Distributed system for evaluation of strategies in the stock market and the soccer betting industry

DZIANIS BAZHKO

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:193

KTH Computer Science and Communication

Master's thesis

Distributed system for evaluation of strategies in the stock market and the soccer betting industry

Author:
Dzianis Bazhko

Supervisor:
Staffan Ulfberg

Examiner:
Vladimir Vlassov

July 4, 2013

Abstract

Development of software and strategies for prices, odds calculation, and risk management in the stock market and the soccer betting industry involves testing functionality using recorded real-world data. This process is called backtesting. If a system for backtesting is represented by software that is run on a single workstation then due to the complexity of mathematical computations, amount of recorded data and variety of strategies — backtesting can require significant amount of wall time. This is a problem because the development cycle increases in time. This paper proposes a design of a distributed system for backtesting that distributes the workload among available workstations considering limitations on the network usage.

Sammanfattning

Utveckling av programvara och strategier för priser, odds beräkning, och risk hantering inom stock marknad och fotboll vadslagning industri engagerar testning funktionalitet med inspelad verklig-värld data. Den här processen heter backtesting. Om ett system för backtesting är representerad av programvara som är kört på en arbetsstation sedan på grund av komplexitet av matematiska beräkningar, mängd av inspelad data och variation av strategier — backtesting kan kräva signifikant mängd av wall time. Det är ett problem för att utvecklingen kretslopp ökar i tid. Det här pappret föreslår en design av ett distribuerat system för backtesting som distribuerar arbetsbelastningen bland tillgängliga arbetsstationer med hänsyn till begränsingar på nätverk användningen.

Contents

Chapter 1

Introduction

1.1 Background

Development of information technologies was one of the factors that made possible creation of electronic communication networks in the finance world and betting exchanges in the betting world.

Electronic communication network is a system that automatically executes trades by matching prices of ”buy“ and ”sell“ offers submitted by trading firms. The logic of trading is the same as trading on the stock market, with a difference that communication with a trading exchange is done through a computer network instead of physically being on a trading floor and verbally communicating with representatives of trading exchange.

Betting exchange is a system that provides an opportunity for gamblers to ”back“ and ”lay“ outcomes (control their ”back“ and ”lay“ offers, get open offers on the market) via computer network. The differences from traditional bookmakers are: gamblers can ”lay“ outcomes on exchange, whereas bookmakers provide ”lay“ prices for backing so that gamblers can only ”back“ those prices; operations are performed via computer network.

The strategy, according which offers are created, is one of the factors that determine success of trading in such systems. It is a challenge for trading companies to come up with a strategy, the usage of which will generate money. One of the ways to make a strategy better is to search for weak sides and flaws by analyzing trades after trading. This approach can be costly since it is needed to analyze a considerable amount of trades in order to make some conclusions. Backtesting can be used to evaluate a strategy and not take a risk. To evaluate a strategy means to determine how good the strategy is, according to some metric. There could be several ways to do that. Some of them are:

- Use the strategy to generate offers by providing it recorded open offers from the market (market data). Establish a measurement of how good the offers, generated for a specific case, are (”case“ is used as a general term for stocks in the stock market and game events in the soccer betting industry).

- If there are several strategies available then one can run them simultaneously so they will compete with each other. Comparing results can give an idea which strategy performs better for a specific case or in general by evaluating on several cases.

Both ways to make some evaluation of a strategy can require reasonable processor and database time because of the complex mathematical computations and amount of market data. The company, where this thesis has been done, uses a separate system for the process of backtesting.

1.1.1 The current system

The current system (Figure 1.1) is a multithreaded, .NET 4.0 console application. The backtesting process is configured by providing the system with a configuration file. Backtesting configuration includes a set of strategies to evaluate, a set of cases and a set of evaluators (functionality that evaluates strategies). The data, that represents a case, is read from a SQL Server database (”historical database“ further). Each strategy is evaluated on a case using the specified evaluators. The evaluation results are written to a separate database(”backtesting database“ further).

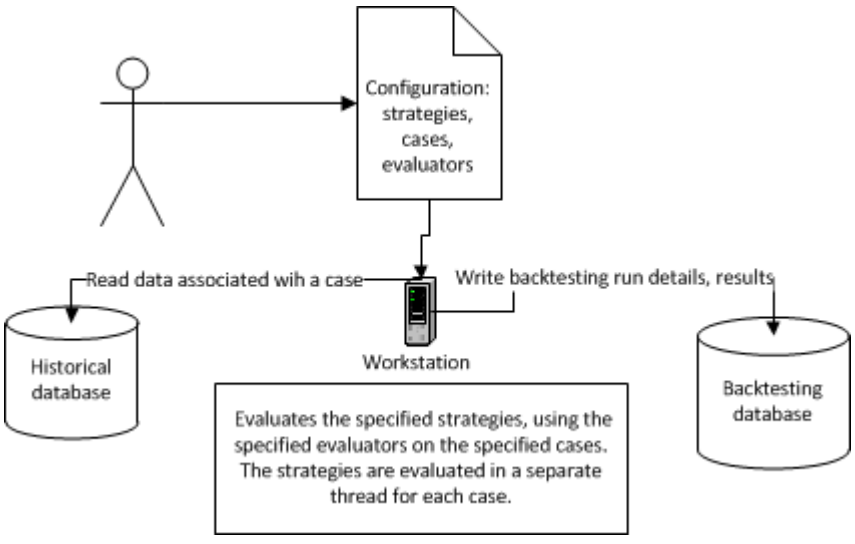

Figure 1.1. Current system configuration.

1.1.2 Workflow

The Figure 1.2 shows the workflow of running a backtesting process in the current system. The backtesting process is started by initializing the configuration

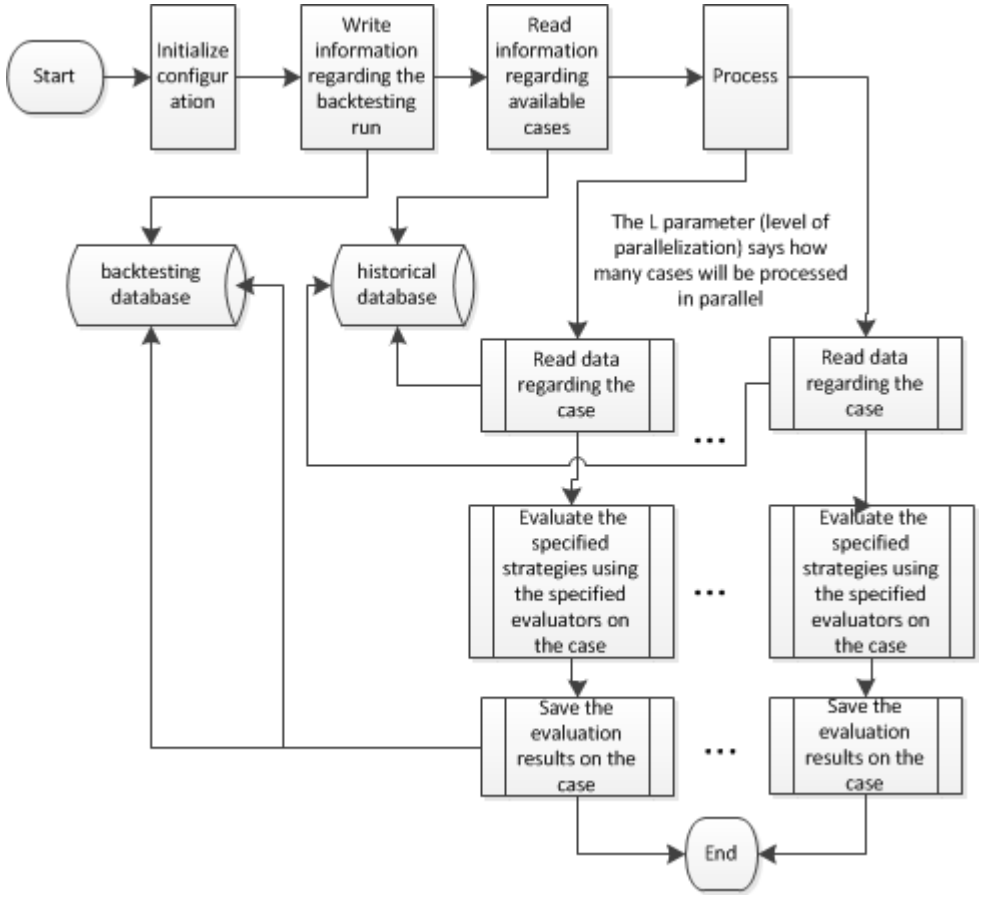

Figure 1.2. Current system workflow

(provided by user: cases, strategies, evaluators) of backtesting run, writing backtesting run information to the backtesting database, reading information regarding available cases from the historical database. The backtesting is represented by evaluation of strategies on all cases and writing evaluation results to the backtestng database. The evaluation of strategies on a case is performed in a separate thread from evaluations of strategies on other cases (multithreading is handled by Task Parallel Library[1]). Backtesting on a single case is represented by reading case related data from the historical database, evaluating strategies on the case and writting evaluation results to the backtesting database.

[1]Task Parallel Library (TPL) is a set of abstractions provided by the .NET framework 4.0 in order to ease the development of multi-threaded systems. The abstractions handle low level details, like synchronization of threads, and provide more high-level APIs to parallelize work execution. [1]

1.1.3 Usage scenario

In order to run the backtesting process using the current system, user performs the following steps:

1. User creates a configuration file, where a set of strategies, a set of evaluators and a set of cases are specified.
2. User runs the system, specifying the name of the configuration file and a level of parallelism, which indicates how many cases will be processed in parallel.
3. User observes a reported progress of the backtesting process and waits until the system reports about the end of backtesting.
4. User uses results of the backtesting process in the backtesting database for analysis.

1.2 Problem definition and motivation

The backtesting process takes from days to weeks to complete, depending on a configuration, and it is getting longer since the amount of collected data increases. This is a problem because the development cycle increases in time.

Some work has been done in order to solve the problem and the following changes to the system were introduced:

- The analysis of performance showed that 40% of CPU usage was spent on garbage collection. The code was updated to generate less garbage. The mode in which garbage collection worked was changed from default (”workstation“) to ”server“.
- The system intensively writes to the database. The amount of threads in the system was increased in order to use the CPU time for computation when the data is being written to the database.
- The tests are sorted by strategy so the resources needed for a strategy implementation can be released quicker.
- The results of some complex computations, that occur more than in one strategy, are cached in the database.

No metrics were established, but the performance increased, because the complexity of an average test run has increased and the wall time of backtesting run stayed approximately the same.

It was concluded that the technical capacity limits performance of backtesting. Adding more technical capacity should increase performance of the system as more

tests will be run in parallel. This is possible, because evaluation of a strategy on a single case can be treated as a separate process from evaluation of a strategy on other cases. The evaluation of a strategy requires: market data of the case which is stored in a database, mathematical calculations and saving results to the database (a separate one from the one where market data is stored). The mathematical calculations require significant CPU time, so it is a good opportunity to distribute the workload among workstations. However, important aspects should be taken into consideration:

- Some workstations are located close to the historical database server with market data and some are far away. The Internet speed between the database server with market data and workstations that are far away is currently limited to 2 megabytes per second.
- Market data (for a single case) uses several megabytes of memory and as the workstations are located in different geographical locations, the time to transfer data and the bandwidth usage should be taken into account.
- The previous point applies to saving evaluation results (for a single case) that can require several megabytes of memory also.
- Selection and distribution of tasks should be preformed by taking into consideration the points above.

The company has a reasonable number of workstations which are not all used all of the time and basically unused during nights. This technical capacity can be used for backtesting. It is also possible to have separate servers for this system. The existing system for backtesting cannot be run on several workstations.

1.3 Requirements

The functional and nonfunctional requirements were established for a new system that was supposed to solve the problem. The functional requirements were derived from the functionality that the current system has and the functionality that would distribute the workload. The nonfunctional requirements were derived from the goal of the project.

1.3.1 Functional requirements

By ”user“ below is meant a person who initiated a backtesting process:

- The system accepts a configuration file that represents a set of cases, a set of strategies to test, a set of evaluators and puts it for execution.
- The system distributes workload among available workstations.
- The system reports an overall progress.

- The system tolerates the failures of nodes.
- New workstations can be added to the system without restarting the system and test runs.
- The level of load of workstations can be controlled either by a user or an owner of machine.
- User can abort the execution of a backtesting run.
- User can change the priority of a job that is waiting to be executed.
- User can provide time constraints for workstations that define the time during which the jobs can be run.
- The system logs execution of tests, preferably in a single log file for each test run/user.

1.3.2 Nonfunctional requirements

Performance. In this project performance means amount of cases evaluated in a certain period of time. The system decreases significantly duration of a backtesting process in comparison to the current system depending on the number of workstations involved in the process.

Scalability. The duration of a backtesting run decreases nearly proportionally with the increasing technical capacity. Due to the fact that the evaluation of case is a separate process and the system has a distributed behavior.

The bandwidth usage between workstations is not increased considerably and thus does not impact the production activities. The system takes into consideration the topology of the network:

- The kinds of jobs that it puts for execution may need different type of data that is located on one of the workstations.
- The geographical distance between the workstations - the cost of transferring a job and the size of a job are taken into account.

1.4 Related work

A research of possible solutions of the problem was done. One of the alternatives was to use existing software for distribution of workload in the backtesting system. The ”Hadoop YARN“ and ”Qizmt“ software systems were considered for usage. Both of the systems had some negative aspects when using in the backtesting system.

Hadoop YARN is a framework that provides a functionality to process big chunks of data by distributing the workload across workstations. It is scalable, highly-available and provides an ability to handle failures on application level. It has a MapReduce programming model [6].

- The system is written in JAVA, thus HadoopStreaming needs to be used. The nodes in the system will exchange data that needs to be converted from JAVA to .NET representation.
- The resources consumption of the workstation can not be controlled.
- The network bandwidth between the workstations is not considered when scheduling jobs.
- It is not obvious how the specifics of the backtesting domain, such as several types of tasks that need to be executed in a specified order, can be handled by the system.

MySpace Qizmt is a software for development and execution of .NET distributed applications on Windows servers. It has a MapReduce programming model. [7]

- There is no way to control the load of the workstations.
- The network bandwidth is not taken into account.
- It is not obvious how the specifics of the backtesting domain, such as several types of tasks that need to be executed in a specified order, can be handled by the system.

Both systems above doesn't satisfy some of the requirements in Section 1.3.

Outsourcing the process of backtesting to a cloud was considered as an alternative for solving the problem. Because of the negative aspects, listed below, when using private and public clouds, outsourcing backtesting process to a cloud was considered to be unsatisfiable. The drawbacks of using a public cloud for backtesting are:

- The data needs to be stored in the cloud. Security issue.
- The software needs to be deployed on the servers of a cloud. Security issue.

The drawbacks of using a private cloud for backtesting are:

- There is a step for setting up a cloud at the company and managing it in future.
- The hardware capacity that company has, may not be enough to run the cloud.

- The usage of a cloud adds an additional layer between application and technical capacity.

1.5 Objective

The goal of this thesis is to significantly reduce duration of a backtesting process by distributing the workload among workstations. In order to make a backtesting process faster, the following steps were performed:

Investigation of performance benefits of distributed approach

- The performance and throughput of the system were analyzed.
- Tasks (parts of functionality that is reasonable to distribute) were identified.
- Approximate throughput of distributed version of the system was calculated based on collected data.

Method

- Functional and non-functional requirements were used to identify use cases and design considerations.
- A high-level architecture of the prototype was designed.

Overview of used technologies

- An appropriate technology for realization of the prototype was found.

Implementation

- The prototype was implemented.

Evaluation

- The performance overhead of heaving an additional layer in the system was measured.
- The performance benefit of using distributed version of the system was analyzed.
- Requirements, that are satisfied and not satisfied by the prototype, were identified.

1.6 Limitations

An investigation was done in order to identify the tasks that should be distributed. An averaged data was used in calculations of amount of data needed for evaluation of a case, amount of processor time to run the evaluator, and amount of data that represents the results. Because of the usage of averaged data in calculations, the throughput that was calculated at the investigation step could vary from the throughput of the prototype calculated at the evaluation step. Also, the performance and scalability of the developed system was evaluated on less than 20 workstations.

Chapter 2

Investigation of performance benefits of distributed approach

This section describes the first step in the "Approach" Section 1.5. The goal of this step is to collect necessary data about the current system: performance, throughput, available resources and environment setup in order to reason about possible solutions of the problem in the "Method" Section 3.

The investigation was started by analyzing performance of the current system. The results of the analysis were used to identify what parts of functionality was reasonable to distribute, i.e. tasks. Based on the collected data, the approximate throughput of the distributed version of the system was calculated.

2.1 Analysis of the system's performance

In order to analyze performance of the system, the *throughput* parameter was used to describe how useful the system was. Performance counters were used to collect performance data of the system and hardware for all steps of the backtesting process. The throughput of the system was measured as a whole and parts of functionality separately.

2.1.1 Throughput definition

The process of backtesting (see Figure 1.2) can be seen as a two steps process: initialization and processing of strategies on a case. During the initialization step, configuration (cases, strategies, evaluators) is prepared, information regarding available cases is received from the historical database, information regarding a backtesting run is saved in the backtesting database. During the processing of strategies on a case step, the data that represents a case is received from the historical database, the evaluation of strategies is performed using the case's data and evaluators, results are saved in the backtesting database.

The throughput parameter, $V_{Throughput}$, was defined as an amount of cases, on which, strategies were evaluated during a certain period of time:

$$V_{Throughput} = N_{NumberOfCases}/T_{TotalTime} \tag{2.1}$$

, where $V_{Throughput}$ — throughput, $T_{TotalTime}$ — total time of backtesting, $N_{NumberOfCases}$ — number of cases on which backtesting was performed.

The total time of backtesting process is comprised of:

- Initialization
  - $T_{ReadAvailableCases}$ — time to read information regarding available cases from historical database.
  - $T_{WriteBacktestingRunInfo}$ — time to write information regarding the backtesting run to backtesting database.
- Processing of strategies on a case.
  - $T_{GetAlreadyRunTestsForCases}$ — time to query backtesting database for tests that have already been run for the case, strategies and evaluators.
  - $T_{ReadCaseData}$ — time to read data regarding a case from historical database.
  - $T_{EvaluateStrategiesOnCase}$ — time to evaluate specified strategies using specified evaluators on a case.
  - $T_{SaveEvaluationOnCaseResults}$ — time to save evaluation results on a case to backtesting database.

Performance counters were used to measure the periods of time presented above and to collect performance data of the hardware.

