SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2012; **42**:131–157
Published online 27 January 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1054

Satisfying quality requirements in the design of a partition-based, distributed stock trading system

Xiaohu Yang[1], Liping Zhao[2], Xinyu Wang[1,*,†], Ye Wang[1], Jie Sun[1]
and Albert Jerry Cristoforo[3]

[1]*College of Computer Science, Zhejiang University, Hangzhou, People's Republic of China*
[2]*School of Computer Science, University of Manchester, Manchester, U.K.*
[3]*State Street Corporation, Boston, MA, U.S.A.*

SUMMARY

Although quality requirements (QRs) have become a major drive in today's software development, there have been very few real-world examples in the literature that demonstrate how to meet these requirements. This paper presents such an example. Specifically, the paper describes the design of a partition-based distributed stock trading service system that satisfies a set of QRs related to resource utilization, performance, scalability and availability. The paper evaluates this design through detailed experiments and discusses some design alternatives and the lessons learned. Central to this design are a static load distribution strategy and a dynamic load balancing strategy. The first strategy is to achieve an initial balanced workload on the system's server cluster during the system initialization time, whereas the second strategy is to maintain this balanced workload throughout the system execution time. Together, these two strategies work in unison to ensure that the server resources are efficiently utilized; the user requests are processed with the required speed; the application is partitioned with sufficient room to scale; and the system is highly available. Copyright © 2011 John Wiley & Sons, Ltd.

Received 30 March 2010; Revised 6 December 2010; Accepted 13 December 2010

KEY WORDS: distributed systems; partition-based server clusters; quality-of-service; quality requirements; software design; static load distribution and dynamic load balancing strategies; stock trading systems

1. INTRODUCTION

A major challenge in the design of service systems is to satisfy their quality requirements (QRs) [1]. Several characteristics underlie this challenge: First, QRs are system-level requirements which cannot be assigned directly to individual system components; instead, they need to be planned at the infrastructure-level as a whole, with their design aspects then entrusted to system components. Second, QRs are often interconnected and their realization in a system requires a collective and coordinated behavior of the system components and a system-level design strategy. Third, QRs are application-specific and their fulfillment necessarily requires a specific design approach germane to their applications. Finally, QRs are not only a design time concern, but most crucially a runtime concern. Satisfying QRs in a system means designing runtime mechanisms that can maintain the system's QRs throughout the execution time.

*Correspondence to: Xinyu Wang, College of Computer Science, Zhejiang University, Hangzhou 310027, People's Republic of China.
†E-mail: wangxinyu@zju.edu.cn

We have learned about these characteristics of QRs from undertaking a re-engineering project for a US-based multinational financial service company—referred to as FSC in this paper. The purpose of that project was to transform a standalone stock trading system into a distributed system, called GETS (Global Equity Trading System), under a set of business and QRs. These requirements are described as follows.

As a stock trading system, GETS's main function is to facilitate its users to sell or buy shares online. GETS's users are traders and operators. Traders are investors who would buy or sell shares for their institutions or themselves, whereas operators are FSC's employees who would use GETS to process shares. GETS is required to provide five essential trading services for its users, which are Order Entry, Order Matching, Trade Management, Order Pricing and Trade Printing. GETS is also required to access the trading data from an existing centralized relational trading database through a set of APIs. GETS's users would typically interact with GETS in the following manner:

(1) The trader would use the Order Entry service to submit his trading data in the form of orders. GETS would then validate the trading orders and send the confirmation to the trader. The validated orders would be placed in order books stored in the trading database to wait for Order Matching. An order book is a record of un-traded orders with the same equity symbol. An equity symbol is a unique identifier for a group of publicly traded shares of a specific company on a particular equity market. GETS should provide one order book for each equity symbol.
(2) Upon receiving the order books, the operator would then apply the Order Matching service to match buy-side orders with suitable sell-side ones according to their equity symbols. A buy-side order is an order to buy some shares of a certain equity symbol, whereas a sell-side order is an order to sell some shares of a certain equity symbol. Once the orders are matched, GETS should find the execution prices for the matched orders through the Order Pricing service and convert the orders into the trades.
(3) The operator would then use the Trade Printing service to check whether the trades are in line with the financial regulations and to publish the trades to the public.
(4) To complete the trading process, the operator would use the Trade Management service to update the trades and trading accounts in the trading database.

In addition to the above business requirements, GETS is required to satisfy the following four QRs[‡].

- *Resource utilization*: GETS should use its computational resources (e.g. disk spaces and processing times) efficiently. As this paper will show, this QR is most important to GETS, as it has a direct impact on other QRs.
- *Performance*: GETS's performance is measured by its throughput, which is the number of the orders that can be matched in one second. GETS is expected to produce at least 300 matched orders per second.
- *Scalability*: GETS's processing capacity should be expandable and the expansion should not affect its performance.
- *Availability*: GETS should ideally maintain a high availability (HA) equivalent to 99.99% of its operation time.

These QRs, as stated in [2], represent a set of key quality-of-service (QoS) requirements for almost all online stock trading systems. In contrast to GETS's business requirements which define 'what' services GETS should provide, these QRs delineate 'how well' GETS should perform these services. They are therefore the ultimate design goals for GETS.

Since its development, GETS has been successful in operation and has played a vital role in its company's core business. Today, GETS supports equity trading in the U.S.A., Europe and Australia, with an average daily trading volume exceeding 200 million shares and an average daily

[‡]Our key QRs do not include security as we have opted to use standard, off-the-shelf security software for GETS.

turnover of more than 10 billion USD. GETS's business success indicates that its design has stood the test of time. The purpose of this paper is to present this design and in particular to show how the design meets the above four QRs.

The remaining paper is presented as follows: Section 2 introduces GETS's development platform and design tasks. Sections 3 and 4 present GETS's design tasks in detail. Section 5 evaluates this design through detailed experiments, whereas Section 6 discusses some design alternatives and lessons learned. Section 7 compares and contrasts our design with closely related work and finally, Section 8 concludes the paper by summarizing the key contributions of this design work.

2. OVERVIEW OF GETS'S DEVELOPMENT PLATFORM AND DESIGN TASKS

This section provides a brief introduction to GETS's development platform and an overview to GETS's design tasks.

2.1. GETS's development platform

GETS's development is based on a cluster of WebSphere Partitioning Facility (WPF) enabled servers [3, 4]. The salient feature of a WPF-enabled server is its support for the concurrent execution of multiple independent processing units called 'service partitions' (SPs). Exactly how many SPs can be executed at the same time depends on the number of the computation threads possessed by the server. The server provides a scheduling and synchronization mechanism to control the execution of its SPs. This feature is particularly attractive to us because we can divide GETS into a set of independent SPs, distribute these SPs across a cluster of servers and then execute concurrently. On each WPF-enabled server cluster, there is a load balancing mechanism in place to ensure a balanced workload on the entire cluster at runtime. This feature is also attractive as GETS's runtime workload on each server can change dynamically due to the client requests.

In addition, each WPF-enabled server cluster has an HA device which monitors each server's conditions periodically at runtime to ensure that failures are detected as early as possible and the jobs on the failed servers are transferred into the healthy ones promptly. This feature is clearly relevant to GETS's availability QR. Finally, a WPF-enabled server provides a runtime data buffer. This means that the application data can be preloaded from the database into the server's data buffer to avoid runtime database access (DA) operations. This feature can therefore increase the data processing speed, which is very important to GETS, because, as a data processing intensive system, GETS's performance is directly related to its data processing speed.

Figure 1 illustrates the main components of a WPF-enabled server cluster.

2.2. GETS's design tasks

By using a WPF-enabled server cluster, GETS's development consists of design, implementation, deployment and system initialization phases. Once these phases are completed, GETS can be put into operation, which is the execution phase or runtime. Here, we aim to provide a conceptual understanding of the tasks involved in the design phase, the process that drives these tasks and the motivation behind these tasks. To help the reader to understand GETS's development cycle, this section will also provide a brief summary to other development phases.

At the design phase, our aim is to design a set of SPs for GETS and to distribute these SPs across the server cluster. Intuitively, since SPs are independent processing units, it makes sense for each of GETS's SPs to provide the same set of trading services (i.e. Order Entry, Order Matching, Trade Management, etc.) but process a different set of trading data. To achieve a balanced workload, it also makes sense for all GETS's SPs to have the same size and an even amount of workload. Furthermore, in order to maintain the server cluster's stability, it also makes sense to distribute these SPs equitably across the servers on the cluster. Finally, to provide a stable development platform, it makes sense too, to use a cluster of identical servers for GETS.

Yet, at the design stage, we can only fix the size and the number of SPs for GETS; we cannot, however, prefix the workload for each SP or each server. This is because each SP's workload

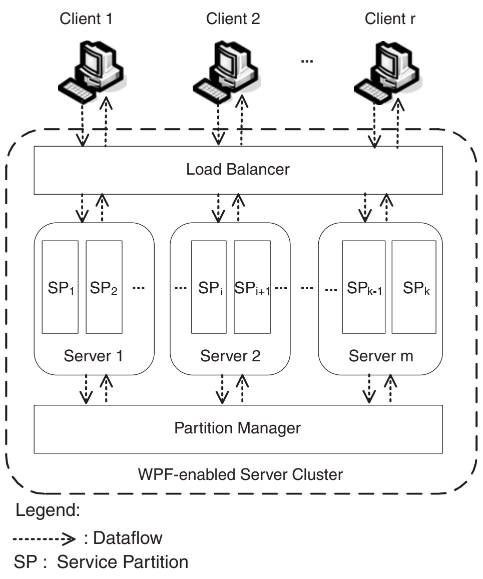


Figure 1. A WPF-enabled application server cluster is a partition-based server cluster supported by a partitioning facility and a load balancing mechanism.

will change at runtime, triggered by the service requests from the clients. The best we should do for the design is to develop a sensible load distribution strategy that can evenly distribute the workload across the SPs as well as the entire server cluster—in order to achieve a balanced workload—and that can reduce the dependency between the SPs—in order to facilitate the SPs' concurrent execution. We should also develop a sensible load balancing strategy that can maintain a balanced workload on the entire server cluster at runtime and that can adjust the servers' workload dynamically when needed.

This analysis suggests that GETS's design should entail three major tasks:

T1. To design a set of identical SPs.
T2. To design a load distribution strategy.
T3. To design a load balancing strategy.

Yet, these tasks are challenged by a set of difficult questions:

Q1. What is GETS's workload and how do we calculate it?
Q2. How should we decompose the trading data so that they can be processed by different SPs independently and concurrently?
Q3. How many identical SPs should GETS have in order to process its workload?
Q4. How many identical servers should GETS have in order to accommodate its workload?
Q5. How should we distribute the workload evenly across the SPs as well as across the entire cluster?
Q6. How do we know a server is overloaded and how do we redistribute the workload across the server cluster dynamically?

In order to answer Q1, the expert developers of the previous standalone trading system at FSC advised us to use the DA values of the trading database to predict GETS's workload. A DA value is a weighted total of the average number of the historical database transactions that has been performed on a unit of the trading data per second. Such a DA value is also called a static DA value as it represents the static workload of the data unit. We shall describe how to calculate static DA values in Section 3.

