Unifying the Build Flow of the PyNN Modules for the FACETS Hardware Systems

Sebastian Jeltsch

November 30, 2010

Contents

1. An Introduction to the Automated Build Flow of the symap2ic Project

This chapter offers a starting point for new users and developers of the `symap2ic` software framework to get an overview of the basic project structure. Furthermore, a step-by-step tutorial is provided to build the software that is necessary to work with the FACETS hardware systems [*Schemmel et al.*, 2010, 2008; *Grübl*, 2007] or virtual versions of these [*Vogginger*, 2010].

Prerequisites The build flow in focus of this chapter is based on the WAF build system in version 1.6 as well as on project checkouts around November 2010. The Python 2.6.5 interpreter has been used to realize and verify the correct build functionality. A comprehensive documentation of the WAF tool can be found in the internet under `http://waf.googlecode.com/svn/docs/wafbook/single.html`.

1.1. Project Structure

The software stack developed within the FACETS Project is distributed over numerous smaller repositories. In these repositories, two different version control systems, namely *subversion (svn)* and *git*, are used. To simplify the usage of the distributed software, a top-level project – called *symap2ic* – has been created. It automatically retrieves the sub-projects – thereby satisfying their mutual dependencies– and builds plus installs them into the right place. An overview of the directory structure can be found in Figure 1.1.

The main directories are `src` and `components`, which comprise the source code of a FACETS logging class and additional WAF functionality (`src`) as well as the collected source code of all sub-projects (`components`). After a successful build process, the `bin` and `lib` folder comprise the resulting program binaries and libraries ready to use. Ultimately, the `doc` folder is the place where generated documentation from the sub-projects is installed to. These documentations are generated via *doxygen*, which parses the program sources and the code comments to automatically assemble a reference manual. After this first insight into the basic project hierarchy and relevant tools, we will now have a look on the conceptual structure of the build process.

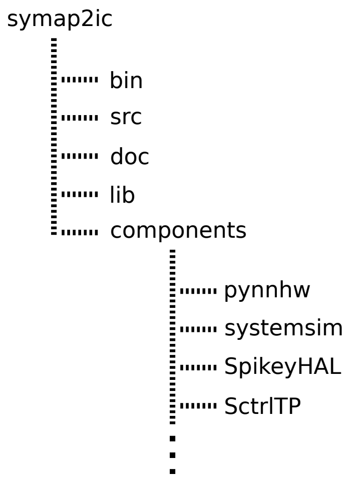

Figure 1.1.: An illustration of the directory structure found in the symap2ic project. The symap2ic meta project itself is kept plain and comprises only little code. It organizes the sub-projects, which are stored in the `components` folder.

1.2. Build Flow Concepts

The basic concept behind the building process is, that each sub-project provides its own description in terms of a *wscript*. Every wscript is a recipe to tell WAF how to build the sub-project and further defines the inter- and intra-project dependencies. Besides, the recipe may include different variants and optional build targets. The main build process from the symap2ic root project then executes the building processes of the sub-projects recursively. The major problems in our case are mutual dependencies of the different projects, e.g. the PyNN module depends both on the low-level software and the ethernet communication stack (see Section 1.3). To resolve such software dependencies, symap2ic offers the functionality to retrieve a set of sub-projects which have been predefined and satisfies all dependencies. This functionality can be accessed via the `--repositories` configuration argument from the command line. More information on that will be provided in the following PyNN building walk through.

1.3. Building the PyNN Modules – a Step by Step Walk Through

PyNN is a simulator independent neural network description language based on Python, which can be used to access the FACETS hardware even as a non-hardware-expert. The high-level PyNN instructions are translated via a custom software backend into low-level operations. This section explains how to build such a backend from the project sources. This is necessary to work with the hardware systems.

First of all you have to make sure that you have access to the relevant project repositories. The rights management for the git repositories is independent from the one for the svn repositories. So make sure you have both.

