Introduction to the Dune autobuild system

Christian Engwer

Institute of Parallel and Distributed Systems, University of Stuttgart

23 January 2007

Introduction

- Software projects grow bigger.
- Developers can't be sure that a change in `part1` does not break `part2`.

→ *Constant integration* systems allow to run unattended tests against the software.

→ `dune-autobuild` is the *Constant integration* or *Automated Test* system from Dune.

Requirements

- We have different scenarios that need to be tested, i.e.
  - full builds once a day,
  - quick tests after each source change,
  - builds of the development branch
  - and of the release branch.
- The tests should be run on different architectures.
- Resources are limited and must be shared among different tests.
- The system should have a minimal set of dependencies, so that it runs on a wide range of architectures.
- Results should be available via a web page.

Requirements

- We have different scenarios that need to be tested, i.e.
    - full builds once a day,
    - quick tests after each source change,
    - builds of the development branch
    - and of the release branch.
- The tests should be run on different architectures.
- Resources are limited and must be shared among different tests.
- The system should have a minimal set of dependencies, so that it runs on a wide range of architectures.
- Results should be available via a web page.

Requirements

- We have different scenarios that need to be tested, i.e.
  - full builds once a day,
  - quick tests after each source change,
  - builds of the development branch
  - and of the release branch.
- The tests should be run on different architectures.
- Resources are limited and must be shared among different tests.
- The system should have a minimal set of dependencies, so that it runs on a wide range of architectures.
- Results should be available via a web page.

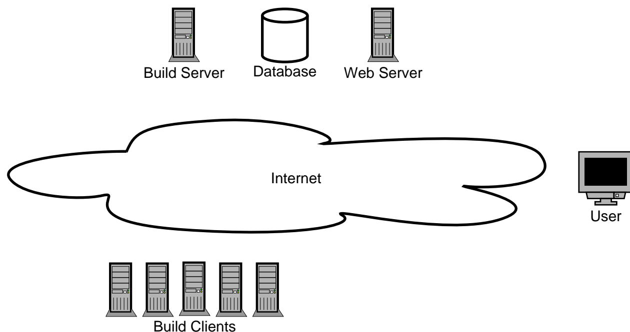
Requirements

- We have different scenarios that need to be tested, i.e.
  - full builds once a day,
  - quick tests after each source change,
  - builds of the development branch
  - and of the release branch.
- The tests should be run on different architectures.
- Resources are limited and must be shared among different tests.
- The system should have a minimal set of dependencies, so that it runs on a wide range of architectures.
- Results should be available via a web page.

Requirements

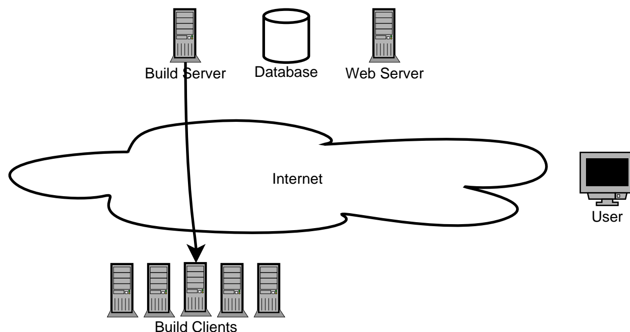
- We have different scenarios that need to be tested, i.e.
  - full builds once a day,
  - quick tests after each source change,
  - builds of the development branch
  - and of the release branch.
- The tests should be run on different architectures.
- Resources are limited and must be shared among different tests.
- The system should have a minimal set of dependencies, so that it runs on a wide range of architectures.
- Results should be available via a web page.

Design



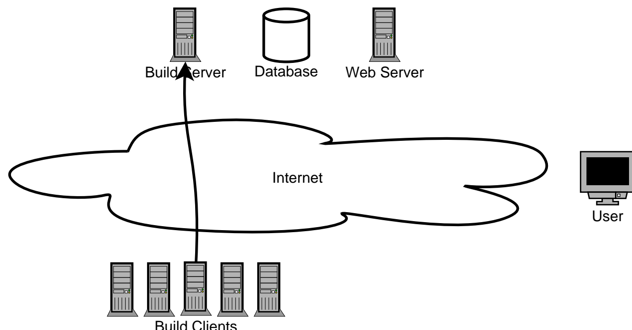
Setup consists of a central *Build Server* and several *Build Clients*.

Design



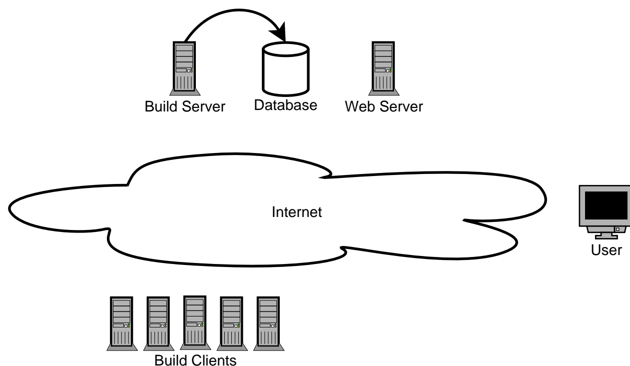
The *Build Server* opens a secure connection to the *Build Clients* via `ssh` and public key authentication. The server send a configuration for a single run, including which tests to run, etc.

