distcc User Manual

Martin Pool mbp@samba.org

$Revision: 1.64 $ $Date: 2002/09/03 00:55:00 $, for distcc-0.9

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. The design is his own invention.

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
```

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 gcc'
```

NOTE: You cannot just set `CC=distcc`, because distcc needs to know the name of the real compiler.

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 gcc' ./configure
$ make CC='distcc gcc' 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.

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 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.

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.)

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.

`gcc` has two options to select at run time the target platform (`-b`) and the gcc version (`-V`) to be used. Several different gcc configurations can be installed side-by-side on any machine, and these options are used by the top-level "driver" program to switch between them. For more information, see *Specifying Target Machine and Compiler Version* in the gcc manual.

For example, adding `-b i386-linux` to `$CFLAGS` ought to make sure the correct compiler is invoked to build Linux/x86 programs. This has no particular effect if all the volunteers are natively of that type, but is very useful if some of the volunteer machines are different: either the correct compiler will be used, or you will see an error message like this if it is not installed.

```
gcc: installation problem, cannot exec ‘cpp0’: No such file or directory
gcc: file path prefix ‘/usr/lib/gcc-lib/i386-freebsd/2.95.4/’ never used
```

The parts of gcc particular to target machines and versions are normally kept in the directory `/usr/local/lib/gcc-lib/MACHINE/VERSION`.

Alternatively, you might specify as the compiler command the name of a script or symbolic link that calls the appropriate version of gcc on each machine. For example:

```
CC='distcc gcc-i386-linux'
```

In general, using the `-b` option is probably better, because it does not require any special creation of scripts on the volunteer machines beyond installing the appropriate gcc configuration. However, using a special compiler name may be useful if you need to make sure that a particular version of gcc's driver program is used, perhaps because you are testing gcc. This approach might also be useful with compilers other than gcc that have no built-in mechanism for choosing a target.

Suggestions for other ways to support cross-compilation or automatically detecting incompatibilities are welcome.

