distcc User Manual

Martin Pool mbp@samba.org $Revision: 1.91 $ $Date: 2002/12/12 09:17:25 $, for distcc-0.14

Contents

Chapter 1

Introduction

"Speed, it seems to me, provides the one genuinely modern pleasure." — Aldous Huxley (1894 - 1963)

1.1 Overview

distcc <http://distcc.samba.org/> is a program to distribute compilation of C or C++ code across several machines on a network. distcc should always generate the same results as a local compile, is simple to install and use, and is often significantly faster than a local compile.

Unlike other distributed build systems, distcc does not require all machines to share a filesystem, have synchronized clocks, or to have the same libraries or header files installed.

Compilation is centrally controlled by a client machine, which is typically the developer's workstation or laptop. The distcc client runs on this machine, as does *make*, the preprocessor, the linker, and other stages of the build process. Any number of "volunteer" machines help the client to build the program, by running the compiler and assembler as required. The volunteer machines run the `distccd` daemon which listens on a network socket for requests.

distcc sends the complete preprocessed source code across the network for each job, so all it requires of the volunteer machines is that they be running the `distccd` daemon, and that they have an appropriate compiler installed.

distcc is designed to be used with GNU make's parallel-build feature (`-j`). Shipping files across the network takes time, but few cycles on the client machine. Any files that can be built remotely are essentially "for free" in terms of client CPU.

1.2 Author

distcc was written by Martin Pool.

distcc was inspired by Andrew Tridgell's ccache program.

1.3 Licence

The author understands the GNU GPL to apply to distcc in the following way: you are allowed to use distcc to compile a non-free program, or to call it from a non-free Make, or to call a non-free compiler. However, you may not distribute a modified version of distcc unless you comply with the terms of the GPL: in particular, giving your users access to the source code and the right to redistribute it, and clearly identifying your changes.

If you find distcc useful, I would appreciate you writing an email to tell me.

1.4 Security Considerations

distcc should only be used on networks where all machines and all users are trusted.

The distcc daemon, `distccd`, allows other machines on the network to run arbitrary commands on the volunteer machine. Anyone that can make a connection to the volunteer machine can run essentially any command as the user running `distccd`.

distcc is suitable for use on a small to medium network of friendly developers. It's certainly not suitable for use on a machine connected to the Internet or a large (e.g. university campus) network without firewalling in place.

`inetd` or `tcpwrappers` can be used to impose access control rules, but this should be done with an eye to the possibility of address spoofing.

In summary, the security level is similar to that of old-style network protocols like X11-over-TCP, NFS or RSH.

1.5 Getting Started

Four straightforward steps are required to install and use distcc:

1. Compile and install the `distcc` package on the client and volunteer machines.
2. Start the `distccd` daemon on all volunteer machines.
3. On the client, set the `DISTCC_HOSTS` environment variable to indicate which volunteer machines to use. For example:

```
DISTCC_HOSTS='angry toey:4202 localhost'
```

4. Set the `CC` variable or edit Makefiles to prefix distcc to calls to the C/C++ compiler. For example:

```
distcc gcc -o hello.o -c hello.c
```

1.6 Reporting Bugs

If you think you have found a bug, please check the manual and the `HACKING` file to see if it is a known restriction. If not, please send a clear and detailed report to Martin Pool `mbp@samba.org`. (For a clear and detailed description of "clear and detailed", see Simon Tatham's advice on reporting bugs, <http://www.chiark.greenend.org.uk/~sgtatham/bugs.html> .)

