Journal of Applied Sciences

ISSN 1812-5654

Journal of Applied Sciences 9 (4): 689-697, 2009
ISSN 1812-5654
© 2009 Asian Network for Scientific Information

Java Versus .NET: A Comparative Analysis of Performance, Size and Complexity of Credit Card Authorization Systems

S. Hafizah Ab. Hamid, M. Hairul N. Md. Nasir and H. Hassan
Faculty of Computer Science and Information Technology,
University of Malaya, 50603, Kuala Lumpur, Malaysia

Abstract: This study presents a comparative analysis of the performance, size and complexity in both the Java and .NET platforms. Two identical prototypes of a credit card authorization engine were developed using the JAVA and .NET programming languages in order to measure and compare the performance of the authorization process as well as to measure and compare the size and development complexity of these two programming languages. The architecture of the singleton design pattern of a credit card authorization system using a multi-threading technique presented in this study supports the dynamic tuning of the size of the thread pool at runtime. It can be observed that the performance of the authentication engine in the .NET platform is slightly better than in the Java platform. Lines of Code (LOC) have been chosen as a metric to measure the size of the multi-threaded credit card authorization system whereby the total length of the multi-threaded credit card authorization system using a thread pool in .NET is 5048, while in Java it is 5199. The Cyclomatic Complexity number for the multi-threaded credit card authorization systems indicates that the .NET version is slightly less complicated than the Java version.

Key words: Multi-threaded, single-threaded, singleton design pattern, shared memory pool, efficiency

INTRODUCTION

Credit card authorization is a process whereby the card issuer decides whether to approve or decline requests to accept transactions performed by cardholders. The approval of the transaction is based on a series of validations of the card's risk management profile to verify that the cardholder's account is open, the transaction amount is within the available credit limit, the transaction is coming from the legitimate card and many other card-related validations. The validation of the card's risk management profile can be classified into two categories, namely card restriction validation and online fraud validation.

Card restriction validation includes the financial and non-financial verification related to the card whereas online fraud validation involves cryptographic operation through the Host Security Module (HSM) to verify the security aspect of the authorization in order to determine the legitimacy of the card. The HSM is an external device connected to the authorization host that keeps the card issuer's secret information in tamper-resistant hardware, which is used to perform the verification of the credit card transaction. Due to various validations being carried out in the authorization system for each credit card transaction, the authorization process will take some time to be completed. Moreover, slow and expensive input/output operations during the card restriction and online fraud validation through the database and HSM also cause some delays in the authorization process. Besides that, the validation of the security aspect of the card itself also takes some time to process due to the complexity of the algorithms involved. With the old payment processing methods of the conventional system, the credit card transaction takes longer during the authorization processing. As a result, the performance of this type of authorization system is affected whenever the number of authorization processes increases. This causes some of the simultaneous authorizations accepted at a single point in time not to be able to respond within the allowed timeframe. Those failure transactions are classified as timed out in the context of electronic financial services.

In this project, a multi-threaded authorization system is developed to improve the response time of the credit card authorization process and to overcome the slow sequential authorization processing of the single-threaded model of current credit card authorization systems. This

Corresponding Author: Mohd Hairul Nizam Md. Nasir, Faculty of Computer Science and Information Technology, University of Malaya, 50603, Kuala Lumpur, Malaysia
Tel: +603-79676435 Fax: +603-21784965

multi-threaded authorization system is constructed in two different platforms, namely .NET and Java, to illustrate the different implementation approaches to this model in these platforms. These platforms are chosen to evaluate the complexity of multi-threading implementation using modern programming languages. Moreover, these languages are the architecture that should be deployed in modern credit card authorization systems. Apart from that, this project also compares the performance as well as the effort required to implement the multi-threading technique in the credit card authorization system in both platforms. Throughout this project, multi-threading implementation enables multiple tasks to be carried out concurrently during the authorization process. Multiple threads are used to perform different validation processing tasks and they save the idle time of waiting for a reply from the slow I/O events while the other threads can still carry out their own processing. This can speed up the throughputs of the process and also reduce the number of timeout transactions, which ultimately optimizes the cost of the authorization service.

