An Analysis of Distributing Test Suites:
Comparing Load Balancing and Sharing

Michael Camara, Colton McCurdy, Gary Miller, & Herbie Torrance

Abstract

This research focuses on the utilizing distributed testing to make the execution of large test suites more efficient. The key trade-offs for taking a distributed approach to testing are be identified and the impacts on using a distributed approach are analyzed. This paper will mainly focus on comparing two distributed test systems, Test Load Balancer (TLB), and our own system, Distributed Test Sharer (DTS).

Contents

1   Introduction

Testing is often considered to be one of the most important components of the software development cycle and therefore it is important to conduct a thorough and correct test suite on a software system. Many large software systems are subject to testing that can take several hours and potentially longer to execute, which can be very costly. For example, in situations where a system's test suite can take several hours or even days to execute, and one of the tests that is not executed early in the testing process fails, the cost of identifying this failure can be exceptionally high. For this reason, approaches to decreasing the cost of executing test suites is a very popular topic in software development. One of approaches to successfully reducing the cost associated with testing a system is to use distributed testing. This is an approach in which test cases are distributed or scattered across many different nodes in a network, executed locally on that given node, and the result of the test case is then returned across the network. The focus of this research is to identify, analyze, and report the results of experiments conducted on distributed testing using two different systems, Test Load Balancer (TLB) and Distributed Test Sharer (DTS). It is hypothesized that by distributing test cases and executing them in parallel on several independent nodes, the overall cost of executing a full test suite can be reduced due to the increase of computational resources and evaded the limitations experienced with computing on a single node.

2   Background On Distributed Testing

In order to conduct an analysis on using a distributed approach to testing, more research on the trade-offs associated with the topic was conducted. First, it is important to understand situations in which distributed testing may be beneficial; the most obvious being when a test suite takes several hours or days to execute. It also may be necessary to conduct distributed testing if the the system of focus must be tested on different web browsers or sites, in the case of a web-based application, or even on different operating systems. In either case, it may be difficult or costly to executing these tests on a single node and therefore distributed the tests to other nodes may be beneficial.

Next, it is essential to identify both the benefits and drawbacks of testing in a distributed fashion. Some of the key benefits associated with distributed testing is that less computation is required on the CPU in which the system is executed on, which allows more work to be done locally and is potentially a large benefit in a situation where it is necessary to execute other processes alongside the test suite. More importantly, the goal is to reduce the runtime of the test suite by executing several test cases in parallel and is an exceptionally more feasible task when using multiple CPUs. One of the most common ways in which the increase in performance is a achieved is by diminishing the cost of dependencies. Tests cases that are dependent on another case often have larger costs because their result or even execution may not occur until another case is completed. By using distributed testing, these dependent test cases can be isolated to a particular node that is responsible for executing the group of dependent cases while other nodes can execute other independent cases that otherwise would have been waiting in a queue if centralized testing was used. This situation is a clear example in which a distributed approach would be more beneficial that a centralized approach.

Though it is clear the distributed testing may be beneficial to use there certainly are drawbacks and challenges that exist with this approach. The greatest of these drawbacks is the cost of communication required to transfer data over a network from one node to another. In order to execute a test case at a remote location, either the code to execute the test must exist at the remote location or the code must be sent across the network. This is a potentially costly situation if the required code is large and takes up a lot of memory. The situation where the

code already exists at the remote location is often unlikely and therefore the test code usually must be transferred to the node that will execute it, therefore it must be determined whether communication costs of distributed testing out weighs the computational and time costs associated with executing the tests locally. Another drawback associated with distributed testing is that it is dependent on a network, which in some cases may be unavailable. If a system uses a distributed test suite and it is unable to gain access to a network, then there is no way to run the tests until a network becomes available. Another unfavorable situation may occur if there are fewer remote nodes available than required. In a situation such as this, both the communications costs and computational limitations may be experienced because all of the data is transferred to a number of locations that cannot execute the tests in a time that justifies distributing the tests.

Alongside the drawbacks of distributing testing, several challenges exist with this approach as well. One of the key challenges associated with distributed testing is the concept of load sharing versus load balancing. Load sharing occurs when the tests or load are sent to several nodes to be executed without any restrictions on how the load is allocated. Load balancing is a more complex approach in which the allocation of the load is sent to nodes such that each node is doing an equivalent amount of work. This allows the load to be executed more efficiently because it prevents situations in which a particular node is doing a significantly greater amount of work than others or a particular node is doing little to no work at all. It is quite evident that load balancer is a more favorable approach to take, but the complexity associated with it is much larger. For example, consider a situation in which a test case is dependent on another, in this case a load balancing system must not only consider the amount of load to allocate to each node, but also when certain components of the load need to be allocated.

3 Overview of Related Work

3.1 Existing Systems

Though distributed testing is a popular and rapidly developing topic in the software industry, not many systems that use his approach exist and work correctly. This is due to the fact that it is not only a relatively new concept, but the complexity of building a system that can run tests on a system and efficiently distribute them to remote nodes is exceptionally high. One existing system that successfully does this is named Test Load Balancer (TLB). TLB claims to support testing of every language on every platform and partitions the tests into subsets that can be executed in parallel on different physical or virtual machines. These subsets are executed in such a way that they all start at the same time and finish at almost the same time as well, therefore the overall time is takes to execute all of the tests can be divided by the number of test subsets that are generated. TLB assures that a particular node will execute only a number of tests proportional to the total number of test divided by the number of available nodes and guarantees mutual exclusion and collective exhaustion of test execution, meaning that no test will be executed more than once and every test will be executed. The TLB system is comprised on two main components, the first of which is a server that is responsible for storing test data and the second is the balancer that partitions and orders the execution of tests. By storing test data, the TLB system will reorder the execution of tests with each run based on previous results. This is a extremely beneficial feature because tests that are known to fail will be executed earlier to avoid unnecessary downtime time resulting from failing tests being executed later. Though it is very important to note that this is only achievable by using tests that are independent of one another which is not always the cases. Unfortunately, large systems very often have test cases that are dependent on other tests and systems such as these are usually examples that would benefit from using distributed testing opposed to centralized

testing. Also, this method can not guarantee the order in which tests are executed due to its fail early approach to executing test, therefore tests must not only be independent of other tests but also independent of the order in which they are executed.

3.2 Distributed Test Sharer

In order to understand and further evaluate the trade-offs associated with distributed testing, we developed a novel system named the Distributed Test Sharer (DTS). This system was created using the Java programming language and consists of two main classes: `Delegator` and `CustomServer`. The relationship between these classes is illustrated in Figure 1, with communication facilitated by Java Remote Method Invocation (RMI) and use of an FTP server. The following section will outline the steps performed by DTS to distribute JUnit tests located on the `Delegator` node to any number of `CustomServer` nodes, which run the tests and return the results obtained.
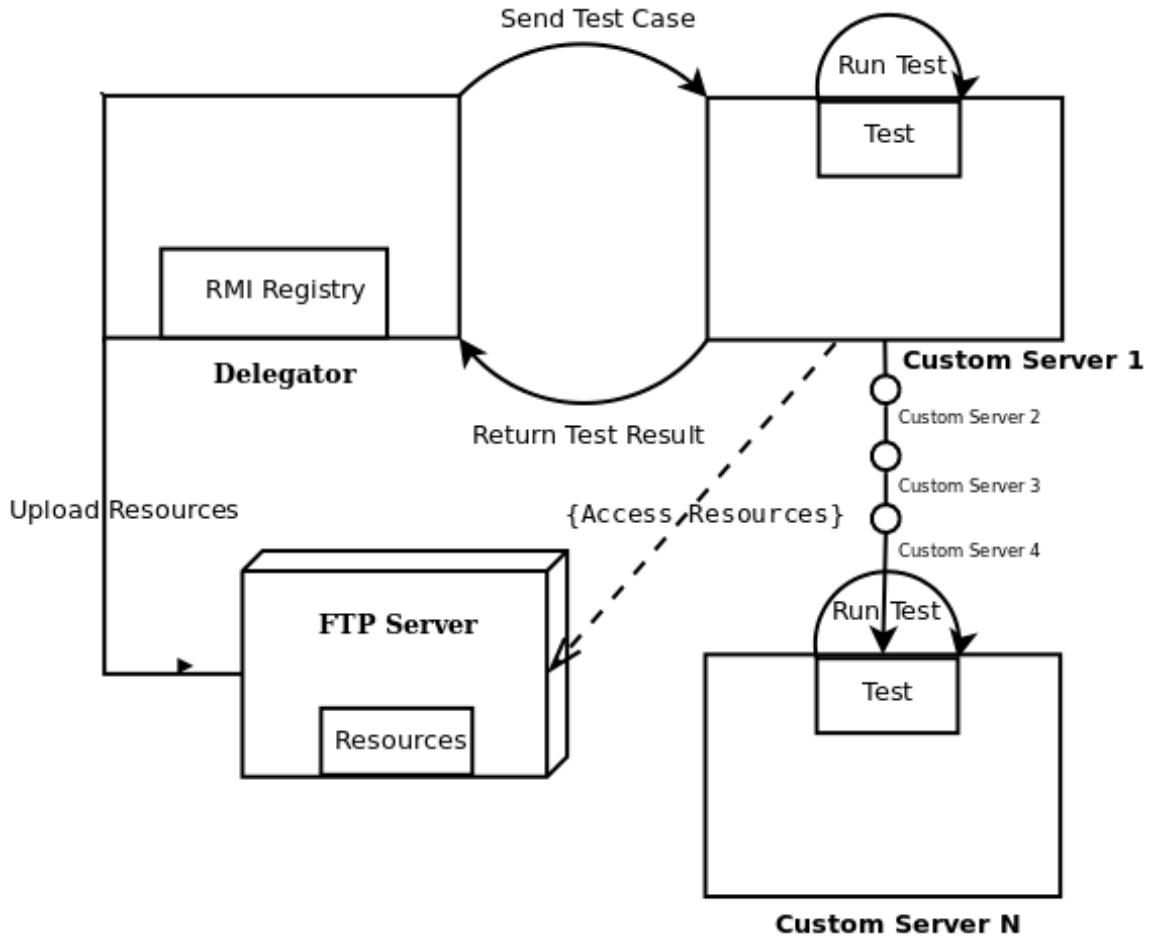

Figure 1: Architecture of the Distributed Test Sharer (DTS).

