Performance Report

[SDA1]
SWEN90007 SM2 2023 Project

In charge of: [Chengyi Huang chengyih@student.unimelb.edu.au,
Hanming Mao: hanmingm@student.unimelb.edu.au,
Yuxin Liu: yuxliu20@student.unimelb.edu.au,
Ziqiang Li: ziqiangl1@student.unimelb.edu.au]

SCHOOL OF
COMPUTING &
INFORMATION
SYSTEMS

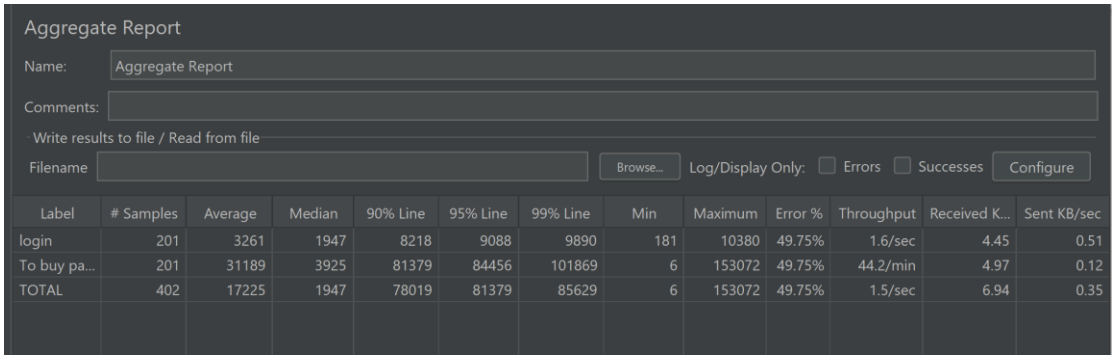
Contents

Design patterns

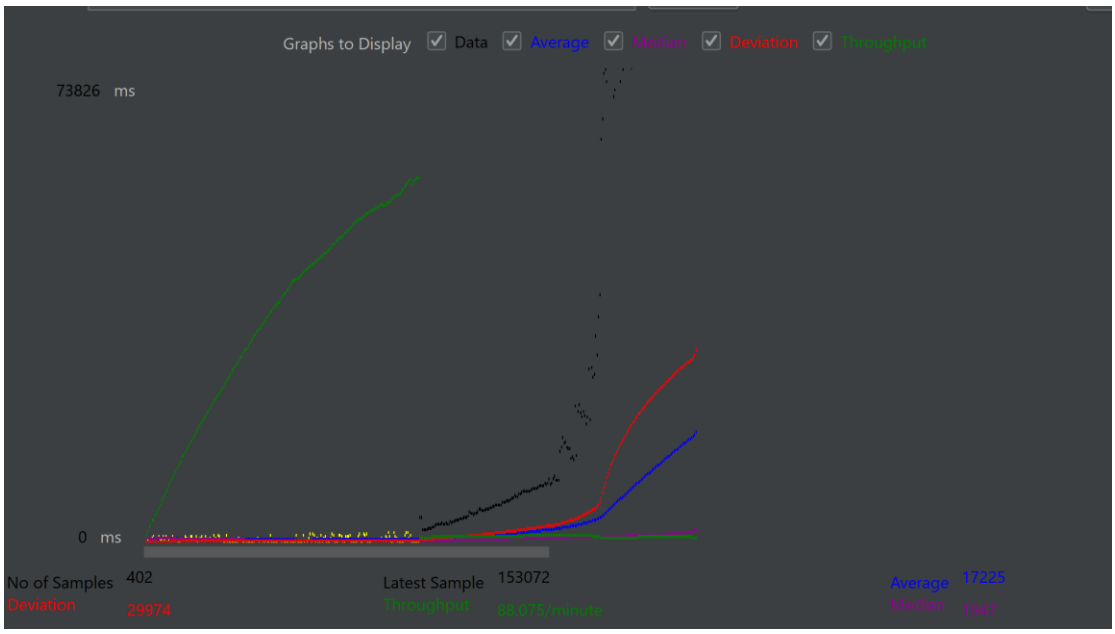
Identity map

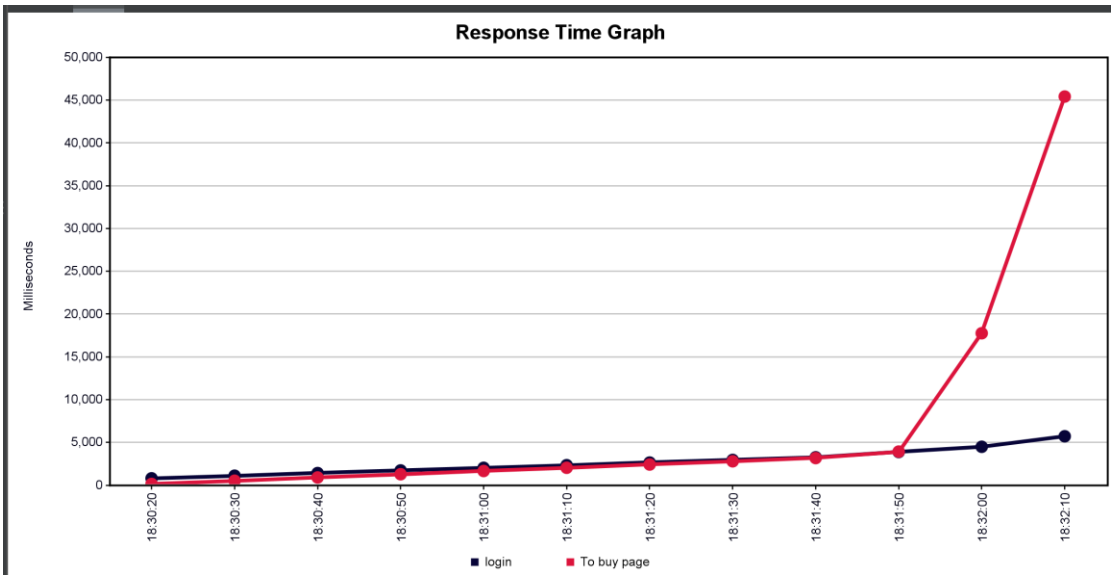
The system we now have does not use an identity map in the code, so for most reading operations, the system always gets the data directly from the database. As we use the online database, the data loading time is considerably long so for now the system's most display functions will take several minutes to show the records.

Take the book ticket page as an example, while navigating to this page, the system gets the specific eventId from request and uses this eventId to request the event's detail from the database. Images shown below are the performance of the book ticket page navigation threads.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received K...	Sent KB/sec
login	201	3261	1947	8218	9088	9890	181	10380	49.75%	1.6/sec	4.45	0.51
To buy pa...	201	31189	3925	81379	84456	101869	6	153072	49.75%	44.2/min	4.97	0.12
TOTAL	402	17225	1947	78019	81379	85629	6	153072	49.75%	1.5/sec	6.94	0.35





The graphs shown here are the result of 100 threads trying to enter the booking page for the same event. As we can see, the throughput is high when all threads try to login to the system but it goes down dramatically to near zero when all threads try to navigate to the book ticket page. The data sent per second of the booking ticket page is much less than that of the login page. The throughput of the booking ticket page is 44.2/min which is 0.737/sec, almost half of the throughput of login. The average time of the booking ticket page is 9 times greater than the login page.

If we add an identity map to this booking ticket page, the system will only interact with the database in the first navigation request (For now, we consider the navigation request separately which means we do not take other users to change this event's data into consideration.). After receiving the event detail data from the database, the system keeps this data record in a map with the eventId as identity. For all following book ticket page navigation requests of this specific event, the system checks that there is an object in the map and can load the data directly from the memory which will highly increase the system's throughput and the display time to even faster than login.