Now we know that GETS's workload is related to its trading data, we should therefore answer Q2 first before Q1. How should we work out the number of identical SPs (Q3) and the number of identical servers (Q4) needed for GETS? In theory, GETS's SPs should be independent of the server cluster and the calculation of the number of SPs needed should be based on GETS's workload. In practice, however, this calculation is restricted by the specification of the server's CPU speed, memory, disk spaces and processing speed. Even trickier for us is that we have preselected a specific type of server for GETS and our design of the SPs is therefore bound by this type of server. Because of this, we should answer Q4 before Q3.

For Q5, we cannot physically allocate the workload to the SPs at the design stage as we do not know what service requests are going to be and what actual workload each SP will process. Nevertheless, we can plan ahead by *logically* allocating the predicted static workload (according to the static DA values) evenly to the SPs and *logically* allocating these SPs to the server cluster. This means that the design of GETS's load distribution strategy is to establish two sets of logical mapping:

- a mapping from GETS's workload to its SPs and
- a mapping from GETS's SPs to its servers.

These two sets of mapping will be stored in a system configuration file to be deployed onto the server cluster at the deployment phase. During the system initialization time, the first set of mapping will be used to guide the physical distribution of the trading data to GETS's SPs and the second set of mapping will be used to allocate the SPs to the servers. At runtime, the first set of mapping will help to determine which service requests should be dispatched to which SPs, whereas the second set of mapping will be used to aid the dynamic load balancing strategy.

To answer Q6, we need to predict GETS's runtime workload. Just as we have used static DA values to predict GETS's static workload, we have adopted Winckler's load prediction method [5] to calculate dynamic database access values (also called DDA values) to estimate GETS's 'near-future' workload. We will describe how to calculate DDA values in Section 4.

Based on the above analysis, it became clear to us that the first two design tasks (T1 and T2) should be further divided into six subtasks (S1–S6). GETS's design process thus consists of three major tasks and six subtasks, as Figure 2 shows. Also shown in this figure is the correspondence between these design tasks and the above-discussed six questions (Q1–Q6). We will use Figure 2 as a roadmap to guide our description of GETS's design in the remaining paper.

We now briefly introduce other development phases. Following the design phase is the implementation where GETS's trading services, its SPs and database APIs will be implemented. At the deployment phase, the source code of the complete set of trading service operations will be installed and compiled on each server and will be shared by the server's SPs at runtime. We have already outlined that during the system initialization phase, the trading data will be allocated to GETS's SPs and the latter will be allocated to the server cluster.

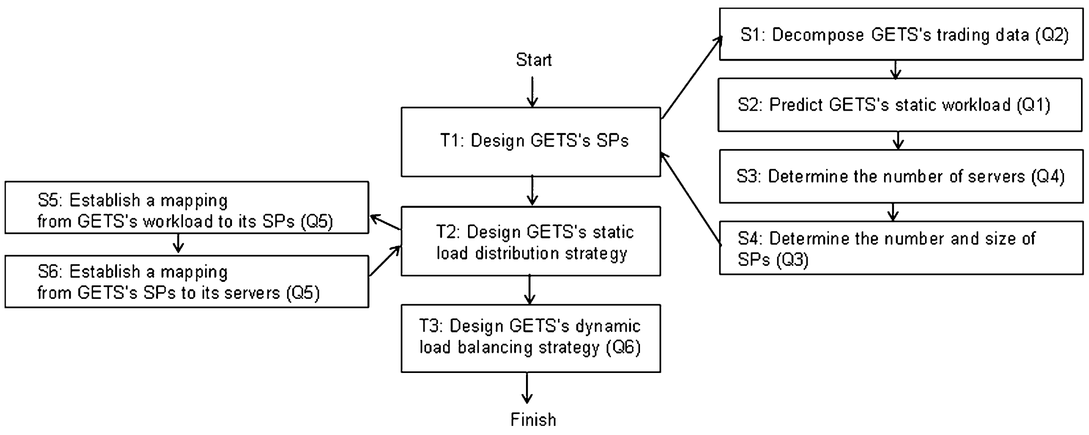

Start
T1: Design GETS's SPs
T2: Design GETS's static load distribution strategy
T3: Design GETS's dynamic load balancing strategy (Q6)
Finish
S1: Decompose GETS's trading data (Q2)
S2: Predict GETS's static workload (Q1)
S3: Determine the number of servers (Q4)
S4: Determine the number and size of SPs (Q3)
S5: Establish a mapping from GETS's workload to its SPs (Q5)
S6: Establish a mapping from GETS's SPs to its servers (Q5)

Figure 2. GETSs design process consists of three tasks (T1–T3) and six subtasks (S1–S6).

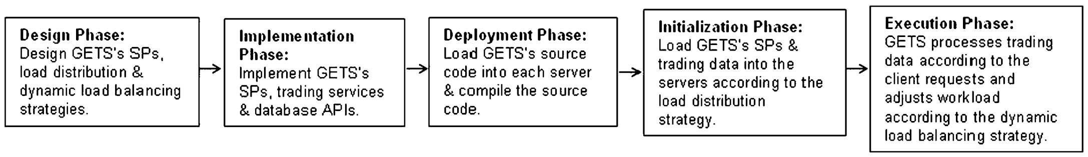

Design Phase:
Design GETS's SPs, load distribution & dynamic load balancing strategies.
Implementation Phase:
Implement GETS's SPs, trading services & database APIs.
Deployment Phase:
Load GETS's source code into each server & compile the source code.
Initialization Phase:
Load GETS's SPs & trading data into the servers according to the load distribution strategy.
Execution Phase:
GETS processes trading data according to the client requests and adjusts workload according to the dynamic load balancing strategy.

Figure 3. GETS's development and execution phases.

Figure 3 summarizes GETS's development and execution phases. Although these phases are displayed as a linear sequence, they are actually progressed iteratively, rather than in a strictly sequential order.

According to GETS's development phases, the load distribution strategy is applied at the system initialization phase to help GETS to achieve a balanced initial (pre-runtime) workload, whereas the load balancing strategy is employed at runtime to direct the dynamic load balancing operation. Owing to this important distinction, these two strategies are respectively called a static load distribution strategy and a dynamic load balancing strategy. Yet, both the strategies need to be designed at GETS's design phase.

Clearly, the design phase is most crucial, as it not only involves very challenging tasks, but also how well we perform these design tasks will directly affect how well GETS can satisfy and manage its QRs. These design tasks will be described in detail in Sections 3 and 4.

3. DESIGNING GETS'S SERVICE PARTITIONS AND STATIC LOAD DISTRIBUTION STRATEGY

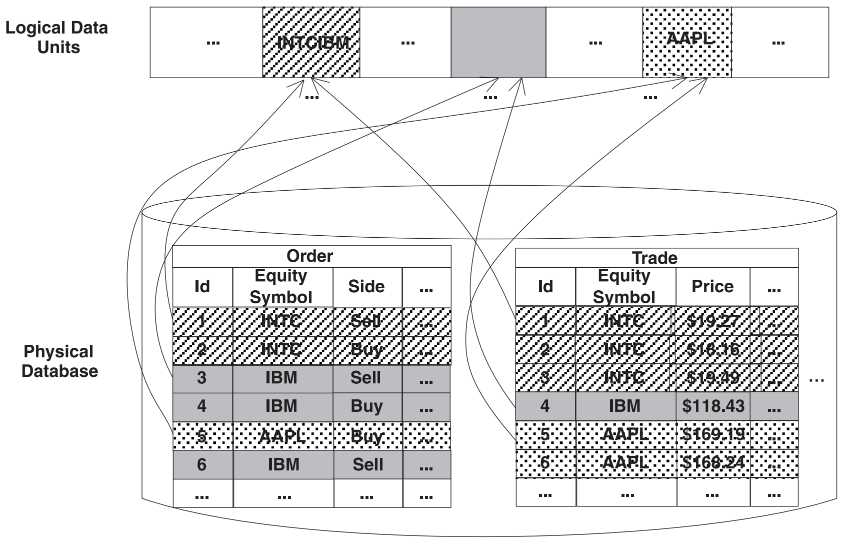
This section presents the first two design tasks (T1 and T2 in Figure 2): to design GETS's SPs and its static load distribution strategy. The ultimate goal of these two design tasks is to collectively attain a balanced static workload for the server cluster. We present these design tasks according to the order of their subtasks (S1–S6 in Figure 2).

3.1. Decomposing GETS's trading data

The purpose of this subtask is to decompose GETS's trading data into a set of independent data units (DUs) so that the workload on each DU can be calculated by their DA values and can be executed independently. Recall in Section 1 that GETS's trading data are stored in a relational database and this means that they are structured as relational tables. Figure 4 shows the Order and Trade tables of the trading database. The Order table stores both sell-side and buy-side order data. Recall also in Section 1 that during the trading process, the Order Matching service will attempt to match the buy-side orders with sell-side orders according to their equity symbols. The successfully matched orders will become the trades, which will be stored in the Trade table. Figure 4 shows three historical trades (INTC, IBM and AAPL). They were the results of the three pairs of the matched orders. For example, the IBM trade in the Trade table is the result of matching the sell-side and buy-side orders with the same IBM equity symbol. This example suggests that we should decompose the trading data into a set of order-related data units based on their equity symbols.

Yet, since at this stage GETS is still being designed, the trading data in the database do not belong to GETS and instead, they are the historical trading data. What we are doing here is to use the historical order-related data to predict GETS's current workload. This method has proved to be accurate in the development of stock trading systems.

In contrast to data fragmentation in distributed database design [6], our data decomposition is only performed *logically*, not physically—that is, we are not physically dividing the database, but instead, we only mark or tag the data in the database according to their equity symbols. We will return to this comparison in Section 7.



Order			
Id	Equity Symbol	Side	...
1	INTC	Sell	...
2	INTC	Buy	...
3	IBM	Sell	...
4	IBM	Buy	...
5	AAPL	Buy	...
6	IBM	Sell	...
...	...	...	...

Trade			
Id	Equity Symbol	Price	...
1	INTC	\$19.27	...
2	INTC	\$18.16	...
3	INTC	\$19.49	...
4	IBM	\$118.43	...
5	AAPL	\$169.19	...
6	AAPL	\$168.24	...
...	...	...	...

Figure 4. The trading data are logically decomposed into a set of independent data units (e.g. INTC, IBM and AAPL) based on their equity symbols.

3.2. *Predicting GETS's static workload*

A simple but effective method for predicting the workload of stock trading systems is to use their DA values. This method is also known to be generally applicable to data processing intensive systems. As stated in Section 2, a DA value of a particular DU in GETS's trading database is a weighted total of the average number of the historical database transactions performed on that DU per second. A database transaction is a database operation performed on reading a DU from the database or writing a DU to the database. We can find all the DUs' database transactions from the database log file and then use the following formula to calculate the DA value for each DU:

$$\mathrm{DA}_{\mathrm{du}} = R + s * W \tag{1}$$

where R is the average number of database reading operation performed on the DU per second; W is the average number of database writing operation performed on the DU per second; s is the weighting (usually $s = 2$) assigned to W to indicate that a database writing operation takes twice as much time as a database reading operation.

The values of R and W for each DU are based on the statistics of the database transactions performed on the DU.

Once we have calculated all the DA values for all the DUs stored in the trading database, we can add them together as shown in Formula (2). This total DA value (DA_{GETS}) will then be used to denote GETS's total static workload

$$DA_{\mathrm{GETS}} = \sum_{i=1}^{N} DA_{\mathrm{du}_i} \tag{2}$$

In Formula (2), N represents the total number of DUs decomposed in the trading database and i points to the ith DU. So far, GETS's static workload has been around 9000 DA values.

