though just because these checks pass doesn't mean the experiment will run! The minimal required structure is as follows:



Blue items are (optional) directories (note that the experiment directory can have any name), green items are required files (the README file can be either a txt file or a md file), and red items are forbidden files that will cause a conflict at run time.

Required files

- config.txt - The config file contains a variety of parameters that affect how Dallinger runs. For more info see...
- experiment.py - This is a python file containing the custom experiment code.
- README.txt/md - This (hopefully) contains a helpful description of the experiment.

Forbidden files

A number of files cannot be included in the experiment directory. This is because, when Dallinger runs, it inserts a number of required files into the experiment directory and will overwrite any files with the same name. The files are as follows:

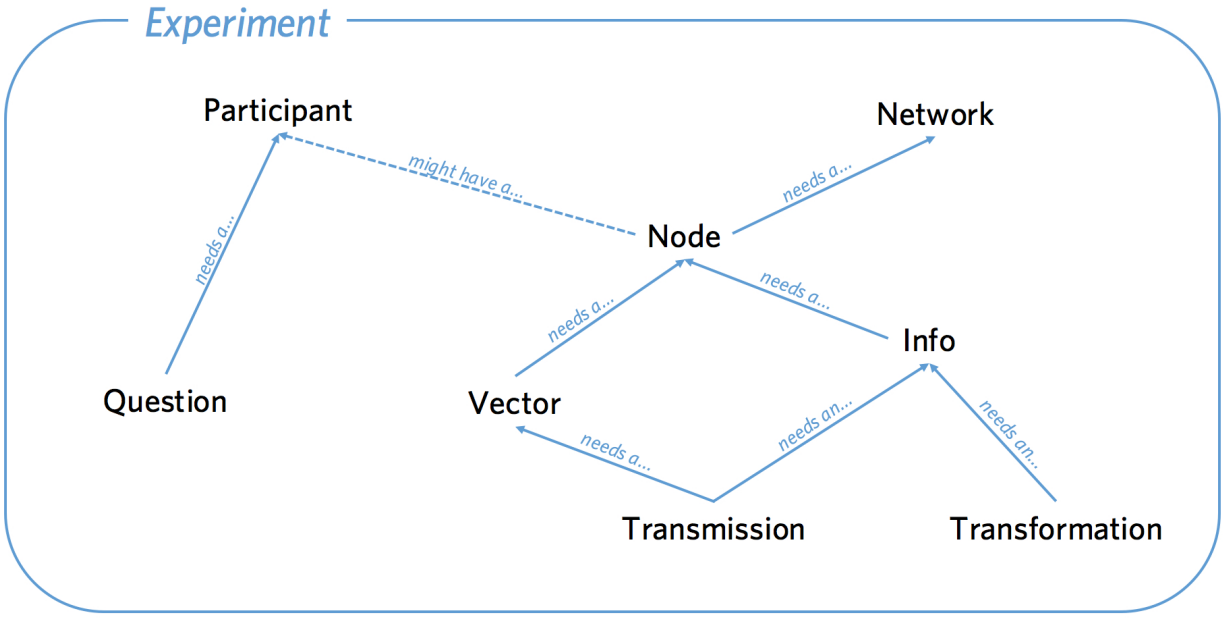
- complete.html - this html page shows when dallinger is run in debug mode and the experiment is complete.
- error_dallinger.html - this is a flexible error page that shows when something goes wrong.
- launch.html - this page is shown when the /launch route is pinged and the experiment starts successfully.
- robots.txt - this file is returned to bots (e.g. from Google) that bump into the experiment when crawling the internet.
- dallinger.js - this is a javascript library with a number of helpful functions.
- reqwest.min.js - this is required for dallinger.js to work.
- dallinger.css - this contains several css classes that are used in the demos.

CHAPTER 23

Database API

The classes involved in a Dallinger experiment are: *Network*, *Node*, *Vector*, *Info*, *Transmission*, *Transformation*, *Participant*, and *Question*. The code for all these classes can be seen in `models.py`. Each class has a corresponding table in the database, with each instance stored as a row in the table. Accordingly, each class is defined, in part, by the columns that constitute the table it is stored in. In addition, the classes have relationships to other objects and a number of functions.

The classes have relationships to each other as shown in the diagram below. Be careful to note which way the arrows point. A *Node* is a point in a *Network* that might be associated with a *Participant*. A *Vector* is a directional connection between a *Node* and another *Node*. An *Info* is information created by a *Node*. A *Transmission* is an instance of an *Info* being sent along a *Vector*. A *Transformation* is a relationship between an *Info* and another *Info*. A *Question* is a survey response created by a *Participant*.



SharedMixin

All Dallinger classes inherit from a `SharedMixin` which provides multiple columns that are common across tables:

`SharedMixin.`**`id`**
: a unique number for every entry. 1, 2, 3 and so on...

