TheServerSide
Your J2EE Community

Published on TheServerSide.com August 19, 2003

Introduction

The JMS API introduces a standard way of incorporating messaging technology to application servers. J2EE application servers can now appreciate the type of application architectures that other types of applications did; such as messaging applications that obtained different levels of decoupling, allowing client server applications to communicate with mainframe applications.

This article addresses some of the application architectural issues in applications that use messaging and JMS in general. It also recommends some better architectures and solutions.

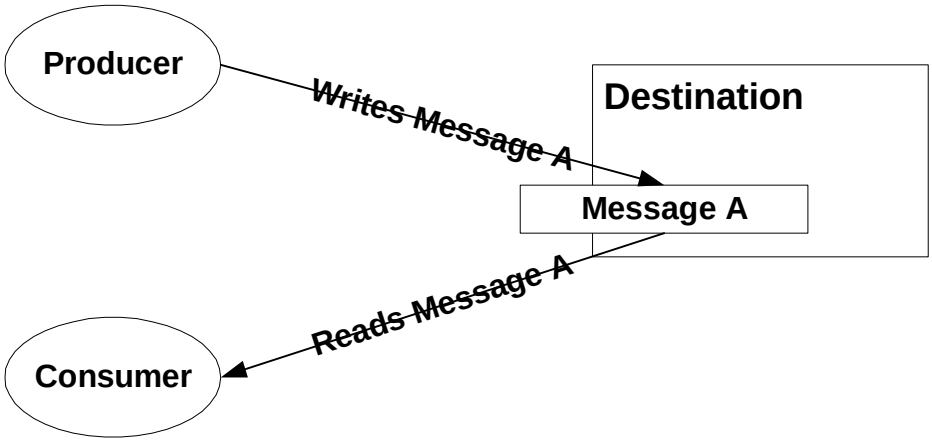
Understanding what it means to be loosely coupled

Messaging technologies really focus on one way delivery. Applications send messages to some destination. Another application can then read the message from that destination. Usually, the application that delivers the message to the destination is called a producer. The application that reads the message is called the consumer. Since producers and consumers rely on the destination, their processes or threads are loosely coupled.

Loosely Coupled Run State

Being loosely coupled is a term that has been thrown around loosely (no pun intended). What does it mean to be loosely coupled? Well it depends what you are talking about. As far as messaging goes, the processes (or running threads) are loosely coupled. This means that one process (producer) can be up, send a message to a destination, and not worry about the consumer of the message being up. Conversely, the consumer can read a message even after the producer has written a message. (See figure 1).

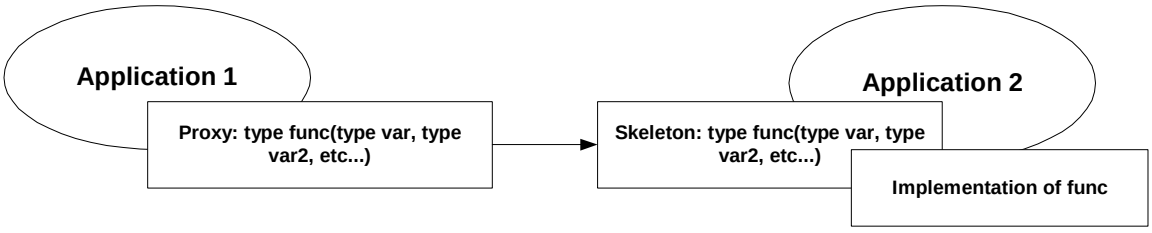
Figure 1:



As you can see, the running states of the Producer and the Consumer are totally independent of each other. The Consumer can run on a different day than the producer and the transaction can succeed.

Loosely Coupled Applications

Messaging technology also creates another type of loose coupling, application level coupling. Because each application is only concerned about reading and writing messages, they usually worry about the API needed to write or read the message. Since JMS is a standard API, the coder knows that changes in the API are less likely to change over time. Instead, messaging carries the business interface between applications. This shifts the application integration from interface centric to data centric. What does this mean? Let's examine the typical RPC (Remote Procedure Call) type integration. RPC technology is used in technologies like EJBs, CORBA, DCOM, RMI, etc… These applications rely on a programmatic interface shared by the requester (client) application and the service (server) application. The client application compiles the interface along with proxies that implement the interface. The proxies hide the complexity of all the underlying communication. These proxies usually make a direct network call into the same application that services the request, or at least a broker such as an ORB that delegates the request. See Figure 2:



Right away you see how the applications lose process de-coupling. Because the application eventually makes a network call into the service application (either directly or

through some delegate like an ORB), you can see that if the service application is down, the request will fail. Sometimes this is ok depending on the requirement. Web applications where users request data to be displayed on a screen will benefit from this. If the remote system is down, the end user will not need to wait. He or she will know right away if the service is down or not available. However, why is this system not application loosely coupled? Because RPC systems are interface centric, both systems share a programmatic interface with parameters and data types. The client application is now bound programmatically to the server. If the server application changes the interface, the client application is now in an inconsistent state. If the requester makes a request to the changed server, the results are unpredictable.

With data centric applications, this does not happen the same way. Because applications that are data centric usually read and parse data, they code for errors in the data. If the contract in the data breaks, the application expecting the wrong format will treat it as a data error and not break. The error conditions are handled gracefully.

What does this have to do with messaging? Well messaging has to be data centric because applications don't communicate with each other directly/conversationally. However, RPC systems can also be data centric. If the two remote systems agree to pass the data as one message parameter, then the application can benefit from application loose coupling. This is the strategy used for example when communicating SOAP messages over HTTP. However, messaging combined with data centric applications gives you a combination of process level and application level decoupling. Because systems are now not process-bound and not programmatically-bound, 2 remote systems that need to communicate do not rely on each other to run.

Understanding Asynchronous Use Cases and Models

Now that we described what loosely coupled systems are, let's talk about some traditional messaging architecture. Before JMS and Message driven beans, there was no standard API for messaging. IBM® MQSeries (now IBM WebSphere MQ™), for example, had different APIs in different languages while MSMQ had another. Much architecture arose from using these APIs. As Web Application Servers such as WebSphere arrived, applications used these asynchronous APIs to communicate with legacy systems such as CICS applications.

The process level de-coupling of the systems provided many benefits. Legacy systems are usually slow moving in their development cycles. It takes much more to roll out systems in legacy environments such as a mainframe. On the other hand, Web applications tend to have a much faster development cycle. Furthermore, legacy systems tend to have different up and down periods. Messaging provides the level of process decoupling needed for these different environments.

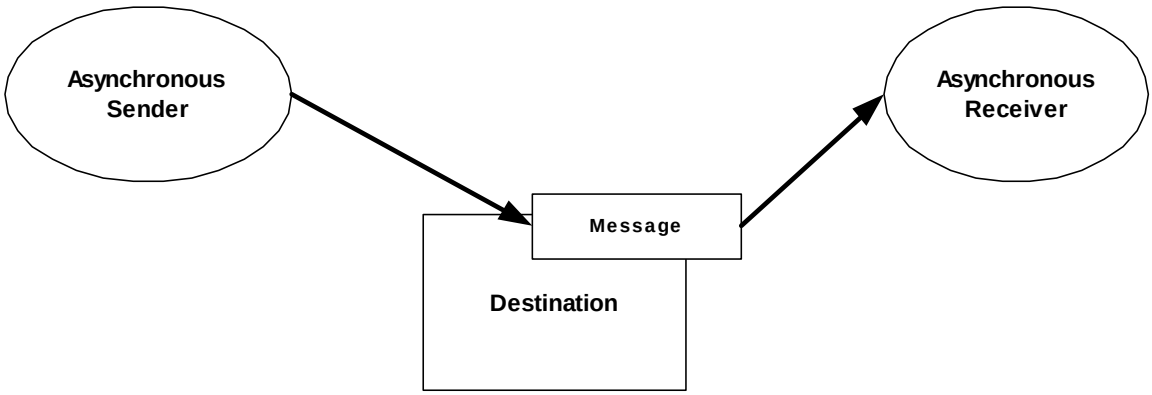
Because these applications rely on messages, they are also data-centric. So Web applications tend to be application de-coupled from legacy systems. Everything may seem fine but there are some very problematic issues that arise. Because Web sites communicate through HTTP which is synchronous and messaging is asynchronous, there is a conflict in methodology. With messaging, Web users issue requests and expect a reply; you send data and don't wait.

In this section, we will talk about various Web requirements and how they have been traditionally implemented when communicating with systems with an asynchronous protocol.

Asynchronous Use Cases

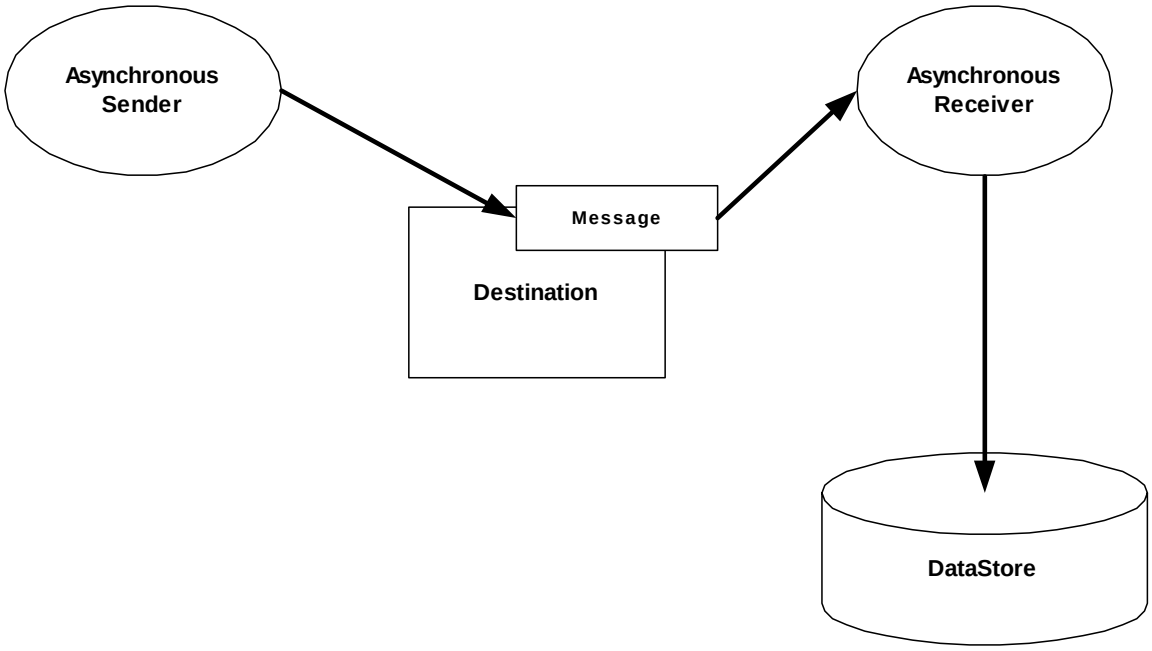
From a requester's point of view, asynchronous messaging means that one process or thread sends a message to a destination and expects no reply. From a consumer's point of view, asynchronous means receiving a message and not sending a reply immediately to the sender.

