



ElectricAccelerator Electric Make Users Guide

version 6.1

Electric Cloud, Inc.
676 W. Maude Avenue
Sunnyvale, CA 94085
www.electric-cloud.com

Copyright © 2002 - 2012 Electric Cloud, Inc. All rights reserved.

Published April 2012

Electric Cloud® believes the information in this publication is accurate as of its publication date. The information is subject to change without notice and does not represent a commitment from the vendor.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED “AS IS.” ELECTRIC CLOUD, INCORPORATED MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying, and distribution of any ELECTRIC CLOUD software described in this publication requires an applicable software license.

Copyright protection includes all forms and matters of copyrightable material and information now allowed by statutory or judicial law or hereinafter granted, including without limitation, material generated from software programs displayed on the screen such as icons, screen display appearance, and so on.

The software and/or databases described in this document are furnished under a license agreement or nondisclosure agreement. The software and/or databases may be used or copied only in accordance with terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license or nondisclosure agreement.

Trademarks

Electric Cloud, ElectricAccelerator, ElectricCommander, ElectricInsight, Electric Make, and SparkBuild are registered trademarks or trademarks of Electric Cloud, Incorporated.

Electric Cloud products—ElectricAccelerator, ElectricCommander, ElectricInsight, and Electric Make—are commonly referred to by their “short names”—Accelerator, Commander, Insight, and eMake—throughout various types of Electric Cloud product-specific documentation.

Cywin is a trademark of Red Hat, Inc., Microsoft Visual Studio is a registered trademark of Microsoft Corporation, ClearCase is a registered trademark of IBM Rational, and any other product names mentioned in this guide may be trademarks or registered trademarks of their respective owners and are hereby acknowledged.

Chapter 1	Electric Make Introduction	
	About ElectricAccelerator	1-1
	Understanding Component Interactions	1-2
	Electric Make and EFS	1-2
	Electric Make and Cluster Manager	1-2
	About ElectricAccelerator Virtualization.....	1-2
	Electric File System (EFS).....	1-2
	System Registry (Windows Only)	1-3
	User Accounts.....	1-3
	Environment Variables.....	1-3
Chapter 2	Configuration	
	Understanding Build Parts	2-1
	Defining Your Build.....	2-1
	Build Sources	2-1
	Build Tools.....	2-1
	Build Environment.....	2-2
	Configuring Your Build	2-2
	Configuring Sources	2-3
	Configuring Tools	2-3
	Configuring Environment Variables	2-4
	Using the Proxy Command.....	2-5
Chapter 3	Starting a Build	
	Working with Electric Make	3-1
	Basic Electric Make Settings	3-1
	Using Electric Make Options.....	3-1
	Overriding Build Class Settings.....	3-2
	Setting the Cluster Manager Host and Port.....	3-2
	Setting the Electric Make Root Directory.....	3-2
	Setting Electric Make Emulation for Ant Builds (all platforms).....	3-3
	Setting Electric Make Emulation for Windows	3-4
	Sample Build.....	3-4
	ElectricAccelerator Developer Edition	3-5
	Monitoring Build Results.....	3-6
	Using Break Points	3-6
	Terminating a Build	3-7
	Shutting Down Cluster Hosts during Builds.....	3-8
	Priority Pools	3-8

Chapter 4	Organizing Builds with Build Classes	
	About Build Classes	4-1
	Tag Definitions	4-1
	Build Class Examples	4-2
	Creating and Modifying Classes.....	4-3
	Using Electric Make Variables	4-4
Chapter 5	Make Compatibility	
	Invoking eMake	5-1
	Single Make Invocation.....	5-2
	Commands that Read from the Console	5-2
	Transactional Command Output.....	5-3
	Stubbed Submake Output	5-4
	Submake Stubs.....	5-4
	Submake Stub Compatibility.....	5-6
	Using Subbuilds.....	5-8
	Subbuild Database Generation	5-8
	Run a Build Using Subbuild.....	5-9
	Limitations.....	5-9
	Hidden Targets.....	5-10
	Wildcard Sort Order.....	5-12
	Delayed Existence Checks.....	5-12
	Enabling Multiple Remakes (GNU Make Only).....	5-13
	NMAKE Inline File Locations (Windows Only)	5-14
	How eMake Processes MAKEFLAGS.....	5-14
Chapter 6	Third-party Integrations	
	Using Cygwin with ElectricAccelerator (Windows Only).....	6-1
	Using ClearCase with ElectricAccelerator	6-1
	Configuring ElectricAccelerator for ClearCase	6-1
	Performance Considerations.....	6-5
	Using Visual Studio with ElectricAccelerator.....	6-5
	Eclipse	6-6
Chapter 7	Dependency Management	
	ElectricAccelerator eDepend	7-1
	Dependency Generation	7-1
	The Problem	7-2
	eDepend Benefits.....	7-2
	Enabling eDepend.....	7-2
	How Does eDepend Work?	7-3
	ElectricAccelerator Ledger File.....	7-5
	The Problem	7-5
	The Electric Make Solution.....	7-6
	Managing the History Data File	7-7
	Setting the History File Location.....	7-7
	Selecting History File Options	7-7
Chapter 8	Annotation	
	What is Annotation?	8-1
	Configuring eMake to Generate an Annotation File	8-1
	Annotation File Splitting	8-2
	Working with Annotation Files	8-2
	Metric Values in Annotation Files	8-5

Chapter 9	Performance Tuning	
	Managing Temporary Files	9-1
	Configuring the Electric Make Temporary Directory	9-1
	Deleting Temporary Files.....	9-2
	Running a Local Job on the Make Machine	9-2
	Serializing All Make Instance Jobs.....	9-4
	Splitting PDBs Using hashstr.exe	9-4
Chapter 10	Electric Make Command-line Options and Environment Variables	
	Introduction.....	10-1
	Command-line Options and Environment Variables List	10-2
Chapter 11	Electrify	
	What is Electrify?	11-1
	Electrify as Part of the Build Process	11-1
	Running Electrify on Windows.....	11-2
	Running Electrify on Linux	11-2
	Important reminders	11-3
	Electrify arguments	11-3
	Additional Information	11-4
Appendix A	ElectricAccelerator Job Types	
	Overview	A-1
	ElectricMake Jobs	A-1
	Electrify Jobs	A-4
Appendix B	Electric Make Debug Log Levels	
	Overview	B-1
	Enabling eMake Debug Logging	B-1
	eMake Debug Log Level Descriptions	B-1



Electric Make Introduction

Electric Make®, the main build application in ElectricAccelerator®, is a new Make version invoked interactively or through build scripts. It reads makefiles in several different formats, including GNU Make and Microsoft NMAKE. Electric Make distributes commands to the cluster for remote execution and services file requests.

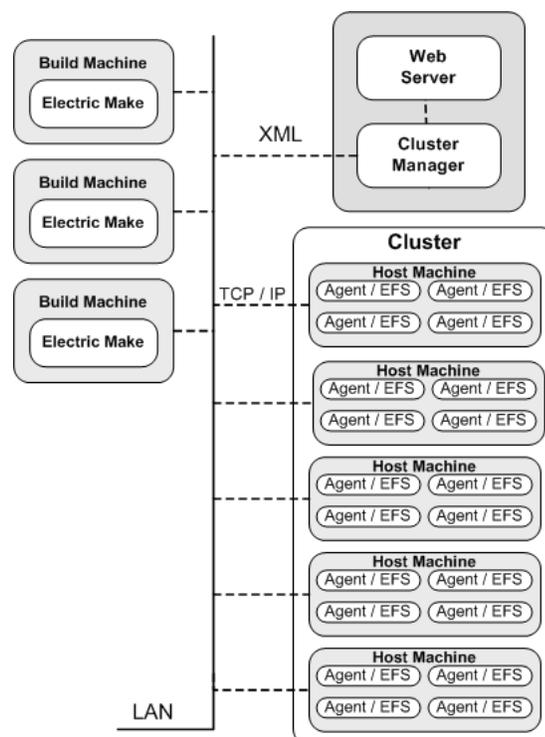
About ElectricAccelerator

ElectricAccelerator is a software build accelerator that dramatically reduces software build times by distributing the build over a large cluster of inexpensive servers. Using a patented dependency management system, ElectricAccelerator identifies and fixes problems in real time that would break traditional parallel builds. ElectricAccelerator plugs into existing Make-based infrastructures seamlessly and includes web-based management and reporting tools.

During ElectricAccelerator installation, the following components were installed:

- Electric Make® (eMake)
- Electric File System (EFS)
- ElectricAccelerator® Agents (Electric Agent or Agent)
- Cluster Manager
- Electrify

The following diagram illustrates where Electric Make “lives” within ElectricAccelerator.



Understanding Component Interactions

To a user, ElectricAccelerator may appear identical to other Make versions—reading makefiles in several different formats and producing identical results. Using a cluster for builds is transparent to the ElectricAccelerator user.

Important differences in ElectricAccelerator build processing versus other distributed systems:

- ElectricAccelerator components work together to achieve faster, more efficient builds. Instead of running a sequential build on a single processor, ElectricAccelerator executes build steps in parallel on a cluster of hosts.
- Job results are isolated until the job completes. If an Agent fails during a job, ElectricAccelerator discards any partial results it might have produced and reruns the job on a different Agent.
- Missing dependencies discovered at runtime are collected in a history file that updates each time a build is invoked. ElectricAccelerator uses this collected data to improve performance of subsequent builds.

Electric Make and EFS

High concurrency levels in ElectricAccelerator are enabled by the Electric File System (EFS). When a job such as a compilation runs on a host, it accesses files such as source files and headers through EFS. EFS records detailed file access data for the build and returns that data to Electric Make.

When a job requests information not found in the EFS kernel-level cache, EFS issues a request to the Agent. Using a TCP-based protocol, the Agent relays the request to Electric Make.

Electric Make acts as a file server for Agents, reading the correct file version from file systems on its machine and passing that information back to the Agents. Agents retain different file version information and do not rely on Electric Make's file sequencing ability to provide the correct version for a job. The Agent receives file data, downloads it into the kernel, notifying EFS, which then completes the original request. At the end of a job, Electric Agent returns any file modifications to Electric Make so it can apply changes to its local file systems.

Electric Make and Cluster Manager

When Electric Make is invoked on the build machine, it communicates with Cluster Manager to acquire a set of Agents it can use for the build. When Electric Make finishes, it sends Cluster Manager the build results, and tells Cluster Manager that Agents are free now to work on other builds. If more than one build is invoked, Cluster Manager allocates agents using a priority-based algorithm. Builds with the same priority share Agents evenly, while higher priority builds are allocated more Agents than lower priority builds. By default, agents running on the same host machine are allocated to the same build. In real time, Cluster Manager dynamically adjusts the number of agents assigned to running builds as each build's needs change, which allows ElectricAccelerator to make the best use of cluster resources.

About ElectricAccelerator Virtualization

ElectricAccelerator is designed to virtualize parts of the build setup so cluster hosts can be configured correctly for the specific build they are executing. Specifically, ElectricAccelerator dynamically mirrors the following host system properties on the Agent:

- File system
- System registry (Windows only)
- User ID (UNIX only)
- Environment variables

Electric File System (EFS)

Files are the most important resource to distribute to hosts. When Electric Make starts, it is given a directory (or list of directories) called `EMAKE_ROOT`. All files in eMake root(s) are automatically mirrored on cluster hosts for the build duration. This powerful feature means you do not need to configure file sharing between the build system and the cluster—simply specify the `EMAKE_ROOT` and the hosts can access files on the host build system.

Almost any file visible to the host build system can be mirrored to the cluster—regardless of how the host system accesses that file (local disk, network mount, and so on). Both sources and tools can be mirrored, but there are important flexibility/performance “trade-offs” to consider when you include tools in the eMake root. See [“Configuring Your Build” on page 2-2](#).

When the build completes, EFS is unmounted and the files are no longer visible on the hosts—this ensures builds do not interfere with local host configuration.

Generally, `EMAKE_ROOT` can be set to include any existing directory on the host build machine. Files in these directories are automatically accessible by commands running on cluster hosts. The exceptions are detailed in [“Setting the Electric Make Root Directory” on page 3-2](#).

System Registry (Windows Only)

A side effect of running builds on Windows: Windows may execute tools that modify the system registry. For example, a build job step may install a program that includes registry modifications that are executed by a subsequent step. If job steps are run on different Agents without registry virtualization, the build may fail because registry modifications on one machine are not visible on another.

ElectricAccelerator solves this situation by mirroring the Windows registry in addition to the file system. You can specify a registry root using the command line `--emake-reg-roots=<path>`. Just as `EMAKE_ROOT` specifies a host file system subset to mirror, the registry root specifies which registry keys should be virtualized to Agents (for example, `HKEY_LOCAL_MACHINE\Software\My Product`). Access and modifications to keys and values under the registry root on Agents are then sent back to the host build system to ensure correct values are propagated to other Agents.

As with the file system, registry mirroring is active during the build only. After the build completes, the registry returns to its original configuration. Processes not running under an Agent on a host cannot see mirrored files and registry entries—they will see only the usual view of the machine.

User Accounts

The same user ID is used by the Electric Agent [running on each host, executing processes] and the Electric Make process started on the host build system. Using the same user ID ensures processes have the same permission and obtain the same results from system calls (such as `getuid`) as they would running serially on the user machine. On Windows, processes are executed by the Electric Agent service user. The Agent user can be configured at installation time, see the *ElectricAccelerator Installation and Configuration Guide*, “Initial Configuration” chapter.

Environment Variables

When Electric Make sends commands for execution to the Agent, it also sends environment variables. In most cases, this automatic environment virtualization is sufficient. Typically, variables that specify output locations, target architecture, debug-vs.-optimized, and so on, are propagated to tools on the host and their remote behavior correctly matches what they do locally. By default, environment variables sent by Electric Make override system-wide settings on the host.

For certain variables, override behavior is not desirable. For the build to operate correctly, tools running on the hosts must see the value from the local Agent, not the host build system. For example, the Windows `SYSTEMROOT` variable depends on the directory where Windows was installed on that machine—values may vary in different Windows releases. The `SYSTEMROOT` value from the host build system is frequently different than the value from the local Agent.

To accommodate these instances, Electric Make allows you to exclude specific environment variables from virtualization. In addition to variables you may explicitly choose to exclude, Electric Make automatically excludes the `TMP`, `TEMP`, and `TMPDIR` variables on all platforms; and the `COMSPEC` and `SYSTEMROOT` variables on Windows. For a complete description of environment variables, see [“Command-line Options and Environment Variables List” on page 10-2](#)

Understanding Build Parts

Software builds are often complex systems with many inputs that must be carefully set up and configured. Correctly distributing system processes over a cluster of computers or hosts requires efficient replication of relevant parts of the build setup on each host.

ElectricAccelerator software is designed to help virtualize your build environment on each cluster host so distributed jobs display the same behavior as compared to how those jobs run serially on one computer.

Because input configurations can vary from one build to the next, ElectricAccelerator virtualization is active only while the build is running. When the build completes, the host returns to its original state. Consequently, different builds can share a cluster without danger of conflicting changes corrupting host configuration.

This chapter describes how to configure ElectricAccelerator to ensure accurate, reliable builds. A complete list of command-line options and environment variables, some of which are referenced in this chapter, can be found in [Chapter 10, “Electric Make Command-line Options and Environment Variables.”](#)

Defining Your Build

Before configuring ElectricAccelerator, define which components go into your build precisely. Generally, a build has three input types:

SOURCES + TOOLS + ENVIRONMENT = BUILD

Build Sources

Build sources include all compiled or packaged files to build your product. For example, these sources include:

- Intermediate files generated during the build (for example, headers or IDL files)
- Makefiles and scripts that control the build
- Third-party source files read during the build
- Any file read by the build in source control

Build Tools

Tools used to create build output include make, compilers, linkers, and anything else operating on the sources during the build:

- Executable post-process tools (for example, *strip*)
- Static analyzers (for example, *lint*) if they run as part of the build
- Packaging tools (for example, *tar* or *zip*)

Sometimes the distinction between sources and tools is blurred. Consider these examples:

- A utility executable compiled from your sources during the build, run during later steps, but not part of final output

- Header source files that are part of the compiler (for example, `stdio.h`) or a third-party package, but not under source control

In this context, *sources* are those files that may change from one build to the next. Thus, a utility executable compiled from your sources as part of the build is considered a source.

By contrast, *tools* change infrequently—tools are often configured once and served from a central location (for example, an NFS share). A standard header such as `stdio.h` is usually considered part of the tool suite.

The distinction between inputs that can change between builds (sources) and inputs that can safely be assumed to be constant (tools) becomes important when configuring ElectricAccelerator virtualization. See [“Configuring Your Build” on page 2-2](#).

Build Environment

Your operating system environment is an essential part of your build. The operating system is easy to overlook because the environment usually is configured once per host and then ignored as long as builds function normally. But when a build is distributed across a shared cluster, it is important to identify which parts of the operating system could affect the build. Some inputs to consider include:

- User environment variables
- System registry (Windows only)
- Operating system version (including patches or service packs)
- User account and user permissions
- Host-specific attributes (for example, machine name, network configuration)

For each of these inputs, consider what impact (if any) they will have on the build.

Generally, some environment variables require correct settings for a build (for example, `PATH` or variables that specify output architecture or source/output file locations).

Another common example of how the operating system can affect the build occurs with the use of tools that require license management. If a tool license (for example, a compiler) is host-locked or it requires contacting a license server to operate, ensure the compiler on the host can acquire the license also (for example, cluster host names could be configured as valid license server clients).

Configuring Your Build

After defining your build environment and identifying all of its inputs, configure the cluster and Electric Make so the system correctly virtualizes:

- Sources, by setting `EMAKE_ROOT`
- Tools, by setting `EMAKE_ROOT` and on Windows systems, setting `--emake-reg-roots=path` (registry roots) using the command-line interface
- Environment, by setting `EMAKE_EXCLUDE_ENV`

In addition, some cases of virtualization and/or distribution of specific job steps are not desirable. For these cases you can configure ElectricAccelerator to:

- Run an individual command from the Agent back on the host system, using the “proxy command” function. See [“Using the Proxy Command” on page 2-5](#).
- Prevent remote job execution by using the `#pragma runlocal` function. See [“Running a Local Job on the Make Machine” on page 9-2](#).

Configuring Sources

To ensure Agent access to source files from the host build system, specify the root path to the source directories using the Electric Make `--emake-root` command-line option or the `EMAKE_ROOT` environment variable. See “[Setting the Electric Make Root Directory](#)” on page 3-2 for examples of setting the eMake root.

If you manually mount a filesystem or change automounter settings on agent machines *after* they are started, you must restart the Agents for them to recognize your changes.

Note: Agent errors regarding the establishment of the virtual filesystem for a particular build will be displayed if there are at least three errors. These errors would occur during the initial setup of the agent's build-specific environment but before any particular build step is run on that agent. The most common type of error involves eMake roots or Cygwin mounts, where virtual filesystem setup is specific to the build but not to any particular build step.

Configuring Tools

Cluster hosts must be able to execute all tools (compilers, linkers, and so on) required during the build. Three execution setup choices:

1. **Tools available on the cluster in the same locations as on the host build system**—this is the simplest setup and requires no special Electric Make or build system configuration.

This is the most common setup if tools are provided from a network mount (for example, an NFS or SMB server). If tools are not supplied from a network share, but are installed on the local disk on each host, those tools must be updated whenever tools on the build system change.

2. **Tools available on the cluster, but not necessarily in the same locations as the host build system**—this configuration is common in environments where users are allowed to install tools on their host system in non-standard locations.

In this case, care must be taken to ensure build steps executing on the hosts find the tools. Usually, this means making the `PATH` environment variable include standard locations used on the cluster hosts—any other variables referencing tool locations need to be modified similarly.

As in the first case, if tools are installed on the cluster hosts' local disk, they must be updated when tools are updated on the host system.

3. **Tools not installed on the cluster**—in this case, when no tools are installed on the cluster, `EMAKE_ROOT` is configured to include the tools directory so the compiler and linker are sent to the Agents by Electric Make [in the same manner as the source files].

This is the most flexible setup because it eliminates the need to install or update tools on the hosts. If tools change frequently or vary between branches, this setup can dramatically reduce setup cost. However, to gain this flexibility, you typically trade performance because network overhead is increased.

Another factor to consider is the possibility that configuration information for a virtualized tool may not work on the host. For example, it is not possible to virtualize the Microsoft Visual Studio IDE because of its tight integration with the local machine's registry.

In some special cases, it may be necessary to use tools outside of the virtualized file system. To support running such tools on the local build machine during a cluster build, see “[Using the Proxy Command](#)” on page 2-5.

Note: Electric Cloud does not recommend this feature for frequent use; use it with caution.

(Windows only) Tools that access or modify the system registry—in addition to files, tools on Windows may access or modify the system registry. Use the `--emake-reg-roots` command-line option to specify a key to mirror. You can specify more than one key by separating multiple entries with semicolons:

```
--emake-reg-roots=HKEY_LOCAL_MACHINE\Software\Foo;
HKEY_LOCAL_MACHINE\Software\Bar
```

In addition, you can specify exception keys to not mirror the system registry by prefixing the key with a “-” character. For example:

```
--emake-reg-roots=HKEY_LOCAL_MACHINE\Software\Foo;
-HKEY_LOCAL_MACHINE\Software\Foo\Base
```

which means “mirror the all keys and values under Foo except the keys in Foo\Base.

To ensure compatibility with Microsoft Visual Studio, the registry root specification automatically includes:

```
HKEY_CLASSES_ROOT;-HKEY_CLASSES_ROOT\Installer;
-HKEY_CLASSES_ROOT\Licenses
```

Configuring Environment Variables

Because Electric Make environment variables are propagated automatically to Agents, most commands running on an Agent will run with the correct environment without modifications.

However, two important exceptions exist.

- As described in “Configuring Tools” on page 2-3, if tools are installed on the cluster host in different locations from the build system, the `PATH` variable (and other variables that reference tool locations) on the build system must be modified to include the locations for the cluster tools.
- Generally, differences between the build system and cluster hosts may indicate it is undesirable to override environment variables with eMake values. In this case, use the `--emake-exclude-env` command-line option or the `EMAKE_EXCLUDE_ENV` environment variable.

For example, consider a build system environment variable called `LICENSE_SERVER` that normally contains the license server name that the system should contact to obtain a tools license. If this variable is machine-specific, eMake overrides the correct machine-specific value on the cluster hosts with the value from the build system. To ensure eMake does not override `LICENSE_SERVER` with the value from the build system, use the option, `--emake-exclude-env`, when running eMake:

```
--emake-exclude-env=LICENSE_SERVER
```

You can specify more than one value by separating them with commas:

```
--emake-exclude-env=LICENSE_SERVER,TOOLS_SERVER
```

Some variables are almost always used to describe the local machine state, so Electric Make always excludes them from mirroring. These variables are:

```
TMP
TEMP
TMPDIR
```

For Windows, this list also includes:

```
COMSPEC
SYSTEMROOT
```

Configuring ccache

The only configuration required to use ccache with ElectricAccelerator is to set the `CCACHE_NOSTATS` environment variable. If you do not set this environment variable, the entire build becomes serialized because ccache continuously writes to a statistics file throughout the build. To learn more about ccache, refer to <http://ccache.samba.org/>.

Using the Proxy Command

Normally, Electric Make sends commands to the Agent for execution. In some cases, however, it may not be desirable or possible to execute a particular command on the Agent. You can use ElectricAccelerator's "proxy command" feature to run a command locally on the Electric Make machine *if both of the following are true*:

- You have a command that cannot run on the Agent, either because it returns incorrect results or because it is not available.
- The command in question *does not read or write build sources or output*—it only makes external [outside of the build] read-only queries.

The second item is particularly important because the command runs on the host build machine, outside of the virtualized file system. Because ElectricAccelerator cannot track the activity of this process, the dependency and conflict resolution mechanisms that prevent build output corruption are circumvented. It is also important the process be read-only [that the command make no changes to whatever system it is querying] because in parallel builds with conflicts, Electric Make could rerun the job producing unintended side effects.

Note: For these reasons, it is important to use the "proxy command" only when necessary. Where possible, try to ensure cluster hosts have the same tools installed as the build system.

During ElectricAccelerator installation, the `proxyCmd` binary is installed on every host.

- On Linux: `/opt/ecloud/i686_Linux/bin/proxyCmd`
- On Solaris: `/opt/ecloud/sun4u_SunOS/bin/proxyCmd`
- On Windows: `C:\ECloud\i686_win32\bin\proxyCmd.exe`

When invoked by the Agent, it is: `proxyCmd <program> <arg1> ...`

ElectricAccelerator executes `<program>` on the host build system and proxies the result back to the Agent so it can continue remote execution.

The simplest *safe* use of the `proxyCmd` is a source control system query. For example, a particular build step queries the source control system for branch identification using a tool called `getbranch` that it embeds in a version string:

```
foo.o:
    gcc -c -DBRANCH=`getbranch` foo.c
```

It is preferable to avoid installing and configuring a full deployment of the source control system on the host when only this simple query command is needed.

In the following example, the `proxyCmd` provides an efficient solution. By replacing `getbranch` with `proxyCmd getbranch`, you avoid having to install the `getbranch` tool and its associated components on the host:

```
foo.o:
    gcc -c -DBRANCH=`proxyCmd getbranch` foo.c
```

A less invasive implementation that does not require makefile modifications and allows compatibility with non-ElectricAccelerator builds is to create a link on all agent machines using the name of the tool [for example, 'getbranch'] found on the eMake host to `proxyCmd`. For Windows operating systems that do not support `symlinks`, use the `copy` command. When invoked under a different name, `proxyCmd` knows to treat the linked name as the `<program>` to execute:

```
# ln -s /opt/ecloud/i686_Linux/bin/proxyCmd /usr/bin/getbranch
```

Working with Electric Make

After all ElectricAccelerator components are installed, and before invoking Electric Make, you need to set a few basic Electric Make options. These options enable Electric Make to communicate with the Cluster Manager and Agents in the cluster hosts. This chapter describes basic Electric Make settings required for a build.

Basic Electric Make Settings

Configure Electric Make by setting options on the command line and/or setting environment variables. Both methods are described here.

In addition to the basic settings, additional Electric Make options are available for configurations specific to your requirements. All available configuration options are in [“Electric Make Command-line Options and Environment Variables” on page 10-1](#); specific use of these variables to enhance ElectricAccelerator performance are included in [“Performance Tuning” on page 9-1](#).