According to Norton and DiPasquale (1997), a thread is an independent flow of control within the process and it has its own sequence of instructions to execute. A thread would share the same address space as the process. Each process contains at least one thread and the initial thread is created automatically by the system when the process starts up. In the multi-threading technique, multiple actions can be performed in a program that leads to multitasking. Therefore, several operations derived from the card's risk management profile can be executed concurrently within the single memory space of the process and all the spawned threads can still share the same system resources during the authorization process of the credit card transaction. Multi-threading will not only enable multiple simultaneous authorization requests to be processed in less time but also optimize the system resources while waiting for slow I/O operations in an authorization process. Apart from that, the multi-threading technique can provide a better response for the user (Broberg *et al*., 2001). It allows the user to perform other tasks simultaneously while running the authorization process in the background.

There are two approaches to multi-threading implementation, which are either through the kernel thread or the user thread. The significant distinction between these two approaches is related to the context switching between the multiple threads running in the process. Context switching is the scheduling scheme to accommodate the resources for both threads where one thread's execution is suspended and swapped off and another thread is swapped onto and its execution is resumed. Through the kernel thread, the context switching is pre-emptively scheduled by the kernel while the context switching of the user thread is tailored based on the application without any interaction with the kernel.

In this project, one-to-one mapping of the user thread to the kernel thread is chosen in the multi-threading technique because it is more efficient compared with the typical user thread. The typical user thread does not increase the percentage of CPU time that the operating system gives to a process (Krisztian *et al*., 2000). As a result, the performance of the system is degraded when user threads cannot be executed while the kernel is busy. Apart from that, it is simpler to use this type of thread because all the aspects of thread management are handled by the operating system kernel and, furthermore, writing a good thread management for user threads is complicated.

There are several basic operations used to control a thread in this project. Like a process, a thread is created, runs and is only deleted after it completes its execution. Since the authorization process is considered a short life task and also has a high number of requests at any single of time, there are two types of thread models that can be used in conjunction with the multi-threading technique. These models are thread-per-request and a thread pool. Thread-per-request creates a brand new thread for each task and once the thread has finished with the task, the thread is disposed of, whereas, through the thread-pool model, a thread is pulled from the pool of threads and assigned to the task. Upon completion of the task, the thread will add itself back to the pool to wait for its next task assignment.

In this project, the thread-pool model is chosen to handle the authorization process of the credit card system because it saves the work of creating brand new threads for this kind of short-lived task as well as minimizing overheads associated with getting a thread started and cleaning it up after its termination. By creating a pool of fixed worker threads, each worker thread from the pool can be recycled over and over again for subsequent tasks. With this model, the response time of the authorization process is also reduced because the worker thread is already started and it is simply waiting to be assigned to a task.

When multiple threads are running together, they will invariably need to communicate with each other in order to synchronize their execution. The use of synchronization can ensure consistency and access to the shared data. In this project, specific classes, namely queue, mutex and timer, have been applied apart from the platform-dependent thread life cycle and communication methods, to ensure safe threads are implemented into the

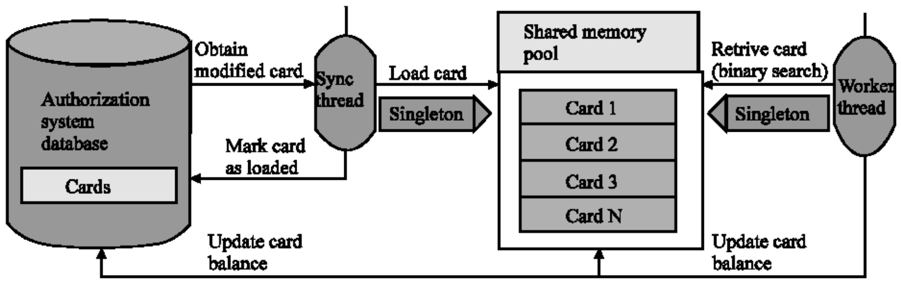

Fig. 1: Singleton design pattern-Shared memory pool

multi-threaded authorization engine as well as to avoid hitting the common pitfall of concurrency controls that leads to an undesirable performance penalty. With the thread synchronization method, the thread-pool implementation shows tremendous speed improvement compared with the thread-per-request implementation.

