

The ETSF Test Farm

Collection of servers configured to compile and test a software project remotely to validate code changes,

- for cross-platform development.
- for automated *continuous integration* testing.
- for workload distribution.

European Theoretical Spectroscopy Facility (ETSF), <http://www.etsf.eu>

Alain Jacques <etsf.software@gmail.com> -The ETSF Test Farm - CECAM Zaragoza June 2010



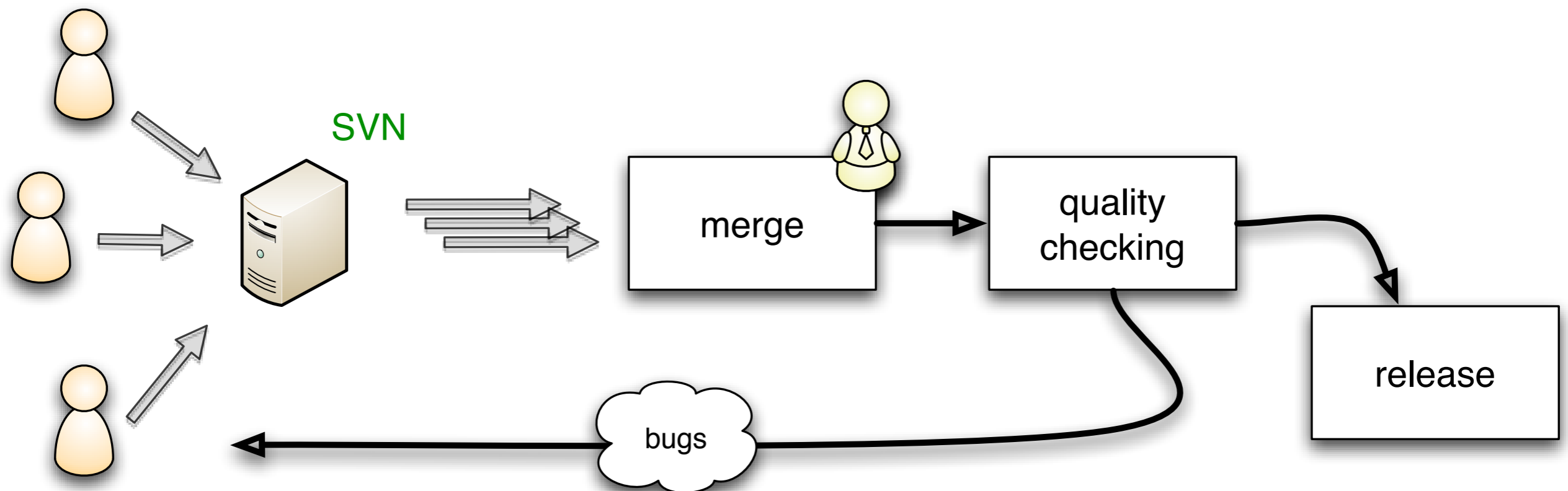
1

A compile or test farm is a collection of computers configured to build and test the code of a software project in order to validate the changes.

The reasons to assemble a test farm can be to sustain a cross-platform development – to expand the users base or perform debugging by compiler diversity – or to adopt the good practices of the “continuous development” method or to distribute the processing load of testing a large code.

In the second case, the use of a test farm modifies the workflow of the code development

