

THE UNIVERSITY OF SYDNEY

COMP5348 Group Project

SEMESTER 1, 2018

Karthikeyan Balasubramanian - 460123835

Sergio Mesina - 450158852

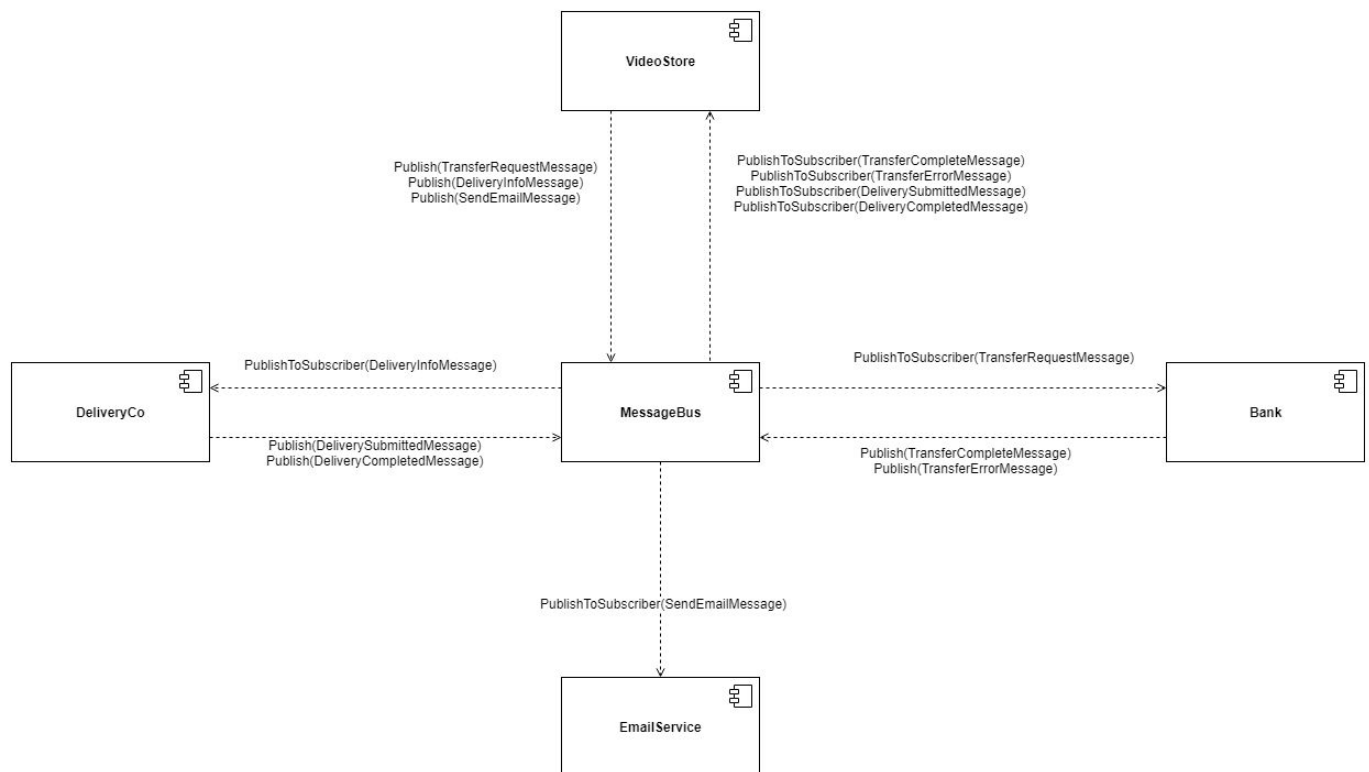
Alex Park - 440426684

1. Solution Description

This section introduces the solution, its various components and gives details into its workings.

1.1 Solution Architecture & Design

The solution makes use of a Broker pattern. This pattern essentially combines the messaging/queueing functionality of a Pub-Sub architecture along with a Hub-Spoke design pattern. The solution has 4 business components and 1 Message Bus. The business components are VideoStore, DeliveryCo, Bank and EmailService.