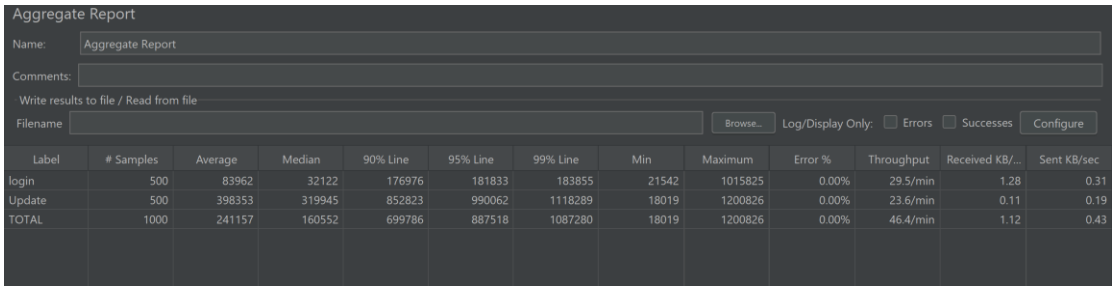
Unit of work

Our system uses the unit of work pattern for multiple operations like updating events, adding events, booking tickets, etc. However, as we incorrectly understand the unit of work's mechanism and our insufficient understanding of the business transaction, most of our unit of work only contains an operation of an object instead of containing all object-related operation in one unit of work. Thus, our system shows no significant performance improvement even though we use the unit of work for almost all operations except reading.

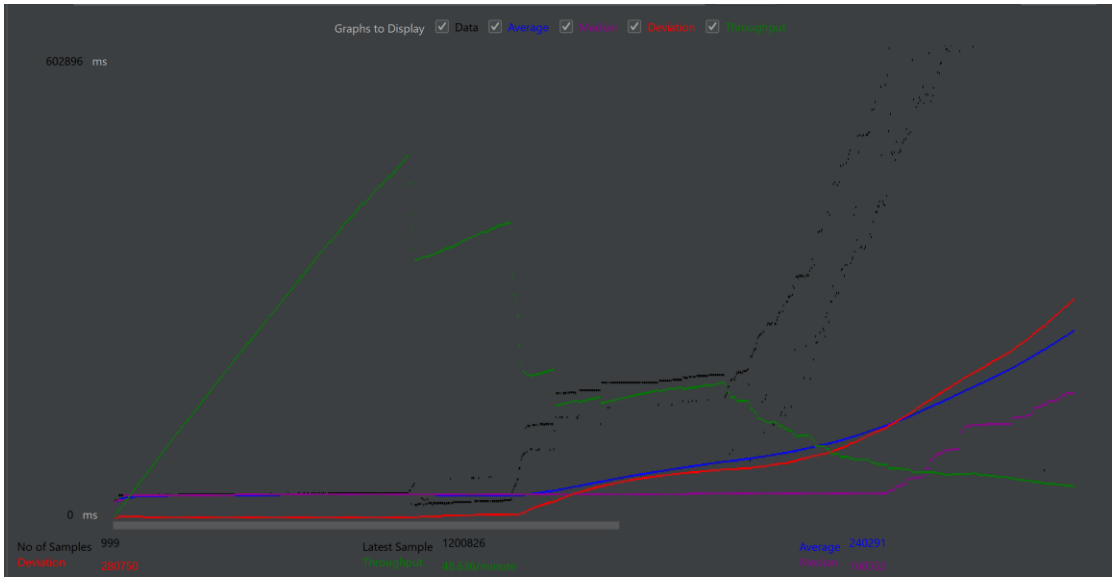
Take the update event operation as an example, in this operation if the unit of work is properly designed, the system can manage updates from different users on different

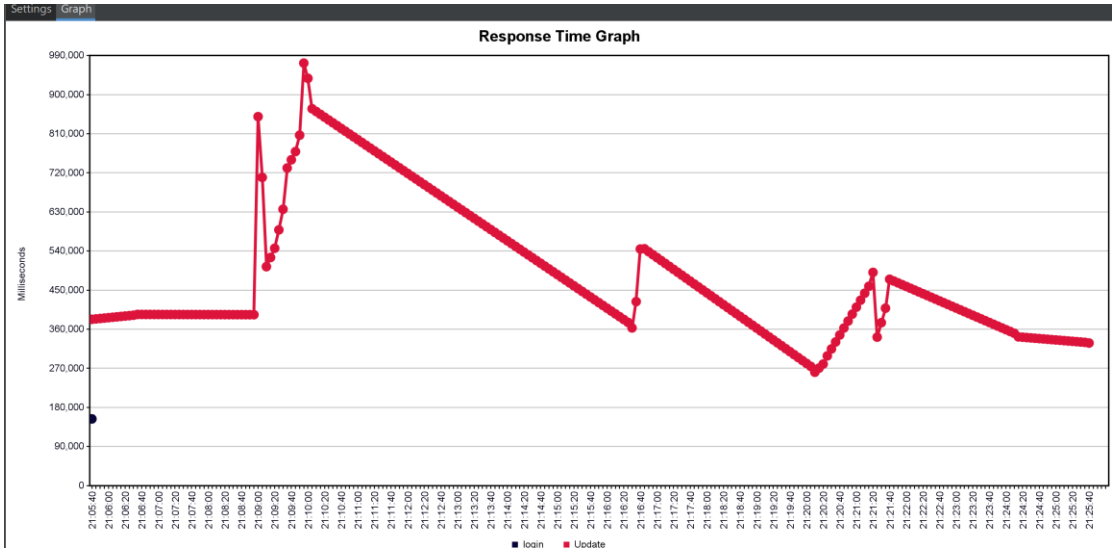
events through a unit of work. The workload should be multiple users trying to modify events and the event should be added to a list and wait until the system runs a commit operation to push all the updates to the database so that the interaction between database and system can be minimized to improve system performance. However, in our system even though the unit of work is introduced, the system still commits each time the update method is used so there is no difference between using the unit of work and using a separate update method as, for now, they all call the database each time an event needs to be updated.