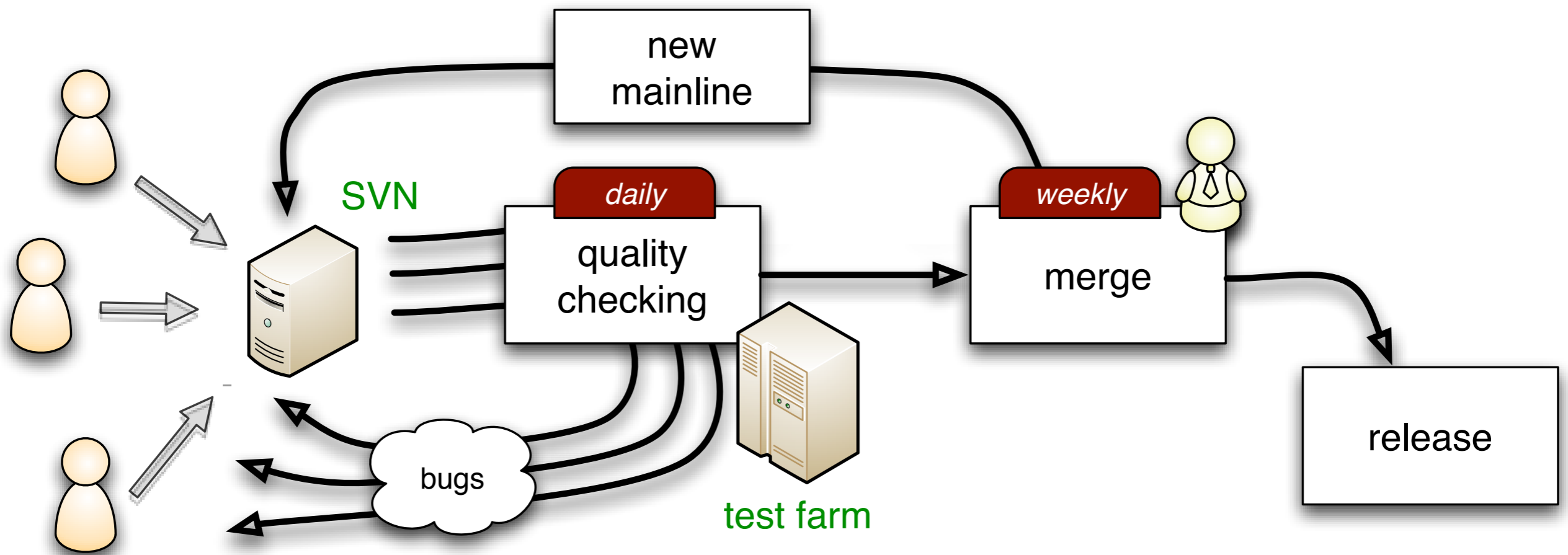
Traditional development workflow



In the traditional way, the developers contributions are stored on a centralized repository and the adoption of some kind of version control system (VCS) like SVN is advisable. When release time approaches, the maintainer of the project will fetch the different contributions in order to merge them into a single source trunk, compile and apply a quality testing procedure like running validation tests or organizing a beta testing. It is very often a nightmare period: conflicts between contributions arise, bugs are reported. Some of them are difficult to analyze because they had the opportunity to perdure due to the obsolescence of the original repository mainline.

An alternative to this process is the “continuous integration”

An alternative ... the continuous integration



Continuous integration is a software engineering concept that reorganizes the workflow of a development project to improve the efficiency.

Developers still use a VCS but now the first message is “commit very often” – at least once a day. And everyday the quality checking procedure – the test suite – is applied on every branches of active contributors. When automatized by a test farm, the management software daily reports the status of the branches to the corresponding developer. The second message is “(partial) merge quite often” – on a weekly basis depending on the activity of the project. The merger works on repeatedly tested branches and produces a new mainline code to be imported on the repository. Therefore, every contributor is aware of the development of the co-workers. Furthermore, the production of a release should be smooth: the last successful merge is a good candidate.

Good practices for continuous integration

- maintain a single source repository and use a VCS.
- automate the build (include all the dependencies, use GNU automake - Ant).
- maintain automated tests (update when code changes, good coverage).
- commit and test the active branches daily.
- (partial) merge often and produce new mainlines (weekly).
- build fast (parallel or distributed compilation).
- test in a clone of the production environment (real or virtual).
- anyone can get the latest executable (for testing, demonstration).
- everyone can see what's happening (why does the code break? who made the changes?).

Fowler, Martin <http://www.martinfowler.com/articles/continuousIntegration.html>

Alain Jacques <etsf.software@gmail.com> - The ETSF Test Farm - CECAM Zaragoza June 2010



The continuous integration concept emerged from the ideas exposed in the book “Extreme Programming Explained” (1999) by Kent Beck. It is summarized in Martin Fowler’s article. Both sources insist on the automatization of the procedure – the code should build on any supported platform with a plain configure/make, the test suite should be integrated. It is advisable to perform these steps on a test farm with a suitable management engine like CruiseControl or Buildbot.

Benefits of continuous integration

- moves the stress from the merger of the code to the quality checking stage.
- early warning of bugs and conflicting code, frequent mainline updates.
- promotes modular programming.
- availability of nightly builds.
- easy diff- debugging.

Drawbacks of continuous integration

- high software investment (design and maintain a reliable test suite).
- high hardware investment (maintain a computers farm).

