Automated Build and Test System

Chris Leger, David Maze and Shelley Duvall
MIT 6.824, Fall 2002
{clleger, dmaze, maduvall}@mit.edu

Abstract

Large software systems under development undergo frequent change. Most such projects have automated build systems, and they frequently have a regression test system to confirm correct operation of the system. However, these systems are generally poorly integrated, and when a nightly build breaks, it becomes the responsibility of a human being to locate the offending change.

This paper describes an integrated build, test, and blame system. Using information available from the version control system, it notifies developers if a change they committed caused a build or regression test to fail. The system also provides a mechanism for developers to track the most recent version of the system known to build or to pass its tests. Furthermore, the system extensively uses interfaces to allow easy extensions to and replacements within the system.

1 Introduction

As software projects grow, the number of tools required to manage them also increases. Projects containing more than a couple of files generally use *make* to describe the build process. As projects grow to span multiple directories, they begin to use intricate combinations of makefiles, with only the ordering of compilation describing possibly complex dependencies between directories. Similarly, simplistic revision-control systems such as SCCS or RCS have given way to larger systems such as CVS that have a concept of multiple directories. Project developers are simultaneously faced with more complex projects being developed and more complex tools to manage the project.

Medium-to-large software projects frequently have some automation to help developers construct a correct system. For instance, the GNU C compiler has a system that periodically runs blind builds and a set of tests.[6] However, when a build or test fails, it is generally up to an administrator to determine which of potentially dozens of check-ins in the previous day "broke the build", and ask the developer responsible to fix it. Furthermore, other developers must find an older version of the project for development, wasting valuable resources.

We propose designing and building a system which can use information readily available from the version control system, build system, and the source language (i.e. dependencies) to perform blame assignment. Through its interaction with the version control system, our system can also provide developers with the most recent workable version of the project. (Here workable means at least compiling, but other specifications may also be met, such as succeeding on a certain test.) The system can thus do blame determination for the project administrator while providing developers with workable versions of the project, reducing the number of mechanical jobs that waste development resources.

1.1 Goals

The goals of the system are:

- *Locate Errors*: When a new build or test error occurs (i.e. an error that did not occur on the last build), the system should determine the minimal set of developers that could have caused the error. After determining possible culprits, the system should notify the suspects of their probable guilt.

- *Store Data*: The system should store data from builds and tests in an easily accessible database. Also, the system should store the commit date of the last compiling version of a project and the last version that succeeded on any specified test.
- *Be Easily Extensible*: The system should contain easily extensible interfaces so that it can work with a variety of revision control systems, build systems, and tests. Also, algorithmic replacements (i.e. module replacements) should be easy.
- *Distribute Tests*: The system should present an easy interface to robustly distribute tests across idle machines. This should speed project testing and thereby extend error resolution capabilities.
- *Require No Administration After Setup*: Once setup, the system should not require any additional administration to perform its task.

1.2 Challenges

This sort of system faces many challenges in its design. These include:

- Weak version control systems. Often programmers make a single change that encompasses several files. Popular version control systems such as RCS, SCCS, and CVS only manage changes on a per-file basis; an aggressive automated system might notice a "broken" build which is simply in between a set of related changes. Other source-control systems such as Perforce have the concept of an atomic "change" that encompasses several files at once.
- Dependency determination. Many popular languages (e.g. C, C++) do not include any intrinsic notion of dependency. Nothing directly relates a function call in one file to its definition in another file; this can be obfuscated further by the introduction of static or shared libraries. Other languages (e.g. Java) do have explicit references to other source files, but even here tracking dependencies can be tricky: one file might import package.BaseClass and use objects of this type, but a change in other.package.DerivedClass – a class not explicitly referenced by the code – might result in problems.

To some extent, these problems can be alleviated by clever use of build tools. It is possible to directly determine the functions defined and called in a particular object file; for example, the GNU C compiler can tell which other source files a given source file directly includes. Similarly, a build system for Java can accumulate a listing of every class used in a program and build a class inheritance graph. However, it remains difficult to fully describe the functions that code in a given file might call: use of C function pointers and interfaces and class inheritance in Java mean that one file might have an indirect dependency on another file, even in the absence of a lexical reference.

