Data Generation Distribution & Management

A Major Qualifying Project submitted to the faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree of Bachelor of Science by:

Claudio Herreros
Jotham Kildea

March 4th, 2011

Report Submitted to:

Professors: Arthur Gerstenfeld
Daniel J. Dougherty
Worcester Polytechnic Institute

Abstract

BNP Paribas requires a high volume of calculations in order to support its front office. In order to perform those calculations in a more efficient way, BNP Paribas requested the implementation of a distributed system. The project outcome was a distributed system using the Oracle Coherence framework, utilizing .NET as the main development framework. The structure provided a flexible system of task distribution to be implemented at BNP Paribas.

Acknowledgements

Over the course of this project, a number of people were instrumental in contributing to our success. Without their generous and patient support along the way, the final result would have undoubtedly been far weaker of a product. These people that have helped us have come from varying directions, not only of course BNP Paribas and WPI but also outside developers from Oracle and jni4net.

Many of those at BNP Paribas that we have interacted with during our project have also helped us in some way. Without exception, they were all generous with their time and happy to help us to achieve our goals. First and foremost, we would like to acknowledge and thank Andy Clark, not only for sponsoring the project and getting the groundwork in place for us, but also for his continued contributions to the project along the way. Also, thank you Jordan Prevé, for being our day-to-day liaison and contact point for whatever information we need, as well as for putting us in contact with whomever may have the answers we need. Thanks as well to Vipin Bharathan and Mu Liu, the in-house gurus of all things Coherence, who patiently helped us get our Coherence framework off the ground and continued with support and ideas whenever we needed them. Also, to the many others at BNP Paribas that never backed away from providing feedback and help throughout the process.

Acknowledgements are certainly in order for professors Dan Dougherty and Art Gerstenfeld. Without their striving to make each year's Wall St. Project Center a success, having such an opportunity would certainly have been impossible. Their feedback on both the final report and presentation was considered in developing each final product, was of vital importance. Additionally, praise is in order for their guidance over the course of the project.

During the project, we were fortunate enough to have input from the lead developers for two of the products we used, Oracle Coherence's Processing Pattern and jni4net. Christer Fahlgren is to be praised for patiently working with and attentively keeping in touch with us as we began to implement the Processing Pattern for our own use. Praise is also in order for Pavel Šavara, whose open-source product jni4net was vital to our being able to achieve one of the biggest requirements of the project, which was to have it operable in .NET. His product, as well as his consistent feedback to issues we encountered, greatly alleviated problems along the way.

Thank you again to those mentioned, as well as to the others who helped us make this project a success.

Executive Summary

BNP Paribas in New York is a firm that requires a high volume of calculations in order to support its front office. These calculations can range from profit and loss analysis to risk analysis to pricing of exotic financial instruments. Currently, these computations are executed locally or handled using task-specific servers, and are often controlled manually. In addition to this, the reliance on specifically created servers and applications makes extensibility of the system to incorporate new products difficult. Furthermore, this current system of individual servers is not fast enough to keep up with the future needs at BNP Paribas.

The solution to the problem facing BNP Paribas was to consolidate all tasks into a single system. The approach to this solution was through the use of a distributed cache system, one that was capable of processing all calculations given to it in a timely manner. Implementing this solution allowed BNP Paribas to move away from its previous design of having numerous task-specific servers, and instead have all servers capable of performing any calculation when asked. This implementation meant a great improvement to the speed, extensibility and consistency of their computation system.

Project Goal

The goal of the project was to have a functioning prototype of an Oracle Coherence cache system that could serve as a proof of concept for the use of such a system at BNP Paribas to manage a varying number of tasks from a varying number of clients in a distributed fashion. Two main objectives were established to achieve this goal:

1. To create an Oracle Coherence application capable of computing any given task. We created and configured this system, which was able to compute tasks in Java and .NET in a distributed fashion. The system created needed to have the following characteristics:

    - **Reliability**: The system needs to be stable, and errors should not have any negative consequences on the system itself
    - **Transparency**: Clients don't have to worry about the system or the computation process. New features or applications added to the system do not have any consequence on the system.
    - **Scalability**: The system needs to be able to handle large amount of users, server and task at the same.
    - **Monitor**: The system needs to be able to keep track of the progress of the calculations and the performance of the calculation.

2. To implement two applications used daily at BNP Paribas into the Coherence cluster implemented. This goal was accomplished in order to demonstrate that Coherence cluster created could handle any giving work in distributed fashion and be more efficient that the system currently used. Two graphical user interfaces for two different applications were implemented in order to accomplish this goal:

    - The first application implemented was PolyPaths, which is a fixed income analytics application (PolyPaths, 2010).
    - The second application implemented was Westminster, which is a market scenario engine.

Results

The outcome of the project was a comprehensive distributed system with the ability to extend to varied performance requirements. This implementation not only met the objectives of the original goal of the project, but also succeeded in improving the performance as compared to the existing system. Together, these accomplishments contributed to a well-rounded framework that was inherited by BNP Paribas to improve the current implementation.

Table of Contents

Abstract i
Acknowledgements ii
Executive Summary iv
Tables & Figures ix
1 Introduction 1
2 Background and Literature Review 3
2.1 Distributed Cache 3
2.2 Oracle Coherence 4
2.3 The Coherence Incubator 5
2.4 The Processing Pattern 6
2.5 JNI Bridge 7
3 Requirements Specification 9
3.1 The Processing Pattern Implementation 9
3.2 Using Algorithms to Distribute PolyPaths Effectively 12
3.2.1 Common Overhead 12
3.2.2 Limited Complexity 14
3.2.3 Limited Number of Tasks 15
3.2.4 Limited Number of Tasks, Average Complexity 16
3.2.5 Group by Security Type 16
3.2.6 Individual Security Tasks 17
4 Results 18
4.1 Coherence Cluster 18
4.1.1 Core Structure 19

4.1.2 Client Structure 20
4.1.3 Server Structure 20
4.1.4 Monitoring and Feedback 21
4.1.5 Extensibility 21
4.2 Command Line PolyPaths as a Task 22
4.2.1 WinForms Application Associated with PolyPaths 25
4.3 Westminster Coherence Client Application 27
5 Analysis of PolyPaths Algorithms 30
6 Further Steps 33
Bibliography 35
7 Technical Documentation 37
8 Work Schedule 53

Tables & Figures

Figure 1 – Performances for Differing Security Types……..……..14

Figure 2 – Structure of Flow Data……...……..……24

Figure 3 – Structure of the PolyPaths Application……25

Figure 4 – Example of PolyPaths Application……..……...……..26

Figure 5 – Westminster Coherence Client Design…….28

Figure 6 – Runtimes of Algorithms & Demand Batch Execution…….31

Figure 7 – Comparison of Best Algorithm to Current Implementation…….32

Table 1 – Work Schedule…….…..56

1 Introduction

Rapid, up to date analysis of market data is absolutely vital to the success and profitability of any investment firm's trading operations. BNP Paribas in New York is a firm that requires a high volume of calculations in order to support its front office, which handles its high number of portfolios and transactions. All of these calculations have to be handled as efficiently as possible, whether they are small problems with only a few calculations, or large batches of algorithms that may take hours to complete. These tasks can range from profit and loss analysis, to risk analysis, to pricing of exotic financial instruments; all of which are crucial to supporting the profitability of the traders, as well as the monitoring of management. Prior to this project, these computations were executed locally or handled using task-specific servers, and were often controlled manually and did not provide the most up-to-date estimates. In addition to this, the reliance on specifically created servers and applications made extensibility of the system to incorporate new products difficult. Furthermore, the existing system of individual servers was too inconsistent and not fast enough to keep up with the future needs at BNP Paribas.

The solution to the problem facing BNP Paribas was to consolidate all tasks into a single system. One approach to this solution was the use of a distributed cache system, one that was capable of processing all calculations given to it in a timely manner. Implementing this solution allowed BNP Paribas to move away from its existing design of having numerous task-specific servers, and instead have all servers capable of performing any calculation when asked. This implementation meant a great improvement to the speed, extensibility and consistency of BNP Paribas' computation system.

For the project, this solution was implemented by the use of a distributed cache framework based upon the Oracle Coherence product. The reason for the selection of Coherence was that it provides a very stable framework to build upon. This framework is highly scalable, has no single point of failure, and is optimized for fast distribution of data and tasks throughout its cluster of services. All of these features made Oracle Coherence an ideal solution to the problem facing BNP Paribas. It was used to effectively overcome their issues regarding consistency and speed of vital calculations that support the front office and managerial office operations.

2 Background and Literature Review

In this chapter we begin by explaining in more depth the principles of the technologies we implemented. Second, we describe the Oracle Coherence framework, along with its features and its advantages over other technologies. In addition, we touch base on the Coherence Incubator projects and the Processing Pattern project, which is an application for Coherence that provides the functionality of distributing work among the nodes in the system.

2.1 Distributed Cache

In order to get a more accurate definition and better understanding of distributed cache systems, it is important to take an overview and define the terms '*distributed systems*' and '*cache*'. A distributed system consists of a computer network containing multiple nodes, where each node interacts with other nodes (Khan, 2009). A great example of distributed system is parallel computation, where a large calculation is broken into smaller calculations and the smaller calculations are then distributed between the nodes to compute the result. The principle of cache is used to increase the performance of a data storage center by allocating a cache memory which contains the data that is most likely to be accessed in the system; this process reduces the I/O overhead in the system. Combining both principles, we get a distributed cache system, which is a form of distributed system, which allows multiple machines to share a cache memory in order to increase the performance of the system. The main purpose of a distributed cache is to provide a scalable solution in order to maximize the performance of any application that constantly requires data.

2.2 Oracle Coherence

This section focuses on the last release of Oracle Coherence 3.6 as well as its features and usage. The Oracle Coherence framework is a distributed cache framework that is based upon the Coherence Data Grid, developed by Tangosol Inc. in 2006 (Oracle Coherence, 2010). One year later, Tangosol Inc. was formally acquired by Oracle (Ledbetter, 2007), and Oracle launched the project under the name of Oracle Coherence. Oracle Coherence has become a popular solution for businesses over the years due to its reliability, consistency and scalability.

Being a distributed cache system, Oracle Coherence provides the capability for an application running on a machine to use the memory of other machines in the cluster as if it were local memory. Oracle Coherence uses a peer-to-peer clustering data protocol. The usage of such protocol while sharing data greatly increases the performance compared to protocols based on central servers. Also, by not relying on a central server, the peer-to-peer protocol benefits in case one of the nodes malfunctions. The Oracle Coherence framework was developed in Java, however clients and servers of Coherence are supported in Java, .NET and C++.