A good bug report for distcc should include:

1. What you're trying to do. For example: "compile KDE", "use gcc's `-MD` option".
2. What actually happens. For example: "distcc fails with error 104", "the compilation never completes", "I get error message XXX".
3. The version of distcc you're using (the output of `--version` on both client and server. If you got it from a distribution rather than building it yourself, then mention that.
4. What other software you're using, in particular the operating system and compiler. For the operating system it's normally enough to give the overall version (FreeBSD CURRENT, RedHat Linux 7.2, ...). For the compiler, use "`gcc -version`".
5. The exact command you're using to run the compilation. If you're using make, then include the line from its output that runs the compiler.
6. The debug logs from the client and server. On the client, you should set `DISTCC_VERBOSE` and `DISTCC_LOG`. On the server, use `--verbose` and `--log-file`. If you can, trim the log files to just the invocation that causes trouble. Grepping for a process id can help with this. If the problem is intermittent, then please leave logging running until it recurs and then pull out a smaller section of logs to send.

Please do not obfuscate your logs. The name of a single source file or machine is probably not confidential information, but the confusion introduced by editing logs can be significant.

Please send a problem description to the `distcc` mailing list, on `lists.samba.org`. Please don't send mail direct to the author: if you use the list, other people may be able to help you, and the answers are publicly archived.

1.7 Test Suite

distcc has a test suite written in Python using the *ComfyChair* framework. It does not yet exercise all functionality, but is improving.

To run the test suite, run `make check` from the distcc source directoy.

Chapter 2

Using distcc

2.1 Invoking distcc

distcc is prefixed to compiler command lines and acts as a wrapper to invoke the compiler either on the local client machine, or on a remote volunteer host.

For example, to compile the standard application program:

```
distcc gcc -o hello.o -c hello.c
```

Standard Makefiles, including those using the GNU autoconf/automake system use the **$CC** variable as the name of the compiler to run. In most cases, it is sufficient to just override this variable, either from the command line, or perhaps from your login script if you wish to use distcc for all compilation. For example:

```
make CC='distcc'
```

2.2 Options

Options to distcc must precede the compiler name. Any arguments or options following the name of the compiler are passed through to the compiler.

`--help`

Print a detailed usage message and exit.

`--version`

Show distcc version and exit.

2.3 Environment Variables

The way in which distcc runs the compiler is controlled by a few environment variables.

NOTE:

Some versions of make do not export Make variables as environment variables by default. Also, assignments to variables within the Makefile may override their definitions in the environment that

calls make. The most reliable method seems to be to set `DISTCC_*` variables in the environment of Make, and to set `CC` on the right-hand-side of the Make command line. For example:

```
$ DISTCC_HOSTS='localhost wistful toey'
$ export DISTCC_HOSTS
$ CC='distcc' ./configure
$ make CC='distcc' all
```

Some Makefiles may, contrary to convention, explicitly call `gcc` or some other compiler, in which case overriding `$CC` will not be enough to call distcc. This should be harmless, however: those jobs will just run locally. The best solution is to update the Makefile to compile and link using `$(CC)` to promote future maintainability.

`DISTCC_HOSTS`

Space-separated list of volunteer host specifications.

`DISTCC_VERBOSE`

If set to `1`, distcc produces explanatory messages on the standard error stream. This can be helpful in debugging problems. Bug reports should include verbose output.

`DISTCC_LOG`

Log file to receive messages from distcc itself, rather than stderr.

`DISTCC_SAVE_TEMPS`

If set to `1`, temporary files are not deleted after use. Good for debugging, or if your disks are too empty.

`DISTCC_TCP_CORK`

If set to `0`, disable use of "TCP corks", even if they're present on this system. Using corks normally helps pack requests into fewer packets and aids performance.

2.4 Which Jobs are Distributed?

Building a C or C++ program on Unix involves several phases:

- Preprocessing source (`.c`) and headers (`.h`) to a preprocessed file (`.i`)
- Compiling preprocessed source (`.i`) to assembly instructions (`.s`)
- Assembling to an object file (`.o`)
- Linking object files and libraries to form an executable, library, or shared library.

distcc only ever runs the compiler and assembler remotely. The preprocessor must always run locally because it needs to access various header files on the local machine which may not be present, or may not be the same, on the volunteer. The linker similarly needs to examine libraries and object files, and so must run locally.