3.3. *Determining the number of servers needed for GETS*

As described in Section 2, we have preselected a specific type of server[§] for GETS. Here, our task is to work out the maximum number of identical servers needed for GETS. This involves

[§]The server we selected consists of one CPU of 2.4 GHz, 2G memory and one hard disk with the speed of 80 M bytes per second.

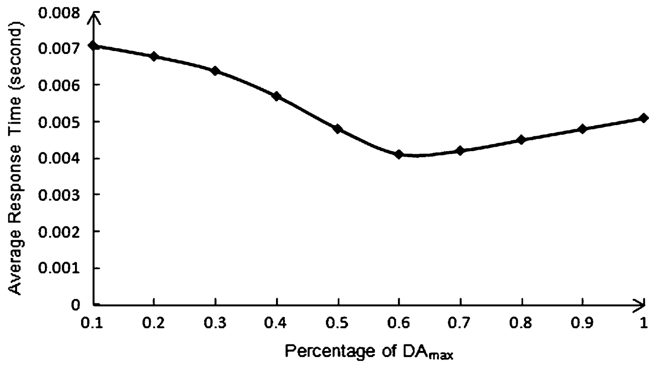


Figure 5. The relationship between a server's response time and its workload, measured by the servers database access value.

finding out what percentage of GETS's static workload each server can process. To do so we use one of our chosen servers to perform the Order Matching operation on GETS's trading database. We then record the maximum number of the database transactions that the server can perform every second. Finally, we use Formula (3) to calculate the maximum DA value ($DA_{\max}$) for this server. The value obtained thus indicates the server's maximum workload capacity, which is a weighted total of the maximum number of database transactions performed by the server on the trading database

$$\mathrm{DA}_{\max}=R+2W \tag{3}$$

Formula (3) suggests that database writing operations require twice as much time as database reading operations. For our chosen server, its $\mathrm{DA}_{\max}$ value is around 5000.

Yet, we cannot directly use the $\mathrm{DA}_{\max}$ value to calculate the number of servers required for GETS, because we need to reserve a portion of this value for the dynamic load balancing operation. To find out what proportion of the $\mathrm{DA}_{\max}$ value should be reserved, we test the server's response times under different workload conditions (i.e. from using 10–100% of its $\mathrm{DA}_{\max}$ value). The experimental results (Figure 5) show that the server has the best response time when its workload falls within 60 and 70% of its $\mathrm{DA}_{\max}$ value. By taking the mean value, we have decided to use 65% of this value as the server's processing capacity and to reserve 35% of this value for the server's dynamic load balancing operation. The number of identical servers needed by GETS can be calculated as follows: $\mathrm{DA}_{\mathrm{GETS}}/(DA_{\max}*65\%)$.

Suppose that GETS's $\mathrm{DA}_{\mathrm{GETS}}$ value is 9000, and the server's $\mathrm{DA}_{\max}$ value is 5000, we can work out that the number of identical servers needed for GETS is three.

3.4. Determining the number of SPs for each server and the size of each SP

To determine the number of identical SPs suitable for each server, we tested a server's performance under different number of SPs. Note that our test only uses 65% of the server's $\mathrm{DA}_{\max}$ value, which is around 3250. The test shows that the server has the best performance when it hosts between four and six SPs (Figure 6). By considering the resources (memory space and processing time) required for managing the SPs, we decided that each server should have at most four SPs. This means that for a cluster of three servers, the total number of SPs should be 12.

For each SP, we can also work out its size—that is, its workload capacity, which is $(DA_{\max}*0.65)/4$. Since $\mathrm{DA}_{\max}=5000$, the size of an SP is therefore 813 DA values. As all SPs are identical, they should all have the same size and the same DA value, that is, the same workload capacity.

3.5. Establishing the mapping from GETS's static workload to its SPs

After performing the above subtasks, we are now ready to design GETS's static load distribution strategy. As stated in Section 2, our design goal for this strategy is to distribute GETS's workload

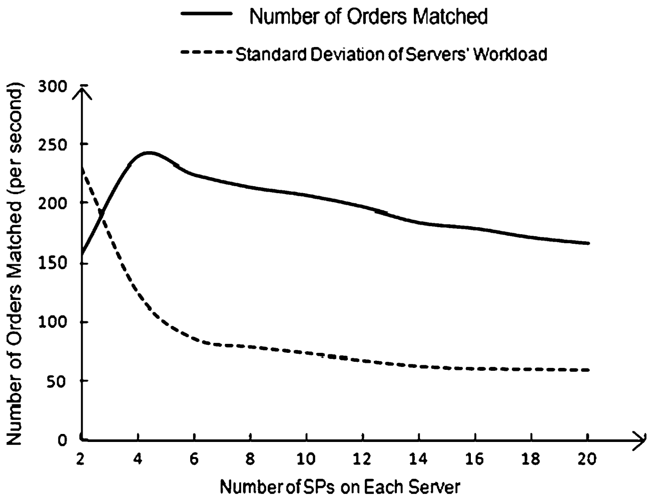


Figure 6. The relationship between the number of SPs on a server and the server's performance, measured by its throughput.

evenly across the SPs as well as across the entire server cluster, and to reduce the dependency between the SPs. We also explained that the design of such a strategy entails the establishment of two sets of logical mapping:

- a mapping from GETS's workload to its SPs and
- a mapping from GETS's SPs to its servers.

In this section, we describe how to establish the first set of mapping. We defer the description of the second set to the following section.

Based on our design goal, there are two conditions under which a mapping from GETS's workload to its SPs should be established: (1) to ensure that all the DUs in the trading database are equitably mapped onto the SPs and (2) to ensure that the related DUs are mapped onto the same SP where possible. The first condition is to achieve a balanced static workload on all the SPs, whereas the second one is to reduce the dependency between the SPs so that the SPs can be executed concurrently. Based on these conditions, we have defined two rules for mapping the DUs to the SPs:

Rule 1. Always map the related DUs to the same SP if possible.
Rule 2. Always map the DU with the highest DA value (i.e. the highest workload) to the SP with the lowest total DA value of its DUs (i.e. the lowest workload).

Rule 1 corresponds to the second condition, whereas Rule 2 is for the first condition. Their reverse order suggests that reducing the dependency between the DUs should always be the foremost criterion for the load distribution and it is under this condition that the second rule applies. For Rule 1, two DUs might be related if they have the same client. For Rule 2, a higher DA value means a higher workload. Note also that the static workload of a DU is not measured by its volume, but instead, by its DA value.

Based on these two rules, we have devised a procedure for mapping GETS's DUs to its SPs. This procedure is presented in Figure 7 and is illustrated as follows.

The mapping begins by ordering all the DUs in the descending order of their DA values, as shown in Figure 8. Since all the SPs are empty at the beginning, their DA values for the DUs should be zero. Therefore, initially, the SPs are ordered by their identification numbers, not by their DA values. To start with, Rule 1 does not apply as no DUs have been mapped yet. Thus according to Rule 2, we map the DU with the highest DA value (e.g. DU_1) to the SP with the lowest DA value (e.g. SP_1). This SP's remaining workload capacity will be decreased as a result of this mapping and its new DA value is now the DU's DA value.

```
Line 1:   Find the DA value for each DU from the database log
Line 2:   repeat
Line 3:     Sort unmapped DUs in the ascending order according to their DA values
Line 4:     Choose the unmapped DU with the highest DA value as the current DU
Line 5:     for all SPs with DA value smaller than Max_SP_Size do
Line 6:       Sort SPs in the descending order according to their DA values
Line 7:       if dependency exists between the current DU and a DU already mapped to a different SP
Line 8:       then Map the current DU to that SP
Line 9:       else
Line 10:        Map the current DU with the highest DA value to the SP with the lowest DA value
Line 11:  until all DUs are mapped to SPs
```

Figure 7. The procedure applied to map GETSs static workload (i.e. DUs) to its service partitions.

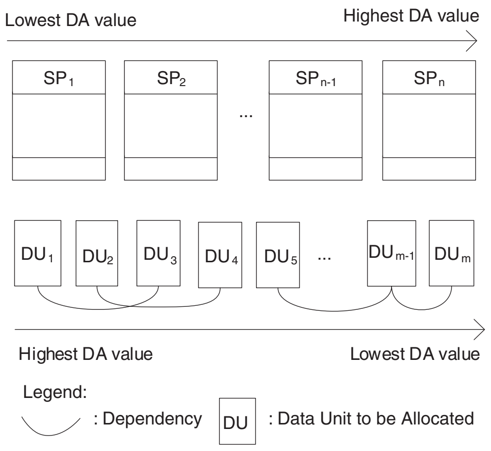


Figure 8. Mapping a set of DUs to the SPs: the starting point.

We then reorder all the SPs based on their new DA values. This time SP_1 has the highest DA value as it has been allocated to DU_1 and will then be placed in the lowest position. According to Rule 1, before allocating DU_2, we check whether it has a dependency with DU_1. If no, then DU_2 is allocated to SP_2 which has the highest DA value. In the third round, we reorder the SPs and place SP_2 before SP_1, as it has a higher DA value than SP_1. When mapping DU_3 we notice that DU_3 has a dependency with DU_1 and according to Rule 1 we map DU_3 to SP_1 which contains DU_1. Figure 9 shows the allocation described so far.

When it comes to mapping DU_{m-1}, we encountered a situation as shown in Figure 10: DU_{m-1} should be allocated to SP_3 because of its relationship with DU_5; however, by doing so SP_3 would run out the space. To solve this problem, DU_{m-1} has to be mapped onto an SP with the lowest DA value, which currently is SP_{n-3}. We repeat this process until all the DUs have been mapped onto the SPs.

3.6. Establishing the mapping from GETS's SPs to its servers

There are also two conditions under which a mapping from GETS's SPs to its servers should be established: (1) to ensure that all the SPs are equitably mapped onto the servers and (2) to ensure that the related SPs are mapped onto the same server where possible. The first condition is to achieve a balanced static workload on all the servers, whereas the second one is to reduce the dependency between the servers so that the SPs can operate concurrently as well as independently. Based on these conditions, we have defined two rules for mapping GETS's SPs to its servers:

Rule 1. Always map the related SPs to the same server if possible.
Rule 2. Always map the SP with the highest DA value (i.e. the highest workload) to the server with the lowest DA value (i.e. the lowest workload).

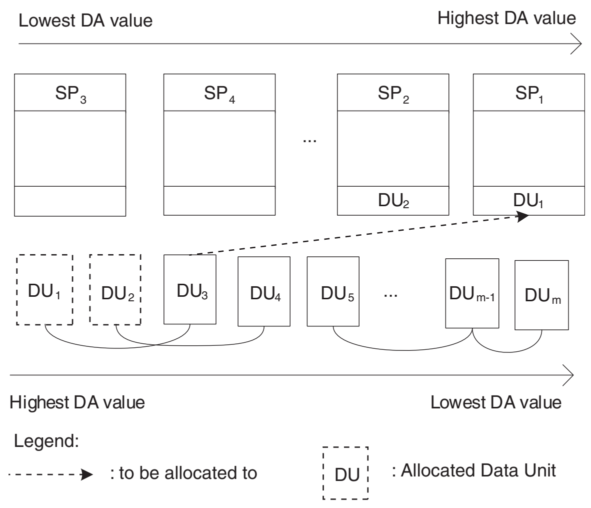


Figure 9. Mapping the DUs to the SPs: no dependency between DU_1 and DU_2.

