A Distributed System for Continuous Integration with JINI\(^1\)

Y. C. Cheng, P.-H. Ou, C.-T. Chen and T.-S. Hsu

Software Systems Lab
Department of Computer Science and Information Engineering
National Taipei University of Technology
Email: \{yccheng,t5598036,s1669021,s4599002\}@ntut.edu.tw

Abstract

The present work extends a previously developed continuous integration system to support platform dependent builds and to reduce turnaround time by incorporating distributed computing capabilities. By adopting the Replicated-Worker Pattern as the distributed architecture and using JINI distributed technology as the means of implementation, both requirements are achieved. Experimental results in support of the claims are presented.

1. Introduction

Continuous Integration (CI) is a software development best practice to ensure software quality and maintain system stability [10]. Typical tasks performed in CI include compilation, unit tests, integration tests, and static code analysis. By integrating different software components periodically and frequently, CI allows developers to obtain feedbacks on the latest integration build for early discovery and correction of bugs. In practice, continuous integration requires tool support. Many tools have been developed, such as CruiseControl by ThoughWorks [6], Apache Gump [2] and Continuum [3], and our own Java Continuous Integration System (JCIS) [13]. Besides the functional requirements, CI tools must meet a number of quality requirements, including extensibility (adding new integration tasks), availability (providing integration service reliably), maintainability (managing platform dependent builds), performance (reducing turn-around time), and so on.

In this paper, we seek to achieve maintainability and performance for JCIS. In terms of maintainability, existing CI tools perform integration builds on a single host. This means that platform dependent builds (e.g., builds of a Java application that runs on Windows and Linux, respectively) require the CI software to be installed on multiple platforms. An immediate consequence is that platform dependent builds become unrelated builds that must be managed separately, which creates a maintainability problem. In terms of performance, builds that take a long time to complete constitute bottlenecks in the use of existing CI tools; the long waiting time hampers the developer’s willingness to run continuous integration frequently. In JCIS, both requirements are viewed as heterogeneous distributed computing problems, which are solved by applying the Replicated-Worker Pattern [7] and the JINI technology [9].

2. Related Works

We survey several well-known open source CI projects, including CruiseControl [6], Anthill OS [1], Gump [2] and Continuum [3]. The open source projects will be introduced and compared with JCIS in the following list.

- CruiseControl supports both Ant and Maven. CruiseControl lets user customize the building process. Multiple hosts can be used to perform distributed builds to cut the turn-around time provided each of the hosts has a copy of CruiseControl installed. Reports on builds are collected to a central dashboard to show overall result. CruiseControl is not based on distributed architecture.
- AntHill is based on Ant and is easy to install and configure. The open source version of AntHill does not support distributed builds.
- Gump supports both Ant and Maven. It allows dependency to be specified among several projects. Gump does not support distributed builds.
- Continuum supports Ant and Maven and is easy to install and use. It provides a web portal for managing integration builds. Continuum does not support distributed builds.

Our survey shows that these open source CI tools are inadequate in achieving the maintainability and performance requirements. This motivates us to extend JCIS using an appropriate distributed computing technology.

\(^1\) Accepted by the 2008 International Conference on Distributed Multimedia Systems (DMS 2008), September 4-6, 2008, Boston, USA
3. System Architecture and Design

From our previous work, JCIS was developed to provide automatic continuous integration. JCIS is extensible through its IBuilder interface. For example, both C++ and Java program are supported by adding the corresponding compilers to JCIS through the IBuilder interface. Figure 1 extends JCIS integration scenario by including distributed computing:

1a. Developers write, compile and test the program on their own computers.
2a. Developers commit their programs to the version control system after passing all unit tests.
1b. Testers check out the current version of the program from the version control system, write and execute tests, e.g., integration tests and functional tests.
2b. Testers commit the test code to the version control system after pass all the test cases.
3. JCIS checks out the current version of the program and test code automatically, and starts the integration builds.
4. Customized integration builds are performed according to project specification. The basic flow includes compiling and testing. Additional builds such as various static analyses are available from JCIS as well.
5~7. JCIS dispatches the building jobs to different integration stations. After all of the integration stations finish their jobs, JCIS aggregates the building results to produce an integration report.

8. JCIS publishes all the artifacts, including the object code, documents and the integration report to the portal.
9. Project Manager, Developers and Testers view the integration report from the portal with a browser.

The Replicated-Worker Pattern [7] is applied to provide the distributed computing capability (Figure 2). This pattern defines three roles: Master, Worker, and Space. The master works with several workers. It is responsible for dividing the integration task to several smaller sub-tasks and for putting the sub-tasks to the central space. The workers then take the sub-tasks from the space and execute the sub-tasks. Upon finishing, a worker writes the execution result back to the space. Finally, when all sub-tasks are completed, the master collects and aggregates these execution results.