2.1.2 Performance counters

Performance counters technology is a diagnostics tool that is built-in in Windows operating system and is used to record and interpret data regarding how well hardware devices or software performs[2]. There is a set of counters included into operating system to measure performance of system activities. It is also possible to write custom counters and integrate them into application to measure application activities. Both types of counters were used for performance analysis. The counters included into operating system were used to measure processor and memory usage of the application. The custom counters were used to measure time and memory used by certain parts of the application.

The application called PowerShell[1] was used for interpreting collected data. It allows

[1] "Windows PowerShell® is a task-based command-line shell and scripting language designed especially for system administration." [3]

to import collected performance counters data as .NET objects and use scripting language to perform operations. There is also a possibility to use PowerShell functionality in a custom application. The application was written that uses PowerShell functionality for interpreting collected data in order to automate the process of collecting performance data. The performance impact of recording performance data by performance counters was considered to be negligible.

The following built-in counters were used to to measure hardware usage:

- % Time in GC — "Displays the percentage of elapsed time that was spent performing a garbage collection since the last garbage collection cycle." [4]
- Allocated Bytes/second — "Displays the number of bytes per second allocated on the garbage collection heap." [4]
- Process: % Processor Time — "This counter will break down how much processor time each process is taking on the CPU." [5]
- Process: % User Time — "This will break down the amount of user time that each process is taking out of the total amount of processor time that the processes is using." [5]

The custom counters were integrated to the system in order to measure the periods of time presented in Section 2.1.1.

2.1.3 Performance of the system

Environment

The current system was run on a server with Intel Xeon processor E7540 @ 2.00GHz (4 processors with 6 physical cores each), 32 GB of RAM. The server had the hyper-threading[2] technology enabled, so the operating system treated such processor as one with 48 logical cores. Both databases were hosted on a separate server with Intel Xeon processor 5130 @ 2.00GHz (2 processors with 2 cores each), 10GB of RAM, running MS SQL Server 2008 R2.

Backtesting configurations

Two configurations (cases, strategies, evaluators) were used that define the smallest and the biggest sets of tests that was reasonable to perform. The $L_{LevelOfParallelization}$ and $N_{NumberOfCases}$ were the parameters that were changed during the tests.
The $L_{LevelOfParallelization}$ parameter is a level of parallelism parameter, which in the system means the number of cases that are processed in parallel. Each case is processed in a separate thread, thus, $L_{LevelOfParallelization}$ parameter effectively

[2]Intel® Hyper-Threading Technology increases performance of the processor by using an additional thread of instuctions per physical core[17].

controls usage of workstation resources. The $N_{NumberOfCases}$ parameter controls the number of cases to evaluate strategies on.

Two types of performance data were collected for each configuration: hardware performance (Processor Time, User Time, Time in GC, Allocated Bytes/second) and application performance ($T_{GetAlreadyRunTestsForCases}$, $T_{ReadCaseData}$, $T_{EvaluateStrategiesOnCase}$, $T_{SaveEvaluationOnCaseResults}$, $T_{TotalTime}$, $V_{Throughput}$, $T_{ProcessCase}$ — time to process a single case).

There was a set of parameters (Section 2.1.1) that were identified to be approximately constant or to have a maximum bound:

- $T_{ReadAvailableCases}$ — time to read information regarding available cases from a historical database on average was 323 seconds. The maximum amount of data read was 17.8 MB.

- $T_{WriteBacktestingRunInfo}$ — time to write information regarding a backtesting run to a backtesting database was on average 0.43 seconds. 797 bytes was on average saved in the backtesting database.

- $T_{GetAlreadyRunTestsForCases}$ — time to query a backtesting database for tests that have already been run for a case was 0.02 seconds if the returned collection was empty.

These parameters were excluded from performance measurements.

Performance measurements for minimum configuration

The Figure 2.1 shows the performance counters measurements of backtesting process with a minimum configuration. The maximum throughput of the system for the minimum configuration was identified to be 3.8 cases/min with $L_{LevelOfParallelization}$=48.

The Figure 2.2 shows performance counters measurements of hardware usage by backtesting process with a minimum configuration.

The hyper-threading technology was enabled on the server. The latter versions of Windows are aware of when hyper-threading is enabled. Thus, there is an additional thread of instructions per physical core which appears as a separate logical processor in Windows operating system. On two-processor machines, the instructions are first planned for execution on the physical cores. When the instructions are planned for execution on all physical cores, the instructions are planned on logical cores [16]. The server had 24 physical cores, thus, the logical cores were not used until more than 24 cases were evaluated by the system simultaneously.

According to the 2.1 the throughput of the system equals 2.9 cases/min and 3.8 cases/min when running on 24 physical cores and 24 physical cores with 24 logical

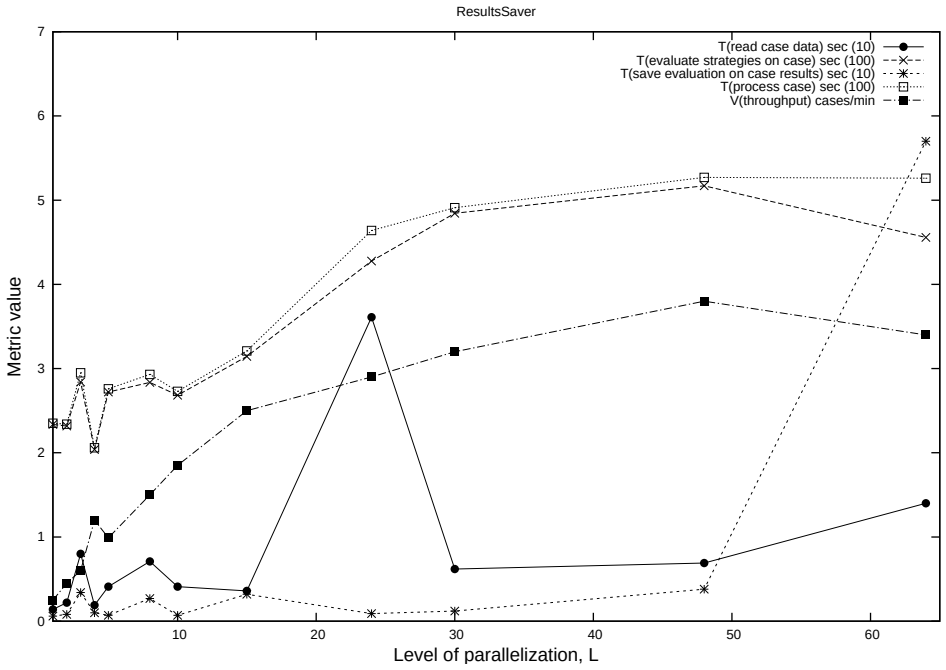

Figure 2.1. The system's performance data for minimum amount of tests.

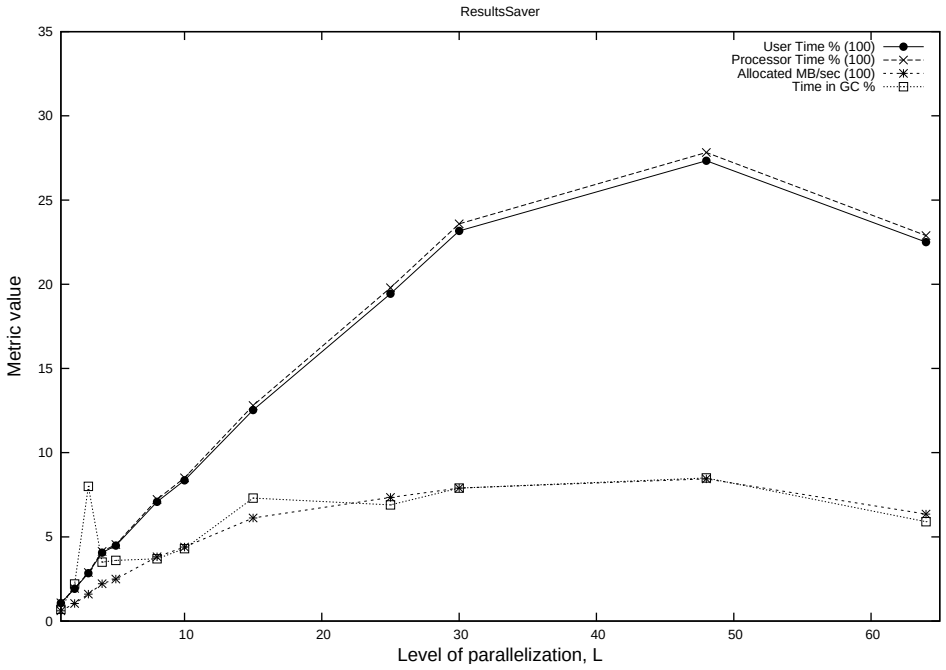

Figure 2.2. Hardware performance data for minimum amount of tests.

cores respectively. That means that hyper-threading gives an increase in performance by 31% for strategies evaluation with minimum configuration.

Performance measurements for maximum configuration

The Figure 2.3 shows the performance counters measurements of backtesting process with a maximum configuration. The maximum throughput of the system with the maximum configuration was identified to be 0.5 cases/min with $L_{LevelOfParallelization}$=15.

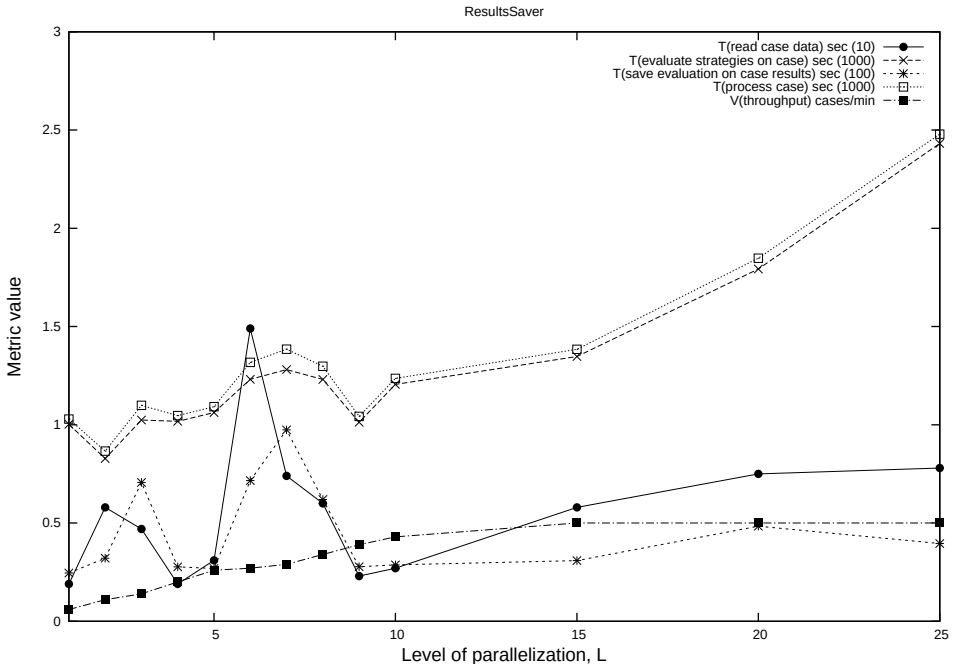

Figure 2.3. The system's performance data for maximum amount of tests.

The Figure 2.4 shows performance counters measurements of hardware usage by backtesting process with a maximum configuration. As the maximum throughput of the system was reached before all the physical cores were used, the hyper-threading did not affect evaluation of strategies with maximum configuration.

2.1.4 Performance of the system's components

Historical database

The historical database throughput was defined as a number of cases, for which data is read during a certain period of time. The system reads data that represents a case when evaluation is being started on a case. The application was modified to send requests simultaneously. The requests for reading were sent in separate threads and

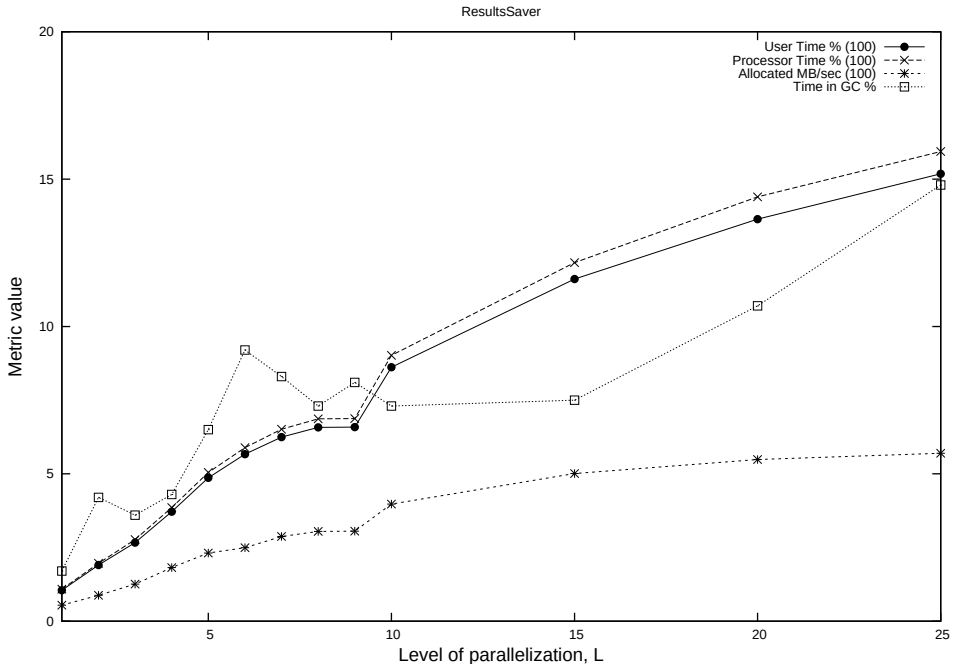

Figure 2.4. Hardware performance data for maximum amount of tests.

a separate SQL connection was created for each read operation. The performance and throughput of this part of the system were measured when there was a constant load. The Figure 2.5 shows the performance counters measurements of hardware usage and calculations of throughput for reading case data from historical database.

The results show that increasing level of parallelization did give a performance benefit. The maximum throughput was chosen with a $L_{LevelOfParallelization}$=5.

The average size of the data read from database was measured separately to be 6.9 MB. The reason for measuring size of data separately was that the serialization of an object tree that represents the data takes significant amount of resources and time in relation to read operation.

Backtesting database

The backtesting database throughput was defined as a number of backtesting results of evaluations on cases, written to a database during a certain period of time. The system writes evaluation results right after the evaluation on a case is finished. The performance and throughput of this part of the system were measured when there was a constant load. The application was modified to wait for evaluation results from all cases and send requests in parallel. The requests for writing were sent in separate threads and a separate SQL connection was created for each write operation. The size of evaluation results depends on the configuration of the back-

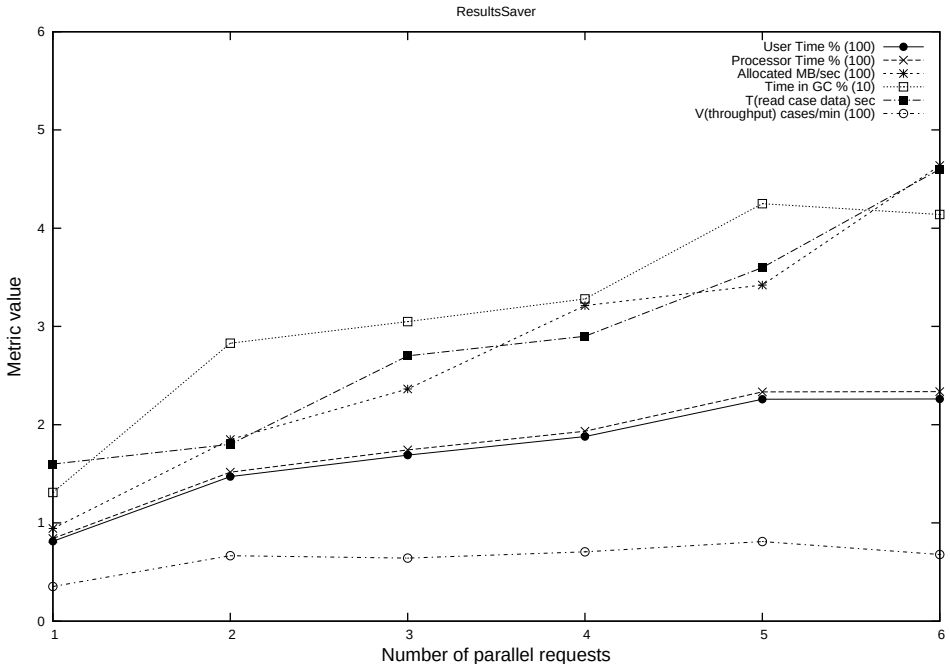

Figure 2.5. Historical database performance when reading data for a case.

testing run. That was the reason for measuring performance for the minimum and the maximum configurations. The Figure 2.6 shows the performance counters measurements of hardware usage and calculations of throughput for writing evaluation results for minimum configuration to backtesting database.

The Figure 2.7 shows the performance counters measurements of hardware usage and calculations of throughput for writing evaluation results for maximum configuration to backtesting database.

The results show that increasing level of parallelization did give a performance benefit. The maximum throughput for minimum configuration was chosen 94.7 cases/min with $L_{LevelOfParallelization}$=3 and for maximum configuration 15.4 cases/min with $L_{LevelOfParallelization}$=4.

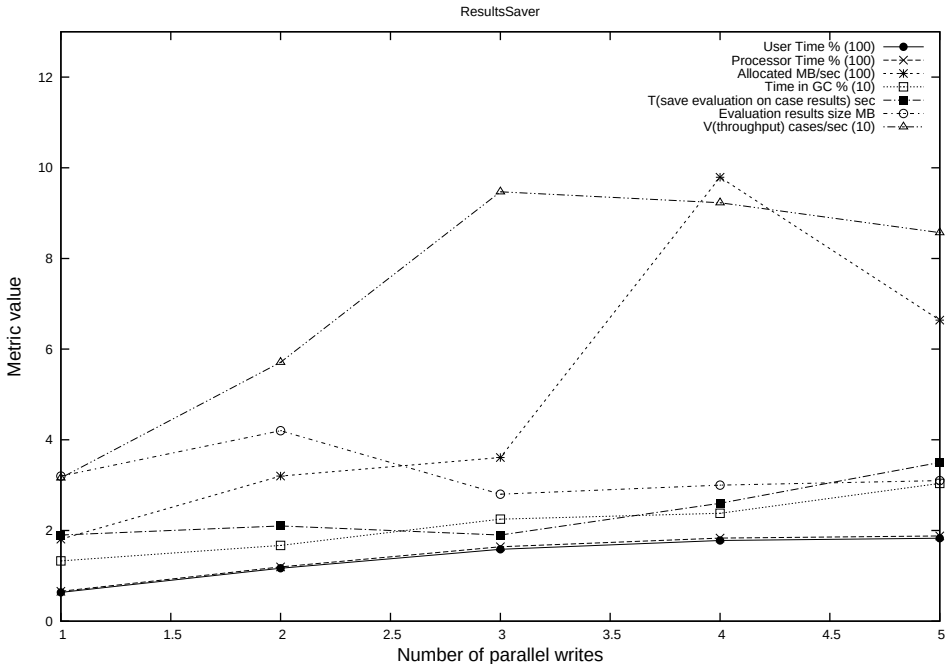

Figure 2.6. Backtesting database performance for minimum configuration.