Continuous integration relies on efficient test farm management and automation

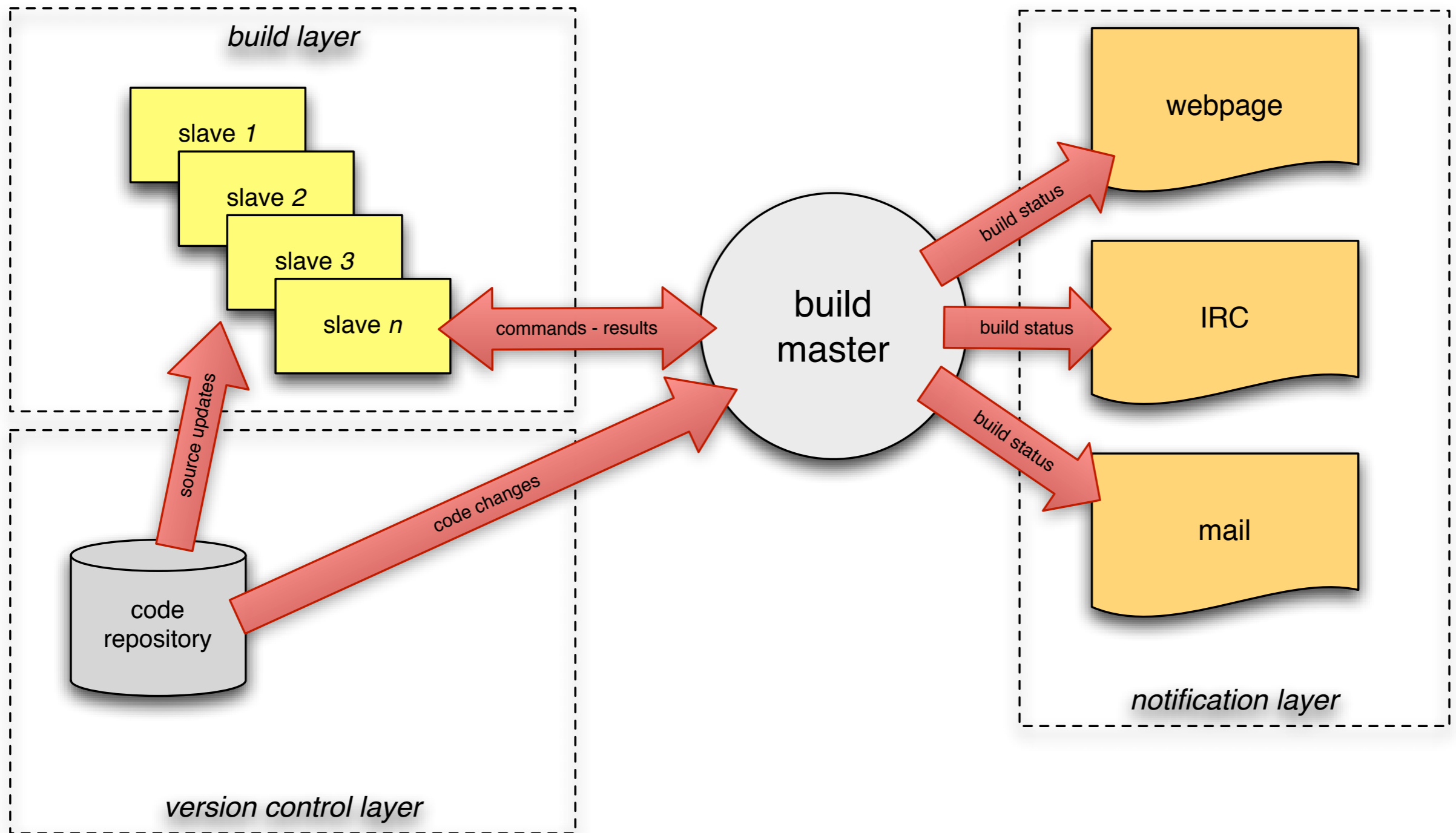
CruiseControl continuous integration framework, <http://cruisecontrol.sourceforge.net/>
Buildbot compile/test automation, <http://buildbot.net/trac>

Alain Jacques <etsf.software@gmail.com> -The ETSF Test Farm - CECAM Zaragoza June 2010



Continuous integration will not reduce the amount of bugs – the daily validation will hopefully catch even more of them. But these bugs correspond to small daily code increments and conflicts shouldn't have the time to mature. Therefore a simple differential debugging method should be enough to analyze them. The main benefit of continuous integration is that it moves the stress of the development process from a person – the merger of the code – to computers of the test farm. If 10, 20 or 50 contributors commit daily, it means that the build and test routine should be applied 10, 20 or times daily. If this figure is multiplied by the number of platforms/compiler, the test farm should be adequately configured to sustain the load. An efficient automation software like Buildbot should manage the job.

Buildbot, a test farm management software



Buildbot is a client/server application that looks for code changes, compile and test active branches on its associated slaves and report the status to the developers

Alain Jacques <etsf.software@gmail.com> -The ETSF Test Farm - CECAM Zaragoza June 2010



The Buildbot master will regularly poll the VCS repository for code changes. When this happens, the master can be programmed to rebuild the application on an available slave. The results of the compilation, test suite, ... is feed-backed to the master. The latter will notify the developers by means of a status webpage, a customized email or messages posted on an IRC server. Buildbot manages all the communications between the master and the slaves and to the authorized users; it schedules the tasks distributed to the slaves. Buildbot has to be configured to reflect the test farm topography and the characteristics of the builders. A sample procedure is described hereafter

My first test farm powered by Buildbot ...