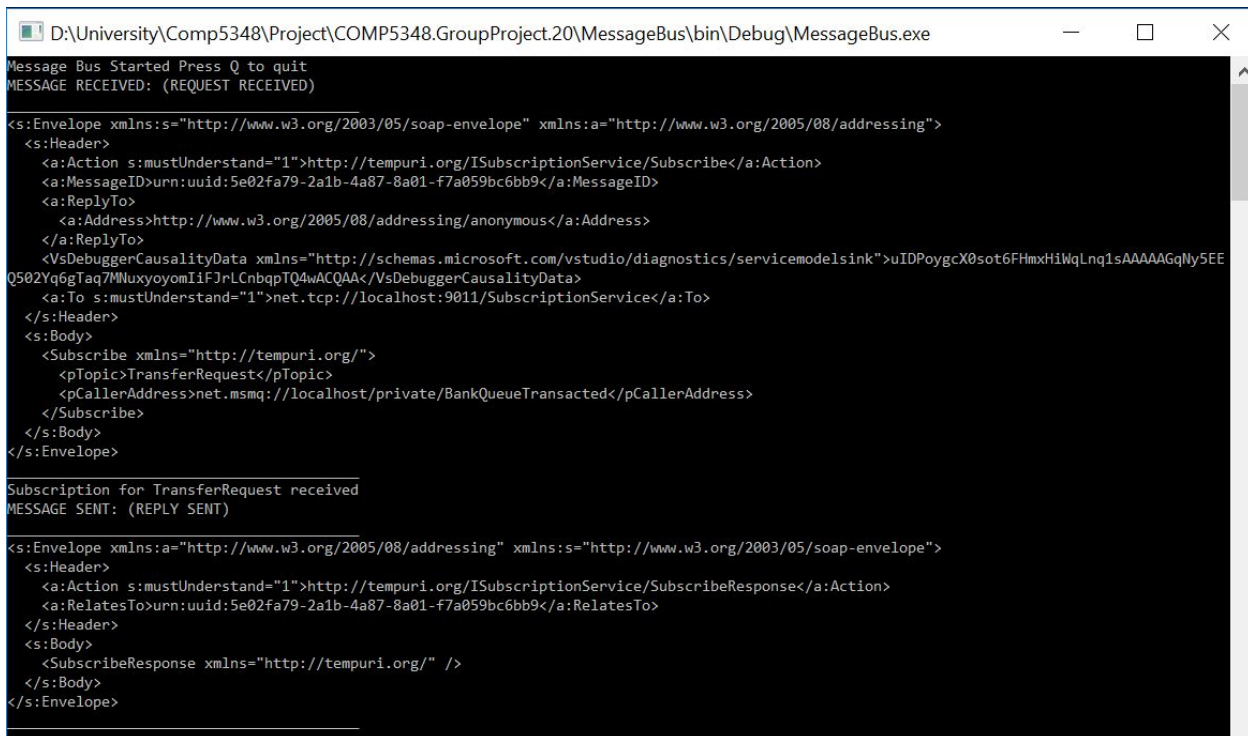
All communication is handled through Pub-Sub. The components in the system subscribe to their respective topics in a MessageBus and can publish messages against these topics. Whenever a message is published to a topic, any component that is subscribed to that topic receives that message and can then perform any business logic as needed.

The above described architecture lends itself well to the use of a Hub-Spoke design pattern to integrate the various components of the system. None of the components (Spokes) are directly aware of each other, instead all communication happens via the central message bus (Hub).

1.2 Implementation Details & Message Exchange Workflow

The solution is implemented using Microsoft Message Queueing (MSMQ). It is a framework that allows applications to communicate with each other even across networks and even with systems that may be temporarily offline. This is done by allowing the applications to “publish” messages to queues against a certain topic. Any other application that is “subscribed” to the topic is allowed to read messages from the queues. The messages themselves are written in Extensible Markup Language (XML) and are sent using the Simple Object Access Protocol (SOAP).