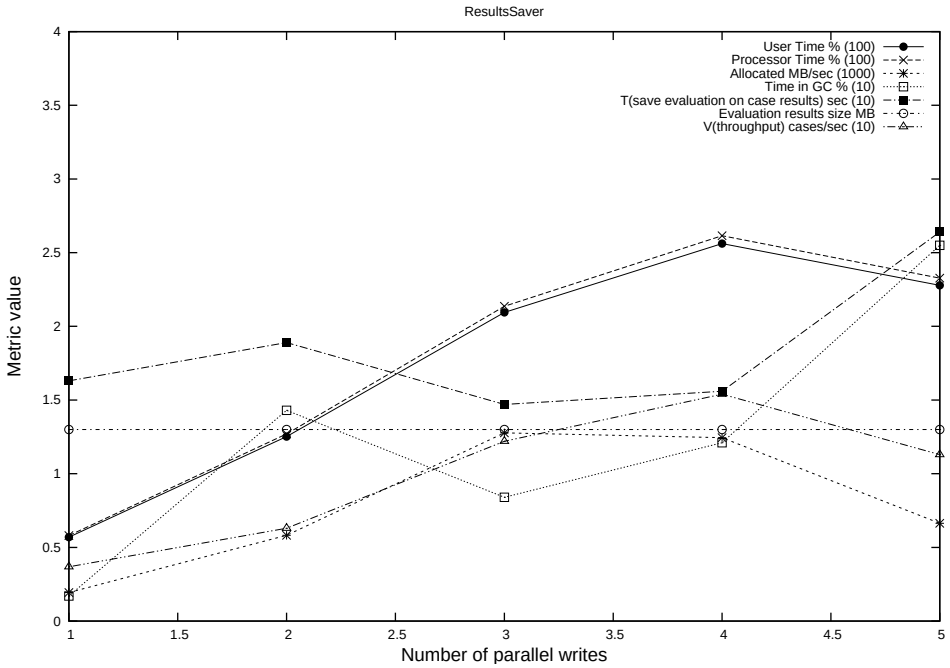

Figure 2.7. Backtesting database performance maximum configuration.

2.2 Identification of tasks

Tasks are parts of functionality that is reasonable to distribute from performance perspective. The Table 2.1 shows resources consumption by the parts of the system.

Step	**Functionality**	**Duration, sec**	**Data size, MB**	**CPU, %**	**Memory, MB**
Initialization	Receive information regarding available cases in historical database	323	17.8	-	-
	Save information regarding the backtesting run in backtesting database	0.43	0.000797	-	-
Evaluation of a case: minimum configuration	Receive data for a case from historical database	1.6	6.9	84.2	94.3
	Evaluate strategies on a case	233.7	-	108.4	60.7
	Save all evaluation results	1.9	3.2	65.9	180.8
Evaluation of a case: maximum configuration	Receive data for a case from historical database	1.6	6.9	84.2	94.3
	Evaluate strategies on a case	1003	-	108.8	54.2
	Save all evaluation results	16.3	1.3	58	193.3

Table 2.1. Resources consumption by the parts of functionality of the system.

The backtesting process consisted of two steps: initialization and evaluation of strategies on cases. The functionality in the initialization step was executed once per backtesting run and it was not considered as a task for distribution.

The evaluation of strategies on a case functionality was executed once per case. It included the following functionality:

- Receive data for a case from historical database. The average data size was 6.9 MB and the functionality required access to the historical database. This functionality was decided to be a ”read data“ task.

- Evaluate strategies on a case. The average time of evaluation of strategies on a case was 233.7 seconds. The functionality did not require access to a database. This functionality was decided to be a ”evaluate strategies“ task.

- Save all evaluation results. The average data size was 3.2 MB and the functionality required access to the backtesting database. This functionality was decided to be a ”save results“ task.

2.3 Approximate throughput of distributed version of the system

In order to know if the distributed version of the system could perform faster, the approximate throughput of the distributed version of the system was calculated. The available technical capacity and approximate throughputs of the components were used to calculate an approximate throughput of the distributed version of the system. Potential bottlenecks were identified based on the approximate throughputs of components.

2.3.1 Available technical capacity

The company had workstations in three locations. All the workstations were connected in a network.

Location A had 5 workstations with 1 physical core and 4 GB of memory each; 1 workstation with 24 cores, 32GB of memory; 1 workstation with 8 cores, 24 GB of memory. The historical database and backtesting database were located in this location.

Location B had 5 workstations with 2 physical cores and 4 GB of memory each.

Location C had 1 workstation with 4 physical cores and 8 GB RAM of memory each.

All workstations had hyper-threading technology enabled. Location A was connected with locations B and C in a network with 2000 Mb/sec speed and 55 ms latency. Location B and C were connected in a network with 30000 Mb/sec speed and 34 ms latency.

2.3.2 Calculation of an approximate throughput

A distributed version of the system was seen as a collection of components that were able to process tasks (see Section 2.2). The performance data for processing tasks in Section 2 was used to calculate an approximate throughput of the components having the available technical capacity presented above.

Read data component had a maximum throughput $V_{ReadCaseData}$=81 cases/min, see Figure 2.5.

Evaluate strategies component's throughput was comprised from throughputs of workstations in locations A, B and C. The following formula was used to calculate the throughput of evaluation of strategies:

$$V_{EvaluateStrategiesOnCase} = V_{EvaluateStrategiesOnCaseA} + V_{EvaluateStrategiesOnCaseB} + V_{EvaluateStrategiesOnCaseC} \quad (2.2)$$

, where $V_{EvaluateStrategiesOnCaseA}$, $V_{EvaluateStrategiesOnCaseB}$ and $V_{EvaluateStrategiesOnCaseC}$ are the maximum throughputs of workstations in locations A, B and C respectively.

The time periods to evaluate strategies from Figure 2.1 and Figure 2.3 were used to calculate $V_{EvaluateStrategiesOnCase}$ for minimum and maximum configurations.
For minimum configuration:
$V_{EvaluateStrategiesOnCaseAMin}$ = 6.6 evaluations/minute,
$V_{EvaluateStrategiesOnCaseBMin}$ = 2.3 evaluations/minute,
$V_{EvaluateStrategiesOnCaseCMin}$ = 1.2 evaluations/minute,
Thus, $V_{EvaluateStrategiesOnCaseMin}$ = 10.1 evaluations/minute.
For maximum configuration:
$V_{EvaluateStrategiesOnCaseAMax}$ = 1.14 evaluations/minute,
$V_{EvaluateStrategiesOnCaseBMax}$ = 0.55 evaluations/minute,
$V_{EvaluateStrategiesOnCaseCMax}$ = 0.2 evaluations/minute,
Thus, $V_{EvaluateStrategiesOnCaseMax}$ = 1.89 evaluations/minute.

Save results component had a maximum throughput
$V_{SaveEvaluationOnCaseResultsMin}$ = 92.3 cases/min for minimum configuration and
$V_{SaveEvaluationOnCaseResultsMax}$=15.4 cases/min for maximum configuration.

Network bandwidth between locations was taken into account. The time to transfer data within LAN was considered to be negligible. The following formula was used to calculate the time to transfer data between locations A, B and C:

$$T = Latency + DataSize/Bandwidth \quad (2.3)$$

The average size of data for a case was 6.9 MB Table 2.1. The average size of evaluation results was 3.26 MB for minimum (Figure 2.6) and 1.3 MB for maximum configurations (Figure 2.7).

The results of throughput calculations are presented in Table 2.3, where

V_{rd} — throughput of read data component.

V_{es} — throughput of evaluate strategies component. A, B and C indicate the locations for which throughput was calculated. T — total throughput.

Path	**Case's data**		**Evaluation results**			
	Trd (sec)	Vrd (case/min)	Tsrmin (sec)	Vsrmin (results/min)	Tsrmax (sec)	Vsrmax (results/min)
Between A and B, C (0.25 MB/sec)	27.6	2.2	13.1	4.6	5.3	11.4
Between B and C (3.75 MB/sec)	1.9	31.5	0.9	66.7	0.38	157.9

Table 2.2. Throughput of the network.

V_{sr} — throughput save results component.

V_{nrdAB} — network throughput for reading case's data.

V_{nsrABC} — network throughput for saving results.

Configuration	**Vrd**	**Ves**				**Vsr**	**Vnrdab**	**Vnsrabc**
		A	**B**	**C**	**T**			
Min	81	6.6	2.3	1.2	10.1	92.3	2.2	4.6
Max	81	1.14	0.55	0.2	1.89	15.4	2.2	11.4

Table 2.3. This table shows throughputs of parts of the system

For minimum configuration: the workstations in locations A, B and C have technical capacity to evaluate strategies on 10.1 cases in a minute simultaneously (V_{esT}). The evaluation results can be written for 92.3 cases in a minute simultaneously (V_{sr}). The data can be read for 81 cases in a minute from historical database simultaneously (V_{rd}). The network bandwidth limits the throughput of workstations in locations B and C from 3.3 cases/min to approximately 2 cases/min.

The maximum throughput of a distributed version of the system was estimated by the lowest component's throughput which is "evaluate strategy" component V_{es}=V_{esA} + V_{esBC}= 6.6 + 2 = 8.6 cases/min, taking into considerations network bandwidth.

For maximum configuration: the workstations in locations A, B and C have technical capacity to evaluate strategies on 1.89 cases in a minute simultaneously (V_{esT}). The evaluation results can be written for 15.4 cases in a minute simultaneously (V_{sr}). The data can be read for 81 cases in a minute from historical database simultaneously (V_{rd}).

The maximum throughput of the distributed version of the system was estimated by the lowest component's throughput which is "evaluate strategy" component V_{es}=1.89 cases/min.

2.4 Conclusions

The throughput of the current system was measured to be 3.8 cases/min and 0.5 cases/min for maximum and minimum configurations respectively. The maximum throughput of the distributed version of the system was estimated to be approximately 8.6 cases/min and 1.89 cases/min for minimum and maximum configurations respectively. According to the data, the benefit of using a distributed version of the system is 2.3 and 3.8 times better performance for minimum and maximum configuration respectively.

Chapter 3

Method

This section describes a proposal of how distribution of backtesting workload could be realized. The system requirements in Section 1.3 and investigation results in Section 2.4 were used during the design process.

It was decided that the functionality that distributes the workload would be a separate layer in the system. The existing system was updated to use the distribution of workload layer. The major reason for this choice was to have an easier maintenance of the system. The layer could be tested before merging it to the existing system. A system for testing was developed to ease testing of the layer.

For simplicity, the distribution of workload layer is referred as ”the layer“ further.

Use cases technique was used to capture main functional requirements.

The design considerations were identified based on non-functional requirements.

The architecture of the layer was designed based on use cases and design considerations.

3.1 Use cases

The use cases were captured based on the functional requirements described in Section 1.3.1. The process of capturing use cases was started with identification of actors. Actors were identified based on the entities, that interact with the layer, and their goals.

The layer is a component in the system, where the system has a specific business domain. The goal of a user is to solve the problem of long time execution of backtesting process with the help of the system. The system uses the layer in the

process of solving the problem by requesting distribution of tasks that represent parts of the problem. Thus, there is an actor called ”System“ that uses the layer to distribute the workload.

The system includes nodes that participate in the problem solving process by processing tasks. Such processing nodes are called as ”nodes“.

The nodes are run and controlled by the user that owns the technical resources, i.e. the person that decides how the technical capacity is utilized. This user and application, used to communicate with the layer, are represented by an actor ”NodeController“.

The Figure 3.1 presents a use cases diagram of the layer.

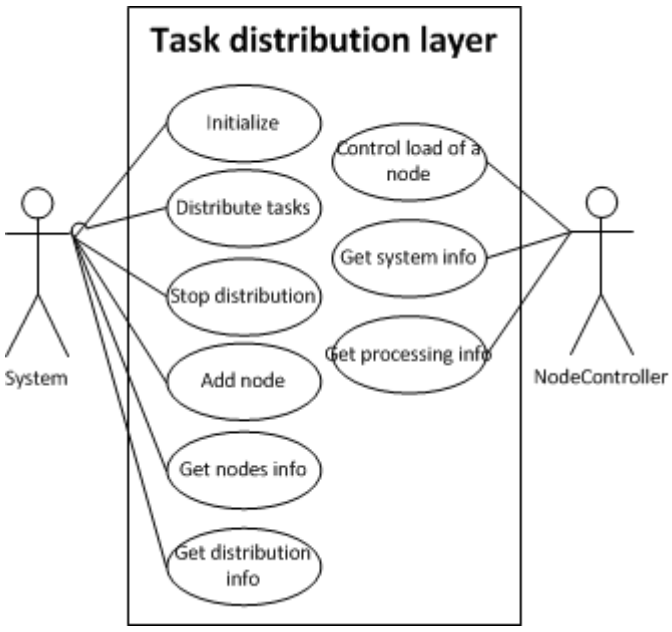

Figure 3.1. System use cases diagram

The use cases, illustrated in Figure 3.1, are described in this section.

The Table 3.1 presents the ”Initialize“ use case.

Name	Initialize
Description	The layer is initialized
Primary Actor	System
Preconditions	None
Trigger	The system invokes initialization of the layer
Basic Course of Events	1. The system requests initialization of the layer. The system provides the layer with information regarding nodes and tasks. 2. The layer performs initialization, requests initialization of the nodes and provides the system with a response.

Table 3.1. Description of ”Initialize” use case.

The Table 3.2 presents the ”Add node“ use case.

Name	Add node
Description	The system adds a new processing node to the layer
Preconditions	1. The layer is initialized.
Basic Course of Events	1. The system requests addition of a node to the layer. The system provides the layer with an address of processing node. 2. The layer adds the node and provides the system with a response.

Table 3.2. Description of ”Add node” use case.

The Table 3.3 presents the ”Distribute tasks“ use case.

Name	Distribute tasks
Description	The layer distributes the tasks, provided by the system, by sending them for processing to available nodes
Preconditions	1. The layer is initialized.
Trigger	The system invokes distribution of tasks
Basic Course of Events	1. The system requests distribution of tasks. The system provides the layer with a list of tasks. 2. The layer distributes the tasks to available nodes and provides the system with a response. 3. The system requests progress information from the layer. 4. The layer provides information regarding how many tasks have been processed, what tasks are being and has been processed by each node. 5. The distribution of tasks is completed when all tasks have been processed.

Table 3.3. Description of ”Distribute tasks” use case.

The Table 3.4 presents the ”Control load of node“ use case.

Name	Control load of node
Description	The NodeController controls the load of the node by the layer
Basic Course of Events	1. The NodeController requests a change of the load of the node. The NodeController provides the layer with a maximum load. 2. The layer changes the load of the node and provides the NodeController with a response.

Table 3.4. Description of ”Control load of node“ use case.

The complexity and priority were set for each of the use case, see Table 3.5.

Name	Actor	Complexity	Priority
Initialize	System	High	High
Distribute tasks	System	High	High
Stop distribution	System	High	High
Add node	System	High	Medium
Get nodes info	System	Medium	Medium
Get distribution info	System	Medium	Medium
Control load of a node	NodeController	High	Medium
Get system info	NodeController	Low	Medium
Get processing info	NodeController	Low	Medium

Table 3.5. The list of use cases.

3.2 Design considerations

The design considerations were identified based on non-functional requirements.

Fault-tolerance

- The layer is resistant to failures of nodes and is able to proceed with the distribution process.
- The layer doesn't tolerate failure of Scheduler.

Maintainability

- The defects are fixed without introducing major modifications to the layer.
- The modifications to the functionality of certain parts of the system doesn't introduce major modifications to other parts of the layer.

Modularity

- The layer consists of loosely-coupled components with well defined responsibilities.
- The components can be tested independently from other components.

Reliability

- The layer is able to perform the distribution of workload process until Scheduler is able to receive distribution requests.

Extensibility

- New features are added to the layer without major modifications to the architecture.

Robustness

- The layer is able to handle requests under stress conditions: large number of simultaneous requests, availability of low amount of memory.
- The layer tolerates invalid input.

Security

- The layer transmits data securely between communicating parties.
- The assemblies that contain system domain specific code are not exposed.

Interoperability

- The layer should support communication with software that is provided by different vendors.

3.3 Architecture

The high-level view of the system was a starting point of the design process. The main elements of the layer were identified based on the high-level view of the system. The interactions between elements in the layer were derived from the use cases (Section 3.1) and were presented in the sequence diagram. Based on the interactions between elements in the layer, the components and dependencies between components in the layer were presented in the component diagram.

3.3.1 High-level view of the system

The Figure 3.2 presents the high-level view of the system.

User uses the client application to control the backtesting process. The Scheduler is a component that is responsible for scheduling tasks and communication between nodes. The Client application controls the backtesting process by calling appropriate methods of Scheduler.

The DataProvider, StrategyEvaluator and ResultsSaver are the computational nodes in the system. The names originate from the task types that they are capable of processing. The processing nodes are autonomous, i.e. they do not require other elements of the system to be running. They receive and process requests from the Scheduler.

