

A Large-Scale Fault-Tolerant Distributed Software-Build Process

Jim Buffenbarger
Boise State University
Hewlett-Packard Company
Boise, Idaho USA

A large software system can be compiled and linked more quickly if its build process is distributed across a network of multiple computers. However, large networks are more likely to contain a computer that causes a build to fail. If such a computer can be identified, it can be excluded from participation. Otherwise, if the failed command can be detected, it can be retried on a different computer.

We describe our experiences designing, implementing, and maintaining a fault-tolerant distributed build process for an industrial software-development environment. We focus on techniques that augment the capabilities of available distributed-build tools.

Our build process produces Hewlett-Packard's laser printer firmware. Our environment includes hundreds of engineers, about one thousand computers, and about two million lines of code. As an example of the speedup provided by distribution, a forced sequential rebuild of all targets requiring about 155 minutes can be accomplished concurrently in about 35 minutes.

1 Introduction

A software product's build process is the computation that transforms the product's source files into what is needed to use the product. A build process typically involves compilation and linking, but it may be much more complex. It may also require a large amount of time. One way to reduce the consumption of this precious resource is to distribute the computation across a network of multiple computers. Some build tools help automate distribution, but different environments have different requirements. One of our requirements is that a build process must not expect all computers to be reliable. In other words, a computer may temporarily or permanently be unable to successfully execute build commands. This paper describes what we have learned while designing, implementing, and maintaining a fault-tolerant distributed build process for Hewlett-Packard Company's (HP) LaserJet firmware. We hope our experiences will benefit build-tool developers and build-process engineers.

Our focus is on *using* distributed-build tools. While we briefly describe how they work, we are not trying to survey them, nor are we proposing new features. Our contribution is a description of the techniques we found to be effective in employing these tools in a large-scale industrial environment.

Our environment is the firmware-development lab for HP's LaserJet laser printers, in Boise, Idaho. Our software configuration management (SCM) system is Rational's ClearCase [8], and we use its build tool ClearMake. We have seven SCM servers and nearly a thousand client workstations. A server is essentially an NFS [13] fileserver, connected to one or more high-speed fibre-channel disk arrays, collectively containing about 200 Gbytes of version-controlled source code. A client performs builds. A client runs HP-UX 10.20, HP-UX 11.11, Red Hat Linux 8.0, or Red Hat Linux 9. An HP-UX client has one or more HP PA-RISC processors. A Linux client has one or more Intel Xeon processors. Servers are connected to network switches via 1 Gbit links. Clients are connected to switches via 100 Mbit links. Typically, our lab is developing firmware for two or three current (but not yet released) printers, and three to five future printers. A printer's firmware is a mix of approximately two million lines of C, C++, Java, IDL, and XML. Our lab interacts, via Rational's MultiSite [9], with about ten remote sites throughout the world, but they are not relevant to our topic. HP's Boise site has been in operation since 1973; the build process described here was deployed in 1999.

There are many aspects to managing a distributed-build environment. Some are conventionally mistreated as static characteristics of the computing infrastructure. A more dynamic perspective improves reliability and performance. Others are clearly part of the build process itself, yet require careful consideration.

On the infrastructure side, the overall set of computers, upon which builds may be distributed, must be partitioned into sets of functionally homogeneous computers. Each set requires special treatment, but within a set uniformity must be established, monitored, and maintained. The computational capability of each computer must be measured and monitored, as must its network-communication capabilities. Each computer's shared and private filesystems must be organized and mounted properly.

On the build-process side, build tools must be portable, or functionally equivalent tools must be provided for each set of homogeneous computers. The process must be divided into those parts that benefit from (and can tolerate) concurrency, and those parts that are best performed sequentially. Resource analysis, load sharing, and load balancing must be considered while selecting a computer for remote-command execution. Any interaction between various concurrent processes must be synchronized carefully. The process must be fault tolerant, with respect to failures from which it can recover. It must allow customization, with respect to its distribution characteristics. Finally, it must be self-auditing, providing performance and reliability statistics.

The rest of the paper is organized as follows: Section 2 describes distributed-build tools, Section 3 presents our techniques for augmenting their capabilities, and Section 4 offers observations and proposes future work.