`SharedMixin.`**`creation_time`**
: the time at which the Network was created.

`SharedMixin.`**`property1`**
: a generic column that can be used to store experiment-specific details in String form.

`SharedMixin.`**`property2`**
: a generic column that can be used to store experiment-specific details in String form.

`SharedMixin.`**`property3`**
: a generic column that can be used to store experiment-specific details in String form.

`SharedMixin.`**`property4`**
: a generic column that can be used to store experiment-specific details in String form.

`SharedMixin.`**`property5`**
: a generic column that can be used to store experiment-specific details in String form.

`SharedMixin.`**`failed`**
: boolean indicating whether the Network has failed which prompts Dallinger to ignore it unless specified otherwise. Objects are usually failed to indicate something has gone wrong.

`SharedMixin.`**`time_of_death`**
: the time at which failing occurred

Network

The *`Network`* object can be imagined as a set of other objects with some functions that perform operations over those objects. The objects that *`Network`*'s have direct access to are all the *`Node`*'s in the network, the *`Vector`*'s between those Nodes, Infos created by those Nodes, Transmissions sent along the Vectors by those Nodes and Transformations of those Infos. Participants and Questions do not exist within Networks. An experiment may involve multiple Networks, Transmissions can only occur within networks, not between them.

class `dallinger.models.`**`Network`**`(`***kwargs*`)`
: Contains and manages a set of Nodes and Vectors etc.

Columns

`Network.`**`type`**
: A String giving the name of the class. Defaults to "network". This allows subclassing.

`Network.`**`max_size`**
: How big the network can get, this number is used by the full() method to decide whether the network is full

`Network.`**`full`**
: Whether the network is currently full

`Network.`**`role`**
: The role of the network. By default dallinger initializes all networks as either "practice" or "experiment"

Relationships

dallinger.models.Network.**all_nodes**
All the Nodes in the network.

dallinger.models.Network.**all_vectors**
All the vectors in the network.

dallinger.models.Network.**all_infos**
All the infos in the network.

dallinger.models.Network.**networks_transmissions**
All the transmissions int he network.

dallinger.models.Network.**networks_transformations**
All the transformations in the network.

Methods

Network.**__repr__**()
The string representation of a network.

Network.**__json__**()
Return json description of a participant.

Network.**calculate_full**()
Set whether the network is full.

Network.**fail**()
Fail an entire network.

Network.**infos**(*type=None*, *failed=False*)
Get infos in the network.

type specifies the type of info (defaults to Info). failed { False, True, "all" } specifies the failed state of the infos. To get infos from a specific node, see the infos() method in class *Node*.

Network.**latest_transmission_recipient**()
Get the node that most recently received a transmission.

Network.**nodes**(*type=None*, *failed=False*, *participant_id=None*)
Get nodes in the network.

type specifies the type of Node. Failed can be "all", False (default) or True. If a participant_id is passed only nodes with that participant_id will be returned.

Network.**print_verbose**()
Print a verbose representation of a network.

Network.**size**(*type=None*, *failed=False*)
How many nodes in a network.

type specifies the class of node, failed can be True/False/all.

Network.**transformations**(*type=None*, *failed=False*)
Get transformations in the network.

type specifies the type of transformation (default = Transformation). failed = { False, True, "all" }

To get transformations from a specific node, see Node.transformations().

Network.**transmissions**(*status='all'*, *failed=False*)
Get transmissions in the network.

status { "all", "received", "pending" } failed { False, True, "all" } To get transmissions from a specific vector, see the transmissions() method in class Vector.

Network.**vectors**(*failed=False*)
Get vectors in the network.

failed = { False, True, "all" } To get the vectors to/from to a specific node, see Node.vectors().

Node

Each Node represents a single point in a single network. A Node must be within a Network and may also be associated with a Participant.

class dallinger.models.**Node**(*network*, *participant=None*)
A point in a network.

Columns

Node.**type**
A String giving the name of the class. Defaults to node. This allows subclassing.

Node.**network_id**
the id of the network that this node is a part of

Node.**participant_id**
the id of the participant whose node this is

Relationships

Node.**network**
the network the node is in

Node.**participant**
the participant the node is associated with

dallinger.models.Node.**all_outgoing_vectors**
All the vectors going out from this Node.

dallinger.models.Node.**all_incoming_vectors**
All the vectors coming in to this Node.

dallinger.models.Node.**all_infos**
All Infos created by this Node.

dallinger.models.Node.**all_outgoing_transmissions**
All Transmissions sent from this Node.

dallinger.models.Node.**all_incoming_transmissions**
All Transmissions sent to this Node.

dallinger.models.Node.**transformations_here**
All transformations that took place at this Node.