In order to access the svn you need a FACETS account from Björn Kindler (bjoern.kindler@kip.uni-heidelberg.de). For access to the git repositories you must register on: `https://gitviz.kip.uni-heidelberg.de` Afterwards you can get read&write permissions for individual projects from:

- Eric Müller (emueller@kip.uni-heidelberg.de)
- Sebastian Jeltsch (sjeltsch@kip.uni-heidelberg.de)

Authentication is required to download any non-public (most projects) project, but the gitviz server is configured to deny password authentication in general (although you can access your projects via password authentication through the web front end). Instead you must use the common user: *git*. You are individually identified by an unique ssh key. In order to do so, you need to store your ssh public key in your gitviz profile via the web front end. If you do not have ssh keys yet, create a pair of private and public keys by typing `ssh-keygen` in the command line and following the instructions on the screen (or just keep pressing return, you can also keep the passphrase empty). The generated ssh keys are commonly stored in your home directory under the hidden `.ssh` sub-folder. In case you used the default values during the key-pair generation you can copy the output of:

```
cd; cat .ssh/id_rsa.pub
```

into the respective field in your gitviz profile. Now, you are ready to checkout an instance of the repositories. To get a copy of the symap2ic project into your current directory type:

```
git clone git@gitviz.kip.uni-heidelberg.de:symap2ic.git
```

A good starting point to get familiar with the git version control software might be the open "Pro Git" book (`http://progit.org/book/`).

Now, go to the symap2ic project directory. When you look at the top-level structure you'll find `waf` and a `wscript`. `waf` is a link to a WAF executable which is provided with the repository. Note, that there is no need to install the build system on your computer, instead you can and should use the provided version. This ensures that no version conflicts arise and simplifies usage as well as maintenance. The `wscript` is the top-level build receipt. If you are interested in the build process and are familiar with Python you may have a look into it. When you enter `python waf configure --help` you'll get an overview on the available command line arguments. To automatically download all repositories necessary for the *Executable System Specification* you would execute:

```
. ./bootstrap.sh
python waf configure --repositories=<repo set>
```

The `bootstrap.sh` is responsible for sourcing the `$SYMAP2IC_PATH` environment variable in your current shell. For a correct functionality you need to source the bootstrap file in each new shell or set the environment variable in the rc-script of your shell yourself (e.g.

`$HOME/.bashrc` for bash). The second command is responsible for actually getting the necessary projects. The `<repo set>` needs to be replaced with either "systemsim" or "Stage1" for either the virtual hardware system or the operation of the Spikey hardware. If your chosen repository collection includes svn sources you will be prompted for your FACETS username and password. If everything went right you can now find new sub-directories in the `components` folder. Furthermore, if you rerun `python waf configure --help` you will now be confronted with extra command line options provided by the added sub-projects. For us the `--stage` option is of special interest. It ultimately lets us define the build target. In case we downloaded the "systemsim" repositories the *Stage2* option would the only valid choice. If we have chosen the Stage1 repositories we could either build the Stage1 multi-chip or singe-chip PyNN module (valid `--stage` options taken from the help prompt are: Stage2, Stage1-single, Stage1-multi). To finally build the project use:

```
python waf configure --stage=<option> install
```

after a successful build and installation you'll find the necessary shared object files for PyNN in your `symap2ic/lib` (it happens from time to time that WAF prompts for some missing libraries, then simply rerun `python waf install`). In the Stage1 case you'll find additional tools in the bin folder to control and operate the hardware system. Experiments on the actual Stage1 hardware require that you specify the Spikey chip you want to run your experiment on (experiments on the virtual hardware require no such files). Therefore, execute:

```
echo station<your station id> > ~/my_Stage1_station
```

For a station number and a personal Spikey chip ask your advisor. If a special PyNN version – with hardware support enabled – is installed and you you have correctly set the `$SYMAP2IC_PATH` environment variable (either via the bootstrap.sh or the rc-script of your shell) you should be ready to run your PyNN scripts on either the virtual hardware or the Stage1 hardware system.

2. Build Flow Unification and Integration

One major goal of the work at hand has been to integrate the new multi-chip system software into the existing MappingTool project and to unify the build system. Therefore, missing build descriptions have been implemented to make all existing PyNN hardware modules (Stage1 single-chip, Stage1 multi-chip and Stage2 virtual hardware) buildable from the top-level symap2ic project (see Section 1.1).

2.1. Multi-Chip Software Integration

Initially, the multi-chip software developed in *Jeltsch* [2010] has been integrated into the existing project structure to make its functionality accessible for everyone. The source code has been adapted to the main development line and the integration has been modularized to reduce the software's footprint as described below.