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 gcc'
```

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"`. The problem is under investigation.

2.11.4 distcc with MOC

MOC is the Qt meta-object compiler.

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.

`--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

Bugs and Future Work

4.1 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> .)

4.2 Test Suite

distcc has a test suite written in Python using the *PyUnit* <http://pyunit.sourceforge.net/> framework. It does not yet exercise all functionality, but is improving. If you discover a bug, or write new functionality, please try to add corresponding tests to make sure that the fix keeps working in the future.

4.3 Known Bugs and Restrictions

There are no known cases where distcc will produce incorrect code, but they may exist. There are some restrictions on distcc, and some possible optimizations that are not yet implemented.

An important general goal is that the code should stay as simple as possible, and secondarily be portable to reasonably current Unix-like systems. Complicating the code, or adding large dependencies is undesirable unless there's an overwhelming advantage.

- distcc needs to handle `$COMPILER_PATH` and `$GCC_EXEC_PREFIX` in some sensible way, if there is one. Not urgent because I have never heard of them being used.
- distcc might usefully verify that the compiler versions and critical parameters are compatible on all machines, for example by running -V. This really should be done in a way that preserves the simplicity of the protocol: we don't want to interactively query the server on each request. Perhaps distcc ought to add `-b` and `-V` options to the compiler, based on whatever is present on the current machine? Or perhaps the user should just do this.
- distcc could reasonably not transmit `-D`, `-I` and any other options that we're sure are handled only by the preprocessor. This would make the server logs slightly more clear and readable and possibly be a very tiny performance boost.

- distcc also needs some standard scripts for doing performance measurement.
- Server-side errors are not directly visible to the client. (The user needs to look at the server's log file.) The server's error messages should be passed back to the client.
- Compressing network traffic (probably using gzip) might help when the network is much slower than the CPUs. `gzip -3` is cheaper in CPU than `cpp` and gives a substantial reduction in the size of a `.i` file.
- distcc waits for too long on unreachable hosts. We probably need to timeout after about a second and build locally. Probably this should be implemented by `connect()` in non-blocking mode, bounded by a `select`.
- The client should have a medium-term local cache about unusable servers, to avoid always retrying connections. Several different cases (unreachable, host down, server down, server broken) will produce slightly different errors.
- Attackers can cause arbitrary damage if they can connect to the volunteer's port, or impersonate a volunteer. distcc should therefore only be used on trusted networks. Running over `ssh` or some other security mechanism might be possible, but will cause some performance loss.
- 4200 is not a registered port; I just picked it out of my hat. If distcc proves popular, it ought to get a proper IANA-allocated port and service name. 4200 is in fact already registered to "VRML Multi User Systems".
- distcc has only been tested on GNU/Linux and BSD. It probably will have minor portability bugs on other platforms.
- distcc ought to work with compilers other than GNU cc, but it has not been tested.
- If a Makefile contains race conditions that make it unsafe for parallel execution then distcc will lose in the same way as a local compiler. Limitations of the Make language mark it hard to write some parallel rules correctly.
- If the Makefile hardcodes the name of the compiler rather than using `$(CC)` then it may have to be updated to work properly. ccache handles these situations by allowing itself to be installed in place of `gcc`. It examines the name under which it was invoked and decides to run another compiler. It may be possible for `distcc` to piggy-back on that.
- There are probably some valid compiler command lines that distcc will fail to understand, and which will therefore be run locally rather than distributed. `cc` argument parsing is complex and not completely standardized.
- distcc (by design) can't handle compilers that need to read other files from the local filesystem. This might be a problem with such things as profile-directed optimizers. distcc tries to detect such commands and run them locally, but there may be cases which are not handled properly.
- distcc's protocol and file IO would probably have trouble with source or object files over 2GB in size. I've never heard of a .c or .o file that large, and I rather suspect gcc would not handle them well either.
- distcc has no protection against network transmission errors other than that in TCP and Ethernet (which are actually generally quite good.) In that respect it is like FTP, NFS, HTTP, and most others. Using gzip compression would allow strong error detection, and using ssh would allow strong error correction. Alternatively we might just send a checksum (e.g. MD4) of the files.

- A GUI to show progress of compilation and distribution of load would be neat. Probably the most sensible way is to make it parse `$DISTCC_LOG`.
- Sometimes `cc` is used just for assembly. This too could be done remotely, by handling the `.s` extension as preprocessed source, and `.S` as unpreprocessed source.
- distcc could support cross-compilation by a per-volunteer option to override the compiler name. On the local host, it might invoke `gcc` directly, but on some volunteers it might be necessary to specify a more detailed description of the compiler to get the appropriate cross tool. This might be insufficient for Makefiles that need to call several different compilers, perhaps `gcc` and `g++` or different versions of gcc. Perhaps they can make do with changing the DISTCC host settings at appropriate times.
- distcc ought not to write any messages to stderr unless there really is a problem or warning or verbosity has been requested, because it confuses ccache.
- distcc could easily handle IPv6, but it doesn't yet. The new sockets API does not work properly on all systems, so we need to support both.
- We ought to link against the tcpwrappers library to allow access control through `/etc/hosts.allow`. That's a moderately good level of security: certainly much cheaper than SSH. Unfortunately this may suddenly break daemons, because many machines are configured to disallow everything by default. We need to either make it a configure option, or just put a big warning in the documentation.
- It would be nice to have a `--ping` client option to contact all the remote servers, and perhaps return some kind of interesting information. This is almost certainly just chrome; though.
- It would be nice to put distcc and appropriate compilers on the

  LNX-BBC <http://www.lnx-bbc.org/> . This could be pretty small because only the compiler would be required, not header files or libraries.
- Also, it would be nice to have an easily installable package for Windows that makes the machine be a Cygwin-based compile volunteer. It probably needs to include cross-compilers for Linux (or whatever), or at least simple instructions for building them.
- Automatic detection ("zero configuration") of compile volunteers is probably not a good idea, because it might be complicated to implement, and would possibly cause breakage by distributing to machines which are not properly configured.
- Notwithstanding the previous point, centralized configuration for a site would be good, and probably quite practical. Setting up a list of machines centrally rather than configuring each one sounds more friendly. The most likely design is to use DNS `SRV` records (RFC2052), or perhaps multi-RR `A` records. For exmaple, `compile.ozlabs.foo.com` would resolve to all relevant machines. Another possibility would be to use SLP, the Service Location Protocol, but that adds a larger dependency and it seems not to be widely deployed.

4.4 Large-scale Distribution

distcc in it's present form works well on small numbers of close machines owned by the same people. It might be an interesting project to investigate scaling up to large numbers of machines, which potentially do not trust each other. This would make distcc somewhat more like other "peer-to-peer" systems like Freenet and Napster.

4.5 Execution across SSH

Running distcc across *OpenSSH* <http://www.openssh.org/> has several security advantages and should be supported in the future. They include:

1. Volunteer machines will not need to open an additional network-facing service.
2. Only authenticated users can use a volunteer machine.
3. Clients have some guarantees that their connections to a volunteer are not being spoofed.

Using SSH is greatly preferable to developing and maintaining a custom security protocol.

If the client or volunteer is subverted, then the other party is not protected. (For example, if the administrator of the volunteer is malicious, or if the volunteer has been compromised, then compilation results might contain trojans.) However, this is the case for practically every Internet protocol.

Using SSH will consume some CPU cycles in computation on both client and volunteer.

A simple implementation would be trivial, since the daemon already works on stdin/stdout. However, this might perform poorly because SSH takes quite a long time to open a connection.

Connections should be hoarded by the client. If the client doesn't already have an ssh connection to the server, `distcc` should fork, with a background task holding the connection open and coordinating access.

4.6 Load Balancing

When running a job locally (such as cpp or ld), distcc ought to count that against the load of localhost. At the moment it is biased towards too much local load.

distcc needs a way to know that some machines have multiple CPUs, and should accept a proportionally larger number of jobs at the same time. It's not clear whether multiprocessor machines should be completely filled before moving on to another machine.

If there are more parallel invocations of `distcc` than available CPUs it's not clear what behaviour would be best. Options include having the remaining children sleep; distributing multiple jobs across available machines; or running all the overflow jobs locally.

In fact, on Linux it seems that running two tasks on a CPU is not much slower than running a single task, because the task-switching overhead is pretty low.

Problems tend to occur when we run more jobs than will fit into available physical memory. It might be nice if there was a "batch mode" scheduler that would finish one before running the next, but in the absence of that we have to do it ourselves. I can't see any clean and portable way to determine when the compiler is using too much memory: it would depend on the RSS of the compiler (which depends on the source file), on the amount of memory and swap, and on what other tasks are running. In addition, on some small boxes compiling large code, you may actually want (or need) to have it swap sometimes.

In addition, it might be nice to have a `--max-load` option, as for GNU Make, to tell it not to accept more than one job (or more than zero?) when the machine's load average is above that number. I don't know if there is a portable way to determine load average but perhaps we can borrow code from GNU Make. On Linux it's easy: just read `/proc/loadavg`.

A server-side administrative restriction on the number of consecutive tasks would probably be a sufficient approximation.

Oscar Esteban suggests that when the server is limiting accepted jobs, it may be better to have it accept source, but defer compiling it. This implies not using fifos, even if they would otherwise be appropriate. This may smooth out network utilization. There may be some undesirable transient effects where we're waiting for one small box to finish all the jobs it has queued.

Chapter 5

Results

5.1 Introduction

The purpose of distcc is to reduce elapsed time to build from a clean directory.

The most important result is elapsed time, measured by `time(1)` on the client machine.

Non-blank, non-comment physical lines of code are measured using `wc -l $(find . -name '*.[ch]')`.

One complete build is run before measurement to attempt to minimize disk caching effect. `make distclean` is run between builds.

5.2 Test Environment

Machines used for testing:

anomic*, *jonquille*, *nevada

Hewlett-Packard X2000 Linux workstations, 1x1GHz Pentium IV, 1GB RAM, 1x20GB SCSI, Linux 2.4.18, Debian GNU/Linux, `ext3fs`, Debian's gcc-2.95.4.

anomic, *jonquille* and *nevada* are connected by a 10Mbps non-switched Ethernet hub.

Machines may be interactively used during testing but are generally lightly loaded.

The `timebuild` script in the source directory (after 0.2) is used to repeatedly build programs for measurement.

5.3 Samba