Methods

Node.**__repr__**()
The string representation of a node.

Node.**__json__**()
The json of a node.

Node.**_to_whom**()
To whom to transmit if to_whom is not specified.

Return the default value of `to_whom` for *transmit()*. Should not return None or a list containing None.

Node.**_what**()
What to transmit if what is not specified.

Return the default value of `what` for *transmit()*. Should not return None or a list containing None.

Node.**connect**(*whom*, *direction='to'*)
Create a vector from self to/from whom.

Return a list of newly created vector between the node and whom. `whom` can be a specific node or a (nested) list of nodes. Nodes can only connect with nodes in the same network. In addition nodes cannot connect with themselves or with Sources. `direction` specifies the direction of the connection it can be "to" (node -> whom), "from" (whom -> node) or both (node <-> whom). The default is "to".

Whom may be a (nested) list of nodes.

Will raise an error if:

1. whom is not a node or list of nodes
2. whom is/contains a source if direction is to or both
3. whom is/contains self
4. whom is/contains a node in a different network

If self is already connected to/from whom a Warning is raised and nothing happens.

This method returns a list of the vectors created (even if there is only one).

Node.**fail**()
Fail a node, setting its status to "failed".

Also fails all vectors that connect to or from the node. You cannot fail a node that has already failed, but you can fail a dead node.

Set node.failed to True and `time_of_death` to now. Instruct all not-failed vectors connected to this node, infos made by this node, transmissions to or from this node and transformations made by this node to fail.

Node.**is_connected**(*whom*, *direction='to'*, *failed=None*)
Check whether this node is connected [to/from] whom.

whom can be a list of nodes or a single node. direction can be "to" (default), "from", "both" or "either".

If whom is a single node this method returns a boolean, otherwise it returns a list of booleans

Node.**infos**(*type=None*, *failed=False*)
Get infos that originate from this node.

Type must be a subclass of *Info*, the default is `Info`. Failed can be True, False or "all".

Node.**mutate**(*info_in*)
Replicate an info + mutation.

To mutate an info, that info must have a method called `_mutated_contents`.

`Node.`**`neighbors`**(*type=None*, *direction='to'*, *failed=None*)
Get a node's neighbors - nodes that are directly connected to it.

Type specifies the class of neighbour and must be a subclass of Node (default is Node). Connection is the direction of the connections and can be "to" (default), "from", "either", or "both".

`Node.`**`receive`**(*what=None*)
Receive some transmissions.

Received transmissions are marked as received, then their infos are passed to update().

"what" can be:

1.None (the default) in which case all pending transmissions are received.

2.a specific transmission.

Will raise an error if the node is told to receive a transmission it has not been sent.

`Node.`**`received_infos`**(*type=None*, *failed=None*)
Get infos that have been sent to this node.

Type must be a subclass of info, the default is Info.

`Node.`**`replicate`**(*info_in*)
Replicate an info.

`Node.`**`transformations`**(*type=None*, *failed=False*)
Get Transformations done by this Node.

type must be a type of Transformation (defaults to Transformation) Failed can be True, False or "all"

`Node.`**`transmissions`**(*direction='outgoing'*, *status='all'*, *failed=False*)
Get transmissions sent to or from this node.

Direction can be "all", "incoming" or "outgoing" (default). Status can be "all" (default), "pending", or "received". failed can be True, False or "all"

`Node.`**`transmit`**(*what=None*, *to_whom=None*)
Transmit one or more infos from one node to another.

"what" dictates which infos are sent, it can be:

1. None (in which case the node's _what method is called).
2. an Info (in which case the node transmits the info)
3. a subclass of Info (in which case the node transmits all its infos of that type)
4. a list of any combination of the above

"to_whom" dictates which node(s) the infos are sent to, it can be:

1. None (in which case the node's _to_whom method is called)
2. a Node (in which case the node transmits to that node)
3. a subclass of Node (in which case the node transmits to all nodes of that type it is connected to)
4. a list of any combination of the above

Will additionally raise an error if:

1. _what() or _to_whom() returns None or a list containing None.
2. what is/contains an info that does not originate from the transmitting node

3. to_whom is/contains a node that the transmitting node does not have a not-failed connection with.

Node.**update**(*infos*)
Process received infos.

Update controls the default behavior of a node when it receives infos. By default it does nothing.

Node.**vectors**(*direction='all'*, *failed=False*)
Get vectors that connect at this node.

Direction can be "incoming", "outgoing" or "all" (default). Failed can be True, False or all

Vector

A vector is a directional link between two nodes. Nodes connected by a vector can send Transmissions to each other, but because Vectors have a direction, two Vectors are needed for bi-directional Transmissions.

class dallinger.models.**Vector**(*origin*, *destination*)
A directed path that links two Nodes.

