



Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)



DEPARTMENT OF INFORMATION TECHNOLOGY

NAME: RISHIKESH SHARMA

BATCH: I1-2

COURSE CODE: DJ19ITC802

DATE: 03/04/2024

COURSE NAME: Design Patterns Laboratory

CLASS: BE - IT

EXPERIMENT NO. 9

CO/LO: Identify and apply the most suitable design pattern to address a given application design problem.

AIM: Case study on Enterprise Integration Patterns

DESCRIPTION: In the world of finance, particularly in the realm of securities trading, efficiency, accuracy, and speed are paramount. Bond trading, in particular, involves intricate processes that demand seamless communication and integration across various systems and entities. This case study delves into the implementation of Enterprise Integration Patterns (EIP) within a Bond Trading System, illustrating how these patterns facilitate the smooth flow of information and transactions within the system.

Introduction to the Bond Trading System: The Bond Trading System is a complex platform used by financial institutions, brokers, and traders to buy and sell bonds in the financial markets. Bonds, being debt securities, involve numerous parties and steps in their trading lifecycle, from issuance to maturity. The Bond Trading System streamlines these processes, ensuring efficient trade execution, settlement, and management.

QUESTIONS & ANSWERS:

- 1. Describe the architecture and components of the bond trading system presented in the case study.**

With this overview of the workflow of the system, we can approach some of the architectural problems we encounter during the design process. Let's take a look at what we know to date. Traders need a very responsive application on both Windows NT and Solaris workstations. Therefore, we decided to implement the client application as a Java thick client because of its



Shri Vile Parle Kelavani Mandal's

DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING

(Autonomous College Affiliated to the University of Mumbai)

NAAC Accredited with "A" Grade (CGPA : 3.18)



platform independence and its ability to quickly respond to user input and market data. On the server side, we are inheriting legacy C++ components that our system will utilize. The market data components communicate with the TIBCO Information Bus (TIB) messaging infrastructure.

We are inheriting the following components:

- Market Data Price Feed Server: Publishes incoming market data to the TIB.
- Analytics Engine: Performs analytics on incoming market data and broadcasts the modified market data to the TIB.
- Contribution Server: Performs all communication with trading venues. The trading venues are third party components not controlled by the bank.