This is the ideal scenario in messaging. For the most part, this scenario applies for database updates; however, some specialized inquiries can be made as well. Usually one application or thread sends some data to another application

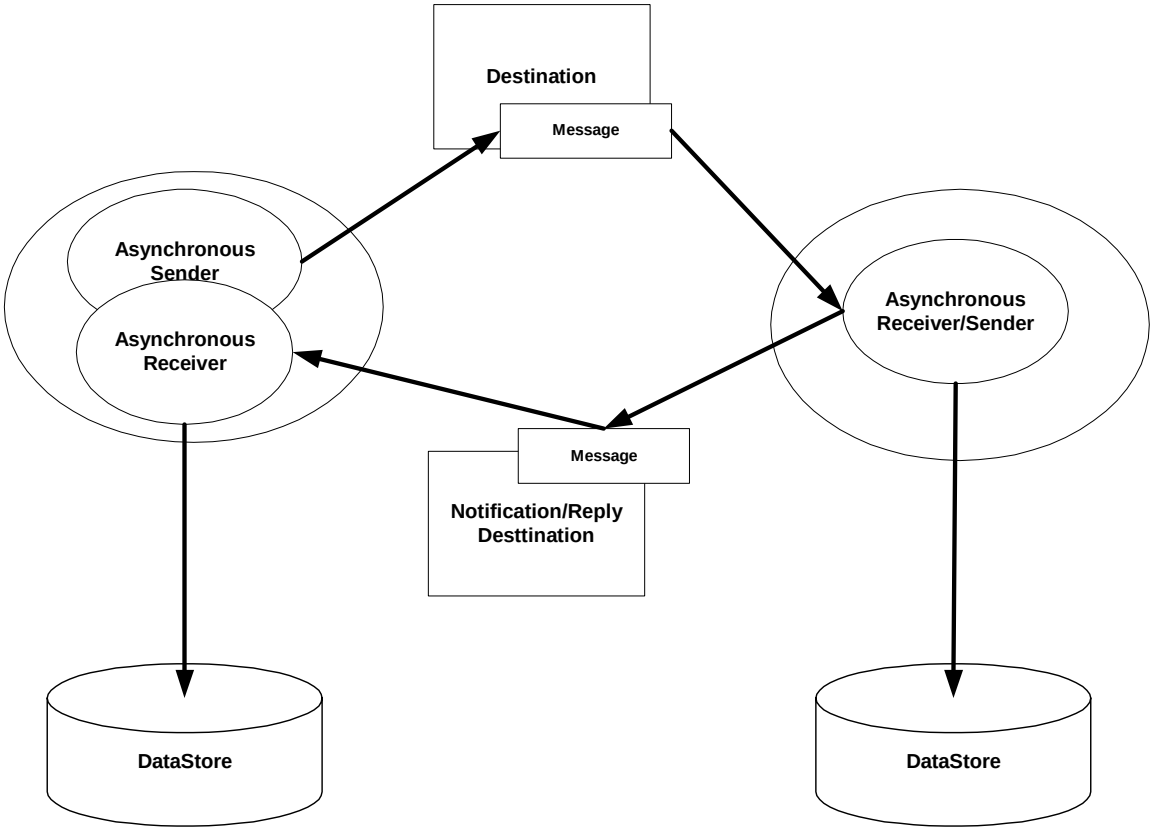


This is ideal for JMS. Let's look at the update case. In many cases the sender is sending some data for the receiver to update in its system. The asynchronous receiver just receives the messages, parses them and may update some data store. This is called Fire and Forget because the sender sends the message and forgets about it. The sender can rely on features such as guaranteed delivery to ensure the message arrives. However careful consideration must be taken on validation. In order for this to work, the sender must make sure the message is valid. If the receiver is serving multiple types of clients,

then this validation needs to be replicated across clients. A change in the validating business rule means that all clients need to change their validation.



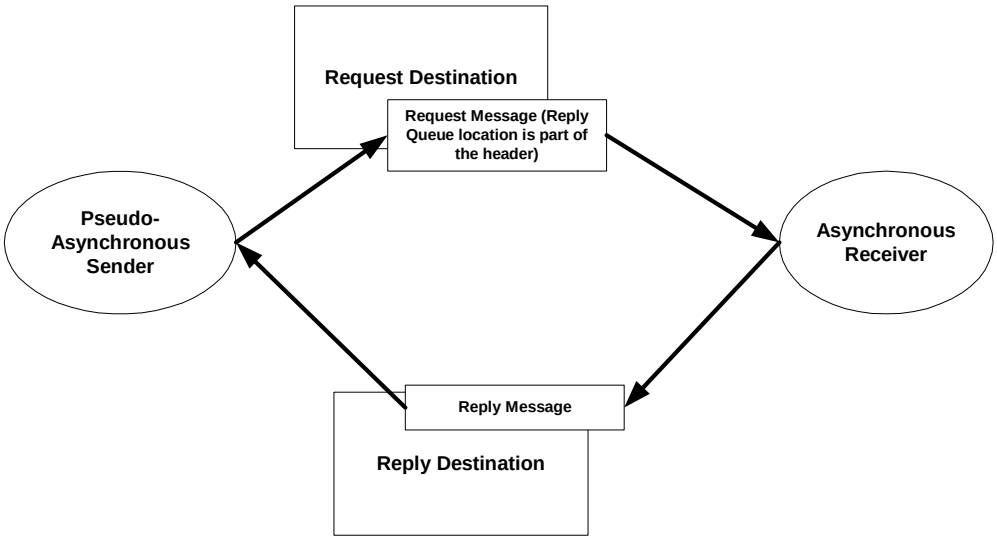
An alternative to fire and forget is to fire and expect a reply sometime later. In this scenario, the asynchronous sender sends an update and does not wait for a reply right away. The receiver will process the update and send a notification to the sender. In essence, both applications become senders and receivers. In this case, the client application sends an update and then returns. The client application also has an asynchronous receiver waiting for any reply.



From the client's perspective, this is considered asynchronous. The client thread that sends the request still deals with Fire and Forget. It does not hang around and wait for a reply. Some other listener thread will pick up the reply message whenever it becomes available. From the server's perspective, this is actually Pseudo-Synchronous. The server process will need to reply and know where to send the reply. The receiver can now handle validation because the process can send some error validation reply.

Pseudo-Synchronous Use Cases (Anti-Pattern)

Pseudo-synchronous is a term I really don't like to use but it is a reality in many applications. Psuedo-synchronous means 'implementing a synchronous use case over some asynchronous interface'. In messaging, this means the client sends a message and then waits for the reply right after sending (on the same thread). Contrast this with the previous scenario, where send and receive are performed on different threads. The figure below illustrates this.



There are a few issues with this scenario. This behavior can be implemented using a remote method call; however, people feel they will lose decoupling. They do not lose application decoupling. Because you are waiting for a reply on the same thread, you have lost running state decoupling already. You can just as easily have a timeout with remote procedure mechanisms.

Nonetheless, there are other reasons people do this; for example, having a single solution for integration or for administrative ease. Companies don't want to have multiple channels into their mainframe for example.

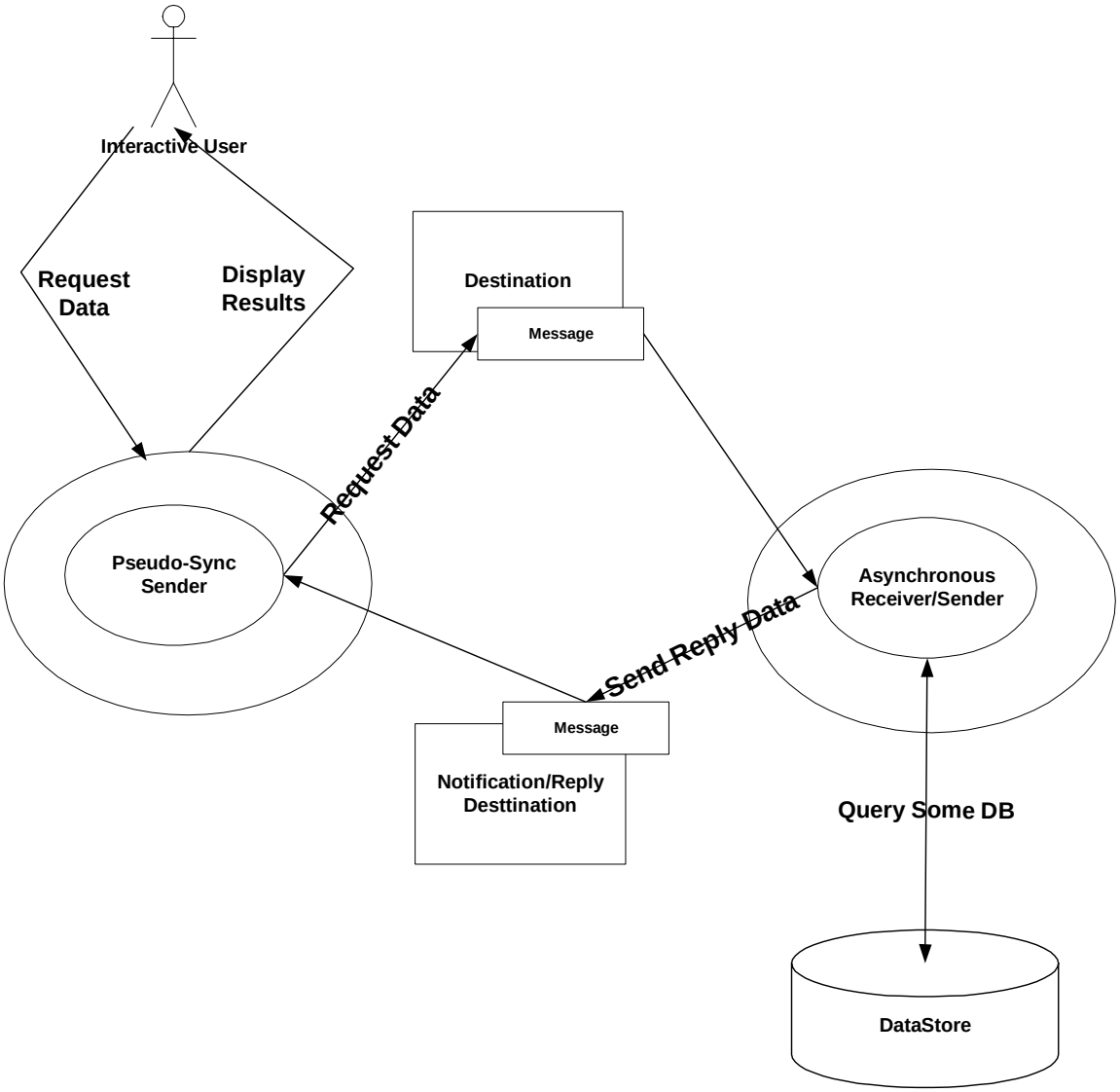
When psuedo-synchronous requests, there are usually 2 types: Inquires or Updates.