Nodes can only send each other information if they are linked by a Vector.

Columns

Vector.**origin_id**
the id of the Node at which the vector originates

Vector.**destination_id**
the id of the Node at which the vector terminates.

Vector.**network_id**
the id of the network the vector is in.

Relationships

Vector.**origin**
the Node at which the vector originates.

Vector.**destination**
the Node at which the vector terminates.

Vector.**network**
the network the vector is in.

dallinger.models.Vector.**all_transmissions**
All Transmissions sent along the Vector.

Methods

Vector.**__repr__**()
The string representation of a vector.

Vector.**__json__**()
The json representation of a vector.

Vector.**fail**()
Fail a vector.

Vector.**transmissions**(*status='all'*)
Get transmissions sent along this Vector.

Status can be "all" (the default), "pending", or "received".

Info

An Info is a piece of information created by a Node. It can be sent along Vectors as part of a Transmission.

class dallinger.models.**Info**(*origin*, *contents=None*)
A unit of information.

Columns

Info.**id**

Info.**creation_time**

Info.**property1**

Info.**property2**

Info.**property3**

Info.**property4**

Info.**property5**

Info.**failed**

Info.**time_of_death**

Info.**type**
a String giving the name of the class. Defaults to "info". This allows subclassing.

Info.**origin_id**
the id of the Node that created the info

Info.**network_id**
the id of the network the info is in

Info.**contents**
the contents of the info. Must be stored as a String.

Relationships

Info.**origin**
the Node that created the info.

Info.**network**
the network the info is in

dallinger.models.Info.**all_transmissions**
All Transmissions of this Info.

dallinger.models.Info.**transformation_applied_to**
All Transformations of which this info is the info_in

dallinger.models.Info.**transformation_whence**
All Transformations of which this info is the info_out

Methods

Info.**__repr__**()
The string representation of an info.

Info.**__json__**()
The json representation of an info.

Info.**_mutated_contents**()
The mutated contents of an info.

When an info is asked to mutate, this method will be executed in order to determine the contents of the new info created.

The base class function raises an error and so must be overwritten to be used.

Info.**fail**()
Fail an info.

Set info.failed to True and *time_of_death* to now. Instruct all transmissions and transformations involving this info to fail.

Info.**transformations**(*relationship='all'*)
Get all the transformations of this info.

Return a list of transformations involving this info. relationship can be "parent" (in which case only transformations where the info is the info_in are returned), "child" (in which case only transformations where the info is the info_out are returned) or all (in which case any transformations where the info is the info_out or the info_in are returned). The default is all

Info.**transmissions**(*status='all'*)
Get all the transmissions of this info.

status can be all/pending/received.

Transmission

A transmission represents an instance of an Info being sent along a Vector. Transmissions are not necessarily received when they are sent (like an email) and must also be received by the Node they are sent to.

class dallinger.models.**Transmission**(*vector*, *info*)
An instance of an Info being sent along a Vector.

Columns

Transmission.**origin_id**
the id of the Node that sent the transmission

Transmission.**destination_id**
the id of the Node that the transmission was sent to

Transmission.**vector_id**
the id of the vector the info was sent along

Transmission.**network_id**
the id of the network the transmission is in

Transmission.**info_id**
the id of the info that was transmitted

Transmission.**receive_time**
the time at which the transmission was received

Transmission.**status**
the status of the transmission, can be "pending", which means the transmission has been sent, but not received; or "received", which means the transmission has been sent and received

Relationships

Transmission.**origin**
the Node that sent the transmission.

Transmission.**destination**
the Node that the transmission was sent to.

Transmission.**vector**
the vector the info was sent along.

Transmission.**network**
the network the transmission is in.

Transmission.**info**
the info that was transmitted.

Methods

Transmission.**__repr__**()
The string representation of a transmission.

Transmission.**__json__**()
The json representation of a transmissions.

Transmission.**fail**()
Fail a transmission.

Transmission.**mark_received**()
Mark a transmission as having been received.

Transformation

A Transformation is a relationship between two Infos. It is similar to how a Vector indicates a relationship between two Nodes, but whereas a Vector allows Nodes to Transmit to each other, Transformations don't allow Infos to do anything new. Instead they are a form of book-keeping allowing you to keep track of relationships between various Infos.

class dallinger.models.**Transformation**(*info_in*, *info_out*)
An instance of one info being transformed into another.

Columns

`Transformation.`**`type`**
a String giving the name of the class. Defaults to "transformation". This allows subclassing.

`Transformation.`**`node_id`**
the id of the Node that did the transformation.

`Transformation.`**`network_id`**
the id of the network the transformation is in.

`Transformation.`**`info_in_id`**
the id of the info that was transformed.

