

Build farms

Armijn Hemel

June 27, 2003

Abstract

This paper describes build farms, collections of environments to perform actions (build, test, benchmark) on software and how build farms can help developers. Problems encountered with current build farm software and possible solutions for these problems are described.

1 What is a build farm?

A build farm is a collection of environments, with different configurations (hardware, operating system, libraries and tools) to perform tasks on a program. One of these tasks is building for portability, but a build farm is not restricted to finding portability problems. Other tasks that could be run on a build farm include benchmarking and integration testing.

Portability becomes more and more of an issue. People want to be able to run the same software on all of their platforms, no matter what hardware they use and what operating system runs on it.

Especially in the free software world portability is an issue. The source of many programs is freely available and because it is free it can be ported. With free software you have the possibility to port software to whatever platform you want, without being restricted by the original author or company that wrote it. With closed software you don't have this possibility, the author decides on which platforms it runs. Porting the software however is not as easy as it sounds.

Looking at Linux, there are several hundreds of distributions (with a lot of similarities, but also with lots of tiny and not so tiny differences), it runs on about 15 architectures (and probably more will be added over time), ranging from tiny handhelds to mainframes. Porting an application just to all flavours of Linux is already a massive effort. And that is just Linux.

Porting is a problem for many developers because they usually lack the infrastructure to test their software on different hardware and software platforms.

Being able to test the software automatically in different environments with different configurations would be a major benefit for developers. This is what a build farm provides.

On the Internet there are a few build farms with different environments available for developers [1, 2].

1.1 Common porting problems

Some common portability problems are:

- hardware differences
- kernel differences
- library and header file differences
- tool differences

Eric Raymond described portability of programs in a broader context in [3].

1.1.1 Hardware differences

The most common errors due to hardware differences are 32 bit versus 64 bit and endianness. On 32 bit architectures (or at least on the x86 architecture) pointers and integers in C are both 32 bit. Pointers look a lot like integers and on a 32 bit platform they can be used in place of one another (even though it is wrong, because integers and pointers are different data types). On 64 bit platforms pointers are 64 bit, but normal integers are still 32 bit. Trying to squeeze a 64 bit pointer into a 32 bit integer is not a good idea, because half of the memory address in the pointer is simply lost.

Endianness can play a role in reading in data files or other operations that involve manipulating bits.

1.1.2 Kernel differences

Using system calls can be a problem if data structures or system calls are used that exist only on one OS (such as the `sysinfo` system call on Linux), or are different on each OS (like `/proc` filesystem).

1.1.3 Header file and library differences

The most common portability error is due to header file differences and library differences.

The GNU C library (glibc) is a good example of where things can get tricky. glibc provides a lot of functionality by default, more than C libraries on other platforms. Things that work out of the box with glibc might require additional libraries to be installed on other operating systems.

This is usually not much of a technical problem, because the libraries are usually available on other platforms, but it is a problem at configuration and compile time (there should be checks for in the configure scripts) and for documentation.

1.1.4 Tool differences

Another problem, which doesn't show up on most Open Source/Free Software platforms (Linux, Hurd and *BSD) is using non-standard C/C++ constructs. The default compiler on these platforms (GCC) has various extensions that make it a lot easier for programmers, but which are not recognized by most other compilers (however, the Intel compiler supports most constructs).

Another problem is that compilers on some platforms are not ANSI C compliant. The default HP-UX compiler (the one that is installed with the operating system, not the one you buy) is an example of a non-ANSI C compliant compiler.

2 In theory

Olsson and Karlsson described iterative development, automatic testing and responsibility based on features as the major cornerstones of a daily build system [4].

Furthermore they concluded that a build farm is advisable for projects with:

- poor project control
- low quality
- excessive lead times

However, not just projects that have gone completely off track can benefit from using a build farm. On the contrary, Martin Fowler said that it is essential for a controlled project [7].

In [4] and [7] it is described that an organization should have the following infrastructure in place to be able to deploy a build farm:

- version control system
- automated build process
- automated tests

Such an infrastructure is considered mandatory for the so called "agile methods", methods for software engineering which have become popular recently, such as eXtreme Programming (XP) [5, 6].