Before starting the system, the user is expected to furnish all of the JUnit test cases and any associated resources needed to run those test cases. All files will be placed in the `ftpserver` directory, or one of its subdirectories. The top level of this directory can be filled with any non-class files or directories that might be required for test suite execution. For instance, the SchemaAnalyst test suite uses a `config` directory that contains various text files filled with different properties. In this example, the user would place the entire `config` directory immediately inside the `ftpserver` directory. Further nested in the `ftpserver` directory is a `resources` directory, which contains `bin`, `lib`, `test_singles` and `test_suites` subdirectories. For simplicity, the user should put the entirety of the directory containing their compiled classes in the `bin` directory. Continuing with the SchemaAnalyst example, this would involve placing the `org`, `paper`, and `parsedcasestudy` directories into the `ftpserver/resources/bin` directory, which are usually located in that system's `build` directory. All `jar` files used by the desired tests should be placed in the `lib` directory. Finally, any individual JUnit test cases can be put in the `ftpserver/resources/test_singles` directory, while any test suites (those that reference multiple JUnit test cases, like an `AllTests` class) can be put in the `ftpserver/resources/test_suites` directory.

DTS execution begins on the side of the `Delegator` with a few preliminary steps. First, several system properties are programmatically altered. These include `java.security.policy`, which is set to a custom policy that grants all permissions to any nodes that attempt to connect to the `Delegator` through Java RMI. Although a more stringent policy would generally be recommended to improve the security of the system, this was deemed appropriate for the purpose of prototyping the system with fewer obstacles. Further, the `java.rmi.server.hostname` property is set to the internet protocol (IP) address of the `Delegator`, which is retrieved through the command line interface at program execution. This specifies the given IP address for use with any registry that is subsequently created, allowing remote nodes to find it.

Next, a Java RMI registry is created using a port number that is currently hard coded as 12345. This port number was chosen because it resides in the range of acceptable ports for the machines used for experimentation, and generally is not assigned to any other system processes. A new instance of `Delegator` is then created and linked to the registry through use of the `rebind()` method, binding the object with a key of "Delegator." This will allow `CustomServer` nodes to access some of the methods offered by `Delegator` in future steps.

Both the `Delegator` and `CustomServer` classes extend `UnicastRemoteObject`, which allows them to be bound to and retrieved from a Java RMI registry. It facilitates the dynamic creation of stubs for such objects, obviating the static method used in previous versions of Java. Each class also implements an interface that extends `Remote`: `DelegatorInterface` and `CustomServerInterface`. The `Remote` marker interface is further required to successfully bind to the registry, and it allows for type checking of remote object invocations at compile time.

After the registry is created, an FTP server is created using the Apache FtpServer framework. This framework was chosen because it allows programmatic creation of an FTP server using Java, instead of relying on a third-party application that would need to be run and configured separately. Various settings are adjusted during this step, such as establishing a user name and password, setting the port and host address, and enabling concurrent logins. Most importantly, the home directory for the FTP server is set to the `ftpRootDir` variable, which is currently hard coded to the local `ftpserver` directory where the test cases and resources were previously placed. Any users that connect successfully with the server will then have immediate access to this directory and any nested directories contained within it.

The final preliminary step taken by `Delegator` is the creation of a list of JUnit test cases. Both the `ftpserver/resources/test_singles` and `ftpserver/resources/test_suites` are traversed recursively to locate all nested `class` files, which are saved as `File` objects and stored into separate global `LinkedLists`: `testSingleList` and `testSuiteList`.

After these steps are completed by the `Delegator`, it waits until the user enters the carriage return character to continue. During this wait, remote nodes can connect to the `Delegator` by running the `CustomServer` class. Importantly, two command line arguments must be specified at execution: the IP address of the `Delegator`, followed by the server's own IP address. Ideally, multicasting would be used by the `Delegator` to broadcast its address, allowing the `CustomServers` to locate it automatically. While this might be pursued in the future, the current implementation requires the `Delegator's` IP address to be known.

Once started, the `CustomServer` sets the system RMI policy in the same manner as done by the `Delegator`. Additionally, it will set the `java.system.class.loader` system property to a `CustomClassLoader`. This class extends a `URLClassLoader` and allows access to some of the normally protected methods through use of reflection. For now, it simply sets the classpath to the local `bin` and `lib` directories. Next, the `CustomServer` will locate the registry on the `Delegator`, using the IP address previously specified, and obtain a remote reference to the bound `Delegator` object using the `Naming.lookup()` method. Using this reference, it will call the `rebindServer()` method to bind the `CustomServer` to the registry. This is repeated for each `CustomServer` node until they are all bound to the registry.

After all `CustomServers` are created and connected, then the user will enter the carriage return character on the `Delegator` node to continue execution. The `Delegator` will first call the remote `updateClassLoader()` method for each of the `CustomServers`. This will add all of the relevant `class` directories and `jar` files to the classpath of each `CustomServer`, accessible via FTP protocol and the FTP server previously created. Additionally, all of the non-class files previously placed in the `ftpserver` directory on the `Delegator` node are actively retrieved through communication with the FTP server and stored locally on each server node. Although efforts were made to obviate the need to store any files on the server, this was deemed a critical step to ensure that subsequent tests could be run successfully.

Next, the `Delegator` creates `TestAgent` objects for each `CustomServer`. These agents extend the `Thread` class and implement `Runnable`, allowing multiple agents to run their respective methods concurrently on separate threads. The `run()` method will simply do a remote call to the `runTest()` method for its assigned `CustomServer`, sending a test case and storing the `Result` generated by the server into the same `ConcurrentLinkedQueue`. These agents are further placed into another `ConcurrentLinkedQueue` for subsequent access by the `Delegator`.

Once these agents are created, the `Delegator` will sequentially iterate first through the `testSingleList`, and then through the `testSuiteList`. With the single test cases, each `File` in the list is assigned as-is to an agent in the queue, provided that agent is not currently active. The `run()` method of the agent is then started, sending the test to the assigned `CustomServer`. If the agent is active, then it is placed back into the queue and the next agent is checked, continuing in that manner. This load sharing approach ensures that each `TestAgent`, and therefore each `CustomServer`, is always running a single test case at all times. This process is similarly repeated while iterating through the test suites, with some additional steps. The test suite `File` is first converted into a byte array, which is then converted into a `Class` object using a `SimpleClassLoader`. This class loader simply overrides the `defineClass()` method of a generic `ClassLoader`, allowing this conversion to take place. Next, the test suite is further converted into an array of `Class` objects by retrieving all test cases listed under the `SuiteClasses` annotation in the suite. This array, containing all of the individual test cases in the suite, is iterated through in the same manner described previously, performing load sharing using the previously created agents.

On the `CustomServer`, there are two `runTest()` methods: one has a `File` parameter, while one has a `Class` parameter. The method for `Class` objects simply executes the static `JUnitCore.runClasses()` method on that object, which returns a `Result` object. This `Result` object contains all of the details about the successes and failures from running the test, and

is returned to the caller. The `runTest()` method for `File` objects first converts the `File` to bytes, and then to a `Class` object, finally using the same `JUnitCore.runClasses()` method to run the test. This is performed by the `CustomServer` to lessen the computation required on the `Delegator` as much as possible. In both cases, running the tests is only possible due to the previously implemented `CustomClassLoader`. This allows the JVM to automatically retrieve any needed `class` or `jar` files via communication with the FTP server on the `Delegator` without any prompting from the user, and without needing to copy those files to the `CustomServer` node.

The `Delegator` continues to perform load sharing, assigning test cases to `TestAgents`. The agents continue to store the `Result` objects returned from the `runTest()` method of each `CustomServer` into the same `ConcurrentLinkedQueue`. Once all tests have been assigned, and all `Results` have been retrieved, then the each `Result` is parsed and the total number of successes and failures are displayed for the user. Each failure is further elaborated to show why a test case might have failed. This completes the process used by the Distributed Test Sharer, with all tests located on the `Delegator` having been executed by separate, remote nodes.

4 Experimentation Protocol

4.1 Systems Specifications

- **Colton's 13-inch, Mid 2012 MacBook Pro**
  - OS / Version: OS X El Capitan version 10.11.4
  - Processor: 2.9 GHz Intel Core i7
  - Memory: 8 GB 1600 MHz DDR3
  - Startup Disk: Samsung 850 Pro SSD
- **Herbie's Hp Envy TS 17 Notebook PC**
  - OS / Version: 64-bit Windows 10
  - Processor: 2.4 GHz Intel Core i7-4700MQ
  - Memory: 16 GB
- **Michael's HP Dv7-6c95dx Notebook PC**
  - OS / Version: 64-bit Windows 10
  - Processor: 2.20 GHz 2nd generation Intel Core i7-2670QM
  - Memory: 8 GB
- **Herbie's Fujitsu lifebook t2020 (x2)**
  - OS / Version: 32-bit Ubuntu 14.04
  - Processor: 1.2 GHz Intel Core 2 Duo U9400
  - Memory: 3 GB

4.2 Experimentation Protocol for TLB

For TLB we ran a Docker container with Java SE Development Kit 8u91 and Apache Ant version 1.9.4. These were both necessary for running the existing tool in the Test Load Balancer (TLB). TLB is compatible with JDK versions as old as version 6, but to keep our experiments as consistent as possible, we installed the latest version (8u91) on all of the machines being used for testing. Having the newest version of Apache Ant was not entirely necessary, but again, we wanted to have the most recent versions of software for this experiment due to possible optimizations in the newest versions.