Pseudo-Synchronous Inquiry

Inquires usually mean sending a message to the consumer. The sender writes some message with some inquiry encoded and waits on the reply queue for the results of the query. The receiver reads the message, retrieves data from its system, perhaps a database query, formats a response message, and writes a message back to the reply queue. The original sender then receives the result and either displays it to a user or maybe updates its own system.

This seems overly complicated for a query when the decoupling goals are really just at the application level. Remember, because we are waiting for a response, we lose running state decoupling anyway.

Let's look at the case where a Web user is making a query and waiting for a response. Let's say, the remote system you are sending the inquiry to, is down. The sender really has no way of knowing this. The end user now has to wait until the messaging reply timeouts. The user then might get a 'remote system not available' error message.

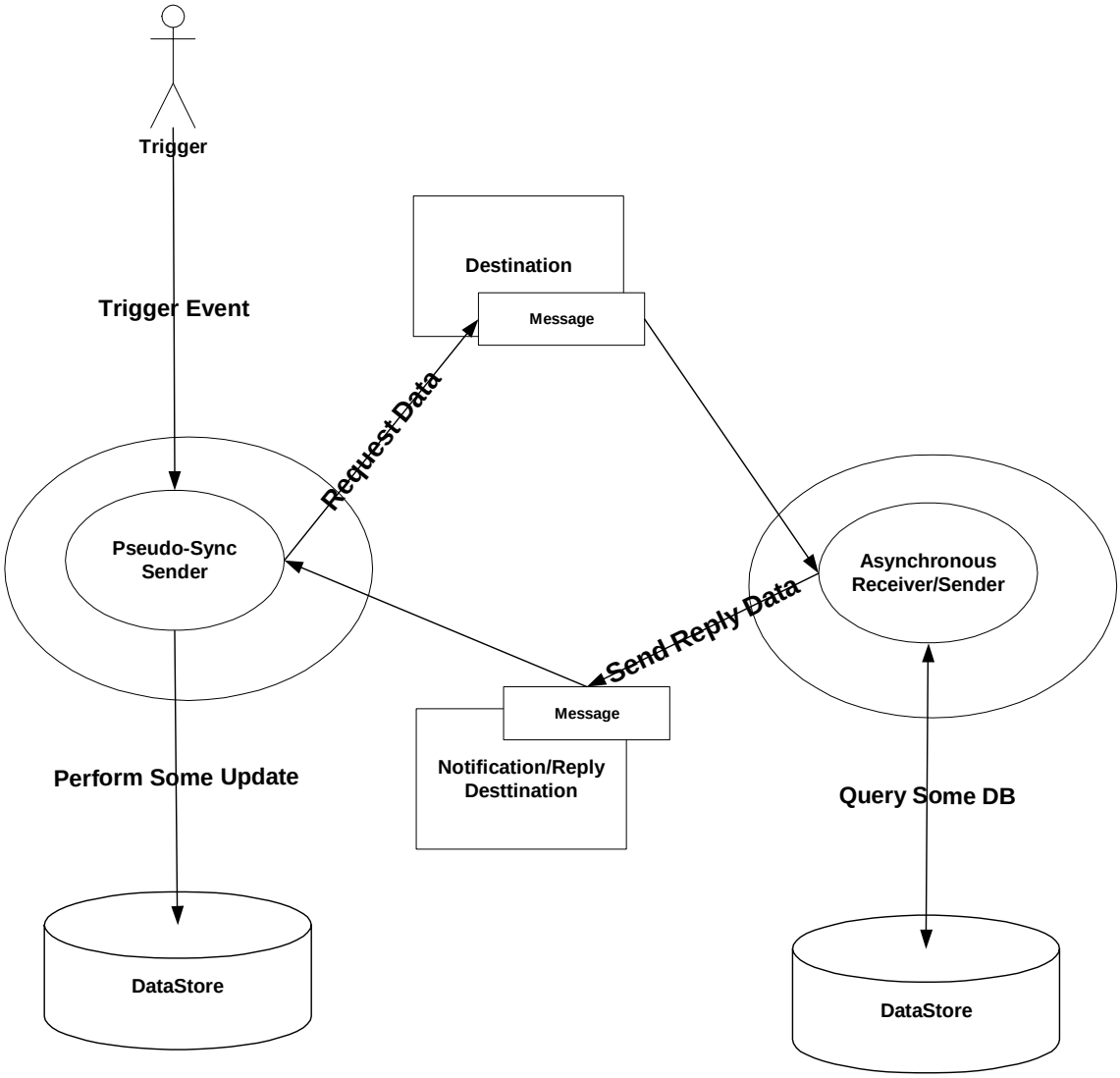

Interactive User
Request Data
Display Results
Destination
Message
Request Data
Pseudo-Sync Sender
Asynchronous Receiver/Sender
Send Reply Data
Message
Notification/Reply Desttination
Query Some DB
DataStore

The end user has to wait perhaps 60 seconds for this message. From the user's point of view, the system is just very slow or cannot handle the volume. This may not be the case, but how can the user know this?

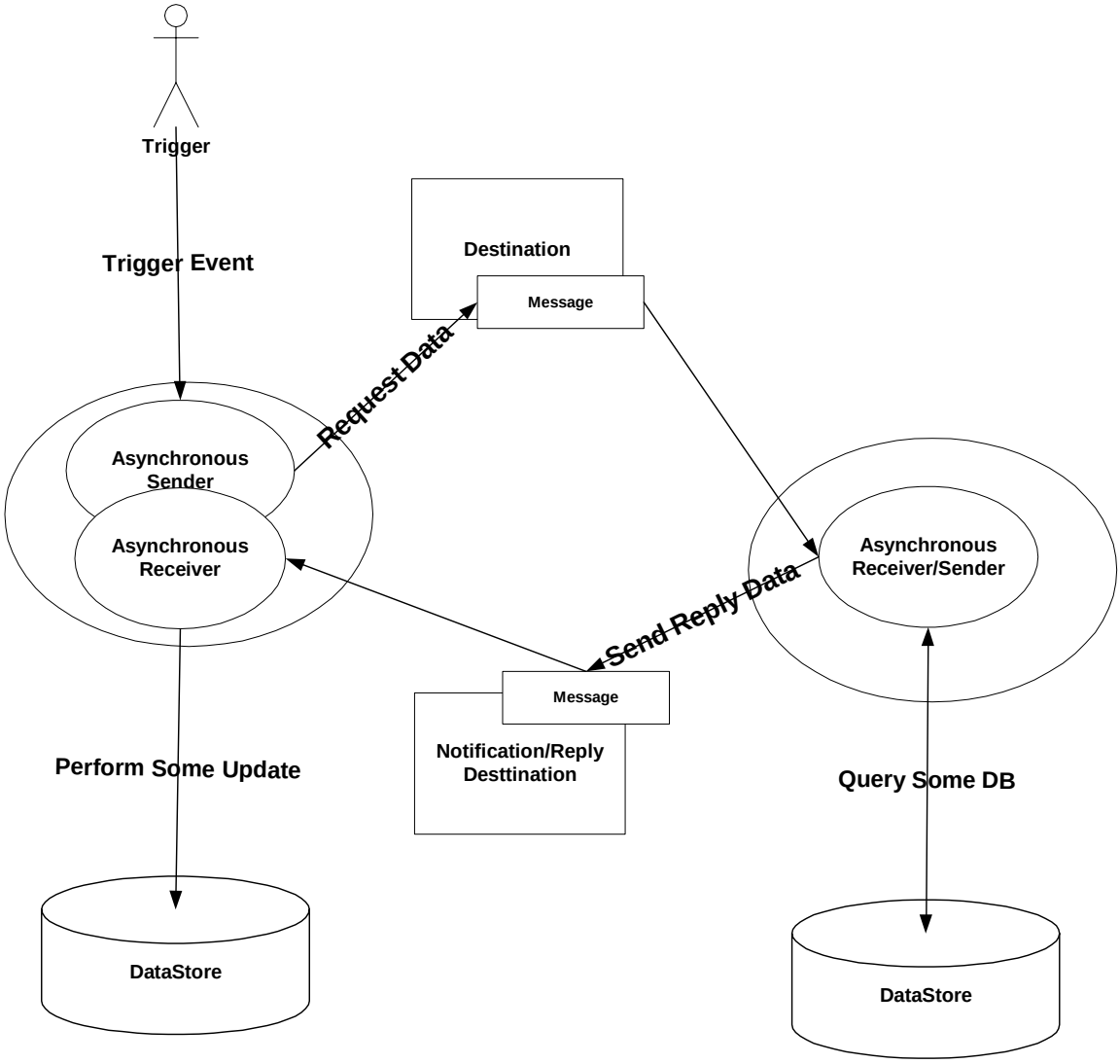
There are other downsides. If the timeout does occur, in most cases, you can allow the reply to expire. Basically, this inquiry is a throw away. The user needs to re-issue the inquiry in most cases. The messaging system now has to use resources to expire the message.