If you do not set the `--emake-emulation` option, Electric Make defaults to support of a subset of GNU Make 3.81 (see [“Unsupported GNU Make 3.81 Features” on page 5-2](#)).

Note: Electric Cloud recommends against running builds in `/tmp`.

Using Electric Make Options

Use Electric Make command-line options to set communication options with other ElectricAccelerator components and specify build preferences. The most important options to set are:

- The Cluster Manager host and port settings
`--emake-cm=<host:port>`
- The Electric Make root directory setting
`--emake-root=<path>`
- **Windows Only:** An additional setting is recommended for eMake emulation (`--emake-emulation`), which is covered in [“Setting Electric Make Emulation for Windows” on page 3-4](#).

If your build environment is fairly constant, you may want to use *environment variables* to avoid respecifying values for each build. Environment variables function the same way as command-line options, but command-line options take precedence over environment variables.

Overriding Build Class Settings

Certain command-line options also take precedence over Cluster Manager build class properties:

- `--emake-priority` overrides build class Priority but can be used only to lower the build priority
- `--emake-maxagents` overrides Max Agents if `--emake-maxagents` is set lower than the build class. The command line does not override Max Agents if `--emake-maxagents` is set higher than the build class.
- `--emake-annodetail` overrides annotation level settings
- `--emake-resource` overrides resource requirement
- `--emake-annupload` overrides default annotation upload setting

Setting the Cluster Manager Host and Port

The `--emake-cm` option (or the `EMAKE_CM` environment variable) sets the Cluster Manager host name (either IP address or machine name) and must be specified. If the host name is not specified, Electric Make operates in local mode and performs like a traditional make program by running jobs on the local machine in serial order—without using the cluster hosts.

For example, if the Cluster Manager host name is “linuxbuilder,” you would invoke a build against the cluster by running:

```
% emake --emake-cm=linuxbuilder
```

Alternatively, use the `EMAKE_CM` environment variable:

```
% setenv EMAKE_CM linuxbuilder
% emake
```

If any value is set for `EMAKE_CM`, or for the `--emake-cm` option, builds will run on a cluster. Conversely, to force a local build (one *not* run on the cluster), leave the Cluster Manager host name undefined by typing:

```
% emake --emake-cm=
```

or, if the `EMAKE_CM` environment variable is undefined:

```
% emake
```

Setting the Electric Make Root Directory

The `--emake-root` option (or the `EMAKE_ROOT` environment variable) specifies the EFS root directory [or directories] location. All files under the Electric Make root directory are automatically mirrored on each Agent.

Electric Make uses the current directory as the default if no other root directory is specified. You must specify the correct root directory [or directories] or the build may fail because Electric Make cannot find the necessary files to complete the build or resolve dependencies.

For best results and performance, be specific when setting the Electric Make root location:

- Include all files created or modified by the build.
- Include all source files.
- Include the location where build output files will go during the build, for example, object files, linker output, and so on.
- Include other files read by the build such as third-party tools and system headers, or other files not modified if you need to. Be aware, however, that including these files can slow performance. See “Configuring Your Build” on page 2-2.

If necessary, specify more than one directory or subdirectory. Separate each location using standard `PATH` variable syntax (a colon for Solaris or Linux, a semicolon for Windows).

For example, for Solaris or Linux you might type:

```
--emake-root=/src/foo:/src/baz
```

In this example, you have streamlined the root path by excluding other `/src` subdirectories not necessary to the build.

For Windows, you might type:

```
--emake-root=C:\Build2;C:\Build4_test
```

Note: Any files used by the build, but not included under an Electric Make root directory, must be preloaded onto all hosts and identical to corresponding files on the system running Electric Make. If these files are not identical, Electric Make could find and use the wrong files for the build. This approach is appropriate for system compilers, libraries, header files, and other files that change infrequently and are used for all builds.

Generally, `EMAKE_ROOT` can be set to include any existing directory on the host build machine. Files in these directories are automatically accessible by commands running on cluster hosts. However, there are a few exceptions:

1. `EMAKE_ROOT` cannot be set to the system root directory (for example, “/” on UNIX or `C:/` on Windows). It may be tempting to try this to specify “mirror everything,” but in practice, this is not desirable because mirroring system directories such as `/var` or `/etc` on UNIX or `C:/Windows` on Windows can lead to unpredictable behavior. eMake will not allow you to specify the root directory as `EMAKE_ROOT`.
2. `/tmp` and the Windows temp directory cannot be included in the eMake root.
3. On Windows, another operating system restriction is imposed: `EMAKE_ROOT` is not a UNC path specification—it must be a drive letter specification or a path relative to a drive letter. It must also be a minimum of three characters.

Setting Electric Make Emulation for Ant Builds (all platforms)

Electric Make’s Apache Ant emulation is activated by using the command-line `--emake-emulation=ant` option directly, setting it in the `EMAKEFLAGS` environment variable, or invoking the executable under the name `ant` (for Linux or Solaris) or `ant.exe` (for Windows).

When building an ant buildfile, eMake creates its own version of the buildfile, rewriting it to talk to Java plug-ins stored in `emake-ant.jar` in your ECloud installation directory. This file is written to the same directory as the original buildfile. In addition, a context directory is created to store Ant context information from targets as they run. This directory must be under an eMake root; by default, eMake creates a temporary directory, but you can specify an explicit directory using the `EMAKE_ANT_CONTEXT` environment variable.

Each task is run by invoking Ant on the modified buildfile. By default, eMake tries to run `ant`, but you can specify this explicitly with the `EMAKE_ANT_INVOCATION` environment variable.

Note: Because `ant` is a script that invokes Java code, Windows systems may not recognize the script as executable. In this case, you may need to specify the command: `sh.exe C:/path/to/ant` and an `ANT_HOME` environment variable to provide context for the script.

If running an Ant emulation in cluster mode, the agent automatically modifies your `CLASSPATH` environment variable to find `emake-ant.jar`. If you run builds in local mode, you must adjust your `CLASSPATH` accordingly (for example: `/opt/ecloud/i686_Linux/bin/emake-ant.jar`).

Note: `emake-ant.jar` is built with JDK 1.5.0_06 and therefore is compatible with JDK 1.5 and 1.6.

The `<input>` task in Ant is used to prompt the user for input. eMake is designed for batch builds and does not have a mechanism for user interaction. If a build needs to use `<input>`, use Ant’s `PropertyFileInputHandler` to script it.

To break builds into separate pieces that can be distributed among multiple agents, eMake’s Ant emulation splits any target containing a sub-build (Ant, Antcall, or Subant) into multiple targets. For example, a target similar to

```
<target name="main">
  <echo message="Running subant"/>
  <ant antfile="subbuild.xml"/>
  <echo message="Subant complete"/>
</target>
```

will be split into “emake-main-1” (containing the first echo task), “emake-main-2” (containing the Ant task), and “main” (containing the second echo task). “emake-main-2” can queue up any number of other jobs, which must complete before the new “main” can run.

Note: For Ant versions 1.7.0 and later, references work with project-level data types only, not target-level data types.

Electric Cloud does not currently provide support for third-party task libraries such as `ant-contrib`. If you are having trouble with a task that is not part of native Ant, please contact your Electric Cloud sales representative to discuss support options.

Sample build and smoke test

To demonstrate the use of `EMAKE_ANT_INVOCATION`, refer to the following examples:

Example build.xml:

```
<project default="main" name="smoke test">
  <property name="include" value="smoketest.xml"/>
  <import file="{include}"/>
  <target name="main">
    <echo>Smoke test ${smoked}.</echo>
  </target>
</project>
```

Matching smoketest.xml:

```
<project name="smoking">
  <property name="smoked" value="succeeded"/>
</project>
```

If the smoke test does not report “Smoke test succeeded.”, the first thing to look at is your `EMAKE_ANT_INVOCATION`, `ANT_HOME`, `JAVA_HOME`, and (if this is a local build) `CLASSPATH`.

Setting Electric Make Emulation for Windows

Electric Make emulates different make variants: GNU Make (gmake), gmake in a Cygwin environment, and Microsoft NMAKE (nmake). The `--emake-emulation` (or `EMAKE_EMULATION` if using an environment variable) is set to gmake by default. If you are using nmake, override the default setting from gmake to nmake.

For example: `emake --emake-emulation=nmake`

This is a Windows-specific setting that must be set prior to invoking a build.

You can cause Electric Make to emulate GNU Make in a Cygwin environment by using `--emake-emulation=cygwin`.

In addition, if you copy the eMake binary to `gmake.exe` or `nmake.exe`, eMake automatically defaults to gmake or nmake emulation modes. See the `--emake-emulation-table` option on [page 10-3](#).

Sample Build

Invoking a Build

After all ElectricAccelerator components are installed and you are familiar with the concepts, try a test build. Using a text editor, create a makefile with the following content:

For UNIX:

```
all: aa bb cc
aa:
    @echo building aa
    @sleep 10
bb:
    @echo building bb
    @sleep 10
```

```
cc:
    @echo building cc
    @sleep 10
```

For Windows:

```
SLEEP=ping -n 10 -w 1000 localhost>NUL
all: aa bb cc
aa:
    @echo building aa
    -$(SLEEP)
bb:
    @echo building bb
    -$(SLEEP)
cc:
    @echo building cc
    -$(SLEEP)
```

Note: “ping” is used in the Windows example because Windows does not have a SLEEP utility.

If you ran this file with GNU Make, you would expect it to finish in approximately 30 seconds—allowing for each 10-second command to run serially. Running against an ElectricAccelerator cluster with at least three Agents, the commands run in parallel allowing the build to complete much faster.

Specify the Cluster Manager by using the `--emake-cm=<host>` option. The Cluster Manager is responsible for assigning Agents to Electric Make for processing jobs.

Next, the Electric Make root directory [or directories] specification must include all directories that contain source or input files required by the build. In the example, the only source file is the makefile, which is in the same directory where Electric Make is invoked. Because the default `EMAKE_ROOT` is the current directory, only the `--emake-cm` option must be specified (using linuxbuilder as the Cluster Manager host): `% emake --emake-cm=linuxbuilder`

```
Starting build: 1
building aa
building bb
building cc
Finished build: 1 Duration: 0:11(m.s)Cluster availability: 100%
```

Cluster availability: 100% indicates the cluster was fully available for the build duration. For more information on cluster sharing and the cluster availability metric, see [“What is Annotation?”](#) on page 8-1.

ElectricAccelerator Developer Edition

Overview

ElectricAccelerator Developer Edition enables the use of local agents, which allow developers to make use of extra cores on their own systems, whether or not a cluster is currently available. A local agent is a regular agent that advertises itself to eMake processes on the same machine through a named pipe, instead of talking to a Cluster Manager. You must have a separate ElectricAccelerator Developer Edition license to run local agents. The `license.xml` file must be placed in the install directory’s root `C:\ECloud` or `/opt/ecloud`, by default.

Using Local Agents

Local agents are disabled by default. To enable eMake to use local agents automatically, add the `--emake-localagents=y` option to the `emake` invocation, or add it to the `EMAKEFLAGS` environment variable.

If you want to run multiple simultaneous eMakes on your machine when using local agents, note that no load balancing occurs because the Cluster Manager does not interact with local agents. This means that, by default, the first build request gets all agents and no other build request can proceed unless you explicitly instruct the first build’s eMake to:

- limit its use of local agents (with `--emake-maxlocalagents`)
- release local agents occasionally (with `--emake-yield-localagents`) so they can be used by another eMake that is looking for local agents, or by the current eMake if there are no other such eMakes running

The eMake pragma `#pragma localagent` requires a job to be run on a local agent. If you run a makefile that expects local agents on a machine that has none, you can disable this pragma on the command line with `--emake-disable-pragma=localagent` or `--emake-disable-pragma=all`.

Local Agents eMake command line arguments:

Argument	Description
<code>--emake-localagents=y</code>	Tells eMake to use any available local agents. Without a Cluster Manager, it will use local agents only. If you specify <code>--emake-localagents=y</code> but not <code>--emake-cm</code> , eMake uses local agents only. If you specify <code>--emake-cm</code> but not <code>--emake-localagents=y</code> , eMake uses cluster agents only. Specifying both options instructs eMake to use both, if available.
<code>--emake-maxlocalagents=<N></code>	Limits the number of local agents used. N=0 uses all available agents (default 0).
<code>--emake-yield-localagents=<N, T></code>	If using more than <i>N</i> local agents, releases those over <i>N</i> every <i>T</i> seconds so they can be used by another eMake that is looking for local agents. This feature is not supported on Solaris 9 due to the absence of a crucial system library function.

Using Local Agents with Cluster Agents

If you have an ElectricAccelerator Cluster Manager, you can specify `--emake-localagents=y` *and* `--emake-cm`, (referred to as hybrid mode) to instruct eMake to use local agents and cluster agents, if available.

Monitoring Build Results

Cluster Manager keeps a record of every build that runs against the cluster.

To view all build records:

1. Open Cluster Manager by typing your Cluster Manager host name in the location bar of your browser window, for example: `http://linuxbuilder`

Enter your user name and password if one has been assigned or if not, you will be logged in as the default user. Click **OK**.
2. From the Home page, click Builds. Information is provided on each build such as which user ran the build, the time the build started, and how long the build ran. A green “check mark” in the Result column indicates the build completed without error (exit status 0).

A cogwheel icon in the Status column shows the build is running.

To view details for a specific build:

Click the build name or ID to access build details. The Build Details page displays additional build properties, including the full command line, environment, and Electric Make version.

After running several test builds, you may need to refine your build environment. Additional ElectricAccelerator configuration options are described in [“Configuration” on page 2-1](#). Command-line option definitions and environment variables are included in [“Electric Make Command-line Options and Environment Variables” on page 10-1](#).

Using Break Points

ECBreakPoint is a tool you can use to help Electric Cloud debug issues related to problematic job steps and stalled processes during the build. `ecbreakpoint` is a simple program launched by agents. During the process, `ecbreakpoint` pauses the agent’s current job step and starts communication with the Cluster Manager. Users can interact with `ecbreakpoint` through the ElectricAccelerator web interface also.

Launching ecbreakpoint

Run directly in the Makefile. The `ecbreakpoint` executable (`ecbreakpoint/ecbreakpoint.exe`) is in the folder `<ECLLOUD_DIR>/<ARCH>/bin`, and should be in your path already. Launch `ecbreakpoint` directly in your Makefile using the following:

```
all:
    ecbreakpoint
```

The build will pause so you can interact with `ecbreakpoint` from the ElectricAccelerator Build Details web page. You must have sufficient permissions to view and use break point operations. For information about permissions, see the ElectricAccelerator online help topic, “Permissions.”

For additional information about break points, see the “Using `ecbreakpoint`” section in the ElectricAccelerator online help topic, “Installed Tools/Files.”

Terminating a Build

You can terminate an in-progress build by one of three methods:

- Press **Ctrl-C** from the terminal where you invoked Electric Make, or
- Use ElectricAccelerator’s web interface, or
- Use **cmtool**—see the *cmtool Reference and User Guide* PDF file, available from within the ElectricAccelerator online help system.

Cluster Manager terminates builds that seem to be hung. If Cluster Manager does not receive a request from eMake for 60 seconds, it considers the build hung and the build will be terminated.

To stop a build from the ElectricAccelerator web interface:

1. Open the web interface by typing the Cluster Manager host name in the location bar of your browser window.
2. Click **Builds**.
3. Click **Stop Build** in the Action column on the row exhibiting your build ID and Name.

To stop a build using cmtool:

Note: This is an “advanced user” option, unless you are already familiar with using command-line tools.

1. Request a list of running builds. The syntax is:

```
% cmtool --cm <clustermanager:port> getBuilds --filter
  <field name>=<value>
```

For example, if `linuxbuilder` is the Cluster Manager host name, type:

```
% cmtool --cm linuxbuilder getBuilds --filter "result !=-1"
```

A list of running builds will display accompanied by a number of attributes for each build—for example: Build ID, machine name, build class, owner, build start time, and so on. The Build ID is used to identify a build for termination. Also you can obtain additional information and/or include comments about the build. For example, to sort builds by start time and request only the first ten builds display, enter:

```
% cmtool --cm linuxbuilder getBuilds --order Id --filter="ID<11"
```

To get failed builds:

```
% cmtool --cm linuxbuilder getBuilds --order "start_time desc"
  --filter "result !=0"
```

2. After you determine which build you need to terminate, use the syntax for ending a build:

```
% cmtool --cm <clustermanager> stopBuild <buildId>
```

For example, to end build 4458, you would type:

```
% cmtool --cm linuxbuilder stopBuild 4458
```

Shutting Down Cluster Hosts during Builds

You can remove agent machines from the cluster during a build without requiring downtime or disabling Agents in the Cluster Manager, with the following advisories:

- If possible, shut down Agents before rebooting agent machines. Not shutting down Agents before a reboot may cause commands to fail unexpectedly, which could manifest in spurious build failures.
- After rebooting, agent machines that automatically start Agents will connect to the Cluster Manager and will be assigned builds and so on.

Priority Pools

Overview

Priority Pools allow you to group resources into pools that can be prioritized differently among groups. Each pool's resources can potentially be utilized by any build, but builds originating from a pool's "owner" always have first priority to use that pool's resources. Using Priority Pools allows you to manage resource allocation for builds more efficiently.

Using Priority Pools

Follow this procedure:

1. Enable Priority Pools through either the Cluster Manager or cmtool.

Using the Cluster Manager, go to Administration > Server Settings and select Priority Pools.

Using cmtool, run this command:

```
cmtool modifyServer --resourceManagerType prioritypool
```

2. Define your pool using the Agents > Resources page. A pool resource uses this form:

`__pool_xxx`, for example, `__pool_a`

Note: When you define a pool on the Resources page, you must include "`__pool_`". When you include a pool in the `--emake-resource` option, you can omit `__pool_` from the option.

3. Go to Agent > Agent Policies and set **Agent allocation policy** to shared.
4. Define a build class that uses the resource. Using Cluster Manager, go to Builds > Build Classes > New Build Class or Edit Build Class > Resource Request field.
5. Launch eMake and include the following option:

```
--emake-resource="<pool resource>:<static resource>"
```

Information about values for `--emake-resource=`

- When adding a pool resource to the option, you can omit `__pool_` from `__pool_xxx:` and use `xxx:` only.
- A pool name (before the `:`) is not required. Not defining the pool name means the build will not use a pool resource.
- A resource name (after the `:`) is not required. Not defining the resource name means the build will attempt to use any unused resources that it is allowed to use.

Additional information

When attempting to use a pool resource as a normal resource, there is a period (default is 60 seconds) during which, before your build starts, a build from the pool resource's owner can take back the resource.

Use Case 1 - High Performance Builds

You have two pools of resources, one for high performance builds and one for low performance builds. You want to ensure that high performance builds can always use the more powerful 8-core machines and that low performance builds use the 4-core machines. You also want to allow high performance builds to use the 4-core machines when low performance builds are not running. And you want to allow low performance builds to use the 8-core machines when high performance builds are not running. You also have two special software packages, so you define a static resource for each.

Pool makeup:

- Pool a - High performance build resources, five 8-core machines, defined on the Resources page as `__pool_a`. The machines are named `h_1`, `h_2`, `h_3`, `s_1`, and `s_2`.
- Pool b - Low performance build resources, five 4-core machines, defined on the Resources page as `__pool_b`. The machines are named `h_4`, `m_1`, `m_2`, `s_3`, and `s_4`.
- Two static resources are also defined on the Resources page:
 - `s` - This resource includes these machines (which have a specific software package): `s_1`, `s_2`, `s_3`, and `s_4` (from using host mask `s*`)
 - `m` - This resource includes these machines (which have a specific software package): `m_1` and `m_2` (from using host mask `m_1, m_2`)

Launching eMake with <code>--emake-resource=</code>	means the build uses
<code>"a:"</code>	Pool "a" and any unused resources
<code>"a:s"</code>	Static resource "s" (four machines). If a low performance build is running, only machines "s_1" and "s_2" (two machines) are used.
<code>"b:m"</code>	Static resource "m" only
<code>"a:m"</code>	Static resource "m" only. If a low performance build is running, no machines are available and the build must wait.
<code>" : s "</code> or <code>" s "</code>	Static resource "s" if low and high performance builds are not running. If a low performance build is running, then only machines "s_1" and "s_2" are used. If a high performance build is running, then only machines "s_3" and "s_4" are used. If low and high performance builds are running, no machines are available and the build must wait.
<code>" : "</code> or <code>" "</code>	Any unused resources (this is considered a common build)

Use Case 2 - Multiple Departments

There are two departments and each department has its own pool of machines. They want to contribute their machines toward a common pool so each department can use the other department's machines while still ensuring that their own machines are available for their department's builds. One department owns 20 machines and the other department owns 10 machines. Each department also owns a small number of 64-bit machines. IT contributes an additional four machines that any department can use.

Pool makeup:

- Pool Depta - Department A's resources, 20 machines, defined on the Resources page as `__pool_Depta`
The machines are named a_01 through a_16 and a_17_64 through a_20_64 (these last four machines are 64-bit).
- Pool Deptb - Department B's resources, 10 machines, defined on the Resources page as `__pool_Deptb`
The machines are named b_01 through b_08 and b_09_64 and b_10_64 (these last two hosts are 64-bit).
- Static resource *64bit* - This resource includes these machines: a_17_64 through a_20_64 and b_09_64 and b_10_64 (from using host mask `*_64`).
- General build machines - IT department-supplied general build machines, four machines

Launching eMake with <code>--emake-resource=</code>	means the build uses
<code>"Deptb: "</code>	Pool "Deptb" and any unused resources, using general build machines first and then Department A's machines if that department is not running a build.
<code>"Depta: "</code>	Pool "Depta" and any unused resources, using general build machines first and then Department B's machines if that department is not running a build.
<code>"Depta:64bit "</code>	Static resource "64bit" (six machines). If Department B is running a build, only "a_17_64" through "a_20_64" (four machines) are used.
<code>" : " or " "</code>	Any unused resource, using general build machines first and then Department A's and Department B's machines if those departments are not running builds (this is considered a common build).

ElectricAccelerator provides features to organize builds in even the most complex environments. You do not need to create a build class—by default, a simple organizational structure is set up for you. But, if you have diverse product lines, or multiple product releases, Electric Cloud advises taking advantage of ElectricAccelerator build class functionality and setting up build classes.

About Build Classes

A *build class* is a flexible, user-defined classification for a designated group of builds. Using build classes is optional, but if you do not assign a build class, Cluster Manager assigns the build to a default class. Electric Cloud recommends using build classes to provide organization to the build management process.

Depending on your company requirements, you might use build classes to organize build groups by version/release, product type, development stage, or platform. You can decide how to use build classes to organize your builds into sets.

Classes have default priorities and boost values. Boost values range from -10 to +10 (default 0) where a higher boost value means builds in that class can use available agents ahead of builds with the same priority and less boost.

Tag Definitions

Each build within a class is identified by a unique string called a *tag*. The build tag definition is a template that expands when a new build starts. The tag is user-defined and generally consists of a generic build name appended with build-specific data constructed from the following variables:

- GC - a globally unique number (Global Counter)
- LC - a number unique to the build class (Local Counter; the build serial number within the class)
- BUILD_CLASS - user-defined build class name
- BUILD_CLASS_ID - system-generated number Cluster Manager uses to identify each class
- USER_NAME - name of the user who invoked Electric Make
- MACHINE_NAME - name of the machine where Electric Make was invoked
- USER_BUILD_LABEL - label specified at the Electric Make command line
for example, `--emake-build-label=my_build`
- BUILD_OS_ID - operating system ID under which the build was invoked (0 = undefined, 1 = Windows, 2 = Solaris, and 3 = Linux)
- DATE - build start date and time using variables `Y`, `y`, `m`, `d`, `H`, `M`, `S` (for example, 2005-01-18 10:14:32 is 20050118101432)
- Y - the year at build start time (YYYY)
- y - the year at build start time (YY)
- m - sequential month number at build start time (1-12)
- d - sequential day of month at build start time (1-31)
- H - hour of the day at build start time (0-23)
- M - minutes at build start time (0-59)

- s - seconds at build start time (0-60)
- a - abbreviated day of week at build start time (WED)
- A - full name day of week at build start time (Wednesday)
- b - abbreviated month name at build start time (AUG)
- B - full month name at build start time (August)
- c - build start date and time using the variables A, B, d, H, M, S, Y (for example, 2005-01-18 10:14:32 is 18/01/05 10:14:32)

Together, the build name and variables are referred to as the *tag definition*. Variable names are case-sensitive.

For example, the tag definition `%BUILD_CLASS%_%LC%_%DATE%` for a build class named `QA_BUILD` creates the following build tag:

```
QA_BUILD_1234_20060123185958
```

When assigning build class tag definitions, choose from the list of tag variables above.

Build Class Examples

Suppose your company has two major product lines: SuperSoftware and MegaSoftware. SuperSoftware runs on Windows and Solaris platforms. MegaSoftware runs on Windows only. You could begin by setting up three classes that include the product name, the platform, and the current version number for each product:

- You could name the first class `Supersoftware_Win_v.2.1`. The tag definition for this class would be:

```
%BUILD_CLASS%_%LC%_%a%_%b%_%d%_%H%_%M%_%S%
```

The result would be a series of builds each named, or *tagged*, with the product name, the platform, the version number, a serial number (unique to the class), and the date for each build, for example:

```
Supersoftware_Win_v.2.1_12345_WED_AUG_22_14_37_12
```

- The second class could be named `Supersoftware_Sol_v.1.7`. The tag definition can be the same as in our first example because it would be distinguished by the second build class name. Build tags in the second class would look like:

```
Supersoftware_Sol_v.1.7_12356_WED_AUG_23_11_14_39
```

- The third class could be named `MegaSoftware_Win_v.1.3`. For this product, the tag definition would be similar to the previous examples, but also could include the name of the user who started the build because the MegaSoftware team is spread over several different locations. For this class, the tag definition might look like:

```
%BUILD_CLASS%_%LC%_%USER_NAME%_%DATE%
```

As in the first two examples, the result would be a sequentially numbered series of builds with the product name, platform, version number, name of the user who ran the build, and the date of each build assigned through the build class:

```
MegaSoftware_Win_v.1.3_12356_JSMITH_20050411100838
```

Additional classes could be created when the development of SuperSoftware or MegaSoftware entered a new phase, such as a new platform release or a new version release. In this way, the builds for each stage of development can be segmented into logical sets facilitating a more manageable and organized workflow.