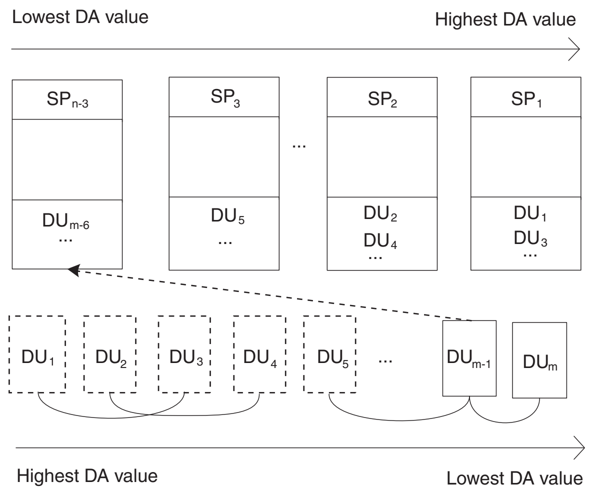


Figure 10. Mapping the DUs to the SPs: Dependencies exist between DU_1 and DU_3, and DU_2 and DU_4.

Rule 1 corresponds to the second condition, whereas Rule 2 is for the first one. Their reverse order suggests that reducing the dependency between the SPs should always be the foremost criterion for the load distribution and it is under this condition that the second rule applies. For Rule 1, two SPs are said to be related if their DUs are related. Since the number of the SPs is much smaller than the number of DUs, we can manually adjust the SPs to ensure a best possible mapping from the SPs to the servers.

At the end of this design step, we have produced two sets of logical mapping: (1) the mapping from the DUs to the SPs and (2) the mapping from the SPs to the servers. These mappings will be stored in the system configuration file to be placed in the server cluster's middleware. During the system initialization phase, they will be used to guide the physical allocation of the trading data to the SPs as well as the allocation of the SPs to the servers. At runtime, the first set of mapping will be used to determine which service requests should be dispatched to which SPs, whereas the second set of mapping will be used to aid the dynamic load balancing strategy. Together, these two sets of mapping form a static load distribution strategy for GETS.

4. DESIGNING GETS'S DYNAMIC LOAD BALANCING STRATEGY

As stated in Section 2, WPF-enabled server clusters provide a dynamic load balancing mechanism for adjusting servers' workload at runtime. How this mechanism works is determined by its load balancing strategy. In this section, we present our design of such a strategy for GETS, which is T3 in Figure 2.

In what follows, we first briefly introduce the standard load balancing mechanism and its strategy provided by WPF-enabled server clusters; we then present a dynamic load balancing strategy designed specifically for GETS.

4.1. GETS's dynamic load balancing mechanism

Each WPF-enabled server cluster provides a standard mechanism for the dynamic load balancing operation. This mechanism consists of three cluster-based components, which are Load Balancer, Data Aggregator and HA Coordinator, and three server-based components, which are Load Recorder, Actuator and HA Manager. Each server has a set of server-based components. The static relationships between these components are depicted in Figure 11. Below we describe how these components interact at runtime to perform the dynamic load balancing operation.

The dynamic load balancing operation is carried out cyclically at runtime to check whether a server is overloaded or crashed. At each cycle, the Data Aggregator collects the control data from each server' Load Recorder and forwards these data to the Load Balancer. The Load Balancer will then use a dynamic load balancing strategy to determine whether the workload on the server cluster needs to be rebalanced. If so, this strategy will generate a new set of mapping for the affected SPs and their servers and cause the system configuration file to change accordingly. According to this mapping, the HA Coordinator will coordinate the Actuators on the affected servers to redistribute the affected SPs. Figure 12 illustrates the interactions between different components in performing the dynamic load balancing operation.

Clearly, the dynamic load balancing strategy plays a central role in the dynamic load balancing operation. WPF-enabled server clusters provide a standard strategy which uses each server's transaction count (the number of operations performed over a certain period of time) and the server's response time (the total response time of all transactions performed over a certain period of time) to control load balancing on each server. This standard strategy assumes that the server cluster's workload is measured by its Performance Monitoring Infrastructure (PMI) data. This strategy, however, is not suitable for GETS as the PMI data are not accurate enough to represent

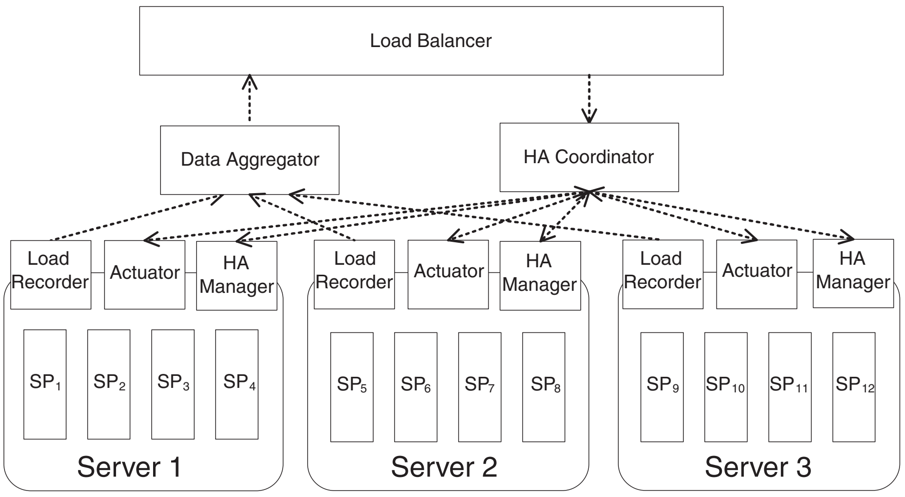


Figure 11. A WPF-enabled server cluster and its dynamic load balancing components.

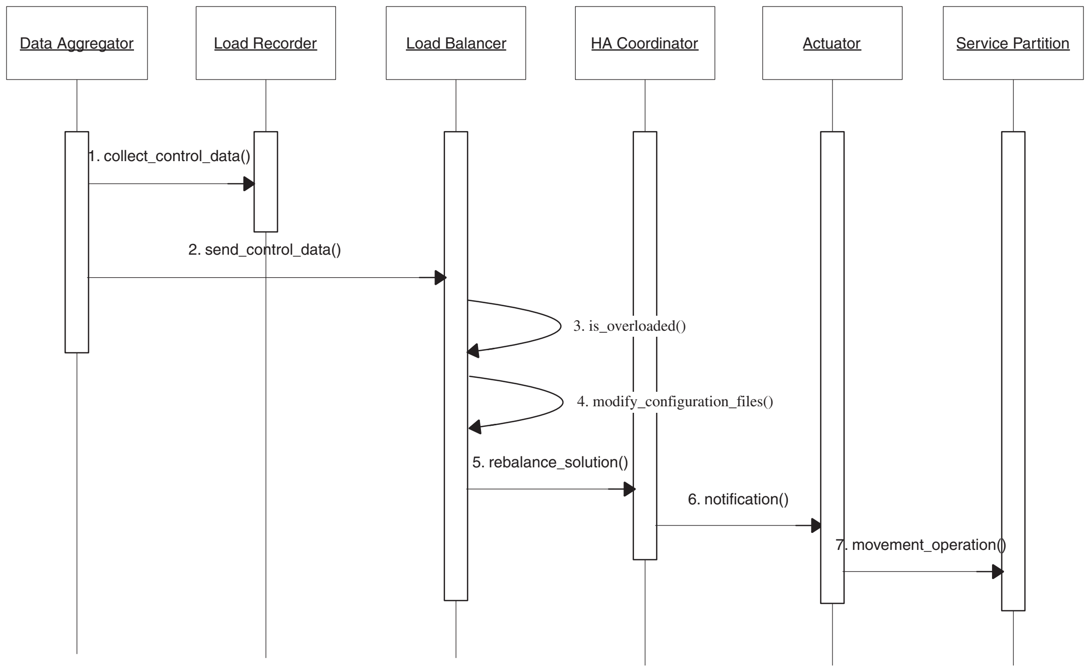


Figure 12. A complete cycle of the dynamic load balancing operation.

GETS's workload. We have therefore designed a new dynamic load balancing strategy for GETS. In this new strategy, the DDA values are used to determine the servers' dynamic workload. In the following sections, we describe this new load balancing strategy and its parameters.

4.2. GETS's dynamic load balancing strategy

As described in the previous section, according to the dynamic load balancing strategy, the workload on some servers may need to be rebalanced. This means that some SPs on the overloaded servers may need to be physically moved to other servers at runtime. This operation is performed by the Actuators on the affected servers and will take the following steps: (1) deactivating the services on the affected SPs, (2) transferring the SPs' DUs and their state data to the recipient servers according to the new mapping generated by the dynamic load balancing strategy, (3) deploying a set of new SPs to the receipent servers to replace the affected SPs and finally (4) reactivating the services on the new SPs. Clearly, from deactivating the services to reactivating them, the services will be interrupted. In the design of GETS's dynamic load balancing strategy, our goal is to minimize the movement of the SPs and thereby to reduce service interruption caused by such movement. Accordingly, we have laid down three design rules for GETS's dynamic load balancing strategy:

Rule 1. Only move an SP when absolutely necessary.
Rule 2. Always move the SP with the lowest workload (i.e. the smallest SP).
Rule 3. Always move the SP to the server with the lowest workload.

Rule 1 suggests that we should reduce the frequency of moving the SPs and in doing so, reduce the service interruption. Rule 2 implies that if we do need to move the SPs, we should move the smallest SP because it will incur less service interruption time than moving the largest SP. Rule 2 also implies that moving the smallest SP is less likely to cause the decomposition of the workload on the SP, whereas moving the largest SP may cause the load decomposition, for example, if the recipient server can only receive part of the load on the largest SP. Rule 2 therefore also aims at reducing the dependency between the servers as well as between the SPs. Rule 3 is used to achieve a balanced workload on the entire server cluster.

Based on these rules, we have designed the following dynamic load balancing strategy for GETS:

(1) For each server, sort its SPs (SP_m, $\mathrm{SP}_{m-1}, \ldots, SP_1$) in the descending order according to their runtime workload. An SP's runtime workload is measured by its DDA value (denoted as DDA_{sp}). An SP's DDA_{sp} predicts the SP's 'near-future workload'—that is, the SP's workload in the next time interval. The higher an SP's DDA_{sp} value, the heavier the SP's workload. We will describe how to calculate this value in Section 4.3.
(2) Sort all the servers (S_m, $S_{m-1}, \ldots, S_1$) in the descending order according to their runtime workload. A server's runtime workload (DDA_s) is the total workload of the server's SPs, that is, the total DDA_{sp} values.
(3) Check whether S_m has exceeded its runtime load limit (denoted as $\mathrm{DDA}_{\max}$). If so, continue; otherwise, abort the algorithm. Since GETS's servers are identical, we only need to use one $\mathrm{DDA}_{\max}$. We will describe how to obtain this value in Section 4.3.
(4) Determine if S_m's SPs need to be moved to S_1 by comparing S_m's DDA_s value with that of S_1 (the server with the least DDA_s). If the load ratio from S_m to S_1 is greater than the allowed load fluctuation value (denoted as $F_{\max}$), continue; otherwise, abort the algorithm. We will describe how to determine $F_{\max}$ in Section 4.3.
(5) Check whether by moving the smallest SP from S_m to S_1 will make S_1 overloaded. This is done by adding the SP's DDA value to S_1's DDA value and comparing this aggregated value with $\mathrm{DDA}_{\max}$. If so, abort the algorithm; otherwise, continue.
(6) Move the SP from S_m to S_1 by deactivating the SP's services, transferring its data units and its state information to S_1, deploying a new SPs to S_1, and then reactivating the services on the new SP.
(7) Modify the mapping for the new SP and its target server in the system configuration file.