The Oracle Coherence framework provides a large amount of features which make the framework reliable, consistent, scalable and very powerful. The peer-to-peer protocol and the storage implementation used by Oracle Coherences allow fast access to frequently used data in the system. Another important factor is that it supports instantaneous data management, which provides cache management in real time. In addition, Coherence provided a scalable solution that was very suitable for the project, since the project sponsors were planning to expand this technology over the following years. Furthermore, according to Oracle, Coherence provides an exclusive system for failures that Oracle describes as not having any single point of failure

(Oracle Coherence, 2010). Should a node become unresponsive or nonfunctional, the system provides the ability to redistribute the data on the cluster. In addition, new nodes and nodes that disconnect or restart are able to automatically join the cluster.

On top of Coherence's reliability and consistency, Coherence offers its own serialization library named Portable Object Format (POF). The POF library is used to encode objects into binary form in order to move them around the cluster. One of the advantages of using POF is that it is supported in the Java, .NET and C++ frameworks. According to Oracle, the POF serialization or deserialization can be up to seven times faster, and the binary result down to one sixth the size compared the standard library offered by Java (Arliss, 2009).

Analysis of the multiple advantages and features that Oracle Coherence provided a better idea of why BNP Paribas wanted to implement the Coherence framework into their systems. Oracle Coherence provides a unique technology that has become more popular over the past years due to the solutions it offers.

2.3 The Coherence Incubator

The Oracle Coherence Incubator offers a repository of different projects. These projects provide multiple solutions for some common design patterns and functionalities using Oracle Coherence (Misek, Coherence Incubator, 2010). In simpler terms, the Incubator is a set of applications for Coherence. All of the projects in the Incubator are distributed as source code and JAR files, which provide great flexibility for developers. The Processing Pattern is a project in the Incubator, which offers an extensible framework for performing distributed computing using Oracle Coherence.

However, the projects in the Incubator are only supported by nodes inside the cluster and Extend nodes written in Java. According to one of the main developers of the Incubator's project, in the near further Oracle will provide the ability to support Extend nodes written in .NET and C++ (Fahlgren, 2010).

2.4 The Processing Pattern

As mentioned previously, the Processing Pattern is an application for Coherence, developed by Oracle; its main purpose is to compute tasks among the nodes in the system. The Processing Pattern uses three different Coherence caches to communicate tasks and results between nodes. The first cache is used by clients and allows them to submit tasks into the Coherence cluster; this cache is named the '*SubmissionsCache*'. The dispatcher, which is inside the Coherence cluster, reads the '*SubmissionsCache*', and posts the tasks into another cache named the '*DispatchersCache*'. This cache is then read by one of the nodes in the list of registered nodes that can execute the tasks. The tasks are executed in those nodes, where each task is executed in a different thread. The thread pool of each processing node is defined in the configuration of the nodes, and it allows configuring the number of threads running on each node. Once a task is complete the result is returned to the client via a '*SubmissionResultCache*' and retrieved by the client using a unique task ID.

The Processing Pattern handles the task distribution between the processing nodes. In order to distribute the tasks the Processing Pattern provides three different policies: "*Round Robin*", "*Random*" and "*Attribute Matching*" (Misek, Processing Pattern, 2010). The three different policies offered by the Processing Pattern provide a very flexible task distribution system. Should the case be that all processing nodes are busy and there is a new task to compute,

the Processing Pattern puts the new task on “wait”, which makes the task wait until one processing node is available.

As part of its flexibility, the Processing Pattern provides different features for the tasks that have been submitted. One of the most noteworthy features is the ability to cancel any task at any given moment. In order to complete this, the Processing Pattern removes the task and the task’s listeners from the corresponding cache(s), and then the processing node stops the process running that task. In addition, the Processing Pattern has the capability of pausing and resuming tasks.

Furthermore, the Processing Pattern handles errors without any consequence on the system itself. In case a task fails while executing for any given reason, the outcome of the task is returned as an exception; this allows the user to find the reason for the failure. Another possible scenario is that a processing node disconnects from the Coherence cluster while computing a task. In this case, the Coherence cluster gets notified that the processing node has disconnected, and the Coherence cluster takes care of redistributing the tasks among the other processing nodes.

All of the previous capabilities mentioned are crucial for the reliability and stability of the system. However, as mentioned in the previous section, the projects in the Incubator are not supported by .NET or C++. The solution to this problem was to set up a JNI bridge between Java and .NET.

2.5 JNI Bridge

Jni4net is an application that provides the ability to create a bridge between Java and .NET (Savara, 2009). This bridge provides the capability of wrapping Java or .NET code and

calling it from either Java or .NET. The application takes a library, either a Dynamic-Link Library (DLL) or a Java Archive (JAR), as an input and then the application automatically generates an interface for each of the classes specified in the library. Once this step has been completed, the application builds the generated classes and it outputs a library (DLL or JAR), which can be used as a normal library in either programming language. In order to access the generated libraries, the developer needs to establish the connection between the proxy and the program. This process is very "light" since both virtual machines use the same process. One of the main advantages is that the jni4net allows having a total object oriented design between both programming languages.

Jni4net still is in Alpha phase, however, and is an open source project which has some limitations handling both programming languages. One of the most notable limitations is that the application cannot handle multi-dimensional arrays in any programming language.

3 Requirements Specification

The goal of the project was to have a functioning prototype of a Coherence cache system that could serve as a proof of concept system. Furthermore, the use of such a system at BNP Paribas needed to manage a varying number of tasks from a varying number of clients in a distributed fashion. At a minimum, the hope was to have a final product capable of coordinating the distribution of tasks from clients to servers through the use of Coherence as a middleware product. The connection from client to server via the Coherence Cluster allowed the passing of work, with the client having the ability to monitor its progress and be notified of completion. Furthermore, it was vital to the project that the system was capable of executing both in a Java as well as .NET environment. This requirement was necessary in order for the system to be able to integrate into the already existing frameworks and operations at BNP Paribas. It became necessary to design and create a simple, yet powerful, application that could be used as a tool to demonstrate the capabilities of the system. This application's purpose was twofold: it not only facilitated demonstration of the efficiency and ease-of-use of the final product, but also provided sample code that would be a base to expand upon by the employees at BNP Paribas.

3.1 The Processing Pattern Implementation

The Processing Pattern needed to be tailored specifically to fit the requirements. The elements that were developed were:

- Establishment of the core cluster nodes
- Use of Extend proxies
- Configuration and instantiation of servers

- Creation of tasks to operate on the system

Each of these pieces of the overall processing pattern were developed individually, but were ultimately combined to create a system that achieved all of the goals for the project.

It was vitally important to the project to be able to establish a level of communication where tasks could be delivered to the server for processing and returned upon completion. Through the use of features within the Processing Pattern, this was accomplished by moving the processing workload away from the client, which was only notified when its requested calculations had been completed. In doing so, the overall overhead related to Coherence handling the communication between client and server was kept to a minimum, as little to no direct interaction was necessary aside from sending a task and relaying results. In the scope of this project, the Coherence Cluster was implemented in such a way that moved the task processing nodes outside of the cluster.

Moving the communication capabilities of both client and server outside the Coherence cluster necessitated the use of Extend proxies. The reasoning behind this is that although the Coherence cluster is capable within an isolated network of containing all actions performed by client and server, this functionality is not available to systems that need access over Wide-Area Networks or personal computers (Howes, 2009). However, the configuration and usage of such proxies provided the cluster with listeners to specific ports, allowed for both clients and servers to communicate transparently to the cluster, regardless of physical location. Ultimately, these Extend proxies were implemented by having designated ports and proxies, one each for both client and server communication, which are automatically connected to the cluster. This decision

also resulted in another framework feature: the use of Single Task Processors set up on dedicated server nodes.

By default, the Processing Pattern comes equipped with the functionality to handle task execution in two places, within the grid through '*Grid Task Processors*', and outside of the grid though the use of '*Single Task Processors*'. While '*Grid Task Processing*' allowed for server-side computation to take place on any grid node that is storage-enabled, it was not capable of functioning through the use of Coherence Extend. On the other hand, '*Single Task Processing*' was implemented as it could handle processing outside of the cluster through connection by Extend proxies. However, it needed to be individually instantiated on each server that is to take part in processing. In order to do so, customized XML files that contain the instructions on how to use the Coherence libraries were used. Furthermore, these configuration specifications also declared the type of tasks that could be processed on a server.

In order for any server to be able to handle executing a task, the server needed to be able to understand how it was structured. In particular, instructions about how the task data and results are serialized and deserialized were stored within the task's Java class file, which was referenced in the configuration for servers as well as members of the cluster. This allowed for application-specific tasks to be written, ultimately being integrated into the distributed cache as a whole. After a task had been written, however, the necessity of managing the execution of each task that was invoked was handled within the cluster, and did not require client action outside of providing data upon which to calculate. This framework was crucial to making the system as adaptable as possible as well as reaching as a wide scope of potential applications used at BNP Paribas.

3.2 Using Algorithms to Distribute PolyPaths Effectively

In order to speed up the rate at which a task could be processed on the Coherence cluster, it was necessary to break it into subtasks. This is because submission of a single task to compute a given number of securities is not as quick as numerous submissions, each with a piece of the overall task, which are recombined later. This was a difficult challenge, as in order to achieve a good performance considerations had to be made for the overall complexity of the task, the composition of individual securities to be calculated, and the number of servers that were available to work. Furthermore, there was a substantial startup cost to operate PolyPaths under some circumstances for certain security types. This startup cost involved loading a large amount of static data and was shared for all securities of a certain type. Therefore, handling each security individually was far from efficient, when securities of similar cost could be grouped for a fraction of the overall cost. Fortunately, PolyPaths was well suited to subdivision, as multiple instances could be instantiated asynchronously, each with their own unit of work to calculate. A full analysis of the metrics of algorithm performances can be found in Section 5. Nonetheless, the following sections are a breakdown of the structure of the algorithms from a functional standpoint.

3.2.1 Common Overhead

In the project's implementation of an application to run PolyPaths, several algorithms and methods of work distribution have been developed and tested. While each of these methodologies has unique components, they also share a common pattern of how to divide and merge the subtasks in a submitted task.

Initially, each uses an object that is written to read through the input file, whether in XML or CSV format, and parse the information contained therein about securities into memory. Next, a database of securities at BNP Paribas is queried to find out what type of security each individual security is. This information is logged for every security, and is indicative of the expected complexity to calculate it. From this type, an estimate of the complexity both of PolyPaths overhead by security as well as an estimated calculation time for the security are stored.