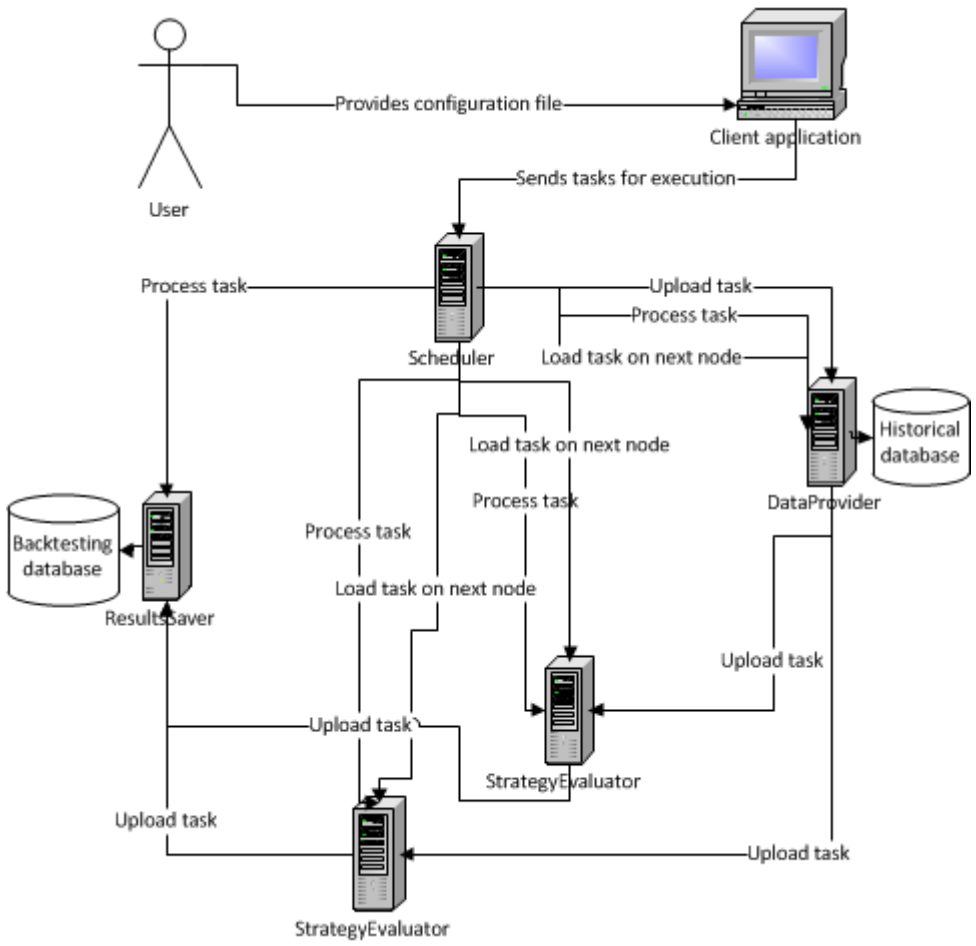

Figure 3.2. High-level view of the system.

The job in the backtesting domain is represented by backtesting on a case. Based on identified tasks in Section 2.2, the job is comprised of ”read data“, ”evaluate strategies“ and ”save results“ tasks. The tasks need to be processed sequentially.

When a job is being scheduled, the Scheduler searches for computational nodes that are able to perform ”read data“ task of the job. The node is chosen and the ”Upload task“ method is called on the node to upload the data that represents a task. Once the task is uploaded, the Scheduler calles ”Process task“ method on the node. The Scheduler is notified when the task has been processed. Then, the Scheduler searches for nodes that are able to process ”evaluate strategies“ task on the case. When such node is found, the Scheduler calls ”load task on next node“ method on the node that has just processed the task (”read data“) to upload the task on the node that will be evaluating strategies. When the task has been loaded on the next node, the Scheduler is notified and it sends a ”Process task“ request. The same sequence of steps is performed to save the evaluation results.

3.3.2 System elements

Task is a unit of work. Based on identified tasks in Section 2.2, there are three types of tasks: get data for a case(”read data“), evaluate strategies on a case(”evaluate strategies“) and save evaluation results to a database(”save results“ task).

Job represents backtesting on a case. It is comprised of tasks types listed above. The client provides the layer with a list of addresses of processing nodes, where the structure of addresses in the list shows the sequence, according which, tasks should be processed. The tasks, that represent a job, must be executed in a defined sequence.

Node, that processes tasks. For each task type there is a node that is able to process it. Such node is called a TaskProcessingNode. TaskProcessingNodes, that are used in backtesting doamin, are named: DataProvider, StrategyEvaluator, ResultsSaver.

Scheduler Execution of tasks in parallel requires a coordination of execution. According to the system requirements (Section 1.3), the Mediator pattern was chosen as the optimal solution. The intent of the Mediator pattern is to encapsulate the interactions between objects functionality into a separate object. By doing so, the interacting objects are coupled to the single object that encapsulates interaction logic and are loosely coupled between other objects [8]. The Mediator pattern is a behavioral pattern of Object Oriented Design. Scheduler is an entity that handles coordination and communication between other elements of the system, which are loosely coupled.

3.3.3 System elements interaction

The sequence diagram was chosen to represent interactions between elements of the system. The workflows of ”Initialize“ and ”Distribute tasks“ use cases are represented in ”Initialization“ and ”Tasks distribution“ sequence diagrams respectively.

Initialization process is shown in Figure 3.3.

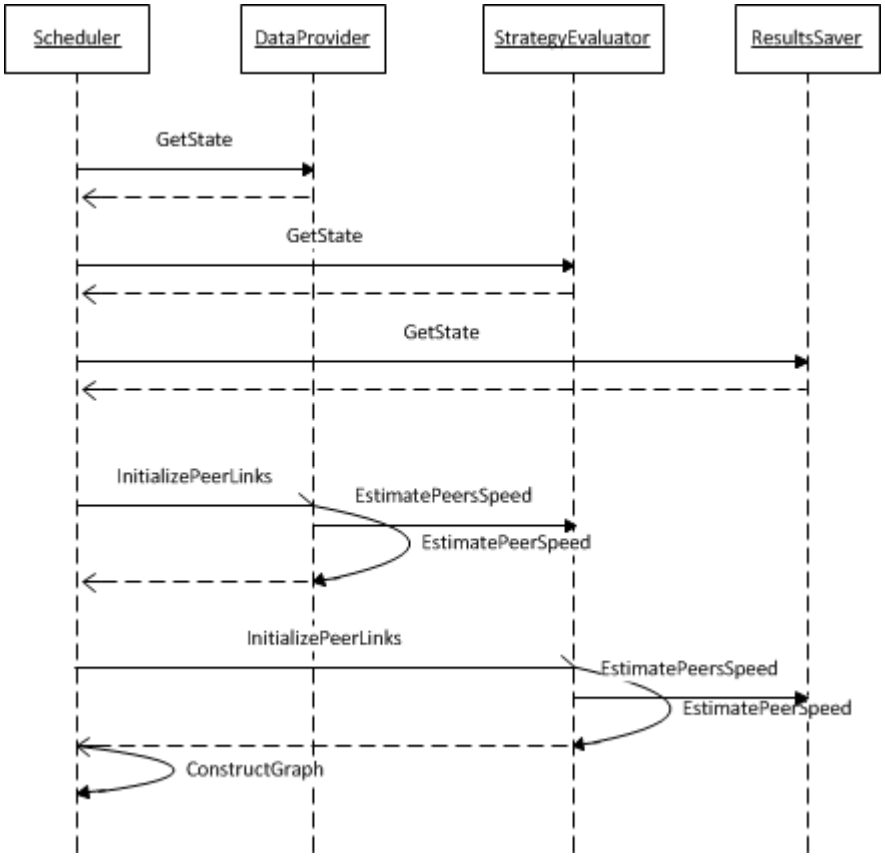

Figure 3.3. ”Initialization“ sequence diagram

1. The Scheduler verifies which processing nodes are running by getting their state.

2. The Scheduler requests initialization of peer links. Each processing TaskProcessingNode estimates the speed of transferring data between its peers, that are next in the sequence of processing a job.

3. The Scheduler constructs a graph, in which the vertices are the processing nodes and directional edges are the peer links.

Tasks distribution process is shown in Figure 3.4. The job, that represents evaluation of strategies on a case, is comprised of tasks of three types. The task of each type is distributed in the same way by the layer.

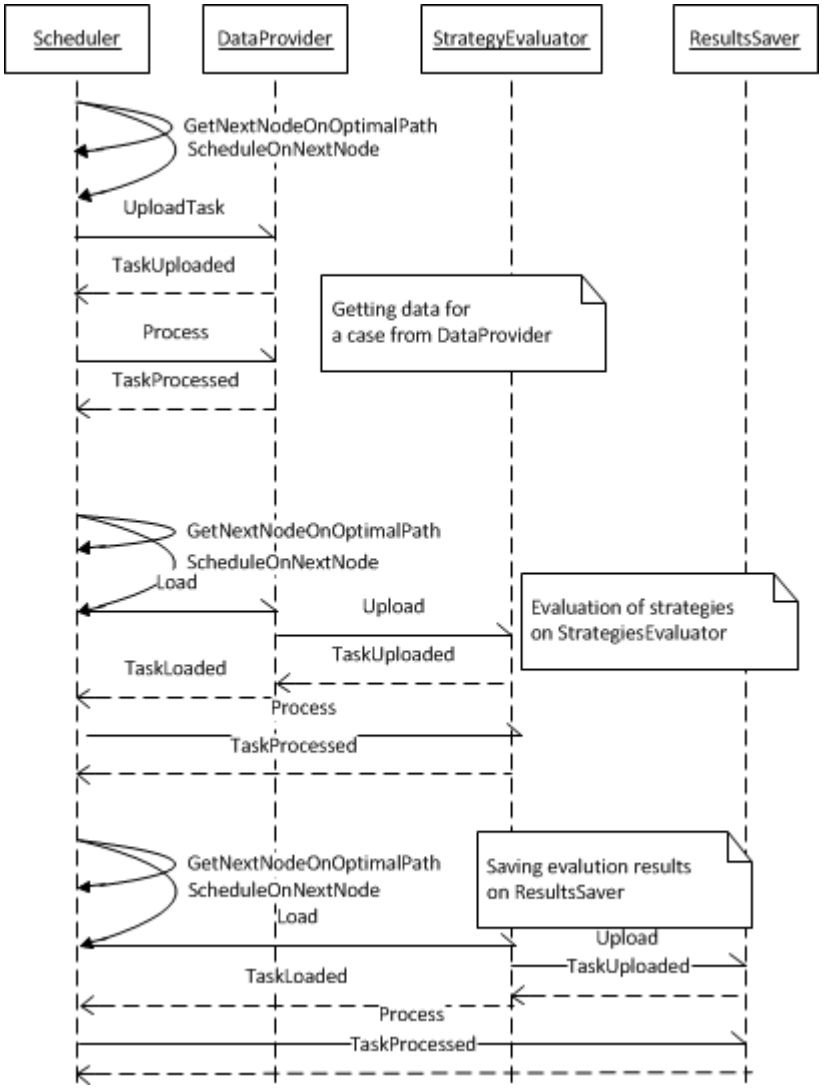

Figure 3.4. "Tasks distribution" sequence diagram

1. The Scheduler starts scheduling a task by selecting next TaskProcessingNode on the optimal execution path.
2. The Scheduler uploads the task on the selected TaskProcessingNode.
3. The TaskProcessingNode notifies the Scheduler when the task is uploaded.
4. The Scheduler requests processing of the task.

5. The TaskProcessingNode notifies the Scheduler when the task is processed.
6. The workflow is repeated from step 1 until all tasks are processed.

3.3.4 Layer component diagram

For the purpose of tasks distribution, the layer doesn't need to operate with all existing task types. The type of the task is relevant only for the component that processes the task. Thus, the layer uses a general concept — Task.

Nodes, that process tasks, can be generalized to a single concept — TaskProcessingNode.

As there can be several nodes, that process tasks of a specific type, there can be different processing paths of the job. ExecutionPathSearcher component represents a functionality, that searches for an execution path that can process a job in the optimal way.

The component diagram, that illustrates components and dependencies between components of the layer, is divided into two diagrams for representation purposes.

The system initialization component diagram is shown on Figure 3.5

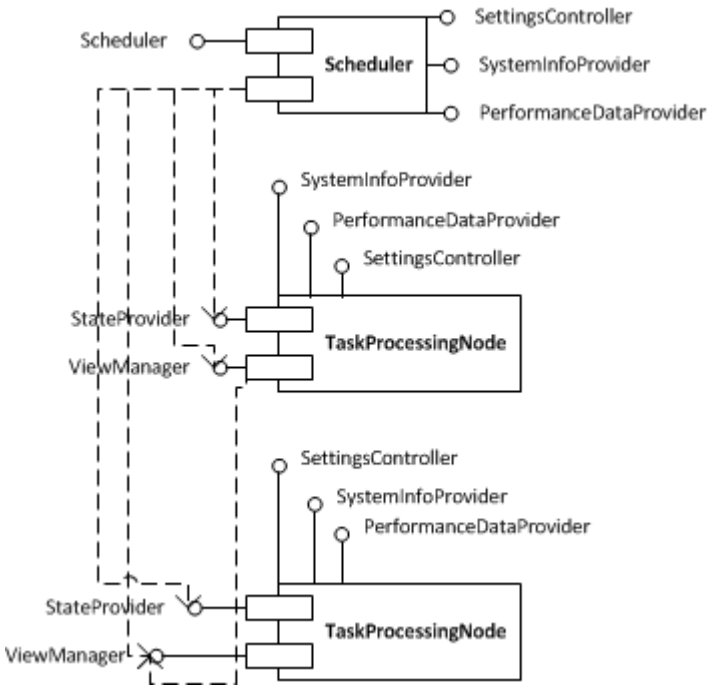

Figure 3.5. System initialization component diagram

The Scheduler exposes the Scheduler interface that contains methods for initialization, distributing tasks and getting backtesting run progress. It is the interface that the system uses to distribute workload.

TaskProcessingNode exposes the StateProvider, ViewManager, SystemInfoProvider, PerformanceDataProvider and SettingsController interfaces.

The StateProvider interface is used by the Scheduler to get TaskProcessingNode's state. The TaskProcessingNode's state is used to determine if the TaskProcessingNode is available and the number of tasks it can process in parallel.

The ViewManager interface is used by the Scheduler to request initialization of the view of TaskProcessingNodes peers. It is also used by TaskProcessingNodes to request estimation of data transfer speed between nodes.

The SystemInfoProvider interface is used by the NodeController to get information regarding what tasks have been and are being processed.

The PerformanceDataProvider interface is used by the NodeController to get information regarding load of workstation resources.

The SettingsController interface is used by the NodeController to change the maximum load of the TaskProcessingNode.

The system processing component diagram is shown on Figure 3.6

Scheduler exposes the TaskLoaderOnNextNodeCallback, TaskUploaderCallback and ExecutorCallback interfaces.

The TaskLoaderOnNextNodeCallback interface is used for notifying about successful or unsuccessful loading of a task on a next TaskProcessingNode.

The TaskUploaderCallback interface is used for notifying about successful or unsuccessful uploading of a task on the TaskProcessingNode.

The ExecutorCallback interface is used for notifying about the task has been processed or cannot be processed.

ExecutionPathSearcher exposes the OptimalPathSearch interface, that defines methods for searching execution paths and the optimal path.

TaskProcessingNode exposes the TaskUploader, TaskLoaderOnNextNode and Executor interfaces.

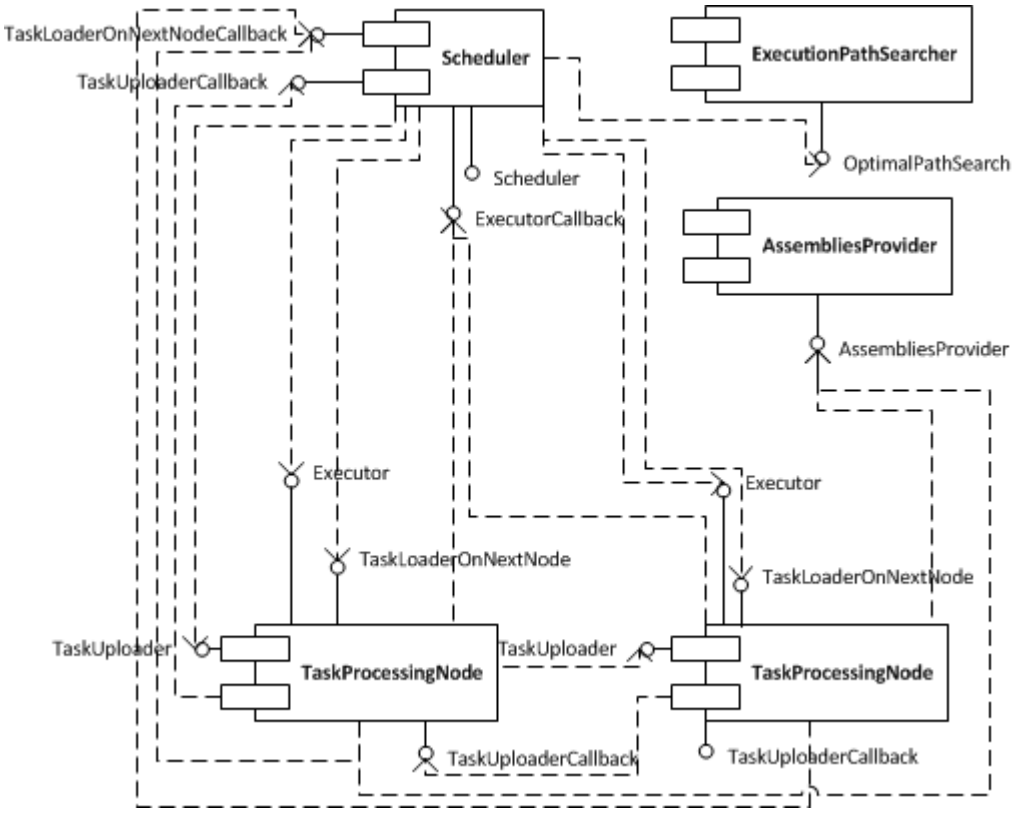

Figure 3.6. System processing component diagram

The TaskUploader interface is used by other components for uploading a task on the TaskProcessingNode.

The TaskLoaderOnNextNode interface is used by the Scheduler to request loading a task from the TaskProcessingNode to another TaskProcessingNode.

The Executor interface is used to request execution of a task on the TaskProcessingNode.

AssembliesProvider is a component that provides TaskProcessingNode with an ability to retrieve code libraries. As TaskProcessingNodes are distributed, there is a security issue with keeping libraries with domain specific code on the workstations that are used for processing tasks. The system domain specific libraries can be stored in a secure location and nodes can request them from AssembliesProvider. The AssembliesProvider component exposes AssembliesProvider interface that is used by TaskProcessingNodes to get code libraries.

3.4 Conclusions

The result of this section is the architecture of the layer that distributes workload among available workstations. In the design process the functional requirements were used to define the use cases and the non-functional requirements were used to establish the design considerations.

Chapter 4

Choice of technology

This section describes the process of choosing an appropriate technology for realization of the system and description of the technology. The requirements to the technology were established based on the system architecture and design considerations presented in the previous section. The .NET technologies for building distributed applications were considered based on the established requirements. The description of the chosen technologies and methodologies is presented in SOA and WCF sections.