4.2.1 Running TLB in a Distributed Fashion

We ran the TLB tool five times for all configurations (e.g. three nodes on the schema-analyst test suite). We decided to only run each configuration five times due to time constraints. We had it calculated that if we ran each configuration for the thiry trials suggested by Traeger et al. that our experiments combined would take approximately 20-hours to run, requiring us to sit at our computers and monitor the tests constantly for this amount of time [1].

Also, for the TLB experimentation setup, we ran Docker containers on each node. There were three total nodes, all of which were the available personal laptops of a team member. The system specifications for the machines used in this experiment can be found in section 4.1. We would have liked to run the experiments in a more controlled and consistent environment by running the Docker containers with TLB on the Alden lab machines, but were denied permission to install Docker for various potential security reasons. To read more about the challenges faced during this assignment, read section 6.

The way TLB is designed requires a user to start the `TLBServer` on one node and start the clients with the `TLBBalancer` on other nodes. Therefore, to collect the data necessary, we started the server on the 13-inch, Mid 2012 MacBook Pro and ran the Docker containers on two other machines as well as on the MacBook Pro. This was the setup for running three clients with the TLB system. For two and a single client, we removed Michael's HP Dv7-6c95dx, which performed the worst when running the test suites in the Docker container. Specifically, for two nodes, we used Herbie's HP Envy as one client node and Colton's MacBook Pro as the other client as well as the server. Ideally, we would have liked to only run the server on Colton's MacBook Pro and the clients on their own individual machines, but again, Michael's laptop took substantially longer time to execute its portion of the test suite. However, for a single node, we did not have to worry about running a server on a different machine. We ran the Apache Ant task `test` which for SchemaAnalyst's test suite called five iterations of the test suite and for Apache Ivy ran the test suite provided with the system.

It was the case that in order to make test suite execution time long enough to see substantial differences between number of nodes, we needed to run multiple iterations of the same test to artificially increase test suite execution time. Additionally, for comparison to a test-sharing system we needed to run this modified version of SchemaAnalyst's test suite.

This is the protocol we followed throughout the experimentation phase of this final laboratory assignment. We acknowledge that there were a lot of places where we did not apply the industry-standard protocols for testing, however due to time constraints and availability of machines, this is the best protocol that we could follow to ensure that the data that we collect was as accurate and consistent as possible.

4.3 Experimentation Protocol for TLB

4.3.1 Running DTS in a Distributed Fashion

5 Analysis of Results

The first analysis will be of running Apache Ivy's test suite on the Test Load Balancer tool. On average, on one node, running the Apache Ivy test suite took approximately six minutes to execute. To clarify, when I refer to running on a single node, I am referring to running the test suite using the `ant test` task in the `build.xml` instead of the `ant test.load.balanced` task.

Where we expected to see a significant performance increase—especially with the Apache Ivy test suite— was when adding additional nodes. However, we did not see this dramatic increase in performance and actually experienced a decrease in performance when adding the

third node. I will note that the third node that was added was Michael's Dv7-6c95dx which we noticed took substantially longer to execute test suites than other nodes on average.

Therefore, we see the greatest performance when balancing the work using the Test Load Balancer tool to two nodes. When executing the Apache Ivy test suite on two remote nodes, we notice that each node is able to complete their portion of the test in approximately 1.25 minutes. Refer to Figures 2 and 3.
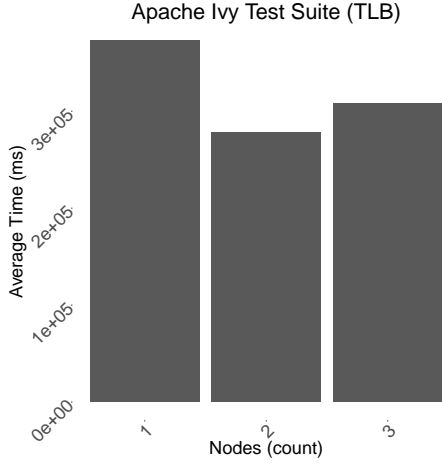

Figure 2: A bar chart displaying average time in milliseconds to run Apache Ivy's test suite on TLB
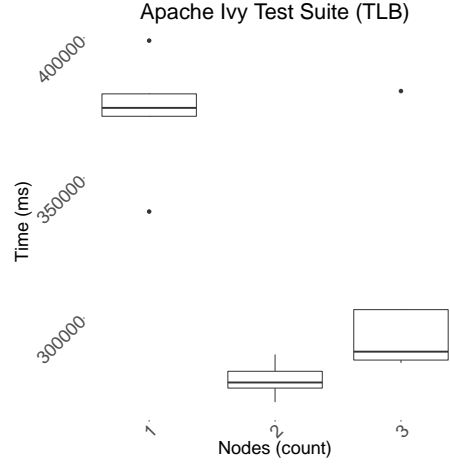

Figure 3: A boxplot displaying average time in milliseconds to run Apache Ivy's test suite on TLB

Secondly, and more importantly, we wanted to analyze the effectiveness of the Test Load Balancer tool at balancing SchemaAnalyst's test suite. This is more important to use in particular because the Distributed Test Sharing tool is—in its current state—only able to execute this test suite. With that said, we are able to compare direct performances of the tools. However, again, there are some inconsistencies such as which laptops were available for the systems to be run on.

Similarly to the Apache Ivy's test suite for the Test Load Balancing tool, the TLB tool experiences its best performance with two nodes. We hypothesize that this is for the same reason as noted previously regarding to the performance of the one node. However, here when the third node is introduced, running the test suite locally, one a single node is more performant. This makes it not worth distributing tests to more than two nodes.

SchemaAnalyst's test suite on a single node took on average 15 seconds to execute. We do note that there are a lot of failing tests and missing resources. This is why we hypothesize that this execution time is so minimal. A lot of the dependencies necessary for this test suite are database management systems that were not installed on many of the machines used in this experiment.

In conlusion, for the TLB tool, it is not worth running either Apache Ivy's or SchemaAnalyst's test suite on more than two nodes. For both of these test suites, it is worth balancing the work to two remote nodes using the TLB tool communicating with the TLBServer to `POST,` `PUT` and `GET` historical test data for a given test suite.

Finally, we are interested in the performance of the system created by Michael, Distributed Test Sharer (DTS). As previously noted, we were only able to run SchemaAnalyst's test suite on DTS.

SchemaAnalyst Test Suite (TLB)
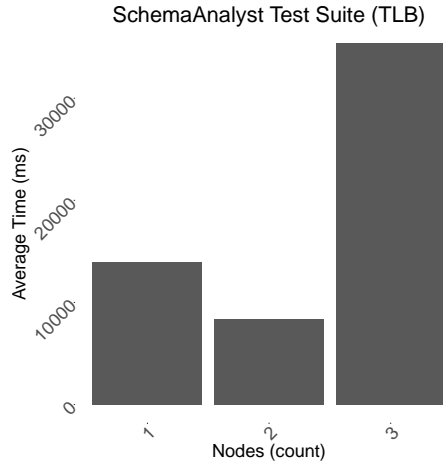
Figure 4: A bar chart displaying average time in milliseconds to run SchemaAnalyst's test suite on TLB

SchemaAnalyst Test Suite (TLB)
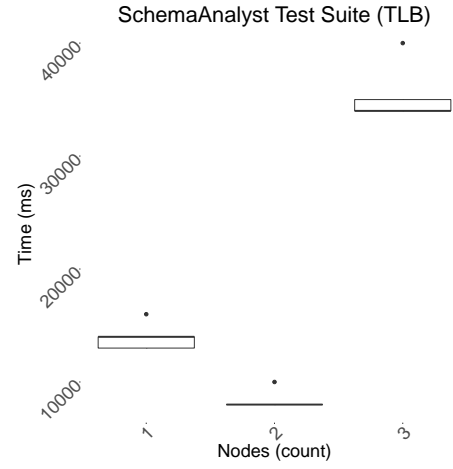
Figure 5: A boxplot displaying average time in milliseconds to run SchemaAnalyst's test suite on TLB

SchemaAnalyst Test Suite (DTS)
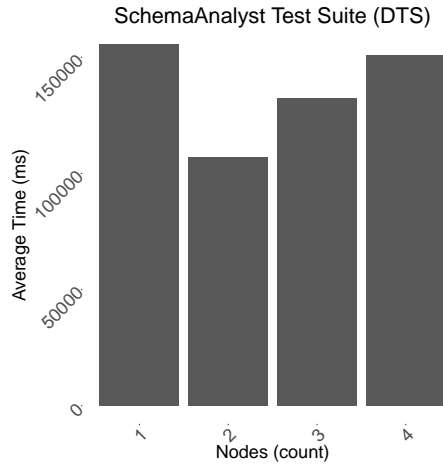
Figure 6: A bar chart displaying average time in milliseconds to run SchemaAnalyst's test suite on DTS

SchemaAnalyst Test Suite (DTS)
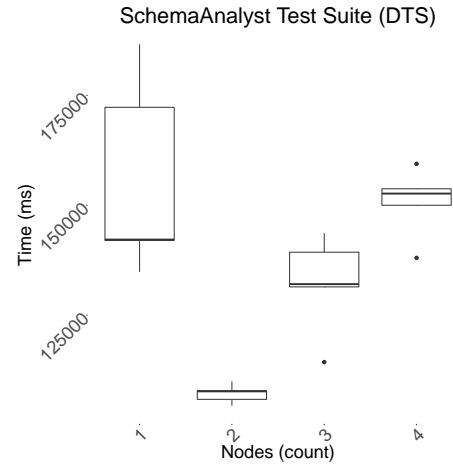
Figure 7: A boxplot displaying average time in milliseconds to run SchemaAnalyst's test suite on DTS

First, in Figures 6 and 7, we see that running the test suite on a single node takes about 2.5 minutes to execute. Then, we notice that when sharing the work with another node, we see a pretty substantial performance increase, reducing the overall time to execute the test suite to a little over a minute and a half. This is a total reduction of about a whole minute.