`Transformation.`**`info_out_id`**
the id of the info produced by the transformation.

Relationships

`Transformation.`**`node`**
the Node that did the transformation.

`Transformation.`**`network`**
the network the transmission is in.

`Transformation.`**`info_in`**
the info that was transformed.

`Transformation.`**`info_out`**
the info produced by the transformation.

Methods

`Transformation.`**`__repr__`**`()`
The string representation of a transformation.

`Transformation.`**`__json__`**`()`
The json representation of a transformation.

`Transformation.`**`fail`**`()`
Fail a transformation.

Participant

The Participant object corresponds to a real world participant. Each person who takes part will have a corresponding entry in the Participant table. Participants can be associated with Nodes and Questions.

class `dallinger.models.`**`Participant`**`(`*worker_id*, *assignment_id*, *hit_id*, *mode*`)`
An ex silico participant.

Columns

`Participant.`**`type`**
a String giving the name of the class. Defaults to "participant". This allows subclassing.

Participant.**worker_id**
A String, the worker id of the participant.

Participant.**assignment_id**
A String, the assignment id of the participant.

Participant.**unique_id**
A String, a concatenation of *worker_id* and *assignment_id*, used by psiTurk.

Participant.**hit_id**
A String, the id of the hit the participant is working on

Participant.**mode**
A String, the mode in which Dallinger is running – live, sandbox or debug.

Participant.**end_time**
The time at which the participant finished.

Participant.**base_pay**
The amount the participant was paid for finishing the experiment.

Participant.**bonus**
the amount the participant was paid as a bonus.

Participant.**status**
String representing the current status of the participant, can be –

- `working` - participant is working
- `submitted` - participant has submitted their work
- `approved` - their work has been approved and they have been paid
- `rejected` - their work has been rejected
- `returned` - they returned the hit before finishing
- `abandoned` - they ran out of time
- `did_not_attend` - the participant finished, but failed the attention check
- `bad_data` - the participant finished, but their data was malformed
- `missing_notification` - this indicates that Dallinger has inferred that a Mechanical Turk notification corresponding to this participant failed to arrive. This is an uncommon, but potentially serious issue.

Relationships

dallinger.models.Participant.**all_questions**
All the questions associated with this participant.

dallinger.models.Participant.**all_nodes**
All the Nodes associated with this participant.

Methods

Participant.**__json__**()
Return json description of a participant.

Participant.**fail**()
Fail a participant.

Set *failed* to True and *time_of_death* to now. Instruct all not-failed nodes associated with the participant to fail.

Participant.**infos**(*type=None*, *failed=False*)
Get all infos created by the participants nodes.

Return a list of infos produced by nodes associated with the participant. If specified, type filters by class. By default, failed infos are excluded, to include only failed nodes use failed=True, for all nodes use failed=all. Note that failed filters the infos, not the nodes - infos from all nodes (whether failed or not) can be returned.

Participant.**nodes**(*type=None*, *failed=False*)
Get nodes associated with this participant.

Return a list of nodes associated with the participant. If specified, type filters by class. By default failed nodes are excluded, to include only failed nodes use failed=True, for all nodes use failed=all.

Participant.**questions**(*type=None*)
Get questions associated with this participant.

Return a list of questions associated with the participant. If specified, type filters by class.

Question

A Question is a way to store information associated with a Participant as opposed to a Node (Infos are made by Nodes, not Participants). Questions are generally useful for storing responses debriefing questions etc.

class dallinger.models.**Question**(*participant*, *question*, *response*, *number*)
Responses of a participant to debriefing questions.

Columns

Question.**type**
a String giving the name of the class. Defaults to "question". This allows subclassing.

Question.**participant_id**
the participant who made the response

Question.**number**
A number identifying the question. e.g., each participant might complete three questions numbered 1, 2, and 3.

Question.**question**
the text of the question

Question.**response**
the participant's response. Stored as a string.

Relationships

Question.**participant**
the participant who answered the question

Methods

Question.**__json__**()
Return json description of a question.

Question.**fail**()
Fail a question.

Set *failed* to True and *time_of_death* to now.

CHAPTER 24

The Experiment Class

Experiments are designed in Dallinger by creating a custom subclass of the base Experiment class. The code for the Experiment class is in experiments.py. Unlike the *other classes*, each experiment involves only a single Experiment object and it is not stored as an entry in a corresponding table, rather each Experiment is a set of instructions that tell the server what to do with the database when the server receives requests from outside.

class dallinger.experiments.**Experiment**(*session*)

Define the structure of an experiment.

verbose

Boolean, determines whether the experiment logs output when running. Default is True.

task

String, the name of the experiment. Default is "Experiment title".

session

session, the experiment's connection to the database.

practice_repeats

int, the number of practice networks (see *role*). Default is 0.

experiment_repeats

int, the number of non practice networks (see *role*). Default is 0.

recruiter

Recruiter, the Dallinger class that recruits participants. Default is PsiTurkRecruiter.

initial_recruitment_size

int, the number of participants requested when the experiment first starts. Default is 1.

known_classes

dictionary, the classes Dallinger can make in response to front-end requests. Experiments can add new classes to this dictionary.

