Enhancing WebSphere Application using MQSeries 1.0

Authors: Sudhakar; Sandeep Goyal

Intended Audience: This paper is primarily intended for users of WebSphere and MQSeries

Further Inquiries and Feedback: info@thbs.com

Notices:

Torry Harris Business Solutions
1090 King Georges Post Rd
Suite 103
Edison NJ 08837
USA

CONTENTS

FIGURES

WebSphere Application Server

1.1. An overview

The marketplace of the World Wide Web continues to grow rapidly. Increasingly, Web sites with dynamic HTML pages gain the competitive edge by offering interactivity and self-serve transactions. For this interactivity, the business logic applications work behind the scenes to provide immediate access to data in response to user requests.

WebSphere Application Server Advanced Edition enables web transactions and interactions with a robust deployment environment for e-business applications. It provides a portable, Java-based web application deployment platform focused on supporting and executing servlets, Java Beans, Java Server Pages (JSP) files, and enterprise beans. It builds on the Standard Edition to provide portability and control of server-side business applications along with the performance and manageability of enterprise beans to offer a comprehensive Java-based Web application platform. It extends the value and versatility of this platform with:

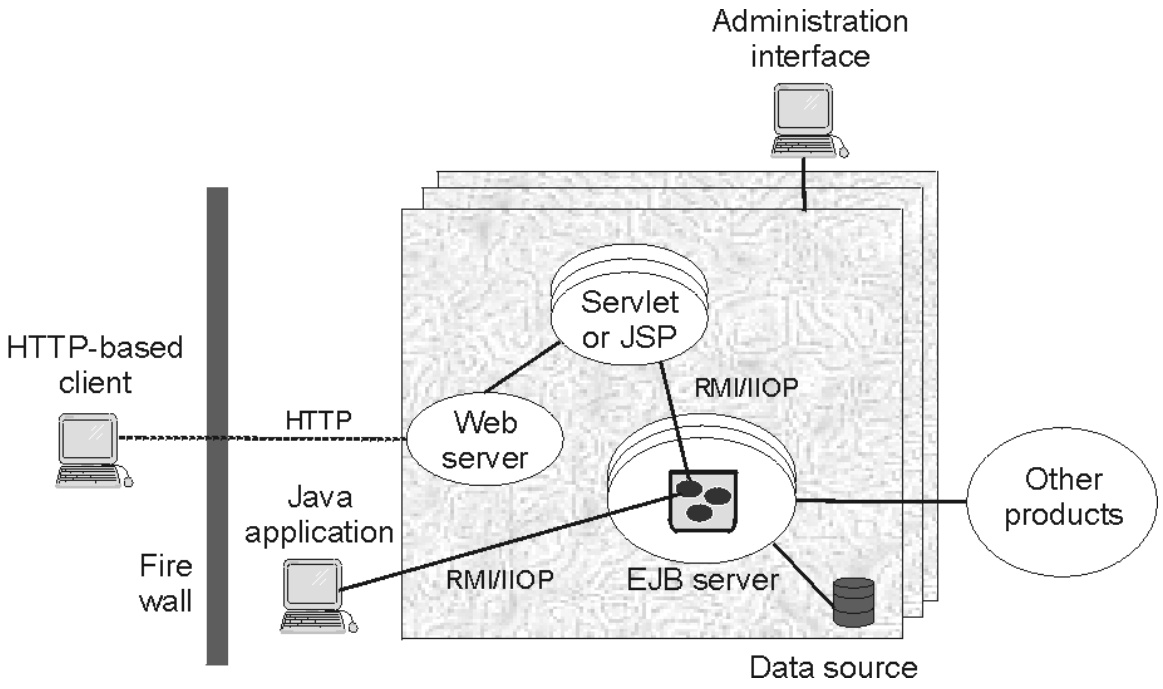
- Complete Java and Enterprise Java support including a server for applications built to the enterprise beans specification. The focus is on medium to high-level transactional environments used with dynamic Web content generation and Web-initiated transactions.
- Inclusion of an Apache-based HTTP server with enhancements for security and control, as well as support for other major Web servers.
- Performance and scaling attributes with support for bean-managed and container-managed persistence, for entity beans and session beans, with transaction management and monitoring. Container management and persistent storage helps provide a high-performance transactional environment using servlets and enterprise beans.
- Support for Intel and Unix-based platforms including Windows NT, Sun Solaris, and AIX on HTTP servers from IBM, Apache, Microsoft, and Netscape.

WebSphere Application Server is integral in managing and integrating enterprise-wide applications while leveraging open Java-based technologies and APIs. It enables powerful interactions with relational databases, transaction processing systems, and other applications. WebSphere application server provides deployment and management of Java, CORBA, and enterprise beans applications.

1.2. The Advanced Application Server environment

The Advanced Application Server contains the following components, which can be combined to create a powerful Java-centered three-tiered system that puts heavy emphasis on a customer's Web site. These components are illustrated in the following figure.

Figure 1 ***Components of the Advanced Application Server environment***



Browser-based applications
Allow users to send and receive information from Web sites by using the Hypertext Transfer Protocol (HTTP). There are three general types of browser-based applications: Java applets, Java servlets, and Java Server Pages(TM) (JSP).

Web servers
Except for stand-alone Java applets, which are restricted by built-in Java security, browser-based applications require that a Web server be installed on at least one machine in your Advanced Application Server environment.

EJB servers and enterprise beans
The WebSphere EJB server contains one or more enterprise beans, which encapsulate the business logic and data used and shared by EJB applications. The enterprise beans installed in an EJB server do not communicate directly with the server; instead, an EJB container provides an interface between the enterprise beans and the EJB server, providing many low-level services such as threading, support for transactions, and management of data storage and retrieval.

Java applications
Java applications can interact directly with an EJB server by using Java remote method invocation over the Internet Inter-ORB Protocol (RMI/IIOP).

Data sources
The EJB server stores and retrieves this persistent data in a database.

Administration server and the administrative interface
The administration server manages servlets, JSP files, enterprise beans, and EJB servers. The WebSphere Application Server administrator who uses the WebSphere Administrative Console, who is the administrative interface to the administration server, directs this management.

1.3. Java standards
The Web server provides the communications link between browser-based applications and the other components of Advanced Application Server.

The Advanced Application Server contains a Java-based servlet engine that is independent of both your Web server and its underlying operating system. It supports many of the most widely used Web servers. The IBM HTTP Server, which is a modified version of the Apache server, comes with the Advanced Application Server.

1.3.1. Java applets and servlets
Java applets are Java applications that run on a browser and extend the browser's capabilities.

Java servlets run on a Java-enabled Web server and extend the server's capabilities. Servlets are Java programs that use the Java Servlet API and the associated classes and methods.

Servlets extend Web server capabilities by creating a framework for providing request and response services over the Web. When a client sends a request to the server, the server can send the request information to a servlet and have the servlet construct the response that the server sends back to the client. Servlets can handle user requests by using threads.

1.3.2. Java Server Pages
JSP is a powerful, new approach to dynamic Web page content. JSP embed servlet functionality into the Web page. In JSP, Java servlet code (or other Java code) is embedded directly into the HTML page. One of the many advantages of JSP is that it enables you to effectively separate the HTML coding from the business logic in your Web pages. You can use JSP to access reusable components, such as servlets, Java beans, enterprise beans, and Java-based Web applications.

1.3.3. EJB servers and enterprise beans
An Enterprise bean is a Java component that can be combined with other enterprise beans and other Java components to create a distributed, three-tiered application.

An EJB server provides the run-time environment for enterprise beans, handling low-level programming tasks like transaction management, naming, and security.

There are two types of enterprise beans:
- An entity bean encapsulates permanent data, which is stored in a data source like a database or a file system, and associated methods to manipulate that data. In most cases, an entity bean must be accessed in some transactional manner. Instances of an entity bean are unique, and multiple users can access them. For example, the information about a bank account can be encapsulated in an entity bean instance. An account enterprise bean might contain an

account ID, an account type (checking or savings), and a balance.

- A session bean encapsulates one or more business tasks and nonpermanent data associated with a particular client. Unlike the data in an entity bean, the data in a session bean is not stored in a permanent data source and no harm is caused if this data is lost. Nevertheless, a session bean can update data in an underlying database, usually by accessing an entity bean. For this reason, a session bean can be transaction aware. When created, instances of a session bean are identical, though some session beans can store semi-permanent data that makes them unique at certain points in their life cycle. A stateful session bean is always associated with a single client, whereas a stateless session bean is associated with a client only during a method call. For example, the task associated with transferring funds between two bank accounts can be encapsulated in a session bean. Such a transfer enterprise bean might find two instances of an account enterprise bean (by using the account IDs), and then subtract a specified amount from one account and add the same amount to the other account.

Before an enterprise bean can be installed in an EJB server, the enterprise beans must be deployed. During deployment, several EJB server-specific classes are generated. The deployment descriptor contains attribute and environment settings that define how the EJB server invokes enterprise bean functionality. Every enterprise bean (both session and entity) must have a deployment descriptor that contains settings used by the EJB server; these attributes can often be set for the entire enterprise bean or for the individual methods in the bean. The Advanced Application Server provides tools for creating deployment descriptors and deploying enterprise beans.

MQSeries – an Overview

MQSeries products enable applications to use message queuing to participate in message-driven processing. With message-driven processing, applications can communicate with each other across a network of unlike components, such as processors, subsystems, operating systems and communication protocols. MQSeries programs use a consistent application program interface (API) across all platforms.