XP has the concept of continuous integration [7], where programmers integrate their changes at least once a day (but usually more often). This integration is done by hand and when all the pre-defined tests have run successfully it means the integration is successful and the new code hasn't broken the program. If it does break the tests, the team should either take out their changes, or fix it.

XP demands using unit testing and test-first programming exhaustively using automated tests [17].

In [7] automation of continuous integration using CruiseControl [8] is described. CruiseControl is a tool which checks a source code repository regularly for newly checked in code, if there is new code it checks it out, builds it, makes a report and starts checking for new code check-ins again.

3 In practice

In [7] the following was stated:

“Developing a disciplined and automated build process is essential to a controlled project. Many software gurus say that, but we’ve found that it’s still a rarity in the field.”

Even though build farms aren’t in as widespread use as they maybe should be, quite a few organizations and Free Software projects use build farms productively. Projects such as Mozilla, Debian and Samba, institutions like CWI (Centrum voor Wiskunde en Informatica) in Amsterdam and companies like Microsoft and Ericsson use it.

Each of these organizations use build farms differently and have different expectations of a build farm. It is not a surprise that all of these organizations use custom-built tools, with different behaviour, functionality and flaws.

3.1 Build farm requirements

A set of basic requirements for a build farm would be:

- build environment
- management of downloaded sources and installed packages
- log processing & visualization
- build control (starting, stopping)
- build file specification language

It should be noted that in an ideal situation these requirements are not implemented in one piece of software, but that they are rather implemented “the UNIX way” (small tools which can be easily replaced by other tools with the same functionality).

3.1.1 Build environment

A build should be done in an environment that can be specified, or at least be described. An environment is the collection of operating system, tools, header files and libraries in which a package gets built.

3.1.2 Build control

Build control is the ability to stop and start builds easily on one machine, all machines or a number of machines. A build farm should provide this, either for the build owners (the developers) or the administrator(s) of the build farm.

This requirement can conflict with security policies. The question that should be asked when designing a build farm is “Who should be in control?”. For developers it is a must to be able to start and stop builds (or at least to have them scheduled for building), but there are limits (from a system administrators point of view). An example where builds can go wrong is in the case of a “rogue build”, where a build takes up too many resources (probably unintentionally) for the build farm to function properly.

As always, starting and stopping processes on machines should be monitored closely and executed with as little permissions as possible. Unfortunately, this clashes with build tasks which are quite common on a build farm, such as testing if a freshly made binary distribution actually installs, which often requires root privileges.

3.1.3 Log processing & visualization

A build farm has no use if the results can't be reported to developers. Good log processing and visualization of logs is perhaps the most important part of a build farm. Without good visualization of logs the build farm is of no use to developers. A log visualization system should give instant feedback if builds have failed and on which platforms, but leave the option open for developers to consult the complete log.

3.1.4 Build specification language

A build specification language is a language in which build tasks can be described.

A build specification language doesn't have to be newly designed, shell script (or a subset) could be sufficient (even though shell scripts do have their limitations).

Note that the developer view of the language and the way the client sees it doesn't have to be necessarily the same. Developers and clients have different needs. Clients need this language to be simple enough to be easily interpreted or executed, but developers need more advanced language constructs, such as abstraction and reuse of variables, functions, etcetera.

3.2 Samba build farm

The build farm of the Samba project is a lightweight distributed build farm [9]. It is used to build the latest CVS sources of Samba (various CVS branches) and related tools, such as `rsync`. The build farm clients are started from cron, download the latest CVS sources with `rsync`, build and send the build logs

back to the server with `rsync`, where they are processed and presented on the Samba web site.

The Samba team uses the build farm mainly as an occasional reference, rather than to perform smoke tests. The control over the build farm is not very strict, because lots of build farm machines are “dead”.

Every 10 minutes a fresh CVS checkout is done from the directory that is exported via `rsync`. This is not an atomic operation, sometimes the CVS checkout is still running when a client tries to get the new sources and `rsync` fails to get the new sources.

Representation of logs is a bit primitive, as it only shows the results of the last build and older logs cannot be consulted.

The Samba build farm does not have many prerequisites. It assumes there is a user called “build” and makes no further assumptions about the environment. Before a build starts no environment checking is done and it is possible to change the environment between builds (making the logs unreliable).