2 Distributed-Build Tools

There are many, perhaps a hundred, implementations of Make [3]. Many of these support parallel builds (e.g., GNU Make [14]). Some of these support distribution to remote computers (e.g., Rational ClearMake [8]; Vesta [7]; GNU Make with Customs [1], PVM [10], or Winner [17]; PMake [2] with or without Customs; and Sun DMake [15]). There is also a distributing compiler [12]. However, from our perspective, distributed-build tools are quite similar. This section briefly discusses how a distributed build works, and relevant features of particular distributed-build tools.

Any build tool reads some kind of makefile, analyzes its targets and dependencies, and executes the fewest commands needed to update the targets.

A parallel-build tool can execute multiple commands concurrently, on the same computer that is executing the tool.

A distributed-build tool can execute multiple commands concurrently, on multiple computers. A configuration file specifies a set of homogeneous computers for remote execution. Tool developers seem to think this file is static, like a domain name server's configuration file. As we shall see, this misconception can be circumvented by "wrapping" a tool with a script that dynamically constructs a configuration file. Some tools start special daemons on the remote computers, for monitoring and communication; others use ordinary remote-execution services (e.g., `rsh`). A remote computer must execute a command in a carefully configured environment. Part of this environment is static (e.g., an appropriate version of an appropriate compiler). Tools assume that this part of the environment has been established. The other part of the remote-execution environment is dynamic (e.g., an environment-variable value, like `$PATH`). Tools establish this part of the environment, automatically.

3 Our Techniques

We extend the capabilities of our distributed-build tool in two ways: those that are external to our build process, and those that part of it.

3.1 External Techniques

Some of our external techniques are simply those prescribed by tool documentation. We use NFS to provide each computer with access to the SCM repository, which contains source code, compilers, and other resources. We use NFS, NIS, and an automounter [16] to provide each computer with access to the current view's storage directory ¹ and the current user's home directory. Home directories are convenient for build-process logging and customization. Remote-command execution is provided by `remsh` (for HP-UX), `rsh`, or `rexec`. NTP [11] provides time synchronization.

¹In ClearCase, a workspace is called a *view*. Each active view has a server process and storage directory.

Our other external techniques are more novel. Collectively, their goal is to provide hostnames for remote execution. With a thousand computers, we cannot possibly hope to track additions, deletions, and moves in a manual way. Furthermore, even if a computer is responsive (e.g., replies to `ping`), it may not be “healthy” (e.g., it may have lost NFS connectivity to the SCM repository).

Therefore, rather than trying to maintain static sets of homogeneous healthy hostnames, we employ a daemon, named “the crawler,” which executes on one of our servers. The crawler accepts, as input, a file containing the names of all of our client computers. This file is only updated when a new computer is installed or an old one is decommissioned. The crawler maintains, as output, a set of files, each of which contains names of homogeneous healthy computers.

While repeatedly iterating through its input file, the crawler performs the following examination of each client computer. First, the client must not have been manually excluded from participation (e.g., the script that halts a computer adds its hostname to the exclusion set). It must reply to several `ping` messages, without packet loss. It must accept and execute a remote command. Its SCM daemons must be active and the associated filesystems must be mounted and accessible. It must have sufficient communication bandwidth to the SCM servers, according to Netperf [6]. Its automounter daemon must be active. It must have sufficient computational power, according to its make and model.

A client computer that passes these tests is deemed an acceptable candidate for distributed-command execution. Its hostname is then added to the appropriate file, according to its: operating-system name, operating-system version, and SCM tool version.

While examining a client computer, the crawler may “get stuck” (e.g., become blocked, for a long time, on an NFS operation). Another daemon, named “the watcher,” detects this condition, kills the stuck crawler process, and starts a new crawler.

A file produced by the crawler is a list of hostnames, upon which, our build tool can execute remote commands. It can contain hundreds of lines. In order to improve the load balancing performed by our build tool, we also randomize the order of those lines. That way, our build tool is forced to balance even its consideration of a client for a command.

Since the crawler is constantly evaluating the condition of computers in our environment, its discoveries allow us to detect and fix problems quickly. When it finds an unhealthy computer, diagnostic information is logged, allowing us to investigate.

3.2 Internal Techniques

Our build process employs a variety of techniques to make distribution more effective.