To start the program, first start a new instance of Messagebus. After that, all of the other components can be started. This is because each component subscribes to a topic in the MessageBus during their respective start up functions.



```
D:\University\Comp5348\Project\COMP5348.GroupProject.20\MessageBus\bin\Debug\MessageBus.exe
Message Bus Started Press Q to quit
MESSAGE RECEIVED: (REQUEST RECEIVED)

<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope" xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <a:Action s:mustUnderstand="1">http://tempuri.org/ISubscriptionService/Subscribe</a:Action>
    <a:MessageID urn:uuid:5e02fa79-2a1b-4a87-8a01-f7a059bc6bb9</a:MessageID>
    <a:ReplyTo>
      <a:Address>http://www.w3.org/2005/08/addressing/anonymous</a:Address>
    </a:ReplyTo>
    <VsDebuggerCausalityData xmlns="http://schemas.microsoft.com/vstudio/diagnostics/servicemodelsink">uIDPoygcX0sot6FHmxH1WqLnq1sAAAAAGqllly5EE
Q502Yq6gTaq7MNUxyoyomIiF3rLCnbqITQ4wACQAA</VsDebuggerCausalityData>
    <a:To s:mustUnderstand="1">net.tcp://localhost:9011/SubscriptionService</a:To>
  </s:Header>
  <s:Body>
    <Subscribe xmlns="http://tempuri.org/">
      <pTopic>TransferRequest</pTopic>
      <pCallerAddress>net.msmq://localhost/private/BankQueueTransacted</pCallerAddress>
    </Subscribe>
  </s:Body>
</s:Envelope>

Subscription for TransferRequest received
MESSAGE SENT: (REPLY SENT)

<s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing" xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Header>
    <a:Action s:mustUnderstand="1">http://tempuri.org/ISubscriptionService/SubscribeResponse</a:Action>
    <a:RelatesTo urn:uuid:5e02fa79-2a1b-4a87-8a01-f7a059bc6bb9</a:RelatesTo>
  </s:Header>
  <s:Body>
    <SubscribeResponse xmlns="http://tempuri.org/" />
  </s:Body>
</s:Envelope>
```

The above image shows how the Bank component subscribes to the topic “TransferRequest” upon startup. This means that anytime any applications publishes a TransferRequest message to the queue, that message will be read by the Bank application from the queue. Similarly all other components (VideoStore, DeliveryCo and EmailService) subscribe to their respective topics upon startup.

Component	Publishes	Subscribes
VideoStore	<ul style="list-style-type: none">TransferRequest	<ul style="list-style-type: none">TransferComplete

	<ul style="list-style-type: none"> • DeliveryInfo • SendEmail 	<ul style="list-style-type: none"> • TransferError • DeliveryCompleted • DeliverySubmitted
Bank	<ul style="list-style-type: none"> • TransferComplete • TransferError 	<ul style="list-style-type: none"> • TransferRequest
DeliveryCo	<ul style="list-style-type: none"> • DeliveryCompleted • DeliverySubmitted 	<ul style="list-style-type: none"> • DeliveryInfo
EmailService	N/A	<ul style="list-style-type: none"> • SendEmail

When a user submits an order through the VideoStore checkout functionality the Message exchange begins.

VideoStore's OrderProvider publishes a message to the queue for a transfer of funds against the TransferRequest topic. As the Bank is subscribed to this topic it picks up the aforementioned message from the queue and begins to attempt a transfer of funds. If the funds transfer is successful the Bank publishes a TransferComplete message or else it publishes a TransferError message. The VideoStore, as it is subscribed to both topics, reads these messages.

In the case of the TransferError the workflow stops and purchase cannot be completed without a transfer of funds. If the TransferComplete is received then the VideoStore proceeds with the workflow. The VideoStore then requests a delivery of the product purchased by publishing a DeliveryInfo message. This is read by the DeliveryCo which then organizes a delivery. It publishes messages when the delivery order is placed (DeliverySubmitted) and when the delivery is completed (DeliveryCompleted).

