distcc, a fast free distributed compiler

Date: December 2003
Author: Martin Pool <mbp@sourcefrog.net>
Presented at: linux.conf.au, Adelaide, 2004
Licence: Verbatim reproduction of this paper is permitted.

1 Abstract

distcc is a program to distribute compilation of C, C++, Objective C and Objective C++ code between several machines on a network. distcc is easy to set up, and can build software projects several times faster than local compilation. It is distributed under the GNU General Public License.

distcc does not require all machines to share a filesystem, have synchronised clocks, or to have the same libraries or header files installed. They can even have different processors or operating systems, if cross-compilers are installed.

This paper begins with some background information on compilation and prior distributed compilers, then describes the design of distcc, presents the outcomes and some applications, and some lessons learned that may be useful to other small free software projects.

2 Background

It is common for a software developer to have access to more than one computer. Perhaps they have a laptop and also a desktop machine, or perhaps they work in a group where each member has a workstation. It would be nice if the spare computing power could be used to speed up software builds.

distcc offers a straightforward solution. The `distccd` server runs on all the machines, with a low priority and a cap on the number of processes. Any client can distribute compilation jobs onto other machines. distcc can be installed by any user, without needing root access or system-wide changes.

Faster compilations allows "clobber" builds (`make clean; make all`) to be run more frequently, which can catch bugs that might be hidden by incorrect Makefile dependencies. It allows test engineers to get builds to test faster. It makes nightly, or hourly check builds more feasible, so bugs that are checked in can be caught and fixed with less disruption. In general it removes one limitation from the developer's work flow and allows them to work the way they want.

The ability to build quickly is particularly useful when building binary software packages with `dpkg` or `rpm`. These tools often want to build the whole tree from scratch so as to ensure a repeatable build. Using distcc and ccache makes repeated package builds faster.

Fast distributed builds are also useful for operating systems distributed primarily in source form, such as Gentoo and Linux From Scratch.

2.1 Previous work

There have been several prior distributed build systems, including pvmmake, doozer, and dmake. These systems typically require the build directory to be on a networked filesystem shared between all machines participating in the build. Furthermore, each machines must have the same header files and libraries installed. If one of the header files is inconsistent, it can, at best, produce confusing error messages, and at worst, it can cause unpredictable crashes when different modules see different definitions. All of the machine clocks must be tightly synchronized so that the timestamps are accurate.

This can be reasonable for a dedicated build cluster, but is less practical in ad-hoc networks, where users may be able to upgrade libraries on their machines at random, or where using NFS may not be desirable.

Another approach is to use kernel-level clustering such as OpenMOSIX. This allows any parallel tasks to be distributed, but introduces very tight coupling between machines. Systems based on process migration may not perform well with the many short-lived processes generated during compilation.

2.2 C compilation

It's worthwhile to briefly discuss how C programs are built. Typically, a build tool (such as *Make*) invokes the C compiler for each source file to produce an object file. Within this compilation, there are several steps.

- The source is run through the `cpp` preprocessor, which folds in header files, handles `#ifdef` and other directives, and produces a single preprocessed output file.
- The compiler proper cc1 reads the preprocessed source and produces assembly output.
- The assembler as reads the assembly text and produces an object file.

One interesting property of the C build process is that the preprocessed text plus the compiler command line completely specifies the output. By running the preprocessor locally, distcc avoids any requirement to have the same headers on all machines.

Before distcc, the compilercache and ccache programs used this observation to remember the results of building a particular source file. This can be useful when trees are cleaned and rebuilt, or when similar trees are built in different directories.

Using GNU make's `-j` option several jobs are run in parallel. Make uses its graph of dependencies between targets to make sure that only independent jobs are run in parallel – or at least it does if the Makefile correctly specifies dependencies.

Although sending a file across a network takes a relatively long time, it takes few CPU cycles. On Linux and BSD, distcc uses the `sendfile` system call so that transmission of the preprocessed source from the client is done in a single system call.