Figure 1: the process of the distributed CI system

Figure 2: Replicated-Worker Pattern
We use the JINI distributed technology [9] to implement the Replicated-Worker pattern. Figure 3 shows the JCIS architecture with JINI. The Integrator plays the Master role; the JavaSpaces plays the Space role; and the Integration Station plays the Worker role. First, the Integrator divides an integration task into a number of sub-tasks by builder types. For example, an integration build involves compiling and unit testing is divided into two sub-tasks of the corresponding types. Metadata (e.g., URL of resources such as source code, object code and executable of a builder) for a sub-task are packed into a JobEntry object, which is then written into the JavaSpaces through the JobPool object. The integration makes the resource available on a web server. An Integration Station retrieves a JobEntry from JavaSpaces, unpacks it, retrieves the resources and performs the build. Upon completion, the Integration Station writes the build result back into JavaSpaces as a JobResult object. The actual build results are compressed and uploaded to the web server. Figure 4 shows the collaborations that take place during a build.

In JCIS, a builder performs a single integration task, such as GCCcompile builder and StatSVN builder. All builders implement the IBuilder interface. The design allows the Integration Station to launch the builders without knowing its specific implementation. Therefore, even if a new builder is added into JCIS, an Integration Stations can simply retrieve the corresponding builder using the information provided in the JobEntry.

Sequencing of the job entries is controlled by the Integrator. Sequencing is required when a build makes reference to other build results. To this end, the job entries are grouped by dependency relationships into parallel and sequential groups. Job entries inside the parallel group are put into JavaSpaces simultaneously for parallel execution; Job entries inside the sequential group are handled one by one. In JCIS, the dependencies are specified with a XML configuration file.

The proposed design of JCIS achieves maintainability. A JINI-facilitated Integration Station can run on Windows or Linux. Thus, platform dependent builds can be defined as an integration task that contains the same builds that run on different platforms. The proposed design of JCIS also achieves performance: turnaround time can be cut by adding more Integration Stations. This is further explored with three experiments.

4. Experiments on performance

We design three experiments to illustrate the performance improvement achieved by making JCIS distributed.
As shown in Figure 4, JCIS incurs many overheads in going distributed, including the time of putting the jobs into JavaSpaces, the time of compressing and uncompressing project resources, and the time of downloading and uploading compressed file. The present experiments will ignore the time of putting the job entry or job result to JavaSpaces since it is relatively small and hard to measure.

The development case under study is a WiMAX system developed by the Software Development Research Center of the National Taipei University of Technology [14]. The WiMAX project contains six sub-projects code-named SYS, VIDEO, MAC, PHY1, PHY2 and CHN, which covers the complete WiMAX protocol stack from application to channel. The development language is C/C++. The target platform is Linux. Project size information is listed in Table 1. Since the sub-project teams operate mostly independently, a check-in to the version control system by any sub-project team can break the entire system. JCIS is employed to perform daily builds for discovering such problems.

Table 1: WiMAX sub-projects information

<table>
<thead>
<tr>
<th>Project Name</th>
<th>The Number of Files</th>
<th>Size(KB)</th>
<th>LOC(lines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS</td>
<td>188</td>
<td>545</td>
<td>6568</td>
</tr>
<tr>
<td>VIDEO</td>
<td>839</td>
<td>8386</td>
<td>15839</td>
</tr>
<tr>
<td>MAC</td>
<td>210</td>
<td>1444</td>
<td>16472</td>
</tr>
<tr>
<td>PHY1</td>
<td>77</td>
<td>7188</td>
<td>3437</td>
</tr>
<tr>
<td>PHY2</td>
<td>89</td>
<td>319</td>
<td>4112</td>
</tr>
<tr>
<td>CHN</td>
<td>66</td>
<td>365</td>
<td>4622</td>
</tr>
</tbody>
</table>

Figure 5 shows the integration flow of the WiMAX system. For each sub-project, JCIS performs three builds: compilation with GCC [8], unit testing with CppUnit [5] and source code size with StatSVN [12]. The compilation builders and the StatSVN builders are executed in parallel. In particular, the compilation builder for SYS makes use of the object code produced by the other five sub-projects and must wait for results to become available. CppUnit builder of SYS then performs the unit testing contributed by all sub-projects.

The specification of the hosts for the Integrator (JCIS server) and Integration Station are as follows:

- **Integrator (JCIS server)**
  - OS: Windows XP
  - CPU: Intel Core2 T7200 2.00GHz
  - RAM: 2048MB
- **Integration Station**
  - OS: Fedora 8

CPU: Intel Core2 6320 1.86GHz
RAM: 2048 MB

![Figure 5: The integration flow of the WiMAX Project](image)

Table 2 shows the execution time of compression and decompression performed by JCIS for each sub-project. We found the compressing and uncompressing time effect is small, and all the compressing ratios are between 11%~44%.