```
More results using cvs head as of Wed May 15 17:44:47 EST 2002

Building samba head, using plain gcc on anomic:

time make:

time DISTCC_HOSTS='localhost jonquille nevada' make -j4
```

```
real    4m4.402s
user    3m41.120s
sys     0m13.460s

time make CC=gcc
real    7m15.747s
user    6m40.920s
sys     0m14.460s
```

Notes: Samba uses lots of header files in every source file, so the preprocessed source is very large, and it is also typically slow to compile on single machines.

5.4 glib

Using distcc CVS on 20 May 2002, building glib-2.0.1 (100,911 lines in `*.[ch]`).

autoconf-generated `configure` scripts run compilations in series, so represent a worst case for distcc. When *jonquille* is the first nominated host, all work will be done remotely, and the `configure` script runs 4.34s (26%) slower. When *localhost* is first, all work is done locally and the overhead of running through distcc is 0.68s (4%) elapsed time.

Note that using distcc should never make any difference to the results of the configure script, assuming that all the installed compilers behave equivalently.

Chapter 6

distcc Internals

6.1 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.

6.2 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.

6.3 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.

Appendix A

Detailed Results

A.1 Results for glib

```
CC=gcc
time make
real    0m41.326s
user    0m34.430s
sys     0m5.380s

time make -j3
real    0m42.148s
user    0m35.280s
sys     0m4.970s

export DISTCC_HOSTS='localhost jonquille nevada'
export CC='distcc gcc'
make distclean
time ./configure
real    0m16.948s
user    0m10.740s
sys     0m4.410s

time make
real    0m42.290s
user    0m35.000s
sys     0m5.710s

time make -j3
real    0m28.121s
user    0m20.650s
sys     0m5.740s

time make -j5
real    0m28.777s
user    0m21.060s
sys     0m5.410s
```

```
export CC='distcc gcc'
export CFLAGS='-pipe'
make distclean
time ./configure
real    0m15.672s
user    0m10.020s
sys     0m4.030s

make clean; time make
real    0m25.287s
user    0m19.130s
sys     0m4.800s

make clean; time make -j3
real    0m20.921s
user    0m14.700s
sys     0m4.830s

make clean; time make -j5
real    0m20.738s
user    0m14.060s
sys     0m4.910s
```

```
export CC='gcc'
export CFLAGS='-pipe'
make distclean
time ./configure
real    0m15.336s
user    0m9.340s
sys     0m4.360s

make clean; time make
real    0m24.841s
user    0m18.980s
sys     0m4.610s

make clean; time make -j3
real    0m24.446s
user    0m18.910s
sys     0m4.550s
```