4.1 Technology requirements

The requirements to the technology were established based on the system architecture and design considerations presented in the previous section.

Architecture

Transfering of tasks between the workstations was taken into account because the size of the tasks can reach several megabytes, based on the findings in Section 2.2. Thus, the technology should be able to transfer large amount data.

Uploading, loading and execution of tasks are asynchronous calls, thus, the Scheduler should be notified about the result of the operation. The technology should support two-way communication.

Design considerations

According to the security requirements, the technology should support secure transferring of data.

According to the modularity and interoperability requirements, it should be possible to use the technology for development the layer according to SOA tenets (see Section 4.3).

Integration to the existing system. The existing system is written in C# language and .NET framework 4.0. The system should use the layer and don't have a big performance impact in transferring data to the layer and back from the layer.

Deployment. The installation and maintaining the system should not cause much interactions from the user side.

4.2 .NET technologies

The .NET platform has the following means for building distributed applications: .NET Remoting, ASP.NET Web Services, Message Queuing technology and Windows Communication Foundation.

.NET Remoting is a technology for an interprocess communication that is accomplished by hosting remotable objects in one process and invoking object's methods from another process [10]. The technology conforms to "Component oriented-programming" paradigm. The components can be hosted on the same or different workstations and communicate with each other by creating remotable objects in one component and sending requests to such objects from other components. This is a .NET Remoting to .NET Remoting only communication, which makes interoperability with other technologies problematic.

ASP.NET Web Services (ASMX) is a technology that eases the development of web services by providing functionality to serialize, deserialize and transport messages using HTTP. [11] The interoperability is handled, but only HTTP can be used as a transport protocol. This makes transferring of big amount of binary data problematic.

Message Queuing (MSMQ) technology provides means for communication in a heterogeneous network, where the communicating parties can not participate in communication for some periods of time. [12]. It is used for applications where it is critical to keep messages even if the recipient is not able to receive a message, for example electronic commerce. The fault-tolerance is handled on the application level in the system, thus, the main feature of MSMQ is not required and is not worth spending resources for.

Windows Communication Foundation (WCF) is a .NET technology for development SOA applications [9]. It is a successor of former .NET technologies for communication. WCF is the only technology in .NET platform that fully satisfies our requirements. It was designed for building software that conforms to SOA principles.

4.3 SOA

The OASIS provides the following definition of SOA: ”Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.“ [14] In other words: *Service-oriented architecture* or *SOA* is an architecture, where the system is represented by a set of loosely coupled services that expose business functionality through well-defined interfaces and communicate with each other to accomplish some goal. According to [13], there are tenets of service-oriented architecture that describe how services should be built.

Explicit service boundaries show what is exposed to the consumers of the service. Only the functionality that desrcibes how to use the service and not how the service is realized should be exposed. The loose coupling between services is achieved by exposing only service contract, operation contracts and data contracts.

Autonomicity of services is the property of services that says that the services operate and are maintained independently from other services.

Operational contracts and data schemas are shared between services in order to be able to communicate with each other (invoke operations, transfer data). The technology specific representation of types, that realize contracts, is hidden from the caller.

Compatibility, based on policy is used to to declare what service is able to do and how to use the service. The policies do not provide information regarding technical realization of the functionality and constraints.

It was decided that the system would be built according to the SOA paradigm and its tenets.

4.4 WCF

WCF was chosen as a technology for building the system based on the established requirements. ”WCF is Microsoft's implementation of a set of industry standards defining service interactions, type conversion, marshaling, and various protocols' management.“ [13] This implementation provides:

- SDK for developing WCF services.
- Runtime environment for WCF services.
- Extensibility model, that is used to add facilities for development services.
- Basic facilities for development services.

This section describes the WCF facilities that were used for building the system.

4.4.1 Services

Service is a software application that exposes functionality via well defined interface. Client of the service is a software that can communicate with service by sending messages according to service definition. Service usually exposes metadata in WSDL format that describes how to use the service. WCF service is associated with a unique address. The address contains a transport scheme and location of the service.

$$< transportscheme >: // < machine >:< optionalport > / < baseaddress > / < URI > \quad (4.1)$$

Service metadata is used by the clients of a WCF service to discover the functionality that the service provides. Thus, the realization of WCF service is opaque for clients. The Web Services Description Language, or shortly WSDL, is used for describing service functionality in metadata that is accessible over HTTP protocol [13].

4.4.2 Bindings

The communication between client and service can be adjusted in many ways: communication pattern, transport protocol, reliability, security and etc.. The number of possible permutations of those parameters is huge and it would be hard to manage all possible choices. In order to manage all possible options of communication, WCF has a concept of "binding". Binding is a set of parameters that indicates how communication is performed. A service can have several bindings which allow to use the same service functionality with different communication settings.

4.4.3 Contracts

Service contracts describe operations of the service and are exposed by all WCF services. The "ServiceContract" attribute is applied on an interface or a class that represents a WCF service. The service contract interfaces can inherit from other service contract interfaces, but all elements in a hierarchy must have a "ServiceContract" attribute applied, as the attribute is not inheritable [13].

Data contracts describe the types of data that is exchanged in communication between client and WCF service. Data contracts are contained in service metadata, which allows clients to convert data that comes from the service to their native representaion [13]. WCF uses .NET serialization and deseriazation to convert data from .NET representation for transfering to client and data from client to a .NET representation respectively. The "DataContract" attribute is used to indicate the classes that participate in communcation between client and WCF service. Every class in a class hierarchy that represents a data contract needs to have a "DataContract" attribute applied, as the attribute is not inheritable [13].

4.4.4 Service operations

WCF provides several ways the service operations can be called. The basic one is a blocking call, when a client calls service operation and blocks until the operation execution is finished. WCF also supports ”one-way“ operations, duplex communication and streaming [13].

”One-way“ operations are also known as ”fire-and-forget“ operations. A client doesn't block execution after sending an operation request and continues execution. The client doesn't receive any operation response in this type of communcation.

Duplex communication allows client to call service operations and service to call operations that client exposes. Usually, it is used when the service operations are long running operations and the client needs to know about the result of operation execution. When the call is being made, the client specifies a callback object that needs to be called when the service operation result is ready.

Streaming is used for transferring big chunks of data. With other kinds of transferring data, WCF buffers messages that represent data on receiving side until the data is fully received. During the time the messages are being received, the client is blocked or in other words waits for response. In case of streaming transfer mode, the client is not blocked while the messages are being received and is able to process the data that has been received. In order to use streaming transfer mode, the operations must use ”Stream“ class as a method or return parameter.

4.5 Conclusions

.NET WCF was chosen as a technology for implementation of communication between the Scheduler and TaskProcessingNodes. The architecture and design considerations were taken into consideration during the decision making process.

Chapter 5

Implementation

The high-level structure, sequence and component diagrams of the layer were presented in Section 3.3. Those diagrams were not technology specific. The technologies for implementation of the layer were chosen in the previous section. This section describes how development of the prototype was performed using the chosen technologies and described architecture.

It was decided to use the latest, at the time of writing the thesis, version of .NET framework which was 4.5.

According to the component diagram in Figure 5.1, there were four types of communicating components in the layer: Scheduler, TaskProcessingNode, ExecutionPathSearcher and AssembliesProvider. ExecutionPathSearcher was represented by the library that Scheduler used (see implementation details in Section 5.3). Thus, there were three types of distributed communicating components that were represented by WCF services.

As stated previously (Section 3.3.4), the layer operates with a general concept of a Task, i.e. the specific type of a task is not relevant for the purpose of distribution. Thus, the layer was developed to be not dependent on backtesting domain. In the explanation of implementation below, it is specified what functionality belongs to the layer and to a specific domain (in our case backtesting).

5.1 TaskProcessingNode

The TaskProcessingNode is a component that is capable of processing tasks of a specific type and is represented by a WCF service. There is a specific implementation of TaskProcessingNode for each specific task type. The difference between implementations of TaskProcessingNodes is the functionality that processes tasks. It is a domain specific functionality and is contained in a separate assembly. The other functionality is common for all TaskProcessingNodes and handles commu-

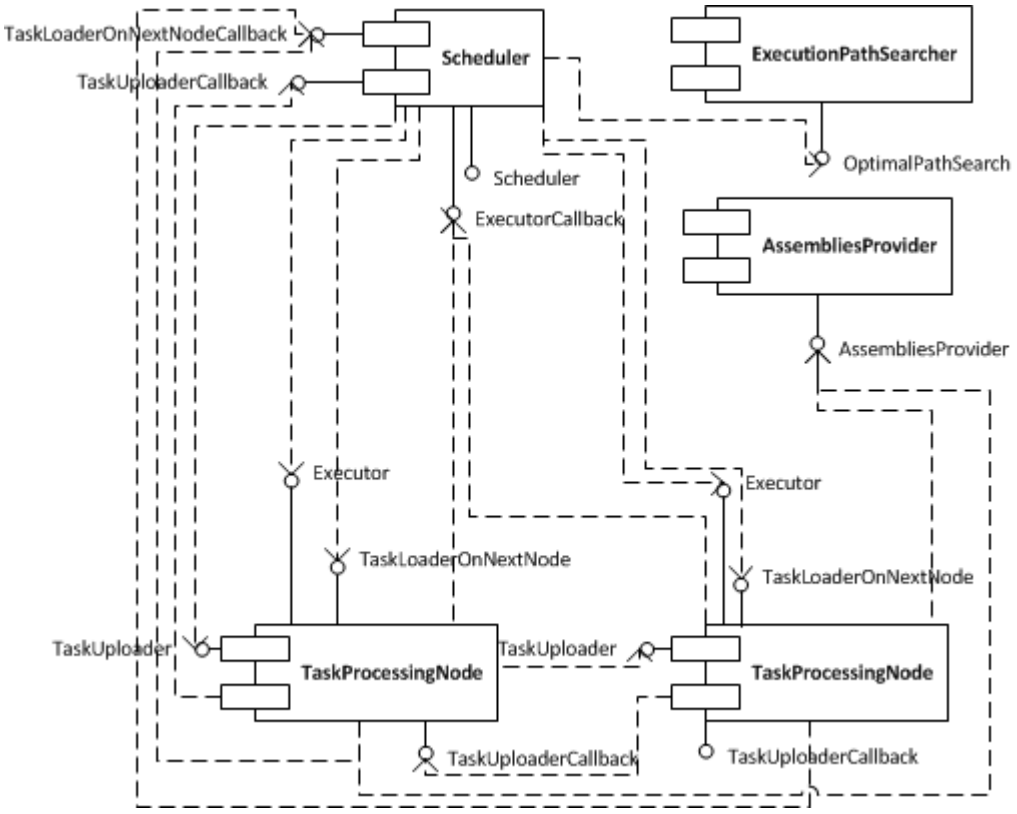

Figure 5.1. System processing component diagram

nication and coordination logic of the layer. The common functionality uses the domain specific functionality when a task is recieved for processing.

The realization of the TaskProcessingNode is explained using the DataProvider service as an example. The Figure 5.2 shows a simplified class diagram of a DataProvider service with packages that represent assemblies[1].

The DataProvider is a console application that hosts DataProvider service, represented by *DataProviderService* class. The DataProvider package references ServiceShared assembly that contains the layer functionality. The *NodeService* class in ServiceShared package contains functionality of TaskProcessingNode to participate in a task distribution process. It also contains means for performance counters usage. The classes that represent WCF services derive from *NodeService* class, thus, *DataProviderService* class is derived from *NodeService* class. The *DataProviderService* class contains functionality that is specific to the task type that DataProvider service processes. It specifies how a task should be processed

[1]Assembly is a .NET concept that represents a collection of types, compiled to a IL language. .NET application is composed from one or more assemblies [15]

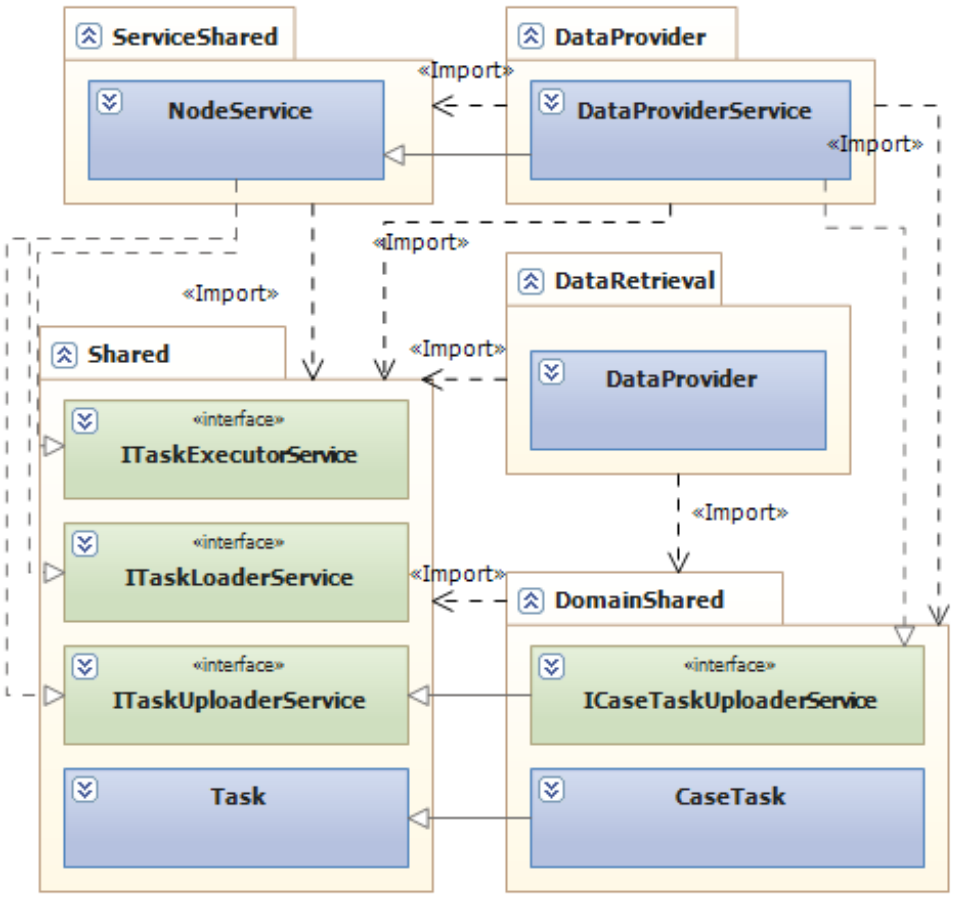

Figure 5.2. Class diagram of the task processing TaskProcessingNode.

when it is received for processing. The functionality that processes tasks and is called by *DataProviderService* class is contained is the separate assembly called DataRetrieval. This assembly contains functionality that evaluates strategies, i.e. represents business domain specific functionality.

The Shared assembly contains the layer functionality for communication between the Scheduler and TaskProcessingNodes. Thus, it is referenced by all components of the system. The interfaces in the Shared assembly represent service contracts that are realized by *NodeService* class.

The DomainShared assembly contains business domain specific functionality that is used for communication between TaskProcessingNodes. The reason is that the data, that is exchanged between TaskProcessingNodes, has domain specific type. The WCF environment needs to be aware of types of objects that are passed between communication parties for serialization and deserialization procedures. Thus,

the DataProvider assembly needs to reference the DomainShared assembly. The $ITaskUploaderService$ service contract has an operation $Upload$ that takes as a parameter an object of $Task$ type:

```
public interface ITaskUploaderService
{
    [OperationContract(IsOneWay = true)]
    void Upload(Task task);
}
```

The $CaseTask$ class is derived from the $Task$ class and represents the data associated with a case in the backtesting process, i.e. data read from historical database and evaluation results. The serialization and deserialization of service operation parameters requires indication of $Task$ successor types in the service contract. In order to support this, the $ICaseTaskUploaderService$ interface derives from $ITaskUploaderService$ interface and specifies that an object of type $CaseTask$ can be passed. The $DataProviderService$ class realizes the $ICaseTaskUploaderService$ interface so that WCF environment can serialize and deserialize data appropriately.

```
[ServiceKnownType(typeof(CaseTask))]
[ServiceContract(CallbackContract = typeof(ITaskUploaderCallback))]
public interface ICaseTaskUploaderService : ITaskUploaderService
{
}
```

TaskProcessingNode is able to process tasks simultaneously. The system has a $L_{LevelOfParallelization}$ level of parallelization parameter which means the amount of tasks that can be processed in parallel. There is a corresponding parameter in TaskProcessingNode, called MaxCpuUsage, that specifies how much CPU time in percents can be used by TaskProcessingNode for processing tasks (more about MaxCpuUsage in Section 5.1.2).

5.1.1 Measuring performance

The performance counters were used for measuring performance of the TaskProcessingNode and the workstation. The $NodeService$ class uses a set of counters, provided by the operating system, for measuring processor and memory usage. The $IPerformanceDataProvider$ interface is located in Shared package and contains methods for getting performance counters and system measurements. The $NodeService$ class realizes the $IPerformanceDataProvider$ interface. As all the classes that represent a TaskProcessingNode derive from the $NodeService$ class, the TaskProcessingNodes are able to provide performance counters and system measurements. The $NodeService$ class defines a virtual method:

```
protected virtual void AddPerformanceData(List<string> processing) { }
```

The *NodeService* successor can override this method to add additional measurements.

5.1.2 Control settings

There are two settings, that can be controlled: *maximum CPU usage for processing tasks* and *outage periods*.

The *maximum CPU usage for processing tasks* setting is used to control the maximum percentage of CPU usage of workstation by TaskProcessingNode. TaskProcessingNode can process tasks in parallel by allocating one thread per task. There is a requirement on processing of tasks, is that it must be executed in a single thread. With this requirement, the number of threads that is used for processing tasks corresponds to the number of tasks being processed. .NET has means for determining the number of logical processors in the workstation - $Environment.ProcessorCount$. The logical processor count determines the maximum number of tasks that can be processed in parallel. Thus, the maximum number of processing tasks can be calculated using the maximum CPU usage:

$$maxTasks = Math.Floor(Environment.ProcessorCount * (maximumCpuLoad/100)) \tag{5.1}$$

If the TaskProcessingNode does not process more than specified maximum number of tasks simultaneously — the CPU usage of the TaskProcessignNode that is spent on processing tasks should not exceed the specified maximum CPU usage. However, as the TaskProcessingNode has other functionality to execute besides processing tasks, the TaskProcessingNode application can use more CPU time than specified maximum limit.

The *outage periods* setting specifies the periods of time when the application doesn't accept tasks for processing. This is achieved by setting the properties of the state of TaskProcessingNode to indicate that the TaskProcessingNode is processing the maximum number of tasks, i.e. the TaskProcessingNode can not accept tasks for processing.