We try to keep the build process as portable as possible. We use the same build tool, ClearMake, on all computers. We avoid esoteric build-tool features, which may vary from version to version. We prefer scripts (e.g., Bash [4] and Gawk [5]), rather than binaries. When necessary, we build binaries such that

they execute on as many versions of an operating system as possible (e.g., a binary compiled on HP-UX 10.20 can also run on HP-UX 11.11, but not the other way around). If a nonportable binary must be executed, its path is computed dynamically (e.g., via `uname`).

Some parts of our build process are performed locally and sequentially; others execute locally in parallel; still others are distributed to multiple remote computers. As a sequential example, a directory containing symbolic links to all interface files is constructed, before any compilation occurs. This prevents a compiler from having to search numerous directories for an interface. As a parallel example, multiple executable files are created locally in parallel, because linking object files across a network is too slow. As a distributed example, the compilation of each subsystem is performed on a remote computer. To compile a subsystem, the remote computer distributes individual compilation commands to other remote computers.

A developer executes the build process from a directory in a workspace. One execution of the process builds one variant of one product. For example, a developer might build an HP-UX 11.11 printer-simulator variant, for a printer named Raven. Another developer might build a MIPS production-firmware variant, for a printer named Kestrel. Some derived objects² can be shared between these two builds (e.g., a generated include file); others must not be shared (e.g., an object file, containing machine instructions). Furthermore, different sets of computers are capable of producing these files (e.g., only HP-UX 10.20 and 11.11 computers can produce code that runs on an HP-UX 11.11 computer). Therefore, our process must ensure that only appropriate derived-object sharing occurs and that build commands are distributed to appropriate computers.

A derived object can be shared, between builds, in two ways. It may already exist in the workspace, or it may be available in another workspace (i.e., a candidate for *winkin*). The first kind of sharing is protected by building a derived object in a directory named according to how it is built. The second kind of sharing is protected by storing the same information in the derived object's configuration record.

Our distributed-build process reads files produced by the crawler, which are described in Section 3.1. When the process is started on a particular computer, the combination of its operating-system name, operating-system version, and SCM-tool version selects a file containing a list of hostnames, upon which remote commands can be executed. Of course, access to this shared file must be synchronized. We use a simple copy-and-compare loop. There is also a way for a user to filter and/or augment the crawler's list.

Although the crawler tries to provide a list of names of healthy computers for build-command distribution, a command failure in an environment as large as ours may be difficult to predict. Such a problem is usually localized. It may be minor but sufficient, or serious but recent. In other words, the crawler's

²In ClearCase, a *derived object* is more than just a file resulting from a build. It has an associated *configuration record*, containing the build command and a list of each file and version that was accessed during the execution of that command. Configuration records allow workspaces to share a derived object safely, via a reference-copy operation called *winkin*.

evaluation of a computer is necessarily incomplete and, in any event, a computer's health can change quickly. For example, a computer may have an old `/usr/include/time.h` or its owner may unplug it during a compilation. Typically, a build tool simply examines each build command's exit code, and a non-zero code causes the build tool itself to exit. Our build process tries to detect unpredictable failures and react appropriately. It monitors an executing command's output, both `stdout` and `stderr`, as well as its exit code. A failed command is retried several times, on different computers, before the build process aborts. A failed command is retried using the latest crawler list, since the current one might be five or ten minutes old.

Our build process also tries to detect and retry a command that is "stuck," perhaps on a blocked NFS operation. The command's output is monitored for indicative messages and its execution is timed.

Retries are also logged, allowing us to investigate unhealthy workstations.

Unlike our other techniques, this retry feature would be a welcome addition to a general-purpose distributed-build tool.

4 Conclusion

Computers are faster than ever, but software systems are bigger than ever. Building a large software system, even just checking that derived objects are up to date, can be very time consuming. Distributing a build across multiple computers can significantly reduce the time a developer spends waiting in each edit/build/debug cycle, especially when module interfaces are changing.

We distribute builds across hundreds of computers, which would probably surprise distributed-build pioneers [1]:

I do not expect the Customs system to have to handle networks of more than two hundred workstations, internet addressing being what it is (the addresses are four bytes long and typically, at least at Berkeley, the first three bytes are used to specify a network, leaving a single byte to differentiate between machines.

Such distribution is worthwhile. For example, the time required for a forced rebuild dropped from about 155 minutes to about 35 minutes. Of course, this is a contrived example. More commonly, only a couple of files need to be rebuilt. Such a build requires only 6 or 7 minutes.