Images shown below are the performance of the update event.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/...	Sent KB/sec
login	500	83962	32122	176976	181833	183855	21542	1015825	0.00%	29.5/min	1.28	0.31
Update	500	398353	319945	852823	990062	1118289	18019	1200826	0.00%	23.6/min	0.11	0.19
TOTAL	1000	241157	160552	699786	887518	1087280	18019	1200826	0.00%	46.4/min	1.12	0.43





The graphs shown here result from 1000 threads trying to do the update event operation. Note that the concurrency-ensuring mechanism and the redirecting to the view event page after the update mechanism are removed while running the threads to test the separate update event operation's performance. In this thread test, two EventPlanners are used to update two events, this means there are a user updating different events and different users updating the same event. As this report mainly focuses on performance, the concurrency insurance mechanism can lead to a large amount of rollback which affects the performance analysis, this mechanism is removed. The unit of work is only added to the update, add and delete event while the view event page uses the read event method which also may affect the performance analysis of the unit of work so the redirection is also removed.

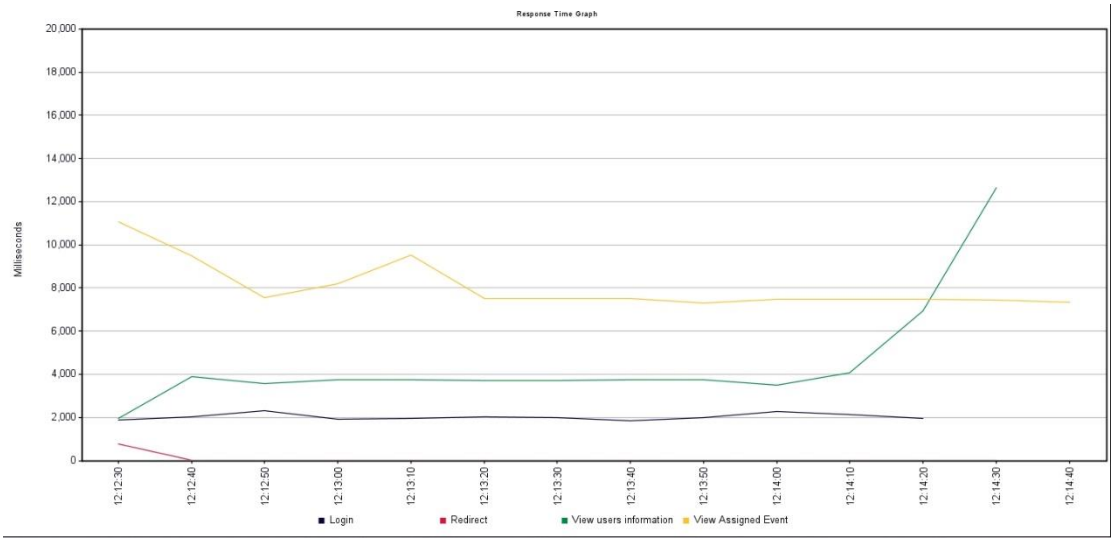
As we can see, the throughput is initially high when the system is still running login operations but when the system starts to run the update event operation, the throughput drops dramatically. With more and more threads working on the update event operation, the throughput keeps getting down and gets to near zero. The update event operation's response time is almost 5 times the login operation's runtime. The data received amount of the login operation's runtime is nearly 10 times greater than that of the update event operation. For the response time graph, the login response time is stable and much shorter than the update event operation. For the first on third of update event operations, the response time is stable while it goes to a peak, almost twice the stable time when more and more threads keep coming in. However, after the peak, the response time gradually returns to near the stable response time and then fluctuates around that data.