When the VideoStore is notified of either of the delivery milestones it sends out an email by publishing a SendEmail message. This is read by the EmailService when then performs the action of sending the appropriate email.

1.3 Common Failure Scenarios

There are a number of possible common failures as listed below. This section lists what solutions are currently available to solve the failures.

1.3.1 Message Bus Started Incorrectly

Description	The message bus is either never started or started out of order.
Type	Hard Fault

Solution	<p>The message bus needs to be started first in order for the components to subscribe to any topic. If the message bus is never started or started out of order (after some components) then those components will never have subscribed to any topic.</p> <p>Start the message bus at the very beginning.</p>
----------	--

1.3.2 Message Bus Crash

Description	The message bus crashes in the middle of the application runtime.
Type	Intermittent "Soft" Fault
Solution	<p>All topics have been subscribed to and all queues have been established. Thus messages are still being sent to their queues. Thus simply restart the message bus.</p> <p>Can be further improved by the future improvement #FI1</p>

1.3.3 Video Store Crash

Description	The video store process crashes in the middle of the application runtime.
Type	Failstop
Solution	<p>The Video Store is the process that contains all the business logic for the website being run. Without it no functionality is possible. Need to restart the Video Store.</p> <p>Can be further improved by the future improvement #FI1, #FI2</p>

1.3.4 Bank Crash

Description	The bank process crashes in the middle of the application runtime.
Type	Intermittent "Soft" Failure
Solution	<p>The Bank is required for any transfer of funds to take place. Without it no order can be placed. All topics have been subscribed to and all queues have been established. Thus messages are still being sent to their queues. Thus simply need to restart the Bank.</p> <p>Can be further improved by the future improvement #FI1, #FI2</p>

1.3.5 Email / DeliveryCo Crash

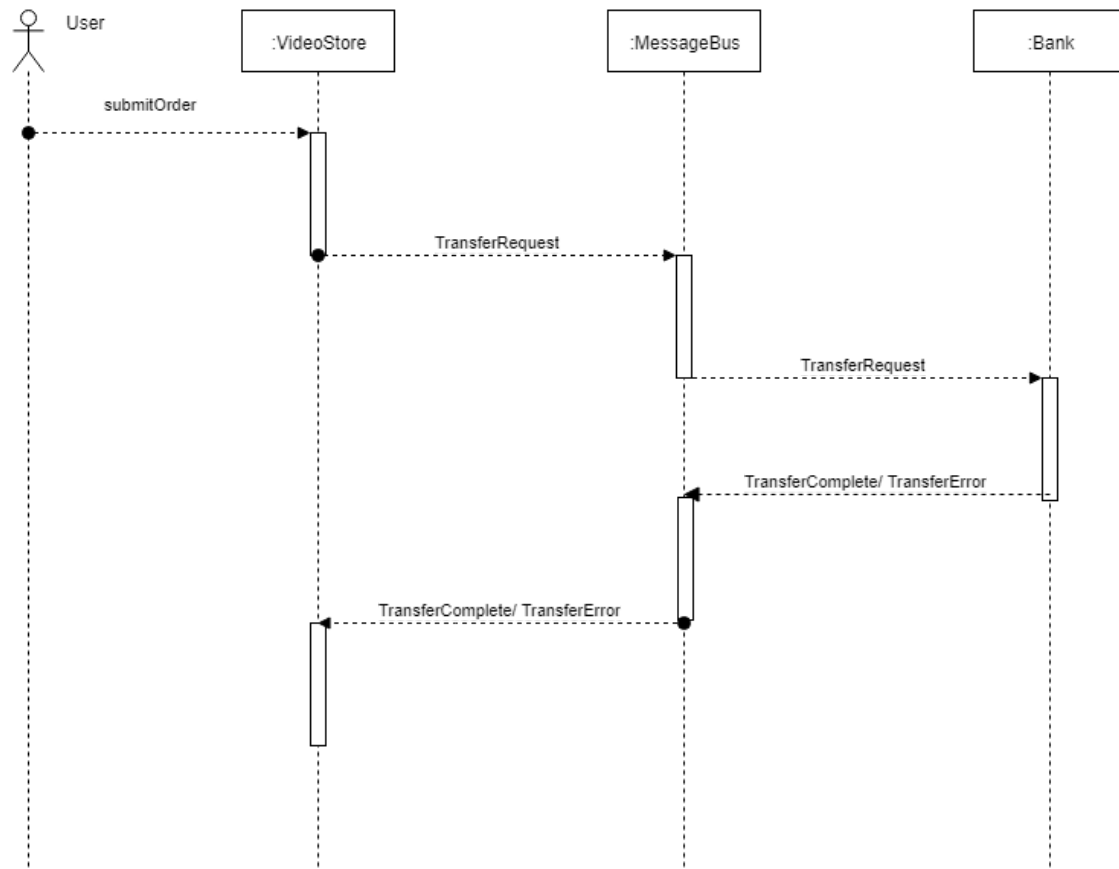
Description	The Email or DeliveryCo process crashes in the middle of the application runtime.
Type	Timing Failure.
Solution	<p>By this point the funds transfer has been completed and the order has been placed. However the delivery request or the email notification could not be sent due to the crash of either component. Simply restart the email / delivery process and the functionality can continue without error.</p> <p>Can be further improved by the future improvement #FI1, #FI3</p>