The $ISettingsController$ interface in Table 5.1 is used for setting the settings of TaskProcessingNode.

5.1.3 Managing view of nearby TaskProcessingNodes

The nodes are organized in a directional graph, where the "nearby TaskProcessingNodes of the TaskProcessingNode" are the nodes, to which the TaskProcessingNode has links. The view of nearby nodes is needed in order to keep the up-to-dated measurements of data transferring speed between the nodes. These measurements are used in the calculations of optimal execution path of the tasks that represent backtesting on a case. The $IViewManager$ interface in Table 5.2 is used by the

Name: ISettingsController, **instance** *sc.*
Events: **Request:** ⟨*sc, SetMaximumCpuLoad*⟩: Sets the maximum value for the CPU load of the workstation. **Request:** ⟨*sc, SetOutageInterval* | *from, to*⟩: Sets the outage interval during which, the TaskProcessingNode does not accept tasks for execution.

Table 5.1. ISettingsController interface.

Scheduler in order to request initialization of peer links and by TaskProcessingNode to request estimation of data transfer speed between its nearby nodes.

Name: IViewManager, **instance** *vm.*
Events: **Request:** ⟨*vm, Initialize* | *nodeView*⟩: requests the TaskProcessingNode to initialize the view of nearby nodes. Measures the speed of transferring data for each of the TaskProcessingNode in the view. **Request:** ⟨*vm, EstimateSpeed* | *data*⟩: records the time when the data is received and sends the time back. **Indication:** ⟨*vm, SpeedEstimated* | *receivedTime, nodeId*⟩: indicates the time, when the data was received by the TaskProcessingNode.

Table 5.2. IViewManager interface.

Estimation of data transfer speed is performed in the following way:

1. The TaskProcessingNode creates an array of bytes, records the current time $Tsending$ and sends the array to the nearby TaskProcessingNode.

2. The nearby TaskProcessingNode receives the array and replies with the current time $Treceived$.

3. The TaskProcessingNode receives the reply and calculates the upload data transferring speed:

$$uploadSpeed = dataSize/(Treceived - Tsending) \tag{5.2}$$

Implements: IViewManager, **instance** *vm*.

upon event ⟨*vm, Init*⟩ do
    nodeView = ∅;

upon event ⟨*vm, Initialize* | *nodeView*⟩ do
    forall(node) ∈ nodeView do
        trigger ⟨*node.vm, EstimateSpeed* | *data*⟩;

upon event ⟨*vm, EstimateSpeed* | *data*⟩ do
    receivedTime := time.Now;
    trigger ⟨*vm, SpeedEstimated* | *receivedTime, nodeId*⟩;

upon event ⟨*vm, SpeedEstimated* | *receivedTime, nodeId*⟩ do
    nodeView[nodeId] := CalculateSpeed(receivedTime);

upon nodeView.AllNodesEstimated do
    trigger ⟨*vm, InitializedView* | *nodeId, nodeView*⟩;

Table 5.3. Implementation of the IViewManager interface in Table 5.2.

The realization of the $IViewManager$ interface is presented in Table 5.3.

The requests for speed estimation are sent sequentially in a loop in order not to load the bandwidth heavily.

The estimation of data transfer speed is performed periodically to keep the data up-to-date. The interval of estimation is controlled via configuration file using $UpdateNodesViewInterval$ key.

5.1.4 Uploading a task on a TaskProcessingNode

According to the "Tasks distribution" sequence diagram 3.4, before a task is executed on a TaskProcessingNode the task related data needs to be uploaded on the TaskProcessingNode. The task related data is uploaded from the TaskProcessingNode that executed a task of the case last. The task related data is uploaded using the $ITaskUploader$ interface in Table 5.4:

The $Upload$ operation is a "One-way" operation (see Section 4.4.4) which means that the calling thread continues execution after a message has been put for sending. When the call is being made on the client side, a callback object is specified in order for the recieving side to be able to notify the client. The class of the callback object realizes the $ITaskUploaderCallback$ interface in Table 5.6: Once the task related data is uploaded, the recieving side calls $HandleTaskUploaded$ method which is

Name: ITaskUploader, **instance** *tu*.

Events:
Request: ⟨*tu, Upload | task*⟩: Uploads the task to be processed.

Table 5.4. ITaskUploader interface.

executed in a separate from the main execution thread on the client side. The implementation of the Interface 5.4 is provided in Algorithm 5.5:

Implements: ITaskUploader, **instance** *tu*.

Uses: ITaskUploaderCallback, **instance** *tuc*;

upon event ⟨*tus, Init*⟩ do
uploadedTasks := ∅;

upon event ⟨*tus, Upload | task*⟩ do
if(processingTasks =>maximumTasksInParallel)
trigger ⟨*tus, HandleNoFreeSpot | task.Id, nodeId*⟩;
uploadedTasks := uploadedTasks ∪ task;
trigger ⟨*tus, HandleTaskUploaded | task.Id, nodeId*⟩;

Table 5.5. Implementation of ITaskUploader interface.

If the maximum number of processing tasks is reached at the moment of uploading the task related data on the TaskProcessingNode, then "HandleNoFreeSpot" method is called on the callback object.

Name: ITaskUploaderCallback, **instance** *tuc*.

Events:
Request: ⟨*tuc, HandleTaskUploaded | taskId, nodeId*⟩: handles the event, when a task has been uploaded to a TaskProcessingNode.
Request: ⟨*tuc, HandleNoFreeSpot | taskId, nodeId*⟩: handles the event, when a task has not been uploaded to a TaskProcessingNode.

Table 5.6. ITaskUploaderCallback interface.

5.1.5 Execution of a task

After a task related data is uploaded on a TaskProcessingNode, the task is ready to be executed (see sequence diagram 3.4). The Scheduler uses $IExecutorService$

interface 5.7 to send a request for task execution.

Name: IExecutor, **instance** *e*.
Events:
Request: $\langle e,\ Execute \mid taskId \rangle$: requests execution of the specified task.
Request: $\langle e,\ Stop \mid runId \rangle$: stops processing of tasks, that belong to the specified backtesting run.

Table 5.7. IExecutor interface.

The *Stop* method is used to stop execution of a backtesting run. When the Scheduler stops distribution process it calls *Stop* method on the TaskProcessingNodes. After the *Stop* method was called on a TaskProcessingNode, all the tasks that are received for processing on the TaskProcessingNode or have been processed by the TaskProcessingNode are removed. The tasks that were in process of execution at the moment of receiving of a request for stopping the backtesting run, are still run but the functionality that executes the task can identify that the backtesting run was stopped. The functionality that executes a task is a domain specific functionality which can be realized in a way that execution is stopped or the execution is continued after the request for stopping execution is received.

The *Execute* operation is a "One-way" operation (see Section 4.4.4). When the execution is finished, the Scheduler is notified by the TaskProcessingNode using a callback object of a class that realizes the $IExecutorCallback$ interface in Table 5.9: The implementation of the $IExecutor$ interface in Table 5.7 is provided in Table 5.8.

In case a task has been successfully processed, the TaskProcessingNode calls *ProcessResult* method. The *CantProcess* method is called in two cases: in case by the time of receiving a request for processing a task the maximum amount or processing tasks is reached; in case there was a problem with execution of a task which was not handled by the domain specific functionality.

5.1.6 Loading a task on a next TaskProcessingNode

When the Scheduler is notified that a task has been processed, it searches for a next TaskProcessingNode that is able to process the task of the next type of the job. In case there is no such a node, the job is considered to be completed. In case there is a next TaskProcessingNode, the Scheduler uses $ITaskLoaderOnNextNode$ interface in Table 5.10 to send a request to the TaskProcessingNode, that processed the task, to upload the task related data on the next TaskProcessingNode.

Implements: IExecutor, **instance** *e*.

Uses: IExecutorCallback, **instance** *ec*;

upon event ⟨*e, Init*⟩ do
    tasksInProcess := ∅;
    tasksToProcess := ∅;

upon event ⟨*e, Execute* | *taskId*⟩ do
    if(tasksInProcess.Count == maxTasksProcessCount)
        trigger ⟨*ec, CantProcess* | *taskId, nodeId*⟩;
    tasksToProcess := tasksToProcess ∪ {taskId};

upon tasksToProcess != ∅ do
    taskId := selectTask(tasksToProcess);
    tasksToProcess := tasksToProcess \{taskId};
    tasksInProcess := tasksInProcess ∪ {taskId};
    Execute(taskId);
    trigger ⟨*es, TaskProcessed* | *taskId, nodeId*⟩;
    tasksInProcess := tasksInProcess \{taskId};

Table 5.8. Implementation of IExecutor interface.

Name: IExecutorCallback, **instance** *ec*.

Events:

Request: ⟨*ec, ProcessResult* | *taskId, nodeId*⟩: processes the result of execution of a task on a TaskProcessingNode.

Request: ⟨*ec, CantProcess* | *taskId, nodeId*⟩: handles the event, when a task can not be processed by a TaskProcessingNode.

Table 5.9. IExecutorCallback interface.

Name: ITaskLoaderOnNextNode, **instance** *tlonn*.

Events:

Request: ⟨*tlonn, Load* | *taskId, nodeId*⟩: requests TaskProcessingNode to load the specified task to the specified TaskProcessingNode.

Table 5.10. ITaskLoaderOnNextNode interface.

The implementation of the ITaskLoaderOnNextNode interface is provided in Table 5.11. In order to load task related data on a TaskProcessingNode, the TaskProcessingNode that received "Load" request uses $ITaskUploader$ interface (see Section 5.1.4).

The *Load* operation is a "One-way" operation (see Section 4.4.4). When the Scheduler sends a *Load* request it specifies a callback object of a class that realizes $ITaskLoaderOnNextNodeCallback$ interface in Table 5.12. The TaskProcessingNode uses this callback object to notify the Scheduler that the task has been uploaded on the next TaskProcessingNode.

Implements: ITaskLoaderOnNextNode, **instance** *tlonn*.

Uses: ITaskLoaderOnNextNodeCallback, **instance** *tlonnc*;
TaskUploader, **instance** *tu*;

upon event ⟨*tlonn, Load* | *taskId, nodeId*⟩ do
node := select(nodeId);
trigger ⟨*node.tu, Upload* | *taskId*⟩;

upon event ⟨*tu, ConfirmUpload* | *taskId, nodeId*⟩ do
trigger ⟨*tlonnc, ConfirmLoad* | *taskId, nodeId*⟩;

upon event ⟨*tu, HandleNoFreeSpot* | *taskId, nodeId*⟩ do
trigger ⟨*tlonnc, HandleNoFreeSpot* | *taskId, nodeId*⟩;

Table 5.11. Implementation of ITaskLoaderOnNextNode interface.

When the TaskProcessingNode is notified that the task related data was uploaded, the TaskProcessingNode calls $HandleTaskUploaded$ method of the callback object which is executed in a separate from the main execution thread on the Scheduler. If the maximum number of processing tasks has been reached at the moment of uploading a task related data on the TaskProcessingNode, then "HandleNoFreeSpot" method is called on the callback object.

5.2 Scheduler

The Scheduler component is represented by a WCF service. A simplified class diagram of the Scheduler component is presented in Figure 5.3.

The Scheduler is a console application that hosts Scheduler service, represented by a $SchedulerService$ class. The $SchedulerService$ class realizes the $ISchedulerService$ interface that contains methods for controling a distribution of workload process.

Name: ITaskLoaderOnNextNodeCallback, **instance** *tlonnc*.
Events: **Request:** ⟨*ec, HandleTaskLoaded* | *taskId, nodeId*⟩: handles the event, when a task has been loaded on a TaskProcessingNode. **Request:** ⟨*ec, HandleNoFreeSpot* | *taskId, nodeId*⟩: handles the event, when a task can not be loaded to a TaskProcessingNode.

Table 5.12. ITaskLoaderOnNextNodeCallback interface.

The *SchedulerService* class derives from the *NodeService* class which means that the Scheduler service is also a TaskProcessingNode. The TaskProcessingNode of the Scheduler is always the first TaskProcessingNode on the execution path for all tasks. This TaskProcessingNode does not have a limit of tasks for processing. When the Scheduler receives tasks for distribution, it saves them on the Scheduler TaskProcessingNode and marks them as processed.

The Scheduling assembly contains functionality that handles scheduling of tasks for execution on TaskProcessingNodes, coordination and communication between the TaskProcessingNodes. The *Scheduler* class is the realization of Mediator pattern as discussed in Section 3.3.

Name: Scheduler, **instance** *s*.
Events: **Request:** ⟨*s, Schedule* | *tasks*⟩: requests scheduling of the tasks. **Request:** ⟨*s, Initialize* | *nodesAddresses*⟩: requests initialization of the layer with specified nodes addresses. **Request:** ⟨*s, Stop* | *id*⟩: requests stopping of scheduling. **Request:** ⟨*s, AddNode* | *collectionId, address*⟩: requests addition of the node.

Table 5.13. Interface: Scheduler.

5.3 ExecutionPathSearcher

The ExecutionPathSearcher component is implemented as an assembly that exports a class, that allows to:

- Get the optimal path in a graph based on speed of transferring data between the vertices.

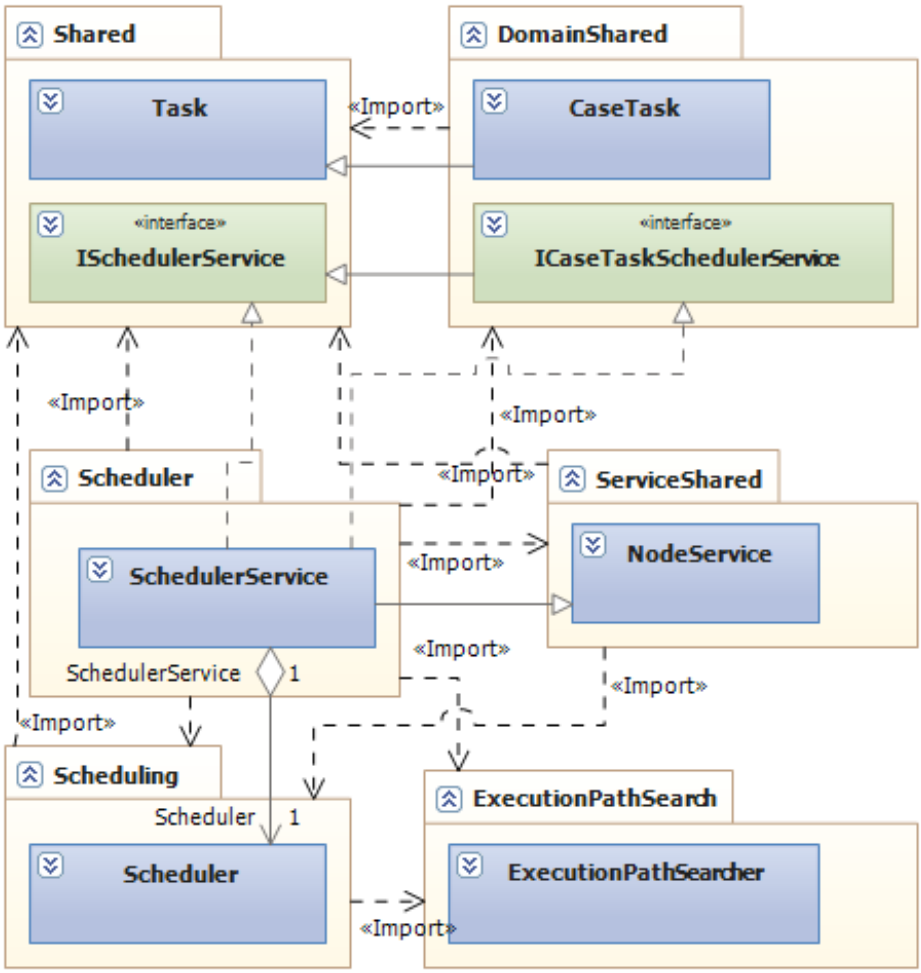

Figure 5.3. Class diagram of the scheduler component.

- Get the optimal path from the specified vertex in a graph based on speed of transferring data between the vertices.

The algorithm for searching for an optimal path is the following:

1. Find all the possible paths starting either from a specific vertex or from the first vertexes.

2. All TaskProcessingNodes estimate the average speed of transferring data to the nodes that they have links to. The speed of transferring data is used as a cost value of each edge in the graph.

3. The paths are sorted by the amount of nodes that don't have a free spot for processing a task at the moment.

4. The paths with minimum amount of busy nodes are sorted by the cost value.

5. The first path in the list is the optimal path.

The *IOptimalPathSearch* interface of the ExecutionPathSearcher component is provided in the Table 5.14.

Name: IOptimalPathSearch, **instance** *ops*.
Events: **Request:** ⟨*ops, GetOptimalPath* | *graph*⟩: searches for an optimal path in the directional graph. **Request:** ⟨*ops, GetOptimalPath* | *current, graph*⟩: searches for an optimal path from the specified vertex in the directional graph.

Table 5.14. IOptimalPathSearch interface.

5.4 AssembliesProvider

The AssembliesProvider component is implemented as a WCF service. In order to transfer an assembly, the AssembliesProvider reads the assembly file into array of bytes. The requesting application uses Reflection[2] to load the assembly into application domain:

```
Assembly.Load(bytes);
```

The Reflection is also used to find a specific type in the assembly to, for example, execute a task.

```
foreach (Type type in m_Assembly.GetTypes())
{
          if (type.Name == typeName)
          {
                    executionType = type;
                    break;
          }
}
```

5.5 Code metrics

Code metrics are the metrics that developers use to analyze some general properties of code. The prototype was developed using Visual Studio 2012 which has a built in

[2]Reflection is a .NET technology that provides an ability to access information regarding assembly, types in the assembly, members in the types, instantiate an object of the type, that is chosen dynamically etc.

support for measuring code metrics. The following code metrics are used in Visual Studio 2012: maintainability index, cyclomatic complexity, depth of inheritance and class coupling. The maintainability index ia a metric that shows how easy it is to maintain the code [18]. The value of the index is between 0 and 100, where the higher the value the better is maintainability. The cyclomatic complexity measures complexity of code by calculating possible code paths [18]. The class coupling indicates a level of coupling of types between each other [18]. The lines of source code is a metric that means count of lines of code in the source files.

The Table 5.15 presents the code metrics for the prototype.