Design



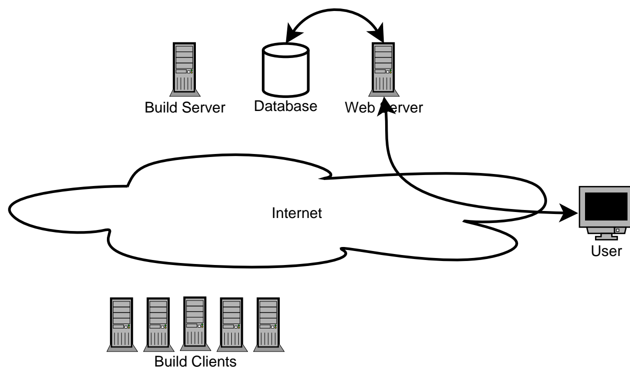
During the build the *Build Clients* writes a set of log files in a certain format. After the build is finished the *Build Clients* open again an `ssh` connection to the *Build Server* and pipe an archive including the log files to the server.

Design



The *Build Server* feeds the log files to database, which allows asynchronous access.

Design



User can query the web server `dune-project.org` for build results. A dynamic web page shows the information, gathered from the database.

The dune-autobuild Components

`autobuild-enqueue` *runs on server side.* Register a job for execution.

`autobuild-qrunner` *runs on server side.* Try to execute previously registered jobs once the resources are available.

`autobuild-client` *runs on client side.* Receives a configuration from `autobuild-run` and runs a set of tests. The results are sent back to the server.

`autobuild-receive` *runs on server side.* Receives an archive with log files from the client.

`autobuild-store` *runs on server side.* Parses the log files and feeds the data into a database

autobuild-enqueue

- Enqueue a certain job to the Queueing System, specified via its *tag*.
- A *tag* specifies a certain configuration (i.e. what should be built, and on which hosts). The *tags* configurations are stored in `tags.d`.
- Each host has a configuration in `hosts.d`, specifying how to reach the host and which compiler etc. to use.
- `autobuild-enqueue` merges *tag* and *host* configuration, and marks the resulting configuration as *wait*. If there are already equivalent requests waiting, these old requests are removed.

`autobuild-qrunner`

- Runs every *n* minutes.
- Jobs marked as *wait* are changed to *trigger*.
- Jobs marked as *trigger* are spawned.
- `autobuild-enqueue` connects to the clients via `ssh` and sends the job configuration.
- If the jobs host is not available, because an other is job is already running, the job remains in state trigger.

autobuild-client

- autobuild-client is started on the incoming ssh connection.

Note: the ssh client is configured such that the server is not allowed to start any other command than autobuild-client.

- The configuration is read from stdin and written to a config file.
- autobuild-client looks for an updated version of itself and restarts.
- All scripts in targets.d are executed. The results are written to log files in spool.
- autobuild-client opens an ssh connection to the *Build Server* and sends all logs as a compressed archive.

autobuild-receive and autobuild-store

- `autobuild-receive` is started on the incoming `ssh` connection.
- Generates a directory name, acquires a lock and creates the directory in the `spool` directory.
- Extracts the received log archive to the new directory. Once it is finished, an entry in `store` signals the existence of this log set.

Note: `autobuild-receive` does not release the lock, this will be done by `autobuild-store`, once all log are processed.

- `autobuild-store` is run periodically and looks in `store` for new log sets.
- The log set is parsed and the log file data is stored to the database.
- Once everything is stored the log files are removed and the lock is released.

Writing tests

- User can write tests in the usual `autotool` fashion.
- Tests are programs or scripts.
- Failure and Success are determined according to the exit code.
- System allows to specify tests that must succeed and tests that must fail.
- With the current system it is very difficult to write tests that must not compile, because the compilation is a prerequisite for test itself.
- Tests are listed in the `Makefile.am` under `TESTS` and `XFAIL_TESTS`
- `check_PROGRAMS` lists all programs needed for the tests.

Accessing the results

- Results are presented in a hierarchic structure.
- Directory entries sum up errors and warnings of all entries.
- Each test shows the amount of errors, warnings and whether it did run in the last run.
- For all non directory entries a time line can be accessed to see when the test started to fail.
- The output of all tests can be inspected. Errors and warnings get highlighted.

FAQ

Why not use an existing tool? We wanted to be able to use the test environment included in the `autotool` tool chain. This was not possible with existing tools, because these only allow one result per operation. This would require us to add each single test to the system.

Why not connect directly to the database? `autobuild-client` should have minimal dependencies except the POSIX system. Including a database client to the client environment would increase the dependencies significantly.

FAQ

Why not use an existing tool? We wanted to be able to use the test environment included in the `autotool` tool chain. This was not possible with existing tools, because these only allow one result per operation. This would require us to add each single test to the system.

Why not connect directly to the database? `autobuild-client` should have minimal dependencies except the `POSIX` system. Including a database client to the client environment would increase the dependencies significantly.

Open Problems

No direct support for test sets For different test runs we would like to have different test sets. I.e.

- *nightly* build should build all
- *head* build should only build latest `svn` version
- *release* build should build the release code from `svn` and from the tar balls.

A broken test can block the whole build Currently the system only has one global timeout for the client.

Better integration with the web site In order to build the website we also need to compile the different `dune` modules. Integrating the website build into the continuous integration process would

- lower the system load,
- help debugging website problems.