3 Design

distcc distributes work from a client machine to any number of volunteer machines. (Since this is free software, we prefer the term *volunteer* to the *slaves* that are used by other systems.)

distcc consists of a client program and a server. The client analyzes the command run, and for jobs that can be distributed it chooses a host, runs the preprocessor, sends the request across the network and reports the results. The server accepts and handles requests containing command lines and source code and responds with object code or error messages.

3.1 Simulating gcc

The client is invoked as a wrapper around the compiler by Make. Because distcc is invoked in place of gcc, it needs to understand every pattern of command line invocations. If the arguments are such that the compilation can be run remotely, distcc forms two new sets of arguments, one to run the preprocessor locally, and one to run the. compiler remotely. If the arguments are not understood by distcc, it takes the safe default of running the command locally. Options that read or write additional local files such assembly listings or profiler tables are run locally

For commands that need to be run locally, distcc runs the compiler locally, avoiding the overhead of the network protocol and of separately running the compiler. Early versions of distcc directly replaced their process with the local compiler using `exec`, but in current releases the compiler is run as a child process so that its progress can be monitored. Some cases are difficult because the precise meaning of some options such as `-MD` has changed between versions of gcc.

The client supports a "masquerade" mode, where a directory such as `/usr/lib/distcc` is filled with symbolic links which have the same name as the real compiler, but point to the distcc client. This directory is prepended to `$PATH`, so that when Make executes `gcc`, it really invokes distcc. distcc specially handles this case to work out the correct real compiler to invoke, and guards against inadvertently running itself recursively. Masquerade mode allows distcc to be used with most existing source trees without the need to change the Makefile or build process.

3.2 Networking

The server runs on each of the volunteer machines, either as a standalone daemon, through `inetd` or indirectly through `sshd`. distcc normally runs on TCP port 3632.

distcc uses a simple custom binary network protocol to encode requests and responses. Variable-length fields are preceded by a length word, allowing a very simple and efficient implementation.

The request includes a protocol version, the compiler command-line arguments, and the preprocessed source. The response from the server includes confirmation of the version, the compiler's exit code, any errors or warnings from the compiler, and the object file if any.

The distcc protocol includes four-letter magic words before each element. These words are not strictly necessary because the order of elements is always the same, but they make it easier to scan network dumps, and provide a protection against networking or program errors.

distcc relies on TCP or SSH to ensure integrity of the stream and does not have a checksum of its own. This is the same level of protection as HTTP or NFS, and no failures have been reported to date.

3.3 Scheduler

distcc uses a basic load-balancing algorithm to choose a volunteer to run each particular job. Scheduling is managed by small state files kept in each user's home directory. In addition, each server imposes a limit on the number of jobs which will be accepted at any time. This prevents any single server being overloaded, and provides basic coordination over servers being used by

several clients. By default, the limit on the number of jobs is set to two greater than the number of processors in the server, to allow for up to two jobs being in transit across the network at any time.

The server uses a *pre-forking* design similar to Apache: the maximum number of client processes are forked at startup, and all of them wait to accept connections from a client. This puts a limit on concurrency, and slightly reduces the latency of accepting a new connection.

Clients keep a temporary note of machines which are unreachable or failing. These notes time out after sixty seconds, so machines which reboot or are temporarily unreachable will shortly rejoin the cluster. Machines which are abruptly terminated or shut down while running jobs may cause those particular compilations to fail, which can normally be addressed by re-running Make. If `distccd` is gracefully terminated it will complete any running jobs and decline future submissions.

Client processes are coordinated by Unix lock files held in `~/.distcc`. `flocks` have the strong advantage that they are always released by the kernel when the process terminates, even if it aborts because of a signal.

As with a kernel scheduler, distcc does not have enough information about future jobs to make perfect scheduling decisions. In particular, make does not communicate the global state of the build and how many jobs will be issued in the future. All distcc knows is the number of jobs are running at the present moment. If it knew about future load, it might schedule jobs in the middle of a build differently from the last few jobs. Jobs on the critical path of the build and, in particular, the last jobs of a build ought to be scheduled for minimum delay rather than maximum throughput, which might mean running them locally rather than remotely.