Requirements

- Python 2.3, 2.4 and 2.5; a few benign warnings for python 2.6
- Twisted networking engine: Core, TwistedMail, TwistedWeb and TwistedWords

Compatibilities

- VCS: CVS, SVN, Mercurial, Bzr, Git, Bitkeeper
- OS: Linux, OSX, AIX, Windows, ...

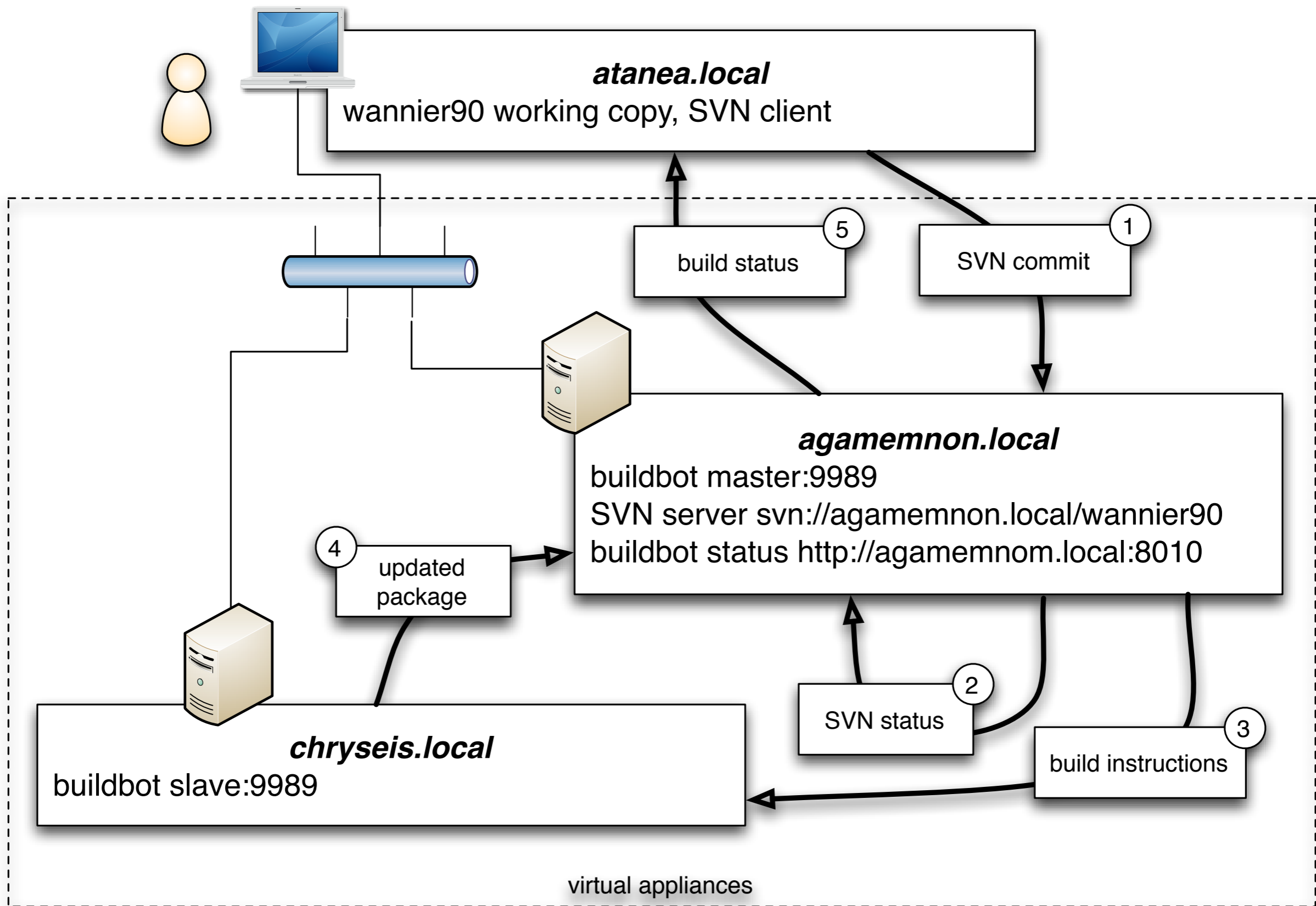
Installation procedure

- standard distutil: “python setup build”, “python setup install”
- or (Debian) “sudo apt-get install buildbot”

Python Programming Language, <http://www.python.org>

Twisted Matrix Labs, <http://twistedmatrix.com/trac/>

The requirements for Buildbot installation are quite low and attainable even for exotic computers: a Python interpreter and Twisted, an asynchronous networking toolbox in Python. It is compatible with a variety of VCS and programming environments. The installation follows the plain Python guidelines.