1.3 Design Overview

The system described in this paper targets projects consisting primarily of Java code, using the CVS revision control system.[2] Java compilers such as Jikes[3] can determine the direct file dependencies of a particular source file, in the same way that gcc can for C code. We choose Java because indirect dependencies only happen through class derivation, and thus only affect a limited number of files. Since the set of potential dependencies for a given file can be statically determined, using Java also allows the native build system to be ignored. CVS is mostly chosen for its popularity, though it also has straightforward hooks to cause some action to be performed when a file is committed.

The build system should not run on developers' local workstations; developers tend to be too protective of their personal compute cycles to embrace such a system. Instead, a dedicated machine exists to build the system. Tests are run in parallel on a set of available machines, since they are generally completely independent of each other.

A long-running server process lives on the build machine. On commit, the server is contacted and informed that something in the source tree has changed. If a build process is not running, one is started; otherwise, the server remembers that a commit has occurred, and when the current build ends, it starts a new build process to handle the intervening changes.

The build process ensures that it has the most recent version of the CVS tree. Following a successful build, it instantiates a tester which runs a number of JUnit tests. If a different set of tests succeed or fail, notification about this is sent to the author(s) of one (or more) of the intervening changes.

2 Dependency

For the purposes of determining the potential extent of a change, one file depends on another file if code from the first file could cause code in the second file to be executed. This is a broader definition of dependency than the traditional definition, in which one file depends on another if it directly references it in its code.

2.1 Direct Dependency

One file *directly depends* on another if there is a lexical reference to the second file in the first. In a procedural language such as C, a source file depends on any header file it includes, as well as any other source files defining functions the file uses. In object-oriented languages such as Java, one class directly depends on any class or interface named in the code.

It is generally straightforward to determine the direct dependencies of a file. Most C compilers have an option to determine this directly; even if it were not available, a straightforward analysis of `#include` directives and object files would yield the desired information. Similarly, the Java compiler *jikes* has an option to find the direct dependencies of a Java input file.

2.2 Indirect Dependency

A file *indirectly depends* on another file if the criteria for dependence are satisfied, but there is not a direct dependence. This generally happens using function pointers in C, or using virtual functions in an object-oriented language.

```
A.c:  void (*fn)(void) = get_function_pointer();
      fn();
B.c:  void (*fn)(void) get_function_pointer(void) { return the_function; }
C.c:  void the_function(void) { }
```

Figure 1: Indirect dependency in C

Figure 1 shows how indirect dependency can occur between files in C. File `A.c` directly depends on `B.c`, since it calls the function `get_function_pointer()`. However, the operation of `A.c` also depends on the function pointed to by the function pointer `fn`. Analysis reveals that `fn` can point to `the_function()` in `C.c`, resulting in an indirect dependency.[8]

```
A.java:           InterfaceB b = a.getB();
                  b.doIt();
InterfaceB.java:  public interface InterfaceB {
                    void doIt();
                  };
ClassC.java:      public class ClassC implements InterfaceB {
                    void doIt() { }
                  };
```

Figure 2: Indirect dependency in Java

The situation in Java is somewhat simpler, since Java does not have function pointers. Figure 2 shows how the use of interfaces can cause an indirect dependency. The `getB()` method returns an arbitrary object implementing `InterfaceB`. This might be an object of `ClassC`, even though there is not a direct reference to `ClassC` in the code; thus, `A` indirectly depends on `ClassC`.

3 Design

The system consists of three main subsections: the server, the build component, and the test component. The server is the interface to our system. As such, it is responsible for answering user queries and scheduling

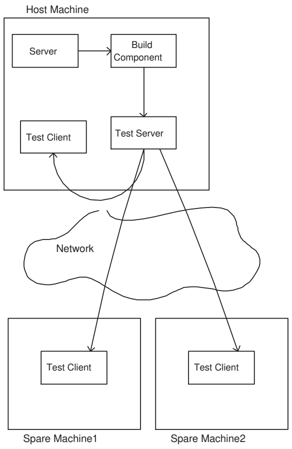

Figure 3: The server, build component, and test component run on a single dedicated machine, presumably also on a dedicated account. Each initiates or kills the next stage as appropriate. The test component distributes tests across all available machines, including the dedicated machine on which it is running. Note that for any testing progress to be made, the dedicated machine must have at least a working network card so that the test component can farm tests off to *localhost*.

the build component. The build component maintains file dependency data, builds the project, initiates the test component, and is responsible for communicating with the results database, CVS, and any developers found to be at fault. The build component is also responsible for determining which tests should be run when initiating the test component and, if a build or test fails, for attempting to locate the source of that failure. Finally, the test component runs the various user-specified test suites, collects results, and returns statistics to the build component, which stores the results into the database. Due to the probable size of the test suites, the test component is responsible for distributing the tests across whichever machines are available to it at that time. See Figure 3.