However, we see a trend when observing these visualizations. We notice that as we add more nodes to share work with there is a direct correlation to the increase in time to execute the test suite. This is due to the overwhelming communication costs between these larger

number of nodes. We are able to eliminate the costs of actually transferring the files necessary for running the test suite by accessing them through a file transfer protocol. Had we added this additional cost on top of already expensive communication, running the DTS tools as a sharing mechanism for SchemaAnalyst's test suite would make it less worth much more quickly.
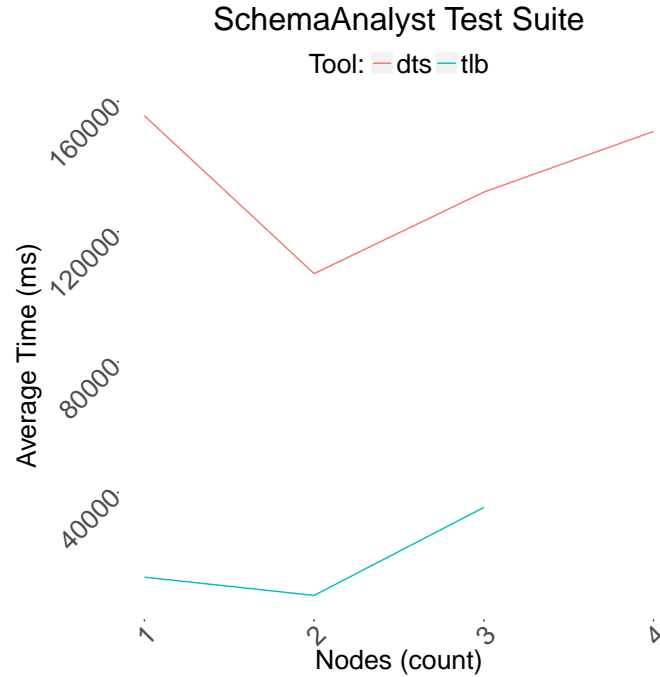
Figure 8: A direct performance comparison of the TLB and DTS tools for running SchemaAnalyst's test suite

In Figure 8, we notice that both TLB and DTS behave similarly when distributing the SchemaAnalyst test suite. Running the test suite on a single node takes longer than the most performant number of nodes—that being two total nodes. At two nodes, both systems perform their absolute best and only decrease in overall performance from there.

Remember, we were limited to only comparing the test suite of the SchemaAnalyst tool and on very few nodes due to the lack of permissions and available resources. We would have liked to run these two systems on many more similar nodes to alleviate any inconsistencies, however this was just not possible for the scope of this final laboratory assignment.

6 Challenges

Various challenges were encountered while developing the Distributed Test Sharer, with many of these challenges revolving around the use of Java RMI for communication between nodes. There were a variety of different settings and properties that I didn't originally know about, despite using this technique in a previous assignment. These included either setting the RMI host name and security policy through the `-Djava` command line parameter, or using the `System.setProperty()` method programmatically. Omitting these steps produced a variety of different, confusing errors, and required much troubleshooting to figure out how to fix them.

There were also problems getting the RMI to work on the computers in Alden Hall, which ultimately prevented those computers from being used in our experiments. Originally, these problems arose from the fact that the Java RMI registry would occasionally use a random port number instead of the one specified in the source code. This created much confusion originally, since the Java API includes various method signatures that require a port number parameter; yet it seemed as though such specification did not have the expected effect. Ultimately, using a custom `SocketServerFactory` and adjusting the parameters for the `UnicastRemoteObject` constructor fixed those initial problems involving randomly assigned ports. However, new problems arose with remote objects being removed from the registry after binding, perhaps due to some garbage collection issue. These problems were not present during testing of DTS on personally owned computers, and unfortunately they could not be resolved despite significant time investment.

Another challenge with DTS was getting the tests to run on a separate node without error, and specifically without needing to copy over any files to the servers. I used a custom `URLClassLoader` to originally overcome this challenge, which is a vital component to the system. Although this allows the JVM to access the FTP server for `class` and `jar` file, it doesn't redirect the paths used for other kinds of files. For instance, the SchemaAnalyst test suite makes use of a `config` directory that contains various properties files. Although I could specify the exact FTP URL for the class loader to locate this directory, the test cases would not find it when they ran on the server. I tried to adjust the system property for the current working directory (`user.dir`), among other fixes, but could never bypass this error completely. Ultimately, in order to get the tests to work as expected, I need to add a step to the `updateClassLoader()` method on the `CustomServers` to retrieve and store some files from the FTP server. This was unfortunate, since a notable benefit of this system is the lack of resource overhead for the servers; but it was necessary in order to have a functional system within our time constraints. This general issue with file dependencies further complicated running DTS on complex systems like `ant-ivy` and `tomcat`, despite the accommodations made to DTS. The prevented DTS to be run on similar systems as TLB, and it would be one of the first areas of improvement if development on this system were to continue.

In addition to the challenges Michael encountered while implementing the DTS tool, there were similar challenges faced while configuring, analyzing and running the existing, Test Load Balancing tool.

First, while the system is on GitHub (`https://github.com/test-load-balancer/tlb`), it is not a working version. GitHub is initially where I found the tool, but in the tool repository the creators direct you to the tool website (`http://test-load-balancer.github.io/`). The tool website is filled with data and comments about the tool, however when it came time to configuring and running the tool, these processes were not made absolutely clear. No where on this page does it say they do not support actually running the clients in a distributed fashion.

Also, while there is a configuration page containing documentation for all of the necessary environment variables, the provided examples and descriptions are not clear at describing all of the variables. Luckily, we were able to reach out and hear back from the creators of this tool. They were able to guide us in the correct direction. But even without the help of the authors, we were able to find the answer to our question regarding how to configure the distribution of a test suite using the TLB tool was answered in a blog post by one of the creators of the TLB tool. Even here though, there were only one or two sentences on the topic. Maybe it was made clear somewhere else, but to me, this fact was not evident.

In order to run the TLB tool in a distributed fashion, we were required to use multiple nodes. We were able to accomplish this by creating containers—which are virtually easy-to-configure virtual machines with all of the necessary dependencies for running TLB. While we did run into one issue with Docker, that being installing Docker on a 32-bit machines, overall,

Docker was a pretty easy-to-use system, especially for Mac OS X.

As mentioned, the only issue we had with Docker directly was when trying to install it on an unsupported build of Windows. To combat this compatibility issue, we asked permission to have Docker installed on the Alden lab machines. If Docker were installed on the lab machines, our experiments would have been much more interesting, consistent and more detailed.

The final issue that I encountered was when trying to run experiments. I knew the TLB tool had been working the evening before trying to run the experiments and I had not manipulated any of the files and the TLB system was not communicating or running the test suites. I tried a number of ports to no avail. Finally, I decided to try the system at home once again and everything worked flawlessly, therefore requiring me to haul all of the necessary machines for the experiment to my house.

7 Conclusion

8 Appendix

8.1 Data

```
tool,number_nodes,trial,time
ant-ivy,3,1,381
ant-ivy,3,2,288
ant-ivy,3,3,284
ant-ivy,3,4,285
ant-ivy,3,5,303
ant-ivy,2,1,270
ant-ivy,2,2,287
ant-ivy,2,3,275
ant-ivy,2,4,277
ant-ivy,2,5,281
ant-ivy,1,1,338
ant-ivy,1,2,399
ant-ivy,1,3,380
ant-ivy,1,4,372
ant-ivy,1,5,375
schema-analyst,3,1,34
schema-analyst,3,2,34
schema-analyst,3,3,40
schema-analyst,3,4,34
schema-analyst,3,5,35
schema-analyst,2,1,8
schema-analyst,2,2,8
schema-analyst,2,3,10
schema-analyst,2,4,8
schema-analyst,2,5,8
schema-analyst,1,1,14
schema-analyst,1,2,13
schema-analyst,1,3,16
schema-analyst,1,4,13
schema-analyst,1,5,14
```

```
tool,number_nodes,trial,time
schema-analyst,4,1,153755
schema-analyst,4,2,138035
schema-analyst,4,3,150007
schema-analyst,4,4,159435
schema-analyst,4,5,152664
schema-analyst,3,1,143639
schema-analyst,3,2,132070
schema-analyst,3,3,139330
schema-analyst,3,4,131438
schema-analyst,3,5,114332
schema-analyst,2,1,109954
schema-analyst,2,2,105859
schema-analyst,2,3,107833
schema-analyst,2,4,107678
schema-analyst,2,5,104399
schema-analyst,1,1,134865
schema-analyst,1,2,142017
schema-analyst,1,3,186619
schema-analyst,1,4,172291
schema-analyst,1,5,142197
```