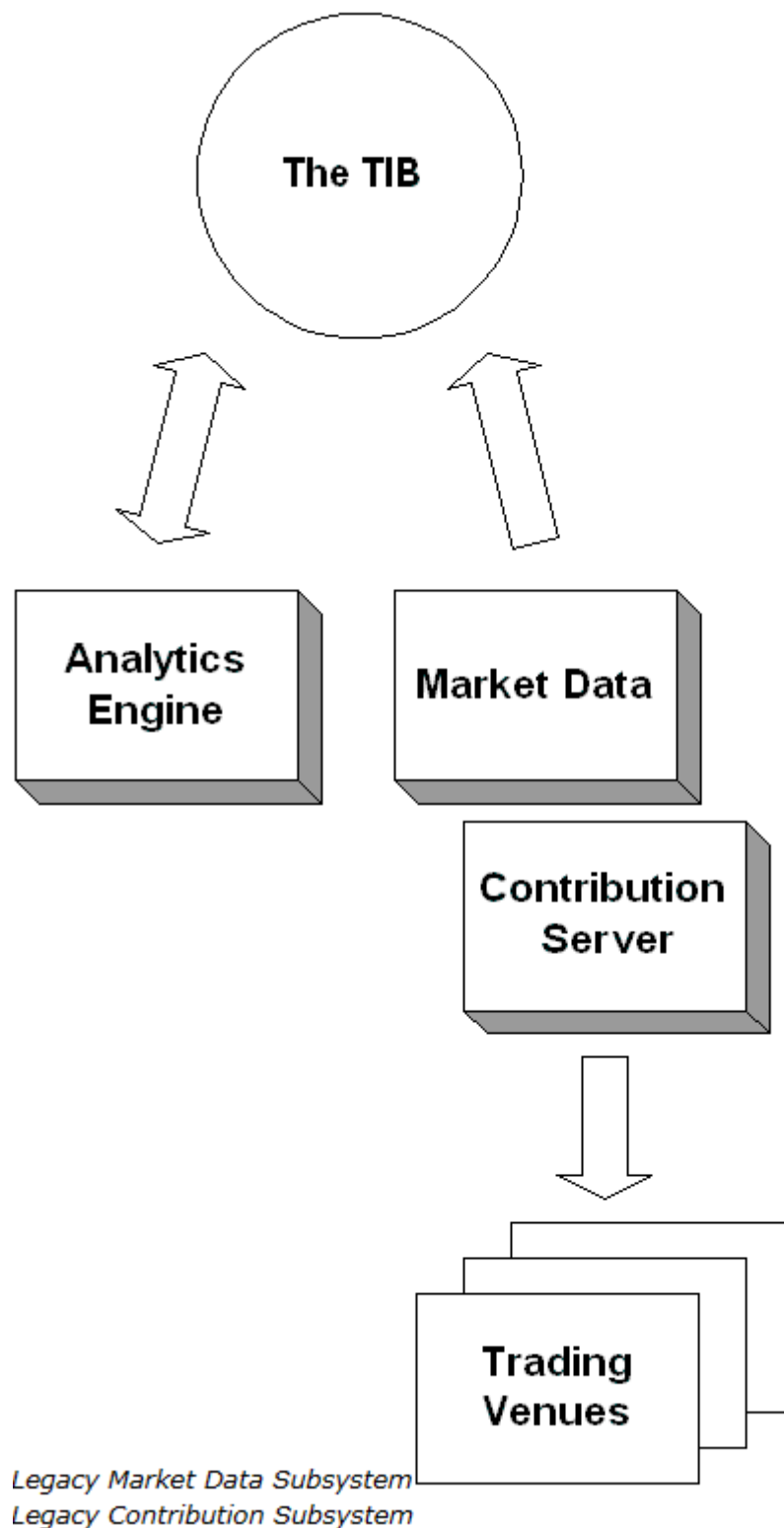


Shri Vile Parle Kelavani Mandal's

DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING

(Autonomous College Affiliated to the University of Mumbai)

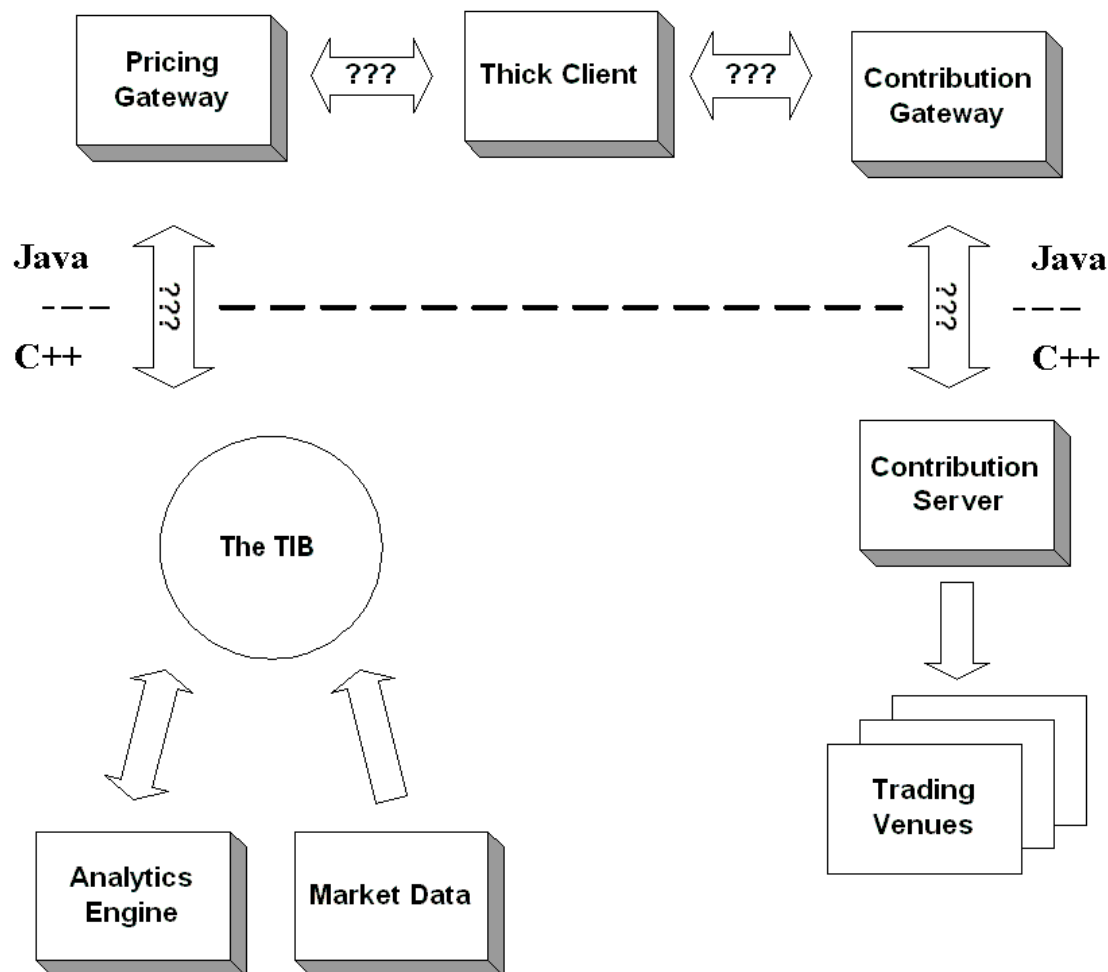
NAAC Accredited with "A" Grade (CGPA : 3.18)