Since the aim of the authorization system is to allow authorization to be processed in less time, the card information has been loaded from the system database to the shared memory pool of the system to accelerate the process of card restriction validation. This is performed to enable the authorization thread to obtain card information from the shared memory pool through a binary search during the authorization process that could help in saving time instead of reading similar information from the system database involving an expensive I/O operation. In order to allow multiple threads to access the only shared memory pool, a singleton pattern is applied in this project. This design pattern allows only a single instance of a class to be created during runtime and this instance could be reused and made accessible to all the other objects.

Through the singleton design pattern as shown in Fig. 1, a class is constructed with only one instance that can be accessed globally within the multi-threaded credit card authorization system. The singleton design pattern is applied to the card object, which is acting as a shared memory pool that holds all the cards' information for authorization purposes. When the listener service is activated, the listening thread will load all the information of the cards into random access memory through a configurable array. After an authorization has been received, a worker thread will obtain the only instance of the cards' object and perform a binary search through the related array of the cards' objects in order to retrieve the information of the card related to the transaction from the shared memory for authorization purposes. In this project, a separate synchronization thread is initialized in the background of the authorization engine to browse the system database for any modified card information requiring updating into the shared memory pool. Once the modified card information has been loaded to the shared memory pool, the synchronization thread will update the system database to mark that the card has been processed.

A singleton design pattern is applied to ensure that all the workers threads can access the shared memory pool for card information during authorization. Without a singleton design pattern, shared memory pool implementation is not possible in an object-oriented environment. Through the shared memory pool, the access time is faster and, hence, improves the response time of the credit card authorization process.

MATERIALS AND METHODS

The research study was conducted from March 2007 until August 2008 at the Faculty of Computer Science and Information Technology, University of Malaya. Data gathering has been carried out to collect information from various sources such as books, journals, articles and reports from libraries as well as the Internet. Most of the useful information was obtained from the ISI Web of Knowledge, IEEE, ACM, SpringerLink, ScienceDirect, SCEAS System, digital libraries, current solution provider manuals, payment system manuals and online encyclopedia to accomplish the initial research work. The acquired information was used to determine the project's significance, plan, definition and scope. Analyses of the preliminary resources and testing strategies required in this project were identified and defined. Through the project plan and definition, the effort of the project could be determined and identified.

With the multi-threading implementation approach, the authorization system will assign several threads to perform the processing of the card's risk management profile for each credit card authorization. The singleton design pattern was also revised to confirm that the software is implemented accordingly. Evaluation has been conducted to measure and compare the performance, size and complexity between Java and .NET in the credit card authorization process. The performance of the authorization system using a shared memory instead of a database during credit card authorization processing was evaluated. The information regarding the size and the effort required to implement a multi-threaded authorization system in these platforms was also evaluated.

Multi-Threading Implementation in .NET and Java .NET and Java classes generally provide almost the same functionalities from the developer's perspective, although the underlying implementation of threads at the operating system level may be different, as it is not specified in the run-time specifications. In this project, there are five aspects of the implementation approach to multi-threading in .NET and Java that will be discussed. These aspects are thread creation, thread state, thread management, thread synchronization and thread pool.

Thread creation: There are two main techniques for creating a thread in Java, either through the Thread subclass or by implementing the Runnable interface. Both techniques require a Run method to be defined so that the new thread can execute this method during its creation. However, the preferred method is to implement the Runnable interface because it can be used to get around the lack of multiple inheritances. In .NET, a thread is created through the ThreadStart delegate. The ThreadStart delegate requires any void method that takes no parameter as a reference method to be executed by a new thread during its creation.