The goal for these systems is to not use messaging. Remote procedure calls are usually better for this scenario. There are other solutions such as JCA or Web Services that can still pass a message as parameters. For application decoupling, all you need is a data centric interface, not a messaging layer. If you cannot avoid messaging, this article offers better solutions later.

For our second inquiry scenario, the sender issues an inquiry in order to update his own system. The interaction can be triggered by an end user or an event.



In this case the user/event is not waiting to look at the data at that time. Instead he is triggering an inquiry from the remote system and then updating his own database with the inquired data. Because the user does not have to wait, having a pseudo-synchronous update is useless. It is better for the client to send an asynchronous request and implement a separate listener for updates.



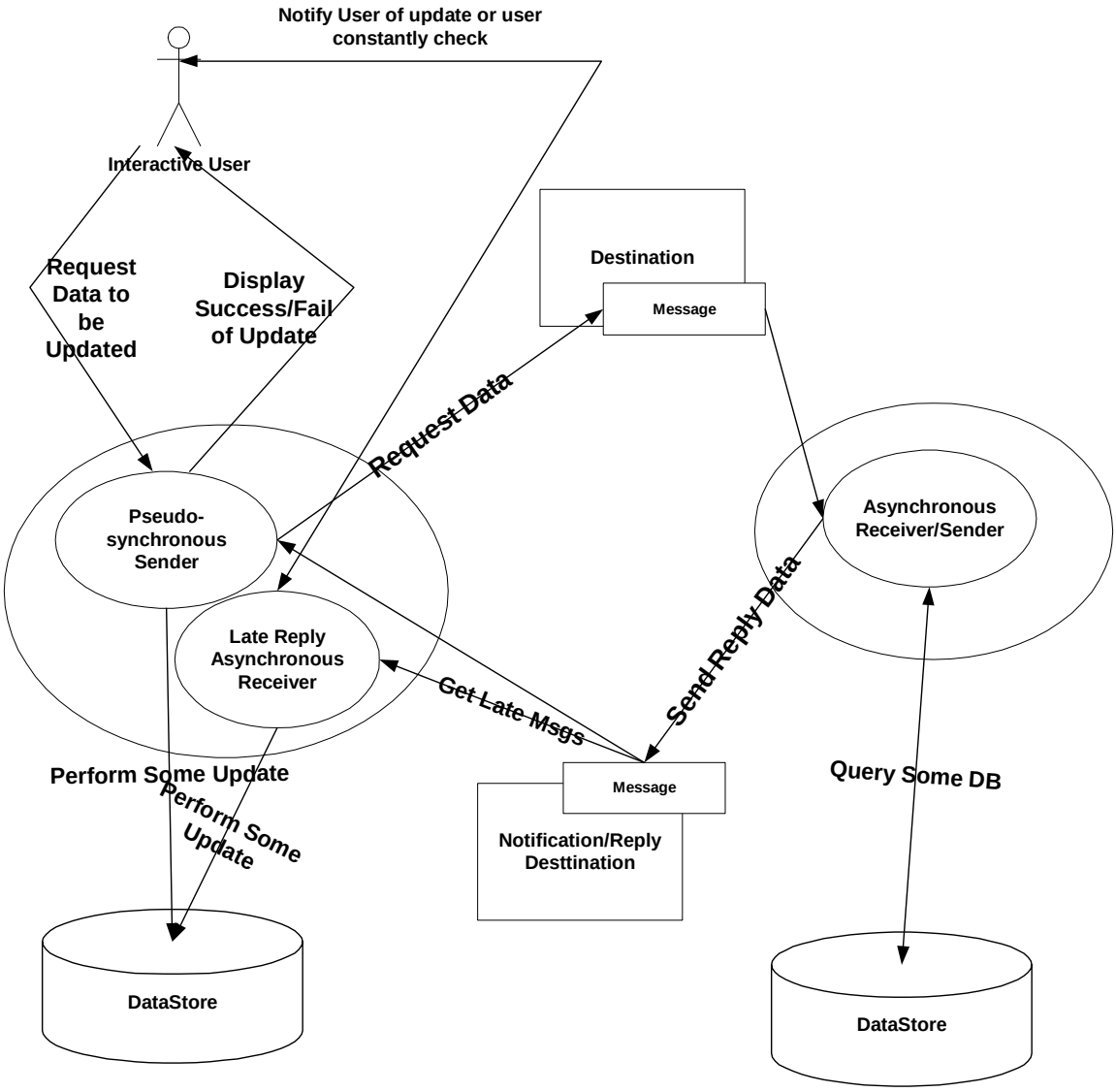
Here the end user, or trigger, is finished once the request has been sent. The asynchronous receiver runs independently of any user or event on the requester's side. If needed, the response receiver can send an email to the user at a later time. Again, we will look at some other solutions further in the article.

Pseudo-Synchronous Update

For updates, the sender is usually sending some data for the remote system to update the data store. So why should this ever be pseudo-synchronous? Well, sometimes the data being sent is not always valid. For example, the user is looking at an old copy of the data and someone else has updated it while he is looking at it. You want the remote system to be able to detect this so that the data is not in an inconsistent state. Or perhaps the user needs to perform some operation based on the successful completion of the update. For example, I can only book the hotel after the rental car has been successfully booked. This

is still not a valid reason why this should be pseudo synchronous. The valid case for pseudo-synchronous is when messaging is the only transport available, and it is absolutely critical that the user know the response immediately after submission.

If the user needs to know the response immediately, let's say because there is a timed deadline and the user needs to correct the error as soon as possible, consider using a synchronous protocol. There are transactional implications, which I will talk about later, and also the problems created by messaging. The situation can become hairy. If the update made it successfully, but the transport goes down before the reply, you are in a bind. Because the receiver is still thread state decoupled, he has no idea the transport went down. This means he can perform the update, and the reply gets delayed. When the sender times out, he cannot assume that he can throw away the reply or that the update did not happen. Basically, the sender system now needs to implement another listener just for late replies.



Unfortunately, architects have to live with some of these extreme cases. The first thing you always want to try is to make the use case asynchronous. However, many times, this requires adding validation on the publisher of the message guaranteeing that the message is valid. This is sometimes not ideal because the client needs to replicate business logic from the server. If you have multiple clients, you have to repeat writing the same code. If there is a change in the validation logic, all the clients need to change accordingly. This is not a realistic solution.

The best way to move to asynchronous is by convincing the user that he or she can wait an extra second for some event or email. Sometimes, this takes a change in philosophy. People like to submit some Web request and know the response immediately. However, having the user rely on notification is just as productive. Many Websites such as shopping cart type applications usually send an email notification to confirm some order.

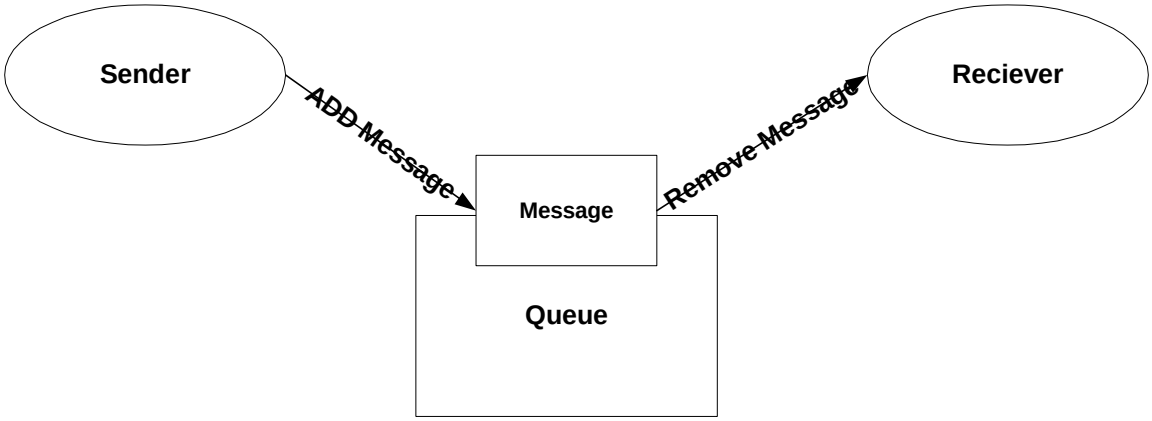
The Pseudo-Synchronous Update scenario should be avoided, if possible; however, later I will offer a better alternative if you need to make Pseudo-Synchronous updates.

Messaging Models

As messaging technologies have evolved, 2 types of messaging models have emerged: Point-to-point messaging and publish and subscribe messaging. The JMS API supports both models. To understand messaging today in the J2EE world, it is important to understand the differences between the 2 models and the problems that each solves.