We need to decide how the separate subsystems (Java thick client, market data, and contribution) are going to communicate. We could have the thick client communicate directly with the legacy servers, but that would require too much business logic on the client. Instead,



we'll build a pair of Java gateways to communicate with the legacy servers—The Pricing Gateway for market data a Contribution Gateway for sending prices to trading venues. This will achieve nice encapsulation of the business logic related to these areas. The current components in the system are shown below. The connections marked as “???” indicate that we are still unsure how some of the components will communicate.



The first communication question is how to integrate the Java thick client and the two Java server components in order to exchange data. Let's look at the four integration styles suggested in this book: File Transfer, Shared Database, Remote Procedure Invocation, and Messaging. We can rule out Shared Database immediately because we wanted to create a layer of abstraction between the client and the database and don't want to have database access code in the client. File Transfer can similarly be ruled out since minimal latency is required to ensure



current prices are sent out to the trading venues. This leaves us with a choice between Remote Procedure Invocation or Messaging.

The Java platform provides built-in support for both Remote Procedure Invocation and Messaging. RPC-style integration can be achieved using Remote Method Invocation (RMI), CORBA, or Enterprise Java Beans (EJB). The Java Messaging Service (JMS) is the common API for messaging-style integration. So both integration styles are easy to implement in Java.

So which will work better for this project, Remote Procedure Invocation or Messaging? There's only one instance of the Pricing Gateway and one instance of the Contribution Gateway in the system, but usually many Thick Clients simultaneously connect to these services (one for each bond trader that happens to be logged in at a particular time). Furthermore, the bank would like this to be a generic pricing system that can be utilized in other applications. So besides an unknown number of Think Clients, there may be an unknown number of other applications using the pricing data coming out of the Gateways.

A Thick Client (or other application using the pricing data) can fairly easily use RPC to make calls to the Gateways to get pricing data and invoke processing. However, pricing data will constantly be published, and certain clients are only interested in certain data, so getting the relevant data to the proper clients in a timely manner could be difficult. The clients could poll the Gateways, but that will create a lot of overhead. It would be better for the Gateways to make the data available to the clients as soon as it is available. This, however, will require each Gateway to keep track of which clients are currently active, and which want what particular data; then, when a new piece of data becomes available (which will happen numerous times per second), the Gateway will have to make an RPC to each interested client to pass the data to the client. Ideally, all clients should be notified simultaneously, so each RPC needs to be made in its own concurrent thread. This can work, but is getting very complicated very fast.