In this project, .NET provides more a flexible approach than Java in terms of thread creation. However, .NET does not allow the passing of parameters from the main thread to a new thread through this reference method, which has constrained the .NET capabilities against the Java approach. Although this constraint in .NET could be overcome by the instantiation of a brand new object every time to pass those parameters to the new thread for processing, it increases the size and also the complexity of the program.

Thread state: A new thread will progress through several states during its life cycle. In Java, these states are new, active, inactive, suspended and dead (Sun Microsystems, Inc., 2004). A thread is new when it has not done anything yet. A thread is active when it is running and occupies a processor at the current time. A thread is inactive when it is running but it does not actually occupy a processor at the current time. A thread is suspended when it is not able to run even if it were given some CPU time. A thread is dead when it has completed the execution. In .NET, these states are unstarted, running, waitsleepjoin, suspendrequested, suspended, abortrequested, aborted and stopped (Microsoft Corporation, 2007). A thread is unstarted when the start method has not been invoked on the thread. A thread is running when the thread has been started and not blocked. A thread is waitsleepjoin when it is started and blocked. A thread is suspendrequested when it is being requested to suspend its execution. A thread is suspended when it has been suspended. A thread is abortrequested when it is being requested to abort. A thread is aborted when it has been aborted after being requested. A thread is stopped when it has finished its execution.

In this project, .NET offers more methods to control the additional states as opposed to Java. Although Java does provide those methods as offered by .NET, most of them are deprecated methods that could leave the objects in inconsistent states when the methods are invoked, such as the destroy, suspend, stop and resume methods. In .NET, this is not the case because the runtime allows the thread to run until it reaches a point where it may be safe. However, a suspended thread that still maintains any locks could probably lead to deadlock when it is not implemented appropriately. Therefore, those deprecated methods are not implemented in this project to avoid undesirable results, although these methods are considered as thread-safe in .NET. Alternatively, other methods are used to provide similar functionalities as per those deprecated methods, such as using a volatile flag variable to terminate the execution of a thread in a graceful manner instead of using the abort method. Also, methods such as wait, sleep and join are used to substitute the suspend or resume methods to be implemented in the multi-threaded application.

Thread management: Java defines a few methods on the object class to manage threads, namely wait, notify and notifyAll (Sun Microsystems, Inc., 2004). The wait method causes the current thread to wait until another thread invokes the notify or notifyAll method for the object. The thread must own the object in order to call wait. The current thread will place itself in the wait set for this object and then release all synchronization claims on this object. The thread becomes disabled for thread scheduling purposes and lies dormant until either another thread invokes the notify method for this object, another thread invokes notifyAll for this object, another thread interrupts the thread or the specified amount of time has elapsed. The corresponding .NET versions are Monitor.Wait, Monitor.Pulse and Monitor.PulseAll (Microsoft Corporation, 2007). In this project, those methods have been used to signal both the worker or child threads when there is a new authorization job received from the payment gateway.

In Java, the Timer and TimerTask classes can be used to allow the scheduling of tasks for future execution and these tasks may be scheduled for one-time execution or for repeated execution at regular intervals (Sun Microsystems, Inc., 2004). Corresponding to each timer object is a single background thread that is used to execute all of the timer's tasks sequentially. To terminate the execution of a timer's task, the timer's cancel method can be invoked. In .NET, timer objects are lightweight objects that enable the developer to specify a delegate to be called at a specified time (Microsoft Corporation, 2007). A thread in the thread pool performs the wait operation pending the execution of the callback method. To cancel a pending timer, the Timer.Dispose method can be invoked. In this project, those methods are used to execute the timeout action when the authorization has exceeded the permitted execution time frame.

In general, both .NET and Java provide a higher level of abstraction for threaded applications where the developer does not have to be concerned with the lower-level details of thread management. Besides that, both .NET and Java offer similar functionalities in aspects of thread management and the thread timer. The only difference in thread management is that, in Java, these methods are contained in the object base class, whereas in .NET, they are contained in the Monitor class. For the thread timer, Java uses the abstract class TimerTask to reference methods for scheduling the execution at preset times, whereas the .NET class uses delegates for referencing methods and TimeSpan objects for referencing time, which offer the same capabilities.