Once the data for individual securities has been calculated, any given algorithm can be used to group securities into subtasks. The next step is to write temporary input files, each with a respective task's assigned securities. Once these have been created, the subtasks are ready to be sent to the Coherence cluster.

Each subtask is sent as an individual task to Coherence, which is passed on to the servers for computation. Once the subtasks receive word that all have been completed, they are then merged into one output file. Internally, all of this is repeated consistently, varying only in what type of algorithm is used, which decides how securities are grouped into subtasks.

Over the course of the project, five algorithms were produced, each of which can be modified, and each of which perform to different degrees. The algorithm that performs the well over all instances is the 'Limited Complexity' algorithm, which groups tasks into subtasks each with a maximum allowable time to completion. Other algorithms take different approaches, such as 'Limited Number of Tasks' and 'Limited Number of Tasks, Average Complexity' which control how many subtasks a task is divided into. Furthermore, an algorithm was developed to group securities into tasks by the type of security they are, an effort to maximize the sharing of

startup costs. Lastly, a simple algorithm was developed to split each security into its own subtask. Each of these methods of task distribution was tested over numerous different tasks and environments, with each performing well in certain circumstances. The algorithm that provided the most consistently efficient performance, however, was 'Limited Complexity' with a predefined means of finding the maximum complexity per task.

3.2.2 Limited Complexity

One of the best performing algorithms used to distribute work on the servers is to create subtasks that have a limit on their individual complexity. This is accomplished first by sorting the securities in decreasing order by complexity. Next, the most complex security is added to the current subtask, provided that doing so would not go over the defined limit. Preference is given to securities of the same type as are already present within a subtask, in order to share in startup overhead. This common overhead cost can be seen in Figure 1, where three types of securities, when calculated, have a specified baseline cost to run the calculation. Failing at using a similar type security, any other type can be viable, provided that it falls within the complexity limit. If no security can be found to fit within these constraints, a new subtask is dynamically created and the process continues with the new subtask. In this manner, a task can be broken into a relatively

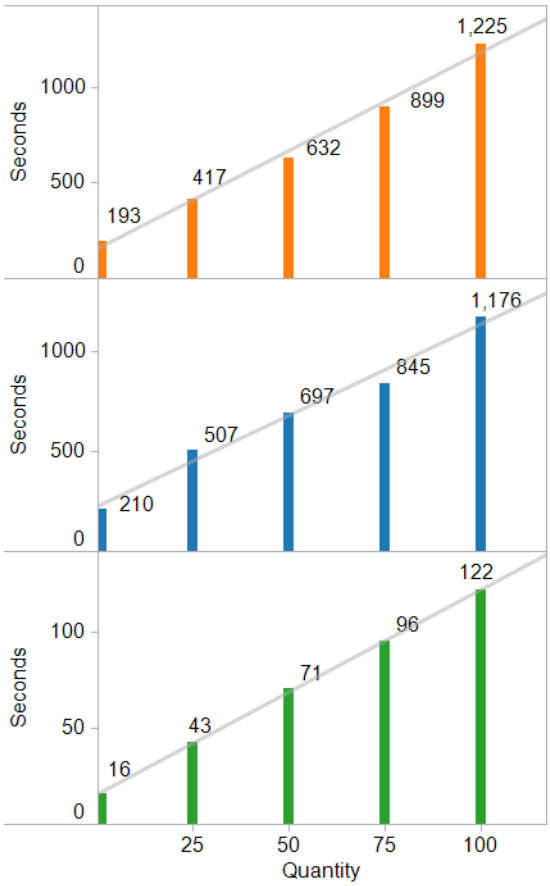


Figure 1 – Performances for Differing Security Types

small number of subtasks that incur low startup costs and take approximately the same time to complete.

Furthermore, in order to strike an effective balance between minimizing task costs and overreaching the capacities of the server farm, an improvement upon this algorithm was developed that effectively provides a sliding scale of complexity limits that is related to the number of securities to be calculated. This system starts off with a small grouping complexity for small tasks, which helps to prioritize small requests to finish as rapidly as possible. For larger tasks, a gradually increasing cap on the complexity is imposed, which helps the subtasks grow gradually to account for the growing complexity while avoiding over or under grouping. Utilizing this approach, an algorithm was derived that proved itself to perform well under all circumstances tested, both for very simple tasks as well as for quite complex tasks.

3.2.3 Limited Number of Tasks

A straightforward yet effective algorithm to distribute the work into subtasks is to simply distribute each security evenly. In this algorithm the list of securities are iterated through and distributed to the set number of subtasks in a round robin fashion, with no regard for the expected complexity of each security. In practice, this method tends to perform reasonably well, as when the securities are randomly distributed they tend to develop tasks with approximately average complexities. However, this method's pitfall is that there is no guarantee of good distribution, and with no consideration of complexity for each security, it is very possible for a large number of complex securities to be put into a single subtask. Therefore, while in most situations this performs well and operates quickly, certain circumstances could lead to great inefficiencies in task distribution.

3.2.4 Limited Number of Tasks, Average Complexity

Another algorithm was developed that performs similarly to a straightforward limited number of tasks method, but additionally considers the weights of tasks in deciding how to distribute workload. This was accomplished by evaluating the complexities of all securities, and then sorting the list of them in descending order of complexity. Once this is done, they are distributed into the predefined number of bins in a modified round-robin fashion. The modification is to distribute them iterating up the list of subtasks and then distributing down the list, rather than always in an increasing manner. Doing so prevents the early tasks from always being given the more complex securities, and provides a more homogeneous weight in each subtask. This improves upon simple sorting of securities without consideration for their complexity, as it helps to avoid unexpected conditions where subtasks are poorly balanced. However, this system does little to consider the wastage generated from redundant startup costs across tasks, as it will lead to similar securities being distributed widely across subtasks.

3.2.5 Group by Security Type

Grouping the securities into subtasks based upon the type of security they are was found to be quite effective, and maximizes sharing of overhead costs. This is achieved by iterating through the list of securities, keeping a list of types that have been encountered up to that point. This list also stores which corresponding subtask is being used to store a certain security type, and can be used for sorting securities that belong to already-encountered types. If it encounters a new type while iterating, it dynamically creates a new subtask, which is reserved for the new security type. This method performs well for small and medium size batches of securities, and keeps a low startup cost regardless of batch size. However, larger batches may result in single

tasks that contain hundreds of complex securities, and will perform quite poorly under these circumstances.

By additionally limiting the size of subtask, this algorithm was improved to address the issue of poor performance with heavy tasks. This was done by establishing a defined limit on the size of any task. The algorithm then performs very similar as to without the limit, however it keeps track of the growing size of subtasks as they are built. If at any one time a subtask would be overloaded by adding another security, this security is instead added to a new subtask and all future securities of that type enter the new subtask. This improvement to the algorithm allows for flexibility to provide high performance across a wide band of task sizes.

3.2.6 Individual Security Tasks

The algorithm that merely divides each security within a task into a subtask can be effective if the number of securities is very small, but otherwise is highly inefficient and burdensome to the cluster as a whole. The delay of writing and reading temporary files for subtasks is small in most circumstances, but the latency to build and write these files is substantial when this is done for a very large number of individual files. Furthermore, if the number of tasks exceeds the number of available nodes on the server, the excess tasks will be queued until other subtasks finish. This latency both slows the outcome dramatically as well as inhibits other tasks from different users from being processed in a timely fashion. Therefore, splitting securities into individual tasks is best reserved for small batches of only a handful of securities, where expediency is desired and is possible without overloading the servers.

4 Results

4.1 Coherence Cluster

In developing the Coherence cache product for BNP Paribas, it was especially important to ensure that the final product met the original requirements of the project. In particular, considerations were constantly made to ensure that the cache was extensible, efficient, and able to be closely and constantly monitored. Doing so required not only a focused interpretation of how each piece of the puzzle was to operate, but also a broad realization that all pieces had to fit perfectly together and work in harmony to achieve the goal. Among the many minute considerations to be made were some large interactions to consider, such as:

- How the cluster itself was to be designed
- How the clients and servers pass data through the Coherence Cluster
- How to make the cluster easily extensible to suit any necessity
- How to harness and control the tasks that are sent through the system and monitor them appropriately
- How to make the end product be flexible enough to apply to a broad range of applications

In completion, all of these features were interwoven into a system that achieves the initial objectives effectively, and provides a strong demonstration of the Oracle Coherence cluster in action.

4.1.1 Core Structure

The internal design of the Coherence cluster was the core of the project that was to be built upon. The initial concept for the design of the cluster was predicted to be very simple and straightforward, however the ultimate development proved to be quite complex. The foundation for the Coherence cluster is the '*Dispatcher*' nodes that reside within the cluster. These took the form of Coherence cache servers that are instantiated in such a way as to persist and automatically join other members of the cluster. Within the framework, their core responsibility is to negotiate the distribution of work to connected servers, as well as monitor progress and relay messages regarding completion. Even though these nodes can be configured to distribute work in several different manners, the usage chosen for the project was to provide a round-robin distribution of work to all servers, effectively balancing the workload amongst the server farm.

In addition to the dispatchers, the Coherence cluster also was designed to incorporate the use of two Extend proxies. These proxies resided in the cluster, but on the fringe. Their purpose was simply to allow points of communication both on the server side as well as the client side. Due to the necessity to operate in .NET as well as to operate in a distributed, non-local network, these proxies were required to contain the cluster locally yet still allow communication to either side of the work distribution. Similar to the dispatchers, these proxies were lightweight and instantiated to persist and connect to other nodes within the cluster.

This final design approach for the core of the Coherence cluster creates a fully functional approach that allows for a wide variety of operations to pass through the cluster. Furthermore, from a design perspective, this allowed for the entire cluster to be thought of as one monolithic entity, with just one access point for the client and another for the server. This created a generic implementation that is more readily extensible to other applications.

4.1.2 Client Structure

In this project, the structure of the client was extremely straightforward and of minimal size. Each client obviously must have some means of communicating to the cluster, which was handled by using a coherence JAR file structured to handle access to the appropriate proxy. With this connection established, the core features of the task submission were to submit the task to the cluster, and to wait for the response. Both of these were provided by Coherence, and in fact only the submission was required, as tasks were able to be submitted without concern for feedback if desired. Both submission and reception allowed for a significant amount of transferal of data, which was still feasible through the current implementation. Although both applications that were developed in the course of this project primarily used shared files for communication and had relatively small amounts of data transfer directly, the functionality for more data transfer directly through the cluster was still provided.