3.1 Flow of Control

The system uses the following flow of control:

- Initial import. The administrator causes the system to perform a CVS "checkout" operation, builds the project the first time, sets up automatic ssh key-gen, and initializes the system. Initialization includes providing the absolute path to the working directory of the system's CVS checkout, a list of tests and their types, a list of machines and their relevant uptime thresholds, and a list of dependencies between tests (see Section 3.4). The administrator places these in the .configure, .test, .machine, and .dependency files respectively in the home directory of our system's dedicated account.
- Update notification. The system is activated on any commit to initiate builds and tests.
- Automated builds. If a cron job or CVS commit hooks into the system, an automated build is scheduled. Before building the project, the build component updates its version of the source tree from CVS.
- Automated tests. After successful builds, the build component may schedule tests to be run on otherwise idle project machines. Testing may qualify a version of the source as "working" or "not working", above and beyond merely "compiling". Additionally, tests may be qualitative.
- Automated report and blame. The build component stores all build and test statistics within a database. If a build or test failed, the build component attempts to tie the failure to a particular file, update, and developer. Finally, all developers at fault are notified of their responsibility.

3.2 The Server

The server is the interface to our system. For convenience, the server may be activated via one of three mechanisms. First, a user (a project administrator or developer) may explicitly activate the server, either to initialize the system or to make requests pertaining to execution status of the system. Second, the CVS

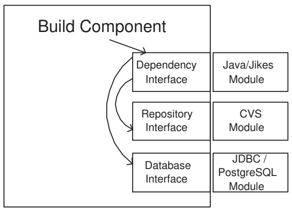

Figure 4: The build component is divided into three modules, represented by interfaces. These are: the version control interface, the results database interface, and the source code/dependency interface. The source code module relies on the general interfaces provided by the version control and results database interfaces.

log info file contains a hook into the server; on any commit, the server is activated to initiate another build/test sequence if needed. Third, a timed mechanism may activate the server (typically once per night) and request a build/test sequence. Thus the server's primary functions are to initialize the system, interact with developers after initialization, and schedule build/test sequences as necessary.

In order to initialize such a system, the server must obtain project related information from an administrator. In specific, the server must be told what to check out, how to build the project, which machines may be available for testing, which tests of which type to run and with what parameters, and how to take or interpret results of the tests. We assume that an administrator builds the project for the first time before initializing our system. Because this system is specifically for large projects written in Java, the administrator does not need to explicitly state how to build the project; a list of top-level classes might be given to aid the build process, but an automated mechanism for determining these classes may be provided by the system. Also, as long as tasks for the types of tests exist (see Section 3.4), the administrator needs only provide the types and names of tests to be run. However, in many of these cases a more general system may be constructed. Due to time constraints, we limit our system with the above assumptions.

The server is primarily responsible for scheduling build/test sequences. A build is run when the server first starts up, in case files were committed while the server was down. Builds may also be scheduled in response to CVS commits, or by explicit user request. If a build is scheduled while a build is already running, another build is run after the current build completes. Multiple build requests received before a build finishes are collapsed to a single build.

3.3 Build Component

The build component is responsible for synchronising the system's working directory with the repository. It is also responsible for compiling modified source files, scheduling the test server, and storing all results into the results database. As shown in Figure 4, the build component is built around a set of three module interfaces: the version control interface, the results database interface, and the source code/dependency interface. In our implementation, these are interfaces to CVS, PostgreSQL[5], and java/jikes, respectively.

3.3.1 Build Component Structure

The version control interface is responsible for interacting with the form of version control managing the project being developed. It must be able to perform an update to synchronise the working directory with the repository and be able to construct a set of updated files. These updated files will be used by the source code/dependency interface to rebuild the dependency tree and compile modified files. The version control interface must also be able to determine the person responsible for commiting any changes to a given file, and the timestamp of any commit. Furthermore, the version control interface must be able to provide the difference between any two revisions, selectively regress certain files to a certain date/revision, and also regress the entire working directory to a specific date and time.