Automatically Generated Documentation The generation and installation of auto-generated doxygen documentation and installation of a corresponding LaTeXpdf has been integrated into the WAF flow. After building the project with die `--with-doxygen` configuration option the generated latex sources and pictures reside in the WAF build folder. The finally typeset documentation is installed in the `symap2ic/doc/pdf` directory. If a HTML documentation is desired one can build the documentation directly from the MappingTool directory with the respective wscript.

2.2. Unified Build Flow

All three build variants for the PyNN backends responsible for the Spikey single-chip, multi-chip and Stage2 virtual hardware are now accessible via the `--stage` configuration option on the project's top-level.

To reduce the dependencies between existing software and speed up the build process `C/C++` macros have been used to provide variant builds of the MappingTool for the different PyNN targets. Otherwise, the build for the virtual hardware would depend on the SpikeyHAL low-level software and the ARQ-ethernet stack [*Schilling*, 2010]. Moreover, the Stage1 multi-chip build would depend on the system simulation. This reduces the size of the final binaries and the number of necessary build targets for any variant of the MappingTool from over 200 to about 130.

3. MongoDB

MongoDB is a free and open non-relational database system. Compared to databases implementing the conventional, relational scheme (e.g. SQL), it allows to store arbitrary data structures without requirements on the format. This makes the system highly attractive for an application in the field of hardware systems under ongoing development, as the calibration data sets can easily develop, too. Also, incomplete and non-standard-conform calibration data sets can be stored and retrieved at any time without additional effort. The database stores these arbitrary sets of data in so-called *collections* which themselves are associated to one database hosted by the server.

3.1. The MongoDB-Setup for Calibration Data

Application of calibration data on the HICANN system and the Stage1 system requires at least mongoDB version 1.6 and the respective development libraries. Pre-build ubuntu packages can be found in the Electronic Vision(s) launchpad repository under: `https://launchpad.net/~visions/+archive/visions-default`. Typical for database server systems in general is that data can be accessed from any computer over the network. Nevertheless, our mongoDB responsible for storing and organizing the calibration data which is necessary for the Stage1 multi-chip system has been set up on the experiment computer "paul" directly. At the time of writing no user name or password were required to authenticate. A description on how to access the database can be found in the following Section 3.1.1.

If you have any problems working with the mongoDB setup or the stored data contact the responsible person, which are listed below. Responsible for the administration of the database servers are:

- Eric Müller (emueller@kip.uni-heidelberg.de)
- Sebastian Jeltsch (sjeltsch@kip.uni-heidelberg.de)

Responsible for the Stage1 calibration datasets is:

- Daniel Brüderle (bruederle@kip.uni-heidelberg.de)

Responsible for the Stage2 calibration datasets is:

- Marc-Olivier Schwartz (marcolivier.schwartz@kip.uni-heidelberg.de)

3.1.1. Short PyMongo Introduction

To convey an idea on how to use the mongoDB we will perform some common tasks like accessing, creating, modifying and deleting information in the database. Basic Python knowledge will proof useful, but is not explicitly required. Since data in the database is weakly typed, working with data in the database is similar to working with a Python dictionary.

Establishing a Database Connection The easies way to connect to the database is to open a remote shell on the computer "paul" (it hosts the calibration database). Here, *pymongo* is already installed and default parameters assumed by pymongo can be used. After connecting to "paul" you should open a interactive python interpreter.

```
ssh paul
$ ipython
```

Before we can establish a connection to the database we need to import the appropriate python module, as follows:

```
In [1]: import pymongo as p
```

We don't need to pass any arguments to the Connection() command. Pymongo will simply use its default connection parameters, which correspond to the mongoDB server running on "localhost" and using the default tcp port.

```
In [2]: con = p.Connection()

In [4]: con.database_names()
Out[4]: [u'localhost', u'tutorial', u'admin', u'local']
```

The second command did return the available databases in the currently running instance of mongoDB. In this tutorial the databases "localhost" and "tutorial" are of special interest to us.

First of all, we are going to have a look into the "localhost" database, which actually stores the calibration data.

```
In [6]: db = con.localhost

In [7]: db.collection_names()
Out[7]: [u'user', u'system.indexes', u'spikeycalibration',
         u'test0', u'trafo', u'spikeyparametertranslation']
```

The collection "spikeyparametertranslation" stores the calibration data and the collection "user" is an address book of people responsible for one or more calibration data sets.