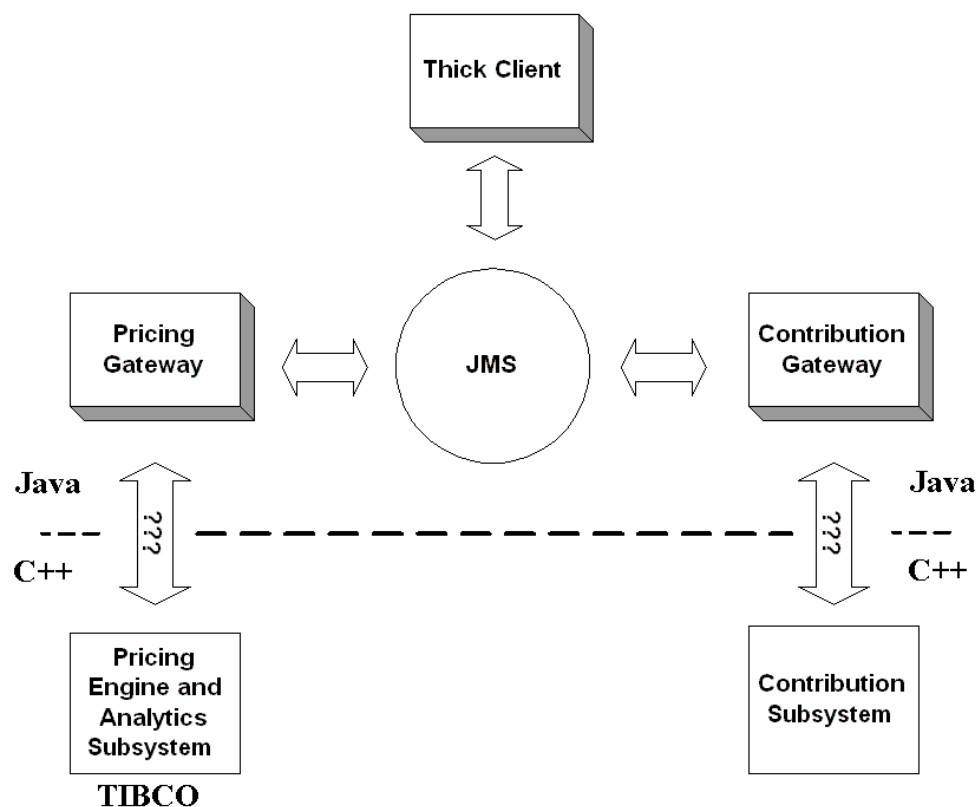
Messaging greatly simplifies this problem. With Messaging, we can define separate channels for the different types of pricing data. Then, when a Gateway gets a new piece of data, it will add a message containing that data to the Publish-Subscribe Channel for that data type. Meanwhile, all clients interested in a certain type of data will listen on the channel for that type.



In this way, the Gateways can easily send out new data to whomever is interested, without needing to know how many listener applications there are or what they are.

The clients still need to be able to invoke behaviour in the Gateways as well. Since there are ever only two Gateways, and the client can probably block while the method is invoked synchronously, these client-to-Gateway invocations can fairly easily be implemented using RPC. However, since we are already using messaging for Gateway-to-client communication, messages are probably just as good a way to implement client-to-Gateway communication as well.

Therefore, all communication between the Gateways and the clients will be accomplished through messaging. Because all of the components are written in Java, JMS presents an easy choice for as the messaging system. This is effectively creating a Message Bus or an architecture that will make it possible for future systems to integrate with the current system with little or no changes to the messaging infrastructure. This way, the business functionality of the application can be easily used by other application the bank develops.





JMS is simply a specification and we need to decide on a JMS-compliant messaging system. We decided to use IBM MQSeries JMS because the bank is an “IBM shop,” using WebSphere application servers and many other IBM products. As a result, we will use MQSeries since we already have a support infrastructure in place and a site license of the product.

The next question is how to connect the MQSeries messaging system with the standalone C++ Contribution server and the TIBCO based Market Data and Analytics Engine servers. We need a way for the MQSeries consumers to have access to the TIB messages. But how? Perhaps we could use the Message Translator pattern to translate TIB messages into MQSeries messages. Although the C++ client for MQSeries serves as a Message Translator, using it would sacrifice JMS server independence. And although TIBCO does have a Java API, the customer architect and manager have rejected it. As a result, the Message Translator approach has to be abandoned.

The bridge from the TIB server to the MQSeries server requires communication between C++ and Java. We could use CORBA, but then what about the messaging? A closer look at the Message Translator pattern shows it is related to the Channel Adapter in its use of communication protocols. The heart of a Channel Adapter is to connect non-messaging systems to messaging systems. A pair of channel adapters that connects two messaging systems is a Messaging Bridge.