4.1.3 Server Structure

The structure of the servers in the overall implementation was simple and shares some characteristics of both the clients and the cluster itself. As with the client, servers connected to the cluster through a proxy connection reserved for such use, and needed no more setup in order to participate in calculations. However, the servers were instantiated in much the same way as the units of the cluster were started: via simple execution scripts that were based on Coherence configuration files and remained active indefinitely. As far as the actual task execution that takes place on the server was concerned, a simple class that inherited the ‘Resumable Task’ interface from Coherence can be used by a server, and was included in its running environment through a simple configuration file. This allowed for servers to be easily extended in order to execute any desired calculations.

4.1.4 Monitoring and Feedback

As monitoring and the overall accessibility of information about tasks as they are executing was of high importance to BNP Paribas, specific considerations were made to bring as much transparency to these aspects as possible. From the client perspective, an application was given the capability to view the status of any submitted task, as well as specific metrics as to the progress within. Also, the ability to terminate tasks before completion was also accessible to the client, which provided important features related to controlling the overall execution.

4.1.5 Extensibility

In designing the project framework into the final iteration that was delivered, several features made the system particularly extensible. Firstly, with concern to extending the size of the cluster and of the servers, creation and startup of new dispatchers, proxies, or servers was relatively simple. This was because of the way in which they dynamically organized themselves, as well as the means through which redundancy and failover considerations were controlled within Coherence. Furthermore, during the project a simple script was developed in order to automatically create a unique ID for any new server that was to join in on handling the workload. This allowed for next to nothing in startup work in order to add servers to server farm.

Aside from expanding the size of the cluster and server farm, extensibility was achieved through the constrained points of entry and exit to passing through the Coherence cluster. This meant that the core within the cluster could remain unmodified, while only adaptors to the proxies needed to be created for a new operation that was to be handled. This improvement drastically improved upon development time, allowing new calculations to be run on the system without the overhead of developing a system that would stand in the place of Coherence for each new application.

4.2 Command Line PolyPaths as a Task

A major proof of concept for the prototype Distributed Coherence Cache framework was to be able to handle the invocation of PolyPaths remotely on servers (PolyPaths, 2010). To accomplish this, considerations for transparency to the client invoking a task, accessibility of the data to be used, implementation within the Coherence framework, and efficiency in task distribution were carefully considered. However, all of this hinged upon the system being designed in such a way as to integrate easily into the current usage of the PolyPaths functionality at BNP Paribas.

Before the availability of the distributed computing architecture that is provided by this project, the primary means by which to perform large volumes of necessary calculations by PolyPaths would have been executed by a command line operation and performed locally on each machine. However, this system can create a large load to a single computer, which is problematic if the computer is a personal desktop used by traders. Also, there are substantial slowdown costs, as each security must be calculated in sequence, while a distributed system could share the workload over many processors. The project's implementation allowed for the invocation of a client that can communicate tasks to the cluster, with little or no difference in the complexity of the operation call. Additionally, part of the project's accomplishment was the creation of a straightforward Windows Form application that can be used to demonstrate and use the Coherence Cluster for PolyPaths calculations through a graphical user interface. This application offered all functionality of command-line calls, as well as the ability to start multiple calculations simultaneously and monitoring of task progress. In both instances, the task processing was handled the same way within the structure of the Coherence Cache.

Within the Coherence framework, the actual implementation of a PolyPaths calculation request was very simple. A task had been written that simply receives the command like arguments that would ordinarily be executed on the client's machine, and instead invokes them on the server. This simple passing of work allows the task to be loaded on the server processor instead of the client's, thereby lightening the load on the client. This task was visible across all sections of the cluster and servers, and therefore allowed for a very lightweight method of migrating work away from the client. However, in order to do so successfully still relied upon the ability for the server to be able to access the input data that needs to be calculated.

The PolyPaths application required access to the data in a compatible form in an input file. Therefore, in order for the calculations to be moved onto a server that is located on a computer other than where the files are stored, the system needed to be able to compensate for this. Fortunately, much of the networking framework in place at BNP Paribas relied on shared drives for file storage. As a result, any input file or output destination could be used by the project's implementation provided that its absolute network path was provided. A benefit from this was that this system of task passing was very easily distributed over numerous servers to speed up processing.

The greatest benefit that the use of a distributed cache for PolyPaths calculation provides was the opportunity to use the processing power of numerous servers to expedite the valuation of large batches of securities. In the implementation of PolyPaths that was used within this project, this was accomplished by shredding the input file into numerous smaller input files, each of

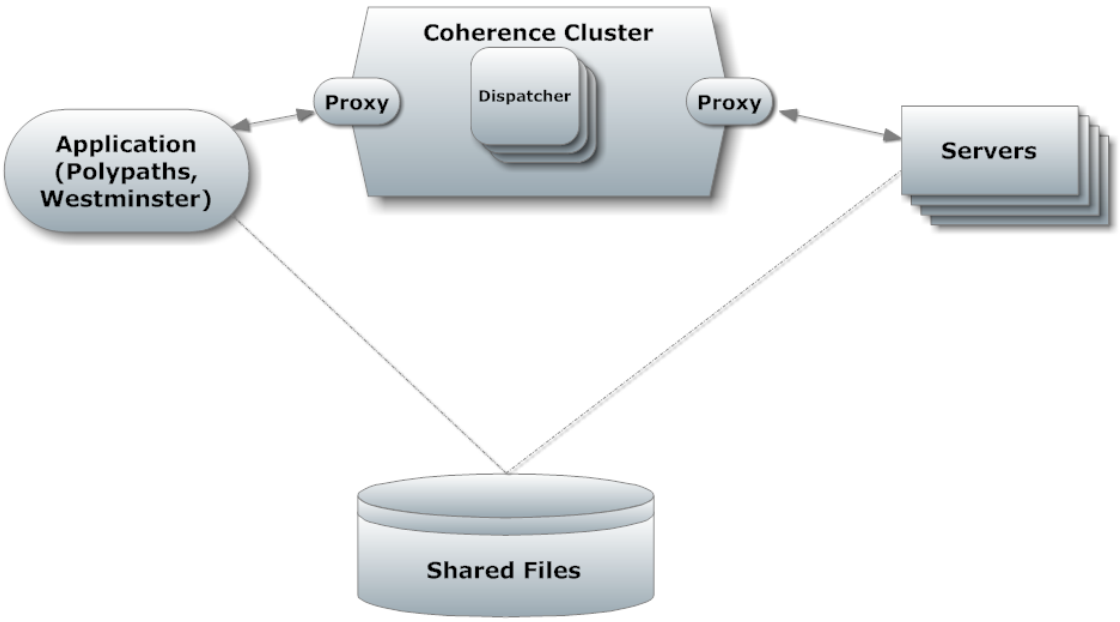


Figure 2 – Structure of Flow Data

which was handled individually on different servers. Each file had the necessary input data, could be calculated in parallel, and was recombined with other files upon completion. This architecture was located within the client specific to the PolyPaths application, and could easily be replicated in other clients, provided an understanding of how the target application's work could be piecemealed and executed.

Figure 2 is an example of the final structure, including how the layers of the process appear in Java versus .NET, as well as the interoperability between the two. As can be seen, it is important to note that the client side makes use of .NET, while the server side remains

exclusively in Java. Also, both sides share the usage of commonly accessible shared files instead of using the cluster as a means of passing of data.

4.2.1 WinForms Application Associated with PolyPaths

The culmination of client-side development for PolyPaths was a comprehensive WinForms application. This application was intended as a demonstrative tool of the power of the system developed. The complete structure of the completed project for PolyPaths can be seen in Figure 3. In doing so, a complex yet intuitive interface was developed to exemplify the different algorithms, different calculation formats, and task monitoring and management options available to the end user. The resulting product of all features can be seen in Figure 4. Starting from the

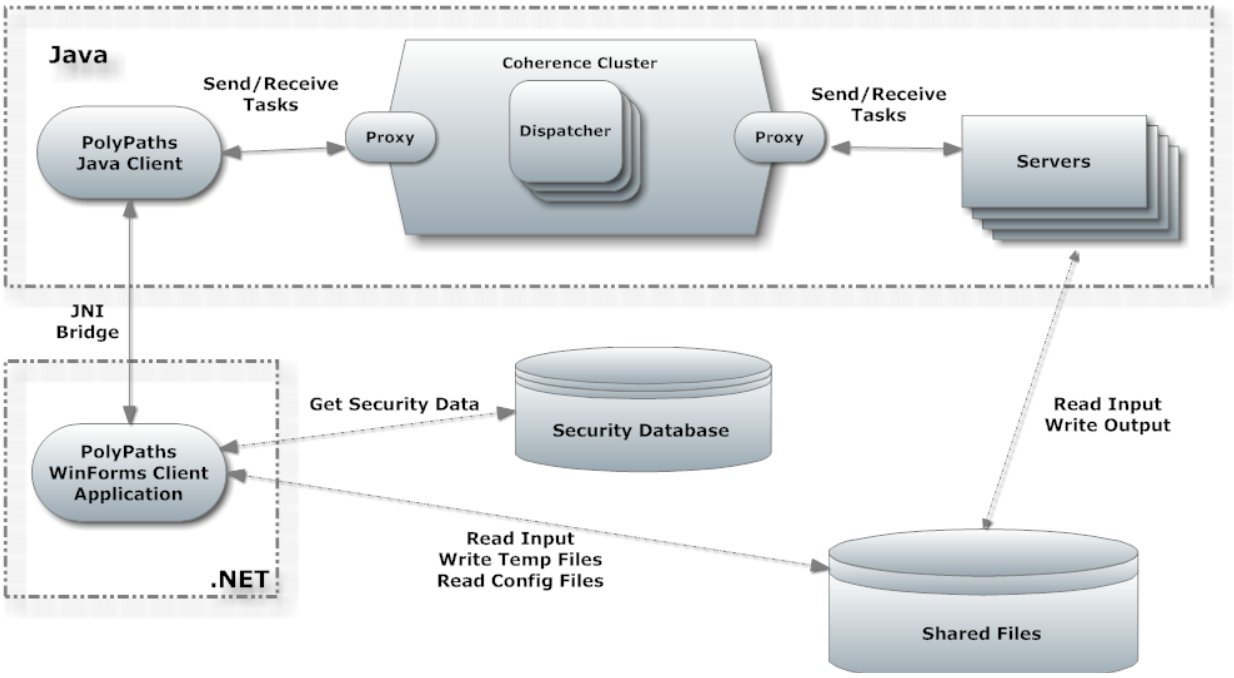


Figure 3 – Structure of the PolyPaths Application

top left of the window pane, it can be seen that different input and output paths can be specified for the calculations, and within these options are the choices for both .xml and .csv file formats. Beneath this is a listing of 'switches' corresponding to different data values that are to be

calculated for the given securities (specific switch names have been omitted from this figure for confidentiality reasons). Any number of these can be chosen to be run together, or alternatively