1.4. Features of MQSeries

MQI - a common application-programming interface

MQSeries products implement MQI, which is used on almost any platform the applications run on. The calls made by the applications and the messages they exchange are common. This makes it much easier to write and maintain applications than using traditional methods. It also facilitates the migration of message queuing applications from one platform to another.

Time-independent applications

With message queuing sending and receiving applications are de-coupled so that the sender can continue processing without having to wait for the receiver to acknowledge the receipt of the message. The receiving application might be busy when the message is sent. Indeed, the receiving application doesn't even need to be running. MQSeries holds the message in the queue until it can be processed.

Message-driven processing

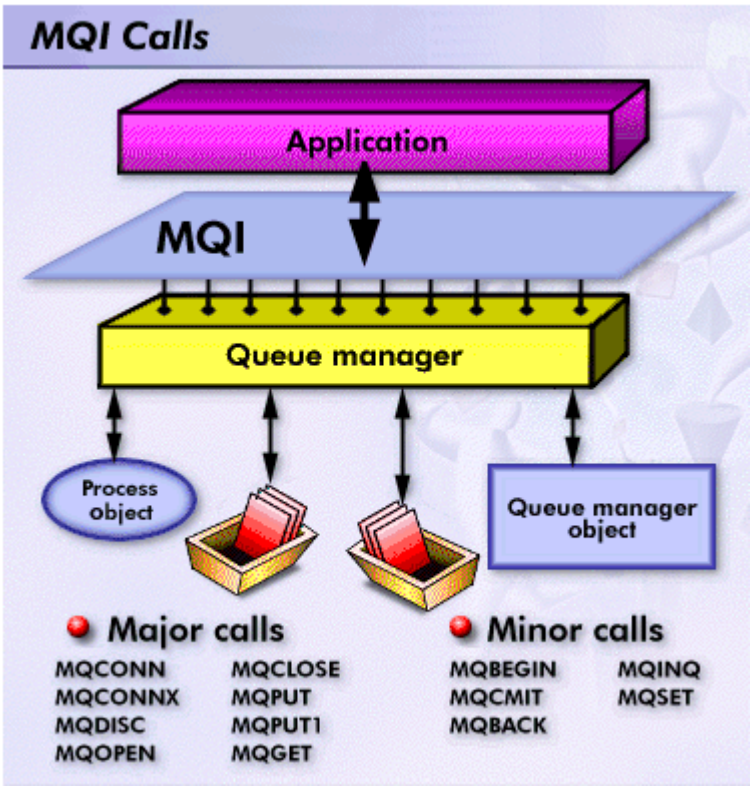
Message-driven processing is a style of application design. With this style, the application is divided into a number of separate, discrete, functional blocks, with each block having well-defined input and output parameters. Each functional block is coded as an application program, with its input and output parameters being interchanged between other application programs by placing their values in messages, which are then put on queues.

This style of application design allows new applications to be built, or existing applications to be modified, more quickly than with some other application design styles.

Data integrity and resource protection

MQSeries applications can transfer data with an extremely high degree of confidence. Message delivery can be implemented using a '*syncpoint*' mechanism and the MQSeries logs or journals-for the recovery of important data in the event of system failure.

Figure 2 D*epicts the high level architectural understanding of MQSeries*



1.5. MQSeries objects

Queue manager

A *queue manager* is that part of an MQSeries product that provides the messaging and queuing services to application programs, through the Message Queue Interface (MQI) program calls.

The applications issue the MQI calls that are implemented by the queue manager. For incoming messages, the queue manager directs them onto their respective destination queues; for outgoing messages, x sends the message to the destination queue manager. The destination queue manager ensures that the message is put onto the correct queue.

Queues

A *message* is a string of bytes that has meaning to the applications that use the message.
A *queue* is a MQSeries object that can store messages.

In MQSeries, there are several types of queue object:

- A local queue object defines a local queue belonging to the queue manager to which the application is connected.
- A remote queue object identifies a queue belonging to another queue manager. The remote queue is usually given a local definition.

- An alias queue object enables applications to access queues by referring to them indirectly in MQI calls. This enables you to change the queues that applications use without changing the application in any way; you merely change the alias definition.
- The model queue object defines a set of queue attributes that are used as a template for a dynamic queue. The queue manager creates dynamic queues when an application makes an open queue request specifying a queue that is a model queue.
  Dynamic queues can be of two types, Temporary and Permanent.

Distribution lists
A distribution list provides a way for an application to send a message to several destinations with a single MQPUT call. The list of destinations is supplied by the application.

Process definitions
A process definition object defines an application to a MQSeries queue manager. A process definition object is used for defining applications to be started by a trigger monitor.

Trigger monitors
A trigger monitor is an application that monitors an initiation queue associated with a queue manager. When a trigger message arrives on the initiation queue, the trigger monitor retrieves it. Typically, the trigger monitor then starts an application that is specified in the message on the initiation queue.

Channels
A channel provides a communication path. There are two types of channel, message channels and MQI channels.

Message channels
A message channel provides a communication path between two queue managers on the same, or different, platforms. The message channel is used for the transmission of messages from one queue manager to another, and shields the application programs from the complexities of the underlying networking protocols. There are six types of message channels, Sender, Server, Receiver, Requester, Cluster-sender and Cluster-receiver.

MQI channels
An MQI channel connects a MQSeries client to a queue manager on a server machine. It is for the transfer of MQI calls and responses only and is bi-directional.

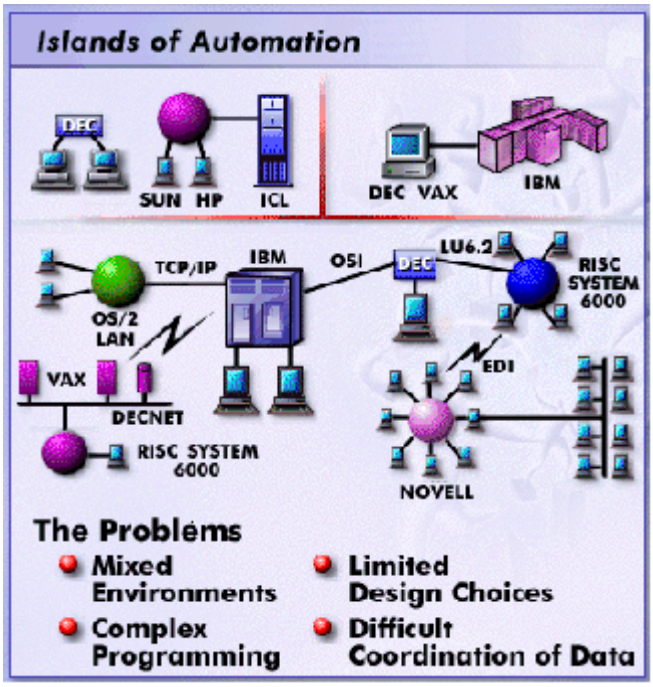
Why MQ?

1.6. Integrating across platforms

Many large organizations would like to like to communicate electronically with their suppliers and customers, but their suppliers and customers might have disparate systems. Even within their own enterprises, many large organizations have IT systems from various manufacturers. The business problem is how to make these disparate systems work together? This problem assumes all the more critical dimensions when we are talking about e-business and Internet presence for the same.

Even where such techniques as OSI exist, there have been severe limitations on the application design choices available. Worse, where data held on different databases on different systems must be kept synchronized, very little is available in the way of protocols to coordinate updates, deletions, etc.

Figure 3 *Islands of Automation*



Many organizations have “legacy “ systems or are running their big terabyte databases on mainframes (enterprise servers), so mainframes need to be included. Integrating these systems is quite complex. There are “one-off” solutions based on OSI or RPC or on SNA, but these are either inflexible in design or need a lot of additional design and code. Finally it is no good exchanging data unless both sides can agree on what has and has not transpired so that their respective databases hold consistent data. For example if you book a hotel room or hire a car, your travel agent needs to know that when his system prints the itinerary, the other guy’s system has updated its database so that when you arrive, the hotel room or rental car is waiting. This may sound trivial

but if the deal were worth $5 million, it would be disastrous if the other could not repudiate a transaction from one side.

1.7. Minimal code rewrite

MQSeries products implement a common application programming interface, the message queue interface (MQI), that is used on whatever platform the applications run on. The calls made by the applications and the messages they exchange are common. It facilitates the migration of message queuing applications from one platform to another.

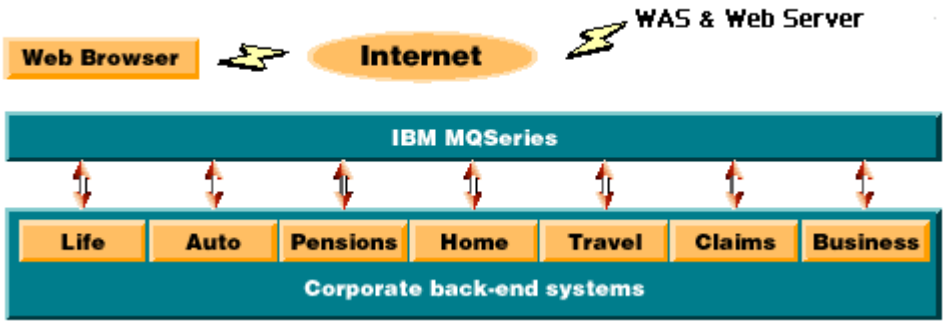
1.8. Accessing back end legacy systems/ processes/ data/ logic

The last time you called your bank to raise the credit limit on your VISA card, did the service representative also offer you life insurance? Or, did your insurance agent advise you how best to invest surplus funds in the capital market?