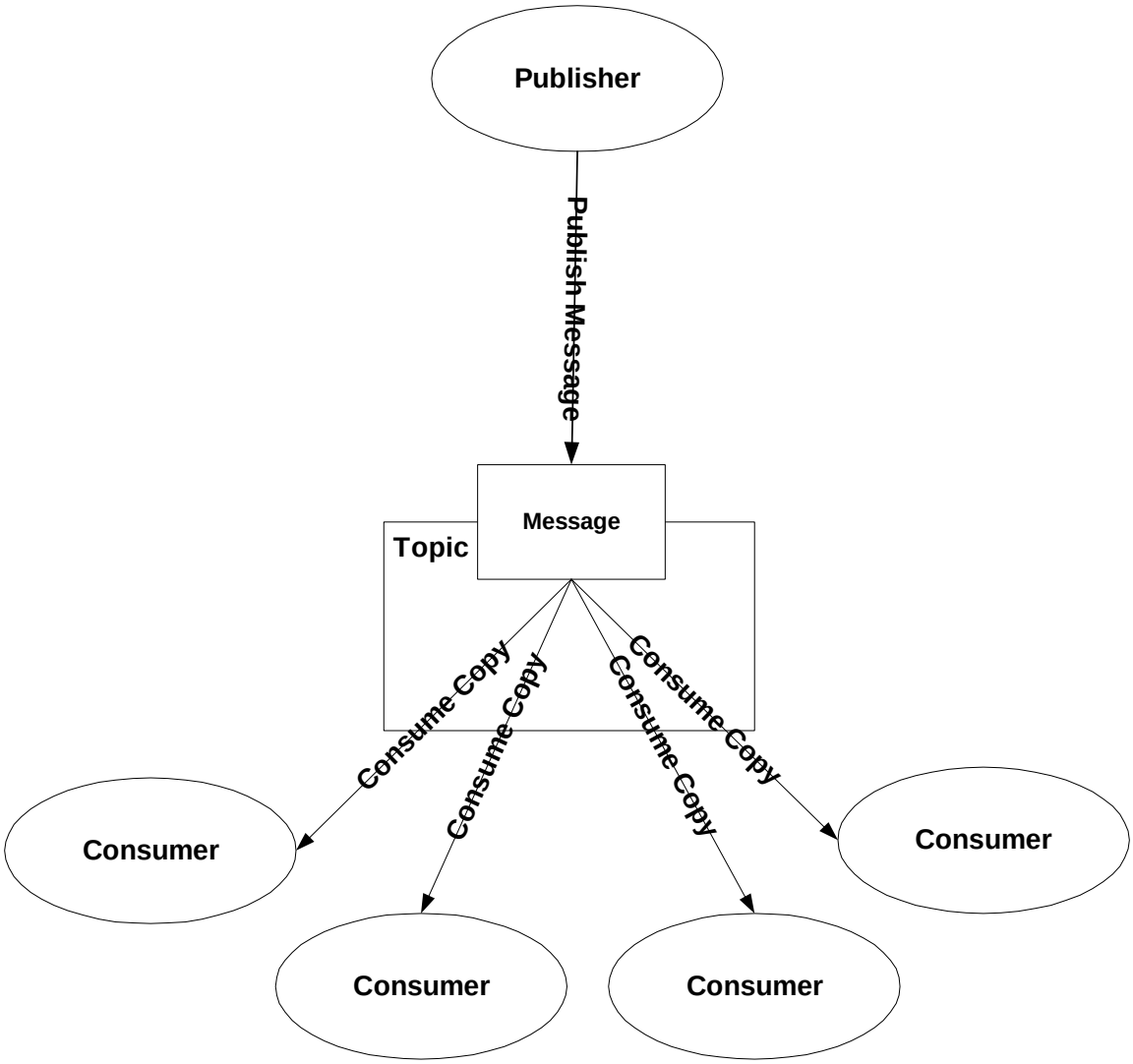
Point-to-Point

In this model, by default, a message is intended for one recipient. This means one application puts a message on the queue and another application reads the message from that queue. Only that one application can process the message. Once that message is read by the application, the message is gone. The destination in a point-to-point model is usually called a queue.



Publish and Subscribe

The Publish and Subscribe model implements the hub architecture. Rather than writing messages to a queue for a single reader, senders publish to a Topic. A Topic is the name of the destination in the publish-and-subscribe model. As a sender, I subscribe to a Topic and I can publish messages. Furthermore, I can subscribe to a Topic in order to receive messages as well. The role of the receiver is called the consumer. The biggest difference is that once a message is published, all the consumers that have subscribed to the queue can receive there own copy of the message. Basically, the publisher can send messages to multiple recipients at one time.



Transactional Effects

Transactions play a big part in complicating the messaging world, especially in the Pseudo-synchronous update case. I will explain a bit about transactions in messaging systems, and talk about the added problems in update scenarios.

How Transactions work in messaging

A transaction, as most of us know, is a unit of work that has 1 or more tasks. In the case of more than one, if any one task fails, the whole transaction should fail. Many of us are familiar with this concept as it pertains to relational databases. Basically, there is some transaction that has a group of database commands that need to happen together. These commands are usually updates, but may contain a query. Queries become part of a transaction when someone needs to block another transaction from accessing the data during a transaction. For the most part though, transactions involve multiple updates.

When thinking about messaging, there are 2 fundamental operations: read a message and write a message. Messaging may need to group reads, writes, or a combination of read write within a single unit of work.

In the database world, transactions that fail are either rolled back or committed. For a roll back, if some update occurs in a transaction, and the transaction fails after the update, the update gets undone. Databases and transactional systems have sophisticated techniques for doing this. For committing, this means that an update happens, then the transaction successfully completes, and the update is marked as final.

Messaging also has the notion of rolling back. In a relational database, rolling back usually has an affect on writing data but not reading data (at most just releasing a lock). In messaging, however, rolling back has an affect on both writing and reading. Messaging also has the notion of commit. When a transaction is committed, the operation is final.

Ignoring grouping of transactions, let's look at the affect of transactions in relation to each messaging operation. There are 2 things we will look at: The state of a message written to a destination during a transaction and what happens when a message is rolled back.

We will first address writing a message. First, let us understand the state of the message. Relational databases usually have a level of control in this area. A database can control whether another transaction can see an update before the transaction has been committed. Messaging, usually, is not that sophisticated. In messaging, any message written to a destination during a transaction cannot be read by any other consumer outside the transaction. This means that a consumer can only see the message when the transaction that put it commits. In the database world, if I allow someone to read dirty transactions, only the dirty transaction is affected (unless he writes back to the database based on the dirty read). In messaging, once I read a message, the message is gone (except for durable publish and subscribe cases). Also, messaging applications usually don't try reading a message twice to see if it is valid. This also makes sense when we examine what happens to a message when a transaction that writes it is rolled back. When a message is rolled back, it is removed from the destination. Allowing a message to be read before it has been committed will cause problems if it is rolled back. The message is not meant to be read until it has been committed. This behavior is the same both in the 'publish and subscribe' model and the 'point to point' model.

Now let's address reading a message. If a message is read by a transaction which has not committed, what is the state of the message in relation to the destination it was read from while the transaction that read it is not committed? To understand this, let's see a simple database example. In the relational database world, queries aren't usually part of a transaction, but there are cases when I use data from a query to update some other source, either in the same database or somewhere else. While I do this query, I do not want anyone updating the data I queried while I am using the data to update some other source. You usually see transactions like this:

```
begin;
select productNumber, qty from Order where orderNumber = 1 for update;
update inventorytable set inventory=qty where productId=productNumber;
commit;
```

While I am executing the update command, no one can read the Order table row where the order number is 1. Understanding this will help you understand the state of the message when reading it during a transaction. Understanding what model you are using, 'point to point' or 'publish and subscribe', is also important for transactions and reading messages.

For point-to-point messaging, when a transaction reads a message, the message becomes available to the transaction, but does not get physically removed from the destination at that time; instead, the messaging system leaves the message on the destination but does not allow other transactions to see it. When the transaction commits, the message is completely removed from the destination; however, if the transaction rolls back, the message stays on the destination and is now allowed to be read by other transactions. This allows for a powerful feature called replay. Replay is the ability to retry the transaction after a rollback. The danger here is you can get caught in an infinite loop trying to replay the transaction. However, JMS systems can make use of the re-delivered count field on the message descriptor. Here, one can determine how many times the message has been re-delivered to a consumer and allow one to stop processing the message.

For publish and subscribe, if the transaction is rolled back, the message is discarded. Because the model is one-to-many, having the message rolled back on a Topic seems wasteful. However, in the publish-and-subscribe model, there exists the notion of a durable subscriber. If a subscriber is in durable mode, the only way to unsubscribe is by explicitly calling unsubscribe. This means that the subscriber can disconnect, and come back to get any messages that it may have missed. The messaging system is responsible for storing any messages for the durable subscriber it may have missed. So if a message is rolled back by a durable subscriber, then the durable subscriber goes back to a state in which it has never processed it. Because the messaging system stores missed messages for durable subscribers, they can replay the message.

Distributed Transaction Issues with Messaging

Because applications that use messages often have to use data in messages to update databases, or perhaps have to forward the message to some other messaging system, messaging applications often need to handle distributed transactions between messaging and non-messaging resources. For the purpose of this discussion, we will concentrate on having messaging and relational databases participate in the same transactions. Let's examine different scenarios for messaging.

- Application receives a message and uses the data to update a database
- User updates a database and sends a message in the same transaction
- Application updates a database, sends an update request, receives a reply, and updates the database based on the reply

We focus on these 3 scenarios to help illustrate the issues that are involved with messaging. There are definitely other scenarios and deviations of requirements, but we want to illustrate architectural problems with distributed transactions. There are other JMS issues such as Topic and Queue systems sharing transactions, but that is more of a programmatic issue than an architectural one.

Application receives a message and uses the data to update a database

This case is the easiest to describe. We have a listener waiting on a destination. When a message arrives, the application listener reads the message, parses the data, and uses the data to update a database. Once the database update happens, the transaction is committed.

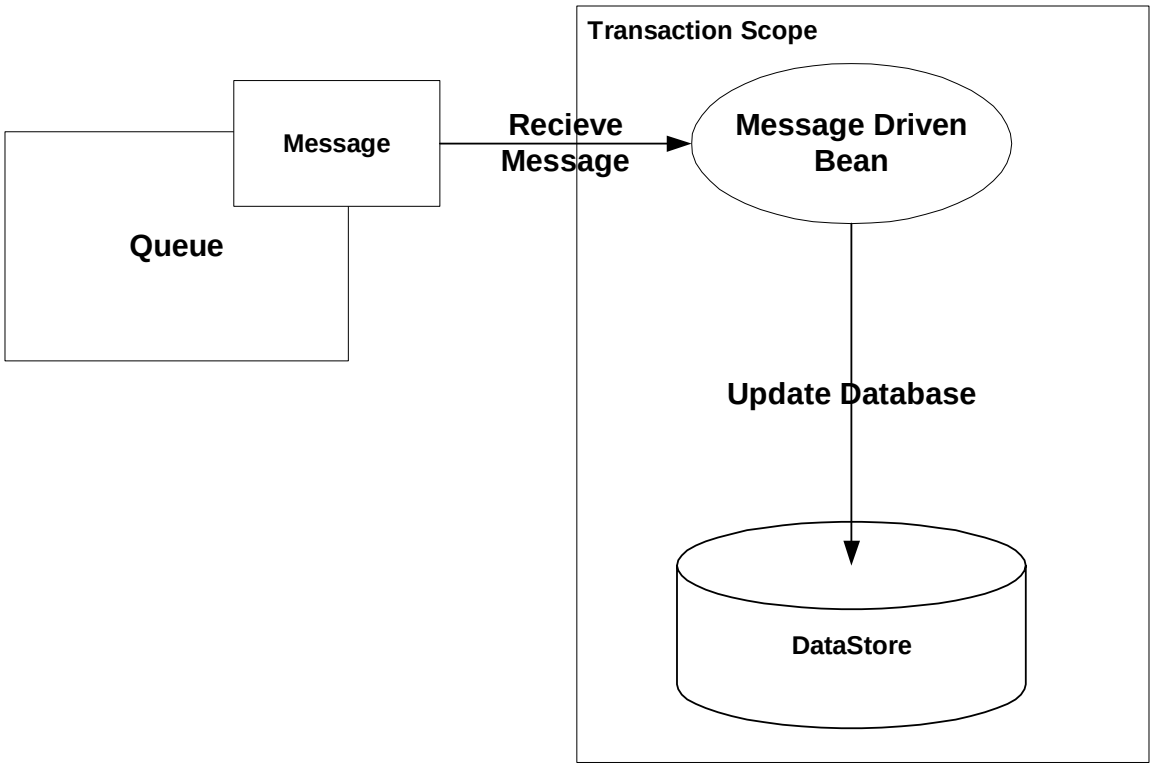
Let's use the point-to-point model to illustrate this example. Before J2EE this was a more complex scenario. Because a transaction requires a 'begin', you needed to hold a transaction context while you are waiting for a message. This is not ideal. One way around this was waiting on a browse (peeking at a message without removing it). When the browse returned with a message, the application would do a '**begin->read->update->commit.'** Then the message would go back to event-based browsing.