In this strategy, Steps 3 and 4 aim at minimizing the movement of the SPs (Rule 1). Specifically, Step 3 determines whether a server is overloaded and Step 4 decides whether the server's load is unbalanced with respect to the least loaded server and whether the load fluctuation is below a certain threshold. Only when the both Steps are true, will the load rebalancing operation be performed. Therefore, these two steps collectively control the frequency of the movement of the SPs and thus reduce the service interruption.

When an overloaded server has to be rebalanced, according to Rule 2, the priority is to move the SP with the smallest DDA value (i.e. the lowest workload) rather than the SP with the largest DDA value (i.e. the highest workload). Step 5 determines which SP should be moved.

Finally, according to Rule 3, our strategy is to move the smallest SP to the least loaded server (Steps 1 and 2).

It is important to note that GETS's dynamic load balancing strategy relies on its static load distribution strategy to provide a stable initial workload for the server cluster and to minimize the dependency relationships between the DUs, between the SPs and between the servers. Building on this foundation, GETS's dynamic load balancing strategy can then direct the dynamic load balancing operation with ease. These two strategies thus work in unison to support GETS's dynamic load balancing operation.

4.3. Determining the parameters for GETS's dynamic load balancing strategy

4.3.1. Determining the dynamic workload for SPs and servers. Just as we use the data access (DA) values of the DUs in GETS's trading database to measure the static workload for GETS's SPs and servers (see Section 3), here we use the DDA values of the DUs to determine the runtime workload for GETS's SPs and servers. Specifically, an SP's dynamic workload can be predicted by its DDA value, which is a weighted total of the average number of the historical database transactions performed on all the DUs of the SP within a second. Each SP's DDA value can be calculated by using the workload prediction method [5], as shown below

$$\mathrm{DDA}_{sp} = R_t * \left(1 + \frac{R_{t1} - R_{t0}}{R_{t0}}\right) + s * W_t * \left(1 + \frac{W_{t1} - W_{t0}}{W_{t0}}\right) \tag{5}$$

where R_t is the number of times an SP reads all its DUs from the trading database within the time unit t, is usually less than a second; W_t is the number of times an SP writes all its DUs back to the trading database within the time unit t, is usually less than a second; s is the weighting (usually is 2) for W_t to emphasize that a database writing operation takes twice as much time as a database reading operation.

$(R_{t1} - R_{t0})/R_{t0}$: This is to predict the trend of reading the DUs from the trading database between t_1 and t_0 in the near future. If the trend is less than zero, the number of reading operations on the DUs in the near future will be smaller than R_t; if the trend is equal to or greater than zero, the number of reading operations on the DUs in the near future will be equal to or greater than R_t. The values of R_{t0} and R_{t1} are calculated by adding the reading times for the SP's DUs within the time units t_0 and t_1, respectively.

$(W_{t1} - W_{t0})/W_{t0}$: This is to predict the trend of writing the DU back to the trading database between t_1 andt_0 in the near future. If the trend is less than zero, the number of writing operations on this DU in the near future will be smaller than W_t; if the trend is equal to or greater than zero, the number of writing operations on this DU in the near future will be equal to or greater than W_t. The values of W_{t0} and W_{t1} are calculated by adding the writing times for the SP's DUs within the time units t_0 and t_1, respectively.

After we have calculated the DDA values for all GETS's SPs, we can obtain the runtime workload for each server (DDA_s) by using the following:

$$\mathrm{DDA}_s = \sum_{i=1}^{N} \mathrm{DDA}_{sp_i} \tag{6}$$

In Formula (6), N represents the total number of the SPs on the server and i points to the ith SP.

4.3.2. Determining the maximum dynamic load limit for each server. A server's maximum dynamic load limit stipulates the maximum dynamic workload that each server can have at runtime. This limit, represented by a server's dynamic database access ($\mathrm{DDA}_{\max}$) value, is used to constrain GETS's dynamic load balancing operation (Section 4.2). Below, we show how to determine this value.

We can derive a server's $\mathrm{DDA}_{\max}$ value from its $\mathrm{DA}_{\max}$ value, which represents the server's maximum static workload (see Section 3). Each server should reserve a portion of this value for the room to support the dynamic load balancing operation. In Section 3, we have showed that a server has the best response time when its static workload falls within 60–70% of its $\mathrm{DA}_{\max}$ value. This suggests that a server's $\mathrm{DDA}_{\max}$ value should be between 60 and 100% of its $\mathrm{DA}_{\max}$ value, as shown below

$$\mathrm{DDA}_{\max} = x * \mathrm{DA}_{\max} \quad \text{where } 0.60 < x < 1 \tag{7}$$

The question is, within this range, what is the best x for a server's $\mathrm{DDA}_{\max}$ value? To answer this question, we need to conduct the following experiments. First, we choose different percentages between 60 and 90% of the $\mathrm{DA}_{\max}$ value using GETS's dynamic load balancing strategy. We set $F_{\max}$ to zero so that Step 4 in the algorithm will always be true—in other words, the dynamic load balancing operation will always be needed. For each percentage of the $\mathrm{DA}_{\max}$ value, we record its corresponding service interruption time. Figure 13 shows a graph obtained by connecting different percentages of the $\mathrm{DA}_{\max}$ value with their corresponding service interruption times. The graph indicates that as the percentage of the $\mathrm{DA}_{\max}$ value increases, the server's service interruption time decreases.

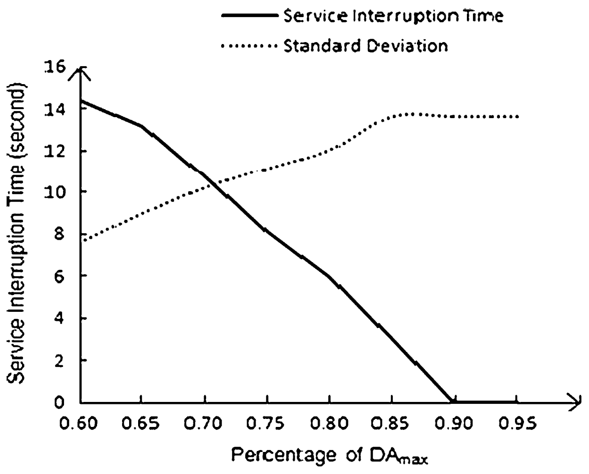


Figure 13. According to the standard deviation of the server cluster, a servers maximum dynamic load limit DDA$_{\text{max}}$ should be within the range of 70–80% of the server's maximum static workload DA$_{\text{max}}$. Below this range, the server will require more frequent dynamic load balancing and thus more service interruption time, whereas above this range the server will have less service interruption time, but its load will be too heavy and thus the server will become unstable.

Second, at each time interval, we calculate the average runtime load on the entire server cluster. This is done by adding each server's runtime workload (DDA$_s$) together and dividing the total by the number of servers on the cluster, as shown below

$$\mu = \frac{1}{N}\sum_{j=1}^{N} \text{DDA}_{sj} \tag{8}$$

Third, for each average load, we calculate the standard deviation (SD) among the servers' workload using Formula (3). The SD represents the load difference among the servers

$$\text{SD} = \sqrt{\frac{1}{N}\sum_{j=1}^{N} (\text{DDA}_{sj} - \mu)^2} \tag{9}$$

By drawing a graph with different SD values obtained at different time intervals and superimposing this graph with that of the percentages of the DA$_{\text{max}}$ value, we found that the two graphs intersect between 70 and 75% of DA$_{\text{max}}$ (see Figure 13). The intersection point represents the optimal percentage for the DDA$_{\text{max}}$ value, but due to the dynamic nature of a server's runtime workload, we have chosen a range of percentages (between 70 and 80% of DA$_{\text{max}}$), as the candidates for calculating a server's DDA$_{\text{max}}$ value. This gives us the freedom to fine-tune GETS's dynamic load balancing strategy by assigning one specific percentage from this range as a server's DDA$_{\text{max}}$ value.

4.3.3. Determining the maximum load fluctuation for the server cluster. The maximum load fluctuation (F_{max}) is a ratio between the server with the highest load and the server with the lowest load. This constraint, together with the server's maximum load limit (DDA$_{\text{max}}$), will be used to restrict the frequency of the movement to the SPs, as described in Section 4.2. In this section, we show how to determine this constraint.

We intuitively know that F_{max} should be between 1.1 and 1.9. This is because $F_{\text{max}} = 1$ and $F_{\text{max}} = 2$ are two extreme situations that do not normally exist in reality. More specifically, $F_{\text{max}} = 1$ means that all servers on the cluster have exactly the same amount of workload and hence no need for load balancing. By contrast, $F_{\text{max}} = 2$ indicates that the workload of one server is twice the

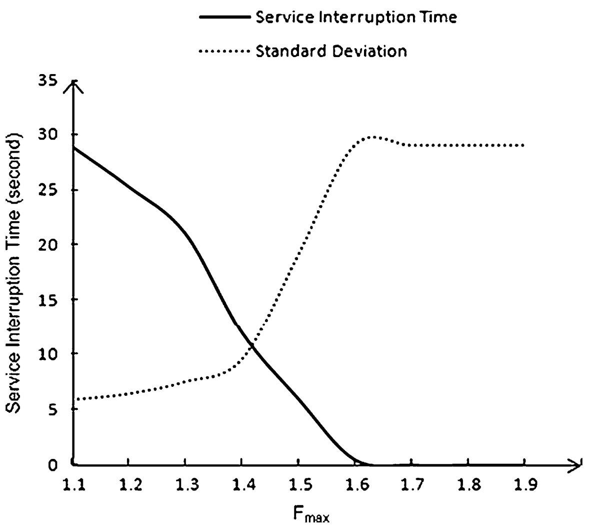


Figure 14. According to the standard deviation of the server cluster, the maximum load fluctuation of the server cluster $F_{\max}$ should be between 1.3 and 1.5. Below this range, the cluster will require more frequent dynamic load balancing and thus more service interruption time, whereas above this range the cluster will have less service interruption time, but the entire cluster will become unstable, due to unbalanced load on the servers.

workload on another server and this implies that the server cluster will always require the dynamic load balancing operation. Therefore, $F_{\max}$ should fall within the range of 1.1 and 1.9 and within this range, the higher the $F_{\max}$, the more unbalancing the server cluster. The question is: What value should be ideal for $F_{\max}$?