In this tutorial we are focusing on the calibration data sets, they are far more interesting than a bunch of names and email addresses to us. Furthermore, playing with the data sets will convey a first feeling about the data format and the amount of data necessary to calibrate the hardware.

Reading Data from the Database After connecting to the server and picking a database we will start with reading data. Please do not try to modify data at this point, as you are working with actual calibration data and we will get to the modification of data later on.

```
In [8]: spikey = db.spikeyparametertranslation

In [9]: spikey.count()
Out[9]: 51
```

As see can see, 51 calibration data sets are organized in the mongoDB at the time of writing.

The next step is optional and yields a lengthly output (you can always interrupt the output with [ctrl+c]). The listed command will print all data stored in the "spikeyparametertranslation" collection on the screen.

```
In [11]: for i in spikey.find():
               print i
Out[11]: ... # lot of stuff
```

So what have we done? The function `find{}` returns an iterator – a so-called cursor in the mongoDB world – which we are using to iterate over all sets, one after another. Normally, we are more interested in requesting specific data from the database. Therefore, We can utilize a common Python dictionary as a filter.

```
In [24]: for i in spikey.find({'chipID': 456, 'transDir':2}):
               print i
Out[24]: ... # should be one data set
```

Consequently, `find` will only return data sets with matching entries.

Storing Data in the Database So far we have seen how to get information from the database. Now, we want to explore how to store new information. To keep the calibration database clean we initially switch to the tutorial database and collection which we have seen previously.

```
In [36]: db_tut = con.tutorial
In [37]: tut = db_tut.tutorial # reference to collection 'tutorial'
```

This should typically be an empty database. So lets check:

```
In [37]: for i in tut.find():
               print i
```

Don't worry if some data is already stored in this database. We can simply ignore it and continue. By executing:

```
In [38]: tut.save({'foo':'bar'})
```

you add a new data set to the collection with the key "foo" and the value "bar" (note, in a relational database you would first need to have defined a new table with a "foo" column storing strings) – and yes – it is really that simple.

Manipulating Data in the Database In the previous paragraph we stored our first, own object in the database. Let's see whether we can find it in the tutorial collection collection and modify it:

```
In [60]: obj = tut.find({'foo':'bar'})[0]
```

Remember that `find` returns an iterator, we therefore pick the start element by means of the array operator `[]`. After we got the object, working with it is like working with a python dictionary: we can modify data and add data by using the `[]`-operator:

```
In [61]: obj['foo'] = 42
In [62]: obj['A'] = 4.2
```

Note that, we have not only changed the value but also the data type, abusing the charm of scheme free database system. The modification are not yet stored in the database. You are working on a local copy of the data. To save the changes run:

```
In [63]: tut.save(obj)
```

Let's check whether the changes have been correctly stored on the server:

```
In [64]: for i in tut.find():
             print i
Out [64]: {u'_id': ObjectId('4cdd4a426c632165c5000000'),
            u'foo': u'bar'
            u'A': 4.2 }
```

Removing Data from the Database Finally, we want to cover up our track and delete the generated data, so that the next person will hopefully find an empty tutorial database. We can delete data either by its unique object id (a random hash) or via references. You may have already noticed the object id associated with each data set, which has been printed to the screen each time you did something like:

```
in [72]: for i in tut.find():
             print i
out [72]: {u'_id': ObjectId('4cdd4a426c632165c5000000'),
            u'foo': u'bar'
            u'A': 4.2 }
```

To finally remove a specific data set you could execute (you need to substitute the object id with your own unique id):

```
in [73]: tut.remove(p.objectid.ObjectId('4cdd4a426c632165c5000000'))
```

Or you can use references: the next command deletes all data in "tutorial.tutorial" by reference:

```
in [74]: for i in tut.find():
     tut.remove(i)
```

Note that you have not been prompted for any confirmation on deleting the data, so be careful whenever you remove data from the database. Otherwise the administrators will be very pleased to restore the original entries from a raw file system backup.