The compiler and assembler take only a single input file, the preprocessed source, produce a single output, the object file. distcc ships these two files across the network and can therefore run the compiler/assembler remotely.

Fortunately, for most programs running the preprocessor is relatively cheap, and the linker is called relatively infrequent, so most of the work can be distributed.

distcc examines its command line to determine which of these phases are being invoked, and whether the job can be distributed. Here is an example of a typical command that can be preprocessed locally and compiled remotely:

```
distcc gcc -o hello.o -DGREETING="hello" -c hello.c
```

The command-line scanner is intended to behave in the same way as gcc. In case of doubt, distcc runs the job locally.

In particular, this means that commands that compile and link in one go cannot be distributed. These are quite rare in realistic projects. Here is one example of a command that could not be distributed, because it calls the compiler and linker

```
distcc gcc -o hello hello.c
```

2.5 Running Jobs in Parallel

Moving source across the network is less efficient to compiling it locally. If you have access to a machine much faster than your workstation, the performance gain may overwhelm the cost of transferring the source code and it may be quicker to ship all your source across the network to compile it there.

In general, it is even better to compile on two or machines in parallel. Any number of invocations of distcc can run at the same time, and they will distribute their work across the available hosts.

distcc does not manage parallelization, but relies on Make or some other build system to invoke compiles in parallel.

With GNU Make, you should use the `-j` option to specify a number of parallel tasks slightly higher than the number of available hosts. For example:

```
$ export DISTCC_HOSTS='angry toey wistful localhost'
$ make -j5
```

2.6 Choosing a Host

The `$DISTCC_HOSTS` variable tells distcc which volunteer machines are available to run jobs. This is a space-separated list of host specifications, each of which has the syntax:

```
HOSTNAME[:PORT]
```

A numeric TCP port may optionally be specified after a colon. If no port is specified, it uses the default, which is currently 4200.

If only one invocation of distcc runs at a time, it will always execute on the first host in the list. (This behaviour is not absolutely guaranteed, however, and may change in future versions.)

The name `localhost` is handled specially by running the compiler in place.

The daemon may be tested on localhost by setting

```
DISTCC_HOSTS=127.0.0.1
```

Although `localhost` causes distcc to execute the job directly, using an IP address will cause it to make a TCP connection to a daemon on localhost. This is slower, but useful for testing.

2.7 Load Distribution Algorithm

When distcc is invoked, it needs to decide which of the volunteers in `DISTCC_HOSTS` should be used to compile a job. It uses a simple heuristic to try to spread load across machines appropriately.

You can imagine all of the compile machines as being leaky buckets, some with larger holes (faster CPUs) than others. The distcc client tries to keep water at the same level on each one (the same number of jobs running), preferring hosts occurring earlier in DISTCC_HOSTS. Over the course of a build, the faster machines will complete jobs more quickly, and therefore be topped up more quickly and do more work overall, but without the client ever actually needing to know which one is fastest.

This design has the advantage of not requiring the client to know in advance the speeds of the volunteers, and being quite simple to implement. It copes quite well with machines that are temporarily slowed down: they are just topped-up more slowly in the future.

Scheduling is coordinated between different invocations of the `distcc` client by lockfiles in the temporary directory. There is no coordination between clients running as different users, on different hosts, or with different `TMPDIR` paths.

On Linux, scheduling slightly too many jobs on any machine is quite harmless, as long as the number is not so high that the machine begins thrashing. So it's OK to provide a `-j` number substantially higher than the number of available processors.

The biggest problem with this design is that it handles multiprocessor machines poorly: they probably ought to have jobs scheduled proportional to the number of processors. At the moment, the best thing is to run with a `-j` factor equal to the product of the maximum number of CPUs in any machine (`MAX_CPUS`) and the number of machines. This should make sure that roughly `MAX_CPUS` tasks run on every machine at all times, and will therefore keep all CPUs loaded, but will cause excessive task-switching on machines with fewer CPUs. Task switching is not very expensive on Linux so it is not a big problem, but it does lose a few percentage points of speed. This should be fixed in a future release.