3.3 Mozilla Tinderbox

The build farm of the Mozilla project [10] is one of the more famous build farms. It uses a more or less generic framework, where machines distributed around the world keep building the same source over and over again. A script that runs all the time checks out the Mozilla source code from CVS, builds the source, sends a report e-mail with logs, waits for a few minutes and then starts over again.

The build “server” parses logs and consults Bonsai [12], to be able to relate broken builds to particular check-ins in CVS and to be able to “blame” the developer who checked in that code (and whose code broke the build).

Tinderbox is strongly tied to CVS, Bonsai and the whole Mozilla project. Tinderbox2 [11] provides a somewhat more generic framework with a bit more support for other version control systems.

Before a build starts no checking of the build environment is done. The fact that e-mail is being used for reporting, which is not guaranteed to be delivered instantly makes the logging not reliable.

The README files in the source directory of Tinderbox 2 are an interesting read about the goals of Tinderbox and how it is used within the Mozilla organization.

3.4 DailyBuildSystem

The DailyBuildSystem [13] is a generic build framework. Unlike the other solutions it is not meant to run in a distributed setup, but more on a local LAN. It depends on NFS to share the home directory of the special build user, optionally uses some NIS functionality, Perl and SSH.

A build is started from cron from a special configuration file which defines which builds should be done. For each build the script launches itself on all the hosts that should execute this build via SSH, and writes its logs back in its home directory (the location is configurable).

Actions for each build can be defined in a build file and overridden again per platform, variables can be defined and used throughout the build file. The build file comes down to Bourne shell code, which is executed. Because it is just Bourne shell code basically anything that can be expressed in shell code can be performed as a task.

The system waits for a build to finish on every machine it runs on, before it launches the next build. This means that slow machines tend to slow down builds. Only the last logs of builds are kept, previous logs are overwritten by default (it can be easily tweaked to keep older logs).

Because DailyBuildSystem depends on having a LAN (due to NFS) it is not easy to distribute it over various administrative domains (such as the Internet). The fact that tasks are started by using SSH doesn't make this easier.

3.5 Debian Buildd

Debian buildd [14] is a collection of machines with different architectures that have only one goal: creating binary Debian packages (.deb). Before each build a clean environment is set up by installing the necessary dependencies by the package manager in a `chroot` environment. When all dependencies are built, the package is built.

4 Build farm gotchas

The build farms described above suffer from several flaws, which make deploying a build farm tricky, especially if you want the build farm to perform extra tasks (benchmarking, package creation, blame system, fancy logging).

Some common gotchas are:

- build environment is not fixed
- synchronous builds versus asynchronous builds
- suboptimal log processing and representation
- no time syncing

Gotcha #1: Build environment is not fixed

Nearly all build farm software solutions don't put any restrictions on the build environment on the client. It is possible (and not even considered an error) to change software that is installed on the client after it has been added to the

build farm. Upgrading a piece of software that's being used during the build turns the build environment in a different environment than there was when the client was added to the build farm. This has some serious side effects. Tracking builds over time essentially becomes useless since the environments can't be properly compared. Problems with upgrades of software might only show up later and cause that bugs are attributed to commits that in fact did not cause those bugs. An example could be that you upgraded a piece of software in the build environment to an experimental version, and after three weeks you add some functionality to your package, which breaks the build due to a bug in the upgraded piece of software. After three weeks it's still possible to remember, but the longer it takes, the easier it is to overlook these things.

Another situation is that build farm clients aren't even under control by the developers, but by other people (this is true for the Samba and Mozilla build farms) and the maintainers can just decide to upgrade some piece of software.

Related to this problem is the management of software on build farms. Adding a new build task to the build farm might require that new software is installed. The correct way to install this in all the appropriate environments is by using the native package manager. This can be very time consuming, but using the native package manager is the way to guarantee that the build farm environment is similar to what users of the software that is being built will be using.

Gotcha #2: Synchronous builds versus asynchronous builds

Some build farm software starts builds from cron. Even though it seems like a good idea to use a mechanism that starts builds synchronously it's not a good idea at all. If a build is still running when another build is started, who knows what happens? A build might fail unexpectedly, or even worse, might all of a sudden succeed when it should fail instead. Another possibility is that builds are eating up all resources (for example the number of open files in the system) and the only way to recover from it is a reboot.