You should now have a basic knowledge about requesting, creating, modifying and removing data from a mongoDB via the pymongo API. For a complete documentation on the pymongo Python module visit: `http://api.mongodb.org/python/`. Besides, there are language bindings for practically any common programming language.

3.2. Horizontal Database Scaling

Calibration data sets for a complete wafer system are significantly larger than the ones necessary for the Stage1 system. You may have already encountered the sets used for the Stage1 system during the mongoDB walk through presented in Section 1.3. To provide the necessary throughput for the wafer-scale system and allow for low latencies access the mongoDB system can be scaled horizontally to span multiple physical servers, each contributing almost linearly to the overall bandwidth.

A schematic of a typical mongoDB sharding setup can be found in Figure 3.1. Compared to a single-server setup multiple daemons control the operation of the database. There needs to be at least one *mongod* process set as configuration server, one *mongod* process configured to act as a shard and one *mongos* routing process. The configuration servers are responsible for the organization of the database instance and communicate with both, the database backends (shards) and the mongos request routers. The so-called shards store the actual data. It is mirrored transparently between the available shards on a chunk-level with predefined chunk sizes. That means, in normal operation data can be provided coherently by all available shards and the router can balance the load on all shards symmetrically. If a data set is too new and not yet mirrored to all shards the mongos route the request to shards with the data available.

To test a multi-server mongoDB a setup can be easily created by means of a script, which can be found on github: `https://github.com/mongodb/mongo-snippets/blob/master/sharding/simple-setup.py` or in the Appendix A. It sets up a database configuration consisting of one configuration server , one mongos and three shard instances. The database can then be accessed via the mongos router running on “localhost” port 27017.

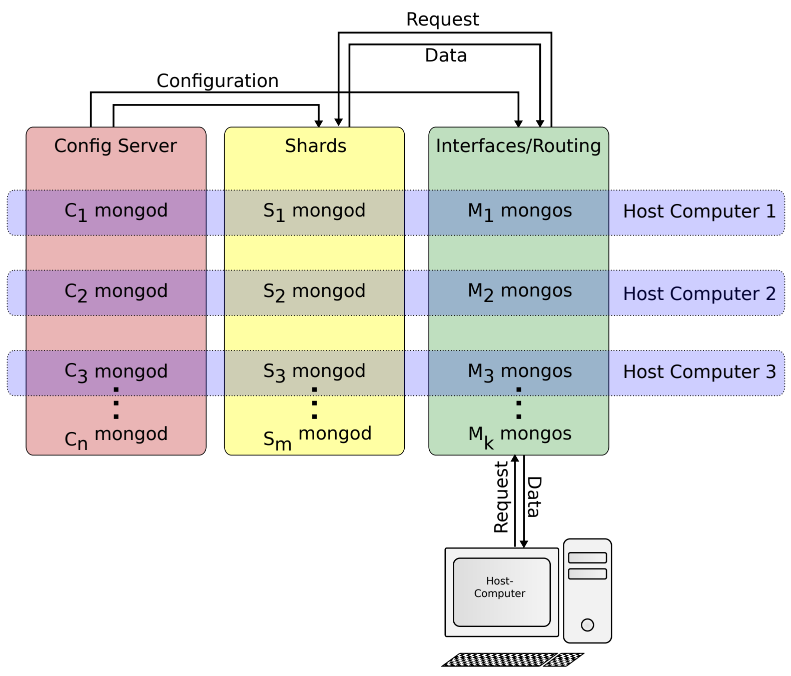

Figure 3.1.: An illustration of a completely redundant sharding mongoDB setup. Stored data sets can be provided transparently by any physical server. Such a multi-server setup improves the overall bandwidth of the database. Note, that this is only one example configuration. Instances of the three different process types can be arbitrarily distributed. But incoming requests are only managed by *mongos*.