2.8 Diagnostic Messages

Error messages or warnings from local or remote compilers are passed through to diagnostic output on the client. The compiler takes all file names and line numbers from pragmas in the preprocessed output, so error messages will always have the correct pathnames for files on the client.

distcc prints a message when it runs a command locally or remotely. For more information, set `$DISTCC_VERBOSE` and look at the server's log file.

By default, distcc prints diagnostic messages to stderr. Sometimes these are too intrusive into the output of the regular compiler, and so they may be selectively redirected by setting the `$DISTCC_LOG` environment variable to a filename.

The current version of the distcc daemon writes diagnostic messages only to files on its own machine. (By default, it uses the syslog `daemon` channel.) If compilation is failing, please examine the log file on the relevant volunteer machine.

2.9 distcc Exit Codes

The exit code of distcc is normally that of the compiler: zero for successful compilation and non-zero otherwise.

If distcc fails to distribute a job to a selected volunteer machine, it will try to run the compiler locally on the client. distcc only tries a single remote machine for each job.

distcc tries to distinguish between a failure to distribute the job, and a "genuine" failure of the compiler on the remote machine, for example because of a syntax error in the program. In the second case, distcc does not re-run the compiler locally, and returns the same exit code as the remote compiler.

If the compiler exits with a signal, distcc returns an exit code of 128 plus the signal number, following Unix convention.

If distcc fails to run the compiler, it may return one one of the following error codes. These are also used by distccd.

100 `EXIT_DISTCC_FAILED`

Generic or unspecified failure in distcc.

102 `EXIT_BIND_FAILED`

Failed to bind and listen on network socket. Port may already be in use.

103 `EXIT_CONNECT_FAILED`

Failed to establish network connection or listen on socket. The host may be invalid or unreachable, or there may be no daemon listening.

104 `EXIT_COMPILER_CRASHED`

The underlying compiler exited because of a signal. This probably indicates a compiler bug, or a problem with the hardware or OS on the server. (Obsolete in 0.13.)

105 `EXIT_OUT_OF_MEMORY`

Obvious.

106 `EXIT_BAD_HOSTSPEC`

`$DISTCC_HOSTS` was undefined, empty, or syntactically invalid. (At the moment, you should never see this code because distcc will fall back to building locally. Let me know if you would prefer a hard error.)

107 `EXIT_IO_ERROR`

There was a disk or network IO error while distributing the job. For example, the network may be failing, or a disk on either end may be out of space.

108 `EXIT_TRUNCATED`

The network socket was closed unexpectedly. This probably indicates a network problem or distcc bug.

109 `EXIT_PROTOCOL_ERROR`

The distcc internal network protocol was not followed by the remote program. This probably indicates a network problem or distcc bug.

110 `EXIT_COMPILER_MISSING`

The specified compiler was not found in the path of the server.

2.10 Cross-Compilation

Cross compilation means building programs to run on a machine with a different processor, architecture, or operating system to where they were compiled. distcc supports cross compilation, including teams of mixed-architecture machines, although some changes to the compilation commands may be required.

The compilation command passed to distcc must be one that will execute properly on every volunteer machine to produce an object file of the appropriate type. If the machines have different processors, then simply using `distcc cc` will probably **not** work, because that will normally invoke the volunteer's native compiler.

Machines with the same instruction set but different operating systems may not necessarily generate compatible .o files. Empirically it seems that the native FreeBSD compiler generates object files compatible with Linux for C programs, but not for C++. It may be a good idea to install a Linux cross compiler on BSD volunteers.