Maybe your mortgage company offered to consolidate your credit card balances with a home equity loan?

In "the old days," the roles of banks, insurance companies and securities firms seemed clearly defined. Today, those lines of distinction have blurred, and financial services companies pitch themselves as full-service organizations catering to all the financial needs of their clients. To support this convergence of services, these companies are looking to integrate numerous, diverse back-end information systems into a cohesive corporate information cosmos.

Figure 4 ***Integrate back-end infrastructure with WAS***



With an integrated back-end infrastructure, financial services companies can present a consolidated view of their business to their customers, employees, business partners and investors. They can improve customer satisfaction by allowing customers to communicate with them through their channel of choice - say, the Internet, a brick-and-mortar office or a call center and also capture opportunities to cross sell.

You could use Enterprise Java Beans to connect the various front-end delivery channels to multiple, heterogeneous back-end administration systems. The core technology could be IBM

WebSphere Application Server Advanced Editions and IBM MQSeries (on multiple platforms), IBM VisualAge for Java and IBM CICS Transaction Server for OS/390.

1.9. MQ Series for WebSphere applications

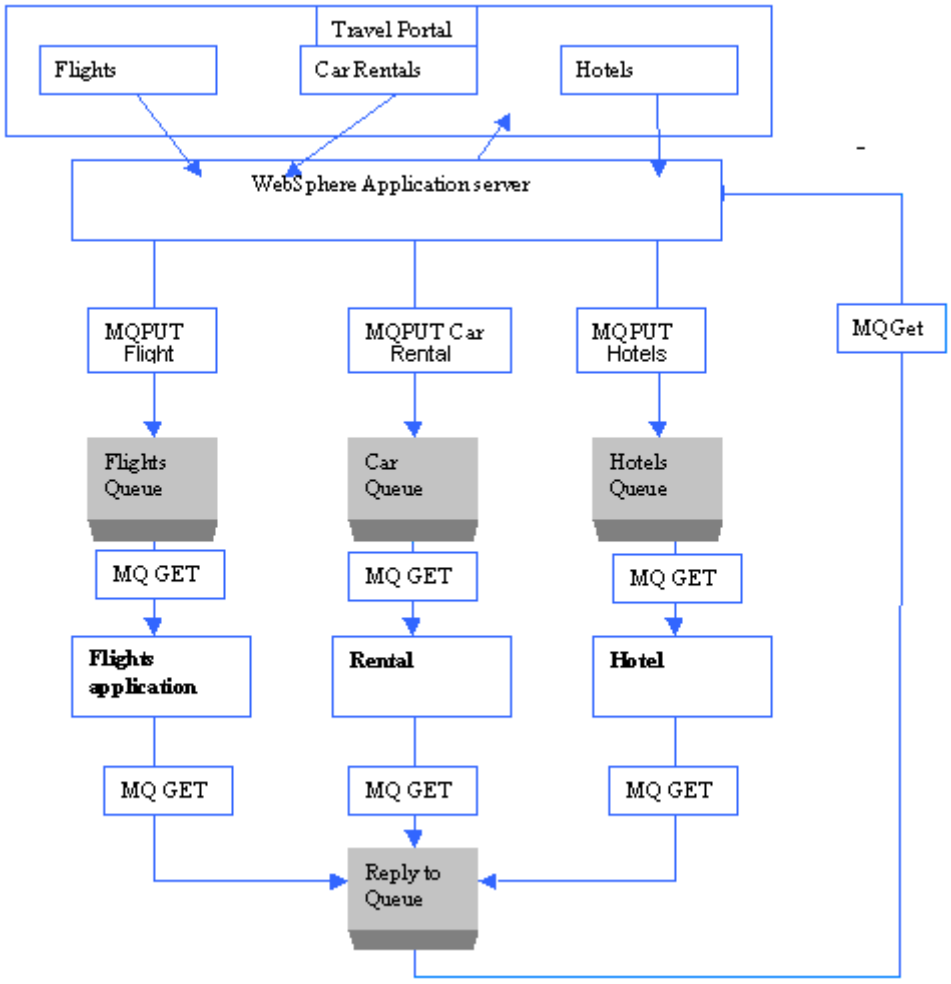
MQSeries provides an excellent infrastructure for access to enterprise applications and for development of complex Web applications. A service request from a Web browser can be queued and processed. This allows a timely response to be sent to the end user regardless of system loading. By placing this queue close to the user in network terms, then the timeliness of the response is not impacted by network loading. In addition the transactional nature of MQSeries messaging means that a simple request from the browser can be expanded safely into a sequence of individual backend processes in a transactional manner.

Advantages of using MQSeries with WebSphere applications

1.10. Increase in performance

MQ Series can significantly improve the performance and response times of co-operating web applications. Consider the scenario of a travel portal. A single customer request might mean calling multiple back end applications. For example, make an airline reservation, book a hotel, run a credit check and so on. All these events must be run before the overall business transaction can be considered complete. Using MQSeries, a request message can be put in each of the three queues which are serving the car rental application, flight reservations application, hotel reservations application and so on. Each application can then perform its respective task in parallel with the other two and put a reply message in the reply-to-queue. The agent's application can then get the three replies and produce a consolidated answer.

Figure 5 ***Agent Application Flow***



This model allows several requests to be sent by an application without the application having to wait for a reply to one request before sending the next. All the requests can then be processed in parallel. Designing the system in this way can improve the overall response time. By using queues, the steps do not have to be performed serially. The application might then wait until it has received all the replies before preparing a consolidated response. Alternatively, the business logic might define a time limit to wait for replies.

1.11. Triggering other processes

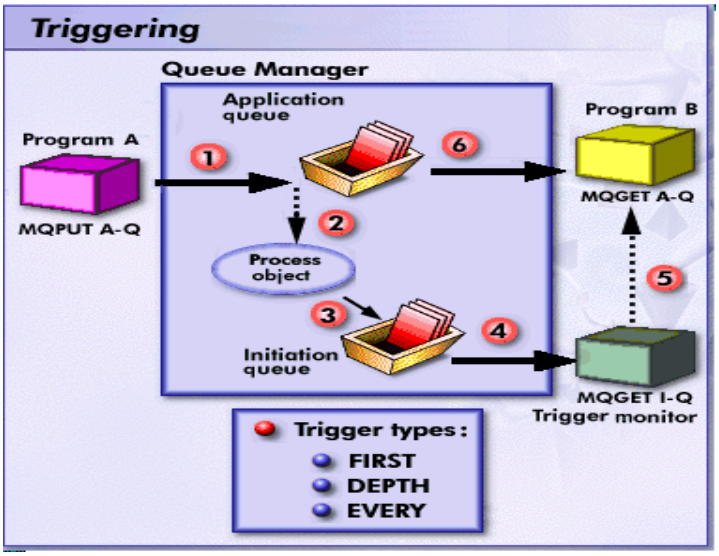
We can use the Trigger mechanism of MQSeries to start some other application to process the queue, which caused the trigger message to be generated. This results in significant performance improvement figures, as we are not continuously monitoring the message queues (which can be an input mechanism to start the second application) to take action.

The queue manager defines certain conditions as constituting "trigger events". If triggering is enabled for a queue and a trigger event occurs, the queue manager sends a trigger message to a

queue called an initiation queue. The presence of the trigger message on the initiation queue indicates that a trigger event has occurred.

The program, which processes the initiation queue, is called a trigger-monitor application, and its function is to read the trigger message and take appropriate action, based on the information contained in the trigger message. Normally this action would be to start some other application to process the queue, which caused the trigger message to be generated.

Figure 6 ***Flow of application and trigger message***



In the above figure, the sequence of events is:

- Application A, which can be either local or remote to the queue manager, puts a message on the application queue.
- The queue manager checks to see if the conditions are met under which it has to generate a trigger event and queue manager examines the process object referenced by application queue.
- The queue manager creates a trigger message and puts it on the initiation queue associated with this application queue.
- The trigger monitor retrieves the trigger message from the initiation queue.
- The trigger monitor issues a command to start program B.
- Application B opens the application queue and retrieves the message.

1.12. Maintains transactional boundaries while sending/receiving messages

MQSeries can participate in unit of work processing in several ways:

- As a stand-alone resource manager coordinating its own resources.
  When MQGET is issued within syncpoint control, the message is not removed from the queue immediately; it is made unavailable to other applications. Only when the unit of work is committed, the message is actually removed from the queue.
- As a transaction manager coordinating its resource and those of some other resource manager.

Using the X/Open XA interface, a queue manager is able to coordinate changes to its resources and to those, of other resource mangers within a unit of work.