A. Sample Script for Horizontal mongoDB Scaling

```
#!/usr/bin/python2
# source:
# https://github.com/mongodb/mongo-snippets/blob/master/sharding/simple-setup.py
# commit: b7774daf9b327ed761ac

import os, sys, shutil, atexit
import pymongo

from socket import error, socket, AF_INET, SOCK_STREAM
from select import select
from subprocess import Popen, PIPE, STDOUT
from threading import Thread
from time import sleep

try:
    # new pymongo
    from bson.son import SON
except ImportError:
    # old pymongo
    from pymongo.son import SON

# BEGIN CONFIGURATION

# some settings can also be set on command line.
# start with --help to see options

BASE_DATA_PATH='/data/db/sharding/' #warning:
                                    #gets wiped every time you run this
MONGO_PATH=os.getenv( "MONGO_HOME" , os.path.expanduser('~/10gen/mongo/') )
N_SHARDS=3
N_CONFIG=1 # must be either 1 or 3
N_MONGOS=1
CHUNK_SIZE=200 # in MB (make small to test splitting)
MONGOS_PORT=27017 if N_MONGOS == 1 else 10000 # start at 10001 when multi

CONFIG_ARGS=[]
MONGOS_ARGS=[]
MONGOD_ARGS=[]
```

```
# Note this reports a lot of false positives.
USE_VALGRIND=False
VALGRIND_ARGS=["valgrind", "--log-file=/tmp/mongos-%p.valgrind", "--leak-check=yes",
               ("--suppressions="+MONGO_PATH+"valgrind.suppressions"), "--"]

# see http://pueblo.sourceforge.net/doc/manual/ansi_color_codes.html
CONFIG_COLOR=31 #red
MONGOS_COLOR=32 #green
MONGOD_COLOR=36 #cyan
BOLD=True

# defaults -- can change on command line
COLLECTION_KEYS = {'foo' : '_id', 'bar': 'key'}

def AFTER_SETUP():
    # feel free to change any of this
    # admin and conn are both defined globaly
    admin.command('enablesharding', 'test')

    for (collection, keystr) in COLLECTION_KEYS.iteritems():
        key=SON((k,1) for k in keystr.split(','))
        admin.command('shardcollection', 'test.'+collection, key=key)

    admin.command('shardcollection', 'test.fs.files', key={'_id':1})
    admin.command('shardcollection', 'test.fs.chunks', key={'files_id':1})


# END CONFIGURATION

for x in sys.argv[1:]:
    opt = x.split("=", 1)
    if opt[0] != '--help' and len(opt) != 2:
        raise Exception("bad arg: " + x )

    if opt[0].startswith('--'):
        opt[0] = opt[0][2:].lower()
        if opt[0] == 'help':
            print sys.argv[0], '[--help] [--chunksize=200] [--port=27017] \
                [--path=/where/is/mongod] [collection=key]'
            sys.exit()
        elif opt[0] == 'chunksize':
            CHUNK_SIZE = int(opt[1])
        elif opt[0] == 'port':
            MONGOS_PORT = int(opt[1])
        elif opt[0] == 'path':
            MONGOS_PATH = opt[1]
        elif opt[0] == 'usevalgrind': #intentionally not in --help
            USE_VALGRIND = int(opt[1])
        else:
            raise( Exception("unknown option: " + opt[0] ) )
    else:
        COLLECTION_KEYS[opt[0]] = opt[1]

print( "MONGO_PATH: " + MONGO_PATH )

if not USE_VALGRIND:
    VALGRIND_ARGS = []

# fixed "colors"
RESET = 0
INVERSE = 7

if os.path.exists(BASE_DATA_PATH):
    shutil.rmtree(BASE_DATA_PATH)

mongod = MONGO_PATH + 'mongod'
mongos = MONGO_PATH + 'mongos'

devnull = open('/dev/null', 'w+')

fds = {}
procs = []

def killAllSubs():
    for proc in procs:
        try:
            proc.terminate()
        except OSError:
            pass #already dead
atexit.register(killAllSubs)

def mkcolor(colorcode):
    base = '\x1b[%sm'
    if BOLD:
        return (base*2) % (1, colorcode)
    else:
        return base % colorcode

def ascolor(color, text):
    return mkcolor(color) + text + mkcolor(RESET)

def waitfor(proc, port):
    trys = 0
    while proc.poll() is None and trys < 40: # ~10 seconds
        trys += 1
        s = socket(AF_INET, SOCK_STREAM)
        try:
            try:
                s.connect(('localhost', port))
                return
            except (IOError, error):
                sleep(0.25)
        finally:
            s.close()

    #extra prints to make line stand out
    print
    print proc.prefix, ascolor(INVERSE, 'failed to start')
    print

    sleep(1)
    killAllSubs()