The purpose of a Messaging Bridge is to transfer messages from one messaging system to another. This is exactly what we are doing with the added complexity of the intra-language Java to C++ communication. We can implement the cross language Messaging Bridge using a combination of Channel Adapters and CORBA. We will build two lightweight Channel Adapter servers, one in C++ managing communication with the TIB, and one in Java managing communication with JMS. These two Channel Adapter, which are Message Endpoints themselves, will communicate with each other via CORBA. Like our choice for MQSeries, we will use CORBA rather than JNI since it is a company standard. The messaging bridge implements the effectively simulated message translation between seemingly incompatible messaging systems and different languages.

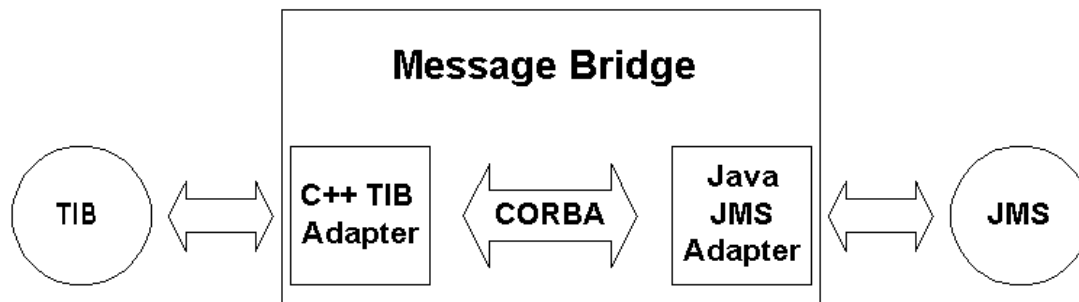


Shri Vile Parle Kelavani Mandal's

DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING

(Autonomous College Affiliated to the University of Mumbai)

NAAC Accredited with "A" Grade (CGPA : 3.18)



The next diagram shows the current system design including the Gateways and other components. This is a good example of pattern application. We combined two Channel Adapters with a non-messaging protocol to implement the Message Translator pattern, effectively using one pattern to implement another pattern. Additionally, we changed the Channel Adapters' context to link two messaging systems with a non-messaging cross language translation protocol rather than connecting a messaging system to a non-messaging system.