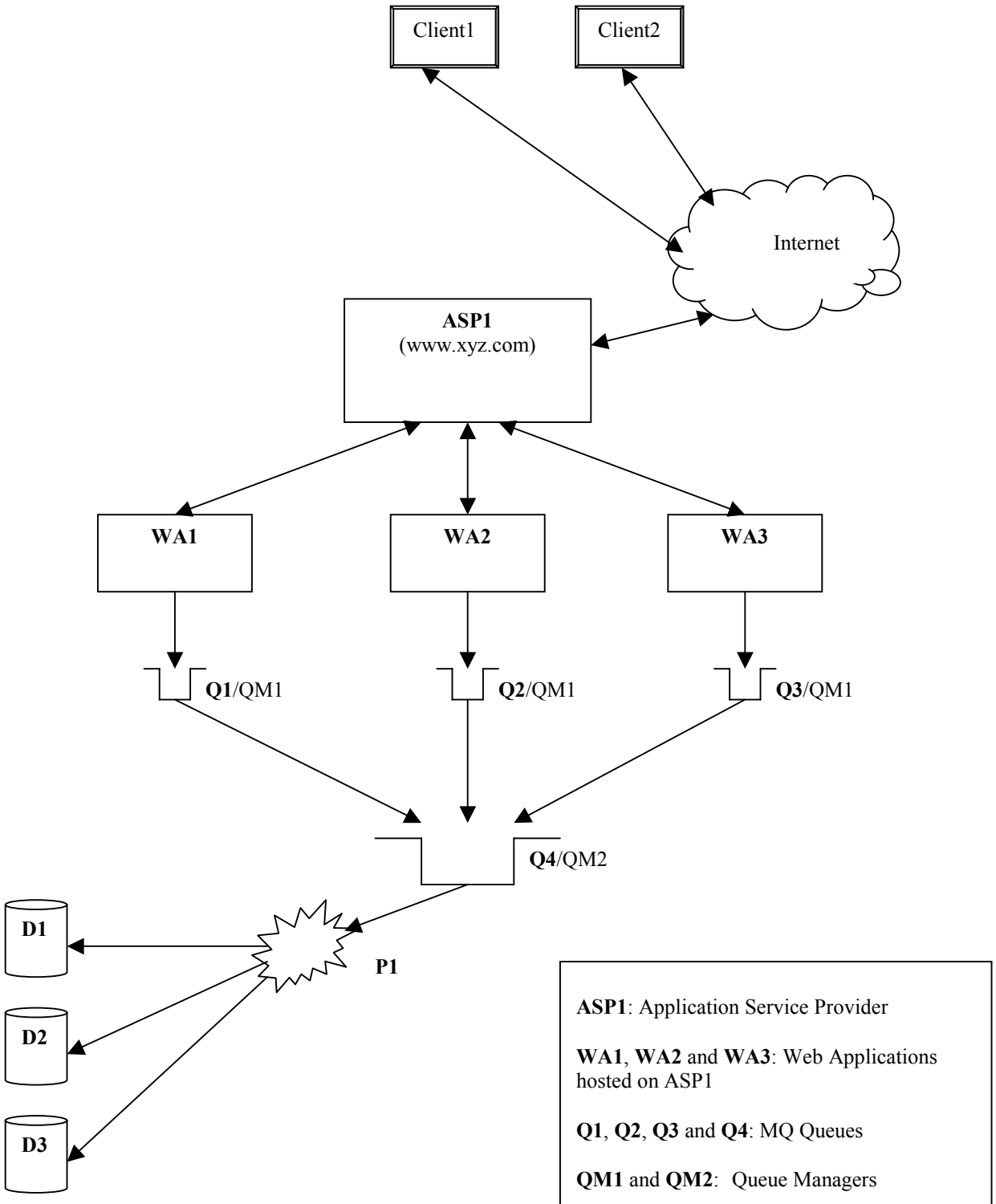
- As a resource manager being managed with other resource manager by a transaction manager.
  An external syncpoint coordinator is required when you have to coordinate those resource managers who are not supported by queue manger.

The following Application Service Provider scenario explains some of the above advantages,

An application service provider (ASP) is a company that offers individuals or enterprises access over the Internet to application programs and related services that would otherwise have to be located in their own personal or enterprise computers. ASP services are expected to become an important alternative, especially for smaller companies with low budgets for information technology.

In our scenario, ASP1 is the application service provider for WA1, WA2 and WA3. Here ASP1 is an instance of WebSphere Application Server and WA1, WA2 and WA3 can be independent Web Applications or WebSphere instances. These applications belong to three different companies who use ASP1's services. Let us assume that all these companies want to sell their services to registered customers across the Internet and charge them accordingly. D1, D2 and D3 are the databases to store the billing information for the applications WA1, WA2 and WA3 respectively.

Figure 7 *ASP Scenario*



The customers would be charged on monthly basis, based on their usage of services. However, transparent to the customers their usage and service charges have to be stored in the database. This operation needn't be synchronous. So the applications dump the billing information in separate queues (Q1, Q2, and Q3). These queues in turn populate a common queue (Q4). Q4 after reaching a specified queue depth (number of messages in a queue) triggers a process. This process populates the respective database (D1, D2 or D3) based on the source of the message (WA1, WA2 or WA3). The information in the database may later be sent to an agency like CyberCash.

QM1 is the Queue Manager for Q1, Q2 and Q3. QM2 is the Queue Manager for Q4. In the ASP scenario, we can observer some of the advantages of MQSeries,

- MQSeries as a stand alone resource manager

QM1 acts as a stand-alone resource manager coordinating its resources, Q1, Q2 and Q3.

- MQSeries as a transaction manager

Here D1, D2 and D3 are registered with QM2 as XA compliant resource managers. After reaching a pre-specified queue depth Q4 triggers P1, which gets the messages in Q4 and populate them in D1, D2 and D3. The entire process is treated as a unit of work and QM2 coordinates among the resource managers and Q4.

- Triggering other processes

Q4 after reaching a pre-specified queue depth triggers a process P1. P1 then gets all the messages in Q4 and populates them in D1, D2 and D3. The triggering option in MQSeries avoids unnecessary daemons listening for new messages in the queue.

1.13. Asynchronous processing (Store and Forward Processing)

The MQSeries assured delivery capability ensure that data is not lost because of failures in the underlying system or network infrastructure. So you can store your requests in queues and processed when possible.

In the ASP scenario discussed earlier, the billing information for the customers has to be updated in the database. As the customers would be charged only on monthly basis, the updating process can be asynchronous. MQSeries fits in this model and also, it assures delivery of data even in case of system or network failure.

1.14. Workload distribution

Using MQSeries' clustering (from MQSeries v5.1), you can benefit from

- Increased availability of your queues and applications
- Faster throughput of messages
- More even distribution of workload in your network

The following scenario will illustrate the workload management performed by MQSeries,

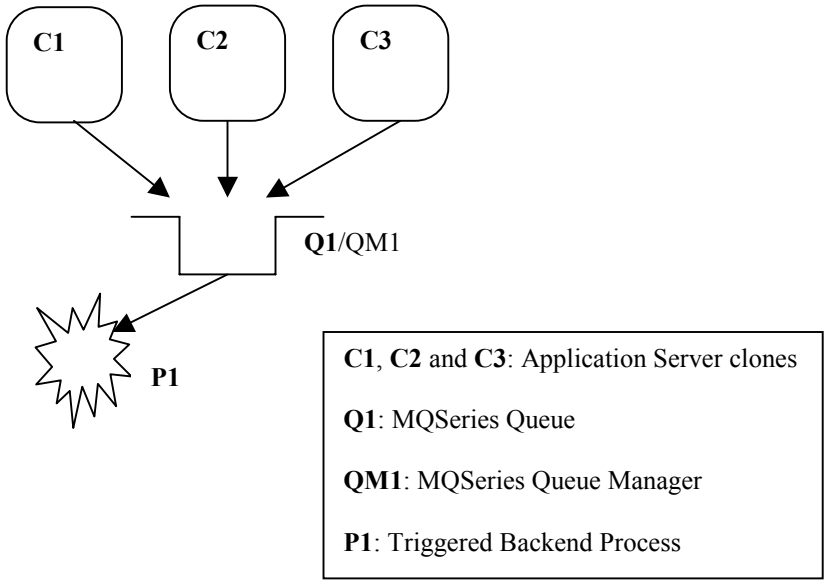


Figure 8 ***WAS Clones without MQSeries Clustering***

Here C1, C2 and C3 are WebSphere application server instance clones. The application servers are cloned for effective workload management. The application needs to put messages in a queue (Q1). Based on the triggering condition set on Q1 a process P1 would be triggered for backend operations. Though workload management is taken care from the WebSphere front there would be a potential bottleneck when it comes to the MQ queue part.

Introducing queue clusters in the picture can solve this bottleneck. The following figure is with queue clustering.