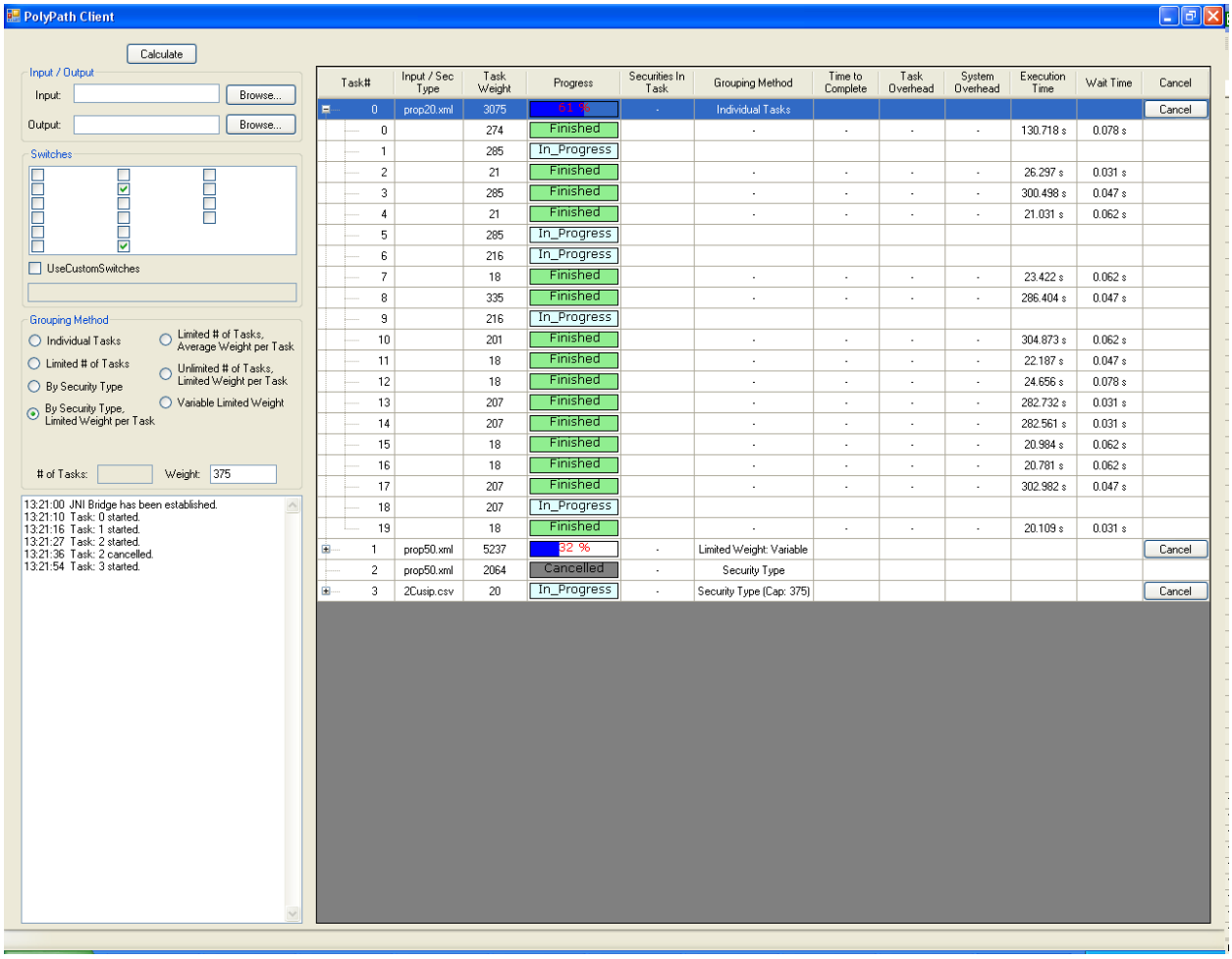


Figure 4 – Example of PolyPaths Application

the box beneath it can be used to directly copy and paste in a specific list of options, allowing for greater flexibility. Next down are the various methods of grouping as described in section 3, and also corresponding configuration data may be entered where available. Lastly on the left column is the output pane, where performance data about the application as a whole may be inspected and tracked. To the right is the pane allowing monitoring and inspection of the currently submitted tasks. Within these collapsible lists it can be seen that a particular task may be inspected with greater granularity to observe what components of it have completed, while also

being able to easily identify any trouble spots. During the execution, a great deal of data is available to the user, such as the progress of the task(s) (as visible in the progress column), the types of securities in each batch (omitted), and the presumed complexity of a task as well as its actual runtime. Furthermore, tasks may be cancelled directly through this interface by means of the ‘cancel’ button associated with a task on the right of the pane. As can be seen from the task in the figure, a cancelled task remains on the list of computations, but is immediately abandoned on the server, and only remains for informational purposes. Collectively, this application demonstrates all of the capabilities present both in the core of the Coherence Cache as well as the adaptation suited to PolyPaths calculations.

4.3 Westminster Coherence Client Application

As a result of proving how fast applications could be incorporated into the cluster, during our last week of work we implemented a graphical user interface to compute tasks using Westminster in a distributed fashion. Westminster was an application used by BNP Paribas that allowed computing market scenarios by inputting a list of parameters for a specific market. Westminster is an application fully written in .NET and developed by BNP Paribas, which has an extreme importance on a daily basis for traders.

In order to prove that the cluster was also able to compute tasks written in .NET, and due to the fact that Westminster had a significant impact on a daily basis for traders, we decided to implement a Westminster application into our Coherence cluster. The application developed was programmed in Java to prove that our system was also able to handle both .NET and Java at the same time, and on the same cluster. In order to implement a Westminster application into our Coherence cluster, we decided to create a Westminster controller, which we will refer as

'*Westminster Server Controller*'. The '*Westminster Server Controller'* was based on a wrapper written as interface in .NET, which called a Westminster application, referred as '*Westminster Server*', through the use of the '*RemoteObject*' library.

Figure 5 represents the design we decided to implement for the Westminster Coherence Client application. The major difference in the design compared to the PolyPaths application created was that Westminster clients were written in Java, and the servers executed a Java task that opened a jni4net bridge to execute the task written in .NET. The .NET task calls the '*Westminster Server Controller*', which takes care of sending a list of parameters to the 'Westminster Server'. Then, the '*Westminster Server*' takes care of running the scenario engine with the parameter provided. Once the market scenario has been created, the output file containing the scenario specification is created on a shared drive where the user can easily retrieve the file.

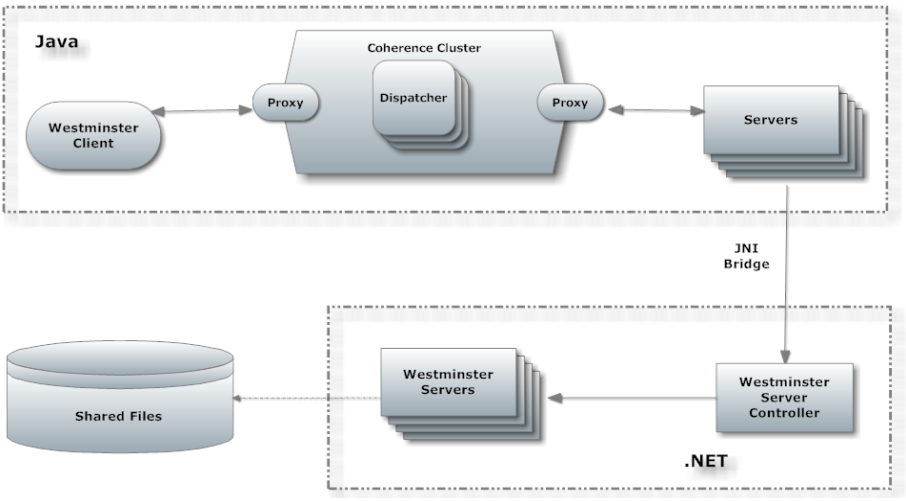


Figure 5 – Westminster Coherence Client Design

As with the PolyPaths application, the Westminster Coherence application allows the users to monitors the tasks launched. In order to monitor the tasks, the client sets up a listener on the output file, once the output file is completed by the '*Westminster Server*', the application was notified and outputs the total time taken to compute and which machine computed the given scenario.

Another key point during the creation of this application was to make a very flexible environment with different capabilities, so developers can keep implementing the application very easily and adapt new functionality to the application without major issues. As a proof of this, our application had two tabs, the first one where the user can launch the creation of a single scenario with a list of parameters. The second has the capability of taking a CSV file with different scenarios parameters and generates the different scenarios in a distributed fashion.[1]

[1] For confidentiality reasons, we cannot post screenshots of the Westminster Coherence Client application.

5 Analysis of PolyPaths Algorithms

In order to more accurately evaluate the effectiveness of different algorithms that were created to organize PolyPaths calculation requests, as well as to spot room for possible improvements, all of these algorithms were run numerous times under differing environments. These benchmarking tests provided useful interpretations of how well a particular algorithm could perform, as well as gave a point of comparison to determine overall improvement. Figure 6 gives a distribution of performance, measured in overall runtime, of all algorithms and previous means of calculation over a variety of task complexities. It is important to note that the '*Command Line*' and '*Demand Batch*' performances are representative of the two means of calculation currently in use at BNP Paribas. The rest of the performance distributions, labeled in green, are the myriad of different algorithms that were implemented. It is important to note that the '*Demand Batch*' calculations were executed on approximately 100 processors and the algorithms used were executed on only 16 processors, yet still outperformed in most circumstances. (For brevity's sake and in order to reasonably understand the data, much of the '*Command Line*' execution was excluded for more complex files. In actual testing, the largest files were found to take in excess of 2 hours to calculate).