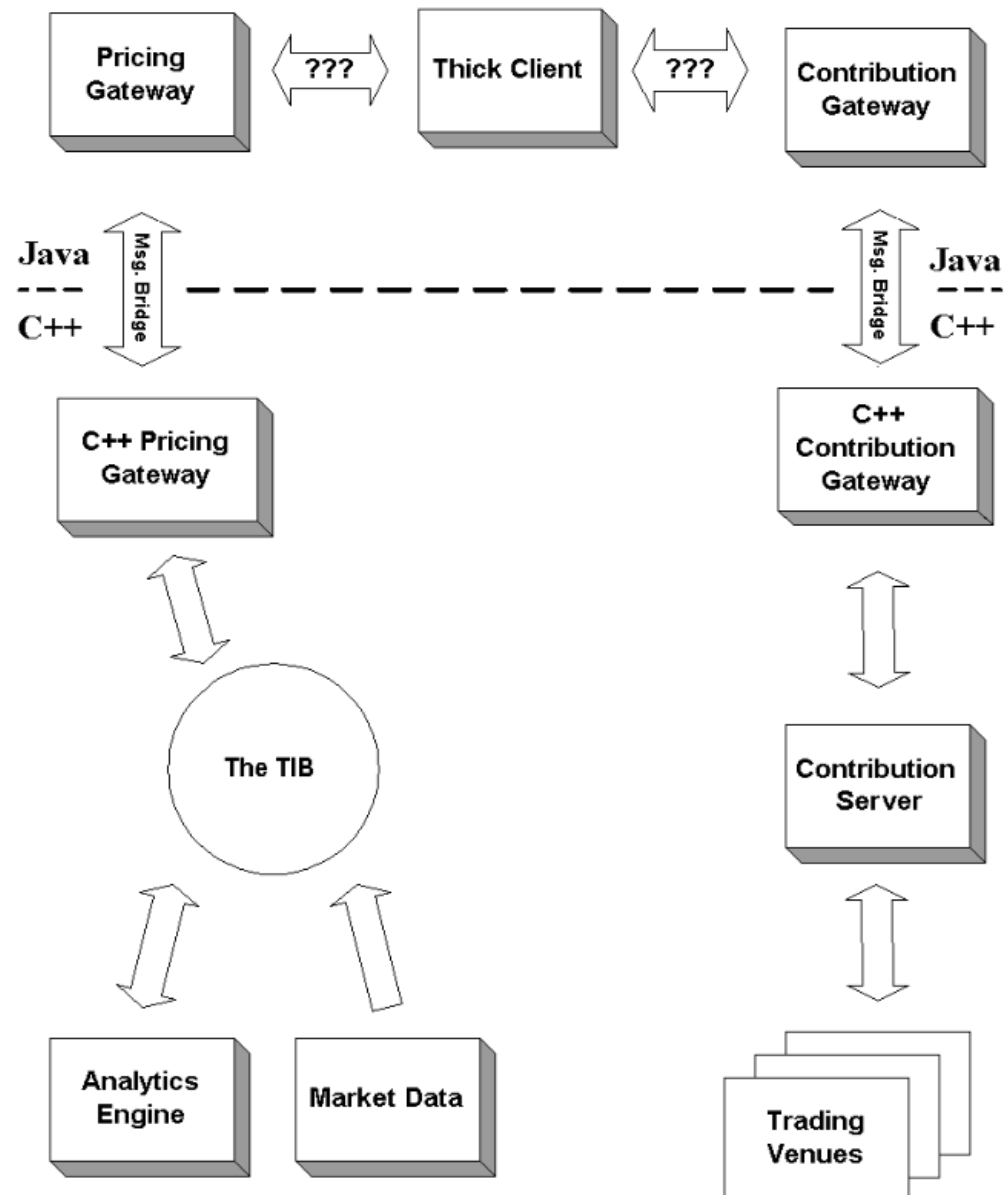


Shri Vile Parle Kelavani Mandal's

DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING

(Autonomous College Affiliated to the University of Mumbai)

NAAC Accredited with "A" Grade (CGPA : 3.18)

*The current system with the Channel Adapters*



2. Which messaging patterns are used in the bond trading system?

According to above data,

Which messaging patterns are used in the bond trading system?

ChatGPT

Based on the provided information, the Bond Trading System employs several messaging patterns as part of its Enterprise Integration Patterns (EIP) implementation. These messaging patterns include:

Message Channels: Used for asynchronous communication to decouple producers and consumers of messages, transmitting trade orders, market data, and settlement instructions.

Message Router: Routes messages to appropriate destinations based on predefined criteria, such as routing trade orders to different trading venues based on instrument type, price, and volume.

Message Translator: Converts messages from one format to another, such as translating trade messages from FIX protocol to proprietary formats used internally.

Content-Based Router: Routes messages based on their content, such as routing trade orders to specific markets or counterparties based on predefined criteria.

Message Filter: Selectively processes messages based on predefined criteria, forwarding only trade orders meeting certain criteria for execution.

Aggregator: Combines multiple messages into a single message, simplifying downstream processing, such as aggregating trade confirmations and settlement instructions.

These messaging patterns collectively facilitate efficient communication, data transformation, and routing within the Bond Trading System, contributing to its seamless operation and integration with various internal and external systems.



3. What strategies are employed to handle peak loads and ensure system responsiveness?

To handle peak loads and ensure system responsiveness in the Bond Trading System, several strategies are employed:

Scalability: The system architecture is designed to scale horizontally and vertically. Horizontal scalability involves adding more machines to distribute the load, while vertical scalability involves adding more resources (such as CPU, memory, or storage) to existing machines. This allows the system to handle increased transaction volumes during peak loads without sacrificing performance.

Load Balancing: Load balancing distributes incoming network traffic across multiple servers to ensure no single server is overwhelmed. By evenly distributing the workload, load balancers help maintain system responsiveness during peak loads and prevent any individual component from becoming a bottleneck.

Caching: Caching frequently accessed data or computations helps reduce the load on backend systems and improves response times. By storing data closer to the users or applications that need it, caching reduces latency and enhances system responsiveness, especially during peak loads.