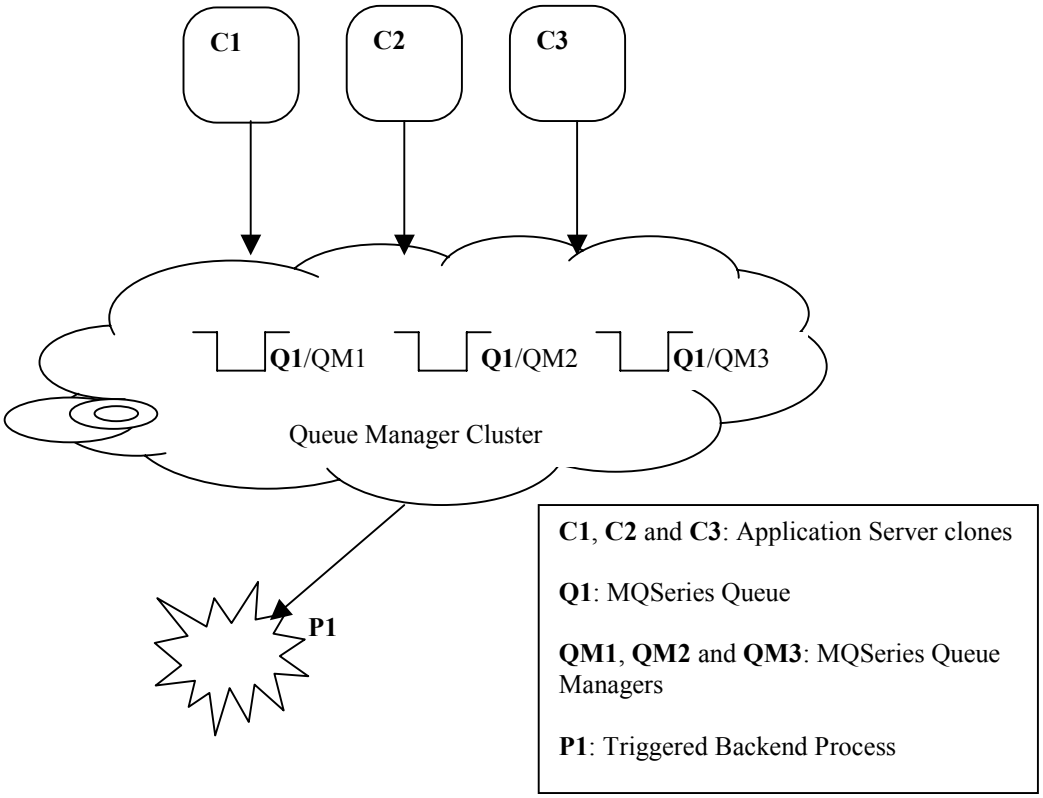


Figure 9 *WAS Clones with MQSeries Clustering*

With clusters the same queue name can be used across queue managers. The queue managers in a cluster can be across machines/platform. The application server clones needn't have knowledge of the cluster. They would put messages in a queue named Q1. MQSeries distributes messages *round robin*. However, our own workload balancing routine can be used.

Each of the three queue managers owns a queue with the name, Q1. By default, the first message is placed in Q1 on QM1, the next in Q1 on QM2, the third goes to QM3 and the fourth message to the queue on QM1 again. Since all the queue managers are in a cluster, even if one queue manager (say QM2) is down, the information is passed on to the other queue managers. Now, no message would be put in Q1/QM2. A change in status of QM2 would be published to the queue managers in the cluster.
With the triggering option enabled in the queues, a backend process would be triggered based on the triggering condition set.

Appendix

1.15. Case Study - I

Objective

To come up with a high performance and reliable solution for your web application, to perform asynchronous activities like logging, tracing, batch operations etc.

Asynchronous processing means that the exchange of data between the sending and receiving applications is time independent. This allows the sending and receiving applications to be de-coupled so that the sender can continue processing, without having to wait for the receiver to acknowledge that it has received the data.

Logging is an asynchronous activity. *Logging* is the process of recording changes during the runtime of an application. The changes can span from any user intervention to any changes in the application itself. The changes can be recorded in any persistent media like file or database. This information can be used later for report generation or fail recovery. *Logging* is a complex process wherein it may be required to handle file operations, database updates, I/O and database errors. The overall performance of the application will reduce if the application handles logging by itself.

You can avoid these complexities by separating the web application and logging logic. A separate web application can take care of logging. Even then the *Logging* module would be part of your WebSphere instance. The following points have to be considered before going in for such a module,

- There could performance issues related to response time, hardware capabilities, network, etc. It would be a burden on the WebSphere Application Server's overall performance.
- If there is a change in the *Logging* logic, then the application server might have to be brought down, to implement the new logic.
- The events to be logged can be stored and forwarded in batches. But then the module needs to handle file/database operation.
- The *Logging* Web Application has to be running always, whether or not logging happens.
- The module needs to ensure redundancy and consistency in data delivery in general and in case of system failures.

To add a *Logging* module to an existing application, the ideal approach would be to use MQSeries in the scenario. Doing so would solve all the above issues. This document describes a practical approach to develop applications wherein the *Logging* module can be totally independent of the WebSphere application.

Scenario

The scenario circles around three applications,

- e-Bank Web Application.
- MQJavaBean a Java Bean
- LogManager a MQ base application

e-Bank Web Application

e-Bank is a WebSphere application, based on servlets, JSPs and EJBs. The application provides the following Internet banking functionality,

- User can create new account
- User can login using his user ID and password
- User can withdraw/deposit/transfer money
- User can view his account balance and transactions

When the user leverages on any one of the above functionality, there is a change in the state of the e-Bank application. This change is an *event*. For example, when a user logs in, the number of '*current user parameter*' is changed. This is an event. The application described here is logging of such events and using MQSeries for the same.

Here we log the following events,

- Login
  The user logs in to the e-Bank application
- Logout
  The user logs out of the e-Bank
- Sign Up
  A new user signs up with e-Bank
- Deposit
  A user deposits money in to a valid account in e-Bank
- With Draw
  A user withdraws money in to a valid account in e-Bank

MQJavaBean a Java Bean

All MQ related operations like connecting to the Queue Manager and Queue, put and get message etc. are implemented in this Java Bean. This approach provides clarity and modularity. It also helps to handle all types MQ related exceptions, to add or modify MQ related functionality in a single place.

LogManager Application

Log Manager is a MQ based application that has the logging logic. This application is responsible for getting event information using MQSeries and put this information in log files corresponding to the event type. This module also takes care of any I/O errors that might occur due to the file operations. The LogManager is triggered by the queue based on the queue depth (number of messages in the queue).

The MQSeries queue used for this application is termed LOG QUEUE.

Overview of the flow

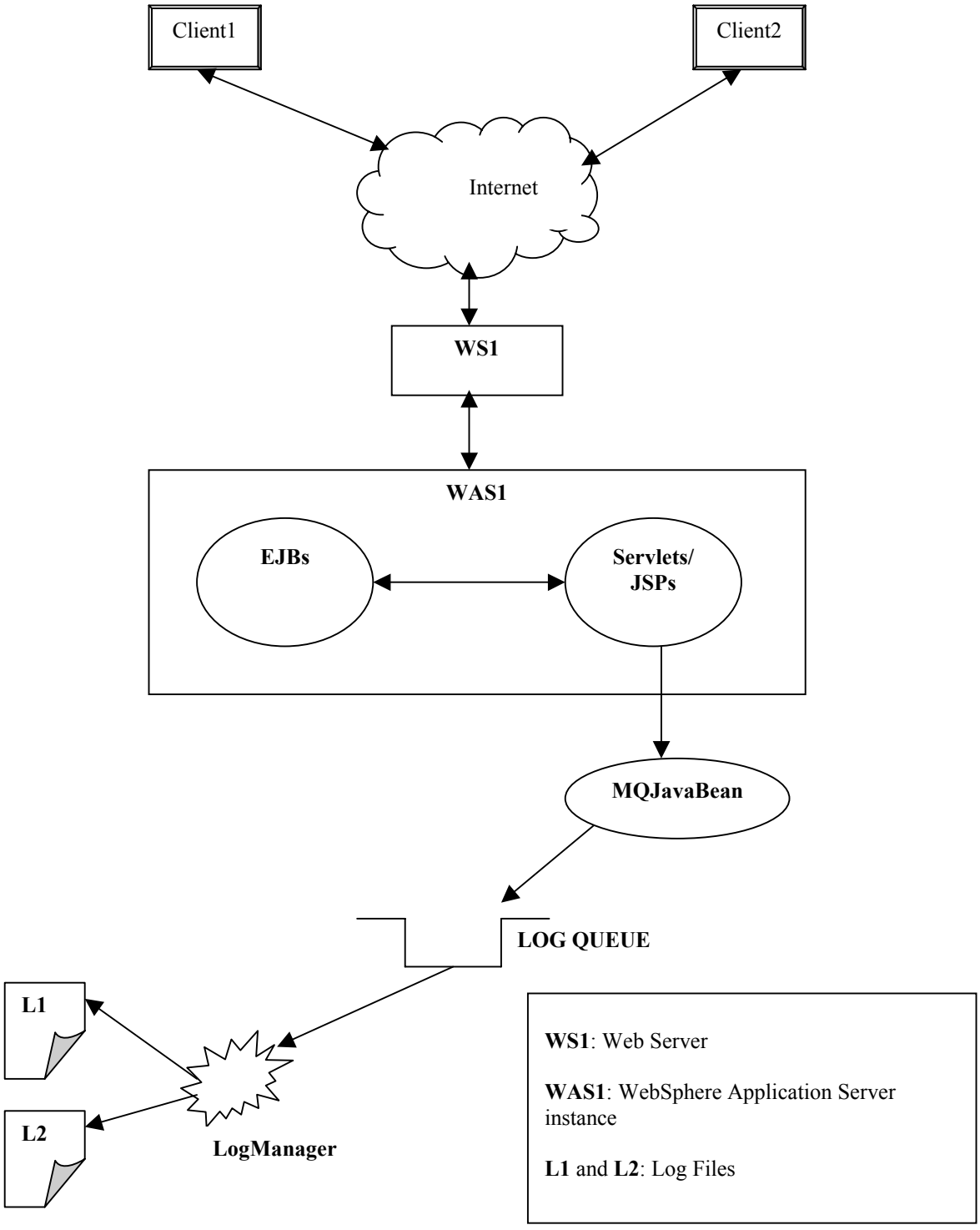


Figure 10 ***Control flow – Case Study I***

- The MQJavaBean is initialized in the 'init' method of the servlets where an *event* may occur. Similarly the MQJavaBean instances are destroyed in the destroy methods of the servlet. Initializing the MQJavaBean would create a handle to the MQSeries Queue Manager and opens a connection to the LOG QUEUE. Destroying would close the LOG QUEUE connection and destroys the Queue Manager handle. The MQJavaBean instance is used in the entire life cycle of the servlets for putting messages to MQSeries queue.
- The parameters of each event are encapsulated in separate objects. This separation allows the LogManager to identify the different events and log them appropriately.
- When an event occurs, the servlet instantiates a new event type object corresponding to that event. The event related information is set in the object and put to the LOG QUEUE. A different *Message Identifier* range is set for each event.
- MQSeries's triggering mechanism is leveraged in this application. The trigger is set for the LOG QUEUE. So, when 100 messages are collected in the LOG QUEUE the LogManager application is triggered.
- The LogManager gets all the event information from the LOG QUEUE. It then writes the events in separate log files based on the event type. The LogManager identifies the different events through the *Message Identifier* range set for each message.