Assembly name	**Maintainability Index**	**Cyclomatic Complexity**	**Depth of Inheritance**	**Class Coupling**	**Lines of Source Code**
DataProvider	74	8	2	26	107
ResultsSaver	74	11	2	28	117
AssembliesProvider	64	9	1	20	116
ResultsSaving	60	12	1	25	178
Scheduler	80	16	2	25	122
StrategyEvaluator	81	27	2	34	229
NodeClient	78	34	9	43	224
SchedulerPathSearch	79	113	1	25	575
DomainShared	90	217	2	74	596
StrategyEvaluation	67	100	2	151	917
Shared	91	231	2	53	1310
Client	84	188	9	123	1189
ServiceShared	81	176	2	85	1233
Scheduling	74	277	2	69	1576
DataRetrieval	75	558	2	108	384

Table 5.15. Code metrics.

The prototype has a good maintainability index and an acceptable complexity. The depth of inheritance equals to two on average and the class coupling is acceptable.

Chapter 6

Evaluation

This section describes evaluation of the prototype. In order to evaluate the prototype, hardware performance and throughput was measured in the same way as for the system in Section 2.1.3.

The process of evaluation is divided into two parts. In the first part, it was evaluated if there was a performance overhead for having an additional layer in the prototype that handles distribution of tasks logic. For this purpose, the prototype was evaluated on the same hardware and using the same configuration files that were used for evaluation of the system. The second part was to determine a performance benefit of running a backtesting process on the prototype in comparison to running it on the system. The throughput was measured in different hardware configurations with an increasing number of TaskProcessingNodes.

The difference between performance measurement of the backtesting process on the prototype and on the system is that the prototype includes minimum three computational nodes (one per each task type) that are represented by separate processes that can be hosted on different workstationa whereas the system is represented by one process. The technical resources consumption was measured for all three TaskProcessingNodes in the first part of evaluation. In the second part, the technical capacity was measured for TaskProcessingNodes that provide data and save the evaluation results, but not for the TaskProcessingNodes that evaluate strategies. The reason for this is that evaluation of strategies was the step that was run on several servers simultaneously.

The goal of the evaluation process was to determine if the prototype satisfies the functional and nonfunctional requirements presented in Section 1.3.

6.1 Evaluation of performance overhead

In order to evaluate a performance overhead for having an additional layer in the system that handles distribution of tasks logic, the backtesting process was run and evaluated using the prototype and the same technical capacity as in Section 2.1.3. The prototype was represented by one DataProvider, one StrategyEvaluator and one ResultsSaver services. The MaxCpuUsage parameter (see Section 5.1.2) was used to specify the number of cases that was processed simultaneously, i.e. to configure the prototype for a specific $L_{LevelOfParallelization}$ parameter.

6.1.1 Minimum configuration

The maximum throughput of the prototype with minimum configuration was 3.8 cases per minute with $L_{LevelOfParallelization}$=40. The Figure 6.1 shows a comparison of performance counters measurements of backtesting runs on the system and the prototype with minimum configuration.

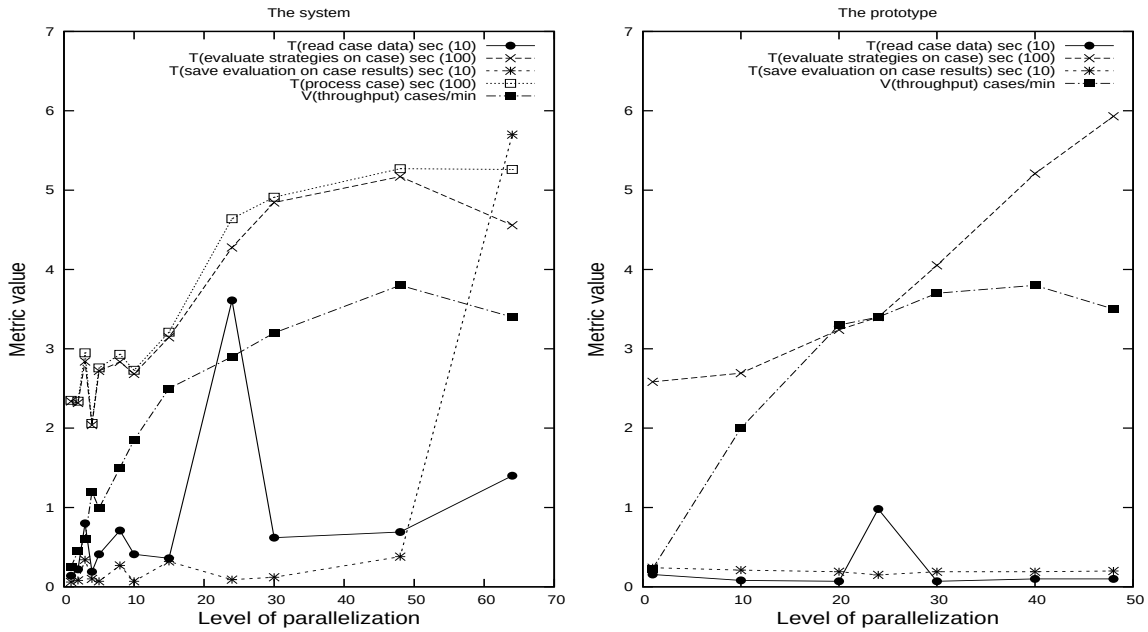

Figure 6.1. Comparison of performance counters measurements of backtesting run with a minimum configuration between the system and the prototype.

The Figure 6.2 shows the comparison between performance counters measurements of hardware usage by the system and StrategyEvaluator during backtesting runs with minimum configuration.

The Figure 6.3 shows performance counters measurements of hardware usage by DataProvider and ResultsSaver during backtesting run with minimum configuration.

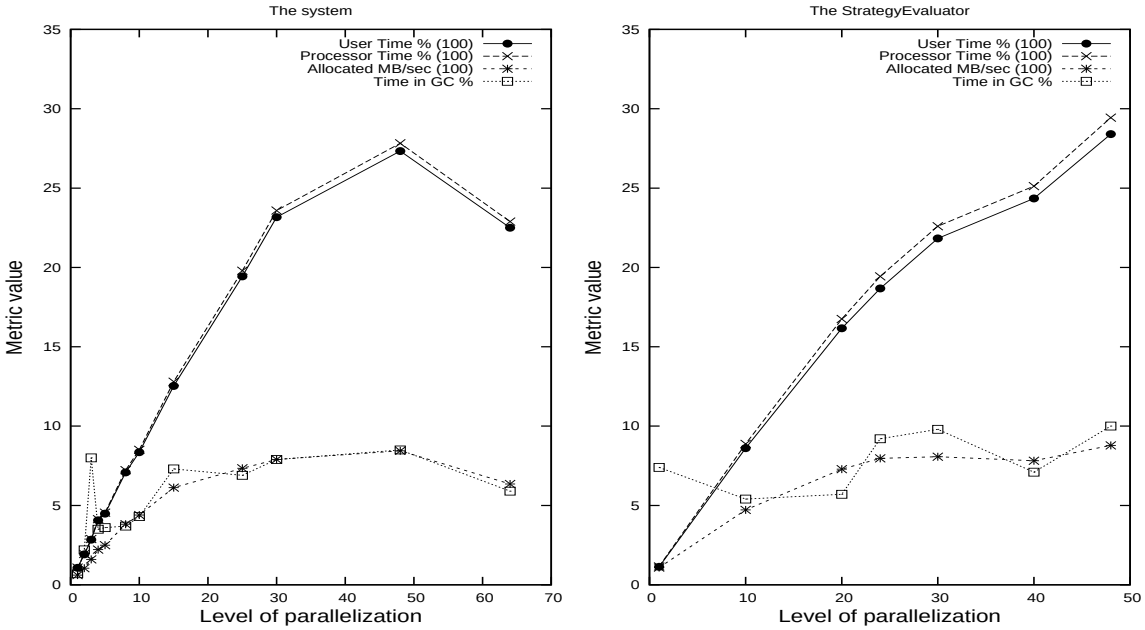

Figure 6.2. The comparison between performance counters measurements of hardware usage by the system and StrategyEvaluator during backtesting runs with minimum configuration.

The throughput 3.8 cases/min was reached with a lower level of parallelization when using the prototype than when using the system. The prototype uses .NET 4.5 that has an updated garbage collector that performs faster than garbage collector in .NET 4.0 that was used by the system. The difference in performance between the prototype and the system was explained by lower time spent on garbage collection (see Figure 6.2).

6.1.2 Maximum configuration

The maximum throughput of the prototype with the maximum configuration was 0.5 cases per minute with $L_{LevelOfParallelization}$=15. The Figure 6.4 shows a comparison of performance counters measurements of backtesting runs on the system and the prototype with maximum configuration.

The Figure 6.5 shows a comparison between performance counters measurements of hardware usage by the system and StrategyEvaluator during backtesting runs with maximum configuration.

The Figure 6.6 shows the performance counters measurements of hardware usage by DataProvider and ResultsSaver during backtesting run with maximum configuration.

The throughput 0.5 cases/min was reached with a lower level of parallelization when using the prototype than when using the system. The difference in perfor-

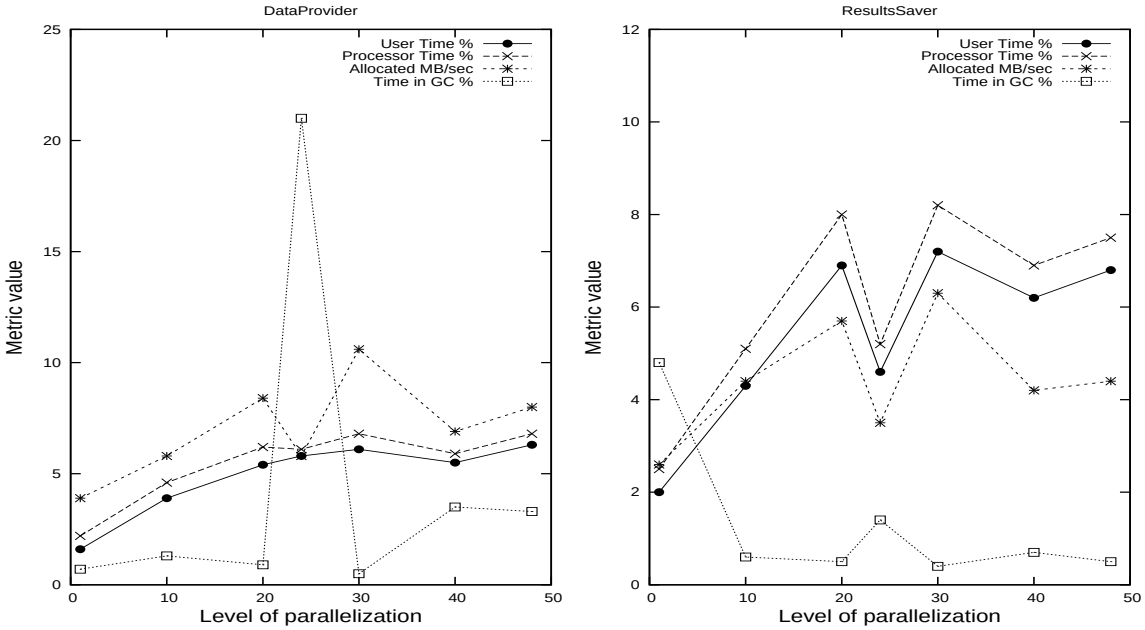

Figure 6.3. The performance counters measurements of hardware usage by DataProvider and ResultsSaver during backtesting run with minimum configuration.

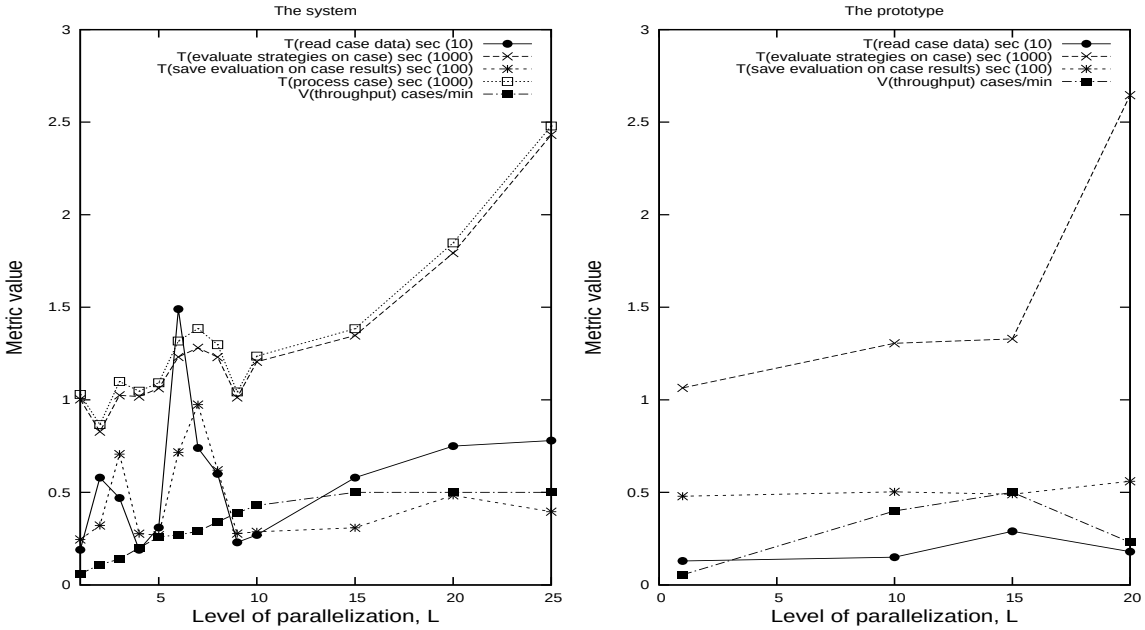

Figure 6.4. The comparison of performance counters measurements of backtesting runs with maximum configuration between the system and the prototype.

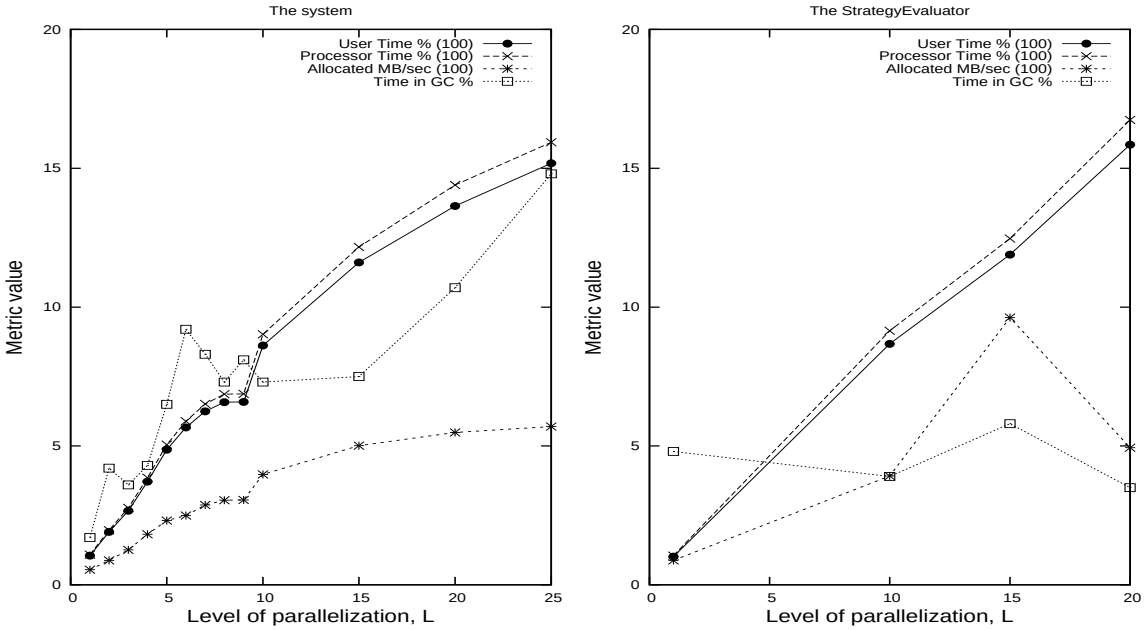

Figure 6.5. The comparison between performance counters measurements of hardware usage by the system and StrategyEvaluator during backtesting runs with maximum configuration.

mance between the prototype and the system was explained by lower time spent on garbage collection (see Figure 6.5).

6.2 Analysis of performance benefit

In this section it was analyzed if the usage of the prototype significantly reduces duration of backtesting process by distributing the workload among workstations, i.e. if it satisfies the goal of the thesis (see 1.5).

It was investigated that the bottleneck in the system was evaluation of strategies for both configurations, see Section 2. Based on the calculations in Section 2.3.2, the expected throughput of the system with distributed evaluation of strategies was 8.6 cases/min for minimum configuration and 1.89 cases/min for maximum configuration.

The available technical capacity (see Section 2.3.1) in the company was used for evaluation of strategies. The usage of the hardware that was used for evaluation of strategies was not analyzed.

There was an assumption that on a multiprocessor workstation it could be ben-

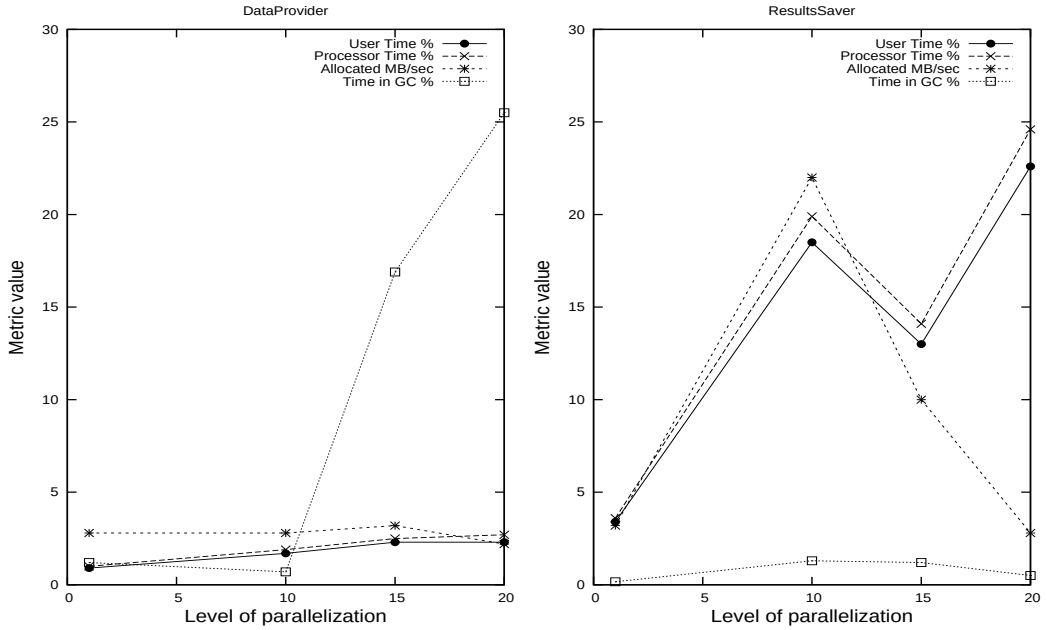

Figure 6.6. The performance counters measurements of hardware usage by DataProvider and ResultsSaver during backtesting run with maximum configuration.

eficial to run several instances of TaskProcessingNodes instead of single instance. The tests showed that there is a 29% increase in throughput for evaluation of strategies with minimum configuration, when four instances of StrategyEvaluator with $L_{LevelOfParallelization}$=10 were run instead of one instance of StrategyEvaluator with $L_{LevelOfParallelization}$=40. It was decided to use several instances of StrategyEvaluator on each of the multiprocessor machines.