Besides that, the thread prioritization has been used to set the listener to have a higher priority than other threads in both platforms in this project. In Java, the runtime allows for more priority levels than .NET. However, neither of the platforms guarantees absolute thread priorities but both are considered as optimization methods. Therefore, there is no difference in thread priority assignment in .NET and Java because both platforms provide similar functionalities.

Thread synchronization: In Java, a mutex is a locking mechanism guaranteed to be atomic. Only one thread can access a mutex at a time and each mutex is associated with every object instance. To request a lock, a synchronized keyword could be used to guarantee that only one thread will be executing the synchronized code at a time. In .NET, there is no synchronized keyword but Monitor.Enter and Monitor.Exit are used to request a lock. If calls are nested, the number of occurrences of the Exit invocation must match the number of times Enter is invoked as a count is maintained. The advantage of .NET is that it provides the Monitor.TryEnter method that attempts to acquire a lock after a pre-defined time has elapsed. Apart from that, .NET also provides a mutex locking mechanism. The similarity between a monitor and a mutex in .NET is only that a thread can own the lock at any given point in time. The difference is that a mutex is not restricted to a single process but any thread in the system is allowed to own the mutex. In .NET, the WaitHandle class is used to handle mutex manipulation and a thread can wait for multiple mutexes. In this project, these methods have been used to synchronize the thread execution when saving the record into the database or writing the authorization log into the file.

Another facility available in .NET with no corresponding Java functionality is Interlocked access. The Interlocked methods provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. The Increment and Decrement functions combine the operations of incrementing or decrementing the variable and checking the resulting value into one atomic operation. The Exchange function atomically exchanges the values of the specified variables. The CompareExchange function combines two operations, namely comparing two values and then storing a third value in one of the variables based on the outcome of the comparison (Microsoft Corporation, 2007). In this project, this method has been used in the .NET platform to synchronize the execution of all the child threads during online fraud validation and also to monitor the status of child threads during online fraud validation so that it only proceeds with the authorization processing when all those threads have completed their own specific validation. In Java, there is no such method available and, thus, common static variable is used to provide a similar functionality.

In .NET, there is another special locking facility that is not available in Java, namely ReaderWriterLock. This locking mechanism allows multiple threads to read a resource concurrently, but requires a thread to wait for an exclusive lock in order to write to the resource. In cases where most accesses are reads and writes are infrequent and of short duration, ReaderWriterLock provides better throughput than a simple one-at-a-time lock. In this project, this method has been used in .NET to write the information to the shared memory pool, whereas in Java, a synchronized keyword is used to handle the kind of functionality since this method is only available in .NET.

In general, .NET provides more fine-grained control over concurrency access of multiple threads than Java. Hence, it allows for better performance in certain situations when using facilities like Interlocked and ReaderWriterLock in .NET compared with Java.

Thread pool: A facility specific to .NET is the ThreadPool class. There is no corresponding functionality included in Java prior to JDK1.4.2 and the developer will have to roll out his or her own pool. The .NET ThreadPool can be used to make much more efficient use of multiple threads without having to instantiate a thread object for each method call. Using thread pooling provides a pool of worker threads that are managed by the system and it enables the system to optimize for better throughput for this process. When a task is complete, a thread from the pool executes the corresponding callback method.

In this project, a custom thread pool has been implemented in both platforms although a ready-to-use class is provided. This is because there is no way to cancel a work item after it has been queued in the ThreadPool class, whereas the custom thread pool provides better management and control over all the worker threads in the pool. Although the custom thread pool is chosen in .NET, it is still simpler to use the queue class in .NET because a lot of the required methods are provided in its API without a need to rewrite those methods from scratch as compared with Java. These methods include obtaining the number of work item counts in the queue as well as managing multiple thread synchronization access to work items in the queue.