The database interface is responsible for the persistent storage of results for a build or test sequence. The database module must be able to maintain a table of dependencies, records for each overall build or test attempt, and records for the status of each test. Although the database stores dependencies as strings, the particular nature of dependencies stored is transparent to the database interface, and is used only by the dependency interface to store and retrieve dependencies. Each overall build or test attempt entry contains the time at which the attempt occurred, the result of the attempt, the time taken for the attempt, and the absolute path of a filename containing any output generated by the attempt. Each single test entry contains the date of the test, the result of the test, the absolute path of a filename containing output generated by running the test, and the date at which the test most recently passed.

The source code interface is responsible for applying the general functionality presented by the database and version control interfaces to the project being monitored. The implementation of the source code module is highly dependent on the nature of the project. Specifically, it must be able to compile the source code in the system's working direcotry of the project. Furthermore, it is responsible for choosing the granularity of dependencies being maintained, and how to update dependencies as files are modified. In our implementation dependencies are tracked on a per-file basis; however, we are currently working on reducing the granularity to the level of java classes. Future work might reduce the granularity even further to individual methods. The level of granularity on which dependencies are maintained represents a trade-off between the time it takes to locate an error and the reliability in assigning blame. When a project is viewed on a per-file basis, each node will depend on a larger set of other nodes, which will itself depend on a larger set, and so on. Thus, the number of indirect dependencies for each node will be greatly increased. Alternatively, when the project is viewed as a collection of classes, there will likely be fewer dependencies per node, but a larger number of nodes to keep track of. If the project is viewed as a collection of methods, once again there will likely be fewer dependencies per node, but a much greater number of nodes. We would like to investigate in the future whether the added complexity of tracking and examining dependencies at lower levels is compensated for by an increased precision in locating errors.

3.3.2 Build Component Execution

The build component uses methods from the source code/dependency interface (which in turn uses the repository and database interfaces) to carry out its responsibilities. The build component progresses through four phases: updating the working directory of the project, attempting to build the project, attempting to test the project, and reporting results. The rest of this section walks through the phases of the build component in our implementation.

When the build component is scheduled (by the server), it first asks the source code interface to perform any updates needed for the source code or dependencies. In our implementation, the source code module does this simply by making a call to the repository interface to update the working directory. The set of files updated is then stored for use in the remaining phases. In our implementation, updating the dependency information is done in the second phase.

The second phase of the build component is building the updated project. The source code module currently performs the build and dependency information by executing the command "jikes +M {set of updated files}". The +M flag generates dependency information in the form used by *make*, which the source code module then interprets to create the dependency tree. If the executed command has an error, the build is reported as a failure, and the source code module informs the build component which files were at fault. Once the building phase has completed, the build component reports the result of the build, including the usernames of anyone who modified the files at fault. If the build completed successfully the build component proceeds to run the tests.

The third phase of the build component is to run the tests described in the project's .test file. Actually executing the tests is done by the test server, which schedules all tests to be run. As each test finishes, the test server reports the result to the build component. The build component maintains a set of all tests that have failed during this testing phase, but did not fail the previous time the test was run. After the test server returns control to the build component, the build component invokes the source code module to try and *locate* the source of any failures. Location of failures is currently done by creating a set of possible culprits by intersecting the set of updated files with the set of all files on which the failed tests depend. This set is then partitioned and ordered by the date and time at which the files were updated. The source code module then proceeds through each equivalence class of commits, regressing each group of commits and retrying the failed tests at each step. If some tests fail at one iteration and not another, then the group of files updated between the two iterations is declared to be at fault. The source code module finds the set of usernames who updated those files and appends a log entry assigning blame. The entry contains the usernames, the updated files, and the failing tests for easy identification. This report mechanism can be extended to provide any desired form of notification. Once the location has regressed the working direcotry to a point where no new tests fail, the source code module stops trying to locate errors. After the location of failures, the results are reported via the database module.

The fourth and final phase of the build component is to update all tables monitored by the database module. The source code component updates all dependency information stored in the database. The build component itself stores the result of the overall build and test attempt, as well as the result of each individual test when reported by the test server.