Apart from builds overwriting each other it is also very difficult (if not impossible) with cron to optimally schedule builds and use resources efficiently. Building packages only once a day is waste of resources if the package builds very quickly, leaving the machine idle the rest of the day.

Scheduling of a one-time build becomes very awkward with synchronous builds.

Gotcha #3: suboptimal log processing and representation

The hardest part of a build farm is presenting the output to the user. In this case presenting just the raw logs is not sufficient. Logs usually are rather verbose and a lot of information is not particularly relevant to builds. Each build farm has a tailored view of the build logs. Most build farms only present the log of the last build. Mozilla Tinderbox presents logs over a longer period of time (12 hours or 24 hours), with the possibility to go back further in time. There are situations when it is desirable to have more than just the basic

representation of whether the last build succeeded or not.

Gotcha #4: No time syncing

Keeping machines in time sync with each other is a must. The builds usually don't suffer from it, but the reporting mechanism needs machines to be in sync, especially if you want to use some blaming mechanism (like Mozilla Tinderbox uses). Logs from machines that are not kept in time sync and then processed for a blame system will confuse that same system, because the time reported does not reflect the time of the build. It could be that because a machine's time is in the future a bug goes unnoticed, but is attributed to a commit that was done later.

Problems can be even more subtle. The `make` program makes decisions based on timestamps whether or not a target is up to date. If dependencies for targets that you don't want to be rebuilt are newer than the target, it will be rebuilt. This can for example happen if the sources are on an NFS partition. If files are written or read the timestamp of the NFS server is used. Even a few milliseconds difference is enough for `make` to decide to rebuild a target.

5 Improvements

For most of the problems described above there are solutions. Most of these solutions will make build farm software more complex.

Solution #1: Check environment before running a build

Before starting a build the build environment should be checked to see if the environment is the same as expected and maybe the build should not be done if the environment has been changed. Assuming it is known in advance what environment is expected (recorded on the server when the client was added) there are various ways to check.

One way is to use TripWire to see if anything has changed. Another way is using just MD5 sums on the files. Another interesting, but time consuming method is to use an `autoconf` generated configure script to pre-check a build environment.

The advantage of TripWire is that it can do thorough checking, including checks for appropriate permissions, which in case of scripts and other executables is an advantage.

MD5 hashing has the benefit that it is a fast algorithm and is implemented for nearly every programming language. With `autoconf` scripts it's possible to check for all kinds of things, such as if programs that are needed are installed, if certain functions exist in libraries, etcetera.

The restriction that an environment is checked before a build starts implies that for every change in the build environment that matters to the build, no

matter how small, requires a new environment. Most build farm uses one environment per machine. Adding a new machine for every change is a bit overdone (not mentioning financial and infrastructural problems). Consolidating several similar environments on a machine becomes a must. A solution for consolidating environments is to use `chroot` environments. This does open up potential security holes if no care is taken, but it is the only solution that is portable between all UNIX and UNIX-like operating systems (and Cygwin) and which provides a strict separation between the different environments. Another additional benefit is that tasks which normal users can't perform (such as installing packages via the package manager, which usually requires root privileges) now can be done and tested. Special care must be taken that processes can't break out of their `chroot` environment and wreak havoc on a system.

Another solution would be tweaking the `PATH` but this does not guarantee anything about which packages will eventually be used (even though deployment systems like Nix [15] make it easier) or using jail (FreeBSD), User Mode Linux (Linux) or VMWare (Linux, Windows).

The pleasant side effect of using `chroot` is that an environment can be stored as a tarball after it is set up, and that MD5 checksumming is used on this tarball, instead on all the files in the environment as an optimization.

`chroot` also has the added benefit that within the `chroot` environment the native package manager can be used. This is the approach that Debian build takes.

Management of installing software also becomes a bit different with `chroot`. Adding a new piece of software in an environment effectively means changing an environment, so it's better to create a new environment. This transforms the problem of managing installing software into the problem of creating new environments automatically. This is a very platform specific problem that needs further research.

Solution #2: schedule builds dynamically

Solving the problem synchronous builds suffer from is to do asynchronous builds by scheduling builds dynamically. The Mozilla build farm has a good basis for this. In the Mozilla build farm clients build in a continuous cycle, building the same thing over and over again. This is already an improvement (clients don't spend much time idling), but not yet optimal. The clients keep building the same thing. Even if there haven't been any commits in the source tree, the clients do a checkout of the source and build it. In the same time the client could have done something more useful, like building another program.