RESULTS AND DISCUSSION

Response time, which is the key component that reveals the performance of the authorization process, is the amount of time taken to perform the risk assessment during the authorization process of the credit card transaction. Since response time will determine the performance of the authorization process, embedded testing tools have been incorporated as part of the authorization engine in this project for both the .NET and Java platforms to collect the timestamp before and after running through the identical set of risk management profiles. The timing testing is carried out to obtain the timestamp of the transaction running through the single-threaded authorization engine against a similar transaction going through the multi-threaded authorization engine in both platforms. The other performance testing is to evaluate the response time of the credit card transaction using the authorization system accessing the shared memory pool for authentication data against the authorization system accessing the system database for authentication data. The result of the testing, which is recorded in terms of response time, will determine the performance of the system.

System performance: The performance is measured by evaluating the response time taken to process a batch of authorizations in both the Java and .NET versions of the multi-threaded credit card authorization system (Table 1). The response time was measured using the embedded testing tools that were built in as part of both authorization systems and the payment gateway to obtain the time taken before and after the transaction was sent and received. The measurement unit for the response time was seconds.

The testing was carried out to access the response time of a group of authorizations performed one after another using the multi-threaded authentication engine accessing the shared memory pool for authentication data and the multi-threaded authentication engine accessing the system database for authentication data. This authorization will be sent upon receiving a response from the previous transaction and there is no simultaneous authorization performed. The number of worker threads and child threads that were used in the multi-threaded authorization of the credit card system is also similar: three and nine, respectively. The test result is recorded based on the best response time taken in five attempts for each category. This is done to minimize the impact of the context switching between multiple threads running in the system over the result obtained and to ensure the accuracy of the testing performed.

Based on the test result as in Table 1 above, it has been confirmed that the performance of the multi-threaded authentication engine is better than the single-threaded authentication engine in both the .NET and Java platforms. Besides that, it can be seen that the performance of the authentication engine in the .NET platform is also slightly better than the Java platform in a

Table 1: Test result of multi-threaded and single-threaded authentication engine

	Best response time (sec)			
	Multi-threaded authentication engine		Single-threaded authentication engine	
No. of sequential authorizations	.NET	Java	.NET	Java
10	4.7	5.5	8.1	9.5
20	9.4	10.5	16.5	19.0
30	14.2	15.9	24.8	28.7
40	18.8	21.0	33.0	38.0
50	23.4	26.7	41.1	47.8
60	28.2	32.7	49.4	56.9
70	32.7	37.2	57.7	66.9
80	37.6	41.9	65.9	76.4
90	42.1	47.5	74.7	86.8
100	46.9	53.2	82.3	95.7

Windows environment. From the result, it can be seen that the performance of the multi-threaded authentication engine is almost double that of the single-threaded authentication engine in both platforms.

Software size: According to Ramasubbu and Balan (2007), the eventual product is built by lines of code in a programming language. Therefore, the line of code is a possible measure for the size of the product due to being easily understood, easily measured, amenable to automate, having language dependencies, the need for minor common ground rules, being applicable to the entire life cycle and penalized compactness.

In this project, Lines of Code have been chosen as the metric to measure the size of the multi-threaded credit card authorization system using a thread pool in both the .NET and Java platforms. In order to accomplish this measurement, the model recommended by Fenton and Pfleeger (1998) has been used as reference. LocMetrics, which is an external tool, has been used to aid in the calculation. The formula to calculate Lines of Code is defined as below.

$$\text{Total Length (LOC)} = \text{Non-Comment Lines (NLOC)} + \text{Comment Lines (CLOC)} + \text{Blank Lines (BLOC)}$$

Based on the calculation, the total length (LOC) of the multi-threaded credit card authorization system using a thread pool in .NET is 5048 whereas in Java is 5199, as displayed in Table 2. Both programs are considered as medium-size applications. On the other hand, Table 2 shows that in .NET, the size of the program is slightly smaller because the non-commented lines of code are slightly fewer than in Java. The possible reason for this is because there are more ready-to-use methods in .NET as compared with Java. Therefore, the effort used to develop the program in .NET is reduced.

