

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

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



Outline

- Introduction
 - build process
 - distribution
 - faults
 - fault tolerance
 - our environment
- Distributed-Build Tools
- Our Techniques
 - external
 - internal
- Conclusion

Introduction (build process)

- The computation that transforms a product's source files into those files needed to use the product.
- It typically involves compilation and linking, but it may be much more complex, especially for embedded systems.
- It may also require a large amount of time (e.g., hours) and other resources.

Introduction (distribution)

- One way to reduce the time required by a build is to distribute the computation across a network of multiple computers.
- Some available build tools help automate this distribution.
- One of our requirements, not satisfied by available build tools, is that some computers in our network may temporarily or permanently be unable to successfully execute commands.

Introduction (faults)

- no network connection
- inactive network services
- insufficient network bandwidth
- mismatch of architecture, operating system, or other tools
- missing filesystems
- insufficient computation resources
- manual exclusion
- build-step error

Introduction (fault tolerance)

- We have identified a set of detectable faults, which can occur on a remote computer during a build.
- If one or more faults are detected during a build step, that step is retried, on a different computer.
- Three retries seems reasonable, in our environment.

Introduction (our environment)

- HP's firmware labs for LaserJet printers.
- Operating systems: HP-UX 10.20, HP-UX 11.11, Red Hat Linux 9, and Novell Linux Distribution 9.
- SCM system: Rational's ClearCase, MultiSite, and ClearMake.
- Network:
 - 1 Gbit to servers (about 10)
 - 100 Mbit to clients (about 1000)
 - NFS and NIS
 - automounted home directories
- Code: 2M lines of C, C++, Java, IDL, and XML.
- Builds: distributed and heterogeneous.

Distributed-Build Tools

- Rational ClearMake (1992)
- GNU Make with Customs (1987)
- GNU Make with PVM (1994)
- GNU Make with Winner (1998)
- PMake with or without Customs (1994)
- Sun Microsystem's DMake (1999)
- distcc (2003)
- Electric Cloud (2002)
- Vesta (1993, 2001)

Our Techniques

- We extend the capabilities of our distributed-build tool in two ways:
 - External: outside of the build process.
 - Internal: part of the build process.

External Techniques (basic)

- As much as various operating systems allow, we ensure a consistent client environment:
 - NFS/NIS for repository and workspace sharing
 - automounter for home-directory sharing
 - NTP for synchronized clocks
 - remsh, rsh, and rexec for remote-command execution

External Techniques (crawler)

- With a thousand clients, we cannot possibly hope to manually track additions, deletions, and moves.
- Even if a client is responsive, it may not be “healthy”.
- The “crawler” is a daemon intended to provide names of healthy clients for remote execution.
- The crawler accepts, as input, a file containing the names of all of our clients. This file is only updated when a new computer is installed or an old one is decommissioned. The crawler maintains, as output, a set of files, each of which contains names of homogeneous healthy clients.

External Techniques (crawler tests: 1 of 2)

- While iterating through its input file, the crawler examines each client:
 - It must not have been manually excluded from participation.
 - It must reply to several “ping” messages, without packet loss.
 - It must accept and execute a remote command.
 - Its SCM daemons must be active and the associated filesystems must be mounted and accessible.
 - It must have sufficient communication bandwidth to the SCM servers.
 - Its automounter daemon must be active.
 - It must have sufficient computational power, according to its make and model.

External Techniques (crawler tests: 2 of 2)

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

External Techniques (watcher)

- While examining a client, the crawler may “get stuck”.
- Another daemon, named the “watcher”, detects this condition, kills the stuck crawler, and starts a new one.
- The watcher cannot get stuck.

Internal Techniques (portability)

- We try to keep the build process as portable as possible:
 - Use the same build tool on all clients.
 - Avoid esoteric build-tool features, which may vary between versions.
 - Prefer scripts over binaries.
 - Build binaries such that they execute on as many versions of an operating system as possible.
 - Compute the path to a nonportable binary.

Internal Techniques (sequential/concurrent local/remote)

- Not all build steps should be performed concurrently and remotely.
 - Example 1: A directory containing symbolic links to all interface files should be constructed sequentially and locally, to avoid collisions.
 - Example 2: Multiple executable files should be created concurrently and locally, because linking object files across a network is slow.

Internal Techniques (derived-object sharing and protection: 1 of 2)

- A derived object is a file created by a build process.
- Some derived objects are very sharable (e.g., foo.h):
 - any client
 - any product
 - any variant
- Other derived objects are less sharable (e.g., bar.o):
 - HP-UX/PA-RISC versus Redhat9/x86
 - monochrome versus color
 - debug versus no debug

Internal Techniques (derived-object sharing and protection: 2 of 2)

- A derived object can be shared, between builds, in two ways:
 - It may already exist in the workspace. This kind of sharing is protected by building it in a directory named according to how it is built.
 - It may be available in another workspace. This kind of sharing is protected by storing how it is built in its configuration record.
- A build process must ensure that only appropriate derived-object sharing occurs and that build commands are distributed to appropriate computers.

Internal Techniques (crawler list)

- Recall that the crawler maintains a set of files, each containing names of homogeneous healthy clients.
- The build process reads one of these files, based on its operating-system name, operating-system version, and SCM-tool version.
- The file contains a list of hostnames, upon which remote commands can be executed.
- A user may filter and/or augment the list.
- Access to this shared file must be synchronized (e.g., a copy-and-compare loop).

Internal Techniques (monitoring, logging, and retry)

- The crawler tries to provide healthy clients, but a client might have undetected or recent problems.
- Typically, a build tool simply examines each build command's exit code. A non-zero code causes the build tool itself to exit.
- We try to detect other failures and react to them.
- A failed build command is retried several times, on different clients, before the build process aborts.
- A failed command is retried using the latest crawler list, since the current one might be five or ten minutes old.
- Retries are also logged, allowing us to investigate unhealthy workstations.

Conclusion

- Distributed builds are worth the extra trouble.
- Distributed-build tools should:
 - scale to large environments
 - be fault tolerant
 - provide failure/retry statistics
 - only retry the step that failed