Figure 6 demonstrates the relative consistency of runtimes throughout the algorithms attempted, but it was necessary to develop upon a single algorithm to create an algorithm that would perform reasonably well under all circumstances. This is the '*Limited Complexity*' algorithm as described in the Methodology section, with a sliding scale for the complexity to be used. In the next graph, it can be seen how this metric performed in comparison to '*Demand Batch*', the best-case means of data generation currently in use at BNP Paribas. As can plainly be seen, in the event of very large and complex tasks being run, the developed system with accompanying algorithm can outperform the current implementation in roughly half the time. This is impressive as this is still being performed on the 16 processors versus the 100 processors in use for the

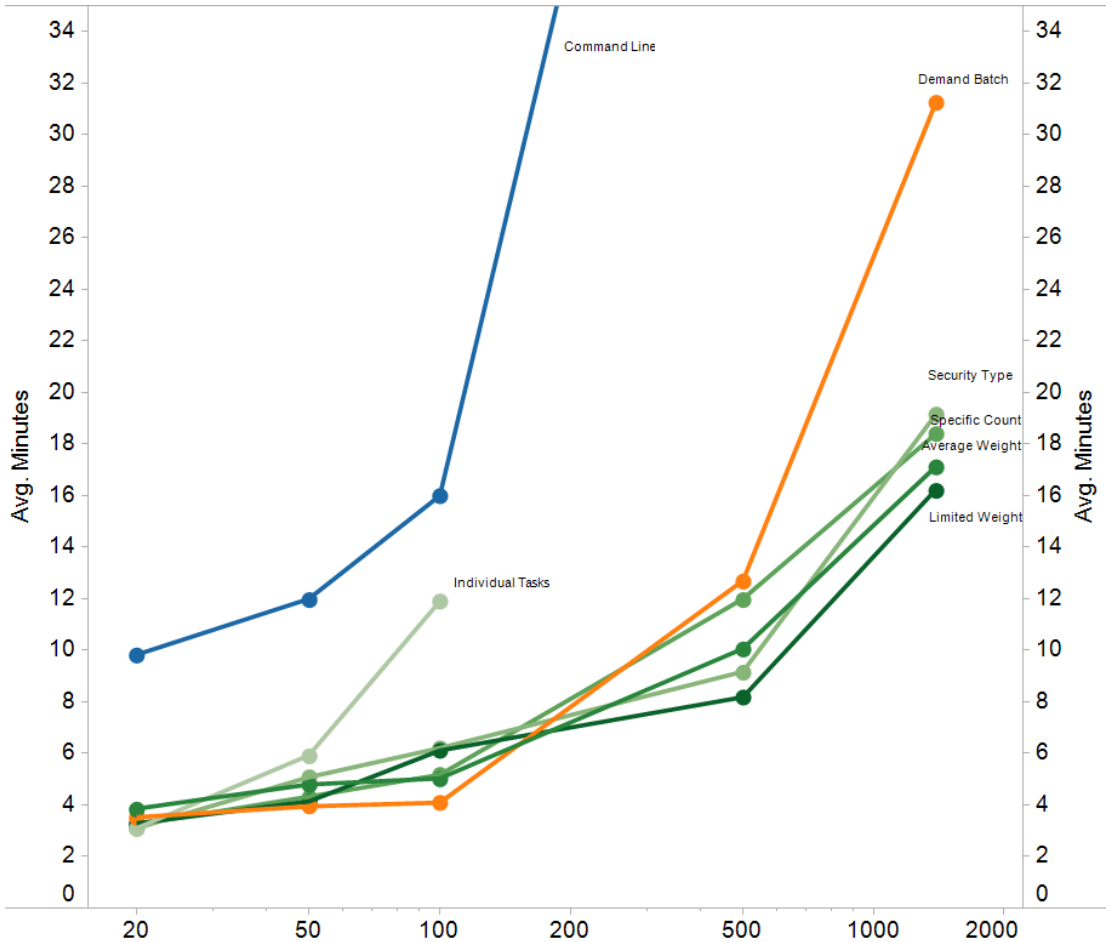


Figure 6 – Runtimes of Algorithms & Demand Batch Execution

current method. The poorer performance experienced by the algorithm for some mid-ranged file sizes can be attributed to this difference in processors in use. If the number of processors within the cluster were to be comparable, it could be expected for this gap to shrink considerably, possibly even reversing.

The major outcome of these benchmarking tests of the created algorithms is positive. When run under comparable features and on identical calculations, the algorithms written in conjunction with the Coherence cluster developed could reasonably match or outperform the current implementation, at times by a factor of 2 This comparison can be seen quite clearly in Figure 7, a comparison between the optimal algorithm chosen and the existing calculation

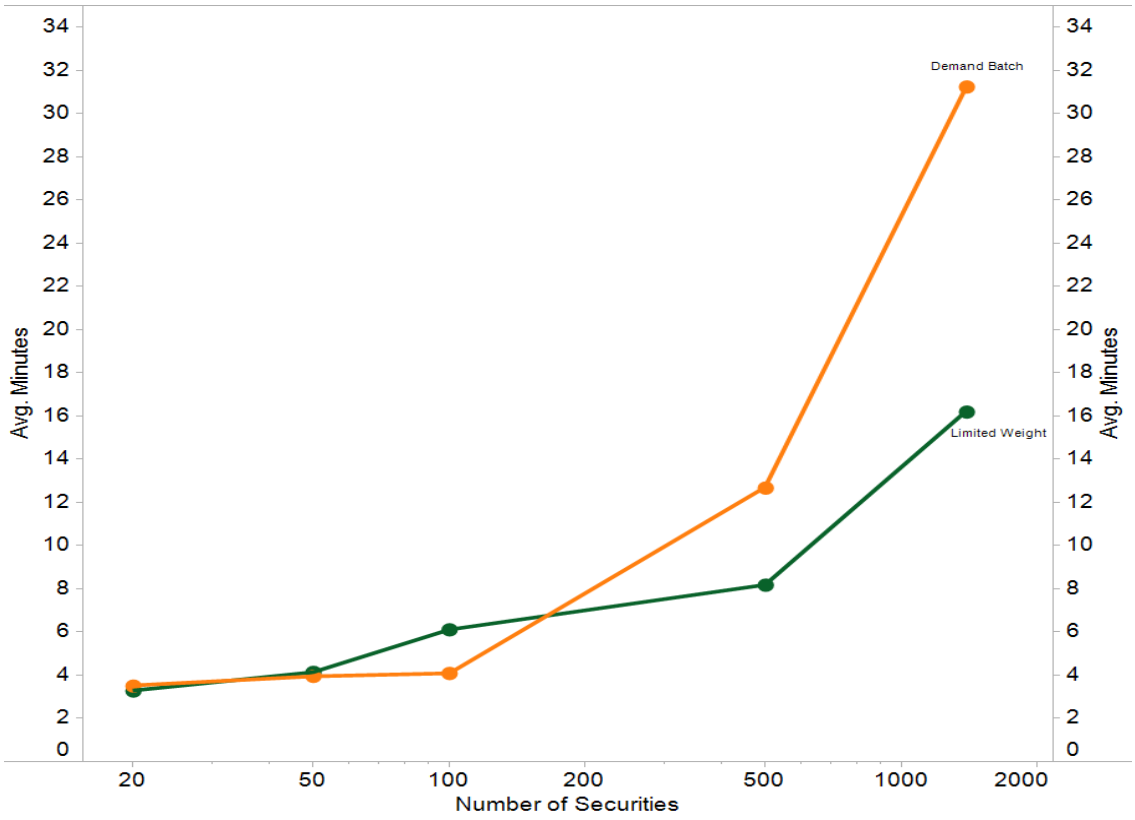


Figure 7 – Comparison of Best Algorithm to Current Implementation

methods in use at BNP Paribas. This is a strong indicator that the Coherence cluster can not only be used as an effective means of having generic distribution of work, but also as a lightweight platform for powerful distribution algorithms that provide noticeable benefits to BNP Paribas.

6 Further Steps

There are more improvements to the product that could be implemented, so it was structured in such a way so as to make their later development possible. The primary improvements that could have been made were to integrate the PolyPaths and Westminster applications into the systems currently in use, to make the cluster operate as a series of windows services, to move the algorithms to distribute work into the cluster itself, and of course to perform calculations for other applications on the system. While all of these would provide their own benefits to the product, the improvement that BNP Paribas could implement quickest in order to see performance improvements would be to integrate the system into the current applications.

Integrating access to the Coherence Cluster into current applications at BNP Paribas could provide performance improvements with relative ease. These integrations would likely not use the applications developed during the project specifically for calculation, as these were merely for demonstrative and testing purposes. Nonetheless, the core concept of the means of accessing the cache could be transferred quite easily, and transparently implemented in systems already in place at BNP Paribas. Once this is done, the next logical step would be to help improve the ease of use of the cluster itself, which would take the form of developing windows services.

Windows services, which are applications that persist in the background of a running operating system, are a perfect candidate for the long-term implementation of the Coherence Cluster developed. It is probable that windows services would be an ideal implementation of the cluster, as they provide the simplicity necessary to manage as well as the reliability desired. Once a reliable and stable server has been established, it would then be feasible to transition the burden of deciding how to distribute work into the cluster itself.

While the developed implementation where the algorithms to distribute PolyPaths are retained within the client application works well, it would be preferable to have this sort of calculation be performed in a generic manner within the cluster. By modifying the implementation of the Processing Pattern to insert an intermediate step wherein work is split, distributed, and rejoined transparently, it would be possible to achieve the performance benefits regardless of the application being distributed. This would lessen the burden to the developers, as they would only have to write a small wrapper to this implementation, rather than rewrite the algorithm for each specific application. Of course, having such ease of extensibility would of course allow for far more applications to be integrated effortlessly.

Lastly, the addition of more and more applications to the list of calculations that can be performed on the developed system is possibly the most evident extension of the software. Due to the fact that the Coherence Cluster is so adaptive and the structure is as flexible as it is, extensions and additions should not be a great challenge for BNP Paribas, and is an opportunity in the near future for even more performance benefits to be gained through the use of the system created.

Bibliography

Oracle Coherence. (2010). Retrieved October 2010, from Oracle: http://www.oracle.com/technetwork/middleware/coherence/overview/index.html

Arliss, N. (2009, June 4). *The Portable Object Format*. Retrieved October 2010, from Oracle: http://coherence.oracle.com/display/COH35UG/The+Portable+Object+Format

Fahlgren, C. (2010, November 02). Oracle Meeting. (C. Herreros, & K. Jotham, Interviewers) New York, MA, United States.

Howes, J. (2009, April 20). *Configuring and Using Coherence Extend.* Retrieved December 12, 2010, from Oracle Coherence: http://coherence.oracle.com/display/COH34UG/Configuring+and+Using+Coherence+Extend

Khan, I. (2009). Distributed caching on the path to scalability. *MSDN Magazine*.