Software complexity: According to McCabe (1976), more complex software requires more effort to carry out testing as it is prone to error. In this project, McCabe's Cyclomatic Complexity is used to measure the complexity of the multi-threaded credit card authorization system in both platforms. This is because it is one of the most widely used and accepted members of a class of static software metrics. It measures the number of linearly independent paths through a program module. This measure provides a single ordinal number that can be compared with the complexity of other programs and it is intended to be independent of language and language format. LocMetrics, which is an external tool, has been used to aid in the calculation. The cyclomatic number is calculated as per the formula below, where F is the flow graph of the code, e is the number of arcs of the flow graph and n is the number of nodes of the flow graph.

$$v(F) = e - n + 2$$

Based on the result from Table 3 the Cyclomatic Complexity number for the multi-threaded credit card authorization system using a thread pool in .NET is 537 units whereas in Java it is 546 units. This indicates that the .NET version of the multi-threaded credit card authorization system is slightly less complicated than the Java version. The difference is due to some of the synchronization methods in .NET already being built-in as part of the standard libraries and, therefore, the effort required to implement the .NET version is reduced.

Based on the Lines of code measurement, the size of the multi-threaded credit card authorization system in .NET is slightly smaller than the similar multi-threaded credit card authorization system in Java. When the size is smaller, the total memory usage required to load the program is less. In this project, the total memory usage of the .NET version of the multi-threaded authorization system after starting up is 22 megabytes, whereas the initial total memory usage of the Java version of the multi-threaded authorization system when started up is 32 megabytes. Also, based on the performance results, the .NET version of the multi-threaded credit card authorization system is slightly better than the Java version because the number of CPU cycles to run the .NET version is lower due to the size of the program in the .NET version being smaller. Besides that, the effort required to develop the multi-threaded authorization of the credit card system in .NET is also reduced when the size of the program is smaller compared with a similar application in Java.

Apart from that, the .NET version of the multi-threaded credit card authorization system is also slightly less complicated compared with the Java version. Based on McCabe's Cyclomatic complexity measurement, the

Table 2: Lines of code metric

Platform	Total length (LOC)	Non-comment lines (NLOC)	Comment lines (CLOC)	Blank lines (BLOC)
.NET	5048	4519	320	209
Java	5199	4715	236	248

Table 3: Cyclomatic complexity metric

Platform	Total No. of arcs of the flow graph (e)	Total No. of nodes of the flow graph (n)	Cyclomatic complexity No. v(F)
.NET	811	276	537
Java	828	284	546

number of paths of the multi-threaded authorization system developed in .NET is smaller than the number of paths of the similar application written in Java. This is because some of the methods of the multi-threading techniques are provided as part of the built-in standard library in the .NET framework, such as Interlocked and ReaderWriterLock. These two examples are considered as optimizer methods provided by the .NET library for inter-thread communication, which could not be found in the Java library. Since the number of paths of the multi-threaded authorization system written in .NET is smaller, the number of CPU cycles to run the application is also reduced. Therefore, the performance of the multi-threaded authorization system in .NET is better than a similar Java version of the multi-threaded authorization system. With additional optimization methods provided by the .NET framework, it will not only increase the speed of the processing but also accelerate the work of producing the multi-threaded application in .NET. The effort is also minimized because some portions of the code are written using the standard library provided in .NET instead of writing similar code from scratch as in Java.

CONCLUSIONS

Nowadays, credit cards are growing in popularity around the world as a form of payment. Current credit card authorization was developed on a single-threaded model whereby the authentication process takes longer to respond due to the sequential process of the card's risk management profile and its limitation of handling a huge number of simultaneous transactions. As a result, the performance of the authorization system was affected during peak hours. This paper presented a comparative analysis of performance, size and complexity in both the Java and .NET platforms in order to provide a solution to improve the response time during the authorization process.