Table 2: Compression and decompression time

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Compressing Time/s</th>
<th>Decompressing Time/s</th>
<th>Compressing Size/KB (Ratio)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS</td>
<td>0.1</td>
<td>0.2</td>
<td>226 (41%)</td>
</tr>
<tr>
<td>VIDEO</td>
<td>1</td>
<td>2</td>
<td>2109 (25%)</td>
</tr>
<tr>
<td>MAC</td>
<td>0.2</td>
<td>0.25</td>
<td>391 (27%)</td>
</tr>
<tr>
<td>PHY1</td>
<td>0.75</td>
<td>0.25</td>
<td>801 (11%)</td>
</tr>
<tr>
<td>PHY2</td>
<td>0.1</td>
<td>0.1</td>
<td>108 (33%)</td>
</tr>
<tr>
<td>CHN</td>
<td>0.1</td>
<td>0.1</td>
<td>162 (44%)</td>
</tr>
</tbody>
</table>

To provide a baseline for measuring performance speedup, Table 3 shows the average execution time (taken from 100 integration builds) of the WiMAX integration executed on sequential JCIS, the release without JINI technology. Note that total execution time the integrated project is more than sum of all the sub-projects, because extra time is spent by JCIS to collect and aggregate the execution results for generating the integration report.
Table 3: Average integration time of WiMAX project on sequential JCIS

<table>
<thead>
<tr>
<th>Project Name</th>
<th>GCC Compile/s</th>
<th>SVN/s</th>
<th>CppUnit/s</th>
<th>Sub-Project Time/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHN</td>
<td>3</td>
<td>21</td>
<td>X</td>
<td>24</td>
</tr>
<tr>
<td>MAC</td>
<td>45</td>
<td>23</td>
<td>X</td>
<td>68</td>
</tr>
<tr>
<td>PHY1</td>
<td>4</td>
<td>20</td>
<td>X</td>
<td>24</td>
</tr>
<tr>
<td>PHY2</td>
<td>15</td>
<td>22</td>
<td>X</td>
<td>37</td>
</tr>
<tr>
<td>VIDEO</td>
<td>86</td>
<td>23</td>
<td>X</td>
<td>109</td>
</tr>
<tr>
<td>SYS</td>
<td>46</td>
<td>24</td>
<td>60</td>
<td>130</td>
</tr>
<tr>
<td><strong>Total Time</strong></td>
<td><strong>422</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Experiment 1:**
The integration environment contains one JCIS Server and one Integration Station. The measurement of the experiment 1 is listed in table 4. We found the total time is 100 seconds (or 23%) more than that of sequential JCIS. The extra time comes from the overheads of compression, decompression, and transmission time for sending the metadata data and resource back and forth.

Table 4: Average integration time of WiMAX project on distributed JCIS with one Integration Station

<table>
<thead>
<tr>
<th>Project Name</th>
<th>GCC Compile/s</th>
<th>SVN/s</th>
<th>CppUnit/s</th>
<th>Sub-Project Time/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHN</td>
<td>6</td>
<td>24</td>
<td>X</td>
<td>30</td>
</tr>
<tr>
<td>MAC</td>
<td>49</td>
<td>27</td>
<td>X</td>
<td>76</td>
</tr>
<tr>
<td>PHY1</td>
<td>7</td>
<td>23</td>
<td>X</td>
<td>30</td>
</tr>
<tr>
<td>PHY2</td>
<td>21</td>
<td>25</td>
<td>X</td>
<td>46</td>
</tr>
<tr>
<td>VIDEO</td>
<td>106</td>
<td>25</td>
<td>X</td>
<td>131</td>
</tr>
<tr>
<td>SYS</td>
<td>57</td>
<td>27</td>
<td>68</td>
<td>152</td>
</tr>
<tr>
<td><strong>Total Time</strong></td>
<td><strong>522</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Experiment 2:**
The integration environment contains one JCIS Server and two Integration Stations. Figure 6 shows the best job allocation of the WiMAX integrated project, and total time is about 285 seconds based on the measurements in table 3. However, since these two Integration Stations are free in taking jobs, it can happen that one is made to wait for the other one in case there is job dependency. Figure 7 shows an example where the total time exceed 370 seconds.

Figure 6: A best case of job allocation with two Integration Stations

Figure 7: A worse case of job allocation with two Integration Stations

Figure 8 show the total execution time of the over 100 measurements. The average is about 280 seconds, which is about 66% of the sequential JCIS time, a significant performance improvement.

Figure 8: Total integration time over 100 measurements in Experiment 2
Experiment 3:
The integration environment contains one JCIS Server and three Integration Stations. Figure 9 (respectively Figure 10) shows a best (respectively, worst) case of job allocation, which takes about 270 seconds (respectively, 330 seconds) to complete.

![Figure 9: A best case of job allocation with three Integration Stations](image)

![Figure 10: A worse case of job allocation with three Integration Stations](image)

Figure 11 show the total time over 100 measurements. The average is about 281 seconds. The value is almost the same with experiment 2: the sequential group of GCC Compilation and CppUnit in SYS constitutes a bottleneck.

![Figure 11: Total integration time over 100 measurements in Experiment 3](image)

5. Conclusion
In this paper, we extend the JCIS architecture to incorporate distributed computing capabilities. Platform dependent builds are more easily managed and performance gains are possible due to parallel execution of the integration builds. A case study with three experiments to demonstrate speedup with distributed JCIS has been presented. We found that the total integration time is reduced when two or more two Integration Stations are used. However, the speedup is constrained by the dependencies between integration builds.

6. Acknowledgements
This research is supported by the National Science Council of Taiwan under the grants NSC95-2221-E-027-046-MY3 and NSC96-2218-E-027-017.

7. References