At the moment the client does not maintain an estimate of the speed of volunteer machines. Estimating this accurately is difficult because the performance depends on the source and computer in a complex way. Different files of similar size can take very different times to compile depending on the difficulty of the code.

The primary shortcoming of the current scheduler is that adding slow machines to a build farm can reduce the overall performance. There is a significant opportunity cost to waiting for a job that on such a slow machine, particularly if it is a dependency for Make to proceed.

3.4 SSH

distcc has the option of using a helper program such as `ssh` to open connections rather than simply opening a TCP socket.

When OpenSSH is used to open connections, all data is strongly encrypted. The volunteer machines do not need any additional listening ports or long-running processes. All remote compilations run under the account of the user that initiated them.

Using OpenSSH or another pluggable connection method is a powerful Unix design pattern. Just a few hundred lines in distcc to fork the connection program and open pipes allows it to use world-class encryption and authentication that easily integrates into existing networks.

A key advantage of this pattern is that distcc's security exposure is very small. Any connection that reaches distcc is by definition authorized either by ssh opening the socket, or because the administrator allowed connections from that address. Most of the distcc code can therefore be quite trusting of the requests that are received.

If someone wanted to implement a simpler transport that did challenge-response password authentication but not encryption, then they could plug it in through the `DISTCC_SSH` variable without needing any changes to distcc itself.

3.5 LZO compression

distcc can use the LZO fast compressor to compress requests and replies. This option is explicitly set by the client in the host list.

The preprocessed source is typically very gassy because of long runs of whitespace and repeated identifiers, and compresses well. Object files also compress moderately well.

Compression improves throughput when the network is the limiting factor. This can be the case when the network is very slow, such as a 10Mbps wired network or a wireless network. Compression can also be useful when there are many remote machines, which can saturate even a fast Ethernet connection.

4 Outcomes and applications

4.1 A performance model

The goal of distcc is to reduce the overall elapsed time to build or refresh a complete tree.

Performance is predicted by the network bandwidth, the time to run the preprocessor and the compiler, the relative speeds of the client and volunteer computers, and the overhead of the distcc client and server. The speed of compilation on a particular machine depends on available memory, contention for the CPU, and the speed of the CPU, RAM, and bus.

In many cases other aspects of the build process become the limiting factor once compilation is parallelized. For small packages, running an `autoconf` script is much slower than actually compiling the package. `Make` can also be a limiting factor, because it has high overheads and cannot schedule efficiently across source directories. One user found that distcc alone doubled build speed, but switching to the SCons construction tool in combination with distcc gave a five-fold improvement.

4.2 Estimate of feasibility

Before any code was written for distcc, the author did a brief feasibility study to see if the idea was practical. This involved measuring the elapsed and CPU time for some of the basic operations: running the preprocessor and the compiler together and separately, copying a file across the network, and starting a simple wrapper program. The compiler used 90% or more of the CPU time in most trials. This indicated that the bulk of compilation could in theory be run remotely, and distcc might win when more than one machine was available.

Feasibility studies such as this are heartily recommended before starting a project, especially one where success depends on achieving a quantifiable result. Knowing that a concept ought to work can be a worthwhile spur to writing a good implementation that achieves the predicted result. Conversely, knowing that an idea is impractical can avoid sinking time or money into a project that can never win. (It might still be worth documenting your results in a web page or email, in case the situation changes in the future or somebody else can think of an improvement.)

4.3 Is it safe?

In eighteen months of widespread use of distcc to build diverse source trees, there have been no reported unexplained failures. Bug reports are due to, in order of frequency:

1. Mismatched compiler versions or broken compiler installations.
2. Concurrency bugs in a makefile or other build process.

3. Known bugs in gcc or other software that are provoked by distcc.

One persistent problem has occurred because of a quirk in the way gcc produces debug information. The compiler's working directory is inserted into the debug stabs to help in finding the source for the program. This is a problem for distcc and ccache because the compiler is invoked in a different directory from the preprocessor. This should be fixed in an upcoming gcc release by including the compilation directory in the preprocessor output.