Creating and Modifying Classes

When a build is invoked in Electric Make, the Electric Make option `--emake-class=<class>` specifies the class assignment for the current build. To use the Electric Make class option, the class must match an existing class in Cluster Manager.

Create a new build class using the web interface:

1. Type-in your Cluster Manager host name in the browser window location bar.
2. Click the **Build > Build Classes** tabs. The **Build Classes** page displays.
3. Click the **New Build Class** link. A blank class details screen appears.
4. Click the **Show Help** link on the right-side of the screen to see field descriptions and fill-in field information accordingly.

The *build class name* identifies the class to Electric Make for invoking a build.

Note: Assign the build's class name through an Electric Make command-line option when the build is invoked: `--emake-class=<exact build class name>`. If no build class is assigned, Cluster Manager assigns the build to the default class. If the class name typed on the eMake command line does not match a class name already in Cluster Manager, eMake exits.

5. In the **Tag Definition** text box, enter the build class tag definition.

Note: To avoid errors, follow standard naming conventions for tag definitions by using numbers, letters, and underscores only **without** leading or trailing white space. Use a percent sign on either side of any variables used. (For example, `%DATE%`.) Use underscores (`_`) instead of spaces.

Continue filling-in the fields, accessing online help for additional information if needed.

Click **OK** when fields are filled-in.

Electric Make automatically creates makefile macros (`ECLLOUD_BUILD_CLASS`, and `ECLLOUD_BUILD_TAG`) from Cluster Manager build class data. These macros can be used to get generated values into your makefiles. For more information, see [“Using Electric Make Variables” on page 4-4](#).

You may add comments to the build class also. Click the **Build > Build Class** tabs, then select the Build Class Name where you want to add a comment.

For more information on adding or editing build class comments, adding or editing class properties, or deleting a build class, see the web interface online help.

Based on the build class, you can enable email notification for builds that complete. In the list of classes, a user can select the checkbox in the **Notify** column for each class they wish to receive email notifications about. When a build from the selected class(es) completes, an email message is sent to each user that requested notification.

Using Electric Make Variables

Electric Make automatically defines several variables that can be used in makefiles to access Cluster Manager specific values during a build. For example, you could insert the build tag into your compilation step by typing:

```
main.o:main.cpp
gcc -DBUILD_TAG="$(ECLLOUD_BUILD_TAG)" -o main.o main.cpp
```

The variables Electric Make automatically creates are described in the following table.

Variable	Definition
ECLLOUD_BUILD_CLASS	The build class as specified on the command line with <code>--emake-class</code> . For more information, see “About Build Classes” on page 4-1 .
ECLLOUD_BUILD_ID	A globally unique build ID. This value is guaranteed unique across all builds in all classes for this Cluster Manager.
ECLLOUD_BUILD_TAG	The tag as configured in Cluster Manager for this build class. For more information, see “Creating and Modifying Classes” on page 4-3 .
ECLLOUD_BUILD_COUNTER	A unique ID for this build within the current build class. This value is guaranteed to be unique across all builds in the same class, but not across builds in different classes.
ECLLOUD_BUILD_TYPE	Allows you to discover the build type: <i>local</i> for a local build; or <i>remote</i> for a cluster build.

ElectricAccelerator is designed to be completely compatible with existing Make variants it emulates. There are, however, some differences in behavior that may require changes to makefiles or scripts included in the build. This section documents those differences and what actions to take to ensure your build is compatible with ElectricAccelerator.

Invoking eMake

The Electric Make executable is called **emake**; consequently, the most important change to your build process is to ensure this executable is invoked in place of the existing Make.

For interactive command-line use, ensure that:

- the ElectricAccelerator bin directory is in your PATH environment variable:
 - For Linux: /opt/ecloud/i686_Linux/bin or /opt/ecloud/i686_Linux/64/bin
 - For Solaris: /opt/ecloud/sun4u_SunOS/bin
 - For Windows: C:\ecloud\i686_win32\bin or C:\ecloud\i686_win32\64\bin
- you type emake in place of gmake or nmake.

For recursive submake invocations in makefiles to ensure eMake is called, use the \$(MAKE) macro for specifying submakes instead of hard-coding references to the tool. For example, instead of using:

```
libs:
    make -C lib
```

use the following \$(MAKE) macro:

```
libs:
    $(MAKE) -C lib
```

Also, you can rename the eMake executable to either gmake or nmake because Electric Make checks the executable to determine which emulation to use. If the name of the submake is hard-coded in many places within your makefiles, a simple solution would be to rename gmake or nmake to *gmake.old* or *nmake.old*, and rename eMake to either gmake or nmake on cluster hosts only. In this way, you can maintain access to your existing make, but all submakes from an ElectricAccelerator build will correctly use Electric Make.

Almost all GNU Make and NMAKE options are valid for use with ElectricAccelerator. However, ElectricAccelerator does not support some GNU Make options, which include:

```
-d (debug)
-j (run parallel)
-l (load limit)
-o (old file)
-p (print database)
-q (question)
-t (touch)
```

Note: Electric Make ignores -j and -l if specified. The following options: -d, -o, -p, -q, and -t, are unsupported and will elicit an error message from eMake if specified.

NMAKE options not supported:

```
/C (suppress output)
/Q (check timestamps, don't build)
/T (update timestamps, don't build)
/NOLOGO (suppresses NMAKE copyright message)
/ERRORREPORT (sends information to Microsoft)
```

Unsupported GNU Make 3.81 Features

Electric Make does not support the following GNU Make 3.81 features:

- Though `$(eval)` is allowed in rule bodies, any variables created by the `$(eval)` exist only in the scope of the rule being processed, *not* at the global scope as in GNU Make. For example, the following is not supported:

```
all:
    $(eval foo: bar)
```

- Using `*$` for an archive member target

Single Make Invocation

It is important to keep the build in a single Make invocation. At many sites, Make is not directly invoked to do a build. Instead, a wrapper script or harness is used to invoke Make, and users (or other scripts) invoke this wrapper. The wrapper script may take its own arguments and may perform both special set up or tear down (checking out sources, setting environment variables, post-processing errors, and so on). Because Electric Make behaves almost exactly like native Make tools, usually it can directly replace the existing makefile in wrapper scripts.

Sometimes, however, the script may invoke more than one Make instance. For example, the script could iterate over project subdirectories or build different product variants. In this case, each of these builds becomes a separate ElectricAccelerator build, with its own build ID, Cluster Manager entry, history file, and so on.

It is much more efficient for Make instances that are logically part of one build to be grouped under the control of a single parent Make invocation. In this way, Electric Make can track dependencies between submakes, ensure maximal parallelization and file caching, and manage the build as a single, cohesive unit.

If your build script invokes more than one submake, consider reorganizing makefile targets so a single Make is invoked that in turn calls Make recursively for submakes.

If a lot of the setup for each instance occurs within the build script, another possible solution is to use a simple top level makefile to wrap the build script; for example,

```
all:
    my-build-harness ...
```

In this instance, `my-build-harness` runs on the Agent much like any other command and sends commands discovered by submake stubs back to the host build machine. This approach works only if each submake's output is not directly read by the script between Make invocations. Otherwise, it may be susceptible to submake stub output problems. See [“Submake Stubs” on page 5-4](#).

Commands that Read from the Console

When GNU Make or NMAKE invokes a makefile target command, that process inherits the standard I/O streams of the Make process. It is then possible to invoke commands that expect input during a build, either from the terminal or passed into the Make standard input stream. For example, a makefile such as:

```
all:
    @cat
```

could be invoked like this:

```
% echo hello | gmake
hello
```

More commonly, a command in a makefile might prompt the user for some input, particularly if the command encounters an error or warning condition:

```
all:
    @rm -i destroy

% gmake
rm: remove regular file `destroy'? y
```

Neither of these constructs is generally recommended because systems that require run-time user input are tedious to invoke and extremely difficult to automate.

Because ElectricAccelerator runs commands on cluster hosts where processes do not inherit the I/O streams and console of the parent eMake, makefiles (such as the examples above) with commands expecting interactive input are not supported.

In the majority of cases, tools that prompt for console input contain options to disable interactive prompting and proceed automatically. For example, invoking “rm” without the “i” enables this behavior. For those that do not, explicitly feeding expected input (either from a file or directly by shell redirection or piping) will suffice. For example:

```
all:
    echo y | rm -i destroy
```

Finally, tools such as Expect (*see* <http://expect.nist.gov>) that automate an interactive session can be used for commands that insist on reading from a console.

Transactional Command Output

Electric Make simulates a serial GNU Make or NMAKE build. Even though Electric Make runs many commands in parallel, the command output (including text written to standard streams and changes to the file system, such as creating or updating files) appears serially in the original order without overlapping. This feature is called *transactional output* (or “serial order execution”) and is unique to ElectricAccelerator among parallel build systems. This feature ensures standard output streams and underlying file systems always reflect a consistent build execution state, regardless of how many jobs Electric Make is actually running concurrently on the cluster.

Transactional output is achieved by buffering the results of every command until the output from all preceding commands is written. Buffering means that while the output *contents* on the standard streams matches GNU Make or NMAKE exactly, the *timing* of its appearance may be a little unexpected. For example:

- **“Bursty” output** – One of the first things you notice when running a build with ElectricAccelerator is that it appears to proceed in bursts, with many jobs finishing in quick succession followed by pauses. This type of output is normal during a highly parallel build because many later jobs may have completed and output is ready to be written as soon as longer, earlier jobs complete. The system remains busy, continuously running jobs throughout the build duration, even if the output appears to have paused momentarily.
- **Output follows job completion** – GNU Make and NMAKE print commands they are executing before they are invoked. Because Electric Make is running many commands in parallel and buffering results to ensure transactional output, command-line text appears with the output from the command *after* the command has completed. For example, the last command printed on standard output is the job that just completed, not the one currently running.
- **Batch output** – As a way to provide feedback to the user during a long-running execution, some commands may write to standard output continuously during their execution. Typically, these commands may print a series of ellipses or hash marks to indicate progress or may write status messages to standard error as they run. More commonly, a job may have several long-running commands separated with echo statements to report on progress during build execution:

For example, consider a rule that uses `rsync` to deploy output:

```
install:
    @echo "Copying output into destination"
    rsync -vr $(OUTPUT) $(DESTINATION)
    @echo "done"
```

With GNU Make, users first see the `Copying` output echo, then the state information from `rsync` as it builds the file list, copies files, and finally, they see the `done` echo as the job completes.

With Electric Make, all output from this job step appears instantaneously in one burst when the job completes. By the time any output from `echo` or `rsync` is visible, the entire job has completed.

Stubbed Submake Output

Recursive Makes (also called *submakes* because they are invoked by a parent or *top-level* Make instance) are often used to partition a build into smaller modules. While submakes simplify the build system by allowing individual components to be built as autonomous units, they can introduce new problems because they fragment the dependency tree. Because the top-level Make does not coordinate with the submake—it is just another command—it is unable to track targets across Make instance boundaries. For a discussion of submake problems, see “Recursive Make Considered Harmful” by Peter Miller (<http://miller.emu.id.au/pmiller/books/rmch/>)

Submakes are particularly problematic for parallel builds because a build composed of separate Make instances is unable to control target serialization and concurrency between them. For example, consider a build divided into two phases: a `libs` submake that creates shared libraries followed by `apps` that builds executables that link against those libraries. A typical top-level makefile that controls this type of build might look like this:

```
all: makelibs makeapps
makelibs:
    $(MAKE) -C libs
makeapps:
    $(MAKE) -C apps
```

This type of makefile works fine for a serialized make, but running in parallel it can quickly become trouble:

- If `makelibs` and `makeapps` run concurrently (as the makefile “all” rule implies), link steps in the `apps` Make instance may fail if they prematurely attempt to read `libs` generated libraries. Worse, they may link against existing, out-of-date library copies, producing incorrect output without error. This is a failure to correctly serialize dependent targets.
- Alternatively, if `apps` is forced to wait until `libs` completes, even `apps` targets that do not depend on `libs` (for example, all the compilation steps, which are likely the bulk of the build) are serialized unnecessarily. This is a failure to maximize concurrency.

Also important to note: Submakes are often spawned indirectly from a script instead of by makefile commands,

```
makelibs:
    # 'do-libs' is a script that will invoke 'make'
    do-libs
```

which can make it difficult for a Make system to identify submake invocations, let alone attempt to ensure their correct, concurrent execution.

These problems are exacerbated with distributed (cluster) parallel builds because each make invocation is running on a remote Agent.

Correct, highly concurrent parallel builds require a single, global dependency tree. Short of re-architecting a build with submakes into a single Make instance, this is very difficult to achieve with existing Make tools.

An ideal solution to parallelizing submakes has the following properties:

- maximizes concurrency, even across make instances
- serializes jobs that depend on output from other jobs
- minimizes changes to the existing system (in particular, does not require eliminating submakes or prohibit their invocation from scripts)

Submake Stubs

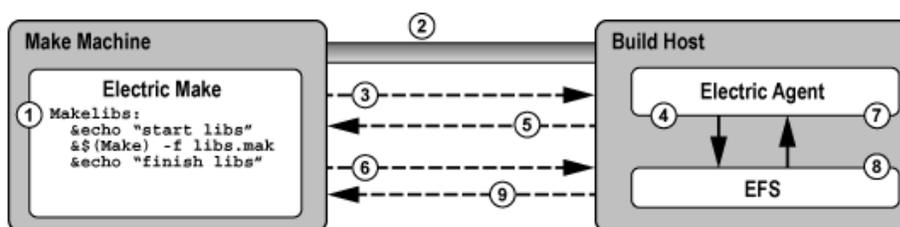
ElectricAccelerator solves the parallel submake problem by introducing submake stubs — Electric Make dispatches *all* commands, regardless of tool (compiler, packager, submake, script, and so on) to cluster Agents. After the Agent executes a command, it sends the results (output to standard streams and exit status) back to Electric Make, which then sends the next job command.

If the command run by the Agent invokes Electric Make (either directly by the expanded `$(MAKE)` variable or indirectly through a script that calls `emake`), a new top-level build is *not* started. Instead, an Electric Make process started on the Agent enters *stub* mode and it simply records details of its invocation (current working directory, command-line, environment, and so on) and immediately exits with status `0` (success) without writing output or reading any makefiles. The Agent then passes invocation details recorded by the stub back to the main Electric Make process on the host build machine, which starts a new Make instance and integrates its targets (which run in parallel on the cluster just like any other job) into the global dependency tree. Commands that follow a submake invocation are logically in a separate job serialized after the last job in the submake.

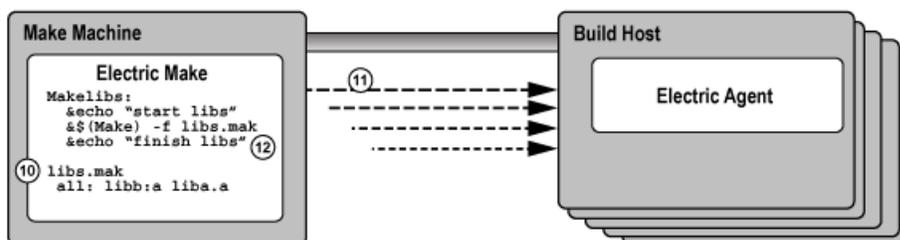
The following example illustrates this process:

```
...
Makelibs:
    @echo "start libs"
    @$(MAKE) -C libs
    @echo "finish libs"
```

This example is diagramed in steps as shown in the following illustrations.

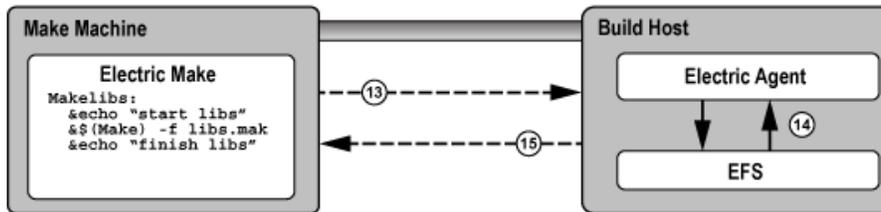


1. Electric Make determines that `Makelibs` target needs to be built.
2. Electric Make connects to Electric Agent.
3. Electric Make sends the first command, `echo "start libs"`.
4. The Agent invokes the command, `echo "start libs"`, and captures the result.
5. The Agent returns the result, `"start libs"`, to Electric Make.
6. Electric Make sends the second command, `emake -f libs.mak`
7. The Agent runs the second command, `emake -f libs.mak`
8. `emake` enters stub mode and records the current working directory, command, and environment, then exits with no output and status code `0`
9. Agent returns the stub mode result and a special message stating a new Make was invoked with recorded properties (Electric Make).



10. Electric Make starts a new Make instance, reads the makefile, `libs.mak`, and integrates the file into the dependency tree.
11. New jobs are created and run against the cluster to build all targets in the `libs.mak` makefile.
12. Electric Make splits the last command in the `Makelibs` target, `echo "finish libs"`, into a continuation job that is defined to have a serial build order later than the last job in the submake, but there is no explicit dependency created that requires it to run later than any of the jobs in the submake. This means that it may run in parallel (or

even before) the jobs in the submake, but if for some reason that is not safe, Electric Make will be able to detect a conflict and rerun the continuation job.



13. Electric Make sends the last command, `echo "finish libs"`.

14. Agent runs the command, `echo "finish libs"`, and captures the result.

15. Agent returns the result, `"finish libs"`, to Electric Make.

The Electric Make Stub solution addresses three basic parallel submake problems by:

- **Running parallel submakes in stub mode** – Stubs finish instantaneously and discover jobs as quickly as possible so the build is as concurrent as possible.
- **Creating a single logical Make instance** – New Make instances are started by the top-level Electric Make process only and their targets are integrated into the global dependency tree. Electric Make can then track dependencies across Make instances and use its conflict resolution technology to re-order jobs that may have run too soon.
- **Capturing submake invocations** – By capturing submake invocations as they occur, the submake stub works with your makefiles and builds scripts for the majority of cases “as-is.” However, the stub introduces a behavior change that may require some makefile changes. See “[Submake Stub Compatibility](#)” on page 5-6.

Submake Stub Compatibility

ElectricAccelerator submake stubs allow your existing recursive builds to behave like a single logical Make instance to ensure fast, correct parallel builds. Stubs do introduce new behavior, however, that may appear as build failures. This section describes what constructs are not supported and what must be changed to make ElectricAccelerator stubs compatible with submake stubs.

At this point, it is useful to revisit the relationship between commands and rules:

```

all:
    echo "this is a command"
    echo "another command that includes a copy" ; \
    cp aa bb
    echo "so does this command" ; \
    cp bb cc
    cp cc dd
  
```

The rule [above] contains four commands: (1) echo, (2) echo-and-copy, (3) another echo-and-copy, and (4) a copy. Note how semicolons, backslashes, and new lines delimit (or continue) commands. The rule becomes a job when Make schedules it because it is needed to build the “all” target.

The most important features of ElectricAccelerator submake stubs are:

- A submake stub never has any output and always exits successfully.
- The Agent sends stub output back (if any) after each command.
- Commands that follow a stub are invoked after the submake in the serial order.

Because a submake stub is really just a way of marking the Make invocation and does not actually do anything, *you cannot rely on its output (stdout/stderr, exit status, or file system changes) in the same command.*

In the following three examples, incompatible ElectricAccelerator commands are described and examples for fixing the incompatibility are provided.

Example 1: A command that reads submake file system changes

```

makelibs:
    $(MAKE) -C libs libcore.aa ; cp libs/libcore.aa /lib
  
```

In this example, a single command spawns a submake that builds `libcore.a` and then copies the new file into the `/lib` directory. If you run this rule as-is with ElectricAccelerator, the following error may appear:

```
cp: cannot stat 'libs/libcore.aa': No such file or directory
make: *** [all] Error 1
```

The submake stub exited immediately and the `cp` begins execution right after it in the same command. Electric Make was not notified of the new Make instance yet, so no jobs to build `libcore.a` even exist. The `cp` fails because the expected file is not present.

An easy fix for this example is to remove the semicolon and make the `cp` a separate command:

```
makelibs:
    $(MAKE) -C libs libcore.aa
    cp libs/libcore.aa /lib
```

Now the `cp` is in a command after the submake stub sends its results back to the build machine. Electric Make forces the `cp` to wait until the submake jobs have completed, thus allowing the copy to succeed because the file is present. Note that this change has no effect on other Make variants so it will not prevent building with existing tools.

Note: In general, ElectricAccelerator build failures that manifest themselves as apparent re-ordering or missing executions are usually because of commands reading the output of submake stubs. In most cases, the fix is simply to split the rule into multiple commands so the submake results are not read until after the submake completes.

Example 2: A command that reads submake *stdout*

```
makelibs:
    $(MAKE) -C libs mkininstall > installer.sh
```

The output is captured in a script that could be replayed later. Running this makefile with ElectricAccelerator always produces an empty `installer.sh` because submake stubs do not write output. When Electric Make does invoke this Make instance, the output goes to standard output, as though no redirection was specified.

Commands that read from a Make *stdout* are relatively unusual. Those that do often read from a Make that does very little actual execution either because it is invoked with `-n` or because it runs a target that writes to *stdout* only. In these cases, it is not necessary to use a submake stub. The Make instance being spawned is small and fast, and running it directly on the Agent in its entirety (instead of returning to the host build machine and distributing individual jobs back to the cluster) does not significantly impact performance.

You can specify that an individual `emake` invocation on the Agent does not enter stub mode, but instead behaves like a local (non-cluster) Make simply by setting the `EMAKE_BUILD_MODE` environment variable for that instance:

```
makelibs:
    EMMAKE_BUILD_MODE=local \
    $(MAKE) -C libs mkininstall >
    installer.sh
```

For Windows:

```
makelibs:
    set EMMAKE_BUILD_MODE=local && $(MAKE) -C libs mkininstall >
    installer
```

Electric Make automatically uses local mode when the `-n` switch is specified.

Example 3: A command that reads submake exit status

```
makelibs:
    $(MAKE) -C libs || echo "failure building libs"
```

This example is a common idiom for reporting errors. The `||` tells the shell to evaluate the second half of the expression *only* if Make exits with non-zero status. Again, because a submake stub always exits with `0`, this clause will never be invoked with ElectricAccelerator, even if it would be invoked with GNU Make. If you need this type of fail-over handling, consider post-processing the output log in the event of a build failure. *Also see* “[Annotation](#)” on page 8-1 for more information.

Another common idiom in makefiles where exit status is read in loop constructs such as:

```
all:
    for i in dir1 dir2 dir3 ; do \
        $(MAKE) -C $$i || exit 1; \
    done
```

This is a single command: a “for” loop that spawns three submakes. The `|| exit 1` is present to prevent GNU Make from continuing to start the next submake if the current one fails. Without the `exit 1` clause, the command exit status is the exit status from the last submake, regardless of whether the preceding submakes succeeded or failed, or regardless of which error handling setting (for example, `-i`, `-k`) was used in the Make. The `|| exit 1` idiom is used to force the submakes to better approximate the behavior of other Make targets, which stops the build on failure.

On first inspection, this looks like an unsupported construct for submake stubs because exit status is read from a stub. ElectricAccelerator never evaluates the `|| exit 1` because the stub always exits with status code **0**. However, because the submakes really are reintegrated as targets in the top-level Make, a failure in one of them halts the build as intended. Explained another way, ElectricAccelerator already treats a submakes loop as a series of independent targets, and the presence or absence of the GNU Make `|| exit 1` hint does not change this behavior. These constructs should be left as-is.

Using Subbuilds

The Electric Make subbuild feature is designed to help speed up component builds through intelligent build avoidance. Currently, the subbuild feature’s scope includes the following use case:

```
Makefile:
-----
.PHONY: a b
all: a b
    @echo all
a b:
    $(MAKE) -C $@

a/Makefile
-----
all: a.lib
    @echo a

a.lib:
    @touch $@

b/Makefile
-----
all: ../a/a.lib
    @echo b
```

Explanation: If something from ‘a’ changes, and you are building from ‘b’, the only way to pick up the new `a.lib` is to build from the top level directory. With subbuilds, you know b’s dependencies so you can build those dependencies directly without having to build *everything* from the top level directory.

The subbuild database must be built beforehand to make the dependency list available without having to parse any Makefiles that are not in the current directory.

The following sections describe how to use subbuilds. Refer to “[Limitations](#)” on page 5-9 for additional information about subbuild limitations.

Subbuild Database Generation

The following command runs your build as normal and in the process generates a subbuild database with the name `emake.subbuild.db`.

```
emake --emake-gen-subbuild-db=1 --emake-root=<path>
      --emake-subbuild-db=emake.subbuild.db
```

`--emake-root` is required for cluster builds. `<path>` is the Electric Make root directory setting.

`--emake-subbuild-db` is optional, and when missing, the default `emake.subbuild.db` name is used.

Run a Build Using Subbuild

```
emake --emake-subbuild-db=emake.subbuild.db
```

You must specify `--emake-subbuild-db=<file>` to run a build with subbuild information. When you specify this option, do *not* specify `--emake-gen-subbuild-db` (or it must be equal to 0), otherwise, eMake regenerates the database.

When you invoke eMake with the `--emake-subbuild-db` option, it uses the dependencies extracted from the makefile and the subbuild database to determine which build components are prerequisites of the desired current make, then rebuilds those components before proceeding as normal.

Limitations

- There is no incremental building of the database. Each time you change something in a makefile in your build, you must rebuild the database by doing a full build.
- The database is not currently optimized for size. This may result in an extremely large database for very large builds.
- Subbuilds do not provide additional gains in non-recursive make builds.
- Because of the manner in which subbuilds are currently scheduled, there is interleaving output for the “Entering directory...” and “Leaving directory...” messages.

For example: If a subbuild database was built for the following build:

```
Makefile:
-----
.PHONY: a b
all: a b
a b:
    $(MAKE) -C $@

a/Makefile
-----
all: a.lib
a.lib:
    echo aaa > $@

b/Makefile:
-----
all: ../a/a.lib
    echo b
```

When you proceed to build just ‘b’ (maybe with “`emake -C b`”) and `a/a.lib` is missing, you receive “entering directory a” after “entering directory b”, even though ‘a’ is supposed to be built before ‘b’.

```
make: Entering directory 'b'
make -C a
make[1]: Entering directory 'a'
echo aaa > a.lib
make[1]: Leaving directory 'a'
echo b
b
make: Leaving directory 'b'
```

For local builds only:

- Rules to build a sub-directory's output files must not overlap. For example: The rule to build `sub1/foo.o` must appear in `sub1/Makefile` only and not `sub2/Makefile`. Default suffix rules can cause eMake to find a way to build `sub1/foo.o` while trying to build `sub2`. In this situation, adding `".SUFFIXES:"` to `sub2/Makefile` can resolve the issue.
- Subbuilds require that the build be componentized to some degree.
- Subbuilds require that you have practiced "good hygiene" in your build tree—there must be explicit dependencies mentioned in the component makefiles.