For this demonstration, a simplified test farm has been “assembled”. It consists of a master (agememnon) and one slave (chryseis), both running Ubuntu Lucid. A sample project – Wannier90 computing maximally localized Wannier functions – has been imported to a SVN server hosted on the buildmaster; a developer modifies a working copy of the code on his laptop (atanea). From time to time, he commits a new revision to the SVN server. The purpose of this test farm is to check the repository for codes changes and, as soon as a new revision appears, to rebuild the wannier90 executable, to test its validity and to assemble a new distribution package for uploading to the buildmaster webspace.

On *agamemnon* (master), create the accounts and projects control directories:

- create a “buildmaster” user: `sudo adduser buildmaster`
- `su buildmaster ,mkdir /home/buildmaster/Buildmasters`

Initialize the buildbot master for the wannier90 project:

- `cd Buildmasters ,invoke buildbot create-master wannier90`
... the wannier90/master.cfg template file has to be edited.

On *chryseis* (slave), create the accounts and work directories:

- create a “buildslave” user: `sudo adduser buildslave`
- `su buildslave ,mkdir /home/buildslave/BuildSlaves`

Initialize the buildbot slave for the wannier90 project:

- `cd Buildslaves ,invoke buildbot create-slave wannier90`
`agamemnon.local:9989 chryseis buildslave`

The first steps create Buildbot accounts and disk workspace on the test farm computers. Then the “buildbot create-master” and “buildbot create-slave” commands are invoked to setup the initial files and templates for the wannier90 project. The buildmaster and buildslave will communicate through the port 9989 (TCP) – to be opened in case of a firewalled environment.

On *agamemnon* (master), edit the *master.cfg* configuration file:

- `vi wannier90/master.cfg`

```
# -*- python -*-
# ex: set syntax=python:

# Defines a dictionary named BuildmasterConfig
c = BuildmasterConfig = {}

##### BUILDSLAVES

from buildbot.buildslave import BuildSlave
c['slaves'] = [BuildSlave("chryseis", "buildslave", max_builds=1)]

# 'slavePortnum' defines the TCP port to listen on
c['slavePortnum'] = 9989
```

The main configuration efforts go to the *master.cfg* file residing on the buildmaster. It consists of a small piece of Python code defining the *BuildmasterConfig* dictionary. Successive sections declare the available slaves, ...

CHANGESOURCES

```
# the 'change_source' setting tells the buildmaster
# how it should find out about source code changes.
from buildbot.changes.pb import PBChangeSource
c['change_source'] = PBChangeSource()

# uses SVNPoller to check repository changes
from buildbot.changes.svnpoller import SVNPoller
source_code_svn_url='svn://agamemnon.local/wannier90'
svn_poller = SVNPoller(
    svnurl=source_code_svn_url,
    pollinterval=30, # seconds
    histmax=10,
    svnbin='/usr/bin/svn',
)
c['change_source'] = [ svn_poller ]
```

SCHEDULERS

```
from buildbot.scheduler import Scheduler
c['schedulers'] = []
c['schedulers'].append(Scheduler(name="all", branch=None,
                                treeStableTimer=10,
                                builderNames=["buildbot-full"]))
```

... the way to poll the repository, the timing of the scheduler, ...

BUILDERS

```
# the 'builders' list defines the Builders. Each one is configured with a
# dictionary, using the following keys:
# name (required): the name used to describe this builder
# slavename (required): which slave to use (must appear in c['bots'])
# builddir (required): which subdirectory to run the builder in
```

```
from buildbot.process import factory
from buildbot.steps import source, shell, transfer
```

```
# factory (required): a BuildFactory to define how the build is run
f1 = factory.BuildFactory()
```

```
f1.addStep(source.SVN(svnurl='svn://agamemnon.local/wannier90',
mode='clobber'))
```

```
f1.addStep(shell.Compile(command=["make"], description=["compiling"],
descriptionDone=["wannier90 compilation"]))
```

```
f1.addStep(shell.Test(command=["make", "test"], maxTime=120, description=
["testing"], descriptionDone=["tests"]))
```

```
f1.addStep(shell.ShellCommand(command=["diff", "wantest.log.ref", "tests/
wantest.log"], description=["validating"], descriptionDone=["tests
validation"]))
```

... and the properties of the unique builder of the farm.
In the builder section, a factory describes the sequence of operations to download the code, compile and test the new revision that triggered the scheduler; the factory steps are transmitted to the active slave by the buildmaster.

```

f1.addStep(shell.ShellCommand(command=["make", "thedoc"],
    descriptionDone=["documentation"]))

f1.addStep(shell.ShellCommand(command=["make", "dist-lite"],
    descriptionDone=["distribution package"]))

f1.addStep(transfer.FileUpload
    (slavesrc="wannier90.tar.gz",masterdest="public_html/
    wannier90_fromChryseis.tar.gz"))

b1 = {'name': "buildbot-full",
    'slavename': "chryseis",
    'builddir': "full",
    'factory': f1,
    }
c['builders'] = [b1]