To determine an optimal value for $F_{\max}$, we set $\mathrm{DDA}_{\max} = 0.8 * \mathrm{DA}_{\max}$ so that it is the upper bound of $\mathrm{DDA}_{\max}$. Based on this value, we execute our load balancing algorithm with different $F_{\max}$ values, from 1.1 to 1.9. For each $F_{\max}$ value, we record its corresponding service interruption time. Figure 14 shows the relationship between different $F_{\max}$ values and their related service interruption times. By superimposing the cluster's SD (Formula 8), we notice that when the range of $F_{\max}$ increases from 1.1 to 1.5, the service interruption time decreases, but the SD increases. This means that the number of SPs to be moved is also becoming smaller. However, when $F_{\max}$ reaches 1.6, the interruption time becomes zero and the SD also reaches the highest value. This means that no SP will be moved (thus the zero interruption time). Thereafter, the increase in $F_{\max}$ will have no effect on the service interruption time and the SD. This means that after $F_{\max}$ reaches 1.6, it becomes too big for the actual load fluctuation. Hence, Step 5 of the dynamic balancing algorithm (Section 4.2) will always yield a false. Therefore, when striking a balance between the service interruption times and the servers' SD, we found that an optimal value for $F_{\max}$ should be between 1.3 and 1.5.

5. EXPERIMENTAL EVALUATION

The best way to assess how well our design has met its QRs (i.e. resource utilization, performance, scalability and availability) is to implement it. The implementation is based on a cluster of three identical WPF-enabled servers. Each server is configured as one CPU of 2.4 GHz, one piece of 2G Memory, one hard disk with the speed of 80 M bytes per second, and all servers are connected to a 100 M high-speed Intranet. We then conducted a series of experiments on this system, each focusing on one specific QR. The following sections discuss our experimental results and evaluate our design against each of the QRs.

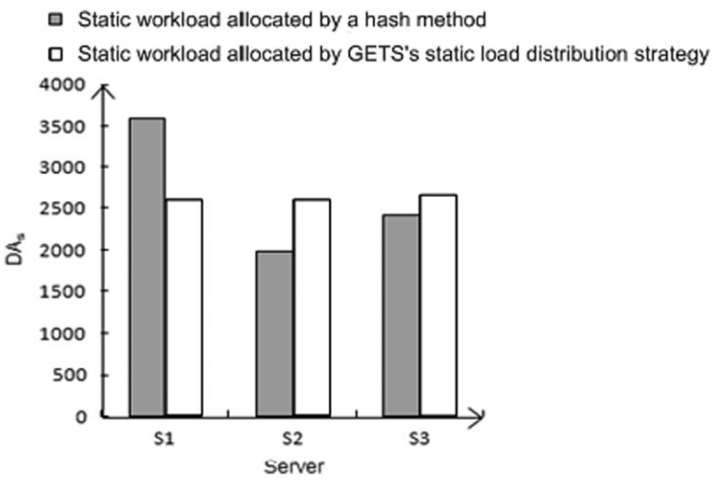


Figure 15. GETS's static load balancing strategy has achieved a better balanced server cluster than a hash allocation method.

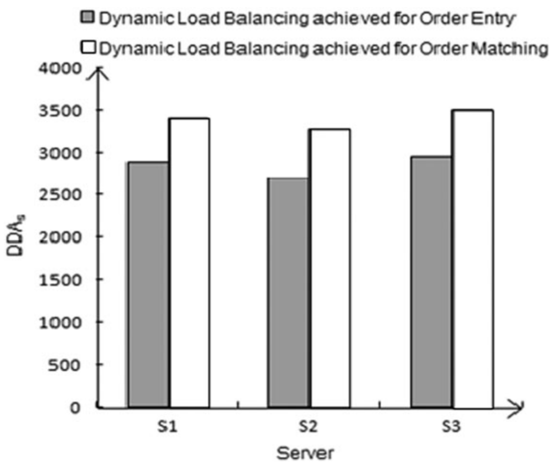


Figure 16. GETS's dynamic load balancing strategy has achieved a balanced workload under both Order Entry and Order Matching services.

5.1. Resource utilization

We have satisfied this QR in the design of a static load distribution strategy and a dynamic load balancing strategy for GETS. To assess the effectiveness of the first strategy, we have compared it with a random hash allocation method. Figure 15 shows that this strategy has achieved a much better balanced static workload, which has a lower SD (0.076) than the hash method, which has a higher SD (0.36).

To evaluate the effectiveness of GETS's dynamic load balancing strategy, we tested it on the Order Entry and Order Matching services, respectively. For each service operation, we have measured the fluctuation value for the server cluster, which is 1.09 for Order Entry and for 1.07 for Order Matching (see Figure 16). Recall in Section 4.3.3 that GETS's maximum fluctuation value ($F_{\max}$) was set to be between 1.3 and 1.5. The fluctuation values for both Order Entry and Order Matching services are much lower than this maximum range, and are near optimal—that is, when the fluctuation value is equal to 1. This demonstrates that GETS's dynamic load balancing strategy is very effective.

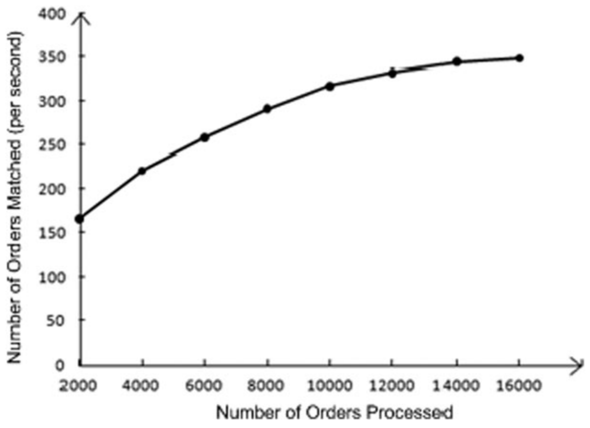


Figure 17. GETS has a good performance, demonstrated by the positive increase in its throughput of the number of the matched orders in relation to the number of processed orders.

5.2. Performance

This QR is measured by GETS's throughput, which is the number of the orders matched per second. In order to meet this QR, our design has taken the following into considerations: First, we have aimed at minimizing the number of the dependencies in the system (Section 3), thus reducing DA time and increasing the system efficiency. Second, we dynamically adjusted the workload on each server (Section 4) to prevent server crashes caused by heavy workload.

To evaluate how well GETS has satisfied this QR, we have used the number of the orders matched per second to measure the throughput of Order Matching service. Figure 17 shows that GETS has a good performance, demonstrated by the positive increase in the number of the matched orders in relation to the number of processed orders. Figure 17 also shows that GETS's throughput slows down when the number of orders reaches a certain threshold, due to the increase in the processing time caused by the large number of processed orders.

5.3. Scalability

This QR requires GETS to be extensible without compromising its performance. This means that (1) GETS should be able to process additional trading data and (2) GETS should be able to be extended with additional servers. The scalability in the first situation has already been considered in GETS's static load distribution strategy which allows new trading data to be decomposed and allocated equitably across the existing servers.

In the second scalability situation, we tested GETS's performance under different number of servers and the results in Figure 18 show that GETS's performance and the number of servers increase in unison, which means that GETS can be scaled up without compromising its performance.

5.4. Availability

This QR has two aspects: (1) GETS's entire server cluster should provide the service continuously for as long as possible; (2) Should a server crash at runtime, its recovery time should be as little as possible. These two aspects have been explicitly considered in our design. First, GETS's dynamic load balancing strategy ensures that the workload on the servers is dynamically adjusted so that no server will exhaust its resources or become overloaded. This strategy therefore helps to minimize the failure of the servers due to resource exhaustion or heavy workload. Second, GETS's static load distribution strategy aims at minimizing inter-data dependencies, and consequently, this strategy helps to shorten data synchronization time and reduce the server recovery time. These strategies, together with the self-recovery mechanism on each server (via the HA Manager), collectively fulfill the availability QR of GETS.

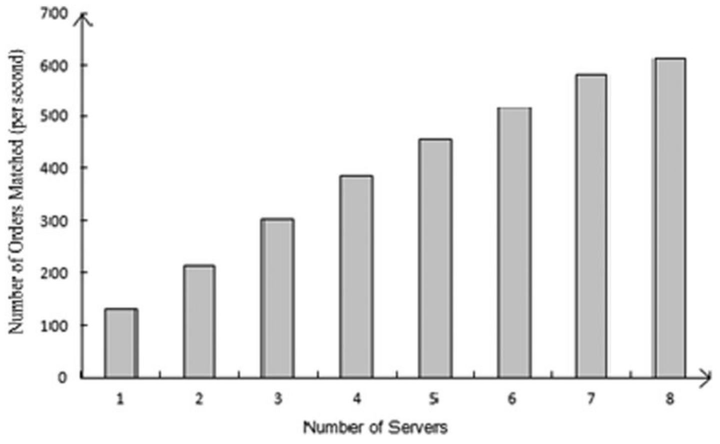


Figure 18. GETS has a good scalability, as shown by the relationship between the performance (the throughput of the matched orders) and the number of servers.

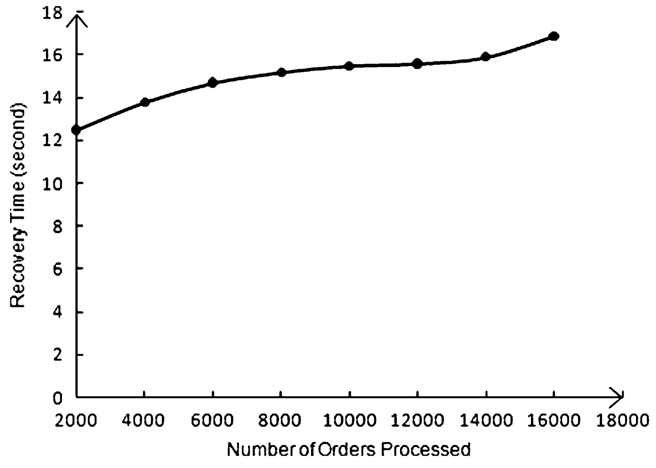


Figure 19. GETS has a good availability, which is measured by the recovery time of its server in relation to the number of orders processed.

To evaluate how well GETS has satisfied the availability QR, we used the recovery time as a yardstick to measure how quickly a server can recover from a crash. The recovery time is the difference between the time when the service on a server is interrupted and the time when the service is resumed. We tested the server cluster on different workloads, from 2000 to 16 000 trading orders. We fed each batch of orders into GETS and ran the Order Matching service on these orders; we then randomly choose a server to shut down and recorded its recovery time. The relationship between different batches of orders and their corresponding recovery times on the server (Figure 19) shows that the recovery times only fluctuate around 14–15 s between different batches of orders. From the history data, we found that GETS has less than 3 crashes a year. According to these data, we can work out the availability of GETS to be $(1-(15*3)/(365*24*60*60)\approx 99.999\%)$. This demonstrates that GETS is robust enough to satisfy the availability QR. Figure 19 also shows that the recovery time increases slightly as the number of orders increases, as loading more data requires more time. Nevertheless, in spite of this, the increase in the recovery time is very small in comparison with the increase in the number of orders. This demonstrates that even under a higher workload situation, GETS still offers an HA.

6. REFLECTION AND DISCUSSION

It has been several years since GETS was developed. During the course of writing this paper, we have retraced our design journey and revisited our design ideas. Such reflections have helped us to understand better why and how we solved GETS's design problems in the way presented in this paper. This section provides a reflection on our design and presents some important lessons we have learned from this design.

6.1. Design tradeoffs and room for improvement

In this section, we discuss some design alternatives that we have considered for GETS, but have not adopted; we speculate how the current design might be improved by combining some alternative approaches.

6.1.1. Static workload calculation and distribution. GETS's static workload calculation is based on the DA values of the data units (Section 3). This method is only feasible if such history data exist, for example, in the case of a re-engineering project like GETS.