8.2 Representative Source Code

```
# FROM ubuntu
# MAINTAINER Colton McCurdy
# ADD tlb /tlb
# RUN apt-get update && \
#     apt-get upgrade -y && \
#     apt-get install -y  software-properties-common && \
#     add-apt-repository ppa:webupd8team/java -y && \
#     apt-get update && \
#     echo oracle-java7-installer shared/accepted-oracle-license-v1-1 select true | /usr/bin/debconf-set-
#     apt-get install -y oracle-java7-installer && \
#     apt-get clean
# RUN wget http://mirror.sdunix.com/apache//ant/binaries/apache-ant-1.9.4-bin.zip && \
#     unzip apache-ant-1.9.4-bin.zip && \
#     mv apache-ant-1.9.4/ /opt/ant && \
#     ln -s /opt/ant/bin/ant /usr/bin/ant && \
#     echo "#!/bin/bash
#     ANT_HOME=/opt/ant
#     PATH=$ANT_HOME/bin:$PATH
#     export PATH ANT_HOME
#     export CLASSPATH=." > /etc/profile.d/ant.sh && \
#     chmod +x /etc/profile.d/ant.sh && \
#     source /etc/profile.d/ant.sh
# RUN ant -version

FROM webratio/java:8
MAINTAINER Colton McCurdy
ADD tlb /tlb

# Installs vim
RUN ["apt-get", "update"]
RUN ["apt-get", "install", "-y", "vim"]

# Installs Ant
ENV ANT_VERSION 1.9.4
RUN cd && \
    wget -q http://archive.apache.org/dist/ant/binaries/apache-ant-${ANT_VERSION}-bin.tar.gz && \
    tar -xzf apache-ant-${ANT_VERSION}-bin.tar.gz && \
    mv apache-ant-${ANT_VERSION} /opt/ant && \
    rm apache-ant-${ANT_VERSION}-bin.tar.gz
ENV ANT_HOME /opt/ant
ENV PATH ${PATH}:/opt/ant/bin
```

```xml
<project name="schemaanalyst" default="" basedir=".">

    <property name="tlb.project.dir" location="${basedir}/../../../tlb"/><!-- used in dev box -->
    <property name="tlb.dist.dir" location="${basedir}/../../java"/><!-- used in packaged distribution(s)

    <!-- Target directories -->
    <property name="lib_dir" value="lib" />
    <property name="src_dir" location="src/" />
    <property name="build_dir" location="bin/" />

    <!-- SETUP VARIABLES AND PATHS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->

    <property name="config_dir" value="ftpserver/config/"/>

        <!-- Load the variable configurations. -->
        <loadproperties srcfile="${config_dir}/locations.properties" />
        <loadproperties srcfile="${config_dir}/database.properties" />

        <!-- The common classpath for building the system. -->
        <path id="build_classpath">
            <fileset dir="lib/" includes="*.jar" />
        </path>

        <!-- The classpath for running the system. -->
        <path id="run_classpath">
            <pathelement path="ftpserver/resources/bin/"/>
            <fileset dir="ftpserver/resources/lib/" includes="*.jar"/>
        </path>

    <!-- CLEANING TASK - "ant clean" -->
    <target name="clean">
        <delete dir="${build_dir}"/>
    </target>

    <!-- INIT TASK - "ant init" -->
    <target name="init">
        <classpath refid="build_classpath"/>
        <classpath refid="run_classpath"/>
    </target>

    <!-- COMPILING TASK - "ant compile" -->
    <target name="compile">
        <mkdir dir="${build_dir}"/>
        <javac srcdir="${src_dir}" destdir="${build_dir}" includeantruntime="false" debug="true" debuglev
            <classpath refid="build_classpath"/>
        </javac>
        <rmic classname="delegator.Delegator" base="${build_dir}"/>
    </target>

    <path id="load.balanced.classpath">
```

```xml
        <fileset dir="${tlb.project.dir}/java/lib" includes="*.jar" erroronmissingdir="false"/>
        <fileset dir="${tlb.project.dir}/target" includes="tlb-java*.jar" erroronmissingdir="false"/>
        <fileset dir="${tlb.dist.dir}/lib" includes="*.jar" erroronmissingdir="false"/>
        <fileset dir="${tlb.dist.dir}" includes="tlb-java*.jar" erroronmissingdir="false"/>
    </path>

    <target name="test.balanced">
        <!-- <typedef name="load-balanced-fileset" classname="tlb.ant.LoadBalancedFileSet" classpathref=".
        <typedef name="load-balanced-fileset" classname="tlb.ant.LoadBalancedFileSet" classpathref="load.l
        <!-- <junit failureproperty="test.failure" printsummary="no" haltonfailure="false" haltonerror="f
            <junit fork="true" dir = "ftpserver/" forkmode = "once">
                <test name="org.schemaanalyst.unittest.AllTests" />
                <test name="org.schemaanalyst.unittest.AllTests" />
                <test name="org.schemaanalyst.unittest.AllTests" />
                <test name="org.schemaanalyst.unittest.AllTests" />
                <test name="org.schemaanalyst.unittest.AllTests" />
                            <classpath refid="run_classpath" />
                <classpath refid="load.balanced.classpath"/>
                            <formatter type="brief" usefile="true" />
                        <!-- <classpath refid="run_classpath" /> -->
            <!-- <batchtest todir="£{reports.dir}"> -->
                <!-- <load-balanced-fileset dir="£{basedir}/ftpserver/resources/test_suites" includes="**
                <!-- <load-balanced-fileset dir="£{test-classes.dir}" includes="**/*Test.class"/> -->
                <formatter classname="tlb.ant.JunitDataRecorder"/>
                <formatter type="plain"/>
            <!-- </batchtest> -->
        </junit>
    </target>

    <!-- TESTING TASK - "ant test" -->
        <target name="test">
            <junit fork="true" dir = "ftpserver/" forkmode = "once">
                <test name="org.schemaanalyst.unittest.AllTests" />
                <test name="org.schemaanalyst.unittest.AllTests" />
                <test name="org.schemaanalyst.unittest.AllTests" />
                <test name="org.schemaanalyst.unittest.AllTests" />
                <test name="org.schemaanalyst.unittest.AllTests" />
                    <classpath refid="run_classpath" />
                    <formatter type="brief" usefile="true" />
            </junit>
        </target>
</project>
```

```bash
#!/bin/bash

export TLB_JOB_NAME='schemaanalyst'
export TLB_BASE_URL='http://192.168.1.139:7019'
export TLB_TOTAL_PARTITIONS=3
export TLB_PARTITION_NUMBER=1
export TEST_TASK='ant test.balanced'
export ALL_PARTITIONS_RAN_VERIFICATION_TASK='ant assert.all.partitions.executed'

source ../recipe.sh

#!/bin/bash

if [[ $1 = no-verbosity ]]; then
    verbose=false
else
    verbose=true
fi

dev_lib_dir=../../tlb/target
dist_lib_dir=../../server

if [ -e $dev_lib_dir ]; then
    export TLB_JAR=`ls $dev_lib_dir/tlb-server*.jar`
else
    export TLB_JAR=`ls $dist_lib_dir/tlb-server*.jar`
fi

export TLB_OUT_FILE=/tmp/tlb_balancer.out
export TLB_ERR_FILE=/tmp/tlb_balancer.err
export TLB_BASE_URL='http://192.168.1.139:7019'
export TLB_JOB_VERSION=`date | sed -e 's# #-#g'`

    $TEST_TASK
```

```xml
<!--
   Licensed to the Apache Software Foundation (ASF) under one
   or more contributor license agreements.  See the NOTICE file
   distributed with this work for additional information
   regarding copyright ownership.  The ASF licenses this file
   to you under the Apache License, Version 2.0 (the
   "License"); you may not use this file except in compliance
   with the License.  You may obtain a copy of the License at

     http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing,
   software distributed under the License is distributed on an
   "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
   KIND, either express or implied.  See the License for the
   specific language governing permissions and limitations
   under the License.
-->
<project name="ivy" default="coverage-report" basedir="." xmlns:ivy="antlib:org.apache.ivy.ant">

    <!-- <property name="tlb.project.dir" location="£{basedir}/../tlb"/> -->
    <!-- <property name="tlb.dist.dir" location="£{basedir}/../tlb/java"/> -->

    <property name="tlb.project.dir" location="${basedir}/../../tlb"/><!-- used in dev box -->
    <property name="tlb.dist.dir" location="${basedir}/../../java"/><!-- used in packaged distribution(s)

    <!-- <path id="src.classpath"> -->
    <!--     <pathelement path="£{classes.dir}"/> -->
    <!-- </path> -->

    <!-- <path id="test.classpath"> -->
    <!--     <path refid="src.classpath"/> -->
    <!--     <pathelement path="£{test-classes.dir}"/> -->
    <!--     <fileset dir="£{test.lib.dir}" includes="*.jar"/> -->
    <!-- </path> -->

        <property environment="env" />
    <property file="version.properties" />
    <property file="build.properties" />

    <property name="final.name" value="ivy.jar" />

    <target name="init-ivy-user-home" unless="ivy.use.local.home">
        <condition property="ivy.home" value="${env.IVY_HOME}">
            <isset property="env.IVY_HOME" />
        </condition>
        <property name="ivy.home" value="${user.home}/.ivy2" />
    </target>

    <target name="init-ivy-local-home" if="ivy.use.local.home">
```

```xml
        <property name="ivy.home" value="${basedir}/.ivy2" />
    </target>

    <target name="init-ivy-home" depends="init-ivy-user-home, init-ivy-local-home" />

    <target name="init-ivy" depends="compile-bootstrap, init-ivy-home">
        <taskdef resource="org/apache/ivy/ant/antlib.xml"
                uri="antlib:org.apache.ivy.ant">
            <classpath>
                <pathelement location="${core.classes.build.dir}" />
                <pathelement location="${bootstrap.classes.build.dir}" />
            </classpath>
        </taskdef>

        <ivy:configure override="true" />
    </target>

    <target name="install" depends="init-ivy-home, jar"
            description="build Ivy and install it in Ivy user home for builds using Ivy user home to load
        <property name="ivy.jar.file" value="${ivy.home}/jars/ivy.jar" />
        <copy file="${artifacts.build.dir}/jars/${final.name}" tofile="${ivy.jar.file}" />
    </target>

    <target name="install-ant" depends="init-ivy-home, jar"
            description="build Ivy and install it in Ant home lib">
        <condition property="ant.home" value="${env.ANT_HOME}">
          <isset property="env.ANT_HOME" />
        </condition>
        <fail unless="ant.home" message="ANT_HOME environment variable or ant.home property required" />
        <copy file="${artifacts.build.dir}/jars/${final.name}" tofile="${ant.home}/lib/ivy.jar" />
    </target>

    <!-- ==================================================================
         PREPARATION, CLEANING AND FLAGS TASKS
         ================================================================== -->
    <target name="init">
        <path id="lib.classpath">
            <fileset dir="${lib.dir}">
                <include name="*.jar" />
            </fileset>
        </path>
        <path id="build.bootstrap.classpath">
            <pathelement location="${core.classes.build.dir}" />
        </path>
        <path id="build.ant.classpath">
            <pathelement location="${core.classes.build.dir}" />
            <path refid="lib.classpath" />
        </path>
        <path id="build.optional.classpath">
            <path refid="build.ant.classpath" />
```

```xml
        </path>
        <path id="run.classpath">
            <pathelement location="${core.classes.build.dir}" />