3.4 Test Component

Finally, the test component runs the various user-specified test suites, collects results, and returns statistics to the build component. The test component is responsible for distributing the tests across whichever machines are available to it at that time. Thus the test component has two primary functions: communicate with other machines (listed during initialization) to determine whether they may be used to run the tests, and distribute the tests and collect results. However, the test component must be robust: it should not waste much time waiting for unresponsive machines or die when an internal protocol is violated, it must be able to reschedule unfinished tests on network failures, and it should always be able to run at least one test (via a connection to *localhost*). In our implementation, the test component is structured as a server that spawns remote clients and dynamically distributes tests to these clients. The server closes all client connections either after all tests have been finished or when the test component is halted by the build component. (Within the rest of this subsection, "the server" will refer to the test component server.)

Communicating with other machines to determine whether they may be used to run the tests is straightforward. The test component server simply attempts to ssh into each listed machine to run a client. The client calls uptime and reports the results, at which point the server determines, based on specified thresholds, whether or not the client's machine is being used for other intensive processes. If so, the server closes the client connection.

Dynamically distributing the tests is slightly more complicated. The test component may need a notion of which tests cannot be run at the same time or which tests should be run before certain other tests; the administrator provides this information via a .dependency file. Furthermore, it collects information about the running time of each test; tests with much longer running times are distributed early in the testing sequence while shorter tests are held until later.

The next few paragraphs discuss implementation design decisions and tradeoffs within the test component. The first issue addresses the need for robustness within the tester, as stated above. The second addresses the ability to run dependent tests at different times. The third addresses our goal of easily extensible and replaceable modules.

The test component must be able to withstand network failures, exceptions thrown from tests, and as many remote software failures as possible. To allow the server S to withstand exceptions thrown from tests, the following protocol is followed. For every test T handed to it, a client C preprocesses T (instantiates the test), runs T, postprocesses T, and catches any exceptions generated within these three steps. If an exception is generated, C sends a failure message to S; otherwise, C sends S the collected results of T. In either case, S then removes T from S's unfinished tests list and reports T's results to the build component. Alternatively, the server could run tests and uptime on client machines directly through ssh rather than through the client code, but our implementation allows long-standing connections, Object Streams (for easy Object transmission), and cleaner transmission of test data.

To be able to withstand network and remote software failures, the test component server disconnects from any client that violates the server-client protocol or with whom there is a socket or connect error. When disconnecting, the server returns any test that the client processor was running to an available list and discards all state associated with the client. The server does not disconnect from clients that fail to send results after being asked to run a test; however, if the build component kills the test component, the server disconnects from all clients. The server attempts to reconnect to unconnected processors every ten minutes.

The unpredictabilitly of failures and client responses naturally suggests an event-driven model for the server. We decided to use a single threaded server to avoid concurrency issues. However, to use resources optimally, single threaded event-driven servers must only use non-blocking calls when processing an event. Unfortunately, ObjectStreams (for easy data transmission) do not have a working *available*() method, and thus reads on ObjectStreams cannot be non-blocking. To get around this, we created a hack *select* by having the server sit in a round-robin read loop across the connected clients with a small (1ms) timeout; whenever it successfully reads an object from a client, it returns a wrapper object containing the client and the object read to the main control loop. The main control loop can then determine the value of the object and attempt to read from the specified client whatever subsequent objects are specified by the server-client protocol. Finally, the main control loop will process the response (e.g. by logging the success or failure of a test) before returning to the server read loop. Obviously this is a waste of server machine resources, but since this does not affect testing speed greatly and since speed was not a primary goal, we accepted this waste as the cost for avoiding concurrency issues.

Because our system assumes a distributed file system such as AFS, two tests may not be able to run on different machines at the same time. Communicated via the .dependency file, a dependency of $T2$ on $T1$ means that $T1$ must be run *to completion* before $T2$ can be begun. Each time the test component is run, the server constructs a graph based on provided dependency information. Each node represents a test, each edge $n1$-}$n2$ represents a dependency of $n2$ on $n1$. Each node with no dependencies (i.e. no parents) is added to an available list, sorted by longest running time, from which all distributed tests are chosen. When a client reports a test completion, the test node is removed from the graph and all of the node's children with no other dependencies are added to the available list. When the server hands a test to a client, the test node is removed from the available list (but not the graph). When the server disconnects from a client holding a test, i.e. on an error, the server re-adds the test node to the available list. The server does not terminate unless the graph contains only a dummy start node. Note that although test dependency cycles cause the server to continue running even when no tests will ever become available, a cycle detector and breaker could easily be added. (We did not do so due to time constraints.)