```

On *chryseis*, the following commands would have been entered to compile and test the code:

- `svn checkout svn://agamemnon.local/wannier90`
- `make , make test ,...`

The factory reproduces the same procedure as it would be entered on the slave's terminal prompt ...

```

svn co svn://agamemnon.local/wannier90
make
make test
diff wantest.log.ref tests/wantest.log
make thedoc
and so on ...

```

STATUS TARGETS

'status' is a list of Status Targets. The results of each build will be
pushed to these targets. buildbot/status/*.py has a variety to choose
from, including web pages, email senders, and IRC bots.

```
c['status'] = []
```

```
from buildbot.status import html  
c['status'].append(html.WebStatus(http_port=8010, allowForce=True))
```

PROJECT IDENTITY

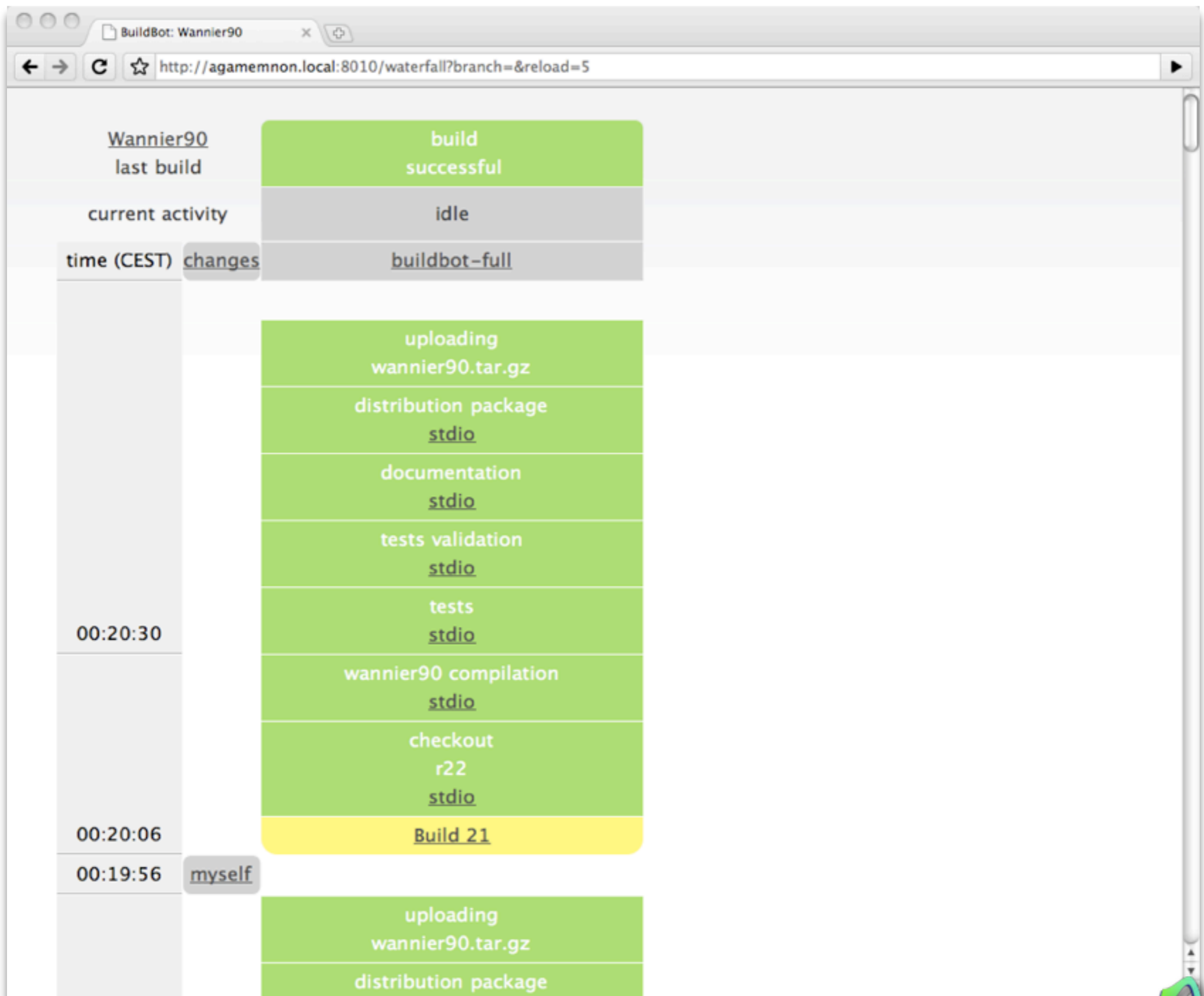
the 'projectName' string will be used to describe the project

```
c['projectName'] = "Wannier90"  
c['projectURL'] = "http://www.wannier.org/"
```

the 'buildbotURL' string should point to the location where
the buildbot's internal web server

```
c['buildbotURL'] = "http://agemnon.local:8010/"
```