__init__(*session*)

Create the experiment class. Sets the default value of attributes.

add_node_to_network(*node*, *network*)

Add a node to a network.

This passes *node* to `add_node()`.

assignment_abandoned(*participant*)
What to do if a participant abandons the hit.

This runs when a notification from AWS is received indicating that *participant* has run out of time. Calls *fail_participant()*.

assignment_returned(*participant*)
What to do if a participant returns the hit.

This runs when a notification from AWS is received indicating that *participant* has returned the experiment assignment. Calls *fail_participant()*.

attention_check(*participant*)
Check if participant performed adequately.

Return a boolean value indicating whether the *participant*'s data is acceptable. This is mean to check the participant's data to determine that they paid attention. This check will run once the *participant* completes the experiment. By default performs no checks and returns True. See also *data_check()*.

attention_check_failed(*participant*)
What to do if a participant fails the attention check.

Runs when *participant* has failed the *attention_check()*. By default calls *fail_participant()*.

bonus(*participant*)
The bonus to be awarded to the given participant.

Return the value of the bonus to be paid to *participant*. By default returns 0.

bonus_reason()
The reason offered to the participant for giving the bonus.

Return a string that will be included in an email sent to the *participant* receiving a bonus. By default it is "Thank you for participating! Here is your bonus."

create_network()
Return a new network.

create_node(*participant*, *network*)
Create a node for a participant.

data_check(*participant*)
Check that the data are acceptable.

Return a boolean value indicating whether the *participant*'s data is acceptable. This is meant to check for missing or invalid data. This check will be run once the *participant* completes the experiment. By default performs no checks and returns True. See also, *attention_check()*.

data_check_failed(*participant*)
What to do if a participant fails the data check.

Runs when *participant* has failed *data_check()*. By default calls *fail_participant()*.

fail_participant(*participant*)
Fail all the nodes of a participant.

get_network_for_participant(*participant*)
Find a network for a participant.

If no networks are available, None will be returned. By default participants can participate only once in each network and participants first complete networks with *role="practice"* before doing all other networks in a random order.

info_get_request(*node*, *infos*)
Run when a request to get infos is complete.

info_post_request(*node*, *info*)
Run when a request to create an info is complete.

log(*text*, *key='?????'*, *force=False*)
Print a string to the logs.

log_summary()
Log a summary of all the participants' status codes.

networks(*role='all'*, *full='all'*)
All the networks in the experiment.

node_get_request(*node=None*, *nodes=None*)
Run when a request to get nodes is complete.

node_post_request(*participant*, *node*)
Run when a request to make a node is complete.

recruit()
Recruit participants to the experiment as needed.

This method runs whenever a participant successfully completes the experiment (participants who fail to finish successfully are automatically replaced). By default it recruits 1 participant at a time until all networks are full.

save(**objects*)
Add all the objects to the session and commit them.

This only needs to be done for networks and participants.

setup()
Create the networks if they don't already exist.

submission_successful(*participant*)
Run when a participant submits successfully.

transformation_get_request(*node*, *transformations*)
Run when a request to get transformations is complete.

transformation_post_request(*node*, *transformation*)
Run when a request to transform an info is complete.

transmission_get_request(*node*, *transmissions*)
Run when a request to get transmissions is complete.

transmission_post_request(*node*, *transmissions*)
Run when a request to transmit is complete.

vector_get_request(*node*, *vectors*)
Run when a request to get vectors is complete.

vector_post_request(*node*, *vectors*)
Run when a request to connect is complete.

CHAPTER 25

Web API

The Dallinger API allows the experiment frontend to communicate with the backend. Many of these routes correspond to specific functions of Dallinger's *classes*, particularly *dallinger.models.Node*. For example, nodes have a `connect` method that creates new vectors between nodes and there is a corresponding `connect/` route that allows the frontend to call this method.

Miscellaneous routes

```
GET /ad_address/<mode>/<hit_id>
```

Used to get the address of the experiment on the psiTurk server and to return participants to Mechanical Turk upon completion of the experiment. This route is pinged automatically by the function `submitAssignment` in dallinger.js.

```
GET /<directory>/<page>
```

Returns the html page with the name `<page>` from the directory called `<directory>`.

```
GET /summary
```

Returns a summary of the statuses of Participants.

```
GET /<page>
```

Returns the html page with the name `<page>`.