In distributing and reporting tests, the server and clients exchange instances of tests extending the $TestTask$ interface. $TestTask$ is based on the general $Task$ interface presented in [9], and is the mechanism by which we allow the test component to be easily extended to run other types of tests. Each $TestTask$ contains *preprocess*, *execute*, and *postprocess* methods. Generally, *preprocess* instantiates any classes needed to run the test and throws an exception on any unrecoverable error (such as $classNotFound$). This causes the client to return an error log for the test. The *execute* method actually runs the test and collects results. The *postprocess* method generally converts collected results into a Serializable form (so that results can be sent back to the server using the *writeObject* method for ObjectStreams). Notice that, as with $Task$, the $TestTask$ interface allows any instance implementing $TestTask$ to be passed between server and client. Thus, adding types of tests to the system requires only two steps: first, implementing a class that performs the desired test and extends $TestTask$, and second, modifying the test instantiator ($FAT.Tester.TestSetup.setupTests()$) to be able to recognize the new type and instantiate it properly. $TestTasks$ can perform any function that can be wrapped in Java; they allow us to meet our goal within the test component of easy replaceability and extensibility. Alternatively, tests could simply be a list of command lines that the test component *executes*, allowing just as much flexibility with no additional code. However, our implementation allows administrators to easily write custom results interpreters and exception handlers, and facilitates processing results to a great extent. Furthermore, since we assume that most tests are JUnit tests or JUnit Suites (as this allows our system to easily determine the project code on which each test depends and so appropriately track bugs), no additional code is required for these cases.

4 Analysis

The following evaluation describes how our system met each of our goals.

- *Locate Errors*: The build component attempts to locate the causes of errors, and assign blame based on who changed the project code on which a failing test depends. Our current implementation works at the granularity of a single file, which we believe should be a fine enough level in the usual case. Most files should not depend on many other files, and, in general, programmers will be working on different files. We would like to apply our system to a representative, active project to investigate the value of trade-offs on the level of granularity, and evaluate our ability to locate errors.
- *Store Data*: For each test run, our system stores the running time, success or failure, and the last date

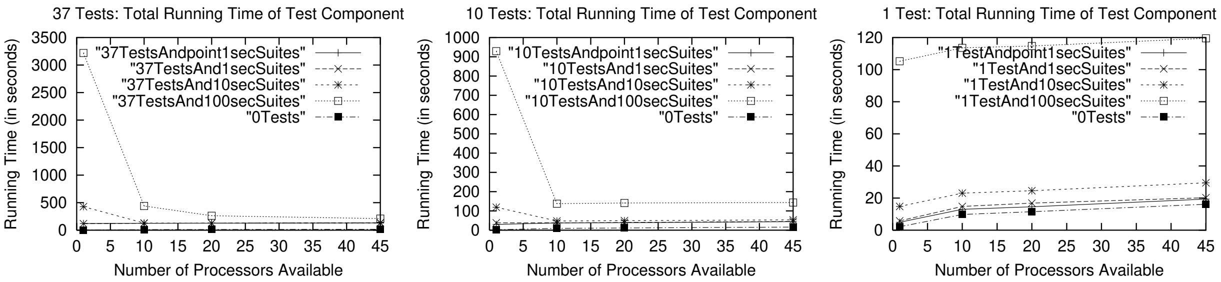

Figure 5: This plots the running time of the entire test component when varied across the number of tests and suites per run({0, 1, 10, 37}), the number of processors ({1, 10, 20, 45}), and the average individual test time in seconds ({.1, 1, 10, 100}).

on which the test passed. The build component concatenates success and error messages for each test into a success and an error log for an entire invocation of the test server; it also stores standard out and standard error from build attempts. Although we haven't implemented a front end for accessing the database, any administrator or developer with database knowledge should be able to easily access the data stored therein.