Thankfully, J2EE containers now user Message Driven Beans to handle this for us. Using Container Managed Transactions, it is now the responsibility of the J2EE container to demarcate the transactions. Basically, once our MDB receives a transactional message, we know that the transaction has begun. MDB's give you the ability to kick off asynchronous transactions inside the J2EE container.

What happens if the database update fails, let's say, because of a network failure between the database and the application server? The message is rolled back on to the Queue. The MDB can re-process the message and try again. In order to not get caught in an infinite loop, we need to make sure we can cap the number of times we process the same

message. The MDB or messaging system can be smart enough to check the re-delivered count of the message and perhaps send it to some back out queue if some max number of retries has been reached.

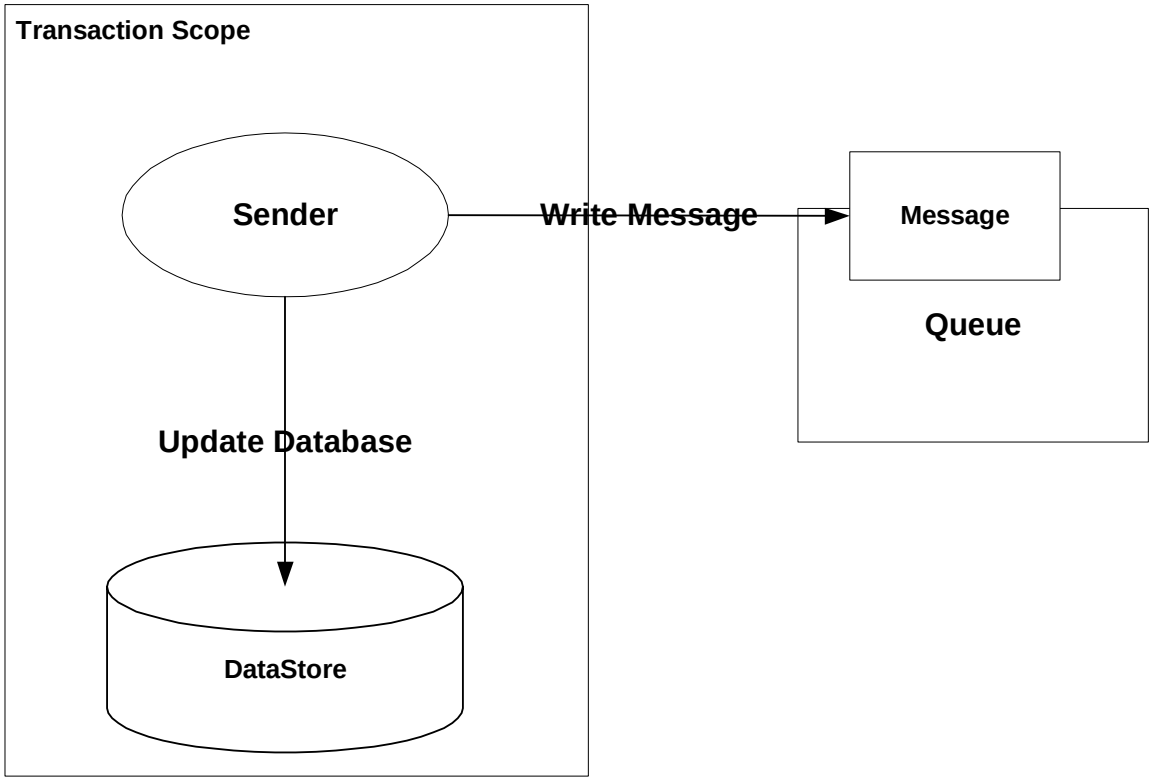
The diagram below illustrates the basic idea behind this type of transaction. (Note: the diagram is meant to illustrate transactional boundary and so application layering is not taken into account)



User updates a database and sends a message in the same transaction

The next type of distributed transaction that exists is a database update with a JMS send. There are several business cases for this model. Perhaps you need to mark a shopping cart order as in process before sending a JMS message to the billing department.

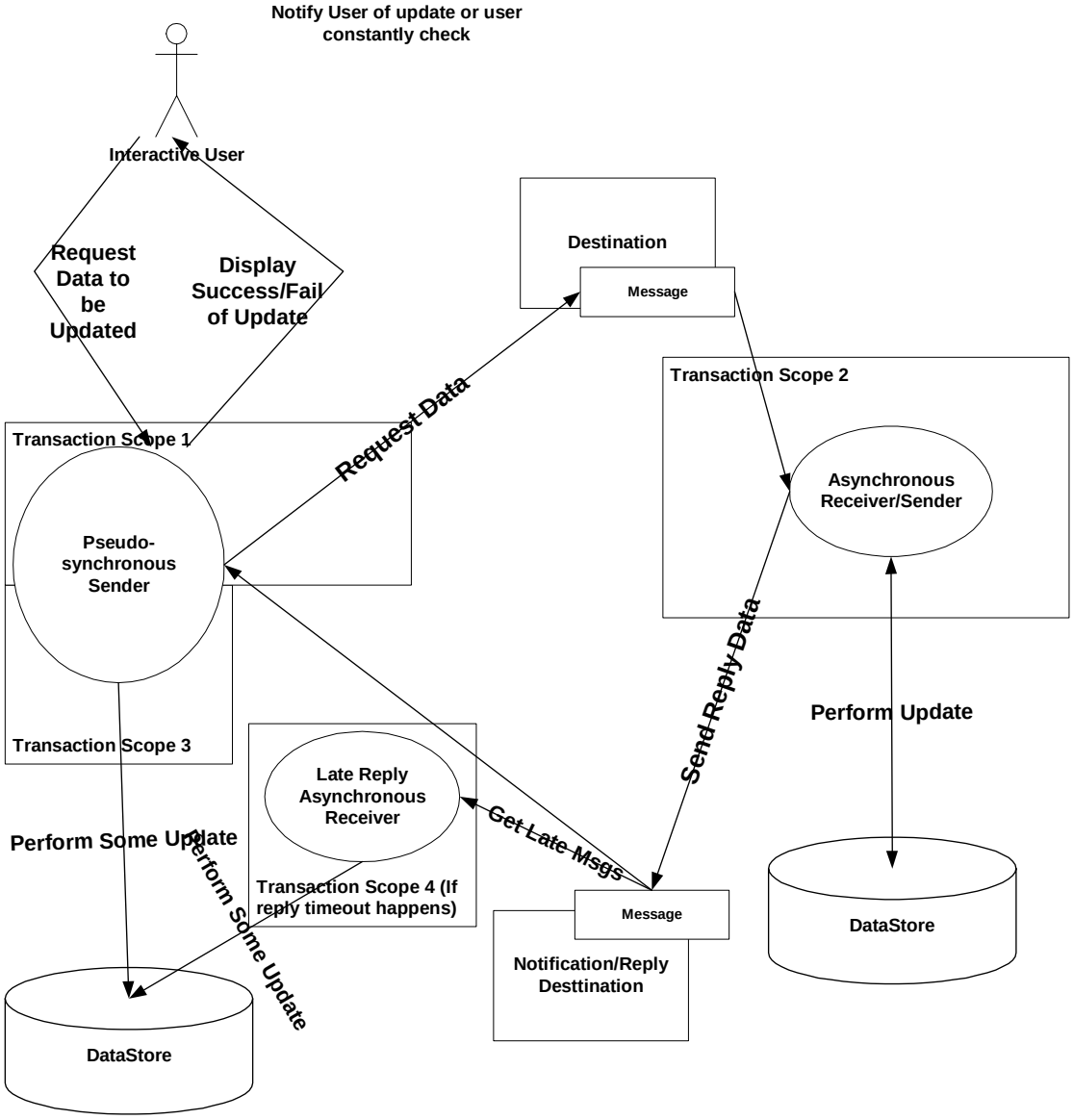
If the send fails, the database update gets undone. One thing to understand is, 'no one else can read the message until the message is committed'. This is important to understand in our next scenario. The figure below illustrates this transaction scenario.



Application updates a database, sends an update request, receives a reply, and updates the database based on the reply.

The next scenario is the transaction semantics that happen in a pseudo-synchronous update scenario with distributed transactions. You will quickly see the complexity of this scenario. Recall, no other transaction can see a message written to a queue within a transaction until the transaction commits. ***So if I am sending a request via JMS and expect a reply within the transaction, I am deadlocked***. Because I haven't committed the transaction, the write of the request is not final and therefore cannot be seen by the recipient. If the receiver cannot see it, he can not process the request to send the reply. In most cases, the business requirement really calls for a single transaction, but because of JMS, we need to separate this into 3 separate distributed transactions. Take into account the late reply possibility, and you add another transaction. This problem is further complicated inside EJB systems that use Container Managed Transactions. Because EJB transactions are marked at the method level, I need to split up calls into separate EJBs with perhaps a REQUIRES_NEW Attribute. Then you probably need a controller EJB to interact with each EJB.

If we take our Pseudo-synchronous update diagram above and apply the transactional scope, we get the following.