Experiment routes

```
GET /experiment/<property>
```

Returns the value of the requested property as a JSON <property>.

```
GET /info/<node_id>/<info_id>
```

Returns a JSON description of the requested info as info. node_id must be specified to ensure the requesting node has access to the requested info. Calls experiment method 'info_get_request(node, info).

```
POST /info/<node_id>
```

Create an info with its origin set to the specified node. *contents* must be passed as data. info_type can be passed as data and will cause the info to be of the specified type. Also calls experiment method info_post_request(node, info).

```
POST /launch
```

Initializes the experiment and opens recruitment. This route is automatically pinged by Dallinger.

```
GET /network/<network_id>
```

Returns a JSON description of the requested network as network.

```
POST /node/<node_id>/connect/<other_node_id>
```

Create vector(s) between the node and other_node by calling node.connect(whom=other_node). Direction can be passed as data and will be forwarded as an argument. Calls experiment method vector_post_request(node, vectors). Returns a list of JSON descriptions of the created vectors as vectors.

```
GET /node/<node_id>/infos
```

Returns a list of JSON descriptions of the infos created by the node as infos. Infos are identified by calling node.infos(). info_type can be passed as data and will be forwarded as an argument. Requesting node and the list of infos are also passed to experiment method info_get_request(node, infos).

```
GET /node/<node_id>/neighbors
```

Returns a list of JSON descriptions of the node's neighbors as nodes. Neighbors are identified by calling node.neighbors(). node_type and connection can be passed as data and will be forwarded as arguments. Requesting node and list of neighbors are also passed to experiment method node_get_request(node, nodes).

```
GET /node/<node_id>/received_infos
```

Returns a list of JSON descriptions of the infos sent to the node as infos. Infos are identified by calling node.received_infos(). info_type can be passed as data and will be forwarded as an argument. Requesting node and the list of infos are also passed to experiment method info_get_request(node, infos).

```
GET /node/<int:node_id>/transformations
```

Returns a list of JSON descriptions of all the transformations of a node identified using node.transformations(). The node id must be specified in the url. You can also pass transformation_type as data and it will be forwarded to node.transformations() as the argument type.

```
GET /node/<node_id>/transmissions
```

Returns a list of JSON descriptions of the transmissions sent to/from the node as transmissions. Transmissions are identified by calling node.transmissions(). direction and status can be passed as data and will

be forwarded as arguments. Requesting node and the list of transmissions are also passed to experiment method `transmission_get_request(node, transmissions)`.

```
POST /node/<node_id>/transmit
```

Transmit to another node by calling `node.transmit()`. The sender's node id must be specified in the url. As with `node.transmit()` the key parameters are `what` and `to_whom` and they should be passed as data. However, the values these accept are more limited than for the backend due to the necessity of serialization.

If `what` and `to_whom` are not specified they will default to `None`. Alternatively you can pass an int (e.g. '5') or a class name (e.g. `Info` or `Agent`). Passing an int will get that info/node, passing a class name will pass the class. Note that if the class you are specifying is a custom class it will need to be added to the dictionary of known_classes in your experiment code.

You may also pass the values property1, property2, property3, property4 and property5. If passed this will fill in the relevant values of the transmissions created with the values you specified.

The transmitting node and a list of created transmissions are sent to experiment method `transmission_post_request(node, transmissions)`. This route returns a list of JSON descriptions of the created transmissions as `transmissions`. For example, to transmit all infos of type Meme to the node with id 10:

```
reqwest({
    url: "/node/" + my_node_id + "/transmit",
    method: 'post',
    type: 'json',
    data: {
        what: "Meme",
        to_whom: 10,
    },
});
```

```
GET /node/<node_id>/vectors
```

Returns a list of JSON descriptions of vectors connected to the node as `vectors`. Vectors are identified by calling `node.vectors().direction` and `failed` can be passed as data and will be forwarded as arguments. Requesting node and list of vectors are also passed to experiment method `vector_get_request(node, vectors)`.

```
POST /node/<participant_id>
```

Create a node for the specified participant. The route calls the following experiment methods: `get_network_for_participant(participant)`, `create_node(network, participant)`, `add_node_to_network(node, network)`, and `node_post_request(participant, node)`. Returns a JSON description of the created node as `node`.

```
POST /notifications
GET /notifications
```

This is the route to which notifications from AWS are sent. It is also possible to send your own notifications to this route, thereby simulating notifications from AWS. Necessary arguments are `Event.1.EventType`, which can be `AssignmentAccepted`, `AssignmentAbandoned`, `AssignmentReturned` or `AssignmentSubmitted`, and `Event.1.AssignmentId`, which is the id of the relevant assignment. In addition, Dallinger uses a custom event type of `NotificationMissing`.

```
GET /participant/<participant_id>
```

Returns a JSON description of the requested participant as `participant`.

```
POST /participant/<worker_id>/<hit_id>/<assignment_id>/<mode>
```

Create a participant. Returns a JSON description of the participant as `participant`.

```
POST /question/<participant_id>
```

Create a question. `question`, `response` and `question_id` should be passed as data. Does not return anything.