As there were more than one workstation involved in the evaluation of strategies, the usage of $L_{LevelOfParallelization}$ parameter for identifying the technical resources consumption was changed to the usage of C parameter, which is the number of physical cores involved in the process.

The $MaxCpuUsage$ parameter was used to configure the prototype for a specific C parameter.

6.2.1 Minimum configuration

The maximum throughput of the prototype with the minimum configuration was identified to be 9.3 cases per minute with C=52. The Figure 6.7 shows the comparison of performance counters measurements of backtesting runs on the system and the prototype with minimum configuration. The throughput of the prototype increases almost linearly with increasing technical capacity. The maximum

throughput of the prototype was 2.4 times higher than the maximum throughput of the system.

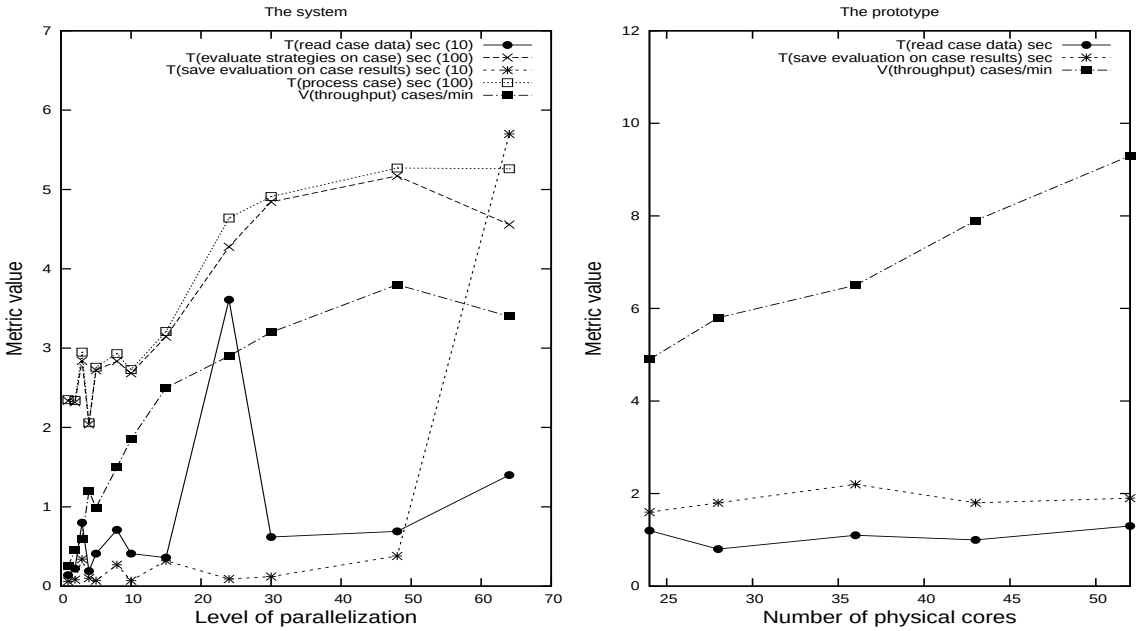

Figure 6.7. Comparison of performance counters measurements of backtesting process with a minimum configuration between the system and the prototype.

The Figure 6.8 shows the performance counters measurements of hardware usage by DataProvider and ResultsSaver during backtesting process with minimum configuration. The measurements show that the components have enough technical capacity for running in different setups.

6.2.2 Maximum configuration

The maximum throughput of the prototype with the maximum configuration was identified to be 1.51 cases per minute with C=28. The Figure 6.9 shows the comparison of performance counters measurements of backtesting process with maximum configuration between the system and the prototype. The throughput of the prototype increases almost linearly with increasing technical capacity. The maximum throughput of the prototype was 3 times higher than the maximum throughput of the system.

The Figure 6.10 shows the performance counters measurements of hardware usage by DataProvider and ResultsSaver during backtesting process with maximum configuration. The measurements show that the components have enough technical capacity for running in different setups.

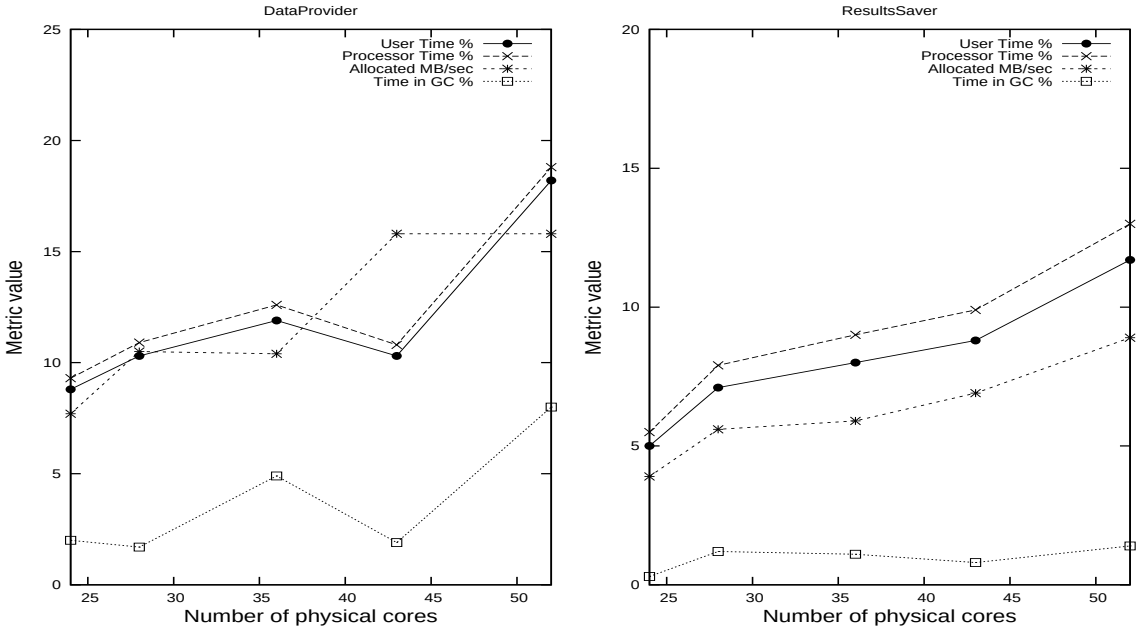

Figure 6.8. The performance counters measurements of hardware usage by DataProvider and ResultsSaver during backtesting process with minimum configuration.

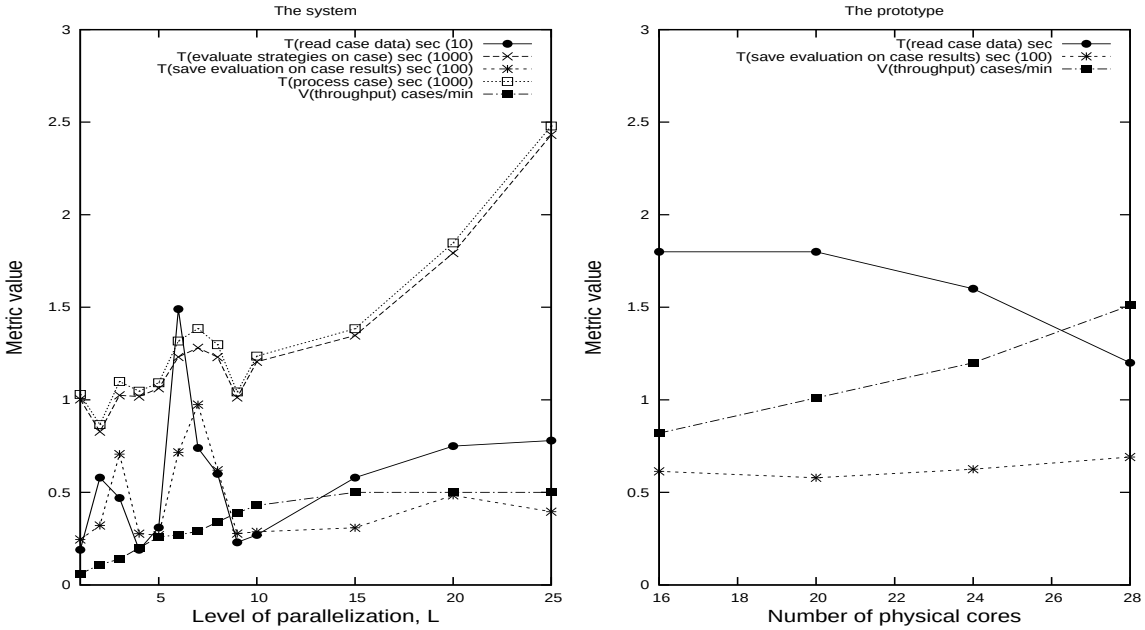

Figure 6.9. The comparison of performance counters measurements of backtesting process with maximum configuration between the system and the prototype.

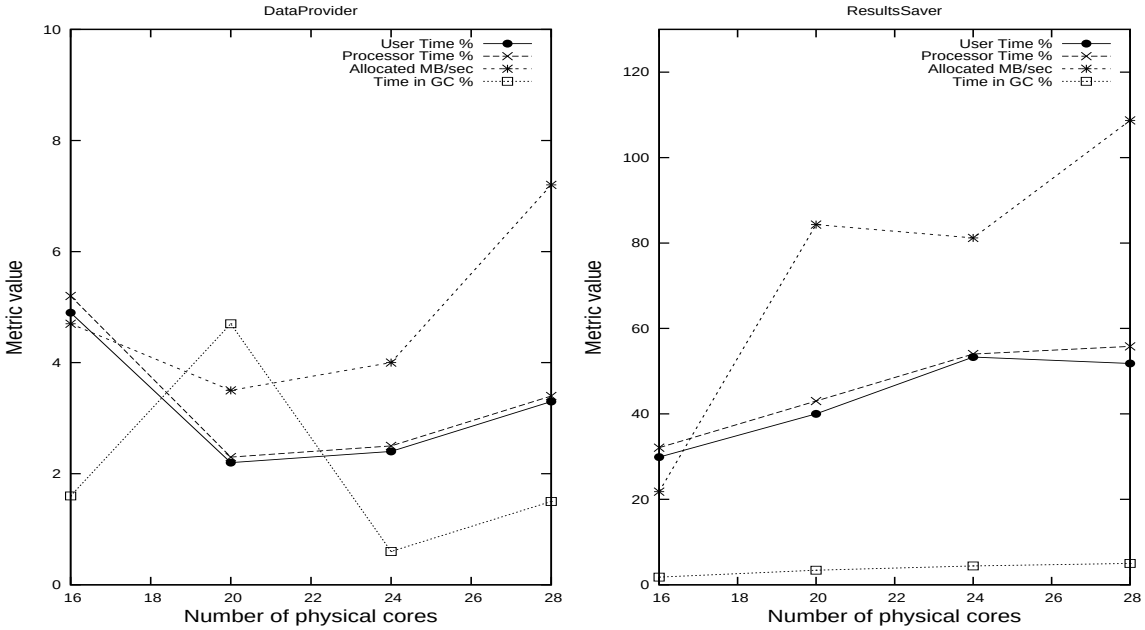

Figure 6.10. The performance counters measurements of hardware usage by DataProvider and ResultsSaver during backtesting process with maximum configuration.

6.3 Verification of the requirements

The functional and non-functional requirements were verified in the prototype.

6.3.1 Functional requirements

By ”user“ below is meant a person who initiated the backtesting process:

- ”The system accepts a configuration file that represents a set of cases, strategies to test and evaluators and puts it for execution“. Implemented.
- ”The system distributes workload among available workstations“. Implemented.
- ”The system reports an overall progress“. Implemented.
- ”The system tolerates the failures of nodes“. Implemented.
- ”New workstations can be added to the system without restarting the system and test runs“. Implemented.
- "The level of load of workstations can be controlled either by a user or an owner of machine“. Implemented.

- ”The user can abort the execution of a test run“. Implemented, but the TaskProcessingNodes need to finish processing of the current tasks.
- ”The user can change the priority of a job that is waiting to be executed“. Not implemented.
- ”The user can provide time constraints for workstations that define the time during which the jobs can be run“. Implemented.
- ”The system logs execution of tests, preferably in a single log file for each test run/user“. Implemented, but each TaskProcessingNode logs to its own local log file.

6.3.2 Nonfunctional requirements

Performance. ”In this project performance means amount of cases evaluated in a certain period of time. The system decreases significantly the duration of the backtesting process in comparison to the current system depending on the number of workstations involved in a process.“ Implemented.

Scalability. ”The duration of backtesting run decreases nearly proportionally with the increasing technical capacity. Due to the fact that the evaluation of case is a separate process and the system has a distributed behavior“. Implemented.

”The bandwidth usage between workstations is not increased considerably and thus does not impact the production activities. The system takes into consideration the topology of the network“:

- ”The kinds of jobs that it puts for execution may need different type of data that is located on one of the workstations“. Implemented.
- ”The geographical distance between the workstations - the cost of transferring a job and the size of a job are taken into account.“ Implemented another solution, i.e. the speed of transferring data is taken into account.

6.4 Conclusions

The evaluation of the prototype showed that the additional layer, that distributes the workload, doesn't introduce a significant performance overhead.

Based on the calculations in Section 2.3.2, the expected throughput of the system with distributed evaluation of strategies was 8.6 cases/min for minimum configuration and 1.89 cases/min for maximum configuration. The throughput of the prototype was measured to be 9.3 cases/min for minimum configuration and 1.51 cases/min for maximum configuration.

The majority of the requirements were satisfied. The requirements that were not satisfied could be realized without major changes to the prototype.

Chapter 7

Conclusions and future work

7.1 Conclusion

The thesis presents a problem of long time execution of backtesting process and proposes a solution to reduce the execution time by distributing the workload among available workstations. The system, that was used for backtesting in the company, had a maximum throughput of 3.8 cases/min and 0.5 cases/min for minimum and maximum configurations respectively. The investigation of performance benefit of distributed approach showed that the maximum throughput of the distributed version of the system was approximately 8.6 cases/min and 1.89 cases/min for minimum and maximum configurations respectively using available techincal capacity in the company. The thesis presentes an architecture and a prototype of the layer that can be used by the backtesting system for distribution of workload. The evaluation results showed that the throughput of the prototype was 9.3 cases/min and 1.51 cases/min for minimum and maximum configurations respectively. Thus, the goal of the thesis, to significantly reduce duration of the backtesting process, was achieved by using the prototype of the system that used the distribution of workload layer.

During the design phase there was a choice either to add functionality to the system that would handle distribution of workload logic or to develop a separate layer that the system would use. The choice was made in favour to the separate layer, that was tested during development using a separate test application. The result of the development is a non business domain specific prototype of the layer that distributes the workload among available workstations.

It was a challenge to decide what functionality should be handled in a generic way as usually it was less effective from performance perspective.

7.2 Future work

During the evaluation part, there was an idea to implement functionality that would track the perfomance of the parts of the system and calculate a potential bottleneck in the system. It would allow user to undertand what technical capacity should be added to the system in order to increase throughput.

The data, that is read from historical database, is kept in objects of data transfer classes. It would be more efficient to transfer a binary representation of data that is read directly from database.

The next step would be evaluation of the prototype using more TaskProcessingNodes. By increasing the load on the system we will gain more data regarding fault-tolerance and scalability of the system.

Bibliography

[1] "Task Parallel Library (TPL)", date accessed: July 4, 2013, http://msdn.microsoft.com/en-us/library/dd460717.aspx

[2] "Performance Counters", date accessed: July 4, 2013, http://msdn.microsoft.com/en-us/library/windows/desktop/aa373083(v=vs.85).aspx

[3] "Windows PowerShell", date accessed: July 4, 2013, http://technet.microsoft.com/en-us/library/bb978526.aspx

[4] "Memory Performance Counters", date accessed: July 4, 2013, http://msdn.microsoft.com/en-us/library/x2tyfybc.aspx

[5] "Performance Monitor Counters", date accessed: July 4, 2013, http://technet.microsoft.com/en-us/library/cc768048.aspx

[6] "Welcome to Apache™Hadoop", date accessed: July 4, 2013, http://hadoop.apache.org/

[7] "qizmt - MySpace Qizmt - MySpace's Open Source Mapreduce Framework - Google Project Hosting", date accessed: July 4, 2013, http://code.google.com/p/qizmt/

[8] "Design Patterns"', by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Addison-Wesley 1994.

[9] "Learning WCF", Michele Leroux Bustamante, O'Reilly Media.

[10] ".NET Remoting Overview", date accessed: July 4, 2013, http://msdn.microsoft.com/en-us/library/kwdt6w2k(v=vs.71).aspx

[11] "Web Services with ASP.NET", date accessed: July 4, 2013, http://msdn.microsoft.com/en-us/library/ms972326.aspx

[12] "Message Queuing (MSMQ)", date accessed: July 4, 2013, http://msdn.microsoft.com/en-us/library/windows/desktop/ms711472(v=vs.85).aspx

[13] "Programming WCF Services", Juval Löwy, O'Reilly Media.

[14] ”Reference Model for Service Oriented Architecture 1.0“, OASIS Standard, 12 October 2006.

[15] ”Assemblies“, date accessed: July 4, 2013, http://msdn.microsoft.com/en-us/library/hk5f40ct(v=vs.71).aspx

[16] ”Hyper-Threading“, date accessed: July 4, 2013, http://msdn.microsoft.com/ru-ru/library/dd335944.aspx

[17] ”Intel®Hyper-Threading Technology“, date accessed: July 4, 2013, http://www.intel.co.uk/content/www/uk/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html

[18] ”Code Metrics Values“, date accessed: July 4, 2013, http://msdn.microsoft.com/en-us/library/bb385914.aspx

TRITA-ICT-EX-2013:193

www.kth.se