Advantages of using MQ for logging in e-Bank web application,

- De-couple the logging logic from the web application. Hence,
  - Modularity of the logging application.
  - Asynchronous processing.
- Using MQ Triggering eliminates the listening overhead of the LogManager application and also automates the logging process.
- Chances of loosing or duplication of messages is zero because MQSeries assures, once-only delivery of message.
- The WebSphere application can use the Java APIs for MQ, whereas the LogManager application can be written in C/C++/Java/Cobol etc. Because MQSeries has APIs for almost all languages.
- LogManager application may be reside on same machine with WAS or on a different machine which may or may not be of the same platform. MQSeries has common APIs (**M**essage **Q**ueue **I**nterface) for almost all platforms (36).

1.16. Case Study - II

Objective

To use MQSeries for failover in case a synchronous call does not go through in your web applications.

Success of an e-business application entirely depends on customer satisfaction. To achieve this your application should provide robust, fast, convenient and efficient services. Almost every e-business application has a backend system it can be a database legacy system or any other server. Speed, robustness, and efficiency of your e- business application highly depends on these backend systems' performance.

A failure happens when the connection between the Web Application and the backend system goes down. This could happen due to system failures, busy backend systems, etc. Your e-business application should be able to handle such conditions.

MQSeries offers asynchronous processing and assures delivery of message even during system failures. This case study explains a complete, robust and efficient model for *failover*.

Scenario

The scenario uses the following applications,

- e-Shop Web Application.
- MQJavaBean a Java Bean
- FailOverManager Application

e-Shop Web Application

e-Shop is a very simple Web Application. It provides the user with an order entry form, wherein the user can choose items and their quantity and place an order.

The item availability, price and other details of the item are stored in the database. After the user selects the items the application displays the total amount that would be charged from the user, if he places an order. To place the order, the user enters his credit card number and submits the details. e-Shop inserts this order information (user information, purchase details) in a database table. If the record insertion fails, then the order information is put in a MQ queue. e-Shop uses EJBs for all database related operation.

MQJavaBean a Java Bean

All MQ related operations like connecting to the Queue Manager and Queue, put and get message etc. are implemented in this Java Bean. This approach provides clarity and modularity. It also helps to handle all types MQ related exceptions, to add or modify MQ related functionality in a single place.

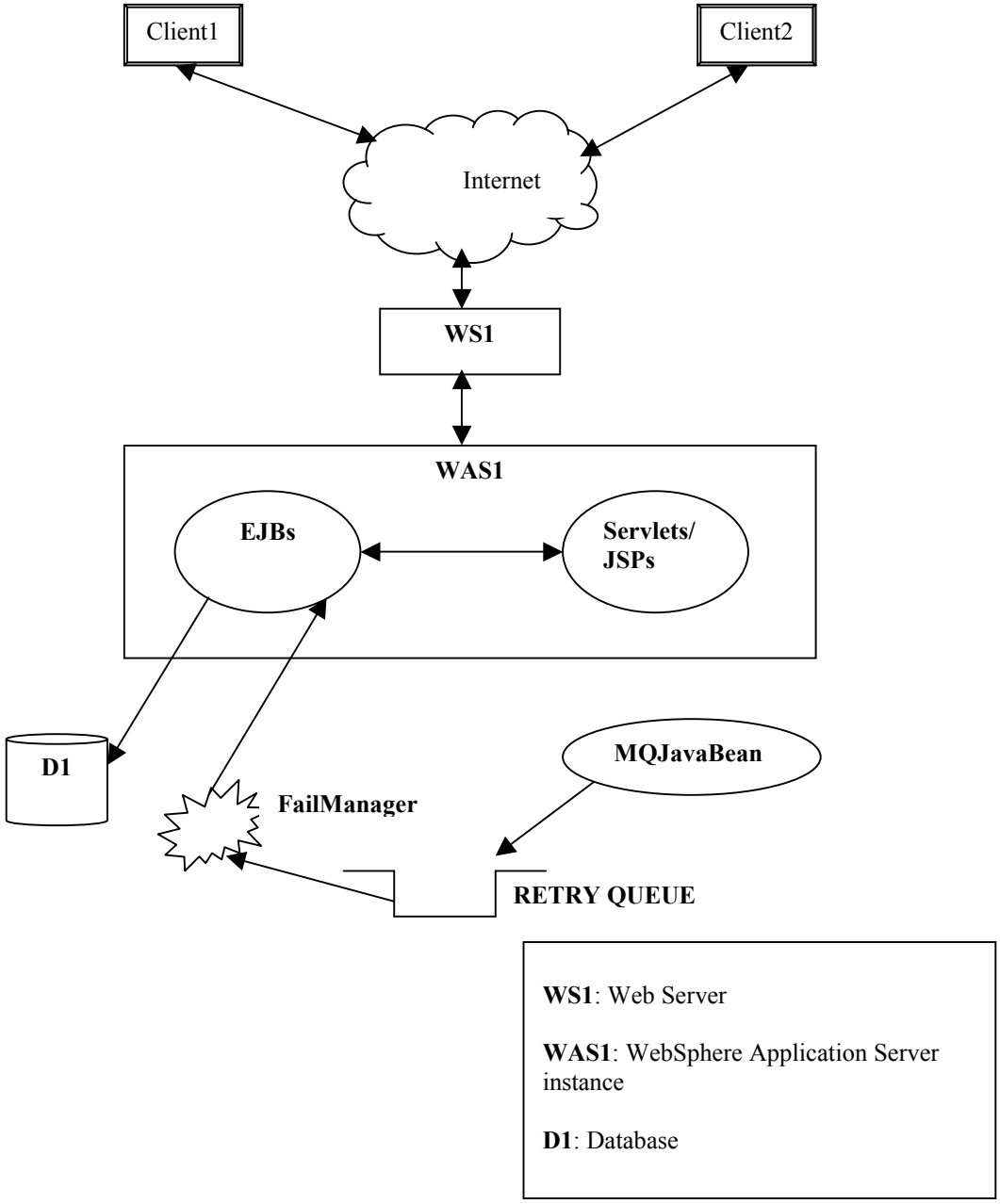
FailOverManager Application

FailOverManager is a Java application, which uses MQSeries Java Libraries and EJBs. This application is responsible for getting the order information from the messages in the queue using MQI and retry updating the database using the EJBs (the same EJBs are used in e-Shop Web Application) until success.

Similar to the previous cases study, the MQJavaBean is initialized in the init method of the servlet in e-Shop and destroyed in the destroy method. Initializing the MQJavaBean would create a handle to the MQSeries Queue Manager and open a connection to the RETRY QUEUE. Destroying would close the RETRY QUEUE connection and destroys the Queue Manager handle. The MQJavaBean instance is used in the entire life cycle of the servlets for putting messages to MQSeries queue.

Overview of the flow

Figure 11 ***Control flow – Case Study II***



- The web client selects the items and their quantity through the browser and places his order.
- In the e-Shop servlet
  - Get the information from the request
  - Insert order information into the Order table using the EJB interface
- If database insertion fails in the previous step, then the order information is put in the MQ queue, RETRY QUEUE.
- The RETRY QUEUE triggers the FailOverManager. The FailOverManager application retrieves the message from the queue and,
  - Extracts the order information from retrieved message
  - Tries to insert the order information into the Order table using the EJB interface
- From retrieving the message from the RETRY QUEUE to calling the EJB, is in one unit of work. To enable this we need to specify the SYNCPOINT option when we call MQGET from the application.
- If the FailOverManager succeeds in inserting the record (through EJBs), MQCOMMIT is invoked to remove the message from the RETRY QUEUE. Else MQBACK is invoked to rollback MQGET. This leaves the message in the RETRY QUEUE.
- If inserting the record is a failure, FailOverManager spawns a new thread, which would keep trying the same in regular intervals for a specified time limit. This step is for optimization.

Advantages of using MQSeries for failover in the e-Shop Web Application:

- Here we use the transactional features of MQSeries, enabling us to rollback MQGET in case of failures.
- Chances of loosing or duplication of messages is zero because MQSeries assures, once-only delivery of messages.
- The asynchronous processing capability of MQSeries doesn't cause any delay in response to the web clients.
- The triggering mechanism is used to trigger the FailOverManager. This avoids a daemon running to get the messages from the queue.