4.4 Scalability

Reports from users indicate, distcc is nearly linearly scalable for small numbers of CPUs. Compiling across three identical machines is typically 2.5 to 2.8 times faster than local compilation. Builds across sixteen machines have been reported at over ten times faster than a local builds. These numbers include the overhead of distcc and Make, and the time for non-parallel or non-distributed tasks.

Amdahl's law states that the possible speedup from parallelizing a task is limited by the reciprocal of the fraction of a task which must be serialized. distcc is able to parallelize the compilation phase, but execution of the makefile, preprocessing, linking, and the distcc client must all be local. The balance of these tasks for a particular source tree determines how much it will benefit from using distcc.

Performance tends to plateau between ten and twenty machines. This is consistent with measurements of the preprocessor using roughly 5-10% of the total CPU time: when twenty preprocessors and distcc clients are running, the client is completely saturated and cannot issue any more jobs.

Lack of access to a large test lab has so far prevented a systematic exploration of the limits of scalability.

4.5 Compiler versions

Incompatibilities between different releases of gcc proved to be a more serious problem than was expected.

It turns out that it is not even sufficient to check the gcc version string. Across different Linux distributions, two gcc binaries claiming to be the same version can have incompatible APIs, because of distributors merging or backporting different patches. gcc is not necessarily compatible across different BSD forks.

In general C++ causes more trouble than C because the ABI is more complex and has been more unstable.

Incompatibilities typically manifest either as errors reading system headers which depend on gcc internals, or as link errors. In the absence of a satisfactory way to check compatibility, for the moment we just advise people to check compiler versions if an error occurs. Some automatic checking may be added in a future version.

4.6 Makefile bugs

Because distcc can use CPUs on more than one machine, it is common to use much higher `-j` levels than for local builds, with ten or even twenty jobs in flight at once. This will sometimes uncover dormant concurrency bugs in Makefiles.

For example, a source file may depend on an automatically generated header, but that dependency may not actually be specified in the Makefile. For a non-parallel build, Make might happen to run the rules in such a way that the header is always generated first. When many

concurrent tasks are run, the compile may start earlier and then fail because the header is not yet ready. As with threading bugs, these failures can be intermittent and hard to track to a root cause. Make's `--debug` will eventually explain the problem after enough scrutiny.

Some large packages such as XFree86 have stubborn Makefile bugs and must be built without parallelism. Concurrency bugs have also been discovered in tools such as `libtool` that are called from makefiles. Another source tree used a single temporary file name for all object files, so that they overwrote each other when run in parallel.

4.7 Lifting compilation

distcc can be useful even without parallel builds. It is fairly common for a developer to be working on a relatively slower machine: perhaps on a laptop, or an embedded system, or perhaps a large server machine is available. It is quite possible for the benefit in running on a faster CPU to be greater than the overhead of running distcc and transferring the build across the network.

Being able to do most of the work of compilation remotely makes it feasible to do development entirely on embedded platforms that are large enough to run an editor and linker, such as the Sharp Zaurus.

distcc support sociable computing by making it possible to do development on quite large projects from a laptop, if compilation is lifted off to one or more larger and noisier servers in another room.

4.8 Security

distcc is intended to be quite secure when used according to the documentation.

Remote execution of compile jobs introduces a trust relationship between the client and server machines. The client completely trusts the server to compile code correctly. A corrupt server could introduce malicious code into the results, or attack the client in other ways. The server completely trusts an authorized client. A malicious client could execute arbitrary commands on the server.

The distccd TCP server does not run as root. If started by root, before accepting connections it gives away its privileges and changes to either the *distcc* or *nobody* user, or to a user specified by the `--user` option.

The distcc client may be run as root. Compiling or doing other unnecessary work as root is generally discouraged on Unix, but distcc permits it because some build processes require it.

In TCP mode network transmissions are not encrypted or signed. An attacker with passive access to the network traffic can see the source or object code. An attacker with active access can modify the source or object code, or execute arbitrary commands on the server.

The TCP server can limit clients by either checking the client address, or listening only on particular IP interfaces. However, in some circumstances an attacker can spoof connections from a false address. On untrusted networks SSH is the only safe choice.