This case requires 3 to 4 transactions. There is overhead with setting up distributed transactions. Furthermore, there are state issues involved. If this were a synchronous call, and we were dealing with an Order, we can mark the Order status from Incomplete to Complete. Because we are using an asynchronous protocol, we need to introduce extra application state, for example, Incomplete, Order in progress, and Complete. Because we need to account for timeouts during the reply phase, we have to do this. Otherwise, the other end would have to be smart enough to handle duplicates which may not be possible in all scenarios.

Exploring Better Asynchronous Architectures

In this section I will illustrate better alternatives to messaging architectures. It will also illustrate where messaging should not be used if possible. We will also introduce some tooling alternatives that may help as well. Our goal is to achieve application decoupling as well as state decoupling, when appropriate.

JMS and Message Driven Beans

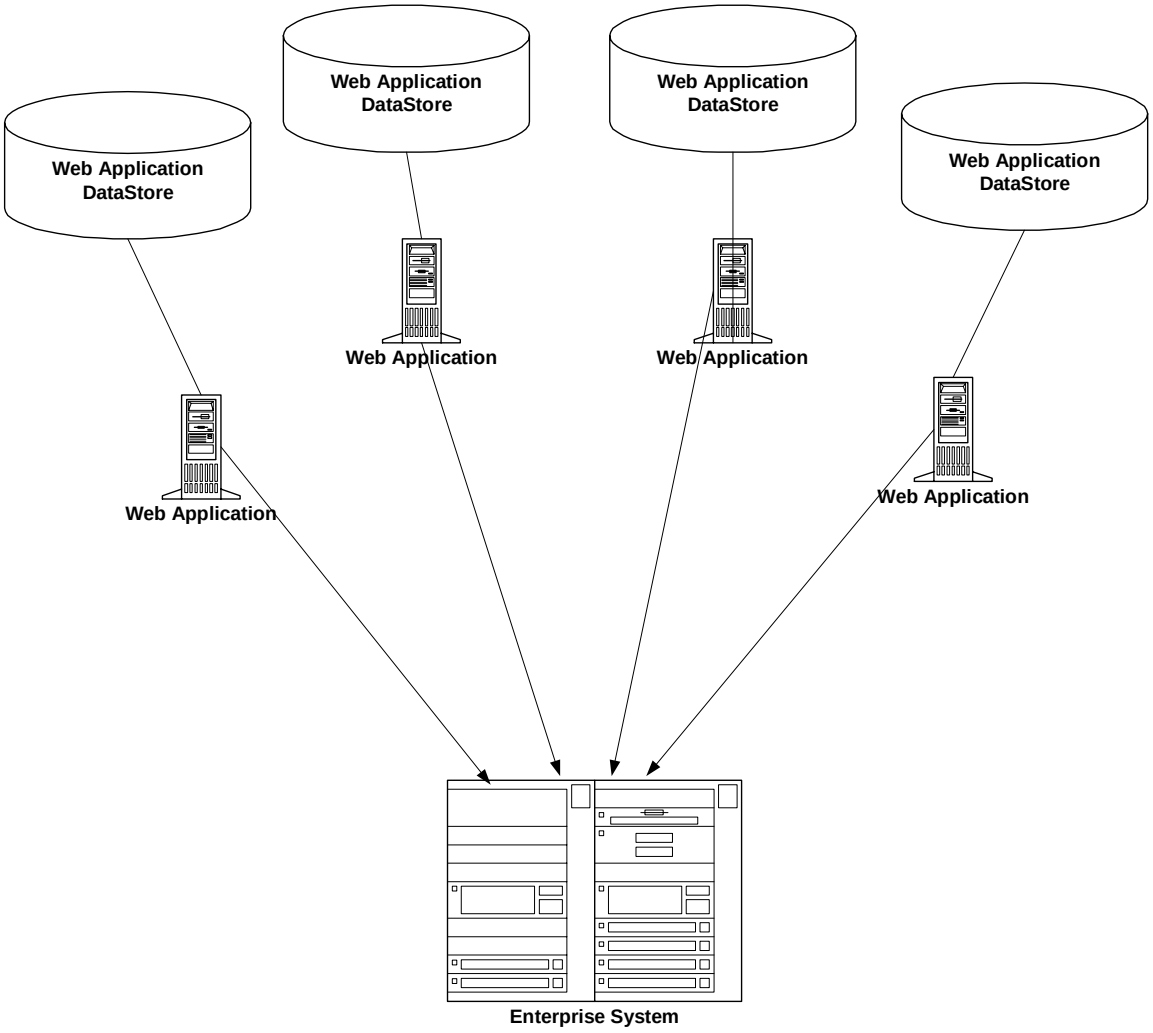
The JMS API gives messaging systems choices of implementation. Because the API supplies a Publish and Subscribe Model as well as a Point to Point Model, vendors now usually implement both standards. Being able to use both models will help you find the correct solution for the correct problem.

Inquiry Use Cases

In many cases, pure inquires between self service applications and enterprise systems will suffer from messaging. If a Web application needs to display data from an enterprise system to display to some end user, consider using a synchronous protocol. SOAP over HTTP is a great candidate (remember, I am speaking about the integration space, not queries within a single application domain). This keeps the system interfaces to each other data centric, thus achieving application decoupling, and gives us desired results. In this case, state level decoupling is not required. We want to know right away on the Web when the request will fail. As stated though, we need to deal with having synchronous calls over JMS for some reason or another.

Many companies have a central enterprise system (or several) depending on the size of the company. Many applications use a subset of the data or functions available. Messaging aside, although the Enterprise System is king, it is the application that drives its use. This can cripple the Enterprise System because the Web applications out number the enterprise and each Web application can add a significant amount of users that the Enterprise System may not be ready to handle.

It is becoming the job of the Enterprise System to service Web applications. The Enterprise System must now take on the traffic of each Web site.



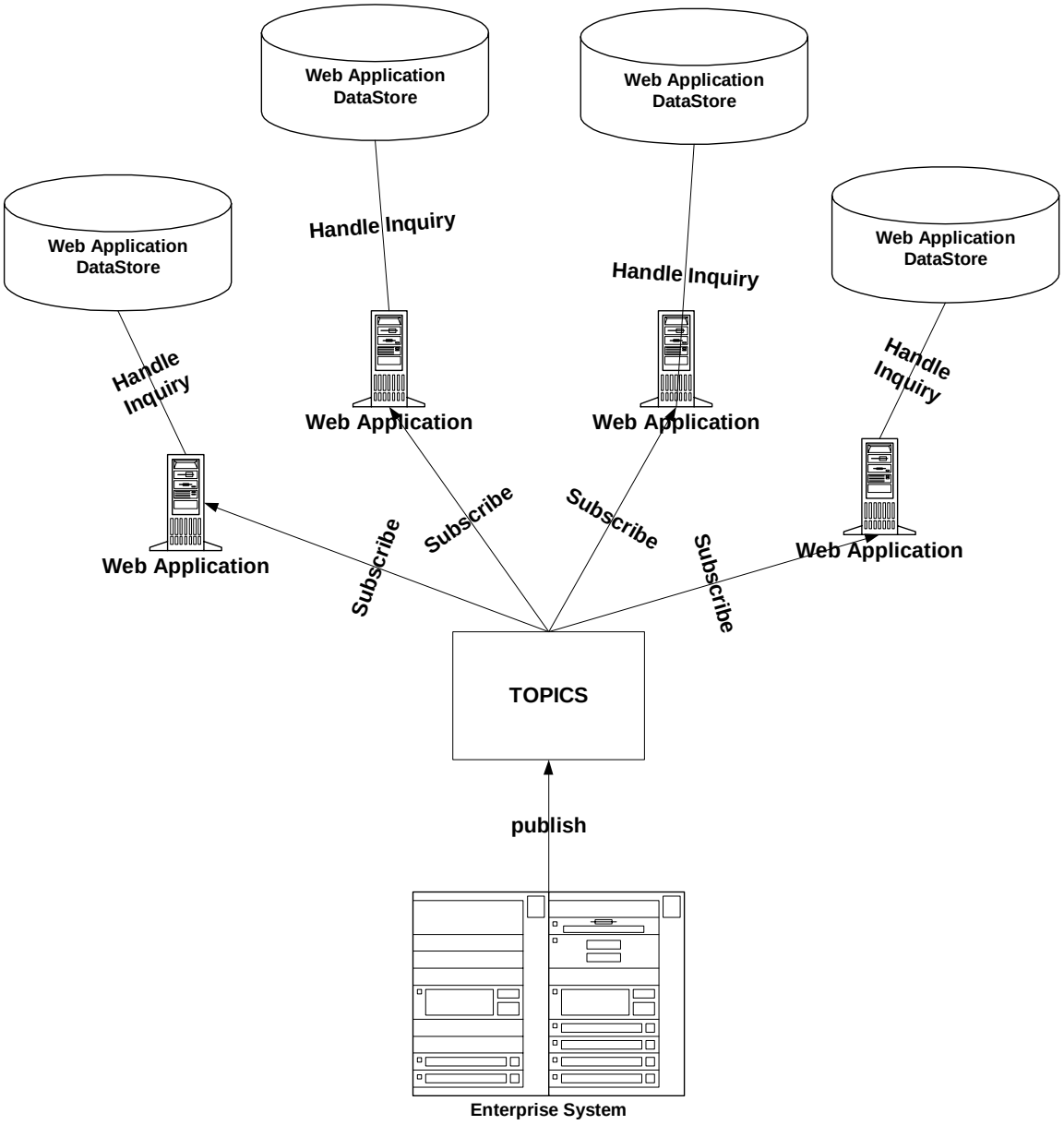
Consolidating Web sites is one alternative; this can be done using portal technologies. However, portals address 'Access Integration'; here, the requirements are transactional application integration. In order to come up with an alternative, we need to set goals.

- Keep de-coupling requirements
- Release the enterprise from the client hold
- Let each Web site manage its own volume

By taking some radical approaches, we can reverse the Enterprise Model and use Publish and Subscribe.

Web applications are usually interested in a sub set of data. Having these applications maintain a copy of this data is no longer a great deal to ask. Many applications do this anyway. Then, have all inquires into the Web site be driven from the local data store. How does the data get updated? Clients now subscribe to the enterprise and receive

updates they are interested in. Whenever an update to a particular section happens, the enterprise sends a message to a corresponding Topic. All clients who subscribe to the Topic now receive the data and can update their local store. All our clients now become asynchronous listeners. They can service inquires more quickly from their own store.