Moreover, even with the existence of the history data, our calculation method can be very time-consuming, as it requires the calculation of all the DUs' DA values in GETS's trading database and for all GETS's SPs. For example, it will take more than 3 h to calculate the DA values for 10 million DUs.

An alternative method is to use a hash algorithm which allocates an average number of DUs to each server, instead of using the DA values of the DUs. This method assumes that all the DUs have the same amount of workload and is therefore simple and fast at design time, but it will incur the runtime overheads, as the system may spend most of its processing time on adjusting the workload of each server.

A possible improvement to GETS's static workload calculation and distribution is to use a hash algorithm for the initial load distribution and to use our current method to fine-tune the load distribution. However, some manual adjustment is still needed as the DA values cannot predict the workload for each data unit accurately.

6.1.2. Static versus dynamic data loading. According to our current design, GETS's data are loaded at the system initialization time. The problem with this static allocation is that most of the system initialization time is spent on loading the data into the servers' data buffers. For example, loading all the DUs for GETS may take tens of seconds or even several minutes, depending on the number of DUs. Since GETS cannot provide the service until its initialization is completed, shortening the data loading time can reduce the out-of-service time and therefore increase the system availability.

One approach to overcome this problem is to load the DUs dynamically on demand at runtime [7]. When a request arrives, the related data units are loaded into the related server and the request is processed. Our initial experiment shows that dynamic DU allocation only requires several seconds for each SP. Since GETS has been using a dynamic data loading method for its dynamic load balancing operation, it should be feasible to switch to the full-scale of on-demand, dynamic loading of the DUs.

6.1.3. Dynamic load balancing strategy. GETS's dynamic load balancing strategy is based on the DDA values of each server. Like the static load distribution strategy, this strategy is only feasible if we have the database reading and writing statistics. In the absence of such data, the PMI data, as originally supported by the WPF-enabled server clusters, might be an alternative.

6.1.4. Dynamic load balancing components. As stated in Section 4, the Load Balancer, Data Aggregator and HA Coordinator provide central services to all the servers on the cluster. Providing a single set of these components for a server cluster has two obvious drawbacks: First, it can cause a single point of failure to the cluster and second it can cause a bottleneck to the server performance. To overcome the first drawback, we have developed two sets of these three central components but only activate one set at runtime and leave another set on stand-by. If the activated set fails, the stand-by set will be put into operation. To ensure the state consistency, the stand-by set will start operation from the beginning of the current load balancing cycle, such that its Data Aggregator will re-collect the control data and the Load Balancer will re-analyze the system's load condition, and so on. Clearly, reloading these backup components will cause interruption in the dynamic load balancing operation, but it can prevent the failure of the system. During the history of GETS, there have only been very few times when the stand-by load balancing components had to be used. Hence, overall, the backup has hardly had any impact on GETS's performance.

For the bottleneck problem, one solution is to use multiple load balancers, one for each server [8]. Our reservation for such a solution is that it solves the bottleneck problem at the expense of additional server space, processing time and frequent inter-server communication. Currently, the bottleneck problem is not acute enough to comprise GETS's performance, but this problem will be monitored and a better solution will be sought to rectify this problem.

6.1.5. Testing and evaluation methods. We can more objectively evaluate how well our design has satisfied its QRs if we use a set of independent benchmark test results. For example, performance tests might be carried out by using TAO's benchmarking software [2], which tests system latency times under different load conditions. How to measure and evaluate QRs is itself a research challenge.

6.2. Lessons learned

In this section, we summarize some important lessons we have learned from the fulfillment of GETS's QRs.

(1) QRs are system-level requirements that cannot be assigned directly to individual system components; instead, they need to be planned at the infrastructure-level as a whole, with their design aspects then entrusted to system components. This is the most important lesson we learned from designing GETS. It is crucial that the development platform provides appropriate quality mechanisms for the application at hand. It is equally crucial that the QRs are carefully designed into the application system with a judicious use of these mechanisms.

(2) QRs are often interconnected and their realization in a system requires a collective and coordinated behavior of the system components and a system-level design strategy. For example, in order to support system performance, we have designed a dynamic load balancing strategy to ensure service interruption times to be minimal. We have also designed a static load distribution strategy to ensure that the related data are allocated to the same SPs or to the same servers. These two strategies have also helped to improve system availability by reducing the risk of system failure due to the imbalanced workload on the servers.

(3) QRs are application-specific and their fulfillment necessarily requires a specific design approach germane to their applications. The most challenging task in GETS's development is design. Sections 3 and 4 have articulated our design effort and showed how we have to develop application-specific techniques to solve design problems. For example, we have chosen to use both static and DDA values to work out a more accurate method for calculating GETS's static and dynamic workload. The application specificity is what makes the same set of QRs differ from one application to another.

(4) QRs are not only a design time concern, but most crucially a runtime concern. Satisfying QRs in a system means to design runtime strategies and mechanisms that can maintain and dynamically adjust the system's workload to avoid the violation of its QRs. GETS's static load distribution and dynamic load balancing strategies work in unison to achieve its QRs by ensuring that the servers' resources are efficiently utilized, the orders are processed

with the required speed and the system is partitioned with sufficient room to scale is also highly available.

7. RELATED WORK

Since GETS is a distributed system, its design should therefore be resonate in the design of distributed systems. Specifically, GETS's design is based on the work in the areas of distributed database systems, middleware-based load balancing and QoS-enabled systems. This section discusses some closely related work in these areas.

7.1. Partition-based, distributed database design

Designing a distributed database typically involves transformation of a pre-existed standalone database (either relational or object-oriented). There are three problems related to this transformation: data partitioning, data allocation and load balancing. These three problems also occur in our design and below we compare our approaches with some used in distributed database design.

Data partitioning is to divide a given standalone database into a distributed database. Assuming that the given database is relational, data partitioning is usually achieved by one of the three approaches [9]: (1) dividing a relation vertically into a set of small relations by projecting the original relation on one or more attributes; (2) dividing a relation horizontally into a set of tuples according to some selection criteria; (3) dividing a relation horizontally and then vertically or vice versa. There has been a continuous effort on developing optimization algorithms for vertical data fragmentation [10–12].

In GETS, data partitioning (called decomposition) is only applied logically, rather than physically. For business performance reasons specific to GETS, we have chosen the horizontal decomposition approach to divide the data into data units, each consisting of one or more tuples, selected according to some business criteria—each data unit shares the same trading equity symbol value. Our aim is to minimize inter-data dependencies.

In distributed database design, data allocation concerns allocating physical data fragments to different locations (sites) at design time, to ensure that the designed system has an initial balanced load. Many approaches have been developed for this task, including near-neighborhood, location-based and fuzzy allocation [13], all aiming at improving the performance of the distributed database systems.

GETS's data allocation has the same purpose as that of distributed databases; but it is done logically rather than physically by establishing the mappings from data units to SPs and to servers. To achieve a balanced initial state for GETS, we have adopted two strategies: to distribute data units equitably across all SPs and to allocate related DUs together. The latter strategy is similar to the near-neighborhood allocation approach [13].

In distributed database systems, data on different sites may become imbalanced at runtime due to data insertion or deletion. Several load balancing approaches have been developed to solve this problem. One approach, proposed by Copeland *et al.* [14], uses a simple heuristic to control data rebalancing: Data fragments on an overloaded site will only be moved to other sites if the time consumed on moving them is shorter than the extra processing time needed for the overloaded site. Another approach, developed by Hua and Lee [15], uses a Best Fit Decreasing strategy which minimizes data movement by moving smaller data fragments first and if possible, leaving larger data fragments untouched.

Our dynamic load balancing approach is similar to that of Hua and Lee [15]. A possible future improvement is to combine Hua and Lee's Best Fit Decreasing strategy with that of Copeland *et al.* [14], taking consideration of the extra processing time versus the overheads of the data movements in our approach.

Finally, there is an important difference between our data allocation and load balancing approaches and those of distributed database systems wherein we calculate the workload based on DA values, whereas transaction counts are usually used to measure the workload in distributed database systems.

7.2. Load balancing strategies for server clusters

The use of load balancing has extended beyond distributed database systems and has become a popular and effective solution to improve the performance of server clusters [2, 16–18]. For example, Othman *et al.* [2] present a set of strategies for CORBA middleware-based load balancing. They view the central role of a Load Balancer as making a balanced decision on which server will process which incoming request at runtime. These authors classify different load balancing strategies broadly into non-adaptive and adaptive. A non-adaptive load balancing strategy uses either a simple round-robin algorithm to allocate requests equally to each server or a randomization algorithm to select a server to handle a particular request. An adaptive load balancing strategy uses runtime information, such as the amount of the idle CPU time to decide which server to handle which request.

Based on an adaptive strategy, Othman *et al.* [19] developed a CORBA-compliant load balancing middleware, called ‘TAO’. TAO is a general-purpose Load Balancer for distributed systems, aiming at improving system scalability and overall system throughput. TAO uses an algorithm called ‘Least loaded’ to ensure that load differences between servers fall within a certain tolerance and select server with the least load to handle a request. Othman and Schmidt [20] elaborate TAO’s design into a set of design patterns.

In TAO’s terminology, GETS’s Load Balancer is also adaptive as it uses runtime information—real-time database access values (DDA values)—to make the balancing decision. GETS’s Load Balancer, however, differs from TAO in one principal way: In TAO, a request that is being processed is directly bound to its processing server and can therefore not be moved to other server. Consequently, TAO cannot rebalance the workload that is being processed on the servers. In GETS, however, since requests are associated with their SPs, they can be moved to a different server together with their SPs and the movement will not affect the client requests. With this flexibility, GETS’s Load Balancer can achieve two related purposes: (1) to ensure that each server has a balanced current workload (requests-in processing) and (2) to ensure that each incoming client request is directed to the server that has the least load. These two purposes, as described in Section 4, are closely related, whereas the second one is a direct consequence of the first one.

Other load balancing mechanisms are also briefly described here. For session-intensive applications, Dutta *et al.* [21] devised a load balancing approach called ‘Request Distribution for the Application Layer (ReDAL)’. This approach checks each incoming request to see whether it matches an existing session; if so, it will send the request to the corresponding application server, provided that the server is not overloaded. Otherwise, it will send the request to the application server with the lowest load. The load index, used by ReDAL to measure each application server’s workload, is the number of requests. However, different requests may consume different amount of system resources. Thus this load index is not always accurate to represent the actual workload.

Viswanathan *et al.* [22] proposed a set of Resource-Aware Dynamic Incremental Scheduling (RADIS) strategies to handle large volumes of computationally intensive and divisible loads for processing in grid systems. A divisible load (job) [23] is one that can be partitioned into arbitrary smaller fractions, such as very large data files in image processing or multimedia processing. Shah *et al.* [24] proposed two job migration strategies for the load balancing of grid systems, which are Modified Estimated Load Information Scheduling Algorithm (MELISA) and Load Balancing on Arrival (LBA). MELISA and LBA are, respectively, applicable to large-scale and small-scale grid systems. All these load balancing strategies are based on the assumption that load fractions/jobs bear no dependencies between each other, and thus can be assigned to any node. GETS’s Load Balancer, by contrast, is designed to process the requests with inter-dependencies.

For communication-intensive systems, Qin *et al.* [25] proposed a communication-aware load balancing scheme (COM-aware) to increase the effective utilization of networks in clusters. This scheme is specific to systems that require frequent inter-server communications. By contrast, GETS's design aims at minimizing the inter-server dependencies.