For example: If a build has two components, `foo` and `bar`, where `foo` produces a library `foo.dll` and `bar` uses that library, the rule might be written to produce `bar.exe` such as this in `bar/Makefile`:

```
bar.exe: $(BAR_OBJS)
    link $(BAR_OBJS) -l $(OUTDIR)/foo/foo.dll
```

For subbuilds to work (in local mode), it must be modified as in the following:

```
bar.exe: $(BAR_OBJS) $(OUTDIR)/foo/foo.dll
    link $(BAR_OBJS) -l $(OUTDIR)/foo/foo.dll
```

Note that it is explicitly stated that `bar.exe` requires `foo.dll`. Also note that it is NOT required to have a rule to build `foo.dll` in `bar/Makefile`.

There cannot be ANY rule at all to build `$(OUTDIR)/foo/foo.dll` in `bar/Makefile`, explicit or implicit, otherwise you will get the wrong information for building `foo/foo.dll` in the subbuilds database. The subbuilds database currently allows updates to existing entries while building the database.

Hidden Targets

Electric Make differs from other Make variants in the way it searches for files needed by pattern rules (also called suffix or implicit rules) in a build.

- At the beginning of each Make instance, Electric Make searches for matching files for all pattern rules *before it runs any commands*. After eMake has rules for every target that needs updating, it schedules the rules [creating jobs] and then runs those jobs in parallel for maximum concurrency.
- Microsoft NMAKE and GNU Make match pattern rules *as they run commands*, interleaving execution and pattern search.

Because of the difference in the way Electric Make and NMAKE match pattern rules, NMAKE and Electric Make can produce different makefile output with *hidden targets*. A hidden target (also known as a "hidden dependency") is a file that is:

- created as an undeclared side-effect of updating another target
- required by a pattern to build a rule

Consider the following makefile example:

```
all: bar.lib foo.obj

bar.lib:
    touch bar.lib foo.c

.c.obj:
    touch $@
```

Notice that `foo.c` is created as a side-effect of updating the `bar.lib` target. Until `bar.lib` is updated, no rule is available to update `foo.obj` because nothing matches the `.c.obj` suffix rule.

NMAKE accepts this construct because it checks for `foo.c` existence before it attempts to update `foo.obj`. NMAKE produces the following result for this makefile:

```
touch bar.lib foo.c
touch foo.obj
```

Electric Make, however, performs the search for files that match the suffix rule once so it can schedule all jobs immediately and maximize concurrency. Electric Make will not *notice* the existence of `foo.c` by the time it attempts to update `foo.obj`, even if `foo.c` was created. Electric Make fails with:

```
NMAKE : fatal error U1073: don't know how to make 'foo.obj'
Stop.
```

The fix is simply to identify `foo.c` as a product for updating the `bar.lib` target, so it is no longer a hidden target. For the example above, adding a line such as `foo.c: bar.lib` is sufficient for Electric Make to understand that `.c.obj` suffix rule matches the `foo.obj` target if `bar.lib` is built first. Adding this line is more accurate and has no effect on NMAKE.

GNU Make is similarly incompatible with Electric Make, but the incompatibility is sometimes masked by the GNU Make directory cache. GNU Make attempts to cache the directory contents on first access to improve performance. Unfortunately, because the time of first directory access can vary widely depending on which targets reference the directory and when they execute, GNU Make can appear to fail or succeed randomly in the presence of hidden targets.

For example, in this makefile, the file `$(DIR)/foo.yy` is a hidden target created as a side-effect of updating `aa` and needed by the pattern rule for `foo.xx`:

```
all: aa bb
aa:
    touch $(DIR)/foo.yy
bb: foo.xx

%.xx: $(DIR)/%.yy

@echo $@
```

Depending on the value of `DIR`, this build may or may not work with GNU Make:

```
% mkdir sub; gmake DIR=sub
touch sub/foo.yy
foo.xx

% gmake DIR=.
touch ./foo.yy
gmake: *** No rule to make target 'foo.xx', needed by 'bb'. Stop.
```

Electric Make does not attempt to emulate this behavior. Instead, it consistently refuses to schedule `foo.xx` because it depends on a hidden target (just as it did in the NMAKE emulation mode in the earlier example). In this case, adding a single line declaring the target: `$(DIR)/foo.yy: aa` is sufficient to ensure it always matches the `%.xx` pattern rule.

Note: If a build completes successfully with Microsoft NMAKE or GNU Make, but fails with “don't know how to make `<x>`” with Electric Make, look for rules that create `<x>` as a side-effect of updating another target. If `<x>` is required by a suffix rule also, it is a hidden target and needs to be declared as explicit output to be compatible with Electric Make.

There are many other reasons why hidden targets are problematic for all Make-based systems and why eliminating them is good practice in general. For more information, *see*:

- “Paul’s Rules of Makefiles” by Paul Smith at <http://www.make.paulandlesley.org/rules.html>. Among other useful guidelines for writing makefiles, the primary author of GNU Make writes, “Every non-`.PHONY` rule **must** update a file with the *exact* name of its target. [...] That way you and GNU Make always agree.”
- “The Trouble with Hidden Targets” by John Graham-Cumming at <http://www.cmcrossroads.com/content/view/full/6519/120/>.

Note: In a limited number of cases, eMake may conclude that a matching pattern rule for an output target does not exist. This occurs because eMake's strict string equality matching for prerequisites determines that the prerequisites are different (even though the paths refer to the same file) and that there is no rule to build it.

Wildcard Sort Order

A number of differences exist between GNU Make and Electric Make regarding the use of `$(wildcard)` and prerequisite wildcard sort order functions. When using the `$(wildcard)` function or using a wildcard in the rule prerequisite list, the resultant wildcard sort order may be different for GNU Make and Electric Make.

Different GNU Make versions are not consistent and exhibit permuted file lists. Even a GNU Make version using different system libraries versions will exhibit inconsistencies in the wildcard sort order.

No difference exists in the file list returned, other than the order. If the sort order is important, you may wrap `$(wildcard)` with `$(sort)`.

For example:

```
$(sort $(wildcard *.foo))
```

Do not rely on the order of rule prerequisites generated with a wildcard. For example, using `target: *.foo`.

Relying on the order of `*.foo` can be dangerous for both GNU Make and Electric Make. Neither GNU Make nor Electric Make guarantees the order in which those prerequisites are executed.

Delayed Existence Checks

All Make variants process makefiles by looking for rules to build targets in the dependency tree. If no target rule is present, Make looks for a file on disk with the same name as the target. If this *existence check* fails, Make notes it has no rule to build the target.

Electric Make also exhibits this behavior, but for performance reasons it delays the existence check for a target without a makefile rule until just before that target is needed. The effect of this optimization is that Electric Make may run more commands to update targets [than GNU Make or NMAKE] *before* it discovers it has no rule to make a target.

For example, consider the following makefile:

```
all: nonexistent aa bb
aa:
    @echo $@
bb:
    @echo $@
```

GNU Make begins by looking for a rule for `nonexistent` and, when it does not find the rule, it does a file existence check. When that fails, GNU Make terminates immediately with:

```
make: *** No rule to make target 'nonexistent', needed by 'all'. Stop.
```

Similarly, NMAKE fails with:

```
NMAKE : fatal error U1073: don't know how to make 'nonexistent' Stop.
```

Electric Make delays the existence check for `nonexistent` until it is ready to run the `all` target. First, Electric Make finishes running commands to update the `aa` and `bb` prerequisites. Electric Make fails in the same way, but executes more targets first:

```
aa
bb
make: *** No rule to make target 'nonexistent', needed by 'all'. Stop.
```

Of course, when the existence check succeeds [as it does in any successful build], there is no behavioral difference between Electric Make and GNU Make or Microsoft NMAKE.

Enabling Multiple Remakes (GNU Make Only)

GNU Make has an advanced feature called Makefile Remaking, which is documented in the GNU Manual, “How Makefiles are Remade,” and available at: <http://www.gnu.org/software/make/manual/make.html#Remaking-Makefiles>

To quote from the GNU Make description:

“Sometimes makefiles can be remade from other files, such as RCS or SCCS files. If a makefile can be remade from other files, you probably want make to get an up-to-date version of the makefile to read in.

“To this end, after reading in all makefiles, make will consider each as a goal target and attempt to update it. If a makefile has a rule which says how to update it (found either in that very makefile or in another one) or if an implicit rule applies to it (see section Using Implicit Rules), it will be updated if necessary. After all makefiles have been checked, if any have actually been changed, make starts with a clean slate and reads all the makefiles over again. (It will also attempt to update each of them over again, but normally this will not change them again, since they are already up to date.)”

This feature can be very useful for writing makefiles that automatically generate and read dependency information with each build. However, this feature can cause GNU Make to loop infinitely if the rule to generate a makefile is always out-of-date:

```
all:
    @echo $@
makefile: force
    @echo "# last updated: 'date'" >> $@
force:
```

In practice, a well-written makefile will not have out-of-date rules that cause it to regenerate. The same problem, however, can occur when Make detects a clock skew—most commonly due to clock drift between the system running Make and the file server hosting the current directory. In this case, Make continues to loop until the rule to rebuild the makefile is no longer out-of-date.

In the example below, DIR1 and DIR2 are both part of the source tree:

```
-include $(DIR1)/foo.dd
all:
    @echo $@
$(DIR1)/foo.dd: $(DIR2)/bar.dd
%.d:
    touch $@
```

If two directories are served by different file servers and the clock on the system hosting DIR2 is slightly faster than DIR1, then even though `foo.dd` is updated after `bar.dd`, it may appear to be older. On remaking, GNU Make will again see `foo.dd` as out-of-date and restart, continuing until the drift is unnoticeable.

This problem is particularly troublesome on Solaris, where GNU Make timestamp checking has nanosecond resolution:

```
% make
touch dir2/bar.dd
touch dir1/foo.dd
gmake: *** Warning: File 'bar.dd' has modification time in the future (2005-04-11
13:52:46.811724 > 2005-04-11 13:52:46.799573198)
touch dir1/foo.dd
touch dir1/foo.dd
all
```

Electric Make fully supports makefile remaking and can be configured to behave exactly as GNU Make. However, by default, to ensure builds do not loop unnecessarily while remaking, Electric Make limits the number of times it restarts a make instance to 10. If your build is looping unnecessarily, you may want to lower this value or disable remaking entirely by setting:

```
--emake-remake-limit=0
```

NMAKE Inline File Locations (Windows Only)

NMAKE contains a feature to create *inline files* with temporary file names. For example, the following makefile creates a temporary inline file containing the word “pass” and then uses “type” to output it.

```
all:
    type <<
        pass
    <<
```

With Microsoft NMAKE, the file is created in the directory where %TMP% points. Electric Make does not respect the %TMP% setting and creates the inline file in the rule working directory that needs the file.

How eMake Processes MAKEFLAGS

eMake uses the following process:

1. Similar to GNU Make, eMake condenses no-value options into one block.
2. When eMake encounters an option with a value, it does what GNU Make does, it appends the value and starts the next option with its own -
3. Certain options are ignored/not created. This changes the layout of the options in MAKEFLAGS (for example -j, -l).
4. eMake-specific options are not added to MAKEFLAGS, but are handled through EMAKEFLAGS.
5. Passing down environment variables as TEST=test renders the same result as in GNU Make (an extra - at the end, followed by the variable=value).
6. On Windows, eMake prepends the --unix or --win32 flag explicitly.

This chapter provides information about how ElectricAccelerator integrates with the following environments:

- Cygwin
- Rational ClearCase
- Microsoft Visual Studio
- Eclipse

Using Cygwin with ElectricAccelerator (Windows Only)

Cygwin is a Linux-like environment for Windows that consists of two parts:

- A DLL (`cygwin1.dll`) that acts as a Linux API emulation layer, providing substantial Linux API functionality.
- A collection of tools that provide a Linux look and feel.

If your builds used `gmake` in a Cygwin environment, you might need to use Electric Make's `--emake-emulation=cygwin` option.

For more information about other Cygwin-specific Electric Make command-line options and corresponding environment variables, see ["Windows-specific Commands"](#) on page 10-6. Specifically, the following command-line options:

```
--emake-cygwin=<Y/N/A>
and
--emake-ignore-cygwin-mounts=<mounts>
```

Using ClearCase with ElectricAccelerator

If your build environment relies on the ClearCase source control system, there are some special considerations for running eMake. ClearCase views can be either "snapshot" or "dynamic."

- ClearCase *snapshot views* behave like a normal file system, so no special support is required.
- ClearCase *dynamic views* have non-standard file system behavior that requires explicit handling by eMake.

Note: ElectricAccelerator does not currently support ClearCase integration on Solaris x86. If you have need of this support, please contact your Electric Cloud sales representative. You can make use of `ecclearcase_fake.so` to provide information about your ClearCase setup through an `ini` file. Refer to "eMake's "fake" interface for ClearCase" on page 6-2.

Configuring ElectricAccelerator for ClearCase

LD_LIBRARY_PATH

Using ElectricAccelerator in a ClearCase environment requires your `LD_LIBRARY_PATH` (on UNIX) or `PATH` (on Windows) to contain a directory that includes libraries required to run "cleartool." Library filenames for Windows or UNIX begin with "libatria" (Windows - `libatria.dll`, UNIX - `libatria.so`).

If you plan to use ClearCase with eMake, you must add the ClearCase shared libraries to the `LD_LIBRARY_PATH` on your system.

For sh:

```
LD_LIBRARY_PATH=/usr/atria/linux_x86/shlib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

For csh:

```
setenv LD_LIBRARY_PATH /usr/atria/linux_x86/shlib:${LD_LIBRARY_PATH}
```

(`/usr/atria/linux_x86/shlib` is an example and may differ on your system depending on what OS you use and where ClearCase is installed.)

To ensure ElectricAccelerator knows where ClearCase is installed, edit `/etc/ld.so.conf` to include the ClearCase installation location. As a second option, you can include the ClearCase installation location in `LD_LIBRARY_PATH`.

ClearCase views on agents

When ElectricAccelerator replicates a ClearCase view on an agent, it appears as a generic file system—ClearCase commands that run as part of a build will not work on the host, even if ClearCase is installed on that machine. The Electric File System masks ClearCase’s VOB mounts. If your build runs ClearCase commands, these commands must be *runlocal* steps. For more information, see “Running a Local Job on the Make Machine” on page 9-2.

Note: Because of the potential adverse interaction between two different filesystems (ClearCase and ElectricAccelerator), Electric Cloud recommends that you do *not* install ClearCase on ElectricAccelerator Agent machines.

If you must run ClearCase on an ElectricAccelerator Agent machine, ensure that whichever one you need to start and stop frequently is configured to start “second” at system startup time.

--emake-clearcase

The Electric Make command-line option `--emake-clearcase` controls which ClearCase features are supported for a build. By default, ClearCase integration for `rofs`, `symlink`, and `vobs` is disabled. To turn on support for these specific ClearCase features, if your build relies on these options, use `--emake-clearcase=LIST`, where `LIST` is a comma-separated list of one or more of the following values:

- **rofs** : detect read-only file systems
eMake queries ClearCase for each file it accesses to determine whether the file should be considered 'read-only'.
- **symlink** : detect symbolic links (Windows only)
eMake queries ClearCase for each file it accesses to determine whether the file is a ClearCase symbolic link.
- **vobs** : configure separate temporary directories for each vob
Normally, eMake uses the `--emake-tmpdir` setting to determine where to place temporary directories for each device. With the 'vobs' option enabled, eMake automatically configures one directory per VOB. On Windows, eMake also communicates with ClearCase to determine which VOB a file belongs to so it can select the correct temporary directory.

Note: If `--emake-clearcase` is not specified on the command line and the environment variable `EMAKE_CLEARCASE` is present, Electric Make takes the options from the environment.

eMake’s “fake” interface for ClearCase

In addition to a direct interface to ClearCase, eMake also provides a “fake” interface that allows the end user to pass information manually to eMake about the ClearCase environment. Normally, you invoke ClearCase functionality by specifying `--emake-clearcase=LIST` to eMake, at which point eMake attempts to load `ecclearcase6.so` and `ecclearcase7.so` (.dll on Windows). Whichever library successfully initializes in the ClearCase environment is used to talk to ClearCase through a provided API that is no longer maintained or supported. You can specify the precise library to load by setting the environment variable `EMAKE_CLEARCASE_LIBRARY` to the path to the desired library.

Under some conditions, the ClearCase API does not function properly. For this circumstance, eMake provides `ecclearcase_fake.so` (.dll on Windows). If you point `EMAKE_CLEARCASE_LIBRARY` to the fake interface, eMake

loads that instead. The fake interface then loads the file specified in the environment by `ECCLEARCASE_FAKE_INI`, defaulting to `ecclearcase_fake.ini`. The ini file has two sections: `[vobs]` and `[attrs]`.

The `[vobs]` section maps a VOB path to a comma-separated set of attributes. Currently, `public` should be present for a public VOB and `ro` for read-only.

The `[attrs]` section maps a filename to `symlink*type`, where `symlink` may be empty if the file is not a symbolic link and `type` can be `null`, `version`, `directory_version`, `symbolic_link`, `view_private`, `view_derived`, `derived_object`, `checked_out_file`, `checked_out_dir`. If `symlink` is not empty, `symbolic_link` is assumed. If `type` is `version` or `directory_version` and `--emake-clearcase=rofs` is active, the EFS returns `EROFS` (or `STATUS_ACCESS_DENIED` on Windows) when an attempt is made to write the file.

If `CLEARCASE_ROOT` is set in the environment (as by `cleartool setview`), all `[attrs]` entries are tracked under their exact path as well as one with the `CLEARCASE_ROOT` prepended. If `CLEARCASE_ROOT` is set to `/view/testview`, setting `/vobs/test/symlink2` in `[attrs]` is the same as setting both `/vobs/test/symlink2` and `/view/testview/vobs/test/symlink2`.

Sample ini files for UNIX:

```
[vobs]
/vobs/test=public
/vobs/readonly=public,ro

[attrs]
/vobs/test/symlink2=symlink
/vobs/test/symlink/alpha=*directory_version
/vobs/test/symlink/beta=alpha
```

and Windows:

```
[vobs]
\test=public
\readonly=public,ro

[attrs]
S:/test/symlink2=symlink
S:/test/symlink/alpha=*directory_version
S:/test/symlink/beta=alpha
```

Where ClearCase dynamic views affect eMake behavior

Read-only mounts

ClearCase can mount files in a read-only mode, which means they appear to be writable, but any attempts to modify these files fail with a “read-only file system” (UNIX) or “access denied” (Windows) error message. Because eMake cannot tell whether a file is modifiable using normal file system interfaces, it does not know to disallow modifications performed by commands running on the agents. This activity leads to failures when eMake attempts to commit changes (incorrectly) allowed on the agent.

A simple test case:

```
unix% cleartool ls
Makefile
clock@/main/2   Rule: /main/LATEST

unix% cat Makefile
all:
mv clock clock.old
```

The file “clock” is checked in to ClearCase. Makefile attempts to rename it. If you just run “make”, it fails immediately, but can be instructed to ignore the error:

```
unix% make -i
mv clock clock.old
mv: cannot move `clock' to `clock.old': Read-only file system
make: [all] Error 1 (ignored)
```

Note that this filesystem is *not* mounted *read-only*, so a Makefile can be created. Because “clock” is checked-in, it cannot be renamed without checking it out first, and ROFS is the error ClearCase gives.

Now try this with eMake:

```

unix% emake --emake-root=/vobs -i
Starting build: 114626
mv clock clock.old
ERROR EC1124: Unable to rename file
/vobs/test/drivel/clock to /vobs/test/drivel/clock.old: Read-only
file system (error code 0x1e): Read-only file system
Interrupted build: 114626   Duration: 0:00 (m:s) Cluster
availability: 100%

```

Without activating ClearCase support, eMake does not know “clock” cannot be moved, so the operation succeeds on the agent, then fails when eMake attempts to commit it to disk. Specifying the “-i” flag to ignore errors will not work here.

```

unix% /home/user/Projects/4.2/i686_Linux/ecloud/emake
/emake --emake-clearcase=rofs --emake-root=/vobs -i
Starting build: 114630
mv clock clock.old
mv: cannot move `clock' to `clock.old': Read-only file system
make: [all] Error 1 (ignored)
Finished build: 114630   Duration: 0:00 (m:s)
Cluster availability: 100%

```

When eMake knows to replicate ClearCase’s behavior, the error occurs on the host and can be handled normally.

Multiple VOBs

eMake writes uncommitted files into temporary directories, and moves them into their correct location after resolving any conflicts. eMake automatically places a temporary directory in the current working directory where it is invoked, and also creates a temporary directory in each location specified by the `--emake-tmpdir` option or the `EMAKE_TMPDIR` environment variable. When possible, eMake writes uncommitted files to the same physical device where the file will be saved when it is committed, which makes the commit operation a lightweight “rename” instead of a heavyweight “copy” operation.

Under ClearCase, each VOB functions as a separate physical disk, so to achieve optimal performance, a temporary directory must be specified for each VOB where the build writes files. `--emake-clearcase=vobs` sets up this directory for you automatically.

- On UNIX, each VOB has a distinct physical device ID, and this option is nothing more than a “shorthand” for specifying `EMAKE_TMPDIR=/vobs/foo:/vobs/bar:...` in the environment.
- On Windows, you must interface with ClearCase directly to make this distinction, so using `--emake-clearcase=vobs` is important to get the most speed for a build that writes to multiple VOBs.

Symbolic links

On Windows, ClearCase conceals the nature of its symbolic links from other programs, so what is actually a single file appears to be two different files to other programs. This situation creates an issue for eMake’s versioning mechanism as it tracks two separate chains of revisions for one underlying entity; a job’s view of the file can get out of sync and lead to build failures. `--emake-clearcase=symlinks` interfaces directly with ClearCase to determine whether a particular ClearCase file is a symbolic link and represents it on the agent as a reparse point, which is the native Windows equivalent of a symbolic link. All file operations are redirected to the target of the symbolic link, avoiding synchronization problems.

This issue does not occur on UNIX platforms because ClearCase uses native file system support for symbolic links.

A simple test case:

Beginning with a directory, “alpha”, and a symlink to that directory, “beta”:

```

windows% cleartool ls -d alpha beta
alpha@@/main/1   Rule: \main\LATEST
beta --> alpha

```

And a makefile:

```
all:
    @echo "Furshlugginer" > alpha/foo
    @echo "Potrzebie" > beta/foo
    @cat alpha/foo
windows% emake --emake-root=. -f symlink.mk
Starting build: 50070 Furshlugginer
Finished build: 50070 Duration: 0:02 (m:s) Cluster availability: 100%
windows% emake --emake-root=. -f symlink.mk --emake-clearcase=symlink
Starting build: 50071 Potrzebie
Finished build: 50071 Duration: 0:01 (m:s) Cluster availability: 100%
```

Explanation: ClearCase has no way to tell the Windows file system that the symlink is a symlink, so alpha/foo and beta/foo appear to be distinct files. (On UNIX, this is not an issue because symlinks are a standard operating system feature, which means ClearCase can show them as such.) If a build does not contain any ClearCase symbolic links, there is no reason to turn on the integration; if it does, though, you can run into the problem where eMake assumes there are two different files when there is actually just one underlying file, in which case you need to turn on the “symlink” part of our ClearCase integration.

Performance Considerations

Running builds from ClearCase dynamic views can impose a considerable performance cost depending on the ClearCase configuration and your build. The best performance is achieved by using ClearCase snapshot views. If using snapshots is not possible, there are a few things to consider when setting up an eMake build.

Enabling the “symlink” or “rofs” options incurs a performance cost because of the need to communicate with the ClearCase server when accessing a file for the first time. Many builds do not need these features, even if they are running inside a ClearCase dynamic view, so consider leaving them turned off unless you encounter unexpected build failures.

Enabling the “vobs” option should have minimal performance cost, and may significantly speed up your build if build output is written back to your dynamic view.

Because of improved caching, eMake may provide a significant performance boost beyond that provided by running build steps in parallel. eMake caches much of the file system state, reducing the total number of requests to the ClearCase server during the build. Depending on how heavily loaded your ClearCase server is, this can significantly improve build performance. If you notice build speedups higher than you would expect given the number of agents in your cluster, improved caching may be the reason.

Using the “fake” interface for ClearCase (“eMake’s “fake” interface for ClearCase” on page 6-2), which lets you specify the details of VOBs and files in a static file is much faster than communicating with ClearCase. This may suffice for many users.

Using Visual Studio with ElectricAccelerator

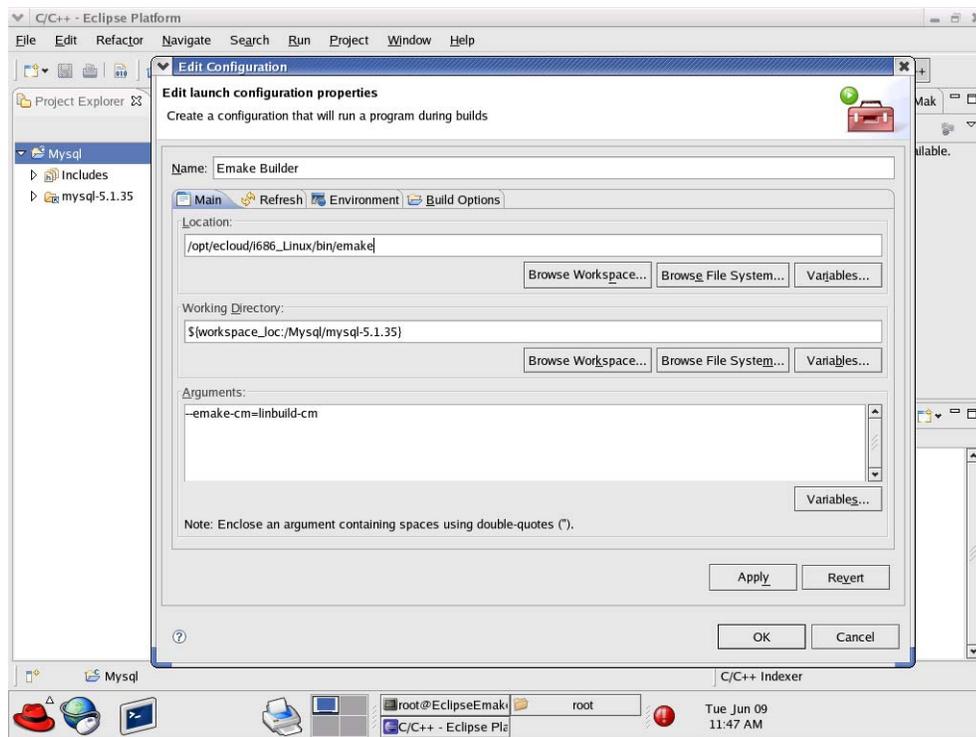
The *ElectricAccelerator Solution Support Add-in Technical Notes* contains additional information about using ElectricAccelerator with Visual Studio.

Eclipse

To configure Eclipse to run eMake, follow this procedure:

1. Open your C++ project.
2. Go to the project's Properties > Builders and click **New**.
3. Select **Program** and click **OK**.
4. Fill-in the following information for the new builder under the Main tab:
 - Name
 - Location (the full path to emake, which is OS dependent)
 - Working Directory
 - Arguments (arguments are specific to your configuration)

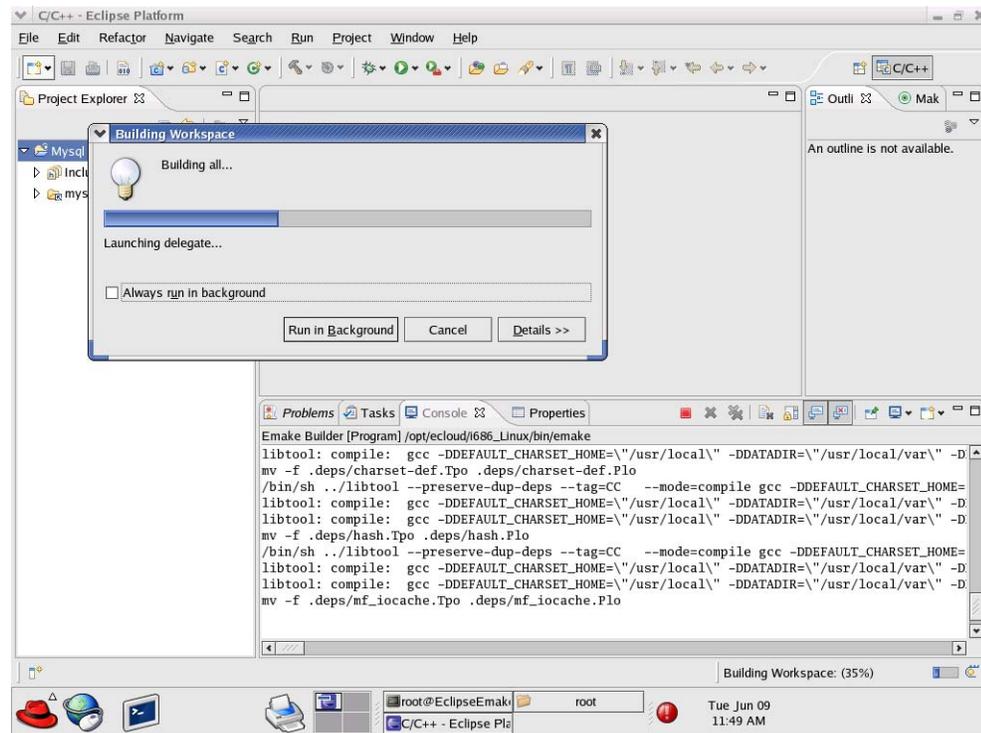
The following screenshot illustrates the Edit Configuration dialog.



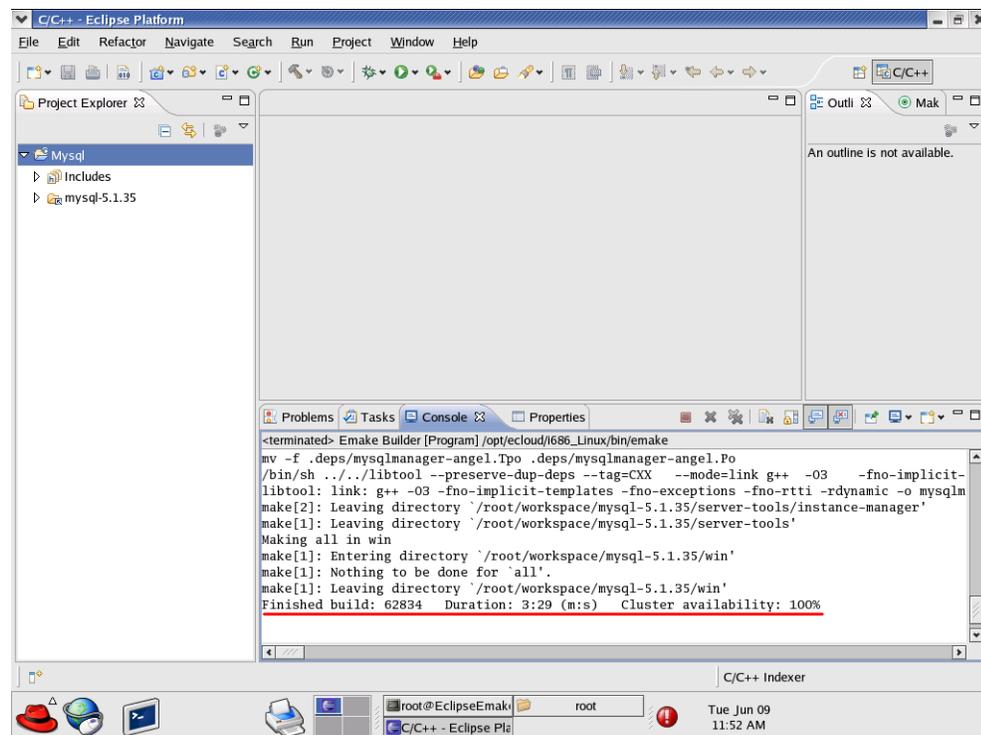
5. Click the Build Options tab. Enable Run the builder for the following *only*:
 - After a “Clean”
 - During manual builds
 - During auto builds
6. Click **OK**. Your new builder is displayed in the Builders pane.
7. Create another builder for “cleans” only. On its Main tab, ensure clean is included for Arguments. On its Build Options tab, enable Run the builder for the following *only*:
 - During a “Clean”
8. Click **OK**. Your second builder is displayed in the Builders pane.
9. Deselect CDT Builder in the Builders pane and then click **OK**.

Now you can build your project. Click **Project > Build all**.

The following screenshot illustrates a build in progress.



The following screenshot illustrates a successfully completed build.



This chapter discusses ElectricAccelerator eDepend, the Ledger, and how ElectricAccelerator handles history data files.

ElectricAccelerator eDepend

In its default configuration, ElectricAccelerator is designed to be a drop-in replacement for your existing Make tool—GNU Make or Microsoft NMAKE. ElectricAccelerator behaves exactly like your existing Make: it will rebuild (or declare up-to-date) the same targets following the same rules Make uses. The file dependency tracking technology in the Electric File System (EFS) and the Electric Make history feature is used to ensure the system reproduces exactly the same results in a parallel, distributed build as it would serially.

Because the system captures and records such detailed information about the relationships between build steps, it is uniquely capable of accomplishing much more than simply ensuring parallel builds are serially correct. In particular, by enabling ElectricAccelerator eDepend, you can wholly replace tools and techniques like **makedepend** or **'gcc -M'**—commonly used to generate makefiles too difficult to maintain by hand (for example, C-file header dependencies).

ElectricAccelerator eDepend is easier to configure, faster, more accurate, and applicable to a much wider range of dependencies than existing dependency generation solutions. If your current build does not employ dependency generation, you can enable eDepend and benefit from more accurate incremental builds without the overhead of configuring and integrating an external dependency generation tool.

The next few sections describe the dependency generation challenge in more detail and how eDepend can improve your build speed and accuracy.

Dependency Generation

Consider a build tree that looks like this:

```
src/
  Makefile          <--- top-level makefile: recurses into mylib and then into main to build the program
  common/
    header1.h
    header2.h
  mylib/
    Makefile        <--- has rules to build mylib.o and create library mylib.a
    mylib.h
    mylib.c         <--- includes common/header1.h and mylib.h
  main/
    Makefile        <--- has rules to build main.o and then to link main using main.o and mylib.a
    main.c          <--- includes common/header1.h, common/header2.h, and lib/mylib.h
```

The Problem

Even in this simple example, the need for dependency generation¹ is apparent: if you make a change to a header file, how do you ensure dependent objects are recompiled when you rebuild?

Makefiles could explicitly declare all header dependencies, but that quickly becomes too cumbersome: each change to an individual source file may or may not require an adjustment in the makefile. Worse, conditionally compiled code can create so many permutations that the problem becomes intractable.

Another possibility is to declare all headers as dependencies unilaterally, but then the build system becomes very inefficient: after a successful build, a modification to `header2.h` should trigger a rebuild only of the `main` module, not `mylib.a` as well.

Clearly, to get accurate, efficient builds, the system must have calculated dependencies automatically *before* it builds.

There are several ways to generate dependencies and update makefiles to reflect these dependencies² (for example, **makedepend** and **'gcc -M'**), but they all have the drawbacks mentioned previously.

eDepend Benefits

ElectricAccelerator eDepend is an Electric Make feature that directly addresses all problems with existing dependency generation solutions. Specifically:

- It is part of Electric Make and requires no external tool to configure, no extra processing time, and it is faster than other solutions.
- It is easily enabled by setting a command-line parameter to Electric Make. No tools or changes to makefiles are required.
- Like ElectricAccelerator itself, it is completely tool and language independent. eDepend automatically records any and all kinds of dependencies, including implicit relationships such as executables on libraries during a link step.
- eDepend dependencies are recorded in Electric Make history files—transparently recorded and used without manifesting as makefile rules.
- eDepend is accurate because it uses file information discovered by the Electric File System at the kernel level as a job executes its commands.

Enabling eDepend

1. Start from a clean (unbuilt) source tree.
2. If your build system already has a dependency generation mechanism, turn it off if possible. If you cannot turn it off, you will still get eDepend's additional accuracy, but you will not be able to improve the performance or shortcomings of your existing system.
3. Build your whole source tree with eDepend enabled.

Use the `--emake-autodepend=1` command-line switch:

```
% emake --emake-autodepend=1 ...
```

Alternatively, insert `--emake-autodepend=1` into the `EMAKEFLAGS` environment variable.

```
% setenv EMAKEFLAGS --emake-autodepend=1
% emake ...
```

4. Make a change to a file consumed by the build, but not listed explicitly in the makefiles.

For example, touch a header file: `% touch someheader.h`

5. Rebuild, again making sure eDepend is enabled.

```
% emake --emake-autodepend=1 ...
```

Notice that without invoking a dependency generator, Electric Make detected the changed header and rebuilt accordingly.

-
1. For details see <http://make.paulandlesley.org/autodep.html>.
 2. See “How Makefiles are Remade” in the GNU Make manual.

How Does eDepend Work?

Internally, eDepend is a simple application of sophisticated file usage information returned by the Electric File System.

1. As a job runs, the Electric File System records all filenames it accesses inside `EMAKE_ROOT`.

This function has two very important implications:

- eDepend can track dependencies within `EMAKE_ROOT` only.
- eDepend can track dependencies for a job only after it has run—this is why you must start with a complete build rather than an incremental build.

2. After a job completes, Electric Make saves the following eDepend information to the eMake history file:

- the working directory for the Make instance
- the target name
- any files actually read (not just checked for existence) and/or explicitly listed as prerequisites in the makefile

Note: eDepend information is stored in the history file along with serialization history information. Commands operating on the history file (for example, those specifying file location or that erase it) apply to eDepend information as well.

3. In a subsequent build, whenever Electric Make schedules a target in a directory for which it has eDepend information, it evaluates file dependencies recorded in the earlier run as it checks to see if the target is up-to-date.

In the example above, the rule to update `mylib.o` may look like this:

```
mylib.o: mylib.c
$(CC) ...
```

`mylib.c` includes `common/header1.h`, which is not explicitly listed as a prerequisite of `mylib.o`, so eDepend records this implicit dependency in the history file.

Directory	Object	Dependency
src/mylib	mylib.o	common/header1.h

If a change is then made to `common/header1.h`, `src/mylib/mylib.o` it will be rebuilt.

Important Notes

1. The eDepend list is consulted only if all other prerequisites in the makefile indicate the target is up-to-date.

Explained another way: If a target is out-of-date because it does not exist or because it is older than one of the prerequisites listed in the makefile, eDepend costs nothing and has no effect.

If the target is newer than all its listed prerequisites, then eDepend is the “11th hour” check to ensure it really is up-to-date, and that there is not a newer implicit dependency. This is the only place eDepend interacts with your build: it forces a target that incorrectly appears to be up-to-date to be rebuilt.

2. eDepend information, unlike traditional Makedepend rules, does not in any way imply anything about needing to build or update the implicit prerequisite.

In the example above, if `header1.h` is renamed or moved, Electric Make just ignores the stale eDepend information. When eMake next updates the `mylib.o` target, it will prune stale dependencies from the eDepend list. This change to the history file occurs regardless of the setting of the `--emake-history-force` parameter to eMake.

Unlike Make, Electric Make does not complain if it does not have a rule to make `header1.h` because eDepend dependencies are not used to schedule targets.

- eDepend's information scope is bound by a directory name and the target name. This means you can build cleanly from the top of a tree, then run accurate incremental builds from individual subdirectories and eDepend information will be used and updated correctly.

However, it does imply if you have a build that

- performs multiple passes or variants over the same directories
- with exactly the same target names, but
- runs significantly different commands

For example, a build that produces objects with the same name for a different architecture or configuration, eDepend information may be over-used unnecessarily. In this case, Electric Make may rebuild more than is necessary, but with no incorrect build results. In this situation, you can achieve fast, correct builds by using separate history files, or ideally, by changing to unique target names across build variants.

Using #pragma noautodep

Some build steps contain many implicit dependencies that may not make sense to check for up-to-dateness. Examples include symbol files consumed by a link step or archive packager input files (for example, *tar*). In both cases, any makefile explicit prerequisites are sufficient to determine if the target should be updated: eDepend information would just add overhead or cause unnecessary rebuilds.

You can selectively disable eDepend information for certain files from any step by supplying Electric Make with the makefile directive:

```
#pragma noautodep *.pdb
%.o: %.c
    $(CL) ...
```

The directive `#pragma noautodep` is applied to the next rule or pattern rule in the makefile. This directive specifies a class of files for eDepend to ignore. Note the following information about `#pragma autodep`:

- Wildcards allowed for `#pragma autodep`:

Wildcard	Meaning
*	matches 0 or more characters
?	matches 1 character
[]	matches a range of characters

- The `noautodep` filters are matched against absolute pathnames. To limit a filter to files in the current directory for the job, use `./`:

```
#pragma noautodep ./foo.h
```

To specify "ignore foo.h" in any directory, use:

```
#pragma noautodep */foo.h
```

- If the supplied pattern has no wildcards and does not specify a path, it will never match.

Electric Make ignores the directive and prints a warning as it parses the makefile:

```
Makefile:2: ignoring autodep filter 'foo',
does not match absolute path(s).
```

ElectricAccelerator Ledger File

Traditional Make facilities rely exclusively on a comparison of filesystem timestamps to determine if the target is up-to-date. More specifically, an existing target is considered out-of-date if its inputs have a “last-modified” timestamp later than the target output.

For typical interactive development, this scheme is adequate: As a developer makes changes to source files, their modification timestamps are updated, which signals Make that dependent targets must be rebuilt. There is, however, a class of workflow styles that cause file timestamps to move arbitrarily into the past or future, and therefore circumvent Make's ability to correctly rebuild targets.

Two common examples are:

- Using a version control system that preserves timestamps on checkout (also known as “sync” or “update”).
The default mode for most source control systems is to set the last-modified timestamp of every file updated in a checkout or sync operation to the current day and time. If you change this behavior to preserve timestamps (or if your tool's default mode is *preserve*), then updating your source files can result in modified contents but with a timestamp in the past (typically, it is the time of the checkin).
- Using file or directory synchronization tools (even simple recursive directory copies) to keep files updated against some other repository.

Here again, while it is easy to modify source file content, the timestamp for modifications may be any of several possibilities: time of copy, last-modified time of source, last-modified time of destination, and so on.

The Problem

In all modified source files cases, we would like the Make system to rebuild any dependent objects. However, because timestamps of modified files are not set reliably, Make may or may not force a target update. Here is an example Makefile:

```
foo.o: foo.c
    gcc -c foo.c

foo.o: foo.h
```

And a build is run without an existing `foo.o` object:

```
% make
gcc -c foo.c

% ls -lt
total 4
-rw-r--r-- 1 jdoe None 21 May 29 13:50 foo.o
-rw-r--r-- 1 jdoe None 21 May 29 13:50 foo.c
-rw-r--r-- 1 jdoe None 20 Apr 25 17:34 foo.h
-rw-r--r-- 1 jdoe None 41 Jan 19 09:27 Makefile
```

The `foo.o` target is updated. Next, suppose we ask our source control system to update the working directory, and it responds by giving us a newer copy of `foo.h`, one that is several weeks newer than what we have, and *that* timestamp is preserved:

```
% <sync>
% ls -lt
total 4
-rw-r--r-- 1 jdoe None 21 May 29 13:50 foo.o
-rw-r--r-- 1 jdoe None 21 May 29 13:50 foo.c
-rw-r--r-- 1 jdoe None 29 May 17 11:21 foo.h <-- notice timestamp change
-rw-r--r-- 1 jdoe None 41 Jan 19 09:27 Makefile
```

Traditional Make programs (here, GNU Make) will not notice the change because the timestamp is still in the past, and will incorrectly report that the target is up-to-date.

```
% make
make: `foo.o' is up to date.
```

Some Make facilities (notably, Rational 'clearmake' in conjunction with Rational ClearCase) have the ability to track timestamp information because they are integrated with the source control system.

The Electric Make Solution

Electric Make solves this problem at the file level, completely independent of the source control system, by keeping a separate database of inputs and outputs called a *ledger*. To use the Ledger, specify which file aspects to check for changes when considering a rebuild. The (nonexclusive) choices are:

- `timestamp` – any timestamp change, regardless of how it relates to the last modified time of the target input file, triggers a target rebuild.
- `size` – any size change, regardless of the timestamp in the input file, triggers a target rebuild.
- `command` – records the text of the command used to create the target. If `makefile` or its variables change, using `command` rebuilds the target. **Important caveat:** If you initialize a variable using the `$(shell)` function, be extremely careful to use the `$(shell)` function with a `':='` assignment to avoid re-evaluating it every time the variable is referenced. `':='` simply expanded variables are expanded immediately upon reading the line.
- `nobackup` – suppresses the automatic backup of the ledger file before its use.
- `nonlocal` – instructs eMake to operate on the ledger file in its current location, even if it is on a network volume. By default, if the file specified by `--emake-ledgerfile` (`emake.ledger` in the current working directory, by default) is not on a local disk, eMake copies that file (if it already exists) to the system temporary directory and opens the copy, then copies it back to the specified location when the build is complete.

Using `nonlocal` removes a safety and may cause problems if the non-local filesystem has issues with memory-mapped I/O (IBM Rational ClearCase MVFS is known to have issues with memory-mapped I/O). If you are confident that you will get efficient and reliable memory-mapped I/O performance from the non-local filesystem, you can remove the safety for improved efficiency because eMake does not spend time at startup and shutdown copying ledger files. Electric Cloud strongly recommends against using `nonlocal` with ClearCase dynamic views. Electric Cloud does not support Ledger-related problems that occur when `nonlocal` is used in conjunction with the MVFS.

Use the `--emake-ledger=<valuelist>` command-line switch (or the `EMAKE_LEDGER` environment variable) to specify one or more of the following: `timestamp`, `size`, `command`, `nobackup`, `nonlocal` (`timestamp`, `size`, `command`, `nobackup`, `nonlocal`). See Ledger options in [Chapter 10, “Electric Make Command-line Options and Environment Variables.”](#)

In the example above, the Ledger can detect if a rebuild is necessary as the timestamps change. If the original build was:

```
% emake --emake-ledger=timestamp
gcc -c foo.c
% <sync> <-- changes header timestamp only
% emake --emake-ledger=timestamp
gcc -c foo.c
```

Electric Make consulted the Ledger and concluded the target needed to be rebuilt.

Important Notes for the Ledger Feature

- The Ledger feature works by comparing an earlier input state with the current state: if the Ledger has no information about a particular input (for example, during the first build after it was added to a makefile), it will not contribute in the up-to-dateness check.
- Only one Ledger is used per build.
- The default ledger file is called `emake.ledger`. It can be adjusted by the `--emake-ledgerfile=<path>` command-line option or `EMAKE_LEDGER_FILE=<path>` environment variable.
- If you specify `--emake-ledgerfile=<path>` but not `--emake-ledger=<valuelist>`, the Ledger still hashes the filenames, so the Ledger is triggered when the filename order changes or a file is added or removed.
- The Ledger automatically backs up the ledger file before using it. This ensures a non-corrupt file is available. If the ledger file is large, copying it could take some time on incremental builds. The `ledger` option, `nobackup`, suppresses the backup.
- Ledger works for local builds and those using a cluster, as well as local submakes in a `runlocal` job, see [“Running a Local Job on the Make Machine”](#) on page 9-2.
- It is not possible, however, to share a Ledger between top-level make instances and local-mode submakes running on the cluster. See `EMAKE_BUILD_MODE=local` on [page 10-2](#).

- eMake consults Ledger information to trigger a rebuild only when a target would otherwise be considered up-to-date. Information in the Ledger never prevents a target from being rebuilt.
- In a GNU Make emulation, the Ledger feature changes the meaning of the '\$?' automatic variable to be synonymous with '\$^' (all prerequisites, regardless of up-to-dateness).
- You cannot change Ledger options for a particular ledger file—you must use the same combination of timestamp, size, and command that was used to create the ledger file.
- If you turn on `--emake-ledger` and `--emake-autodepend` at the same time, the Ledger keeps track of both implicit and explicit dependencies. This feature is comparable to using ClearMake under ClearCase, but is independent of ClearCase information records.
- Order-only prerequisites, in keeping with their semantic meaning, never affect Ledger behavior.

Managing the History Data File

When ElectricAccelerator runs a build for the first time, it takes aggressive action to run all jobs as fast as possible in parallel. Jobs that run in the wrong order because of missing makefile dependencies are automatically re-run to ensure correct output. (These failed steps are referred to also as *conflicts*.)

To avoid the cost of re-running jobs on subsequent builds, Electric Make saves the missing dependency information in a *history data file*. The history data file evolves with each new run and enables ElectricAccelerator to continually run builds at peak efficiency.

You can choose the location of the history file and how it is updated.

Setting the History File Location

By default, Electric Make creates the history file in the directory you use to invoke the build and names it `emake.data`.

The file location can be explicitly specified using the command-line option

```
--emake-historyfile=<pathname>.
```

Selecting History File Options

The history file (by default, `emake.data`) is used for two separate operations during an Electric Make cluster build:

- Input – Electric Make reads the history file as it starts a build to improve build performance.
- Output – Electric Make *writes* to the history file *as it completes a build* to improve performance of subsequent builds.

Input Rules

If the history file (`emake.data` or whatever was specified with `--emake-historyfile`) exists, it is *always* read and used to improve build performance at the time of build execution.

Output Rules

Data written to the history file when the build completes depends on the `--emake-history` option setting. Three options are available:

1. Merge – By default, Electric Make merges any new dependencies it discovers into the existing history file. In this way, the history file automatically evolves as your makefiles change, learning dependencies that accelerate the builds you run.
2. Create – If `--emake-history` is set to `create`, the old history file contents are completely overwritten by new dependencies discovered in the run that just completed. Use this setting to start a fresh history file, effectively eliminating any stale information from the file.
3. Read – If `--emake-history` is set to `read`, no data is written to the history file at build completion, and any new dependencies Electric Make discovered are discarded. Use this setting when you are sharing a single, static copy of the history file between developers.

By default, the history file is updated even if the build fails, regardless of the set value for `--emake-history`. You can override this behavior by setting the command-line option to `--emake-history-force=0`.

The history file directly impacts the number of conflicts the build can encounter. Ideally, an ElectricAccelerator cluster build with good history should have close to “0” conflicts. If you see conflicts starting to rise, ensure you have a current history file for the build you are executing.

Important Notes for Creating and Using the History File

- One-Agent build to guarantee correct history – Some parallel builds will not succeed without a good history file. In particular, builds that use wildcard or globbing operators to produce build generated lists of files and operate on those lists may fail. For example, a makefile may use `ld *.o` as shorthand to avoid enumerating all the `*.o` files in a directory. Running the build against one Agent (different from running in local mode—local builds do not use history) guarantees the build will succeed and a history file is created for use by subsequent parallel builds.
- Relative `EMAKE_ROOT` locations must match – The history file records target filenames relative to the `EMAKE_ROOT` specified during that run. For a subsequent build to correctly use the history file, target filenames must have the same path name relative to the eMake root.

For example:

If your eMake root is set to: `/home/alice/builds`
 and your build references a path name in that root: `/home/alice/builds/lib/foo.o`
 then the history file records it as `lib/foo.o`
 If a subsequent build sets the eMake root to: `/home/bob/builds`
 the history file will match correctly.

If, however, the eMake root is set to: `/home/bob`
 then the file that exists on the disk as: `/home/bob/builds/lib/foo.o`
 gets the root-relative name of: `builds/lib/foo.o`
 which does not match the name `lib/foo.o` in the history file generated above.

Because the history file does not match, performance can suffer.

Note: Be sure your `EMAKE_ROOT` matches the same location relative to sources as the `EMAKE_ROOT` used to create the history file you are using.

- For builds with multiple roots, the roots must have the same alphabetical sorting order in each build in order for history to match.

What is Annotation?

As Electric Make runs a build, it discovers a large amount of information about the build structure. This information can be written to an “annotation” file for use after the build completes. Annotation information is represented as an XML document so it is easy to parse.

eMake collects many different types of information about the build depending on various command-line switches. The information eMake collects includes:

- makefile structure
- commands and command output
- list of file accesses by each job
- dependencies between jobs
- detailed timing measurements
- eMake invocation arguments and environment
- performance metrics

eMake can also be configured to upload annotation information to the Cluster Manager for centralized reporting purposes.

Configuring eMake to Generate an Annotation File

By default, Electric Make collects configuration information and performance metrics only, which it sends to the Cluster Manager at the end of the build. This data is used to display reports on the **Build Details** page.

Optionally, eMake can be configured to collect additional information. This extra annotation information is written to an XML file in the build directory (`emake.xml` by default). The `--emake-annodetail` command-line switch controls the amount of information eMake should collect. Default annotation detail is determined by the build class for that build.

Supported annotation detail flags:

- **basic** - The basic mode collects information about every command run by the build. Detailed information about each “job” in the build is recorded, including command arguments, output, exit code, timing, and source location. In addition, the build structure is represented as a tree where each recursive make level is represented in the XML output.
- **env** - The env mode adds information about environment variable modifications.
- **file** - The file mode adds information about files read or written by each job.
- **history** - The history mode adds information about missing serializations discovered by eMake. This includes information about which file caused two jobs to become serialized by the eMake history mechanism.
- **lookup** - The lookup mode adds information about files that were looked up by each job. **Note:** *This mode can cause the annotation file to become quite large.*
- **md5** - The md5 mode computes MD5 checksums for files read and written by the build, and includes that information in annotation as an md5 attribute on appropriate `<op>` tags. The list of operation types that will include the checksum is read, create, and modify. No checksum is generated or emitted for operations on directories

or symlinks, or for append operations. If a read file was appended to, and the read occurs before the appended update is committed, you will see a zero checksum on that read operation (by design because reading files that were appended to is a rare occurrence). The md5 mode implies “file” level annotation.

- **registry** - The registry mode adds information about registry operations.
- **waiting** - The waiting mode adds information about the complete dependency graph for the build.

All of the detail settings automatically enable “basic” annotation.

The `--emake-annoupload` command-line switch controls whether or not eMake sends a copy of the annotation file to the Cluster Manager as the build runs. By default, eMake sends minimal information to the Cluster Manager, even if more detailed annotation is enabled. eMake sends the full annotation file if annotation uploading is configured by the build class or by the eMake command line.

Note: You cannot disable `mergestreams` if you enable annotation. Enabling annotation automatically enables `mergestreams`, even if it was explicitly disabled on the command line.

Annotation File Splitting

Due to a gcc file size write limit, annotation file size is limited to 1.6 GB. Large annotation files are split to remain under this limit. This limit is not configurable.

ElectricInsight requires a single annotation file. To rejoin split annotation files:

On UNIX:

```
cat file1 > result_file
cat file2 >> result_file
```

On Windows:

```
type file1 > result_file
type file2 >> result_file
```

Note: Though there is no predetermined limit at which an annotation file cannot be loaded by ElectricInsight, the maximum size is limited by the size of the process because ElectricInsight is a 32-bit application.

Working with Annotation Files

The simplest way to use an eMake annotation file is to load it into the Electric Cloud ElectricInsight® product. This program allows the user to see a graphical representation of the build, search the annotation file for interesting patterns, and perform sophisticated build analysis using its built-in reporting tools.

Also, you can write your own tools to perform simple tasks using annotation output. For example, reporting on failures in the build can be accomplished by looking for “failed” elements inside job elements and then reporting various details about the failed job such as the commands, their output, and the line of the makefile that contains the rule for the command. Refer to the DTD for the annotation file format below.

Annotation XML DTD

```
<!-- build.dtd -->
<!-- The DTD for Emake's annotated output. -->
<!-- -->
<!-- Copyright (c) 2004-2008 Electric Cloud, Inc. All rights reserved. -->

<!ENTITY % hexnum "CDATA">
<!ENTITY % job "(message*, job)">
<!ENTITY % valueName "name NMTOKEN #REQUIRED">
<!-- Can't use NMTOKEN because Windows has environment variables like
      "=D:". -->
<!ENTITY % envValueName "name CDATA #REQUIRED">

<!ELEMENT build
  (properties?, environment?, (message* | make)+, fs?, metrics? )
>
<!ATTLIST build
  id      CDATA #REQUIRED
```

```

    cm    CDATA #IMPLIED
    start CDATA #REQUIRED
  >
<!-- Out of band build messages -->

<!ELEMENT message (#PCDATA) >
<!ATTLIST message
  thread    %hexnum;          #REQUIRED
  time      CDATA             #REQUIRED
  code      CDATA             #REQUIRED
  severity  ( warning | error ) #REQUIRED
>
<!-- Properties list -->

<!ELEMENT properties (property*) >
<!ELEMENT property  (#PCDATA) >
<!ATTLIST property
  %valueName;
>
<!-- Environment list -->

<!ELEMENT environment (var*) >
<!ELEMENT var    (#PCDATA) >
<!ATTLIST var
  %envValueName;
  op ( add | modify | delete ) "add"
>
<!-- File system dump -->

<!ELEMENT fs      (roots, symRoots, (content|name)*) >
<!ELEMENT roots  (root+) >
<!ELEMENT root   (#PCDATA) >
<!ATTLIST root
  nameid CDATA #REQUIRED
>
<!ELEMENT symRoots (symRoot*) >
<!ELEMENT symRoot (#PCDATA) >
<!ATTLIST symRoot
  symLinkPath CDATA #REQUIRED
>
<!ELEMENT content (contentver+) >
<!ATTLIST content
  contentid CDATA #REQUIRED
>
<!ELEMENT contentver EMPTY >
<!ATTLIST contentver
  job CDATA #REQUIRED
>
<!ELEMENT name (namever*) >
<!ATTLIST name
  nameid CDATA #REQUIRED
  dir    CDATA #REQUIRED
  name   CDATA #REQUIRED
>
<!ELEMENT namever EMPTY >
<!ATTLIST namever
  job      CDATA #REQUIRED
  contentid CDATA #REQUIRED
>
<!-- Metrics list -->

<!ELEMENT metrics (metric*) >
<!ELEMENT metric  (#PCDATA) >
<!ATTLIST metric
  %valueName;
>
<!-- Make subtree -->

<!ELEMENT make
  (environment?, ( message | job | make )*)
>
<!ATTLIST make
  level CDATA #REQUIRED
  cmd   CDATA #REQUIRED
  cwd   CDATA #REQUIRED
  mode  ( gmake | nmake | symbian | ant ) #REQUIRED
>
<!-- Job -->

```

```

<!ELEMENT job
  (environment?,(output | command | conflict)*,depList?,opList?,
  registryOpList?,timing+,failed?,waitingJobs?)
>
<!ATTLIST job
  thread      %hexnum; #REQUIRED
  id          ID #REQUIRED
  status      ( normal | rerun | conflict | reverted | skipped ) "normal"
  type        ( continuation | end | exist |
              follow | parse | remake | rule ) #REQUIRED
  name        CDATA #IMPLIED
  file        CDATA #IMPLIED
  line        CDATA #IMPLIED
  neededby    IDREF #IMPLIED
  partof      IDREF #IMPLIED
  node        CDATA #IMPLIED
>
<!-- Command and related output, output blocks can contain nested -->
<!-- make subtrees in local mode. -->

<!ELEMENT command
  (argv,inline*,(output | make)*)
>
<!ATTLIST command
  line CDATA #IMPLIED
>
<!ELEMENT argv (#PCDATA)>
<!ELEMENT inline (#PCDATA)>
<!ATTLIST inline
  file CDATA #REQUIRED
>
`<!ELEMENT output (#PCDATA)>
<!ATTLIST output
  src ( prog | make ) "make"
>
<!-- Conflict description -->

<!ELEMENT conflict EMPTY>
<!ATTLIST conflict
  type      ( file | cascade | name | key | value ) "cascade"
  writejob  IDREF #IMPLIED
  file      CDATA #IMPLIED
  rerunby   IDREF #IMPLIED
  hkey      CDATA #IMPLIED
  path      CDATA #IMPLIED
  value     CDATA #IMPLIED
>
<!-- Job failure code -->

<!ELEMENT failed EMPTY>
<!ATTLIST failed
  code CDATA #REQUIRED
>
<!-- List of jobs waiting for this job, local mode only -->

<!ELEMENT waitingJobs EMPTY>
<!ATTLIST waitingJobs
  idList IDREFS #IMPLIED
>
<!-- Start and stop times of this job -->

<!ELEMENT timing EMPTY>
<!ATTLIST timing
  invoked   CDATA #REQUIRED
  completed CDATA #REQUIRED
  node      CDATA #IMPLIED
>
<!-- Dependency list, only used when annoDetail includes 'history' -->

<!ELEMENT depList (dep*)>
<!ELEMENT dep      EMPTY>
<!ATTLIST dep
  writejob IDREF #REQUIRED
  file     CDATA #REQUIRED
>
<!-- Operation list, only present when annoDetail includes -->
<!-- 'file' or 'lookup' -->

```

```

<!ELEMENT opList (op*)>
<!ELEMENT op      EMPTY>
<!ATTLIST op
  type      ( lookup | read | create | modify | unlink | rename |
             link | modifyAttrs | append | blindcreate ) #REQUIRED
  file      CDATA #REQUIRED
  other     CDATA #IMPLIED
  found     ( 1 | 0 ) "1"
  isdir     ( 1 | 0 ) "0"
  filetype  ( file | symlink | dir ) "file"
  atts     CDATA #IMPLIED
>
<!-- Registry operation list, only present when annoDetail includes -->
<!-- 'registry' -->

<!ELEMENT registryOpList (regop*)>
<!ELEMENT regop      (#PCDATA)>
<!ATTLIST regop
  type      ( createkey | deletekey | setvalue | deletevalue |
             lookupkey | readkey ) #REQUIRED
  hkey     CDATA #REQUIRED
  path     CDATA #REQUIRED
  name     CDATA #IMPLIED
  datatype ( none | sz | expandsz | binary | dword | dwordbe |
            link | multisz | resourcelist | resourcedesc |
            resourcereqs | qword ) "none"
>

```

Metric Values in Annotation Files

The following values are available when you select the Metrics option from the dropdown menu on the Build Details page in the web interface.

Metric	Description
terminated	Number of jobs that ran to completion. These jobs may not necessarily have done anything.
conflicts	Number of jobs that ran into conflicts and had to be re-run.
runjobs	Number of jobs that did work.
writeThrottleWaitTime	Amount of time spent waiting for the write throttle (which is in place to avoid slowing the build by seeking the disk head back and forth too often).
duration	Duration of build.
clusterAvailability	Cluster availability percentage.
totalAgentCount	Total number of agents in cluster.
noAgentsWaitTime	Amount of time spent waiting with no agents allocated to the build.
totalChains	Total number of FSChains in the build—anything in the file system tracked by eMake for versioning is an FSChain, for example, file names, file contents. Because both names and contents are tracked, this should be at least twice the number of files accessed in the build.
chainLatestReads	Number of times the latest version of an FSChain was requested.
chainSerialReads	Number of times an FSChain was requested to match a particular spot in the serial order.
chainWrites	Number of new versions created, which occurs any time a new name is created, or the content of a file changes.
totalVersions	Total number of FSChain versions. Available with <code>--emake-debug=g</code> .

Metric	Description
totalNameVersions	Total number of Name versions. Available with <code>--emake-debug=g</code> .
emptyNameVersions	Total number of Name versions with no associated Content. Available with <code>--emake-debug=g</code> .
elapsed	The total elapsed time for the build.
nodeReadBytes	The number of bytes read from agents.
nodeWriteBytes	The number of bytes written to agents.
nodeReadTime	Time spent reading data from agents.
nodeWriteTime	Time spent writing data to agents.
nodeReadWaitTime	Time spent waiting for data to be read from agents.
nodeWriteWaitTime	Time spent waiting for data to be written to agents.
localAgentReadBytes	Bytes copied from local agents.
localAgentWriteBytes	Bytes copied to local agents.
localAgentReadTime	Time spent copying data from local agents.
localAgentWriteTime	Time spent copying data to local agents.
diskWriteBytes	The number of bytes written to the local disk.
diskReadBytes	The number of bytes read from the local disk.
diskReadTime	Time spent reading data from the local disk.
diskWriteTime	Time spent writing data to the local disk.
diskReadWaitTime	Time spent waiting for data to be read from the local disk.
diskWriteWaitTime	Time spent waiting for data to be written to the local disk.
hiddenWarningCount	The number of warning messages hidden by the eMake client and all remote parse jobs, with one count for every message number for which at least one message was hidden. The count does not include messages hidden by eMake stubs or rlocal-mode emakes.
usageBytes	The number of bytes received for usage data.
symlinkReads	The number of times the application read symlinks from the local disk.
termDiskCopiedBytes	The number of bytes committed by copying. This should be a small number if possible. If you are using ClearCase and this number is nonzero, look into the <code>--emake-clearcase=vobs</code> option.
termDiskMovedBytes	The number of bytes committed by moving. You want this to be big number if possible.
termDiskRemovedBytes	The number of bytes committed by remove (original file).
termDiskRemovedTmpBytes	The number of bytes committed by remove (temporary file).
termDiskCopiedFiles	The number of files committed by copying. This number should be small if possible.
termDiskMovedFiles	The number of files committed by moving. You want this to be a big number if possible.

Metric	Description
termDiskRemovedFiles	The number of files committed by remove (original file).
termDiskRemovedTmpFiles	The number of files committed by remove (temporary file).
compressTime	Time spent compressing data.
compressBytesIn	The number of bytes passed in for compression.
compressBytesOut	The number of bytes returned from compression.
decompressTime	Time spent decompressing data.
decompressBytesIn	The number of bytes passed in for decompression.
decompressBytesOut	The number of bytes returned from decompression.
maxMakeCount	Peak active make instances during the build.
maxArenaCount	Peak active arenas during the build.
freezeTime	Time the job queue was frozen, which means only high priority items are taken of the queue.

Timers

Note that most timers will not be available unless you run with `--emake-debug=g` (for profiling). These timers correspond to the amount of time eMake spent in certain areas of the code or in a certain state.

Metric	Description
timerThreadCount	Number of threads in eMake.
timer:agentManager.shouldRequestAgents	Time spent checking to see if eMake should be requesting agents, which involves talking to the Cluster Manager.
timer:agentManager.startup	The time spent for the Agent Manager to start up.
timer:agentManager.stop	Time spent for the Agent Manager to shut down.
timer:agentManager.work	Time spent with the Agent Manager actively doing work outside <code>shouldRequestAgents</code> .
timer:annoUpload.startup	Time spent starting up the thread to upload annotation.
timer:annoUpload.work	Time spent uploading annotation.
timer:bench	Benchmark showing the cost of 100 invocations of the timer code (start/stop).
timer:directory.populate	Time spent making sure that eMake's model of the directory contents is fully populated.
timer:history.parsePrune	Time spent in parse jobs signaling stale history entries to prune stale events.
timer:history.pruneFollowers	Time spent signaling stale submakes (for jobs that have followers) to be pruned.
timer:history.pruneNoFollower	Time spent signaling stale submakes (for jobs that have no followers) to be pruned in jobs.
timer:idle.agentManager	Time spent in the Agent Manager sleeping between <code>shouldRequestAgents</code> checks.
timer:idle.agentManagerRequest	Time spent in the Agent Manager waiting for the Cluster Manager to respond to a request for agents.
timer:idle.agentRun	Time spent by worker threads waiting for requests from the agent.

Metric	Description
timer:idle.annoUpload	Time spent by the annotation upload thread waiting for data.
timer:idle.noJobs	Time spent by worker threads waiting for a new runnable job to enter the job queue.
timer:idle.untilCompleted	Time spent by the Terminator thread waiting for jobs to be completed.
timer:idle.waitForAgent	Time spent by worker threads waiting for an agent to become available.
timer:jobqueue	Time spent within the lock guarding the job queue.
timer:Ledger.commit	Time spent committing Ledger data to the database.
timer:Ledger.close	Time spent closing the Ledger and flushing data to disk.
timer:Ledger.isUpToDate	Time spent querying the Ledger to find if a file is up to date.
timer:Ledger.staleAttributes	Time spent by the Ledger code statting files to ensure recorded attributes match the actual attributes on disk.
timer:Ledger.update	Time spent updating the Ledger.
timer:main.commit	Time spent by the Terminator thread committing jobs, less time spent flushing deferred writes to disk.
timer:main.history	Time spent reading and writing history files.
timer:main.terminate	Time spent by the Terminator thread terminating jobs.
timer:main.lockedWriteToDisk	Time spent by the Terminator thread flushing deferred writes to disk.
timer:main.writeToDisk	Time spent by the Terminator thread writing operations to disk, less the time covered by any of the other main.writeToDisk timers.
timer:main.writeToDisk.append	Time spent appending to existing files on disk.
timer:main.writeToDisk.create dir	Time spent creating directories on disk.
timer:main.writeToDisk.create dir.attrs	Time spent changing directory attributes on disk for newly created directories.
timer:main.writeToDisk.create dir.chown	Time spent changing directory ownership on disk for newly created directories.
timer:main.writeToDisk.create dir.times	Time spent changing directory times on disk for newly created directories.
timer:main.writeToDisk.createfile	Time spent writing file data to disk for newly created files.
timer:main.writeToDisk.createfile.attrs	Time spent changing file attributes on disk for newly created files.
timer:main.writeToDisk.createfile.chown	Time spent changing file ownership on disk for newly created files.
timer:main.writeToDisk.createfile.times	Time spent changing file times on disk for newly created files.
timer:main.writeToDisk.link	Time spent creating links on disk.
timer:main.writeToDisk.modify	Time spent modifying existing files on disk.

Metric	Description
timer:main.writeToDisk.modifyAttrs	Other time spent in writing attribute changes to disk (mostly notifying the file system that attributes have gone “stale”).
timer:main.writeToDisk.modifyAttrs.attrs	Time spent modifying attributes of existing files on disk.
timer:main.writeToDisk.modifyAttrs.chown	Time spent modifying ownership of existing files on disk.
timer:main.writeToDisk.modifyAttrs.times	Time spent modifying times of existing files on disk.
timer:main.writeToDisk.unlink	Time spent unlinking existing files on disk.
timer:main.writeToDisk.unlink.data	Time spent recording the fact that a file was removed.
timer:main.writeToDisk.unlink.tree	Time spent removing entire trees on disk.
timer:mergeArchiveRefs	Time spent modifying word lists for multi-word archive references like <code>lib(member1 member2 ...)</code> .
timer:mux.DirCache	Time spent waiting for a directory cache.
timer:mux.filedata.nodelist	Time spent within the lock guarding the list of agents allocated to the build.
timer:mux.jobcreate	Time spent within the lock used to synchronize the terminator and worker threads when creating new jobs.
timer:mux.joblist	Time spent within the lock used to protect the job list.
timer:mux.jobrunstate	Time spent within the lock used to coordinate starting and canceling jobs.
timer:mux.nodeinit	Time spent within the lock used to protect the list of hosts while initializing agents.
timer:mux.target	Time spent within the lock protecting failure tracking on a target.
timer:node.putAllVersions.getShortName	Time spent getting file short names on Windows when doing an <code>E2A_PUT_ALL_VERSIONS</code> .
timer:node.setup	Time spent connecting to hosts and initializing them.
timer:node.svc.getData	Time spent handling <code>A2E_GET_FILE_DATA</code> and <code>A2E_RESEND_FILE_DATA</code> , not including the time spent sending data in response.
timer:node.svc.getData.acquireLock	Time spent waiting for the ChainLock when handling <code>A2E_GET_FILE_DATA</code> and <code>A2E_RESEND_FILE_DATA</code> .
timer:node.svc.getData.copy	Time spent copying file data for <code>E2A_LOAD_LOCAL_FILE</code> . This is the local agent version of <code>timer:node.svc.getData.send</code> .
timer:node.svc.getData.insideLock	Time spent holding the ChainLock when handling <code>A2E_GET_FILE_DATA</code> and <code>A2E_RESEND_FILE_DATA</code> .
timer:node.svc.getData.send	Time spent sending file data during <code>E2A_PUT_FILE_DATA</code> and <code>E2A_PUT_BIG_FILE_DATA</code> .
timer:node.svc.getDir	Time spent sending directory entry data to the agent.
timer:node.svc.getVersions	Time spent handling <code>A2E_GET_ALL_VERSIONS</code> requests.

Metric	Description
timer:node.svc.getVersions.acquireLock	Time spent waiting for the ChainLock when handling A2E_GET_ALL_VERSIONS.
timer:node.svc.getVersions.insideLock	Time spent holding the ChainLock when handling A2E_GET_ALL_VERSIONS.
timer:node.svc.runCommand	Time spent handling A2E_RUN_COMMAND requests.
timer:usage.io	Time spent reading and responding to usage data.
timer:usage.io.makedata	Time spent creating file data from usage.
timer:usage.io.makedata.local	Time spent copying file data from usage reported by local agents; a subset of timer:usage.io.makedata.
timer:usage.latency	Time spent dispatching incoming usage data and saving output files.
timer:usage.record	Time spent recording usage data, including resolving new name IDs, removing duplicate lookup records, and so on.
timer:worker.continuationjob	Time spent by worker threads running continuationJobs.
timer:worker.endjob	Time spent by worker threads running endJobs.
timer:worker.existencejob	Time spent by worker threads running existenceJobs.
timer:worker.followjob	Time spent by worker threads running followJobs.
timer:worker.invoke	Time spent invoking remote jobs.
timer:worker.invokelocal	Time spent invoking local jobs.
timer:worker.other	Otherwise unaccounted time spent by worker threads.
timer:worker.parsejob	Time spent by worker threads running parseJobs.
timer:worker.remakejob	Time spent by worker threads running remakeJobs.
timer:worker.rulejob	Time spent by worker threads running ruleJobs, less the time spent actually running commands and figuring out if it needs to run.
timer:worker.rulejob.needtorun	Time spent by ruleJobs figuring out if they need to run.
timer:worker.runcommands	Time spent by worker threads running commands in commandJobs.
timer:worker.shouldRequestAgents	Time spent by worker threads checking to see if they should request agents.
timer:worker.startup	Time spent by worker threads initializing.
timer:worker.stop	Time spent shutting down worker threads.

Managing Temporary Files

During the build process, temporary files are created, then deleted automatically when a build completes. To maintain optimum efficiency, it is important to identify where these files are created in relation to the final output files.

Configuring the Electric Make Temporary Directory

Electric Make runs commands and collects output in parallel, but the results are written in order, serially. This feature ensures a build behaves exactly as if it were run locally. See [“Transactional Command Output” on page 5-3](#).

The Electric Make *temporary directory* is used to store output files from commands (for example, object files) that finished running but are waiting their turn to be relocated to a final destination on disk. By default, Electric Make creates a temporary directory in the current working directory. The directory name will have the form:

```
ecloud_tmp_<pid>_<n>
```

where `pid` is the eMake process identifier.

where `n` is a counter assigned by eMake to differentiate multiple temporary directories created during a single build if you specify multiple directories.

Electric Make removes the temporary directory on exit (including **Ctrl-C** user interrupt). If Electric Make is terminated unexpectedly (for example, by the operating system), the temporary directory may persist and need to be removed manually.

Important temporary directory requirements include:

- Must be writable by the eMake process
- Must not be on an NFS share
- Must have enough space to contain the build output
 - Ideally, the temporary directory should have enough space to hold the complete build output; in practice however, it may need only enough space for output from the largest commands.
 - The exact temporary directory space requirement varies greatly with the build size, the number of Agents used, and build system speed.

You can use the `--emake-tmpdir` command-line option or the `EMAKE_TMPDIR` environment variable to change the temporary directory default location:

```
% emake --emake-tmpdir=/var/tmp ...
```

Files are relocated from the temporary directory to their final location. Keeping the temporary directory on the same file system as the `EMAKE_ROOT` helps performance because files can simply be renamed in place, instead of copied.

Electric Make and agent machines reset the values for `TEMP`, `TMP`, and `TMPDIR`. This is necessary to avoid possible conflicts with multiple build agents/jobs running on the same machine.

If you have more than one `EMAKE_ROOT` that spans multiple file systems, you can specify more than one temporary directory to ensure Electric Make can always rename files in place, instead of copying them.

For example, if your `EMAKE_ROOT` contains three directories:

```
% setenv EMAKE_ROOT /home/alice:/local/output:/local/libs
```

that reside on two different file systems

```
% df /home/alice /local
Filesystem      1K-blocks  Used   Available Use% Mounted on
filer:/vol/     76376484 47234436 25262352  66% /home
vol0/space
/dev/hdb1       76896316 51957460 21032656   2% /local
```

to ensure better performance, specify two temporary directories—write permission at both locations is required:

```
% emake --emake-tmpdir=/home/alice:/local ...
```

When specifying multiple temporary directories, note the following:

- If a temporary directory was specified for a particular file system, Electric Make automatically uses that directory for any files destined to reside on that file system.
- If a temporary directory was not specified for a particular file system, Electric Make uses the *first* directory specified for files destined to reside on that file system.

Deleting Temporary Files

During a build, Electric Make creates a temporary directory inside the directory specified by the `EMAKE_TMPDIR` environment variable or in the directory specified by the `--emake-tmpdir` command-line option (or the current working directory if no Electric Make temporary directory is specified). If a specified directory does not exist, eMake creates it. All temporary directories created by eMake are automatically deleted when a build completes.

For example, if you set your temporary directory to `/foo/bar/baz` and only `/foo/bar` exists, eMake creates `/foo/bar/baz` and `/foo/bar/baz/ecloud_tmp_<pid>_<n>`, and then deletes both directories and all their contents when it exits. If `/foo/bar/baz` exists at the start of the build, only `/foo/bar/baz/ecloud_tmp_<pid>_<n>` is created and deleted.

However, if the build is aborted in-progress, eMake will not have the opportunity to remove the temporary directory.

Subsequent Electric Make invocations automatically delete temporary directories if they are more than 24 hours old. You can reclaim disk space more quickly by deleting temporary directories and their contents by hand. However, *do not delete the temporary directory while a build is in-progress*—this action causes the build to fail.

Temporary directory names are in this form:

```
ecloud_tmp_<pid>_<n>
```

where `pid` is the eMake process identifier.

where `n` is a counter assigned by eMake to differentiate multiple temporary directories created during a single build if you specify multiple directories.

Running a Local Job on the Make Machine

Normally, ElectricAccelerator executes all build commands on the cluster hosts. Any file access by commands is served by in-memory EFS cache or fetched from Electric Make and monitored to ensure dependency information is captured, thus ensuring correct results.

Sometimes it is not desirable to run a command on an Agent (which could be in the cluster or locally installed by ElectricAccelerator Developer Edition). For example, the command could exercise such heavy file I/O, creating overhead [running remotely] to significantly degrade performance. In these instances, ElectricAccelerator provides the `#pragma runlocal` directive. In a makefile, the `#pragma runlocal` directive allows you to specify a job to run locally on the host build machine, instead of remotely.

A run-local job executes on the host build machine where no Electric File System exists to virtualize and monitor file access. Therefore, special care must be taken for build success. The following are two important properties of run-local jobs:

1. **Forced serial mode** – Electric Make executes a run-local job when all previous commands are complete, and does not allow other commands to run until the run-local job is complete. In other words, while the run-local job is running, no other build steps are executed.
2. **Current directory changes only** – The run-local job must not make any changes outside of its current working directory. If changes occur, subsequent jobs will not see the output outside of the current working directory and could fail—precise interaction is dependent on the Electric Make file system state at the time the job is invoked.

Important: After a run-local job, eMake reloads the filesystem state for the current working directory of the Make, not the job.

Note: Marking a job that does not meet this requirement with `#pragma runlocal` is not a supported configuration.

Because of these restrictions, run-local jobs are intended only for jobs that:

- Perform heavy I/O [consequently, very inefficient to run remotely]
- Execute near the end of the build (for example, forcing the build into serial mode would have minimal performance impact)
- Logically produce a final output not consumed by steps later in the build

Typically, final link or packaging steps are the only commands that should be marked *runlocal*.

If a run-local job modifies the registry, these modifications will not be visible to later jobs in the build. Ensure you do not mark a job as *runlocal* if that job modifies the registry in a way that later parts of the build depend on.

To mark a job runlocal, precede its rule in the makefile with `#pragma runlocal`. For example:

```
#pragma runlocal
myexecutable:
    $(CC) -o $@ $(OBJS) ...
```

Also, you can specify `#pragma runlocal` for a pattern rule. In this case, all jobs instantiated from that rule are marked run-local:

```
#pragma runlocal
%.exe: %.obj
    $(CC) -o $@ $*
```

You can cause all jobs after a certain point in the build to run locally by using `#pragma runlocal sticky`. This variant means all jobs that occur later in the serial order than the job specifically marked as “runlocal sticky” are run locally, as if “runlocal” were specified for all of them. This is NOT the same as “all jobs declared after this job in the makefile”! The order of declaration of jobs in a makefile has no relationship to the serial order of those jobs, so the difference is significant.

Serializing All Make Instance Jobs

Normally, Electric Make runs all jobs in a Make instance in parallel on multiple distinct Agents. For most builds, this process ensures the best possible performance.

In some cases, however, all jobs in a Make instance are interdependent and must be run serially, for example, a set of jobs updating a shared file. In particular, Microsoft Visual C/C++ compilers exhibit this behavior when they create a common program database (PDB) file to store symbol and debug information to all object files.

Example

The following makefile invokes `cl` with the `/Zi` flag to specify the program database file be created with type and symbolic debugging information:

```
all:
    $(MAKE) my.exe
    $(MAKE) other.exe

my.exe: a.obj b.obj c.obj my.obj
    cl /nologo /Zi $^ /Fe'my.exe'

%.obj: %.c
    cl /nologo /Zi /c /Fo$@ $^
```

In this build, the `a.obj`, `b.obj`, `c.obj` and `my.obj` jobs are implicitly serialized because they all write to the shared PDB file (by default, `vc70.pdb`). In this case, jobs run in parallel, and running them on separate Agents only introduces unnecessary network overhead. This job type needs to run serially so it can correctly update the PDB file.

The Electric Make special directive, `#pragma allserial` used in the makefile, allows you to disable a parallel build in a Make instance and run the job serially on a single Agent. By inserting the `#pragma allserial` directive at the beginning of a line anywhere in the makefile, the directive specifies that all jobs in that make instance be serialized. This process maximizes network and file cache efficiency.

In the example above, by prefixing the `%.obj` pattern rule with the `#pragma allserial` directive:

```
#pragma allserial
%.obj: %.c
    cl /nologo /Zi /c /Fo$@ $^
```

Electric Make runs compiles and links for the `my.exe` Make instance in serial on the same Agent.

Splitting PDBs Using `hashstr.exe`

The `hashstr.exe` utility creates a hash of the filename given a modulus (maximum number of PDBs that will be produced). A given file must always produce the same PDB or history would constantly change. The hash should only include the filename and not its full path. Precompiled headers (PCHs) must be turned off.

Example

Usage: `hashstr "mystring" [modulus]`

Where `mystring` is the string from which to generate the hash value, and `modulus` is the number of hash bins you want to use.

You can add this to a pattern rule for builds that suffer from performance degradation due to PDB serialization, with something similar to the following:

```
%.o: %.c
    $(CC) /c $(cflags) $(PCH_USE_FLAGS) $(cvars) $(cplus_flags) $(LOCAL_INCLUDE)
    $(PCB_INCLUDE) $< /Fo$@ /Fd$(shell ${path-to-hashstr}/hashstr.exe "$@"
    ${hashstr-modulus}).pdb
```

You can configure Electric Make options from the command line for a specific build and/or use Electric Make environment variables to set persistent options.

Introduction

A few caveats for using these option types:

- The environment variable `EMAKEFLAGS` can be used to set any command-line option. For example, this **emake** invocation:

```
% emake --emake-cm=mycm --emake-root=/home/joe
```

is equivalent to the following in `csh`:

```
% setenv EMAKEFLAGS "--emake-cm=mycm --emake-root=/home/joe"  
% emake
```

The *bash* equivalent is:

```
$ export "EMAKEFLAGS=--emake-cm=mycm --emake-root=/home/joe"  
$ emake
```

and in a Windows command shell:

```
C:\> set EMAKEFLAGS=--emake-cm=mycm --emake-root=C:\home\joe
```

- The hierarchy or precedence for setting an Electric Make option is:
 - Command-line options
 - `EMAKEFLAGS`
 - Environment variables

Using command-line options to set Electric Make values overrides values set using both `EMAKEFLAGS` and environment variables. Using `EMAKEFLAGS` to set options overrides the use of environment variables.

Command-line Options and Environment Variables List

Command-line options are listed in alphabetical order except for platform-specific options that are listed *after* platform-independent options. Debug options are listed at the end of the table.

Note: The `--emake-volatile` command-line option is deprecated and no longer has any effect. If the option is specified, it is ignored.

Command-line Options	Environment Variables	Description
	EMAKE_BUILD_MODE	Always set to local
<code>--emake-annodetail= var1[,var2[,...]]</code>		Specifies the level of detail to include in annotation output—a comma separated list for any of the following values: basic: Basic annotation history: Serialization details file: Files read or written lookup: All file names accessed waiting: Jobs that waited registry: updates to registry env: enhanced environment variables This option takes precedence over build class annotation settings set on the Cluster Manager.
<code>--emake-annofile=<file></code>		Specifies the name of the XML-formatted log output file. By default, the annotation file, <code>emake.xml</code> , is created in the directory where eMake is run. If specified, implies at least “basic” annotation details. The following macros are available: @ECLLOUD_BUILD_ID@ expands into the unique eMake build ID. @ECLLOUD_BUILD_TAG@ expands into the build tag from the Cluster Manager (this is displayed in the Name column on the Build tab in the Cluster Manager UI). @ECLLOUD_BUILD_DATE@ expands into an 8-digit code that represents the local system date where the build began, in the form YYYYMMDD . @ECLLOUD_BUILD_TIME@ expands into a 6-digit code that represents the 24-hour local system time where the build began, in the form HHMMSS . Example: <code>--emake-annofile=annofile-@ECLLOUD_BUILD_ID@-@ECLLOUD_BUILD_TAG@-@ECLLOUD_BUILD_DATE@-@ECLLOUD_BUILD_TIME@.xml</code> results in: <code>annofile-4-default_4_20090220184128-20090220-184128.xml</code>
<code>--emake-annoupload=<0/1></code>		Enables (1) or disables (0=default) annotation file upload on the Cluster Manager. Overrides build class settings.
<code>--emake-autodepend=<0/1></code>		Enables (1) or disables (0=default) the eDepend feature.
<code>--emake-big-file-size= <N></code>		Sets the minimum file size (in bytes) to send through Agent-to-Agent transfers for direct file sharing between hosts. Default=10KB.
<code>--emake-build-label= <label></code>	ECLLOUD_BUILD_LABEL	Sets a customized build label. These labels are literal strings and do not use available tags when defining labels for build classes.
<code>--emake-class=<class></code>	ECLLOUD_BUILD_CLASS	Specifies the build class for the current build. The class must match an existing Cluster Manager class previously created by a user. If this option is not used, or if the class does not match, Electric Make assigns the default class to the build.

Command-line Options	Environment Variables	Description
<code>--emake-clearcase= var1[,var2[,...]]</code>	EMAKE_CLEARCASE	Turns on support for specified ClearCase features—a comma separated list of any of the following values: symlink : symbolic links vobs : per-VOB caching (for speed) rofs : read-only file system
<code>--emake-cluster-timeout= <N></code>		If no Agents are available when the build starts, this option specifies the number of seconds to try to acquire Agents before giving up. The default is -1 (infinite) so the build waits indefinitely for Agents.
<code>--emake-cm=<host>[:port]</code>	EMAKE_CM	Sets the host name (either the IP address or machine name) and port for Cluster Manager.
<code>--emake-collapse=<0/1></code>		Turns history collapsing on or off. When collapsing is enabled, dependencies between a single or several jobs in another <i>make</i> instance are replaced with a serialization between the job and the other Make instance. This action typically results in significant history file size reduction, but may cause some over-serialization. In most builds, this has little or no impact on build time. In some builds, disabling collapsing improves performance at the cost of increased history file size. Default=1 (on)
<code>--emake-disable-pragma= var1[,var2[,...]]</code>		Comma separated list of pragma directives to ignore—can be one or more of: allserial, runlocal, noautodep, or all to disable all pragmas.
<code>--emake-disable-variable -pruning=<0/1></code>		Disables variable table pruning. Default=0 (off)
<code>--emake-emulation=<mode></code>	EMAKE_EMULATION	Sets Make-type emulation to mode. Default emulation type is gmake. You can rename <code>emake.exe</code> to <code>nmake.exe</code> or <code>gmake.exe</code> to change the emulation type for all builds automatically. Legal values: gmake, gmake3.80, gmake3.81, ant, symbian, nmake, nmake7, nmake8, or cygwin.
<code>--emake-emulation-table= <table></code>		Configures default emulation modes for Make programs. TABLE is a comma separated list of NAME=MODE, where NAME is the name of a Make executable and MODE is the emulation mode to use if <code>emake</code> is invoked as NAME.
<code>--emake-exclude-env=var1 [,var2[,...]]</code>	EMAKE_EXCLUDE_ ENV	Specifies which environment variables must not be replicated to the hosts.
<code>--emake-hide-warning= <list></code>	EMAKE_HIDE_ WARNING	Hides one or more Accelerator-generated warning numbers. List is a comma-separated list of numbers you want to hide.
<code>--emake-history= <read/create/merge></code>		Specifies the history mode creation model. Default=merge.
<code>--emake-history-force= <0/1></code>		Honors history mode even if the build fails. Default=1 (on)

Command-line Options	Environment Variables	Description
<code>--emake-historyfile= <path/file></code>		<p>Specifies which history file to use for a specific build. Allows you to change the default name and path for the history file <code>emake.data</code> set automatically by Electric Make.</p> <p>The following macros are available:</p> <p><code>@ECLLOUD_BUILD_ID@</code> expands into the unique eMake build ID.</p> <p><code>@ECLLOUD_BUILD_TAG@</code> expands into the build tag from the Cluster Manager (this is displayed in the Name column on the Build tab in the Cluster Manager UI).</p> <p><code>@ECLLOUD_BUILD_DATE@</code> expands into an 8-digit code that represents the local system date where the build began, in the form <code>YYYYMMDD</code>.</p> <p><code>@ECLLOUD_BUILD_TIME@</code> expands into a 6-digit code that represents the 24-hour local system time where the build began, in the form <code>HHMMSS</code>.</p> <p>Example: <code>--emake-historyfile=historyfile-@ECLLOUD_BUILD_ID@ -@ECLLOUD_BUILD_TAG@-@ECLLOUD_BUILD_DATE@-@ECLLOUD_B UILD_TIME@.xml</code> results in: <code>historyfile-4-default_4_20090220184128-20090220-1 84128.xml</code></p>
<code>--emake-idle-time=<N></code>		Sets the number of seconds (<i>N</i>) before idle Agents are released to the cluster. Default=10
<code>--emake-impersonate- user=<name></code>	<code>ECLLOUD_ IMPERSONATE_USER</code>	<p>Run the build as this Cluster Manager user.</p> <p>Note: Impersonate changes the user recorded by the Cluster Manager and that user's permissions. This option does not affect OS user permissions.</p> <p>For additional information, see the ElectricAccelerator online help topic, "Permissions."</p>
<code>--emake-job-limit=<N></code>		Limits the maximum number of uncommitted jobs to <i>N</i> where 0 means unlimited. Default=0
<code>--emake-ledger= <valuelist></code>	<code>EMAKE_LEDGER</code>	Enables the ledger capability. <i>Valuelist</i> is a comma-separated list that includes one or more of: <code>timestamp</code> , <code>size</code> , <code>command</code> , and <code>nobackup</code> .
<code>--emake-ledgerfile= <path/file></code>	<code>EMAKE_LEDGERFILE</code>	The name of the ledger file. Default= <code>emake.ledger</code>
<code>--emake-maxagents=<N></code>		<p>Limits the number of Agents used by this build. <i>N=0</i> uses all available Agents. Default=0</p> <p>If <code>--emake-maxagents</code> is set lower than the build class, the command line takes precedence. The command line does not take precedence if set to a higher value than the build class.</p> <p>This behavior prevents a user from overriding the upper limit for a class that was set up by the build administrator.</p>
<code>--emake-mem-limit=<N></code>		Controls the amount of memory Electric Make devotes to uncommitted jobs. When the limit is exceeded, Electric Make stops parsing new Make instances. Default=1,000,000,000 (1 GB).
<code>--emake-mergestreams= <0/1></code>	<code>EMAKE_MERGE_ STREAMS</code>	Indicates whether to merge the <code>stdout/stderr</code> output streams, yes (1) or no (0). The default is merge the streams (1). For most situations, this is the correct value. If you re-direct standard output and standard error separately, specify no (0) for this option.
<code>--emake-monitor <hostname/IP>:<port></code>		<p>Sets the hostname/IP and port of the system from where you want to view the ElectricInsight live monitor data.</p> <p>To monitor live build data, you must launch the ElectricInsight live monitor before the build.</p>

Command-line Options	Environment Variables	Description
<code>--emake-pedantic=<0/1></code>		Turns <i>pedantic mode</i> on (1) or off (0=default). When pedantic mode is on, warnings appear when invalid switches are used, or potential problems are identified (for example, rules with no targets or reading from a variable that was not written). When pedantic mode is off, Electric Make ignores irrelevant switches or exits without warning if it encounters unresolvable errors.
<code>--emake-priority=<low normal></code>		Sets the priority for the current build's use of Agents in the cluster. When set to <code>normal</code> (default), the build uses at least the minimum number of Agents set in Cluster Manager. This option overrides the priority associated with a build class but can be used to lower the build priority only .
<code>--emake-read-only=<path></code>	EMAKE_READ_ONLY	All paths starting at the directories specified in <code>--emake-read-only</code> will be marked as read-only file systems when they are accessed on the agent. On UNIX, any attempt to create new files or write to existing files under those directories will fail with EROFS, "Read-only file system". On Windows, it will fail with ERROR_ACCESS_DENIED, "Access is denied".
<code>--emake-readdir-conflicts=<0/1></code>		Explicitly enables conflict detection on directory read operations (commonly called "glob conflicts," which is but one manifestation of the problem). Allowed values are 0 (disabled, the default value) and 1 (enabled). If your build is susceptible to readdir conflict failures, you can enable these checks and get a correct build even if you do not conduct a single-agent build. The resulting history file is identical to a single-agent build result. Though the initial run with this feature may be over-serialized (a consequence of readdir conflicts), a good history file allows builds to go full speed, without conflicts, the next time. You do <i>not</i> want to enable this option all the time. Correct usage: Enable it for one run if you suspect a globbing problem, and then disable it, but use the history file generated by the previous run. Another possible strategy to use if you are not familiar with the build you are building, is to enable the option until you get a successful build, and then disable it after you have a complete, good history file.
<code>--emake-remake-limit=<N></code>		This option defaults to 10. If set to 0, makefiles are not added as goals at all and no remaking occurs. Setting the value to 1 is equivalent to the deprecated <code>--emake-multiemake=0</code> .
<code>--emake-resource=<resource></code>	EMAKE_RESOURCE	The resource requirement for this build.
<code>--emake-root=<path></code>	EMAKE_ROOT	Specifies the Electric Make root directory(s) location. Particularly on Windows, this parameter should not be used to virtualize your tool chain. Tools should be installed on each agent host for performance reasons and to avoid having to virtualize registry parts. The semi-colon is the delimiter between drives. Example: build@winbuild-cm\$ emake --emake-cm=winbuild-cm --emake-emulation=cygwin --emake-root=/c/cygwin/tmp;/c/tmp Starting build: 867 make: Nothing to be done for `foo'. Finished build: 867 Duration: 0:00 (m:s) Cluster availability: 100% In this example, the C: drive is mounted on /c
<code>--emake-showinfo=<0/1></code>		Turns build information reporting on (1=default) or off (0). Information includes build time to completion and average cluster utilization.
<code>--emake-tmpdir=<path></code>	EMAKE_TMPDIR	Sets the Electric Make file temporary directory.

Command-line Options	Environment Variables	Description
Apache Ant-specific Commands		
<code>--emake-emulation=ant</code>		Use this command-line option if you need to emulate Apache Ant for your builds.
	<code>EMAKE_ANT_CONTEXT</code>	Use this variable to specify an explicit context directory for Ant builds. More information about Ant builds is available at “Setting Electric Make Emulation for Ant Builds (all platforms)” on page 3-3.
	<code>EMAKE_ANT_INVOCATION</code>	Use this variable to ensure eMake can find the original Ant script.
UNIX-specific Commands		
<code>--emake-crossmake=<linux/solaris/solarisx86></code>		Use this option to choose a specific operating system, so Cluster Manager will use Linux or Solaris hosts, specifically.
	<code>ECLLOUD_ICONV_LOCALE</code>	Allows you to set the iconv locale. Use <code>iconv -l</code> to list the available locales. Usage Note: If you receive an emake assertion failure that contains information similar to: emake: ../util/StringUtilities.h:406: std::string to_utf8(const std::string&): Assertion `c != iconv_t(-1)' failed. This could mean that your system is missing an internationalization package, or the locale package has a different name for ISO 8859-1. (This particular issue is due to the conversion between 8-bit strings and UTF-8.) Example: Your system recognizes “ISO8859-1”, “ISO_8859-1”, and “ISO-8859-1” only. Set <code>ECLLOUD_ICONV_LOCALE</code> to one of those valid locale names.
Windows-specific Commands		
<code>--emake-case-sensitive=<0/1></code>		Sets case sensitivity for target and pattern name matching. This is inherited by all submakes in the build. The option applies when using gmake/symbian/cygwin emulation modes only; nmake and ant modes are not affected by this option. Default for gmake and cygwin is on (1). Default for symbian mode is off (0). This option replaces <code>--case-sensitive</code> .
<code>--emake-cygwin=<Y N A></code>	<code>EMAKE_CYGWIN</code>	<code>Y</code> =requires <code>cygwin1.dll</code> <code>N</code> =ignore <code>cygwin1.dll</code> <code>A</code> =use <code>cygwin1.dll</code> if available Default= <code>A</code> , if launched from a Cygwin shell, if not, then <code>N</code> . Default= <code>Y</code> , if eMake <code>emulation=cygwin</code> was set.
<code>--emake-ignore-cygwin-mounts=<mounts></code>	<code>EMAKE_IGNORE_CYGWIN_MOUNTS</code>	Comma separated list of Cygwin mounts to ignore. Unless listed, Cygwin mount points are replicated on the Agent.
<code>--emake-reg-limit=<N></code>		Limits the number of registry keys sent automatically for each key. Default=50.
<code>--emake-reg-roots=<path></code>		Sets the registry virtualization path on Windows machines. The syntax is <code>--emake-reg-roots=path</code> [<code>;</code> path]. Do not use this parameter to virtualize your tool chain. Tools must be installed on each agent host for performance reasons and to avoid figuring out which registry parts to virtualize.

Command-line Options	Environment Variables	Description
Debug Commands		
<code>--emake-debug=<level></code>	EMAKE_DEBUG	Sets the local debug log level. For a list of possible values, see the <code>emake --help</code> message.
<code>--emake-logfile=<file></code>	EMAKE_LOGFILE	<p>Sets the debug log file name. Default is <code>stderr</code>.</p> <p>The following macros are available:</p> <ul style="list-style-type: none"> <code>@ECLLOUD_BUILD_ID@</code> expands into the unique eMake build ID. <code>@ECLLOUD_BUILD_TAG@</code> expands into the build tag from the Cluster Manager (this is displayed in the Name column on the Build tab in the Cluster Manager UI). <code>@ECLLOUD_BUILD_DATE@</code> expands into an 8-digit code that represents the local system date where the build began, in the form YYYYMMDD. <code>@ECLLOUD_BUILD_TIME@</code> expands into a 6-digit code that represents the 24-hour local system time where the build began, in the form HHMMSS. <p>Example: <code>--emake-logfile=logfile-@ECLLOUD_BUILD_ID@-@ECLLOUD_BUILD_TAG@-@ECLLOUD_BUILD_DATE@-@ECLLOUD_BUILD_TIME@.xml</code> results in: <code>logfile-4-default_4_20090220184128-20090220-184128.xml</code></p>
<code>--emake-rdebug=<level></code>	EMAKE_RDEBUG	Sets the remote debug log level. For a list of possible values, see the <code>emake --help</code> message.
<code>--emake-rlogdir=<dir></code>	EMAKE_RLOGDIR	Sets the directory for remote debug logs.
Local Agent Commands		
<code>--emake-localagents=Y</code>		<p>Instructs eMake to use any available local agents. Without a Cluster Manager, eMake will use local agents only.</p> <p>If you specify <code>--emake-localagents=y</code> but not <code>--emake-cm</code>, eMake uses local agents only. If you specify <code>--emake-cm</code> but not <code>--emake-localagents=y</code>, eMake uses cluster agents only. If you specify both, eMake uses both, if available.</p>
<code>--emake-maxlocalagents=<N></code>		Limits the number of local agents used. <code>N=0</code> uses all available agents (default 0).
<code>--emake-yield-localagents=<N,T></code>		If using more than <code>N</code> local agents, releases those over <code>N</code> every <code>T</code> seconds so they can be used by another eMake that is looking for local agents.

What is Electrify?

Electrify accelerates builds by parallelizing the build process and distributing build steps across clustered resources.

Known Limitations

- The tool you want to monitor must provide parallel support, such as SCons `-j`, or MSBuild `/m` switch, and so on.
- Electrify does not provide any of Electric Make's dependency detection or correction features. The build tools you use with Electrify must be capable of accurate parallel execution on their own.
- The information written into annotation is more limited with Electrify than what is provided by Electric Make. Electrify annotation provides information only on the commands executed on the cluster, including command lines, file usage, and raw command output. Electrify does not provide information about dependencies, job relationships, targets, or other logical build structure data.

Recommendations

Electric Cloud has evaluated the following build tools for use with Electrify.

- SCons - Electric Cloud recommends using SCons with Electrify. Using SCons does not have any known limitations.
- MSBuild - Using MSBuild with Electrify is supported for Visual Basic and C# projects.
- Devenv - Using Devenv with Electrify is supported for C++ projects.
- Ant - Electric Cloud does not currently recommend using Ant with Electrify.

Electrify as Part of the Build Process

The following sections describe how to run builds using Electrify.

Important reminders about MSBuild

- For MSBuild 3.5 (included in Visual Studio 2008) and MSBuild 4.0 (included in Visual Studio 2010) :
MSBuild 3.5 requires special configuration when distributing calls to the `vbc.exe` and `csc.exe` compilers. Under MSBuild 3.5, these tools use files written to the TMP directory. Ensure that the directory specified by the TMP environment variable is in the eMake root, or these tools will not operate correctly. If you wish to change the value of the TMP environment variable in order to locate it within the eMake root, you can do so by using the following command: `set "TMP=your path"`
Make sure that the entire 'TMP=' expression is enclosed in double-quotes if your TMP directory path contains any space characters.
- By default, MSBuild does not terminate its worker processes, but keeps them resident for the next build. This behavior can interfere with the correct operation of ElectricAccelerator. To avoid this, you must specify `/nr:false` so that `msbuild.exe` child processes exit when the build finishes.

- Information regarding use of the `/maxcpucount:[number]` switch (where *number* is the count of worker threads):
 - You can use `/m:[number]` as the short form.
 - Set `/m:[number]` to the number of agents that you want to use.
 - If you do not set `/m:[number]`, it is automatically set to the number of CPUs detected locally.

Important reminders about Devenv

- Devenv does not use the `/m` switch. To set the number of agents used in the build, do the following:
 1. From Devenv, select Tools > Options > Project and Solutions > Build and Run
 2. This displays a dialog box with the “maximum number of parallel project builds” option.

Note: Because the ElectricAccelerator Solution Support Add-in requires the “maximum number of parallel project builds” to be 1, you must use either Electrify *or* the add-in.

Running Electrify on Windows

Example

```
electrify [args] other tools' command line
```

Note: You can use Devenv to distribute `csc.exe` for Visual Studio 2010, but not for Visual Studio 2008 (because Devenv does not launch `csc.exe`).

Visual Studio C++ projects using MSBuild

```
electrify --emake-cm=<CM name> --electrify-remote=cl.exe;lib.exe;link.exe
--emake-root=<project root directory>%temp% msbuild /m:4 /nr:false /t:build
<solution name>.sln
```

Visual Studio C++ projects using Devenv

```
electrify --emake-cm=<CM name> --electrify-remote=cl.exe;lib.exe;link.exe devenv
<solution name>.sln /build
```

Running Electrify on Linux

Example

```
electrify [args] other tools' command line
```

SCons example

```
electrify --emake-cm=<CM name> --electrify-remote=g++:gcc:ranlib:ar scons -j 4
```

GNU Make C++ example