Taking away knowledge of what to build from the client and deciding on the server what the client should build next after a build is finished, allows for one-time builds or rescheduling of important builds (for example, to generate new binary distributions after applying a security patch). A scheduler on the server needs to have more intelligence than servers in the current build farm software implementations do and the client software needs to be more generic than it is

now.

Having a scheduler does not mean the server is in full control and starts and stops builds on clients, using `rexec` (or better `ssh`) or using some trigger to control builds (e-mail, some web service). Using a pull mechanism for a build farm, where the client would download the build specification itself (via HTTP?) is very well possible. Which solution is best depends on the architecture of the build farm (distributed versus local, security considerations).

Benefit of using a pure push mechanism (where the server is in control) is that build control can be very fine grained (builds can be stopped/started/resumed easily), whereas this is not the case with a pure pull mechanism. The pull mechanism has the advantage that the clients are more independent and can easily get build tasks from more than one server. As always the truth probably lies somewhere in between and depending on the exact build farm requirements it will be more push or pull.

Solution #3: No presentation logic on client side

Presenting logs is a tough problem, because every tool formats its output differently. To be as flexible as possible a client should not do any formatting of the build logs, with perhaps an exception for time stamps and leave the formatting for another program. Because there are so many different needs for visualization of logs there is no “best solution”. Therefore it is imperative that logs should not be polluted with presentation dependent things (HTML markup for example).

Solution #4: run NTP often (or continuously)

To avoid machines being out of sync, you have to sync them often. The best way to do this is with the Network Time Protocol (NTP), at least before and after a build, to keep time differences minimal. NTP can also run in continuous mode and take system clock drift into account. Running NTP in continuous mode would be the best solution.

6 Further research

Apart from building source code continuously a build farm can also be used for other tasks, such as benchmarking programs (making the name “build farm” a bit of a misnomer). This makes it even more important that build farm software is modular and flexible.

An interesting application is to combine binary packaging of successful builds (creating RPM or DEB packages) with advanced installation and upgrade mechanisms, such as Ximian Red Carpet [16], providing updates and new versions of packages instantly after a build has completed.

Another area of research would be representation of build logs, especially logs

gathered over a long period of time to try to find parts in a program that give problems with portability. The challenge in data-mining build logs is that the log data are highly unstructured and application dependent.

A front-end for a build farm would be useful, for example to be able to inject build tasks easily (one-time build), stop/start builds, monitor builds or delete scheduled tasks (taking into account permissions). Now an admin still has to hack many scripts and configuration files to add machines or build tasks. A web based front end (via web services?) seems to be a good choice for this.

7 Related work

Other generic build farm solutions that need further research are [18] and [19].

References

- [1] <http://sourceforge.net/>
- [2] <http://www.testdrive.hp.com/>
- [3] <http://www.catb.org/~esr/writings/taoup/html/portabilitychapter.html>
- [4] Kent Olsson, Even-André Karlsson, *Daily Build*, <http://www.sectra.se/spin/dailybuildreport.pdf>
- [5] Kent Beck, *Extreme Programming Explained*, Addison-Wesley
- [6] <http://www.extremeprogramming.org>
- [7] <http://www.martinfowler.com/articles/continuousIntegration.html>
- [8] <http://cruisecontrol.sourceforge.net/>
- [9] <http://build.samba.org/>
- [10] <http://www.mozilla.org/tinderbox.html>
- [11] <http://www.mozilla.org/projects/tinderbox/>
- [12] <http://www.mozilla.org/bonsai.html>
- [13] <http://www.program-transformation.org/twiki/bin/view/Tools/DailyBuildSystem>
- [14] <http://buildd.debian.org/>
- [15] <http://www.cs.uu.nl/groups/ST/twiki/bin/view/Trace/Nix>
- [16] <http://www.ximian.com/products/redcarpet/>

- [17] Kent Beck, *Test-Driven Development*, Addison-Wesley
- [18] <http://public.kitware.com/Dart/HTML/Index.shtml>
- [19] <http://www.lysator.liu.se/xenofarm>