This is a radical approach, but will make use of your messaging investment.

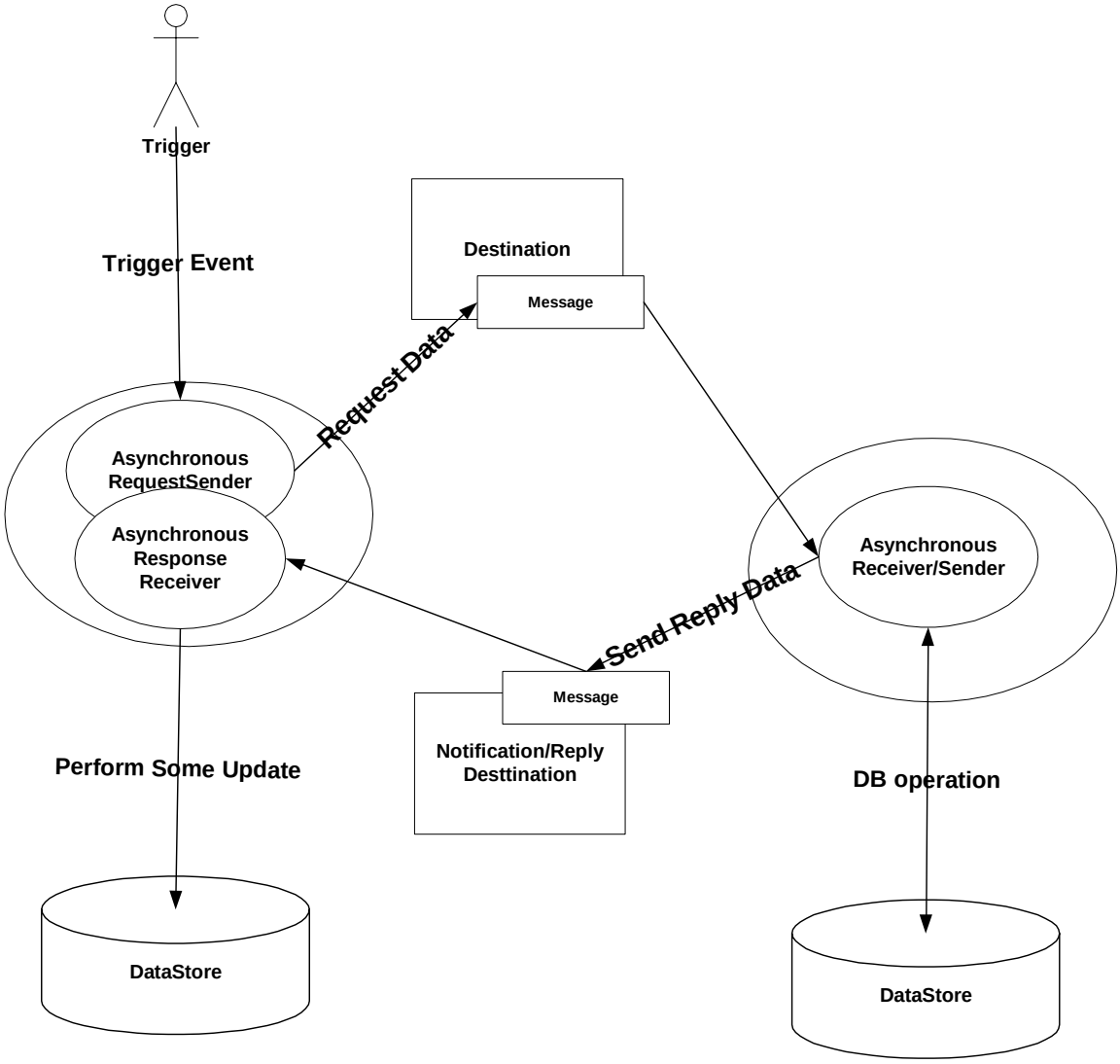
If radical changes are usually long term and take a while to plan out, the best recommendation is to consider a synchronous protocol with a message interface.

Update Use Cases

From the client point of view, updates are driven by the client. Whether to use publish and subscribe versus point to point does not help certain issues, especially, with transactional semantics. The client suffers the most here for all the reasons stated above. In the Pseudo-Synchronous Case, to use publish and subscribe also puts the overhead of using durable subscribers for the broker. Asynchronous updates do not suffer this problem and messaging is ideal and strongly recommended for this case.

The first alternative is to push the business requirement to asynchronous. Does the end user need to know during the same Web invocation? Can they wait 2 minutes for an email? Can the user check the status?

If so, then we can go to our asynchronous model for request and reply.

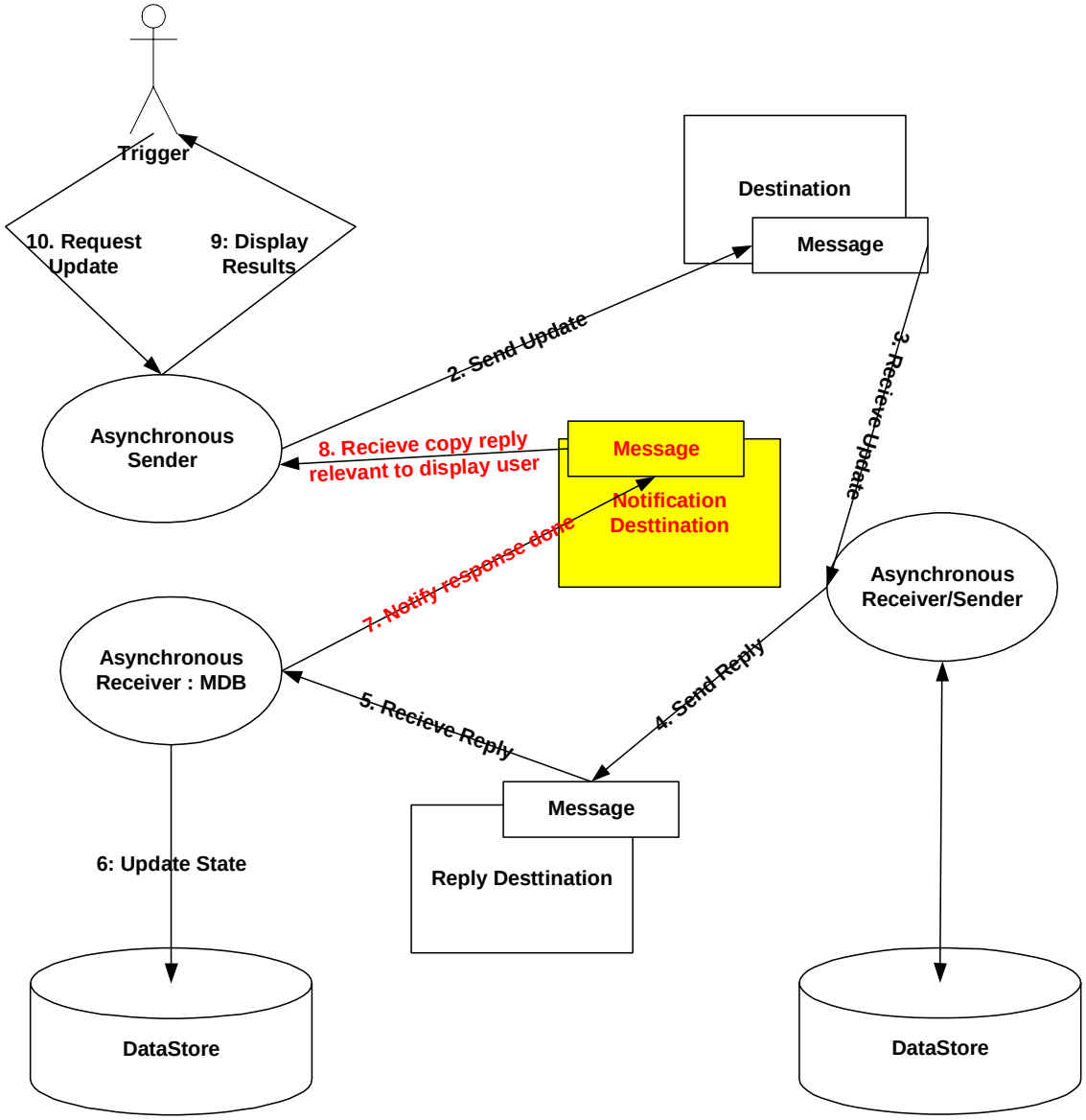


You can have a message driven bean implement the response.

If the requirement cannot change, what can we do? If you look at the asynchronous request – reply model above, we can easily extend it to meet our Pseudo-Synchronous need.

Although you still lose state decoupling, you can eliminate the need for late response special handling and keep the model asynchronous.

Basically, rather than having the request thread wait for the reply on the actual reply to queue, they can wait for it on an event queue. Then, the asynchronous listener can notify that the response has been processed and updated.



In this model, the transactional state is truly decoupled. The requesting thread does not need to manage transactions. If he times out, the notification can expire. The MDB can handle late and real time requests.

Conclusion

Messaging is a great technology that is often misused. Understanding the effects of its misuse is very important. It is essential that you first understand the transactional, integration, and the business requirements.

In this article, we exposed many problems with certain JMS topologies.

- The first thing to understand is what the decoupling requirements are. This also includes understanding what type of decoupling problem you are trying to solve: state or application decoupling.

- Next, you need to understand whether the use case is truly a synchronous one or an asynchronous one and if messaging is the correct solution for that case. You need to understand that the 'Pseudo-synchronous' case is really an anti-pattern and should be used with caution.

- Next you need understand the transactional requirements and consequences of certain topologies.

- Finally, you need to construct a solution that best utilizes your messaging investment by making use of the true asynchronous power of messaging.

About the Author

Roland Barcia is a Consulting IT Specialist with IBM's Software Services for WebSphere organization in the software division, working as a WebSphere Studio Application Developer and WebSphere Application Server Consultant. He has helped several clients implement WebSphere, J2EE, and Web Services solutions as an architect and a mentor. Roland has published articles in WebSphere Technical Journal and WebSphere Developer Domain and has worked in the Application Integration space for over 6 years. He is also the lead author of the up and coming book titled, IBM WebSphere: Enterprise Deployment and Administration. Roland has Bachelor Degrees of Science in Mathematics and Computer Science from Saint Peter's College and is currently pursuing a Masters of Science in Computer Science at the New Jersey Institute of Technology.