The status is reported to the user by means of dynamic HTML pages updated with the output of the slave's stdio, error messages, ... It is presented by grid and waterfall displays of the builder's activity.



Alain Jacques <etsf.software@gmail.com> -The ETSF Test Farm - CECAM Zaragoza June 2010



For example, the “waterfall” display of the test farm after a successful build as indicated by a green background.



Alain Jacques <etsf.software@gmail.com> -The ETSF Test Farm - CECAM Zaragoza June 2010



In this case, the validation step – the “diff” between the actual tests output and a reference file – went wrong. It is colored in red. Clicking on the red link shows extra debugging information.

A large scale project powered by Buildbot - the ETSF farm

- the initial hardware investment > 40.000 EUR (not including the hosting environment and software licensing fees).
- the farm is serving Abinit and several sister projects related to first principles applications - DP, Exc, Octopus ... and more to come.
- tens of active contributors are registered (Abinit). Each developer receives a private, a public and a training branch.
- for Abinit, several mainlines are active : a production release and a “bleeding edge” development code.
- 12 different builders are currently defined (Abinit) on a variety of platforms and operating systems.
- all active branched are systematically tested daily; there is an extra “on-demand” service.
- Abinit consists of about 450.000 lines of code as of version 6.2 and its validation suite contains about 800 test cases.

DP linear response TDDFT code, <http://dp-code.org/>

Exc exciton code, <http://www.bethe-salpeter.org/>

Octopus, http://www.tddft.org/programs/octopus/wiki/index.php/Main_Page

Alain Jacques <etsf.software@gmail.com> -The ETSF Test Farm - CECAM Zaragoza June 2010



A few facts about the ETSF test farm to highlight the scale of the facility.

ETSF test farm hardware inventory

Slave	Brand	CPU	Processor	RAM	OS	bots
testf	Bull R423-E2	Intel Xeon	2x Quad	12 GB	CentOS 5.3	2
chum	Sun X4200M2	AMD Opteron	2x Dual	32 GB	CentOS 5.3	1
green-bb	Dell PowerEdge	Intel Xeon	2x Quad	16 GB	SLinux 5.3	2
bigmac	Apple Mac Pro	Intel Xeon	2x Quad	6 GB	OSX 10.5	2
chpit	HP rx4640	Intel Itanium2	4	8 GB	Debian 5.0.1	1
fock	IBM OP 720	IBM Power5	2x Dual	32 GB	Suse 9.3	1
max	Apple XServe	IBM PowerPC G5	18x Dual	18x 4 GB	OSX 10.4	1
buda	SuperMicro	Intel Xeon	2x Quad	12 GB	CentOS 5.4	1
ibm6	IBM OP 520	IBM Power6	2x Dual	8 GB	AIX 6.1	1

Quite a diversity of platforms : desktop, servers as in cluster environments. Several variants of Linux, OSX, AIX to test the code in a clone of the actual production environment.

ETSF test farm software environments

Builder	g95	gfortran	ifort	Open64	PGI	Path-Scale	Sun-Studio	XLF	MPI
testf		4.4	11.1						openMPI 1.3
chum	0.9	4.2, 4.3	9.1, 10.1	x	7.3	3.2	12		openMPI 1.2, 1.3
green-bb	0.9	4.2, 4.3	10.1						openMPI 1.2, 1.3
bigmac		4.3, 4.4							openMPI 1.3
chpit		4.4	11.1						openMPI 1.3
fock								9.1	mpich2 1.0.8
max	0.9								openMPI 1.3
buda		4.3	11.1						mpich2 1.2.1
ibm6								12.1	poe

Live demonstration ...

- Abinit: <http://www.abinit.org> and <http://buildbot.abinit.org>
- Octopus: <http://www.tddft.org/programs/octopus/buildbot/>

... depending on the ZCAM network goodwill.