            <pathelement location="${ant.classes.build.dir}" />
            <pathelement location="${optional.classes.build.dir}" />
            <path refid="lib.classpath" />
        </path>
        <path id="test.classpath">
            <pathelement location="${coverage.classes.dir}" />
            <fileset dir="${lib.dir}">
                <include name="*.jar" />
                <exclude name="ant.jar" />
                <exclude name="ant-launcher.jar" />
                    <exclude name="ant-nodeps.jar"/>
                    <exclude name="ant-trax.jar"/>
            </fileset>
            <!-- <fileset dir="£{test.lib.dir}" includes="*.jar"/> -->
            <pathelement location="${core.classes.build.dir}" />
            <pathelement location="${ant.classes.build.dir}" />
            <pathelement location="${optional.classes.build.dir}" />
            <pathelement path="${test.build.dir}" />
        </path>
</target>

<target name="prepare" depends="init">
    <mkdir dir="${classes.build.dir}" />
    <mkdir dir="${core.classes.build.dir}" />
        <mkdir dir="${bootstrap.classes.build.dir}" />
    <mkdir dir="${ant.classes.build.dir}" />
    <mkdir dir="${optional.classes.build.dir}" />
    <mkdir dir="${all.classes.build.dir}" />
    <mkdir dir="${test.build.dir}" />
    <mkdir dir="${artifacts.build.dir}" />
    <mkdir dir="${test.report.dir}" />
        <mkdir dir="${ivy.report.dir}" />
</target>

<target name="clean" description="delete all generated files keeping sources only">
    <delete dir="${classes.build.dir}" />
    <delete dir="${test.build.dir}" />
    <delete dir="${artifacts.build.dir}" />
    <delete dir="${test.report.dir}" />
    <delete dir="${javadoc.build.dir}" />
    <delete dir="${doc.build.dir}" />
    <delete dir="${build.dir}" />
</target>

<target name="clean-lib">
    <delete dir="${lib.dir}" />
</target>
```

```xml
<target name="clean-ivy-cache" depends="init-ivy-home">
    <delete dir="${ivy.home}/cache" />
</target>

<target name="clean-ivy-home" depends="init-ivy-home">
    <delete dir="${ivy.home}" />
</target>

<target name="clean-examples" description="clean all examples">
    <subant target="clean" failonerror="false">
        <fileset dir="${example.dir}" includes="**/build.xml" />
    </subant>
</target>

<target name="clean-all" depends="clean, clean-lib, clean-examples" />

<target name="/noresolve" description="use to skip dependency resolution">
    <property name="no.resolve" value="true" />
</target>

<target name="/notest" description="use to skip tests">
    <property name="skip.test" value="true" />
</target>

<target name="/nojavadoc" description="use to skip javadoc">
    <property name="skip.javadoc" value="true" />
</target>

<target name="/localivy" description="use a local ivy home">
    <property name="ivy.use.local.home" value="true" />
</target>

<target name="/offline" depends="/noresolve" description="use to indicate no internet connection is a
    <property name="offline" value="true" />
</target>

<target name="default-version">
    <tstamp>
        <format property="pubdate" pattern="yyyyMMddHHmmss" />
    </tstamp>
    <property name="version.prefix" value="${target.ivy.version}-local-" />
    <property name="build.version" value="${version.prefix}${pubdate}" />
    <property name="bundle.version" value="${target.ivy.bundle.version}.${target.ivy.bundle.version.q
</target>

<!-- ================================================================
     DEPENDENCY MNGT, COMPILATION AND JAR
     ================================================================ -->
<target name="resolve" depends="init-ivy" unless="no.resolve">
```

```xml
        <ivy:retrieve conf="default,test" pattern="${lib.dir}/[artifact].[ext]" sync="yes"/>
</target>

<target name="compile-core" depends="prepare">
    <javac  srcdir="${src.dir}"
            destdir="${core.classes.build.dir}"
            sourcepath=""
            source="${ivy.minimum.javaversion}"
            target="${ivy.minimum.javaversion}"
            debug="${debug.mode}"
            includeantruntime="no">
        <excludesfile name="ant.patterns" />
        <excludesfile name="optional.patterns" />
    </javac>
    <copy todir="${core.classes.build.dir}" includeEmptyDirs="false">
        <fileset dir="${src.dir}">
            <exclude name="**/*.java" />
            <excludesfile name="ant.patterns" />
            <excludesfile name="optional.patterns" />
        </fileset>
    </copy>

    <!-- copy settings files for backward compatibility with ivyconf naming -->
    <copy file="${core.classes.build.dir}/org/apache/ivy/core/settings/ivysettings-local.xml"
        tofile="${core.classes.build.dir}/org/apache/ivy/core/settings/ivyconf-local.xml" />
    <copy file="${core.classes.build.dir}/org/apache/ivy/core/settings/ivysettings-default-chain.xml"
        tofile="${core.classes.build.dir}/org/apache/ivy/core/settings/ivyconf-default-chain.xml" />
    <copy file="${core.classes.build.dir}/org/apache/ivy/core/settings/ivysettings-main-chain.xml"
        tofile="${core.classes.build.dir}/org/apache/ivy/core/settings/ivyconf-main-chain.xml" />
    <copy file="${core.classes.build.dir}/org/apache/ivy/core/settings/ivysettings-public.xml"
        tofile="${core.classes.build.dir}/org/apache/ivy/core/settings/ivyconf-public.xml" />
    <copy file="${core.classes.build.dir}/org/apache/ivy/core/settings/ivysettings-shared.xml"
        tofile="${core.classes.build.dir}/org/apache/ivy/core/settings/ivyconf-shared.xml" />
    <copy file="${core.classes.build.dir}/org/apache/ivy/core/settings/ivysettings.xml"
        tofile="${core.classes.build.dir}/org/apache/ivy/core/settings/ivyconf.xml" />
</target>

    <!-- Build the Ant tasks with the current Ant runtime -->
<target name="compile-bootstrap" depends="compile-core">
    <javac  srcdir="${src.dir}"
            destdir="${bootstrap.classes.build.dir}"
            sourcepath=""
            classpathref="build.bootstrap.classpath"
            source="${ivy.minimum.javaversion}"
            target="${ivy.minimum.javaversion}"
            debug="${debug.mode}"
            includeantruntime="yes">
        <includesfile name="ant.patterns" />
    </javac>
    <copy todir="${bootstrap.classes.build.dir}" includeEmptyDirs="false">
```

```xml
            <fileset dir="${src.dir}">
                <includesfile name="ant.patterns" />
                <exclude name="**/*.java" />
            </fileset>
        </copy>
    </target>

    <!-- Build the Ant tasks with the minimal Ant runtime -->
    <target name="compile-ant" depends="compile-core, resolve">
        <javac  srcdir="${src.dir}"
                destdir="${ant.classes.build.dir}"
                sourcepath=""
                classpathref="build.ant.classpath"
                source="${ivy.minimum.javaversion}"
                target="${ivy.minimum.javaversion}"
                debug="${debug.mode}"
                includeantruntime="no">
            <includesfile name="ant.patterns" />
        </javac>
        <copy todir="${ant.classes.build.dir}" includeEmptyDirs="false">
            <fileset dir="${src.dir}">
                <includesfile name="ant.patterns" />
                <exclude name="**/*.java" />
            </fileset>
        </copy>

        <!-- copy antlib for backward compatibility with fr.jayasoft.ivy package -->
        <copy file="${ant.classes.build.dir}/org/apache/ivy/ant/antlib.xml"
                todir="${ant.classes.build.dir}/fr/jayasoft/ivy/ant" />
    </target>

    <target name="compile-optional" depends="compile-ant, resolve">
        <javac  srcdir="${src.dir}"
                destdir="${optional.classes.build.dir}"
                sourcepath=""
                classpathref="build.optional.classpath"
                source="${ivy.minimum.javaversion}"
                target="${ivy.minimum.javaversion}"
                debug="${debug.mode}"
                includeantruntime="no">
            <includesfile name="optional.patterns" />
            <includesfile name="ant.patterns" />
        </javac>
        <copy todir="${core.classes.build.dir}" includeEmptyDirs="false">
            <fileset dir="${src.dir}">
                <includesfile name="optional.patterns" />
                <exclude name="**/*.java" />
            </fileset>
        </copy>
    </target>
```

```xml
<!-- ================================================================
     Create the two jar files (Ivy core and Ivy core + Ant tasks)
     ================================================================ -->
<target name="jar" depends="compile-optional, default-version" description="Create Jar files">
    <!-- identify compiled ivy version -->
    <echo message="version=${build.version}${line.separator}" file="${core.classes.build.dir}/module.p
    <echo message="date=${pubdate}${line.separator}" file="${core.classes.build.dir}/module.propertie

        <mkdir dir="${artifacts.build.dir}/jars/"/>

        <!--
                there is a default Bundle-Version attribute in the source MANIFEST, used to ease
                development in eclipse.
                We remove this line to make sure we get the Bundle-Version as set in the jar task
        -->
        <copy file="${basedir}/META-INF/MANIFEST.MF" tofile="${artifacts.build.dir}/MANIFEST.MF">
                <filterchain>
                        <replaceregex pattern="Bundle-Version:.*" replace="Bundle-Version: ${bundle.ve
                </filterchain>
        </copy>

    <copy todir="${all.classes.build.dir}">
        <fileset dir="${core.classes.build.dir}" />
        <fileset dir="${ant.classes.build.dir}" />
        <fileset dir="${optional.classes.build.dir}" />
    </copy>

    <jar destfile="${artifacts.build.dir}/jars/${final.name}"
            manifest="${artifacts.build.dir}/MANIFEST.MF">
        <metainf dir="${basedir}" includes="LICENSE,NOTICE" />
        <manifest>
            <attribute name="Specification-Title" value="Apache Ivy with Ant tasks" />
            <attribute name="Specification-Version" value="${build.version}" />
            <attribute name="Specification-Vendor" value="Apache Software Foundation" />
            <attribute name="Implementation-Title" value="org.apache.ivy" />
            <attribute name="Implementation-Version" value="${build.version}" />
            <attribute name="Implementation-Vendor" value="Apache Software Foundation" />
            <attribute name="Implementation-Vendor-Id" value="org.apache" />
            <attribute name="Extension-name" value="org.apache.ivy" />
            <attribute name="Build-Version" value="${build.version}" />
        </manifest>
        <fileset dir="${all.classes.build.dir}" />
    </jar>
        <!-- copy main jar to ease its use as an OSGi bundle -->
        <copy file="${artifacts.build.dir}/jars/${final.name}"
                tofile="${artifacts.build.dir}/org.apache.ivy_${bundle.version}.jar" />

    <!-- clean generated module properties file -->
    <delete file="${core.classes.build.dir}/module.properties" />
```