In conclusion, as the update operation frequently calls the method to update the database, it consumes many system resources and leads to poor system performance. If we can use an identity map in this business transaction and implement the unit of work properly, the event object would be put into the identity map and then all update operations can interact with the existing object in the system before they try to call the database. The update method would register dirty with the existing event object and then after proper other operations, all the operations within this period can be committed to the database. In this case, the system transforms multiple frequent database modifications into a large amount of consistent database operations which will save resources and reduce response time since the connection to the database needs time and resources.

Lazy Load

Prior to the execution of the performance test, fundamental parameters and configurations were defined to tailor the test according to our specific requirements. The focus of our test is on the performance of the "events assigned" functionality, particularly when accessed by an administrator account, which is used Lazy Load design pattern. The number of threads was set to one, aligning with the fact that there

is only one administrator account in our system. The loop count was configured to simulate the repeated access of the "events assigned" page by the administrator. Setting a specific loop count to 10 allows the simulation of sustained user activity, providing a more comprehensive assessment of the system's performance under continuous load.



From the provided graph illustrating the response times of various requests, a focused analysis on the "View Assigned Event" (depicted by the green line) which implements lazy loading reveals several insights into its performance implications:

The response times for the "View Assigned Event" appear relatively consistent, devoid of abrupt spikes or drops. This suggests a level of stability in handling requests throughout the test, attributed to the lazy loading strategy.

Compared to operations like "Login" and "Redirect," the "View Assigned Event" manifests a higher response time. This could be reflective of the more extensive data loading and processing inherent in this operation.

Lazy loading aims to enhance performance by deferring the initialization of objects until they are needed. The graph hints at the efficacy of lazy loading in maintaining a semblance of stability in the response times, even with increased iterations. Because the amount of data loaded by the "View Assigned Event" page is not large, lazy loading may not bring significant performance improvements. But with future system expansion, Lazy Load will show better performance advantages.

DTO and Remote Façade

In our existing system, the DTO and Remote Façade design patterns have not been adopted. DTO, Data Transfer Object, is an object used to encapsulate multiple data items so that they can be transmitted as a unit through a single network call. In the

Music Event system, DTO could be utilized to retrieve and transmit all relevant information of a concert event in one go, such as the artists, venue, time, ticket prices, and remaining ticket quantities.

Remote Façade is a design pattern that provides a unified interface to access a set of interfaces of remote objects, thus simplifying the client code. Correspondingly, we could offer a remote Façade for functionalities like ticket purchasing, refunding, and viewing concert details. This way, the client can operate through a simplified interface without having to interact directly with multiple remote objects.

Implementing the DTO and Remote Façade design patterns in our Music Event system could enhance its performance including quicker response times and higher throughput by optimizing network traffic, improving data transfer efficiency, and reducing the number of network requests.

Design principles

Pessimistic offline lock

In our system, pessimistic offline lock is used to resolve concurrency issues regarding Event Planners, such as adding/updating events at the same time.

The LockManagerEx class is used to manage the locks. It contains two maps, lockMap and orderLockMap, which map lockable resources to their respective owners. When creating or updating an event, the system attempts to acquire a lock on the venue entity by calling the acquireLock method of LockManagerEx. This method checks if the lockable resource (venue) is already locked by another transaction. If it is not locked, the method grants the lock to the current transaction and stores the lockable resource and its owner in the lockMap. If the resource is already locked, a RuntimeException is thrown, indicating that the lock could not be acquired. After the transaction is completed the lock is then released.