Our integration team performs between 400 and 800 distributed variant builds per day. They think in terms of product builds, where a product build is four variant builds, so they perform between 100 and 200 product builds per day. Without retries, about 5% of their product builds fail due to problems with the computing environment. Considering the complexity of our environment, we accept this degree of reliability. However, a failed build that is retried has at least a 95% chance of succeeding. If we're really unlucky, we try again. If we only retry the part of the build that failed, we don't waste time checking what

has already been built. Furthermore, a second failure is less likely, because less is being built.

It's like reading a book. If you started over at the beginning of the book every time you lost your place in a sentence, you'd never finish the book. When you lose your place in a sentence, you start over at the beginning of the sentence or, at worst, at the beginning of the paragraph.

Our distribution and retry mechanism has two main parts. Outside the build process, we perpetually maintain sets of healthy homogeneous hostnames, on which to execute remote build commands. Within the build process, we monitor the execution of module-level build commands, detecting failures that we expect not to recur when the command is retried. Before a failed command is retried, in fact, before every module-level build command is executed, the latest appropriate set of healthy hostnames is obtained.

In the future, we plan to use the result of failure detection to improve failure prevention. Specifically, when a build has detected a command failure on a particular computer, its hostname will be removed from the set of healthy hostnames used to retry the command. Also, the crawler will be notified of the hostname, so other builds may be spared similar failures.

5 Acknowledgments

The author wishes to thank the other members of his SCM team, within HP's Core Technology Lab: Kirk Gruell, Doug Dunlap, Gerry Weber, Bruce Rust, Tori Lewis, and Gary Ackaret.

References

- [1] A. de Boor. *Customs — A Load Balancing System*, 1987.
<ftp://ftp.icsi.berkeley.edu/pub/ai/stolcke/software/>.
- [2] A. de Boor. *PMake — A Tutorial*, 1994.
<ftp://ftp.icsi.berkeley.edu/pub/ai/stolcke/software/>.
- [3] S. Feldman. Make: A program for maintaining computer programs. *Software — Practice and Experience*, 9(4), April 1979.
- [4] Free Software Foundation, Inc. *The GNU Bash Reference Manual*, 2002.
<http://www.gnu.org/software/bash/manual/bash.html>.
- [5] Free Software Foundation, Inc. *GAWK: Effective AWK Programming: A User's Guide for GNU Awk*, 2003.
<http://www.gnu.org/software/gawk/manual/gawk.html>.
- [6] Hewlett-Packard Company. *Netperf: A Network Performance Benchmark*, 1995. <http://www.netperf.org/>.

- [7] A. Heydon, R. Levin, T. Mann, and Y. Yu. The Vesta Approach to Software Configuration Management. Technical Report 168, Compaq Systems Research Center, 2001. <http://www.vestasys.org/>.
- [8] International Business Machines Corporation. *Rational ClearCase*, 2004. <http://www.ibm.com/products/us/>.
- [9] International Business Machines Corporation. *Rational ClearCase MultiSite*, 2004. <http://www.ibm.com/products/us/>.
- [10] A. Lih and E. Zadok. PGMAKE: A Portable Distributed Make System. Technical Report CUCS-035-94, Computer Science Department, Columbia University, 1994. <http://www1.cs.columbia.edu/~ezk/research/pgmake/>.
- [11] D. Mills. *The Network Time Protocol (NTP) Distribution*, 2004. <http://www.ntp.org/>.
- [12] M. Pool. *distcc — A Distributed C/C++ Compiler*, 2003. <http://distcc.samba.org/>.
- [13] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *1985 Summer USENIX Conference*, pages 119–130. USENIX, 1985.
- [14] R. Stallman and R. McGrath. *GNU Make: A Program for Directing Recompilation*, 2002. <http://www.gnu.org/software/make/manual/make.html>.
- [15] Sun Microsystems, Inc. *Sun WorkShop TeamWare User's Guide*, 1999. <http://docs.sun.com/>.
- [16] The NetBSD Foundation, Inc. *NetBSD Documentation*, 2004. <http://www.netbsd.org/Documentation/>.
- [17] F. Thilo. *wmake — Parallel and Distributed Make*, 1998. <http://www.informatik.uni-siegen.de/softdocs/winner/>.