Different versions of the compiler may generate incompatible object files. This seems to be much more of a problem with C++ than with C, because the C++ ABI (application binary interface) has changed in recent years. If you will be building C++ programs, it may be a good idea to install the same version of `g++` on all machines.

Several different gcc configurations can be installed side-by-side on any machine. If you build gcc from source, you should use the `--program-suffix` configuration options to cause it to be installed with a name that encodes the gcc version and the target platform, such as `gcc-3.2` or `gcc-3.2-arm`. The compiler must be installed under the same name on the client and on every volunteer machine.

gcc also has `-b` and `-V` options to specify a target and version, but at the moment the gcc team recommend using a qualified compiler name instead.

For more information on cross-compiling, see *Specifying Target Machine and Compiler Version* in the gcc manual, and the gcc installation guide.

2.11 distcc Compatibility

2.11.1 distcc with ccache

distcc works well with the ccache tool for caching compilation results. To use the two of them together, simply set

```
CC='ccache distcc'
```

2.11.2 distcc with autoconf

distcc works quite well with autoconf.

`DISTCC_VERBOSE` can give autoconf trouble because autoconf tries to parse error messages from the compiler. If you redirect distcc's diagnostics using `DISTCC_LOG` then it seems to be fine.

Some autoconf-based systems "freeze" the compiler name used for configure into their Makefiles. To make them use distcc, you must either set `$CC` when running `./configure`, and/or override `$CC` on the right-hand-side of the Make command line.

Some poorly-written shell scripts may assume that `$CC` is a single word. At the moment the best fix is to use a shell script that calls `distcc`.

2.11.3 distcc with libtool

Some versions of libtool seem not to cope well when CC is set to more than one word, such as `"distcc gcc"`. Setting `CC=distcc`, which is supported in 0.10 and later, seems to work well.

2.11.4 distcc with Gentoo Linux

Gentoo is a "ports"-based free software distribution, in which packages are always built from source on installation. distcc works well with Gentoo to speed installation.

You can install distcc either using the upstream tarball from `distcc.samba.org` (which may be newer), or using `emerge distcc` to get the Gentoo port, which may be better integrated. You can also get ccache with `emerge ccache`.

After installing distcc, starting servers and setting `DISTCC_HOSTS`, use a command like this to build addition packages using distcc:

```
MAKEOPTS="-j9" CC=distcc CXX=distcc emerge
packagename
```

The `-j9` is only an example; set it appropriately for the number of processors available.

To use distcc for all compilations, edit `/etc/make.conf` to include something like these lines, appropriately adjusted for your systems:

```
CC="distcc"
CXX="distcc g++"
MAKEOPTS="-j4"
DISTCC_HOSTS="host1 host2 localhost"
```

2.11.5 distcc with gcc dependency computation

gcc has the ability to produce information about header dependencies as a side-effect of preprocessing. These can be included in Makefiles in various ways to make sure that files are up-to-date.

This feature is enabled using `-MD`, `-M` and related options.

Unfortunately, gcc changed the behaviour of this feature between gcc 2.95 and 3.x in such a way that it seems properly for distcc to generally support it. The difficulty is that the filename to which dependencies are written depends in a very complicated way on the gcc command line. distcc needs to change the command line to run the preprocessor locally and the compiler remotely, and this can sometimes cause problems. (This also causes problems for Makefiles that are supposed to work with both versions of the compiler.)

`-M` causes gcc to produce dependency information instead of compiling. distcc understands this and passes the option straight through to gcc. It should work correctly.

With gcc 2.95, `-MD` always writes dependencies into the preprocessor's working directory. distcc should work fine.

With gcc 3.2, `-MD` writes the output into either the source directory or output directory, depending on the presence of the `-o` option. However, gcc 3.2 also has a `-MF` option that can be used to explicitly set the dependency output file, and this works well with distcc.

In summary: for gcc 2.95, no changes are required. For gcc 3.2, `-MF` should be used to specify the file to write dependencies to.