    sys.exit(1)


def printer():
    while not fds: sleep(0.01) # wait until there is at least one fd to watch

    while fds:
        (files, _, errors) = select(fds.keys(), [], fds.keys(), 1)
        for file in set(files + errors):
            # try to print related lines together
            while select([file], [], [], 0)[0]:
                line = file.readline().rstrip()
                if line:
                    print fds[file].prefix, line
                else:
                    if fds[file].poll() is not None:
                        print fds[file].prefix, ascolor(INVERSE, 'EXITED'), \
                            fds[file].returncode
                        del fds[file]
                        break
                break

printer_thread = Thread(target=printer)
printer_thread.start()

configs = []
for i in range(1, N_CONFIG+1):
    path = BASE_DATA_PATH +'config_' + str(i)
    os.makedirs(path)
    config = Popen(
        [mongod, '--port', str(20000 + i), '--configsvr', '--dbpath', path] \
        + CONFIG_ARGS, stdin=devnull, stdout=PIPE, stderr=STDOUT)
    config.prefix = ascolor(CONFIG_COLOR, 'C' + str(i)) + ':'
    fds[config.stdout] = config
    procs.append(config)
    waitfor(config, 20000 + i)
    configs.append('localhost:' + str(20000 + i))


for i in range(1, N_SHARDS+1):
    path = BASE_DATA_PATH +'shard_' + str(i)
    os.makedirs(path)
    shard = Popen(
        [mongod, '--port', str(30000 + i), '--shardsvr', '--dbpath', path] \
        + MONGOD_ARGS, stdin=devnull, stdout=PIPE, stderr=STDOUT)
    shard.prefix = ascolor(MONGOD_COLOR, 'M' + str(i)) + ':'
    fds[shard.stdout] = shard
    procs.append(shard)
    waitfor(shard, 30000 + i)

#this must be done before starting mongos
for config_str in configs:
    host, port = config_str.split(':')
    config = pymongo.Connection(host, int(port)).config
    config.settings.save({'_id':'chunksize', 'value':CHUNK_SIZE}, safe=True)
del config #don't leave around connection directly to config server

if N_MONGOS == 1:
    MONGOS_PORT -= 1 # added back in loop

for i in range(1, N_MONGOS+1):
    router = Popen(VALGRIND_ARGS + [mongos, '--port', str(MONGOS_PORT+i),
                    '--configdb' , ','.join(configs)] + MONGOS_ARGS,
                    stdin=devnull, stdout=PIPE, stderr=STDOUT)
    router.prefix = ascolor(MONGOS_COLOR, 'S' + str(i)) + ':'
    fds[router.stdout] = router
    procs.append(router)

    waitfor(router, MONGOS_PORT + i)

conn = pymongo.Connection('localhost', MONGOS_PORT + 1)
admin = conn.admin

for i in range(1, N_SHARDS+1):
    admin.command('addshard', 'localhost:3000'+str(i), allowLocal=True)

AFTER_SETUP()

sleep(2) # just to be safe

print '*** READY ***'
print
print
```

```
try:
    printer_thread.join()
except KeyboardInterrupt:
    pass
```

Bibliography

Grübl, A., VLSI implementation of a spiking neural network, Ph.D. thesis, Ruprecht-Karls-Universität, Heidelberg, 2007, document no. HD-KIP 07-10.

Jeltsch, S., Computing with transient states on a neuromorphic multi-chip environment, Diploma thesis, Ruprecht-Karls-Universität, Heidelberg, HD-KIP 10-54, `http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=2095`, 2010.

Schemmel, J., J. Fieres, and K. Meier, Wafer-scale integration of analog neural networks, in *Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN)*, 2008.

Schemmel, J., D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner, A wafer-scale neuromorphic hardware system for large-scale neural modeling, in *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS'10)*, IEEE Press, 2010.

Schilling, M., A highly efficient transport layer for the connection of neuromorphic hardware systems, Diploma thesis, Ruprecht-Karls-Universität, Heidelberg, HD-KIP-10-09, `http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=2000`, 2010.

Vogginger, B., Testing the operation workflow of a neuromorphic hardware system with a functionally accurate model, Diploma thesis, Ruprecht-Karls-Universität, Heidelberg, HD-KIP-10-12, `http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=2003`, 2010.