```
electrify --emake-cm=<CM name> --electrify-remote=g++:gcc:<any other tools used in
the build> make -j 4 -f makefile
```

Important reminders

- Ensure `cl.exe`, `link.exe`, and so on, are those of Microsoft Visual Studio. The wrapper application may have changed them to its version.
- Though all of the usual eMake arguments are available, Electrify uses only a subset of them.
- You must use 64-bit `electrify` and `electrifymon` on 64-bit Windows. Using 32-bit versions results in an error message and then the program exits. When you specify the 64-bit `electrify` and `electrifymon` directories in `PATH`, they must *precede* the 32-bit directories.
- When using Visual Studio 2010, building with `devenv` or `msbuild` with `electrifymon` errors out with a file/application not found error. To complete the build, use the appropriate workaround:
 - For `msbuild`, add `tracker.exe` (`--electrify-not-remote=msbuild.exe;tracker.exe`)
 - For `devenv`, add `msbuild.exe` and `tracker.exe` (`--electrify-not-remote=devenv.exe;devenv.com;msbuild.exe;tracker.exe`)
- On UNIX platforms, `electrifymon` must locate `electrifymon.so` so it can tell monitored programs to load the monitoring library that reports back to `electrifymon`. By default, `electrifymon` looks in the following locations:

Platform	32-bit	64-bit
Linux	/opt/ecloud/i686_Linux/lib	/opt/ecloud/i686_Linux/64/lib
Solaris (SPARC)	/opt/ecloud/sun4u_SunOS/lib	/opt/ecloud/sun4u_SunOS/64/lib
Solaris (x86)	/opt/ecloud/i686_SunOS.5.10/lib	/opt/ecloud/i686_SunOS.5.10/64/lib

You can override these locations using the following environment variables: `ELECTRIFYMON32DIR` and `ELECTRIFYMON64DIR`.

- On Linux, you can change the mode that Electrify uses for monitoring. Use `--emake-electrify=<mode>`, where mode can be: `preload` for LD_PRELOAD intercept, or `trace` for ptrace.

Electrify arguments

All arguments are optional.

<code>--electrify-remote=x;y</code>	<p><code>x</code> and <code>y</code> are commands that are distributed to the cluster. Use the command's full name, such as <code>cl.exe</code>, <code>link.exe</code>, <code>gcc.exe</code>, without the path. The name is case insensitive. In a Cygwin environment, you can use ':' (colon) instead of ';' (semicolon). Limited to 2048 characters Environment variable: <code>ELECTRIFY_REMOTE</code> <code>devenv</code> for C# projects is not supported.</p>
<code>--electrify-not-remote=x;y</code>	<p><code>x</code> and <code>y</code> are commands that <i>are not</i> distributed to the cluster. Use the command's full name, such as <code>cl.exe</code>, <code>link.exe</code>, <code>gcc.exe</code>, without the path. The name is case insensitive. In a Cygwin environment, you can use ':' (colon) instead of ';' (semicolon). <code>--electrify-not-remote</code> and <code>--electrify-remote</code> are mutually exclusive. If you use <code>--electrify-not-remote</code>, all other tools' command lines are executed remotely, by default. Generally, this is not what you want, so to do this, you must add a command to this list. <code>devenv</code> for C# projects is not supported.</p>

<code>--electrify-not-intercept=x;y</code>	xxx and yyy are commands that you do <i>not</i> want to be monitored, meaning the monitor process does not inject a dll to them and their child processes, so they will <i>not</i> be distributed. Environment variable: ELECTRIFY_NOT_INTERCEPT
<code>--electrify-log=fullpath</code>	fullpath is the path of the file you want to log. This logs all process creation and interception information. Environment variable: ELECTRIFY_LOG
<code>--electrify-statistics=y</code>	Specifying this argument with y means the statistics of distributed and undistributed commands will be collected and printed. This argument implicitly uses the log file. If the log file is not specified by <code>--electrify-log</code> , it automatically generates a temporary file in a temporary path and deletes it when <code>electrifymon</code> is about to exit.
<code>--electrify-localfile=x</code>	(Windows only) Integrates local file access (create, rename, and so on) by locally running tools with the remote file system. You can set x to two different flags: nt or y. Set nt if you want to monitor undocumented low-level file access Nt functions. This monitors the following functions: NtCreateFile, NtDeleteFile, NtClose, NtWriteFile, and NtSetInformationFile. Though this includes only five functions, their functionality is rich, so this selection includes nearly all scenarios where the local file system changes. Set y to monitor documented Win32 APIs for file access. This monitors the following Win32 APIs: CreateFileW, CreateDirectoryA, CreateDirectoryExA, CreateDirectoryExW, CreateDirectoryW, DeleteFileA, DeleteFileW, MoveFileA, MoveFileExA, MoveFileExW, MoveFileW, RemoveDirectoryA, RemoveDirectoryW, SetFileAttributesA, and SetFileAttributesW. Though there are many functions, their functionality is less than Nt functions, particularly because some tools such as Cygwin cp.exe use NtCreateFile and so on. In general, use the y flag for testing purposes only.

Additional Information

Selecting commands to parallelize

A large portion of the build process acceleration will be achieved through parallelizing a small number of specific commands, such as compiling and linking. Expending additional effort to select and parallelize many different additional commands may not result in a significant amount of further acceleration.

Note: If a file is created or modified by one or more parallelized commands, then you should parallelize all commands that use that file.

Using Electrify with GNU Make

If you intend to use Electrify with GNU Make, Electric Cloud recommends using eMake instead. eMake provides superior performance and correctness and full annotation information.

How an Electrify build differs from an eMake build

An important difference between an eMake build and an Electrify build is what portion of the build activity occurs remotely versus locally. In an eMake build, effectively all build activity (except “#pragma runlocal” jobs) takes place on the cluster, where the EFS is used to monitor filesystem accesses and propagate changes made by one job to other jobs in the build.

In an Electrify build, a greater portion of the build activity takes place on the local system—at the very least, the build process itself (such as SCons) runs locally. Typically, filesystem modifications made by processes running locally are

“invisible” to Electrify, and therefore to processes running on the cluster—just as “#pragma runlocal” jobs may make changes that are invisible to eMake. Electrifymon provides a means to update the virtual filesystem state in Electrify in response to filesystem modifications made by local processes.

If you know that a build will not run any processes locally that modify filesystem, you need not use electrifymon when invoking Electrify. However, some build tools will themselves make changes to the filesystem (for example, when SCons employs its build-avoidance mechanism by copying a previously built object in lieu of invoking the compiler), so the safest choice is to use electrifymon to start.

In addition to filesystem monitoring, electrifymon on Windows provides a sophisticated mechanism for intercepting processes invocations and determining which processes to distribute to the cluster. On Linux, process interception is handled by the explicit use of proxy commands.

ElectricAccelerator Job Types

This appendix discusses ElectricAccelerator job types and their use to help you understand Accelerator performance and behavior. Content is adapted from the “ElectricAccelerator Job Compendium” blog post on <http://blog.melski.net/>.

Overview

ElectricAccelerator’s fundamental unit of work is the *job*. Most of the time, you can think of a job as all commands that must be run in order to create or update a single build output, but in truth that describes only one *type* of job. There are actually several different job types, each with a distinct purpose in the build structure and the way Accelerator executes the build. You can determine the job type from the `<job>` tag’s *type* attribute in Accelerator annotation files.

ElectricMake Jobs

To make descriptions more concrete, this chapter includes a simple reference build that uses each of the most common job types, so you can see exactly how jobs relate to a real build. The reference build makefile:

```
prog:
    @$(MAKE) sub/prog
    @cp sub/prog ./prog

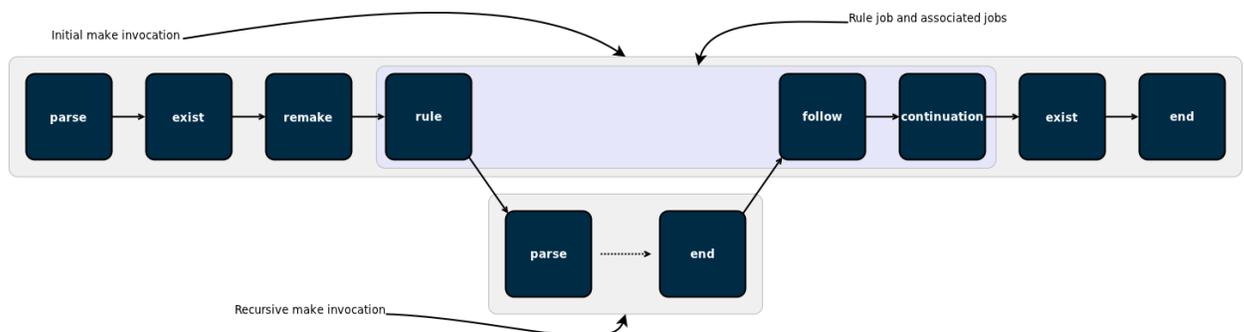
sub/prog: sub/main.c
    @cat $< > $@

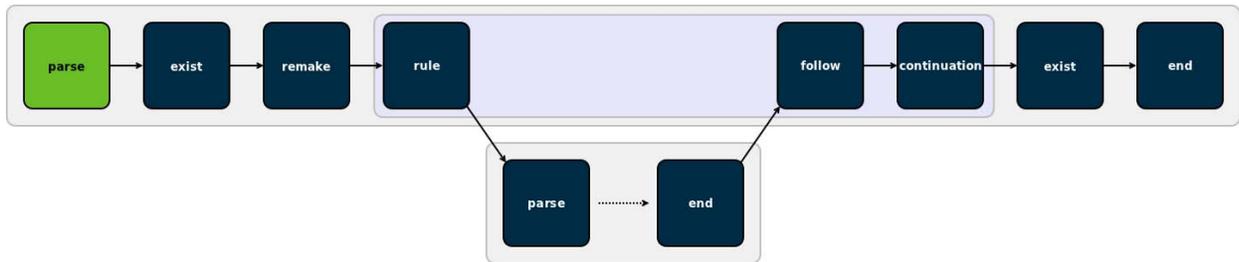
setup:
    rm -rf prog sub
    mkdir sub
    echo "int main() { return 0; }" > sub/main.c
    touch prog2
```

To run the build:

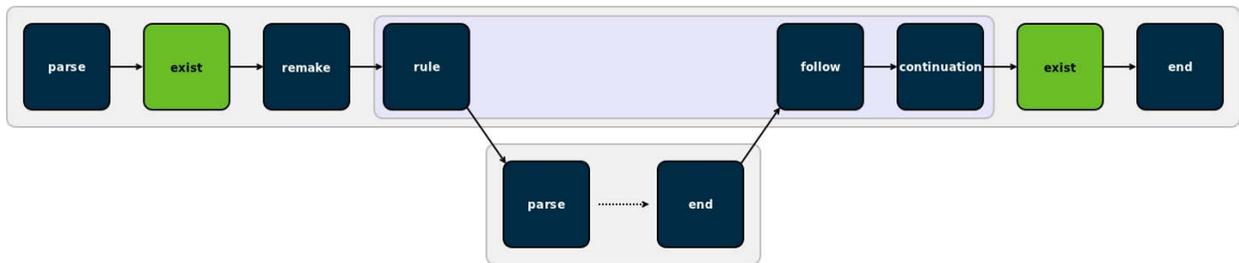
1. Run `emake setup`. This creates the files and directory structure the build needs.
2. Run `emake --emake-maxagents=1 --emake-annodetail=basic --emake-annofile=emake.xml prog prog2`. This produces a build with thirteen different jobs, in two different make instances.

The following image is an overview of how those jobs fit together.

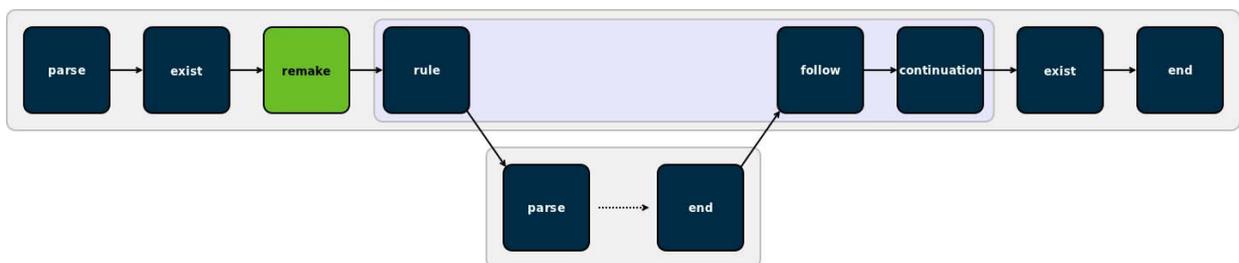


parse

The first job in any make invocation (and therefore the first job in any eMake build) is a *parse* job, during which eMake reads and interprets the makefiles used in that make instance. The parse job’s output is a list of all jobs in the make instance, along with a list of targets that must be built, the commands to build those targets, and the relationships between them.

exist

Existence jobs (marked as type “exist” in annotation) check for the existence of makefiles and command-line goals for which no rule was found. In the reference build, you can see an existence job in the top-level make for the makefile itself, as well as one for the file prog2, which has no rule in the makefile.

remake

Remake jobs are at the center of eMake’s emulation of GNU Make makefile remaking feature. In a remake job, eMake checks every makefile that was read during the parse job for two things:

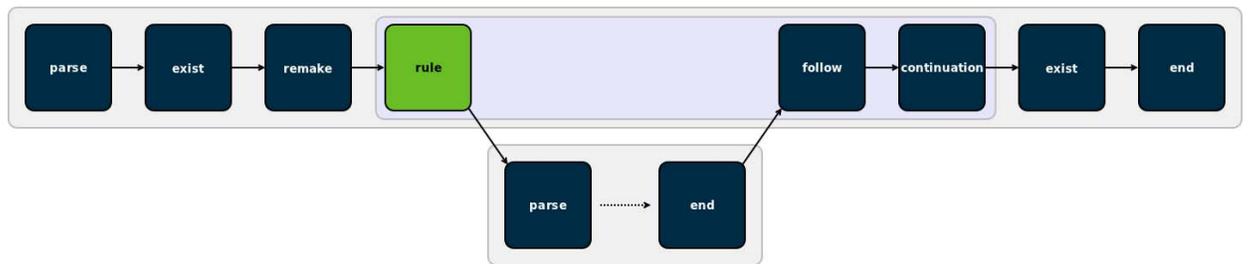
- is there a rule to rebuild the makefile
- was the makefile actually rebuilt (because it was out-of-date)

If *any* makefile was rebuilt, eMake restarts the make instance—all the way back to the parse job.

Because makefile remaking is a GNU Make-specific feature, you only see remake jobs when eMake emulates GNU Make—not when it emulates NMAKE.

eMake overloads the use of the `<failed>` tag inside a remake job to indicate not failure, but whether or not the job determined that the make instance should be restarted. If yes, the remake job includes a `<failed>` tag with the code attribute set to 1; if not, the remake job does not have a `<failed>` tag.

rule

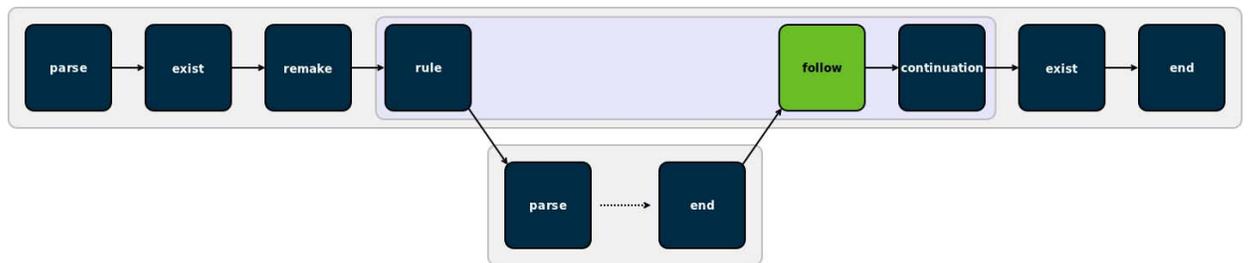


Rule jobs are the real workhorses of a build. Each rule job encapsulates the commands needed to update one target in the build—literally the body of a rule from a makefile—and a rule always has an associated output target (or targets), which is identified by the *name* attribute of the job tag in annotation. In the reference build, the rule job in the top-level make instance corresponds to this rule in the makefile:

```
prog:
    @$(MAKE) sub/prog
    @cp sub/prog ./prog
```

In the job's annotation, only `$(MAKE) sub/prog` is actually in the rule job—because the `cp sub/prog ./prog` was automatically split into a *continuation* job when eMake detected the recursive make invocation.

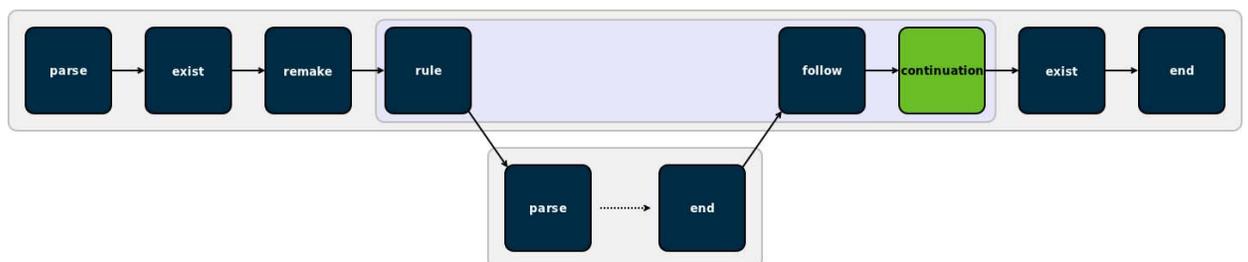
follow



A *follow* job serves two purposes. First, it is a connection point that allows eMake to tie the end job of a recursive make invocation back into the ordered list of jobs in the parent make. Second, it is the means by which eMake propagates the error status of a recursive make to the parent make.

Follow jobs always have an associated rule or continuation job, identified by the *partof* attribute of the job tag in annotation. That is the job that spawned the recursive make with which the follow job is associated. Not all rule and continuation jobs have an associated follow job—follow jobs occur only when a recursive make was invoked.

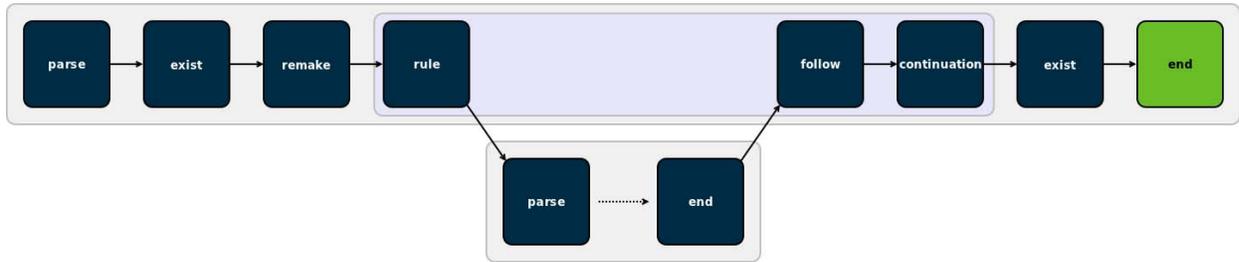
continuation



A *continuation* job represents “leftover” commands that follow a recursive make invocation in a rule (or continuation) job. In the reference build, eMake creates a continuation job for `cp sub/prog ./prog` because that command follows a recursive make invocation. Continuation jobs, like follow jobs, are always associated with a rule (or another continuation) job, identified by the *partof* attribute of the job tag in annotation.

The extra commands are split into a separate job to allow eMake to easily specify the serial order of those commands relative to the jobs in the recursive make—the commands in the continuation come after the jobs in the recursive make.

end



The last job in every make instance (and therefore the last job of every emake build) is an *end* job. End jobs exist primarily to handle end-of-the-make cleanup, such as removing intermediate targets or temporary inline files that were created while executing the other jobs in the make.

subbuild (not pictured)

Subbuild jobs, which are not used in the reference build, are part of eMake’s subbuild feature. Subbuild jobs are a mechanism to inject a recursive make invocation into the build, just as you would get if you had a rule job with a `$(MAKE)` command, but without the tedium of actually running that command.

Electrify Jobs

Electrify uses much of the same underlying machinery that eMake does, but Electrify has its own set of job types for its particular needs.

alpha

Every Electrify build starts with an *alpha* job, which is a trivial placeholder marking the start of the list of jobs.

external

External jobs represent commands invoked by the electrified build tool that are distributed to the cluster. These are similar to eMake’s rule jobs, except they only run a single command, while rule jobs may run any number of commands.

update

Update jobs represent filesystem modifications made by commands invoked by the electrified build tool but not distributed to the cluster. These modifications are detected with the aid of `electrifymon`.

omega

Every Electrify build ends with an *omega* job, which, like the alpha job, is a trivial placeholder.

Electric Make Debug Log Levels

This appendix discusses Electric Make debug log levels. Content is adapted from the “Electric Make debug log levels” blog post on <http://blog.melski.net/>.

Overview

Often when analyzing builds executed with Electric Make, all of the information you need is in the annotation file—an easily digested XML file containing data such as the relationships between the jobs, the commands run, and the timing of each job. But sometimes you need more detail, and that is where the eMake debug log is useful.

Enabling eMake Debug Logging

To enable eMake debug logging, specify this pair of command-line arguments:

```
--emake-debug=value
```

This argument specifies the types of debug logging to enable. Provide a set of single-letter values, such as “jng”.

```
--emake-logfile=path
```

This argument specifies the location of the debug log.

eMake Debug Log Level Descriptions

Available log levels:

a: agent allocation	l: ledger
c: cache	m: memory
e: environment	n: node
f: filesystem	o: parse output
g: profiling	p: parse
h: history	r: parse relocation
j: job	s: subbuild
L: nmake lexer	Y: security

Disclaimer: eMake debug logs are intended for use by Electric Cloud engineering and support staff. Debug logging contents and availability are subject to change in any release, for any or no reason. The information in this article refers to ElectricAccelerator 6.0.

a: agent allocation

Agent allocation logging provides detailed information about eMake's attempts to procure agents from the Cluster Manager during the build. If you think eMake may be stalled trying to acquire agents, allocation logging will help to understand what is happening.

c: cache

Cache logging records details about the filesystem cache used by eMake to accelerate parse jobs in cluster builds. For example, it logs when a directory's contents are added to the cache, and the result of lookups in the cache. Because it is only used during remote parse jobs, you must use it with the `--emake-rdebug=value` option. Use cache logging if you suspect a problem with the *cached local filesystem*.

e: environment

Environment logging augments [node](#) logging with a dump of the entire environment block for every job as it is sent to an agent. Normally this is omitted because it is quite verbose (could be as much as 32 KB per job). Generally, it is better to use env-level annotation, which is more compact and easier to parse.

f: filesystem

Filesystem logging records numerous details about eMake's interaction with its versioned filesystem data structure. In particular, it logs every time that eMake looks up a file (when doing up-to-date checks, for example), and it logs every update to the versioned file system caused by file usage during the build's execution. This level of logging is very verbose, so it is not usually enabled. It is most often used when diagnosing issues related to the versioned filesystem and conflicts.

g: profiling

Profiling logging is one of the easiest to interpret and most useful types of debug logging. When enabled, eMake will emit hundreds of performance metrics at the end of the build. This is a very lightweight logging level, and is safe (even advisable) to enable for all builds.

h: history

History logging prints messages related to the data tracked in the eMake history file—both filesystem dependencies and autodep information. When history logging is enabled, eMake will print a message every time a dependency is added to the history file, and it will print information about the files checked during up-to-date checks based on autodep data. Enable history logging if you suspect a problem with autodep behavior.

j: job

Job logging prints minimal messages related to the creation and execution of jobs. For each job you will see a message when it starts running, when it finishes running, and when eMake checks the job for conflicts. If there is a conflict in the job, you will see a message about that, too. If you just want a general overview of how the build is progressing, j-level logging is a good choice.

L: nmake lexer

eMake uses a generated parser to process portions of NMAKE makefiles. Lexer debug logging enables the debug logging in that generated code. This is generally not useful to end-users because it is too low-level.

I: ledger

Ledger debug logging prints information about build decisions based on data in the ledger file, as well as updates made to the ledger file. Enable it if you believe the ledger is not functioning correctly.

m: memory

When memory logging is enabled, eMake prints memory usage metrics to the debug log once per second. This includes the total process memory usage as well as current and peak memory usage grouped into several "buckets" that correspond to various types of data in eMake. For example, the "Operation" bucket indicates the amount of memory used to store file operations; the "Variable" bucket is the amount of memory used for makefile variables. This is most useful when you are experiencing an out-of-memory failure in eMake because it can provide guidance about how memory is being utilized during the build, and how quickly it is growing.

n: node

Node logging prints detailed information about all messages between eMake and the agents, including filesystem data and commands executed. Together with job logging, this can give a very comprehensive picture of the behavior of a build. However, node logging is extremely verbose, so enable it only when you are chasing a specific problem.

o: parse output

Parse output logging instructs eMake to preserve the raw result of parsing a makefile. The result is a binary file containing information about all targets, rules, dependencies, and variables extracted from makefiles read during a parse job. This can be useful when investigating parser incompatibility issues and scheduling issues (for example, if a rule is not being scheduled for execution when you expect). Note that this debug level only makes sense when parsing, which means you must specify it in the `--emake-rdebug` option. The parse results will be saved in the `--emake-rlogdir` directory, named as `parse_jobid.out`. Note that the directory may be on the local disk of the remote nodes, depending on the value you specify.

p: parse

Parse debug logging prints extremely detailed information about the reading and interpretation of makefiles during a parse job. This is most useful when investigating parser compatibility issues. This output is very verbose, so enable it only when you are pursuing a specific problem. Like parse output logging, this debug level only makes sense during parsing, which means you must specify it in the `--emake-rdebug` option. The parse log files will be saved in the `--emake-rlogdir` directory, named as `parse_jobid.dlog`. Note that the directory may be on the local disk of the remote nodes, depending on the value you specify.

r: parse relocation

Parse relocation logging prints low-level information about the process of transmitting parse result data to eMake at the end of a parse job. It is used only internally when the parse result format is being extended, so is unlikely to be of interest to end-users.

s: subbuild

Subbuild logging prints details about decisions made while using the eMake subbuild feature. Enable it if you believe that the subbuild feature is not working correctly.

Y: authentication

Authentication logging is a subset of node logging that prints only those messages related to authenticating eMake to agents and vice-versa. Enable this debug level, if you are having problems using the authentication feature.