1.3.6 Database Crash

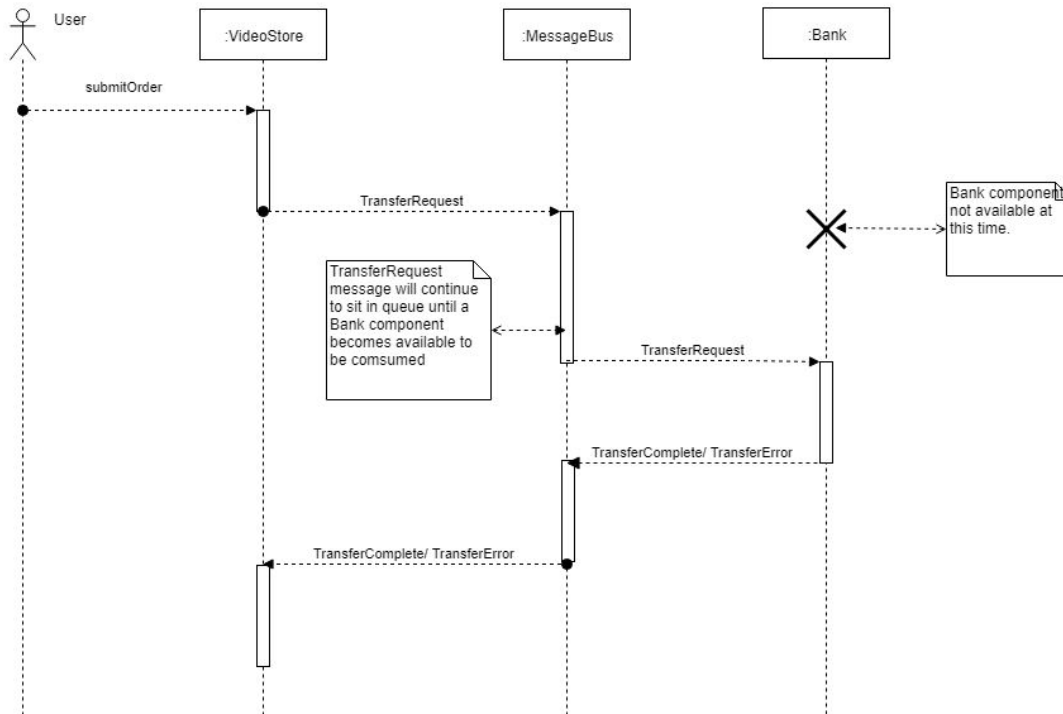
Description	The DB crashes in the middle of the application runtime.
Type	Failstop
Solution	<p>Without the DB no information can be accessed or saved permanently. This affects everything from Bank transfers to Order placements to even Signup/Login. The DB must be restarted.</p> <p>Can be further improved by the future improvement #FI1, #FI2</p>