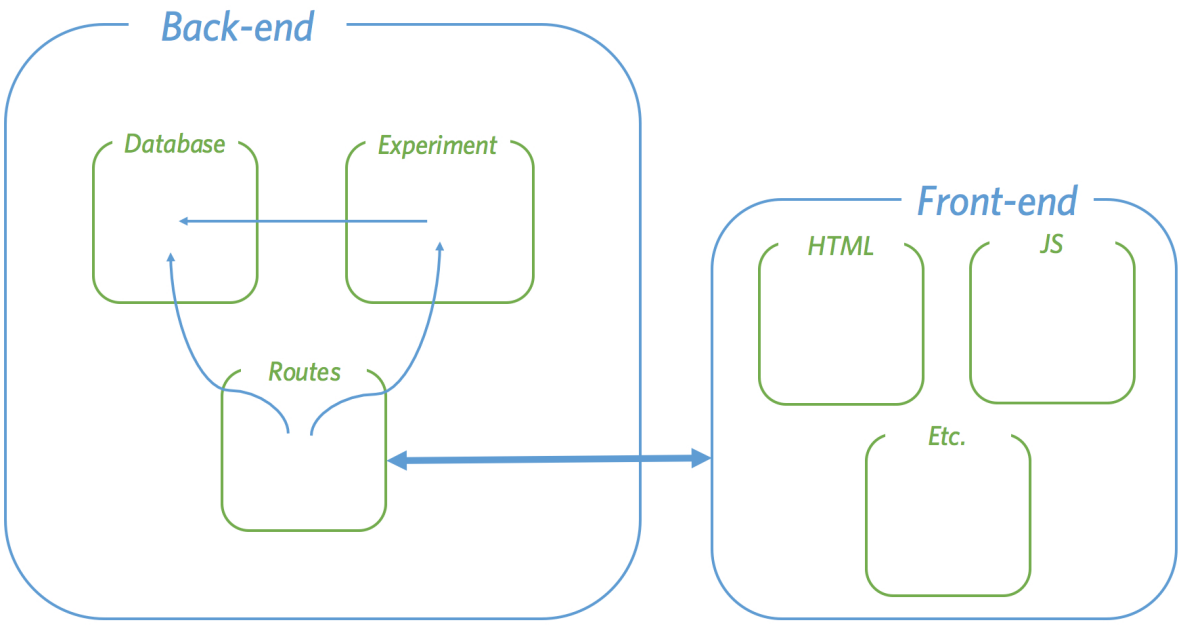
```
POST /transformation/<int:node_id>/<int:info_in_id>/<int:info_out_id>
```

Create a transformation from `info_in` to `info_out` at the specified node. `transformation_type` can be passed as data and the transformation will be of that class if it is a known class. Returns a JSON description of the created transformation.

CHAPTER 26

Communicating With the Server

When an experiment is running, the database and the experiment class (i.e. the instructions for what to do with the database) will be hosted on a server, the server is also known as the "back-end". However, participants will take part in experiments via an interactive web-site (the "front-end"). Accordingly for an experiment to proceed there must be a means of communication between the front and back ends. This is achieved with routes:



Routes are specific web addresses on the server that respond to requests from the front-end. Routes have direct access to the database, though most of the time they will pass requests to the experiment which will in turn access the database. As such, changing the behavior of the experiment is the easiest way to create a new experiment. However it is also possible to change the behavior of the routes or add new routes entirely.

Requests generally come in two types: "get" requests, which ask for information from the database, and "post" requests

which send new information to be added to the database. Once a request is complete the back-end sends a response back to the front-end. Minimally, this will include a notification that the request was successfully processed, but often it will also include additional information.

As long as requests are properly formatted and correctly addressed to routes, the back-end will send the appropriate response. This means that the front-end could take any form. For instance requests could come from a standard HTML/CSS/JS webpage, a more sophisticated web-app, or even from the experiment itself.

CHAPTER 27

Acknowledgments

Dallinger is sponsored by the Defense Advanced Research Projects Agency through the NGS2 program. The contents of this documentation does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Dallinger's predecessor, Wallace, was supported in part by the National Science Foundation through grants 1456709 and 1408652.

CHAPTER 28

Dallinger's incubator

Dallinger was one of the first scientists to perform experimental evolution. See his Wikipedia article for the specifics of his incubation experiments.

Index

Symbols

A

B

C

D