This research provides a solution to optimize the performance of the credit card authorization system through the multi-threading technique in the Java and .NET platforms. This technique enables the authorization of credit card transactions to be processed in a shorter amount of time. From the business point of view, a fast and reliable authorization process will generate more revenue for the organization, whereas from the customer point of view, the authorization process in time builds the confidence of the cardholder in using the credit card as a payment method. In short, this project provides a win-win situation for both the organization and the community as both parties will gain the benefits of implementation of a multi-threaded credit card authorization system. Besides that, the multi-threaded credit card authorization system implemented in this project enables several tasks related to the card's risk management profile validation to be executed concurrently during the authorization process. This will not only provide a better response time for the authorization process but it will also enable more credit card transactions to be processed in a multi-threaded authorization system in a shorter amount of time.

The architecture of the singleton design presented in this project supports the dynamic tuning of the size of the thread pool running at runtime. The number of fixed worker threads and child threads can be adjusted to ensure the utilization of the multiple threads to their optimal level. This is implemented to ensure the capacity of the thread pool matches the necessities of the application based on the estimated volume and velocity of the credit card transactions processed in a specified period.

A shared memory pool is also used in conjunction with the multi-threading technique. Since multiple threads are running in a single process space, a shared memory pool is implemented to keep all the card information that will be used for the credit card authorization process in the random access memory area. This is implemented to allow the authorization process to access the shared memory pool for card information, which is faster than accessing similar information from the system database because it involves a less expensive I/O operation. For this reason, a synchronization thread is introduced to maintain the information in the shared memory pool so that any update in the system database will be reflected in the shared memory pool. Through shared memory implementation, the response time of the authorization process is further improved.

Performance testing has been used to evaluate the response time of the authorization process under different circumstances. The response time was measured using the embedded testing tools that were built in as part of both authorization systems and the payment gateway to obtain the time taken before and after the transaction was sent and received. The measurement unit for the response time was seconds. The performance of the authorization system using the shared memory instead of the database in both the Java and .NET platforms during credit card authorization processing was also evaluated.

Apart from that, the size and complexity was measured to determine the amount of work required in software development, especially in testing. The Cyclomatic Complexity number for the multi-threaded credit card authorization system indicates that the .NET version of the multi-threaded credit card authorization system is slightly less complicated than the Java one. Another unique feature implemented in this project is that any changes made to the card are recorded for audit purposes. Both the multi-threaded credit card authorization systems implemented in this project can accept multiple connections from the payment system at a single port number. This is implemented to allow more simultaneous authorizations to be received through these multiple links for load balancing usage in future.

ACKNOWLEDGMENTS

First and foremost we would like to express our gratitude to the Almighty, who gave us the possibility to complete the research work successfully. Secondly, we would like to forward our deepest thanks to our colleagues, lecturers and technical staff from the Department of Software Engineering for their endless assistance, technical advice and co-operation.

REFERENCES

Broberg, M., L. Lundberg and H. Grahn, 2001. Performance optimization using extended path analysis in multithreaded programs on multiprocessors. J. Parallel Distribut. Comput., 61: 115-136.

Fenton, N.E. and S.L. Pfleeger, 1998. Software Metrics: A Rigorous and Practical Approach. 2nd Edn. PWS Publishing, Boston, ISBN-10: 0534954251, pp: 320-336.

Krisztian, F., U. Rich, R. Steve and M. Trevor, 2000. Parallelism and interactive performance of desktop applications. Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-IX, 2000, Cambridge, Massachusetts, USA., pp: 129-138.

McCabe, T.J., 1976. A complexity measure. Software Engin., IEEE Trans., 2: 308-320.

Microsoft Corporation, 2007. Microsoft developer network, viewed 15 September 2007. http://msdn2.microsoft.com/en-us/library/system.threading.aspx.

Norton, S.J. and M.D. DiPasquale, 1997. Multithreaded Programming Guide. 1st Edn., Prentice Hall, New Jersey, USA., ISBN-10: 0131900676.

Ramasubbu, N. and R.K. Balan, 2007. Globally distributed software development project performance: An empirical analysis. Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium of the Foundations of Software Engineering, 2007 ESEC-FSE ACM, NY., pp: 125-134.

Sun Microsystems, Inc. 2004. JavaTM 2 platform standard Ed. 5.0, viewed 06 September 2007. http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.State.html.