In SSH mode the server is started by each user under their own account on the server. There is no long-lived daemon. Connections are authenticated and encrypted. Every user is allowed only the privileges that their normal account allows.

Because the server and client necessarily trust each other, there has been no security audit of the code that runs after a connection is established. It is possible that a hostile server could gain control of a client directly, as well as modifying the object code.

Some people have proposed that the server should perform reasonableness checks on the command or source code submitted from clients. This is probably futile because gcc is not secure against hostile input, and it might be possible for an attacker to gain control of the gcc

process through a carefully crafted input file or command line. The only safe assumption is that anyone able to submit jobs is able to execute arbitrary operations under the account used by distccd.

Alternatively it has been suggested that client should check whether a server is trustworthy by re-running compilations on a different server. This approach can be useful in some distributed-computing problem spaces such as SETI@home, but is not practical for distcc. Firstly, the damage from a single compromised object file is very high, so every single file would have to be extensively checked. Secondly, the performance constraints are such that building every single file two or more times would often make distribution worthless. Thirdly there would sometimes be false alerts from minor differences in gcc patchlevel that are not visible in normal use but that do change the output file at a byte level.

Only one security issue has arisen to date, which was that distcc used a predictable temporary directory name. A local attacker by creating that directory could cause distcc to fail, but no more damage.

4.9 distcc press-gangs

Systems have been developed which allow non-Unix machines to be co-opted to contribute to compilation.

One such is distccKNOPPIX, which allows a machine to be rebooted and to work as a compiler volunteer without needing any installation onto its hard disk. The compiler and distcc server are loaded across http, so that they can easily be matched to the versions on the client.

distcc packages are now available for Cygwin, allowing cross compilers to be run on Microsoft systems.

4.10 Good logging

Most releases of distcc has discovered bugs in either distcc or other software that could not be reproduced on the author's machine. It has the honour of helping find a serious bug in the Linux 2.5 TCP stack by being the first program to flood the network fast enough to provoke a timer wraparound.

At the default logging level, distcc emits only warnings and errors, such as when a compile host is unreachable or compilation fails.

When verbose logging is enabled, distcc reports the success or failure of most non-trivial system calls and internal functions. Failure can indicate either a problem on the machine, or a bug in distcc. In the first case, the logs can help the user find out where the problem is. If there was a bug in distcc, the trace helps work out where exactly the failure occurred, what actions led up to that point, and what the consequences were. Many of these bugs are at heart incorrect assumptions about, for example, how distcc will be called or how a system call behaves on different platforms. The trace often calls out which assumptions were incorrect.

The goal of distcc's error handling is to have any problem cause at most one failure. It ought to be possible to look at a verbose trace and work out what went wrong and the fix.

Errors and trace messages from the server are written both into a log file, and also into the compiler error stream passed back to the client. This makes it more likely that if a problem occurs on the server, it will be obvious to the remote user without their having to look in the server's log file. This pattern is very useful, though it might not be appropriate for software that handles requests from untrusted users. Errors during server startup are written to error output as well as to the log file, to make it more obvious to the administrator that the server did not actually start.

4.11 Generalisations

There has been some interest in adapting distcc to compile other languages such as Java or C#. These languages have quite different mappings from source files to output, and in particular there is no separate preprocessing phase. A Java compiler must read class files as directed by `import` statements, and those classes in turn may need to be rebuilt from source. A distributed compiler for these languages would be possible but might need to be more tightly integrated with the compiler than distcc is.

distcc could be generalised still further to distribute arbitrary non-interactive jobs, rather than just compilation. A good step towards this would be to move all the intelligence about command lines into the client, so that the server just managed temporary files and executed specified commands. To date no compelling application for this sort of distribution has presented. Scientific and technical computing users seem reasonably well served by clustering and batch-queueing software.

4.12 Changes in gcc

gcc version 3.3 includes some performance improvements that make distcc less immediately useful.