26

```xml
    </target>

    <!-- ================================================================
         PUBLISH LOCAL
         ================================================================ -->
<target name="publish-local" depends="jar,sources" description="publishes Ivy to Ivy local repository
    <ivy:publish resolver="local" pubrevision="${build.version}"
        artifactsPattern="${artifacts.build.dir}/[type]s/[artifact].[ext]"
        forcedeliver="true" />
</target>

    <!-- ================================================================
         TESTS
         ================================================================ -->
<target name="build-custom-resolver-jar" depends="jar">
    <mkdir dir="${build.dir}/custom-classpath" />
    <javac  srcdir="${basedir}/test/custom-classpath"
            destdir="${build.dir}/custom-classpath"
            classpathref="run.classpath"
            source="${ivy.minimum.javaversion}"
            target="${ivy.minimum.javaversion}"
            debug="${debug.mode}"
            includeantruntime="no" />
    <jar    destfile="${test.dir}/org/apache/ivy/core/settings/custom-resolver.jar"
            basedir="${build.dir}/custom-classpath" />
</target>

<target name="init-tests-offline" if="offline">
        <fileset id="test.fileset" dir="${test.dir}">
            <include name="**/${test.class.pattern}.java" />
            <exclude name="**/Abstract*Test.java" />
            <not><contains text="remote.test" /></not>
        </fileset>
</target>

<target name="init-tests-online" unless="offline">
        <fileset id="test.fileset" dir="${test.dir}">
            <include name="**/${test.class.pattern}.java" />
            <exclude name="**/Abstract*Test.java" />
        </fileset>
</target>

<target name="init-tests" depends="init-tests-offline, init-tests-online" />

<target name="emma" depends="jar" unless="skip.test">
    <ivy:cachepath organisation="emma" module="emma" revision="2.0.5312"
                   inline="true" conf="default" pathid="emma.classpath"
                   log="download-only" />
    <ivy:cachepath organisation="emma" module="emma_ant" revision="2.0.5312"
                   inline="true" conf="default" pathid="emma.ant.classpath" transitive="false"
```

```xml
                    log="download-only" />
        <taskdef resource="emma_ant.properties">
            <classpath refid="emma.classpath" />
            <classpath refid="emma.ant.classpath" />
        </taskdef>
        <property name="emma.enabled" value="true" />
        <property name="coverage.dir" value="${build.dir}/coverage" />
        <property name="coverage.classes.dir" value="${coverage.dir}/classes" />
        <mkdir dir="${coverage.dir}" />
        <mkdir dir="${coverage.classes.dir}" />
        <emma enabled="${emma.enabled}">
          <instr mode="copy"
                destdir="${coverage.dir}/classes"
                metadatafile="${coverage.dir}/metadata.emma">
            <instrpath>
                <pathelement location="${core.classes.build.dir}" />
                <pathelement location="${ant.classes.build.dir}" />
                <pathelement location="${optional.classes.build.dir}" />
            </instrpath>
          </instr>
        </emma>
        <delete file="${coverage.dir}/coverage.emma" />
        <!-- add emma path to test path, because emma jars need to be available when running
             instrumented classes -->
        <ivy:addpath topath="test.classpath" first="true">
          <pathelement location="${coverage.dir}/classes" />
          <path refid="emma.classpath" />
        </ivy:addpath>
    </target>

    <target name="build-test" depends="jar">
        <javac  srcdir="${test.dir}"
                destdir="${test.build.dir}"
                classpathref="run.classpath"
                source="${ivy.minimum.javaversion}"
                target="${ivy.minimum.javaversion}"
                debug="${debug.mode}"
                encoding="ISO-8859-1"
                includeantruntime="no" />
        <copy todir="${test.build.dir}">
            <fileset dir="${test.dir}">
                <exclude name="**/*.java" />
            </fileset>
        </copy>
    </target>

    <target name="prepare-osgi-tests" depends="resolve" unless="skip.test">
    <ant dir="${basedir}/test/test-repo" target="generate-bundles" />
    </target>
```

```
<target name="prepare-test-jar-repositories" unless="skip.test">
        <mkdir dir="${basedir}/test/jar-repos" />
    <jar destfile="${basedir}/test/jar-repos/jarrepo1.jar" >
        <fileset dir="${basedir}/test/repositories/1" />
    </jar>
    <jar destfile="${basedir}/test/jar-repos/jarrepo1_subdir.jar">
        <fileset dir="${basedir}/test/repositories" includes="1/**/*" />
    </jar>
</target>

    <target name="test-internal" depends="build-test, init-tests, prepare-osgi-tests, prepare-test-ja
    <mkdir dir="${test.xml.dir}" />

    <junit
        haltonfailure="off"
        haltonerror="off"
        errorproperty="test.failed"
        failureproperty="test.failed"
        showoutput="no"
        printsummary="yes"
        includeantruntime="yes"
        dir="${basedir}"
        fork="true">
         <classpath>
            <path refid="test.classpath" />
            <pathelement path="${ant.home}/lib/ant-nodeps.jar"/>
            <pathelement path="${ant.home}/lib/ant-trax.jar"/>
         </classpath>

            <!-- pass the proxy properties to the forked junit process to use correct proxy -->
            <syspropertyset>
                    <propertyref prefix="http" />
                </syspropertyset>
        <jvmarg value="-Demma.coverage.out.file=${coverage.dir}/coverage.emma" />
        <jvmarg value="-Demma.coverage.out.merge=true" />

        <!-- Added this to test IVY-65 -->
        <jvmarg value="-Duser.region=TR" />
        <jvmarg value="-Duser.language=tr" />

        <formatter type="xml"/>
        <batchtest todir="${test.xml.dir}">
            <fileset refid="test.fileset" />
        </batchtest>
    </junit>
</target>

<target name="test" depends="test-internal" description="Run the test">
    <fail if="test.failed"
            message="At least one test has failed. See logs (in ${test.xml.dir}) for details (use the ta
```

```xml
    </target>

    <!-- ====================================================================
         REPORTS AND DOCUMENTATION
         ==================================================================== -->
    <target name="test-report" depends="test-internal" unless="skip.test">
        <junitreport todir="${test.xml.dir}">
            <fileset dir="${test.xml.dir}">
                <include name="TEST-*.xml" />
            </fileset>
            <report format="frames" todir="${test.report.dir}" />
        </junitreport>
        <fail if="test.failed"
              message="At least one test has failed. See logs (in ${test.xml.dir}) or report (in ${test.re
    </target>

    <target name="coverage-report" depends="emma, test-report"  unless="skip.test"
            description="run tests with instrumentation and generate coverage report">
        <mkdir dir="${coverage.report.dir}" />
        <emma>
            <report sourcepath="${src.dir}">
                <fileset dir="${coverage.dir}">
                    <include name="*.emma" />
                </fileset>

                <txt outfile="${coverage.report.dir}/coverage.txt" />
                <html outfile="${coverage.report.dir}/coverage.html" />
            </report>
        </emma>
    </target>

    <target name="ivy-report" depends="resolve">
        <ivy:report todir="${ivy.report.dir}"/>
    </target>

    <target name="javadoc" unless="skip.javadoc">
        <javadoc destdir="${javadoc.build.dir}" useexternalfile="true">
            <fileset dir="${src.dir}" includes="**/*.java" />
        </javadoc>
    </target>

    <target name="sources" depends="default-version" description="Create source archive files">
        <mkdir dir="${artifacts.build.dir}/sources/"/>
        <jar destfile="${artifacts.build.dir}/sources/${final.name}">
            <metainf dir="${basedir}" includes="LICENSE,NOTICE" />
            <manifest>
                <attribute name="Specification-Title" value="Apache Ivy Sources" />
                <attribute name="Specification-Version" value="${build.version}" />
                <attribute name="Specification-Vendor" value="Apache Software Foundation" />
            </manifest>
```

```xml
                <fileset dir="${src.dir}" />
        </jar>
    </target>


<target name="fixcrlf">
    <property name="eol.native.includes"
        value="**/*.html,**/*.json,**/*.java,**/*.xml,**/*.txt,**/*.MF,**/*.properties,**/*.patterns,
    <property name="eol.native.excludes"