Ledbetter, L. (2007, March 23). *Oracle Buys In-Memory Data Grid Leader Tangosol.* Retrieved October 2010, from Oracle: http://www.oracle.com/corporate/press/2007_mar/tangosol.html

Misek, R. (2010, October 4). *Coherence Incubator*. Retrieved October 2010, from Oracle Coherence: http://coherence.oracle.com/display/INCUBATOR/Home

Misek, R. (2010, 06 24). *Processing Pattern*. (N. Arliss, Editor, & Oracle) Retrieved 11 27, 2010, from The Coherence Incubator: http://coherence.oracle.com/display/INCUBATOR/Processing+Pattern

PolyPaths. (2010, 12). *PolyPaths Fixed Income Systems*. Retrieved 12 2010, from PolyPaths: www.PolyPaths.com

Savara, P. (2009). *JNI4Net*. Retrieved 11 26, 2010, from JNI4Net: http://jni4net.sourceforge.net/

Waldo, J., Wyant, G., Wollrath, A., & Kendall, S. (1994). *A note on distributed computing.* SML Technical Report Series, Sun Microsystems Laboratories, Mountain View, CA.

7 Technical Documentation

WPI 2010
DDGM

Technical Documentation

Claudio Herreros

Jotham Kildea

12/17/2010

Table of Contents

1 Coherence Cluster 2
1.1 How to Run it 2
1.2 How it Works 2
2 NodeConfigCreator 4
2.1 How it Works 4
2.2 Requirements 4
3 Jni4.Net 5
3.1 Restrictions 5
4 Polypaths 6
4.1 How to Compile it 6
4.2 PolyClientWinForm (.Net) 6
4.3 PolyClient (Java) 7
4.4 Known Unresolved Bugs 9
5 Westminster 10
5.1 How to Compile it: 10
5.2 How to Run it 10
5.3 WestTest (.Net) 10
5.4 Java (application, gui, utilities) 10
5.5 Known Unresolved Bugs 11
6 How to Develop the Current Structure Further 13
7 Contact 15

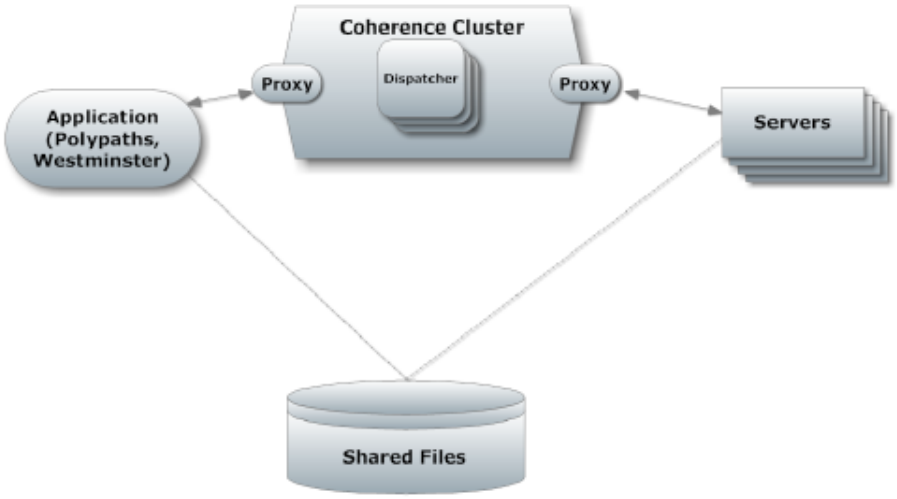
1 Coherence Cluster

1.1 How to Run it

1. Launch *"cache.cmd"*
2. Wait until *"cache.cmd"* has been initialized
3. Launch *"client-proxy.cmd"*
4. Launch *"server-proxy.cmd"*
5. Wait until *"server-proxy.cmd"* has been initialized
6. Launch *"server.cmd"*

1.2 How it Works

- All the configurations for the nodes (cache, proxy, server and client) can be found at http://coherence.oracle.com/display/INCUBATOR/Configuration+for+the+Processing+Pattern
- *custom-pof-config.xml*: POF serialization contains the path to the task locations. The POF configuration file will look for *.class.
- Inside the *PPServer* folder there are two folders: *application* and *process*. These folders contain the class files for the Tasks.
- *cache.cmd* : Launches the cache and dispatcher, the cache needs to be executed with the following classpaths: coherence.jar, common.jar and processingpattern.jar. The libraries can be founded at: (http://coherence.oracle.com/display/INCUBATOR/Home). The configuration file initializes the dispatcher.
- *server-proxy.cmd* & *client-proxy.cmd* : Create a proxy to connect the servers and clients. Need to make sure that the proxies for the server and client use different ports.
- *server.cmd :* Launches a processing node.



Final positioning and function of the coherence cluster in the overall architecture

3

2 NodeConfigCreator

This solution is executed as part of the execution of *server.cmd* when creating a new node. Once the NodeConfigCreator project has been built, the created .exe file needs to be copied into the WPI directory for *server.cmd* to access it. It should not need to be run on its own. The executable file takes as an argument the absolute path to the configuration file of the server.

2.1 How it Works

- Loads the *server.xml* file, and replaces the 'ID' with a dynamically generated unique ID. The unique ID is currently setup to be the current time.
- Necessary to quickly start up servers, as no two servers may share an ID

2.2 Requirements

- The absolute path to the configuration file of the server. *(server.xml)*
- Be invoked by launching *server.cmd*

3 Jni4.Net

JNI4.Net is an application that allows to port libraries from .Net to Java and from Java to .Net. The application auto generates the required code from one language to the other. We used JNI in order to address the problem of supporting .Net in the Processing Pattern. JNI4.Net requires having the following directories:

- *lib*: this folder contains all the libraries to run JNI4.Net, also the user needs to put the library he wants to port into this folder and modify generateProxies, to reference the library correctly
- *work*: this folder is where the work gets computed and where the ported library is outputted
- *bin:* this folder contains the application *proxygen.*

3.1 Restrictions

- The JAR file to port cannot start with an upper letter case
- Proxygen needs to be inside a folder named *bin*
- Jni4.Net does not support multidimensional arrays
- The bridge cannot be initialized on a shared driver. It needs to be initialized locally
- Questions http://groups.google.com/group/jni4net?hl=en

4 Polypaths

4.1 How to Compile it

1. Once the Java code is ready, we need to create a JAR file
2. Copy the created JAR into the *lib* folder
3. Run *generateProxies.cmd* (This will create a DLL named *polyclient.j4n.dll)*
4. Need to make sure that VS is closed (or any application that is using the C# classes)
5. Run *copyJarDll.cmd* (This will copy the DLL and JAR files into the *Debug* folder of the solution
6. Make sure that the Coherence cluster is running
7. Execute the solution

Notes:

- In case of making any changes to *CommandResumableTask.java*, you will need to get the binary file (.class) and copy it into the server.

4.2 PolyClientWinForm (.Net)

This is the primary client application and WinForm that is used to interact with the cluster and send calculations. As it is a VS project, it can be run simply by building the project and either running it from VS or through the .exe file that is created. Within the project itself, there are several components to look at specifically:

- *Form.cs, Program.cs* - Rather self-explanatory, these are what interfaces with the GUI and handle the default work to set up a WinForm
- *Task.cs* - This is the main class that handles all things related to parsing a task, sending it, receiving it, and aggregating it
- *PolyFile.cs and related classes* – These classes implement the PolyFile interface, and are used to specifically handle the shredding and merging of their respective file types.

- *FileJoin.cs and related classes* – These classes implement the FileJoin interface, and are all of the different algorithms used to distribute files. By looking at them as a foundation, more algorithms can easily be added.
- *ProgressColumn.cs and CustomCancelColumn.cs* – Used exclusively in the UI for the grid view displaying progress of tasks and offering the ability to cancel tasks.
- *DatabaseConnection.cs* – Sets up and persists a connection to the MBS database, to be used for querying information about specific tasks.

The core of the application to look at is within the LaunchTask method within the Task.cs class. In here, it can be seen where the work is split up, each subtask is sent individually, and how. Also it covers merging of results and sending feedback and metrics back to the Form.cs class.

Requirements:

- Jni4net must be used to create the proxies to the PolyClient (see jni4net documentation for more detailed instructions). These proxy files must be included into the project and located in the same directory as the .exe
- Jni4net .dll and .jar files must be located in the bin\Debug directory, or located with the .exe file in order to run
- Within the *coherence.jar* file, there is a file called *coherence-cache-config.xml* which **must be modified to list the port(s) and machine(s) that it is to attempt to connect to.**
- Within the *coherence.jar* file, the *pof-config.xml* file **must be modified to list any tasks that it must understand how to execute.**

4.3 PolyClient (Java)

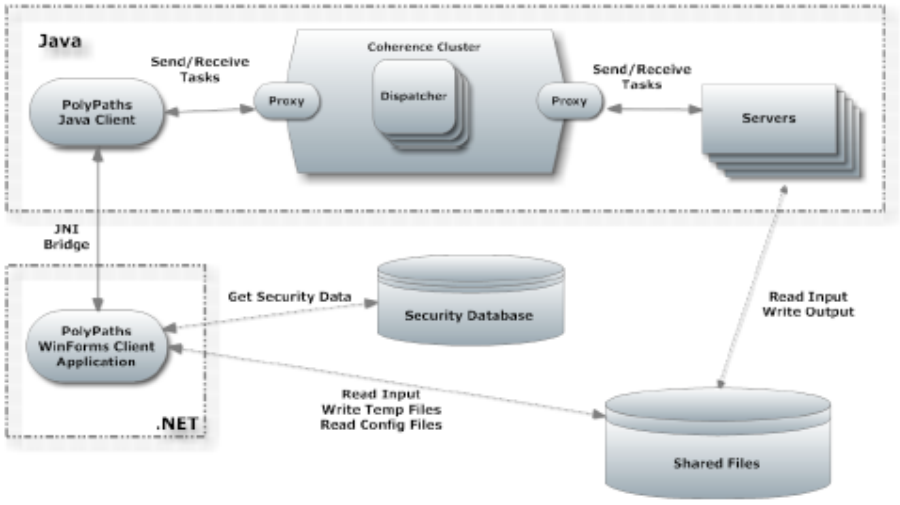
This project contains the java classes necessary for actual execution of tasks. There are three classes to consider:

CommandResumableTask.java – this class is the 'task' that is the instructions to the server specifying what work is to be done. In this application, this is simply to take

the given command line argument and executed it as a process. It implements the ResumableTask and PortableObject interfaces, allowing for it to be used in the Coherence Cluster. This class appears in two specific locations, the resulting jar file which a proxy to .NET is built upon, and in a specified folder within the same directory as the server, to be referenced in the server's pof configuration. (more specifics about this can be found in the jni4net and cluster sections, respectively)

TaskExecution.java – This class is the client side to the operation, and is what is responsible for entering calculation requests into the cluster. It does so by use of the 'submit' method, and waits for the outcome with the 'wait' method. It can be used directly or also through creating a jni4net bridge to the resulting jar file that can be invoked from .NET.