Optimized Data Processing: Optimizing data processing algorithms and workflows can improve system performance and reduce processing times. This includes optimizing database queries, streamlining data transformation processes, and minimizing unnecessary computations to ensure efficient handling of peak loads.

Asynchronous Processing: Leveraging asynchronous processing techniques, such as asynchronous messaging and event-driven architecture, allows the system to handle bursts of activity more effectively. By decoupling components and processing tasks asynchronously, the system can continue to operate smoothly even during periods of high demand.

Resource Provisioning: Dynamic resource provisioning and auto-scaling mechanisms automatically allocate resources based on demand. By dynamically adjusting resource allocation in response to changes in workload, these mechanisms ensure that the system has sufficient capacity to handle peak loads while minimizing resource waste during periods of lower demand.



Shri Vile Parle Kelavani Mandal's

DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING

(Autonomous College Affiliated to the University of Mumbai)

NAAC Accredited with "A" Grade (CGPA : 3.18)



Performance Monitoring and Optimization: Continuous performance monitoring and optimization help identify performance bottlenecks and inefficiencies in the system. By monitoring key metrics such as response times, throughput, and resource utilization, system administrators can proactively identify areas for improvement and optimize system performance to ensure responsiveness during peak loads.

By employing these strategies in combination, the Bond Trading System can effectively handle peak loads while maintaining system responsiveness and ensuring smooth operation even under high demand.



Shri Vile Parle Kelavani Mandal's

DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING

(Autonomous College Affiliated to the University of Mumbai)

NAAC Accredited with "A" Grade (CGPA : 3.18)



4. Explain how performance metrics are collected and analysed to optimize system performance.

In the Bond Trading System, performance metrics are collected and analyzed to optimize system performance through a systematic approach involving the following steps:

Identifying Key Performance Indicators (KPIs): The first step is to identify the key performance indicators (KPIs) that are critical for measuring the performance of the system. These KPIs may include metrics such as response times, throughput, error rates, resource utilization, latency, and system availability.

Instrumentation and Monitoring: The system is instrumented to collect data on the identified KPIs continuously. This involves integrating monitoring tools and agents into the system architecture to capture performance metrics in real-time. Various monitoring techniques such as application performance monitoring (APM), network monitoring, and infrastructure monitoring may be employed to collect relevant data.

Data Collection and Aggregation: Collected performance data is aggregated and stored in a centralized repository or database for analysis. This repository serves as a single source of truth for performance metrics across the entire system.

Data Analysis: Performance data is analyzed using various techniques such as statistical analysis, trend analysis, and correlation analysis. Data visualization tools may be used to create dashboards and reports that provide insights into system performance over time. Performance anomalies, trends, and patterns are identified and analyzed to understand the underlying causes of performance issues and areas for optimization.

Root Cause Analysis: In cases where performance issues are identified, root cause analysis (RCA) techniques are employed to determine the underlying factors contributing to the problem. This may involve tracing the performance bottleneck to specific components, subsystems, or architectural design decisions.

Optimization Strategies: Based on the insights gained from performance analysis and root cause analysis, optimization strategies are devised to address identified performance issues. These strategies may include code optimization, database tuning, resource allocation adjustments, architectural redesigns, or infrastructure upgrades.

Performance Testing: After implementing optimization strategies, performance testing is conducted to validate the effectiveness of the changes and ensure that performance



Shri Vile Parle Kelavani Mandal's

DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING

(Autonomous College Affiliated to the University of Mumbai)

NAAC Accredited with "A" Grade (CGPA : 3.18)



objectives are met. This may involve load testing, stress testing, and scalability testing to simulate various scenarios and assess system performance under different conditions.

Continuous Improvement: Performance optimization is an ongoing process, and continuous monitoring and analysis are essential to ensure that the system remains optimized over time. Performance metrics are continually monitored, and optimization efforts are iteratively refined based on evolving system requirements and performance goals.

By following this approach, the Bond Trading System can effectively collect, analyze, and leverage performance metrics to optimize system performance, enhance user experience, and meet business objectives.

CONCLUSION: Hence, we have successfully studied the Case study on Enterprise Integration Patterns.

REFERENCES:

- [1] <https://www.enterpriseintegrationpatterns.com/patterns/messaging/BondTradingCaseStudy.html>