1.17. Case Study - III

Objectives

- To explore the transactional features of MQSeries and how to integrate them with your web application.
- To enable MQSeries as a reliable communication channel between your web application and a legacy system.

To explore the transactional features of MQSeries and how to integrate them with your web application

Transaction is an essential part of any critical application. Operations like transfer of money across bank accounts has more then one logically related task. All of these tasks have to be successful for data integrity. If one task succeeds while another fails then data integrity would be lost. So, there's a need to keep all such tasks within a transaction boundary.

IBM MQSeries supports transactional messaging. MQSeries can participate in *global* unit of work in which resources belonging to other resource managers, such as XA-compliant databases, are also updated with MQSeries resource. Two-phase commit procedure must be used and this unit of work may be coordinated by the queue manager itself, or externally by another XA-compliant transaction manager.

In your WebSphere application, you can either allow the EJB Server to manage the transactions or they can be managed by the application using javax.transaction.UserTransaction interface (JTA). This case study explores the options for integrating the transactional features of MQSeries and WebSphere. However there is no explicit support in WebSphere Advanced Edition 3.02 for MQSeries. We use JMS implementation from IBM, within our Web Application for communicating with MQSeries. IBM's JMS implementation supports transactional sessions. So, we can perform the MQ related operations within a unit of work. *But, the JMS XA* interfaces defined by Sun Microsystems haven't been implemented by IBM till date (06/22/2000) and support for JMS in EJB is only from EJB 2.0. So, the MQ operations within the Web Application won't be part of either the user transaction created in the EJB clients or the WebSphere (Advanced Edition v 3.02) managed transaction.*

To enable MQSeries as a reliable communication channel between your web application and a legacy system.

Old days are gone when it is usual to apply a single architecture and set of standards to an application, but now its time for Enterprise Application Integration. But the root problem is the lack of Common Architectures. IBM MQSeries is a solution for Enterprise Application Integration because it enables asynchronous, loose coupling of distributed applications.

In this case study, we try to integrate two existing Bank applications. One of them runs on WebSphere and the other on DCICS and they use MQSeries as an asynchronous communication link. Wrapper applications and helper classes were written on both ends, to minimize code rewrite of the existing applications.

Scenario

A customer with e-Bank wants to transfer some money to another account in CICS-Bank. This operation requires two tasks, debit from e-Bank and credit to CICS-Bank. Both these tasks have to be in a unit of work. To enable this, we need a distributed transaction. Its not possible to achieve this with our current setup (WAS AE3.02, MQSeries5.1 and DCICS – TXSeries4.2). So, we can have a set of three transactions,

- Debit from e-Bank and MQPUT in a MQ queue. The message put will consist of the CICS-Bank account number and the amount to be transferred.
- MQGET, credit the transferred amount to the CICS-Bank account and MQPUT the acknowledgement message. The acknowledgement will have the information, whether the credit was successful or not.
- MQGET and if a 'failure' acknowledgement, credit the transferred amount back to the e-Bank account.

e-Bank Web Application

e-Bank is a WebSphere application, based on servlets, JSPs and EJBs. e-Bank is an existing application. The application provides the following Internet banking functionality,

- User can create new account
- User can login using his user ID and password
- User can withdraw/deposit/transfer money
- User can view his account balance and transactions

In our case study we would be using the transfer feature of e-Bank. Using this an e-Bank account holder can transfer money to other accounts in e-Bank or CICS-Bank.

When the user chooses to transfer amount to a CICS-bank account, the servlet invoked debits the amount specified from the user's e-Bank account's ACCOUNT_TABLE. Apart from that, the servlet also creates a record of the transfer details in the CLEARANCE_TABLE. When the user wants to view the current status of his account, information is pulled out from both the ACCOUNT_TABLE and CLEARANCE_TABLE and displayed to the user.

The different modules on the e-Bank front,

Account EJB

All operations regarding an Account like deposit, withdrawal, getting balance and transferring the amount and type of the account etc, are implemented in Account EJB.

Transaction EJB

Records all the transactions involved with an account. Each time the bean creates a record it also creates a unique transaction ID.

Process EJB

In our case, if the transfer operation failed, the amount withdrawn from the e-Bank account has to be deposited back to the source account. Since the entire process of transfer is asynchronous, there would be data inconsistency between the database and the balance amount in the user's current session. In order to remove this inconsistency we have come up with the concept of *funds*

in clearance. The *funds in clearance* holds the total amount transferred to CICS-Bank accounts, by an e-Bank account holder. For every account, we have the account balance and funds in clearance in the current session. The Process EJB takes care of this functionality.

MQJMSBean

All MQ related operations like connecting to the Queue Manager and Queue, put and get message etc. are implemented in this Java Bean. IBM's JMS implementation (MA88) is used for the MQ related operations. This approach provides clarity and modularity. It also helps to handle all MQ related exceptions; to add or modify MQ related functionality in a single place. *The other alternative in the future would be EJB 2.0 (yet to be released). The Enterprise JavaBeans 2.0 specification extends the EJB architecture with new features including the integration with the Java Message Service. So, with EJB 2.0, we can use a JMS EJB instead of a JavaBean taking care of JMS operations.*

TransferAck

TransferAck is a standalone Java application. This application would be triggered upon receiving a message in the TRANSFER.ACK QUEUE. Upon triggering, the application uses the MQJMSBean to retrieve the message from the queue. If the message was a 'failure' acknowledgement, then the application credits back the amount withdrawn from the e-Bank account. Upon receiving the message from the queue, this application removes that particular transaction record from the CLEARANCE_TABLE using the Process EJB.

CICS-Bank Application

CICS-Bank is a simple DCICS application on Windows NT, which portrays a bank model. The application provides functionality to create accounts, credit money, debit money and transfer money across CICS-Bank accounts. CICS-Bank is an existing application.

The different modules on the CICS-Bank front,

CICS-Bank is a set of three CICS Server Programs[1] (Create, Debit, Credit) written in C++. Each of these, map to a CICS Transaction[2]. CICS-Bank also has a common library, consisting of 4 classes,

- Account class: Provides an interface for all account related activities like, creation of accounts, credit, debit and transfer of money across accounts.
- Person class: Provides an interface for all customer-related information like, adding and getting information about them.
- Database class: In CICS, we can map a database table to a CICS File[3]. This class provides an interface for reading from and writing to a file.
- Application class: Interface for reading from and writing to the COMMAREA[4].

The application flow is as follows,

- The CICS Client (an ECI Program) invokes the banking application after writing the user's requests/information to the COMAMREA
- The CICS-Bank uses the *Application* class to read the request/information from the COMMAREA
- Based on the request CICS-Bank invokes the Create/Debit/Credit function using the *Account* class

- Then user's account information is updated in the database using the *Database* class
- After updating, CICS-Bank writes the status of the transaction in the COMAMREA.
- The CICS Client can now read the COMAMREA to know the status

Wrapper to CICS-Bank

In traditional CICS applications standard ways for inter process communication is using COMMAREA. CICS-Bank application like any other traditional CICS application gets the data form COMMAREA and after processing writes back data to the COMMAREA.

In our case study, we are using MQSeries for communication between the web application (e-Bank) and the CICS-Bank. The wrapper is basically a CICS program that enables data transfer between the CICS-Bank application and the MQ queue.

The functionality of the wrapper is,

- To get the message from the MQ queue (WAS.TO.CICS QUEUE)
- To extract data from the queue and write it to the COMMAREA
- To pass the control to the CICS-Bank Server Program
- To retrieve the data from the COMMAREA after the execution of the CICS-Bank Server Program
- To create a MQ message using the data and put back an acknowledgement message in the MQ queue (TRANSFER.ACK QUEUE).

A transactional boundary is maintained, which starts from the point of getting the message from the queue and putting back the acknowledgment in the queue.

Note: You have to register the MQ Queue Manager with DCICS as a XA compliant resource manager.

The scenario in our case study is to transfer money from e-Bank to CICS-Bank. So we would be using the '*WrapperCredit Program*'.

BankECI Program

The common ways of invoking a CICS program/transaction are, through a non-CICS application, by another CICS program or using the CICS terminal (CICS Client 3270 Terminal Emulator). There are two types of interfaces for calling a CICS program from a non-CICS application, ECI [5] and EPI[6].

In our case study we would be using ECI programs (*BankECI*). The BankECI program would be triggered by the WAS.TO.CICS queue when it receives a message. The functionality of this program is to invoke the *WrapperCredit* CICS Server Program. Since MQ can't trigger a CICS program/transaction explicitly, we use this BankECI Program.

CICS on Windows NT (TXSeries 4.2) - Terminology

1.) CICS Server Programs

CICS offers a standard set of commands used by application programs to request services from CICS regions. This set of commands is called the *CICS application-programming interface (API)*. CICS Server Program is a program, which uses these APIs.

2.) CICS Transaction

A *transaction* is a unit of processing consisting of one or more application programs. When you develop your application program, you associate it with a transaction that is used to request the program. You can then use the CICS Clients or a **cicsteld** command to run the transaction and, therefore, run your application program. The transaction is executed under the control of CICS, which provides the services requested by each API command.

3.) File

Transaction processing systems provide users with timely, accurate, and reliable access to data. Such user data can be in the form of files, queues, and database entries. The files can be either in Structured File Server (SFS) or an RDBMS like DB2. Data in files is organized as a collection of records. A *record* is a grouping of related information with a predefined size and a predefined number and layout of *fields*. Each record in a file has the same number and layout of fields, which hold specific parts of the record's information.