2. Availability and Reliability Quality Attributes

Availability and reliability quality attributes are improved by making the workflow asynchronous. This means that not all of the components have to be running at the same time in order for the workflow to be completed. For example, when the VideoStore sends a TransferRequest message to the Bank component, only the Bank Component is required to be available in order for the workflow to proceed for this section. Furthermore, the solution utilises a message queue mechanism. As long as messages are not dropped or the queue is destroyed, they will continue to exist before they are consumed. This means that even if the Bank component is unavailable when the message was created, the workflow will continue once the Bank comes online and is able to consume the message.



When the Bank component is available, the workflow will proceed as expected. Once VideoStore receives the reply from Bank, it will continue on in the workflow. A similar diagram is applicable for any communication between components.



When the Bank component is unavailable, the workflow will not be able to proceed but the message will sit in the queue until the Bank component becomes available to consume the message. A similar diagram is applicable for any communication between components.

Ultimately, making the workflow asynchronous improves Availability by eliminating single points of failure. In other words, the system is still available and functioning even if not all of the components are not available.

The solution also improves on Reliability through the use of message queues. As a result, unavailable systems will not cause the workflow to terminate. This is because the messages and requests will continue to sit on the queue until the destination component becomes available to consume the messages. As long as the messages are consumed, the system will continue to operate in the expected way over time.

3. Quality Attribute Design Tradeoffs

Quality Attribute	Issues
Performance	In the previous design, messages are sent straight to the destination component. However, the new design now requires each message to be sent through a message bus. Performance is lowered due to the overhead introduced by the asynchronous design. This is because messages are now required to be

	transformed into a common model to be published and also when messages are received by each component.
Scalability	As each component are no longer tightly coupled, the system is now more scalable. This is because each component now communicates through a common interface (the message bus). As a result, each component can be easily scaled horizontally or vertically as long as it conforms to the common interface.
Modifiability	Similar to Scalability. Changes to components do not affect other components as long as they conform to the common interface.
Security	<p>In the previous design, each component represents a single point of failure. I.e. if a single component becomes unavailable, the workflow stops. Furthermore, if a component is hacked, only the messages which are transmitted and received will be intercepted.</p> <p>In the new design, the system will still function even when not all of the components are available. However, the Message Bus can be a vulnerable point as all of the messages pass through it.</p>
Portability	The system may be less portable as it uses MSMQ which is heavily used in Windows Platform-based applications. Therefore, the current solution may not be suitable for Linux-based systems. Alternatives could be RabbitMQ or JustSaying.

In the real world, each of these quality attributes are important. For example, each component could represent actual businesses and organisations. I.e. Customers would have different Bank institutions or the Video Store would outsource deliveries to delivery companies such as Australia Post. Not all of these business may use Windows or Azure services and thus cannot implement MSMQ. As a result, Portability is one of the quality attributes affected the most.

4. Future Improvements

To further improve the Quality Attributes here are the improvements we would have made in the future. These improvements are inspired by real world applications and how they handle errors.

ID	Name	Description
#F11	Replication	To improve availability services can be replicated. This improves availability since if one of the duplicates goes down, the other can immediately fill the void while waiting for the original to be restarted.
#F12	Unblocking the User	If a non-critical component such as Email crashes there is no reason to cancel the entire order or block the user while waiting for

		<p>a service restart. The user can be allowed to carry on with their use of the website and the email can simply be delayed.</p> <p>This is common practice in real world e-commerce enterprises such as Amazon where email delivery is often not instantaneous.</p>
#FI3	Graceful Exit	<p>If a mission critical component such as the Video Store or the Database crashes then it causes a Failstop error. In this case showing the user a user-friendly error message is highly recommended.</p> <p>This can drastically increase user-retention as it improves their experience. It is thus common in many real world websites.</p>