One is precompiled headers, whereby the internal state of the compiler is dumped out to disk after reading a set of common header files. This is useful for programs that include a large and fixed set of headers, where parsing the headers takes a large fraction of the compile time. At the moment distcc can support this by putting the `.pch` state file on a shared filesystem. A better solution may be added to distcc in the future.

gcc has also recently gained an integrated preprocessor option, to avoid the overhead of running the preprocessor as a separate task. Since distcc must run the preprocessor separately, it cannot benefit from the integrated preprocessor and it's performance compared to gcc is slightly worse.

4.13 Worse is better

distcc adheres to the "Worse is Better" philosophy, summarized by Richard Gabriel as four points:

> - *Simplicity: the design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design.*

distcc does not require changes to the kernel, the build system, or the compiler. It can be described in a couple of pages, and installed in a matter of minutes. There is no central coordinating process or global knowledge of the cluster's state.

The most prominent place where the implementation is less simple than necesssary is in network IO, which is moderately optimized. This is perhaps justifiable as network speed is important to the program being useful.

Masquerading as gcc makes the implementation more complex, because distcc must identify that it is masqueraded, and then make appropriate adjustments to the path to invoke the real compiler. However, masquerading makes the installation and use of distcc much easier, and so it is a reasonable tradeoff. If it had turned out to be hard to find a reliable and reasonably simple masquerading algorithm, the feature might have been dropped.

distcc is written in C, which makes a correct implementation slightly harder and more complex than it would be in a scripting language. This is justified because to perform adequately, distcc

absolutely must have a very small startup cost, which is not the case for any popular very-high-level language.

> - *Correctness: the design must be correct in all observable aspects. It is slightly better to be simple than correct.*

There are no known cases where distcc generates an incorrect and unexplained result. The edges cases such as debug information are being fixed in a simple manner, by correcting gcc, rather than by complicating distcc.

Almost all free software trees will build, with the restriction that trees with concurrency bugs cannot be built in parallel.

When distcc encounters an unexpected condition, it tries to "fail safe" by, for example, running the compilation locally, backing off from a particular machine and avoiding accidental recursive invocations.

> - *Consistency: the design must not be overly inconsistent. Consistency can be sacrificed for simplicity in some cases, but it is better to drop those parts of the design that deal with less common circumstances than to introduce either implementational complexity or inconsistency.*

Almost all the code in distcc is common across platforms, and most of the code is reached on any particular execution: there are relatively few special cases.

There are some special cases and inconsistencies in the way the daemon starts up, in an attempt to make it more friendly to administrators and as historical accidents.

distcc is very consistent with modern free Unix practice in details such as responding to `--help`, using the GNU autoconf system for installation, and installing into standard directories.

> - *Completeness: the design must cover as many important situations as is practical. All reasonably expected cases should be covered. Completeness can be sacrificed in favor of any other quality. In fact, completeness must sacrificed whenever implementation simplicity is jeopardized. Consistency can be sacrificed to achieve completeness if simplicity is retained; especially worthless is consistency of interface.*

distcc is intentionally not a general-purpose clustering system. They are too hard to write, and none that can be installed in a matter of minutes. distcc has very incomplete coverage of all the tasks people might want to distribute across machines, but it does handle a large task that is important to a significant audience.

Within the space of C compilation, most tasks can be distributed. Even if a few jobs during a software build cannot be distributed, others run in parallel might be, so there is still an overall benefit. distcc's documentation and diagnostic output make it reasonably easy to identify problem commands.

Some gcc options that could in practice be distributed are run locally for the sake of implementation simplicity. Commands that produce assembly listings are one example: there is no reason why the assembly output could not be relocated back to the client, but this option is rarely used so the complexity is not justified.

5 Conclusions

distcc has a simple design and gains large performance improvements for software builds. It is reliable and straightforward to install and manage.

A few maxims about demonstration of software projects have been borne out by distcc's development. Before development began, a feasibility study demonstrated that the concept could work. The design is intentionally simple and the scope modestly focussed on a subset of important problems. Strong logging helps get useful results from distributed community testing. Performance features such as compression and new network APIs were used selectively in places where they are shown to increase distcc's speed.

As the world's body of source code continues to grow, distcc will help get it built, tested and released more quickly and painlessly.