        value="build/**,bin/**,lib/**" />

    <fileset id="eol.native.fileset"
        dir="${basedir}"
        includes="${eol.native.includes}"
        excludes="${eol.native.excludes}" />

    <fixcrlf srcdir="${basedir}"
        includes="${eol.native.includes}"
        excludes="${eol.native.excludes}" />
    <apply executable="svn">
        <fileset refid="eol.native.fileset" />
        <arg value="propset" />
        <arg value="svn:eol-style" />
        <arg value='"native"' />
    </apply>
</target>


<path id="load.balanced.classpath">
    <path refid="test.classpath"/>
    <fileset dir="${tlb.project.dir}/java/lib" includes="*.jar" erroronmissingdir="false"/>
    <fileset dir="${tlb.project.dir}/target" includes="tlb-java*.jar" erroronmissingdir="false"/>
    <fileset dir="${tlb.dist.dir}/lib" includes="*.jar" erroronmissingdir="false"/>
    <fileset dir="${tlb.dist.dir}" includes="tlb-java*.jar" erroronmissingdir="false"/>
</path>

    <!-- Change the 'load.balanced.classpath' so that it is your test classpath along with the TLB jar
its dependencies. You can also tweak the 'depends' to fix the task dependencies of your build.
 You can change the fileset's includes pattern to include your tests. -->
<target name="test.balanced" depends="init, build-test, init-tests, prepare-osgi-tests, prepare-test-
    <!-- <typedef name="load-balanced-fileset" classname="tlb.ant.LoadBalancedFileSet" classpathref="
    <typedef name="load-balanced-fileset" classname="tlb.ant.LoadBalancedFileSet" classpathref="load.
    <junit failureproperty="test.failure" printsummary="no" haltonfailure="false" haltonerror="false"
            showoutput="false" fork="true">
        <classpath refid="load.balanced.classpath"/>
        <batchtest todir="${reports.dir}">
            <load-balanced-fileset dir="${basedir}/build/test" includes="**/*Test.class"/>
            <!-- <load-balanced-fileset dir="£{test-classes.dir}" includes="**/*Test.class"/> -->
            <formatter classname="tlb.ant.JunitDataRecorder"/>
            <formatter type="plain"/>
        </batchtest>
    </junit>
```

```xml
</target>

<!-- Checks Ivy codebase according to £{checkstyle.src.dir}/checkstyle-config  -->
<target name="checkstyle-internal" depends="jar">
    <ivy:cachepath organisation="checkstyle" module="checkstyle" revision="5.0"
            inline="true" conf="default" pathid="checkstyle.classpath" transitive="true"
                log="download-only"/>
    <taskdef resource="checkstyletask.properties" classpathref="checkstyle.classpath" />

    <mkdir dir="${checkstyle.report.dir}" />
    <checkstyle config="${checkstyle.src.dir}/checkstyle-config"
            failOnViolation="false" failureProperty="checkstyle.failed">
            <classpath>
                    <path refid="run.classpath" />
            </classpath>
        <formatter type="xml" toFile="${checkstyle.report.dir}/checkstyle.xml" />
        <fileset dir="${src.dir}">
            <include name="**/*.java" />
        </fileset>
        <fileset dir="${example.dir}">
            <include name="**/*.java" />
        </fileset>
    </checkstyle>
      </target>

<target name="checkstyle" depends="checkstyle-internal" description="checks Ivy codebase according to
        <fail if="checkstyle.failed"
                message="Checkstyle has errors. See report in ${checkstyle.report.dir}" />
</target>

<target name="checkstyle-report" depends="checkstyle-internal">
    <property name="checkstyle.basedir" location="${src.dir}" />
    <xslt in="${checkstyle.report.dir}/checkstyle.xml"
            style="${checkstyle.src.dir}/checkstyle-frames.xsl"
            out="${checkstyle.report.dir}/output.txt">
        <param name="basedir" expression="${checkstyle.basedir}" />
    </xslt>
</target>

<target name="init-findbugs" unless="findbugs.home">
    <!-- Findbugs: Getting Findbugs -->
    <property name="findbugs.download.name"
                value="findbugs-1.3.5"
                description="Name of the download file without suffix. Also the internal root directory
    <property name="findbugs.download.file"
                value="${findbugs.download.name}.zip"
                description="The filename of the ZIP."/>
    <property name="findbugs.download.url"
                value="http://garr.dl.sourceforge.net/sourceforge/findbugs/${findbugs.download.file}"
                description="The download adress at a mirror of Sourceforge."/>
```

```xml
        <property name="findbugs.download.to"
                  value="${build.dir}/.downloads"
                  description="Where to store the download and 'install' Findbugs."/>
        <available
          property="findbugs.home"
          value="${findbugs.download.to}/${findbugs.download.name}"
          file="${findbugs.download.to}/${findbugs.download.name}/lib/findbugs.jar"
          description="Check if Findbugs is already installed."
        />

        <!-- Findbugs: Running Findbugs -->
        <property name="findbugs.reportdir"
                  location="${reports.dir}/findbugs"
                  description="Where to store Findbugs results"/>
        <property name="findbugs.raw"
                  value="raw.xml"
                  description="Findbugs Output xml-file"/>
        <property name="findbugs.xsl"
                  value="fancy.xsl"
                  description="Which XSL to use for generating Output: default, fancy, plain, summary"/>
        <property name="findbugs.jvmargs"
                  value="-Xms128m -Xmx512m"
                  description="JVMArgs for invoking Findbugs"/>

        <mkdir dir="${findbugs.download.to}"/>
        <get src="${findbugs.download.url}" dest="${findbugs.download.to}/${findbugs.download.file}"/>
        <unzip src="${findbugs.download.to}/${findbugs.download.file}" dest="${findbugs.download.to}"/>
        <property name="findbugs.home" location="${findbugs.download.to}/${findbugs.download.name}"/>
        <mkdir dir="${findbugs.home}/plugin"/>
</target>

<target name="findbugs" description="checks Ivy codebase with Findbugs" depends="init-findbugs,compil
        <path id="findbugs.real.classpath">
            <fileset dir="${findbugs.home}/lib" includes="*.jar"/>
        </path>

        <!-- Load the Findbugs AntTasks -->
        <taskdef uri="http://findbugs.sourceforge.net/" resource="edu/umd/cs/findbugs/anttask/tasks.prope

        <!-- Start Findbugs -->
        <mkdir dir="${findbugs.reportdir}"/>
        <fb:findbugs home="${findbugs.home}"
                     classpathref="findbugs.real.classpath"
                     output="xml:withMessages"
                     outputFile="${findbugs.reportdir}/${findbugs.raw}"
                     jvmargs="${findbugs.jvmargs}"
                     projectName="${Name} ${project.version}">
            <class location="${core.classes.build.dir}" />
            <sourcePath path="${src.dir}" />
        </fb:findbugs>
```

```xml
        <!-- Generate (human) readable output -->
        <xslt basedir="${findbugs.reportdir}" includes="${findbugs.raw}" destdir="${findbugs.reportdir}">
            <style>
                <javaresource name="${findbugs.xsl}" classpathref="findbugs.real.classpath"/>
            </style>
        </xslt>
    </target>


    <!-- ================================================================
         IDE SPECIFIC
         ================================================================ -->
    <available file="${basedir}/.classpath" property="eclipse.classpath.exists" />
    <target name="check-eclipse-classpath-overwrite" if="eclipse.classpath.exists">
        <input message=".classpath file already exists.${line.separator}Are you sure you want to overwrite
               validargs="Y,N,y,n" addproperty="eclipse.classpath.confirm" />
        <condition property="eclipse.classpath.abort">
            <equals arg1="${eclipse.classpath.confirm}" arg2="N" casesensitive="false" />
        </condition>
    </target>


    <target name="eclipse-default" depends="resolve, check-eclipse-classpath-overwrite"
            unless="eclipse.classpath.abort"
            description="creates a default .classpath for eclipse, using jars resolved by this ant build">
        <copy file="${basedir}/.classpath.default" tofile="${basedir}/.classpath" overwrite="true" />
    </target>


    <target name="eclipse-ivyde" depends="check-eclipse-classpath-overwrite"
            unless="eclipse.classpath.abort"
            description="creates a .classpath for eclipse using Apache IvyDE version">
        <copy file="${basedir}/.classpath.ivyde" tofile="${basedir}/.classpath" overwrite="true" />
    </target>
</project>
```

References

[1] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P Wright. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS)*, 4(2):5, 2008.