CoherenceTaskStats.java – This is a simple class that allows for multiple runtime metrics to be returned to the calling platform. It is specifically within the Polypaths class structure, but is not specific to the Polypaths function.



Final structure of the Polypaths implementation

8

4.4 Known Unresolved Bugs

- In certain circumstances, when the PolyClientWinForm should happen to encounter an unresolved exception and crash, there is a chance that the temporary output files of BatchCal executing on the servers may occasionally be written into the directory that the task processing servers are run from. This is not easily repeatable, and is only observable through the appearance of errant output files appearing in the server's directory.
    - Likely cause: upon the client crashing, the server has no application waiting for its feedback, and the 'current directory' may be getting reset to point to the local directory where the server was spawned from. When the writing of the output file occurs, it may be written here instead of the intended location

5 Westminster

5.1 How to Compile it:

1. Once the solution is done editing, build it (create the DLL)
2. Copy the DLL into the *lib* directory
3. Run *generateProxies.cmd*
4. Run the application

Notes:

- In case of making any changes to *CommandResumableTask.java*, you will need to get the binary file (.class) and copy it into the server.
- If the solution is changed, copy the DLL and the created JAR into the Coherence server folder
- Make sure that the Coherence Cluster executes the JNI bridge locally (cannot be on a shared drive), otherwise this will create a fatal java error.

5.2 How to Run it

1. The Coherence cluster needs to be initialized
2. WestServer needs to be running on the server(s)
3. Run *WestClient.cmd*

5.3 WestTest (.Net)

This application is a server controller for the Westminster server. This project calls the Westminster server through an interface. The code was originally created by Andrew Clark.

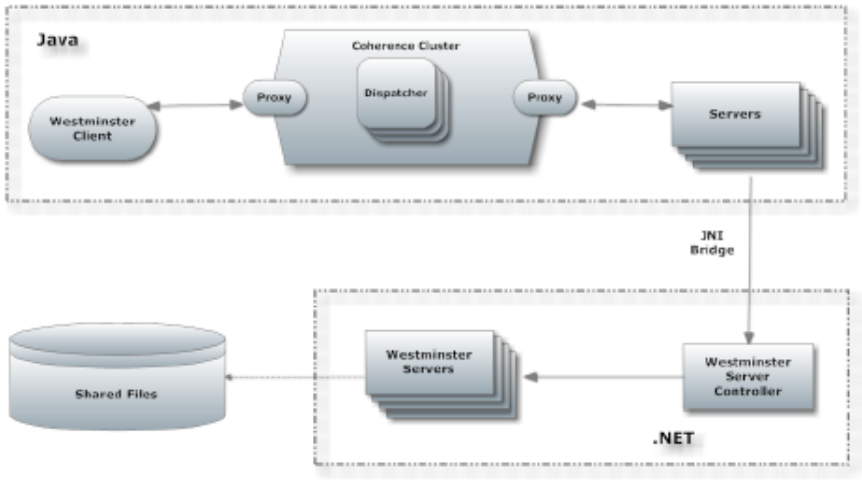
5.4 Java (application, gui, utilities)

The project is divided into three different packages:

- **Application:** This package contains the main class to execute the task.
    - Client.java: The main class to execute the program
    - CommandResumableTask.java: A class containing information of how to execute the task into the server.
    - TaskSender.java: Wrapper to the task for the GUI.

- **Gui:** This package contains all the classes to display the Graphical User Interface

- **Utilities:** This package contains two helper classes. One to read and split the CSV files, and the second one to filter the files by name.



Final structure of the Westminster implementation

5.5 Known Unresolved Bugs

- The way the program checks if the task is done, is by checking the output path of the file. The application checks if there is a new file with the specific name. In

theory this should work, however when there is a repeated file Westminster creates a filename with a number. The main bug with this is that if the user sends two tasks with the same output filename, the program will indicate that the two tasks are done, even if the second one has not even started.

12

6 How to Develop the Current Structure Further

In order to adapt the architecture to handle new applications and new types of tasks, between two and four new classes must be written in order to expand the capacity. For the sake of explanation, the focus will be on hypothetically adding the functionality to execute Python scripts in a distributed fashion through the existing framework.

Task (Java) – The first step is to write a new task, which is the actual instructions that will be provided to the server regarding how to execute the work. This class must implement both ResumbaleTask and PortableObject interfaces. This can be closely based off of the existing tasks that have been written, and will contain the instructions specifying what is to be done. In the case of the python script execution, this could likely take the form of a direct call to a python interpreter with the given script (passed either as a parameter to the server or as a reference to a script stored in a shared directory). This would be performed on the server(s) and relying on the server resources for processing power. Once this task is written, it must first and foremost be copied (entire directory) into the folder of the call to execute the task processing servers. Furthermore, the server's *pof-config.xml* must be modified to include a reference to the path to the new task. This will cause the server to be able to recognize how to perform the given task. Also, the task class must appear in the compiled jar file that is used with jni4net to build a proxy that can be accessible from .NET, if this step is necessary on the client.

Client (Java) – next, it will be necessary to create a client class in Java that is used to connect to the cluster and send work to it. This can be closely based off of the already existing examples of Polypaths and Westminster, and the primary concern is how to send data to the task. Depending on the structure of the task, it will be either sending the arguments to perform work on directly, or provide a reference to where they can be found. This will be the likely implementation to be seen in a situation if a Python script-handling task was developed. This class must also appear in the .jar file that will be necessary for any .NET client that might invoke it. Examples of this can be seen with Polypaths, while a standalone version is seen with Westminster.

Server (.NET) [possible] – A possible extension of code that may be necessary as well is the creation of a server-side application in .NET. This is requirement specific, and largely depends on whether the task will require direct interaction with other .NET code bases. In the Polypaths example this was not necessary, as the process could be started directly from Java. However, Westminster provides a good example of one possible means of creating this connection, again with references to jni4net for how to establish this. In the Python example, this section could be necessary if it is needed to be a persistent program that will be listening for incoming work, but for most circumstances the server would only need to be in Java.

Client (.NET) [possible] – Just as the server side may or may not need functionality in .NET, the client side has the option to do so if necessary. This would likely not be necessary for a Python script scenario, but can be needed for situations such as Polypaths, where the ultimate goal is to have a client framework that can be plugged right into an existing .NET system. Such a system would require the establishment of a JNI Bridge in the .NET code, as well as inclusion of a proxy to the client (in Java). A full example of this can be seen in the PolyClientWinForm.

With the above configured correctly, it then should be possible to have new tasks be added into the capabilities of the server farm. Note that it should require no modifications to the existing core of the coherence cluster, only to periphery applications on the client and server.

7 Contact

In case of any question please contact:

- Claudio Herreros <cherreros@wpi.edu>
- Jotham Kildea <jothamk@wpi.edu>

8 Work Schedule

While working on the project, the goals and objectives moved fluidly and continuously from week to week. As expected, there was a decent amount of setup work to begin acclimating to the workplace. By the end of the first week, however, much of the work of obtaining ID badges, installing software, and receiving administrative rights on computers had been accomplished.

Once this was over, the work of setting up a simple coherence cluster began. This required a large amount of effort towards setting up the computer environments correctly, and began to cause difficulties particularly with using the Processing Pattern in .NET. Fortunately, early in the second week a meeting was held with representatives from Oracle, which produced an introduction to and means of contacting one of the lead developers of the Processing Pattern. His assistance with fixing bugs in their software allowed the transmission of tasks from clients to servers, a major accomplishment for the overall project, to be achieved by the end of the second week.

Upon having a proof of concept for creation of a Coherence Cluster was completed, the next consideration was to assess the ability to extend this across multiple computers in a distributed fashion. Fortunately, the design structure that had been used and the functionality provided by Coherence allowed for this step to happen very quickly, and only required a few days of development before this could be achieved reliably and easily.

With a truly distributed Coherence system functioning, work began on outfitting it to be used to perform specifically needed tasks. This process required concise code that would be in Java to act on both sides of the cluster. Although the server side was able to remain in Java, the

client side also needed to have functionality in .NET. In order to achieve this goal, a complex process of developing the client side into a product that could be interoperable with .NET was undertaken. Through some trial and error, the conclusion was reached to use jni4net, a product specifically designed to provide limited interaction between the two languages. This product's implementation proved to be very difficult to work with but its use was unavoidable as the bridge between the two languages was necessary. Once jni4net had been implemented, the step of developing the .NET application to be used could begin.

Development of the graphical application that could be used to interact with the already developed Coherence Cluster was the next major objective, and began roughly in the early part of the fourth week of the project. This application underwent numerous iterations and evolved considerably throughout the project. Its development proceeded uninterrupted for the next few weeks, and was only finalized shortly before the final product was delivered.

As an additional focus of the development of the Cluster for use with PolyPaths was the creation of the assortment of algorithms used to partition work. These began to be developed approximately in the fifth week of the project, and were continually improved and added on to up until the middle of the sixth week. Along the way they had been developed to operate seamlessly within the developed application itself, and therefore little work was needed in order to combine the two.

Simultaneously, beginning towards the end of the fifth week, the development of an entirely separate feature to the Coherence cluster was begun. This took the form of implementing operation with Westminster, and additionally required the usage of jni4net on the client as well as the server. Unfortunately, the difficult problems with the server side were overcome and there

existed serious limitations to the jni4net product that prevented the client side from functioning in the desired fashion. As a result, the Westminster implementation remained in Java on the client side, and development lasted until the end of the sixth week. Nonetheless, this addition provided an exceptional proof of the fact that the cluster was as versatile as desired, and capable of handling vastly different tasks in parallel.

As the code development aspect of the project drew to a close the focus shifted once again, this time to the development of the final demonstration and presentation of the accomplishments. This involved not only the development of the final presentation, but also the benchmarking of performances of varying algorithms as well as structuring applications to be understood easily. This phase began roughly in the middle of the sixth week and continued through the end of the project. The presentation also underwent several revisions as presentations uncovered to improvements that could be made to the overall delivery.

In the final two weeks, the transfer of the code base occurred, and involved extensive commenting and documenting of the code base, as well as reviewing it with sponsors at BNP Paribas to familiarize them with the approach. In addition, due to the complexity of the systems implemented a technical guide had to be created, in order to introduce new developers into the system. This technical guide can be found in Technical Documentation. This handing over of the final product was the major conclusion to the project as a whole. Table 1 resumes how the team managed his time during the course of the project.

Table 1 – Work Schedule

Plan of Work	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
Meet with Contacts, familiarize with environment							
Develop product code, expand functionality and features							
Test code, fix any bugs that appear							
Prepare Presentation							
Write Report							