For agent-oriented systems, Jiang and Jiang [26] presented a task allocation and load balancing approach based on the contextual-resource negotiation. The number of allocated tasks on an agent is directly proportional to not only its own resources, but also to the resources of its interacting agents. Their approach allocates incoming tasks to agents based on the length of each resource's task queue, but ignores the fact that different tasks may lead to different workloads.

7.3. QoS mechanisms for distributed systems

In the recent years, the increase in Internet-based enterprise service systems and the advent of Web Service technologies have placed heavy demands on stringent QRs on system development. A large number of quality control mechanisms have been developed based on load balancing strategies [16, 27, 28]. For example, Garcia *et al.* [27] have extended the Load Balancer with a QoS controller for a Web Service server cluster. The goal of this controller is to improve the system response time. It uses an SLA (Service-Level Agreement) to determine what levels of service and response time should be delivered to which client groups. In TAO's terminology, this approach is non-adaptive, because it does not make use of system runtime information.

Gmach *et al.* [29] presented a QoS management mechanism (called AutoGlobe) to support SOA-based enterprise application development. AutoGlobe provides three quality control components: (1) A static resource management component which produces an optimally global allocation of services to servers via monitoring and periodically evaluates workload data of services; (2) a dynamic resource management component which handles the exceptional situations (e.g. failures and overload situations) at runtime and supervises services with a fuzzy logic-based controller; (3) a request scheduler to prioritize the requests of individual services based on a SLA. AutoGlobe and GETS are similar in their treatment of static and dynamic load balancing, but AutoGlobe's load balancing strategy ignores the inter-dependencies between the services. By contrast, in GETS, inter-service dependencies are crucial to both static and dynamic balancing.

Shankaran *et al.* [30] proposed the Integrated Planning, Allocation, and Control (IPAC) resource management architecture for distributed real-time embedded systems. IPAC provides (1) a fine-grained mechanism, which achieves the workload balancing on a system by fine-tuning the system's parameters, such as by decreasing the component execution rate to reduce the load of a certain component; and (2) a coarse-grained mechanism, which recovers the system from server failures by coarse-grained adaptation operations, such as adding or removing the components under execution. Different from the real-time embedded systems, the load of GETS's SPs is determined by the client requests. GETS itself is not able to reduce the load via decreasing the SP execution rate or other fine-grained operations. Hence, in GETS, both situations of load imbalance and server failure are handled by coarse-grained operations—that is, SP movements, as described in Section 4.

Saito *et al.* [8] described a design of a mail server system that aims at system manageability, performance and availability. Their work has some similarities with ours as it also considers initial load balance and automatic reconfiguration of client request dispatching. Their design, however, differs from ours as it provides each server with a Load Balancer which keeps track of the server load and makes the balancing decision independently. The problem of such a design is that it requires frequent communications among the servers and hence affects system performance. In contrast, GETS only uses one load balancer for the entire server cluster and can avoid the additional inter-server communication cost. To prevent the system from the single point of failure resulted by the Load Balancer, GETS has a backup Load Balancer stored in a server, as discussed in Section 4.

8. CONCLUSION

Although QRs have become a major drive in today's software development, there have been very few real-world examples in the literature that demonstrate how to meet these requirements. This paper provides such an example. Specifically, the paper describes a complete design process for a partition-based distributed stock trading service system called GETS and explains in detail every design task, its challenge and rationale. The paper shows how the design has satisfied a set of QRs related to resource utilization, performance, scalability and availability. Finally, the paper evaluates this design through detailed experiments and discusses a number of design alternatives and important lessons.

The main features of this design are the development of a static load distribution strategy and a dynamic load balancing strategy. The first strategy is to achieve an initial balanced workload on the system's server cluster during the system initialization time, whereas the second strategy is to maintain this balanced workload throughout the system execution time. Together, these two strategies work in unison to ensure that the server resources are efficiently utilized; the user requests are processed with the required speed; the application is partitioned with sufficient room to scale; and the system is highly available.

In conclusion, this paper has made an important contribution by showing in detailed steps how to fulfill a set of QRs in the design of a stock trading system. We believe that the principles and methods employed in this design are relevant to the development of a wide range of quality-based software systems and hope that we have provided you, the reader, with some key take-home points for you to apply the principles presented in this paper in a project of your own.

ACKNOWLEDGEMENTS

We are indebted to our three anonymous reviewers for their thoughtful and detailed comments and suggestions. Our special thanks are due to the second reviewer who has helped us to improve the quality of the writing by providing us with an excellent commentary on this paper. We also thank the editor of the journal for giving us the opportunity to improve the paper. This work was jointly supported by the State Street Cooperation in U.S.A., by the Fundamental Research Funds for the Central Universities and by the National 973 Fundamental Research and Development Program of China under the Grant 2009CB320701.

REFERENCES

1. Mehra A, Indiresan A, Shin KG. Structuring communication software for quality-of-service guarantees. *IEEE Transactions on Software Engineering* 1997; **23**(10):616–634. DOI: 10.1109/REAL.1996.563710.
2. Othman O, O'Ryan C, Schmidt DC. Strategies for CORBA middleware-based load balancing. *IEEE Distributed Systems Online* 2001; **2**(3). Available at: http://www.computer.org/portal/web/csdl/doi?doc=abs/mags/ds/2001/03/o3001abs.htm [5 October 2009].
3. Java2 Platform. Available at: http://java.sun.com/j2ee [5 October 2009].
4. WebSphere Partition Facility (WPF) Overview. Available at: http://publib.boulder.ibm.com/infocenter/wxddoc51/topic/com.ibm.wasxd.doc/WPFUserGuide.pdf [5 October 2009].
5. Winckler A. Scheduling of near-future workload in distributedcomputing systems. *Proceedings of IEEE Region 10 Conferences on Computer*, *Communication*, *Control and Power Engineering*. IEEE Computer Society Press: Silver Spring, MD, 1993; 169–172. DOI: 10.1109/TENCON.1993.319955.
6. Date CJ. *An Introduction to Database Systems*. Addison-Wesley: Boston, MA, 1990.
7. Apers PMG. Data allocation in distributed database systems. *ACM Transactions on Database Systems* 1988; **13**(3):263–304. DOI: 10.1145/44498.45063.
8. Saito Y, Bershad BN, Levy HM. Manageability, availability and performance in Porcupine: A highly scalable cluster-based mail service. *ACM Transactions on Computer Systems* 2000; **18**(3):298–332. DOI: 10.1145/1326561.1326569.
9. Runceanu A. Fragmentation in distributed databases. *Innovations and Advanced Techniques in Systems*, *Computing Sciences and Software Engineering* 2008; 57–62. DOI: 10.1007/978-1-4020-8735-6_12.
10. Sacca D, Wiederhold G. Database partitioning in a cluster of processors. *ACM Transactions on Database Systems* 1985; **10**(1):29–56. DOI: 10.1145/3148.3161.
11. Navathe SB, Ra M. Vertical partitioning for database design: A graphical algorithm. *ACM SIGMOD Record* 1989; **18**(2):440–450. DOI: 10.1145/67544.66966.
12. Ceri S, Pernici B, Wiederhold G. Distributed database design methodologies. *The IEEE* 1987; **75**(5):533–546.
13. Huang Y, Chen J. Fragment allocation in distributed database design. *Journal of Information Science and Engineering* 2001; **17**(3):491–506.
14. Copeland G, Alexander W, Boughter E, Keller T. Data placement in Bubba. *ACM SIGMOD International Conferences on Management of Data*. ACM: New York, 1988; 99–108. DOI: 10.1145/971701.50213.

15. Hua KA, Lee C. An adaptive data placement scheme for parallel database computer systems. *Proceedings of the 16th International Conference on very Large Databases*. Morgan Kaufmann Publishers: San Francisco, CA, 1990; 493–506.
16. Balasubramanian J, Schmidt DC, Dowdy L, Othman O. Evaluating the performance of middleware load balancing strategies. *Proceedings of the Eighth IEEE International Enterprise Distributed Object Computing Conference*. IEEE Computer Society: Washington, DC, 2004; 135–146. DOI: 10.1109/EDOC.2004.11.
17. Aleksy M, Korthaus A, Schader M. Design and implementation of a flexible load balancing service for CORBA-based applications. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*. CSREA Press: New York, 2001; 2140–2144.
18. Ho KS, Leong HV. An extended CORBA event service with support for load balancing and fault-tolerance. *Proceedings of the International Symposium on Distributed Objects and Application*. IEEE Computer Society: Washington, DC, 2000; 49–58.
19. Othman O, Balasubramanian1 J, Schmidt DC. The design of an adaptive middleware load balancing and monitoring service. *Proceedings of the Third International Workshop on Self-adaptive Software*. ACM Press: New York, 2003; 205–213.
20. Othman O, Schmidt DC. Optimizing distributed system performance via adaptive middleware load balancing. *Proceedings of the Third International Workshop on Self-adaptive Software*. ACM Press: New York, 2003.
21. Dutta K, Datta A, VanderMeer D, Thomas H, Ramamritham K. ReDAL: An efficient and practical request distribution technique for application server clusters. *IEEE Transactions on Parallel and Distributed Systems* 2007; **18**(11):1516–1528. DOI: 10.1109/TPDS.2007.1105.
22. Viswanathan S, Veeravalli B, Robertazzi TG. Resource-aware distributed scheduling strategies for large-scale computational cluster/grid systems. *IEEE Transactions on Parallel and Distributed Systems* 2007; **18**(10):1450–1461. DOI: 10.1109/TPDS.2007.1073.
23. Bharadwaj V, Ghose D, Robertazzi TG. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing* 2003; **6**(1):7–17. DOI: 10.1023/A:1020958815308.
24. Shah R, Veeravalli B, Misra M. On the design of adaptive and decentralized load-balancing algorithms with load estimation for computational grid environments. *IEEE Transactions on Parallel and Distributed Systems* 2007; **18**(12):1675–1685. DOI: 10.1109/TPDS.2007.1115.
25. Qin X, Jiang H, Manzanares A, Ruan X, Yin S. Communication-aware load balancing for parallel applications on clusters. *IEEE Transactions on Computers* 2010; **59**(1):42–52. DOI: 10.1109/TC.2009.108.
26. Jiang Y, Jiang J. Contextual resource negotiation-based task allocation and load balancing in complex software systems. *IEEE Transactions on Parallel and Distributed Systems* 2009; **20**(5):641–653. DOI: 10.1109/TPDS.2008.133.
27. García DF *et al*. A QoS control mechanism to provide service differentiation and overload protection to internet scalable servers. *IEEE Transactions on Services Computing* 2009; **2**(1):3–16. DOI: 10.1109/TSC.2009.3.
28. Zeng L, Benatallah B, Ngu AH, Dumas M, Kalagnanam J, Chang H. QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering* 2004; **30**(5):311–327. DOI: 10.1109/TSE.2004.11.
29. Gmach D, Krompass S, Scholz A, Wimmer M, Kemper A. Adaptive quality of service management for enterprise services. *ACM Transactions on Web* 2008; **2**(1):8. DOI: 10.1145/1326561.1326569.
30. Shankaran N, Kinnebrew JS, Koutsoukos XD, Lu C, Schmidt DC, Biswas G. An integrated planning and adaptive resource management architecture for distributed real-time embedded systems. *IEEE Transactions on Computers* 2009; **58**(11):1485–1499. DOI: 10.1109/TC.2009.44.