2.12 File Metadata

distcc transfers only the binary contents of source, error, and object files, without any concern for metadata, attributes, character sets or end-of-line conventions.

distcc never transmits file times across the network or modifies them, and so should not care whether the clocks on the client and volunteer machines are synchronized or not. When an object file is received onto the client, its modification time will be the current time on the client machine.

Chapter 3

The distccd Server

The distccd server may be started either from a super-server such as `inetd`, or as a stand-alone daemon.

distccd does not need to run as root and should not.

distccd does not have a configuration file; it's behaviour is controlled only by command-line options and requests from clients.

3.1 Invoking distccd

These options may be used for either inetd or standalone mode.

If you want to see if the daemon started properly, look in the log file. By default this is something like `/var/log/daemon` or `/var/log/messages`, depending on your system.

`--help`

Explains usage of the daemon and exits.

`--version`

Shows the daemon version and exits.

`-N, --nice NICENESS`

Makes the daemon more nice about giving up the CPU to other tasks on the machine. *NICENESS* is a value from 0 (regular priority) to 20 (lowest priority). This option is good if you want to run distccd in the background on a machine used for other purposes.

`-p, --port PORT`

Set the TCP port to listen on. (Standalone mode only.)

`-P, --pid-file FILE`

Save daemon process id to file.

–verbose

Include debug messages in log.

`--no-fork`

Don't fork or detach (for debugging).

`--no-fifo`

Send input to the compiler by writing to a temporary file, rather than using a pipe. This is required when the server's temporary directory is on NFS, on at least some machines. It may be faster in some circumstances, but probably is not.

`--log-file=FILE`

Send messages here instead of syslog.

`--log-stderr`

Send log messages to stderr, rather than to a file or syslog. This is mainly intended for use in debugging.

`--inetd`

Serve a client connected to stdin/stdout. As the name suggests, this option should be used when distccd is run from within a super-server like `inetd`. distccd assumes inetd mode when stdin is a socket.

`--daemon`

Bind and listen on a socket, rather than running from inetd. This is used for standalone mode. distccd assumes daemon mode at startup if stdin is a tty, so `--daemon` should be explicitly specified when starting distccd from a script or in a non-interactive ssh connection.

3.2 distccd Exit Codes

As for distcc 2.9 ().

3.3 distccd Environment Variables

`DISTCC_SAVE_TEMPS`

If set to `1`, temporary files are not deleted after use. Good for debugging.

Chapter 4

distcc Internals

4.1 Protocol

distcc uses a simple, application-specific protocol running directly over a TCP socket. A new request socket is opened for each job.

The request and response begin with a magic number and version number, allowing incompatible versions or misconfigurations to be identified. At the moment there is only one deployed protocol version, and no attempt to support backward or forward compatibility, though this could be added in the future.

The request and response consist of tagged, length-preceded elements. Each element of the request contains a four-character ASCII token, an eight-digit ASCII hexadecimal length or value, and, depending on the tag, a byte stream whose length is determined by the hexadecimal field.

The complete request is sent to the server before the reply begins. Opening the TCP socket is performed concurrently with execution of the preprocessor on the client.

The request from the client contains

1. Magic number and version
2. Compiler command line
3. Preprocessed source code

The response from the server contains

1. Magic number and version
2. Compiler exit code & status
3. Compiler error messages
4. Compiler stdout
5. Object file (if any)

Consult the source for more information.

4.2 Working files

distcc stores working files in a subdirectory of `/tmp`. These include synchronization files, and compiler input/output temporary files.

Temporary files should normally be cleaned up when the program exits. If distcc misbehaves, these files may be useful in tracking down the cause. Any that remain can be removed by the system's temporary file reaper, or by hand.

4.3 Lock files

distcc uses lock files to allow each client to balance its jobs across available volunteer machines. For each volunteer host, a zero-length file is created. Clients using that volunteer hold a `flock` lock on the file while running.