- *Be Easily Extensible*: The system makes heavy use of interfaces to allow easy extensibility and replaceability. The server relies only on a command line hook into the builder. The builder relies on database and version control system interfaces rather than given implementations, so the particular database and version control system used is independent of our system. Also, the builder uses an interface to interact with the dependency structures; thus the system can be easily changed to use a different dependency granularity than file granularity (e.g. method granularity). The tester relies on the *TestTask* interface, a Graph manipulation specification, and a server-client protocol including "wait", "execute", "uptime", and "failure"; any of these portions could be replaced if done with care. The builder and tester have a small five-method interface, and are thus primarily independent.
- *Distribute Tests*: Again, once the .machine, .test, and .dependency files have been specified, testing and results processing is automatic. Thus the interface for distributing tests is very simple: list the machines and tests and any dependencies to use, and let the system run. As can be seen in Figure 5, the distribution of tests does in fact speed up testing as the size of the tests and the number of tests increase. For very small tests, the overhead of the system is quite large (about 2000% for tests that run within 1 second). However, distributing the tests provides substantial speedup for tests greater than 100 seconds in length, and is still profitable for tests around 10 seconds in length. As tests for real development projects are probably quite long (for example, StreamIt's [1] JUnit tests require at least 10 seconds, and frequently much more), distributing the tests actually does speed the testing process quite a bit.
- *Require No Administration After Setup*: Although the administrator is required to take several steps to initialize the system, once finished, all of the information needed to run the system is present. The only administration necessary requires editing the .test and (possibly) .dependency files when tests are added or removed or editing the .machine file when the server machine changes or available machines need to be added or removed. Hence, the entire system requires no administration to continue functioning as setup once it has been setup.

5 Related Work

There have been many ad hoc implementations of an automated build and test system. Most implementations simply do a full CVS checkout, run make, and possibly run a set of tests, e-mailing the results to developers. These systems generally do not keep track of the most recent working version, use incremental builds, or have a well-defined user interface for customization. The GCC Automated Testing System [6] maintains the status of the most recent build and test, and e-mails the results to an archived list.

The Tinderbox [4] project is an OpenSource implementation of an automatic build master. Version 1.0 of Tinderbox was designed specifically to monitor the CVS tree used in the development of Mozilla.

This version relied on the existence of a CVS tree interface created for development of Mozilla (Bonsai). Tinderbox has been rewritten to be a more general development monitoring system (Tinderbox 2.0) that can have modules added to work with a wider variety of systems.

The OpenRISC 1000 automated testing system [7] attempts to check out the HEAD revision for the OpenRISC 1000 CVS tree and build it. If the build fails, an e-mail is sent to a list of developers. This system does not maintain any state for previous executions of the builder/tester.

6 Conclusion

We have implemented and documented an automated build, test, and blame system for use with large Java development projects with JUnit tests. The system consists of three parts: the server, the build component, and the test component. The server provides the interface to administrators and developers. The build component inerfaces with a database, the test component, and a revision control system; the build component is responsible also for locating the cause of failures (currently at the file granularity) and blaming the appropriate developers. Finally, the test component robustly distributes tests across available machines and collects test results, interfacing with the build component to report results.

The entire system is built to allow easy extensibility and replaceability: it is not tied to any single version control system, database, test type, dependency granularity, or main module. Extensive use of interfaces has allowed us to keep our system modular. Furthermore, the system does not require administration after setup; once everything has been set up and initialized, the system is able to run on its own.

Thanks to Robert Morris for editing our rough draft and helping guide our project.

References

[1] S. Amarasinghe, M. Gordon, H. Hoffmann, M. Karczmarek, A. Lamb, C. Leger, J. Lin, D. Maze, A. Meli, W. Thies, and J. Wong. A stream compiler for communication-exposed architectures. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[2] P. Cederqvist and et. al. Version management with CVS. 1992, 1993.

[3] P. Charles and et. al. Jikes. *http://www.opencores.org/cores/or1k-new/ats/*, 1997.

[4] K. Estes. Tinderbox 2.0. *http://www.mozilla.org/projects/tinderbox/index.html*.

[5] PostgreSQL Global Development Group. PostgreSQL 7.2 user's guide. 1996-2001.

[6] G. Keating. GCC Automated Testing System. *http://people.redhat.com/geoffk/gcc-regression/*.

[7] D. Lampret. OpenRISC 1000 Project: Automated Testing System. *http://www.opencores.org/cores/or1k-new/ats/*.

[8] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[9] J. Waldo and A. Wollrath. Trail: RMI: Designing a remote interface. 1995-2002.