4.) CICS COMMAREA

A communication area (COMMAREA) is a command-level facility used to transfer information between two programs within a transaction or between two transactions from the same terminal.

5.) ECI (The External Call Interface)

The external call interface (ECI) enables a non-CICS application running on the client machine to call a CICS server program running on a CICS region. The client program can call the server program synchronously or asynchronously as a subroutine.

6.) EPI (The External Presentation Interface)

The EPI allows a non-CICS application program to be viewed as a 3270 terminal by a CICS server system to which it is connected

Overview of the flow

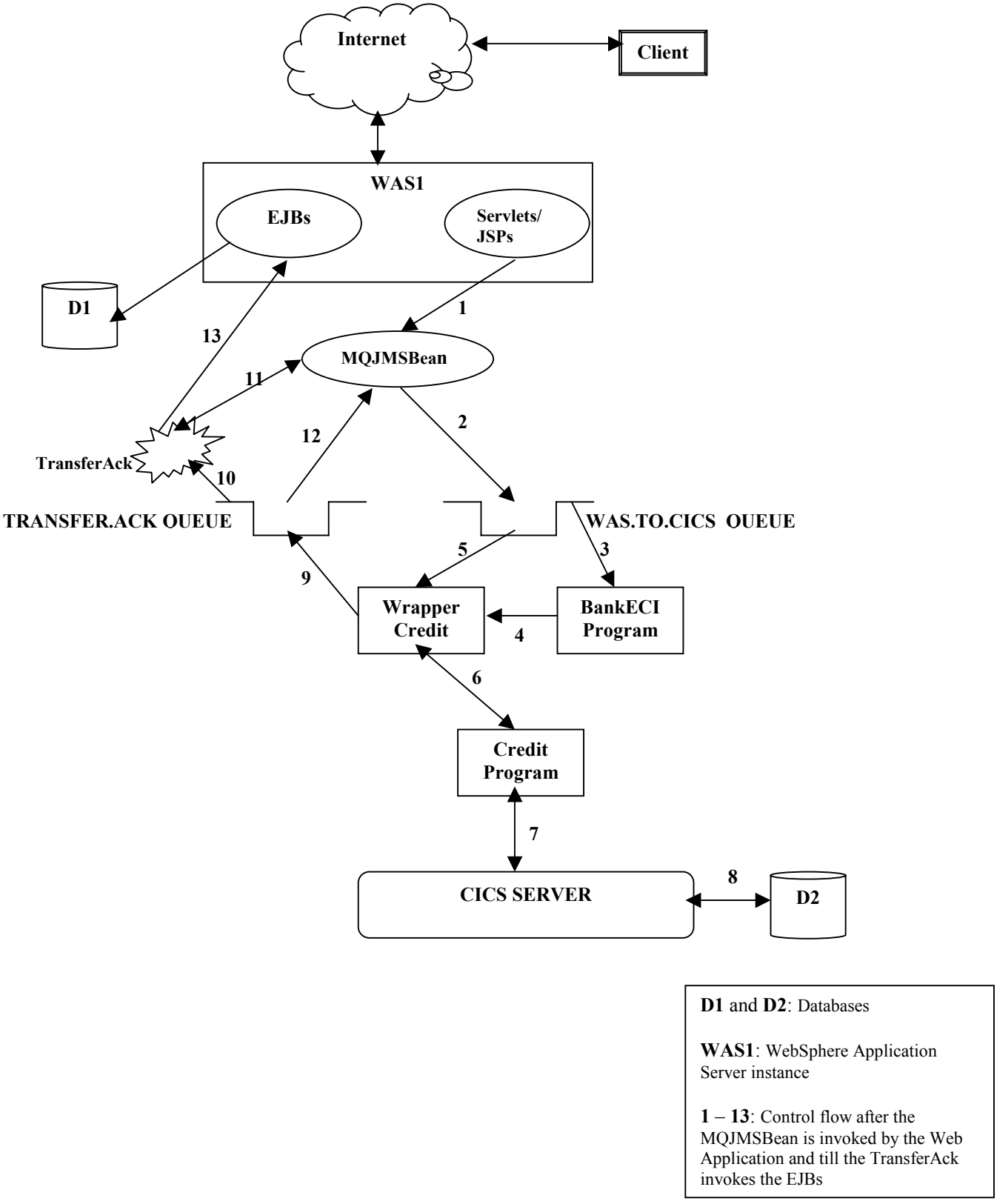


Figure 12 ***Control flow – Case Study III***

- The e-Bank account holder chooses to transfer some amount from his account to a CICS-Bank account.
- The e-Bank verifies his account credibility and withdraws the amount from his account using the Account EJB. The Transaction EJB creates a unique transaction ID and records this transaction.
- The amount withdrawn would be made *Funds in Clearance* and recorded with the generated transaction Id in the CLEARANCE_TABLE using the Process EJB.
- e-Bank invokes the MQJMSBean. The CICS-Bank account number, e-Bank account number, the transaction ID and the amount to be transferred are put in the MQ queue (WAS.TO.CICS) as a message using the MQJMSBean. The message type is *persistent.* ***(Step 1 and 2 in Figure 12)***
- From withdrawing the amount from the e-Bank account to putting the message in a queue forms a single unit of work. The transaction is started in the servlet invoked, when the user chooses to transfer, and propagated to the MQJMSBean and across the Account EJB, Transaction EJB and the Process EJB. *But, the JMS XA* interfaces defined by Sun Microsystems haven't been implemented by IBM till date (06/22/2000) and support for JMS in EJB is only from EJB 2.0. So, the MQ operations within the Web Application wont be part of either the user transaction created in the EJB clients or the WebSphere (Advanced Edition v 3.02) managed transaction.*
- The WAS.TO.CICS QUEUE is configured such that it would trigger the BankECI Program whenever it receives a message. ***(Step 3 in Figure 12)***
- The BankECI Program invokes the '*WrapperCredit'* CICS Server Program and exits. ***(Step 4 in Figure 12)***
- The wrapper program gets the message from WAS.TO.CICS QUEUE. The SYNCPOINT option is set for this particular MQGET call. Doing so brings the MQ operation under the CICS transaction boundary. ***(Step 5 in Figure 12)***
- The CICS-Bank uses a standard data structure for passing information between itself and the CICS Client. The wrapper program leverages on the same structure for its communication with CICS-Bank. The wrapper extracts the information from the message and populates the structure with the data. The data essentially consists of the CICS-Bank account number and the amount to be credited.
- The wrapper program then calls (CICS LINK) the CICS Credit Program and passes the structure (pointer to the structure) to the COMMAREA. ***(Step 6 in Figure 12)***
- The CICS Credit Program reads the information from the COMAMREA and credits the amount to the specified CICS-Bank account. During this process the Credit Program uses the Application, Account and Database classes. ***(Step 7 and 8 in Figure 12)***
- Then the Credit Program updates the data structure with the new account balance and the status of the operation. Depending on the success of the operation the status is either SUCCESS or FAILURE.
- The COMMAREA is updated with the new structure.
- The wrapper program reads from the COMMAREA and based on the status of the Credit operation puts an appropriate message in the TRANSFER.ACK QUEUE. The SYNCPOINT option is set for the MQPUT call. ***(Step 9 in Figure 12)***

- The MQSeries Queue Managers of both the queues are registered with the CICS Region as XA Compliant Resource Managers. So, the transaction is managed by CICS. From the point of MQGET from WAS.TO.CICS QUEUE to MQPUT to TRANSFER.ACK QUEUE is a transaction.
- The TRANSFER.ACK QUEUE is configured such that it would trigger the TransferAck application whenever it receives a message. ***(Step 10 in Figure 12)***
- The TransferAck application uses the MQJMSBean to retrieve the message from the TRANSFER.ACK QUEUE. ***(Step 11 and 12 in Figure 12)***
- The application then extracts the transaction ID, the e-Bank account number, the transfer amount and the status from the message.
- Using the Process EJB, the TransferAck application removes the record from the CLEARANCE TABLE corresponding to the retrieved transaction ID. ***(Step 13 in Figure 12)***
- The previous operation is performed whether or not the transfer operation was successful.
- If the transfer operation was a FAILURE, the TransferAck application credits back the transferred amount to the e-Bank account using the Account EJB. ***(Step 13 in Figure 12)***
- From getting the message from the TRANSFER.ACK QUEUE to depositing the transferred amount to the e-Bank account (in case of FAILURE) forms a single unit of work. The transaction is started in the TransferAck application and propogated to the MQJMSBean and the Account EJB.

Advantages of using MQ

- MQSeries Triggering mechanism eliminates the listening overhead of the BankECI program and the TransferAck application. It also helps to automate the process.
- MQSeries can act as a resource manager being managed with other resource managers by a transaction manager. On the CICS-Bank front, we have registered the MQSeries Queue Managers as XA Compliant resource managers with the DCICS Region. In this scenario DCICS acts as the Transaction coordinator for its own resources and MQSeries' resources.
- Chances of loosing or duplication of messages is zero because MQSeries assures once-only delivery of message. So, even though the money transfer from the e-Bank to the CICS-Bank is broken into three different transactions, with MQSeries it appears to be a single global transaction.
- Since MQSeries is available on many different platforms, integrating your Web Application with a backend system is made simple with MQSeries.
- MQSeries and Web Application integration is made possible with IBM's JMS implementation.