The lock does ensure the consistency of correctness, however, in terms of performance, the lock suffers from low liveness. Firstly, the lock has a relatively large granularity, as it locks out the venueId when a user is accessing event information, so even if another event planner tried to create or update an event to a valid and non-clashing time slot to the same venue, the planner will still be locked out. To further improve the performance, Instead of locking at the venue level, the system may lock at a finer granularity such as individual events or specific time slots in a venue. This can reduce contention and allow more concurrency. Another performance issue is that the lock acquisition methods do not have a timeout mechanism, which means that a thread can potentially be blocked indefinitely while waiting for a lock. To improve, a timeout for lock acquisition should be implemented. If a thread cannot acquire a lock

within a specified time, it should release any acquired locks and retry or abort the operation. Finally, the lock mechanism suffers from no Fairness in Lock Acquisition. The lock acquisition methods do not guarantee fairness, meaning that there is no guarantee on the order in which waiting threads will acquire the lock. This could lead to thread starvation, where a thread is indefinitely blocked because other threads keep acquiring the lock. To improve, the system can maintain a first-come-first-serve queue of threads that are waiting to acquire the lock.Once a thread acquires the lock and completes its critical section, it removes itself from the front of the queue, allowing the next thread in line to acquire the lock.

Optimistic Lock

Besides the pessimistic lock that was mentioned earlier, optimistic lock was also implemented in the system to handle concurrency. In our code, the version number of the event is checked before updating, If the version number has changed since the event was last read, the update is aborted, and an OptimisticLockException is thrown. This prevents concurrent updates from conflicting and ensures that changes made by different threads do not overwrite each other. The OptimisticLockException is then caught and handled by rolling back the transaction and returning an error message to the user.

Performance wise, the optimistic lock works well because the contention for the same event is relatively low (compared to customer purchasing tickets) and it avoids the overhead of acquiring and releasing locks for every operation. However, if the system was to be scaled up, its performance can degrade in scenarios with high contention, as conflicts will lead to frequent rollbacks and retries, thus increasing the response time. One way to improve the performance is to implement a retry mechanism in case of an OptimisticLockException. Instead of immediately returning an error, the system could automatically retry the operation a certain number of times before returning an error. This could increase the chances of the update succeeding without requiring user intervention. If the contention for modifying events is potentially high in the future, using a pessimistic lock as mentioned above might be better as it acquires a lock on the data before updating it and holds the lock until the update is complete, preventing other threads from accessing the data in the meantime. This can reduce the likelihood of conflicts and rollbacks but comes at the cost of increased lock contention and potential bottlenecks.

Bell’s principle

Bell's principle emphasizes the importance of simplicity in system design.It is believed that simple designs often exhibit better performance than more complex ones, and features or complexity should be avoided. This aligns with our design concept, as our system were developed with these features in mind:

Minimalism in Design

The system only includes the necessary components to meet the requirements. For example, the Unit of Work pattern is employed to manage transactions in a streamlined way, without adding unnecessary complexity. This focused design ensures that the system does what it needs to do without any superfluous features.

Efficiency in Implementation

By utilizing mappers, the system can efficiently transform data between different formats or objects, reducing the need for redundant code and minimizing the potential for bugs. This efficiency in implementation directly results from the simple and clear design of the system.

Pipelining

In our music event system, we did not use pipelining as it is unsuitable for our system. The pipelining principle is useful when the requests are not dependent on each other and can be parallel. It is most effective when tasks can be broken down into independent stages that can be processed concurrently. However, in our event booking system, transactions are highly dependent on each other. For example, the event planner can not process the delete event operation when the client is booking the ticket for that event. There is a sequential dependency on the tasks. Another example is that the client can not buy the ticket before he selects the event he tries to book. These operations need to happen in sequential order, the next transaction has to wait until the current transaction successfully proceeds. This is contradicted by the pipelining principle. If we apply pipelining in our system, pipelining could increase the throughput and efficiency of those tasks that are independent and involves large data streaming, such as deleting ticket and displaying event. However, it could add unnecessary complexity to the system as most of its operations depend on each other, increasing the difficulty of debugging. It could also damage the data consistency in the system, pipelining introduces latency between stages which potentially leads to scenarios in which the client books a ticket that does not exist which is not ideal.

Caching

We add caching in the system which stores the copies of frequently accessed data in the cache such as available tickets for each event. This can reduce the number of requests to the server which can be beneficial during high volume request periods such as when a popular event happened and sales tickets for that event. It increases the system's efficiency and reduces the chance of breaking the server.