BuildBot: Wannier90 x BuildBot: Octopus x Latest Build

http://buildbot.abinit.org/one_box_per_builder

Latest builds:

testf gcc44	#1488 build successful	idle
testf gcc44_serial	#1227 build successful	idle
green intel10_sernoplug	#355 build successful	idle
green_g95	#1202 build successful	idle
coba2 gcc44_noplugs	#809 build successful	idle
chum_psc	#1215 build successful	idle
biqmac gcc43	#1243 build successful	idle
chpit_intel11	#1252 build successful	idle
buda gcc43 mpiio	#592 build successful	idle
inca gcc44_sdebug	#637 build successful	idle
ibm6_xlf12	#388 build successful	idle
fock_xlf_sernoplug	#352 build successful	idle

To force a build on all Builders, fill out the following fields and push the 'Force Build' button

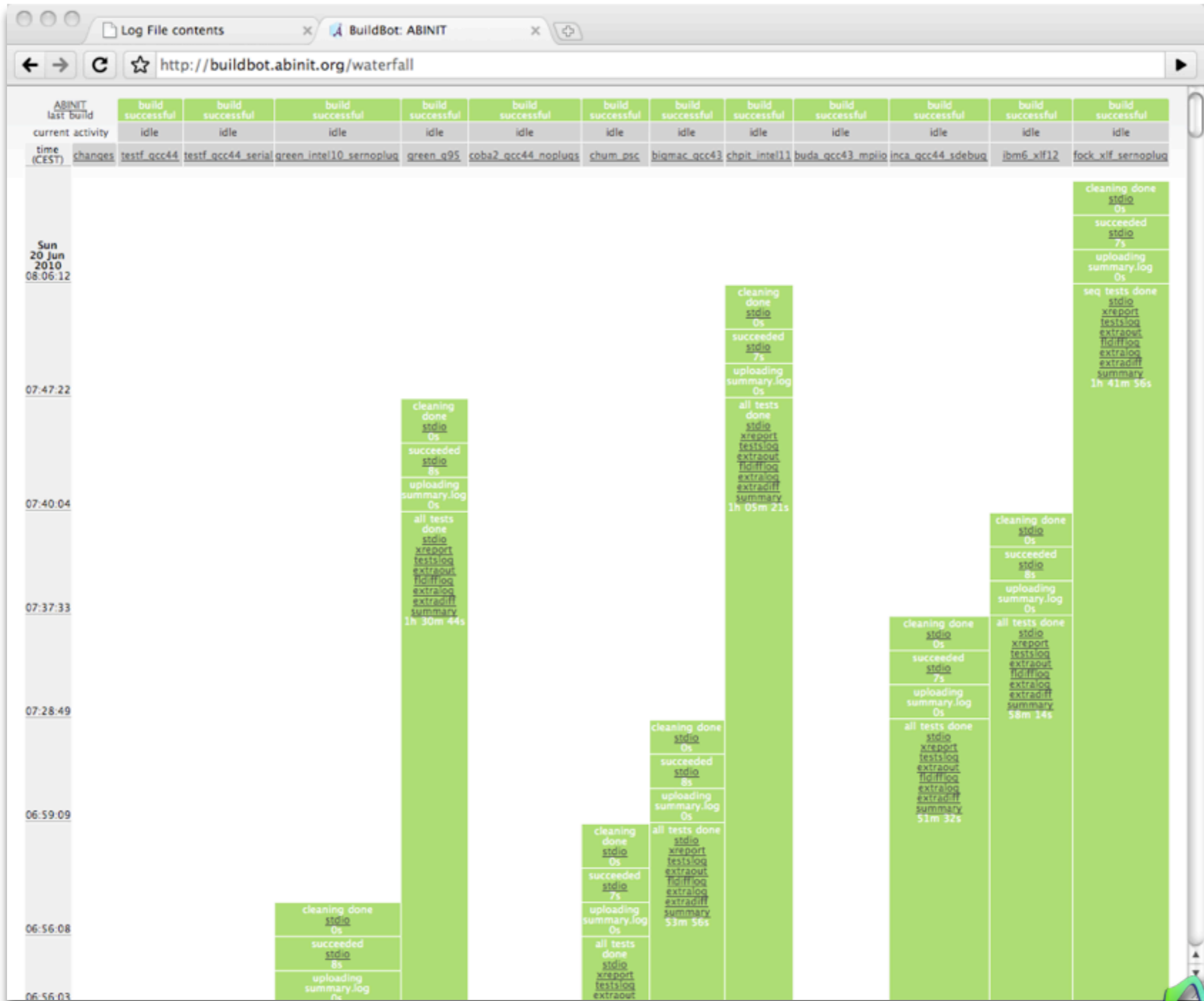
Your username:

Your password:

Alain Jacques <etsf.software@gmail.com> - The ETSF Test Farm - CECAM Zaragoza June 2010



This grid displays the available builders of the ETSF test farm working on Abinit code



Alain Jacques <etsf.software@gmail.com> -The ETSF Test Farm - CECAM Zaragoza June 2010



The “waterfall” display of the ETSF test farm showing jobs sorted by time.