Universiteit van Amsterdam

Research Project 2

Load balancing in ESB based service platform

Author:
Nick Barendregt
nick.barendregt@os3.nl

Supervisor:
Yuri Demchenko
y.demchenko@uva.nl

September 4, 2012

Abstract

The load balancing is an important problem in the enterprise applications built using the cloud Platform as a Service (PaaS). Most of the currently used web application platforms already have mechanisms that handles the load balancing at the level of OS and programming platform. This project will look at load balancing techniques at higher level at applications and service composition platforms such as the Enterprise Service Bus (ESB), that should allow upper layer applications to control or set priorities for applications execution on the platform. This research is a part of a bigger research on Composable Services Architecture in the GÉANT3 project what provides a motivation and criteria for the final results.

The goal of this research project is to research possible solutions for optimal load balancing and optimal resources distribution between services inside a (Fuse)ESB based platform. The project investigates a dynamic way to inform set of interconnected brokers that should allow load balancing in multi-domain environment.

The project investigates the possible configuration options within the FuseESB platform that should allow load balancing. The proposed solution can provide a basis for dynamic services configuration or additional services deployment to handle required load. This research is based on already done research in this field by the University of Amsterdam (UvA) in combination with other research groups. The project demonstrated that there is no straightforward solution in ESB to provide control over execution application to the applications themselves.

Balancing load inside a FuseESB based platform is possible, there are some factors that have to be taken in account to specify configuration options. The presented report explain what steps need to be taken to achieve to create the dynamically configured infrastructure that allows for balancing load. The report provides example of code and sample configuration files that demonstrate how the broker and individual services must be configured.

Contents

1 Introduction

Cloud computing defines three basic service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Cloud PaaS service model provides an environment for creating and deploying applications using one of popular development platforms such as current available on the market Windows Azure, Google App Engine, or SaleForce.coms Force.com. Customers use PaaS services to deploy applications on controlled, uniform execution environments available through the network. IaaS gives a way to bind hardware, operating systems, storage and network capacity over the Internet. The cloud based service delivery model allows the customer to acquire virtualized servers and associated services on demand.

1.1 Project objectives

This project will investigate/research the GEMBus (GÉANT Multi-domain Bus)[5] as one of prospective Cloud PaaS platform for creating composable services in the GÉANT[1] environment. The GEMBus is being developed as a part of it is built upon and extends the industry accepted Enterprise Service Bus (ESB) platform with automated services composition functionality and core services to support federated network access to distributed applications and resources[1]. With this implementation of ESB providing services like PaaS, deployment can be fully automated.

An enterprise service bus (ESB) is a software architecture model used for designing and implementing the interaction and communication between mutually interacting software applications in a Service Oriented Architecture. As a software architecture model for distributed computing it is a specialty variant of the more general client server software architecture model and promotes strictly asynchronous message oriented design for communication and interaction between applications. Its primary use is in Enterprise Application Integration of heterogeneous and complex landscapes like GÉANT. [2]

Composable services are useful in bigger enterprises where all kinds of services are wanted, not only on a daily bases, but maybe just for a couple of days, weeks or months. A baseline structure that allows for composable services, such as PaaS, SaaS or IaaS, is a pre in these enterprise environments.

1.2 Problem statement

As a proof of concept the GÉANT3 project created a simple testbed to demonstrate an example of the distributed multi-domain services composition and deployment. A simple demo test program has been created on the testbed that automatically starts up multiple Virtual Machines (VM's), which get configured during the startup, automatically geared with the needed modules and preform their tasks[6].

In this example all connections are setup in advance, so when deploying multiple times there is a possibility that the load on the broker(s) can get to high that it can not handle every message that comes in. The presented research

[2]The GÉANT network is the fast and reliable pan-European communications infrastructure serving Europes research and education community. `http://www.geant.net`

[2]`http://en.wikipedia.org/wiki/Enterprise_service_bus`

is focused on investigating the real time services operation and interaction in a distributed environment and proposing possible solutions for optimal load and available resources distribution between services. The intended solutions should provide a way to automatically balance the load and possibly with extra adjustments of a report mechanism which reports its load and based on that route traffic in an other way.

In the Internet traffic gets routed through defined paths, when a router gets overloaded with high volumes of traffic, the rest of the traffic should be redirected over an other route. The same holds within an ESB based platform, messages get routed over different brokers inside the network, most of the time over already defined paths. When a broker gets overloaded the rest of the messages should to take an alternative route. The goal of this research is to investigate the possibilities of load balancing within an ESB Based platform using FuseESB software. Because the intention is that services get deployed on a dynamic basis. When possible investigate investigate in a real time load balancing mechanism, to automatically balance load in real time services operations.

As mentioned above with Internet traffic options are already in place on a lower level to arrange all the rerouting of traffic. Also when we look at web services for example, a lot of different solutions are out there to control the load on all servers inside the network. All this is done on a lower level, as application there is not much to load balance. Creating a program which allows for feedback mechanisms and is implemented inside an ESB based platform can then greatly improve performance.

1.3 Research questions

The intention of the research is made clear in the problem statement, to form this for the project a main research question is formed with a small sub question.

- Is it possible in GEMBus/ESB to create such mechanisms that will allow controlling of load (load balancing) from applications or from external brokers in a multi domain environment.
- Does topology play a role in how load is being balanced.

1.4 Project layout

This project made use of the GEMBus/ESB testbed developed at UvA as an environment and a starting point to investigate the issues with the real time services operation and synchronization and the existing/proposed load balancing techniques.

The presented report is organized as follows. Section 3 provides detailed information about the possible configuration, where section 4 describes how these techniques can be applied to a Setup. Section 5 gives an overview how the test setup looked like and with what modifications.

2 GEMBus as a platform for Cloud based services composition

2.1 Composable Services Architecture (CSA)

Composable Services Architecture (CSA) provides a framework for cloud based services design, composition, deployment and operation. CSA allows for flexible services integration of existing component services. The CSA infrastructure provides functionalities related to Control and Management planes, allowing the integration of existing distributed applications and provisioning systems, thus simplifying their deployment across network and cloud infrastructures. CSA provides also a basis for provisioning distributed composite services on-demand by defining composable services lifecycle that starts from the service request and ends with the service decommissioning. CSA is based on the virtualisation of component services that in its own turn is based on the logical abstraction of the component services and their dynamic composition. Composition mechanisms are provided as CSA infrastructure services.

2.2 GEMBus

The GEMBus framework, being developed within the GÉANT project, aims to build a multi-domain service bus for the GÉANT community to provide a common platform for integration of the existing and new GÉANT services. With the GEMBus as a development and integration platform, new services and applications can be created by using existing services as building blocks. The foundation of the GEMBus framework includes the necessary functionality to create composite (composed) services and effectively use the widely accepted Service Oriented Architecture (SOA) to building autonomic and manageable services using the provided mechanisms for composition, adaptation, and integration of services

More detailed information about how everything works together with authorisation and more detailed explanation can be found in [1].

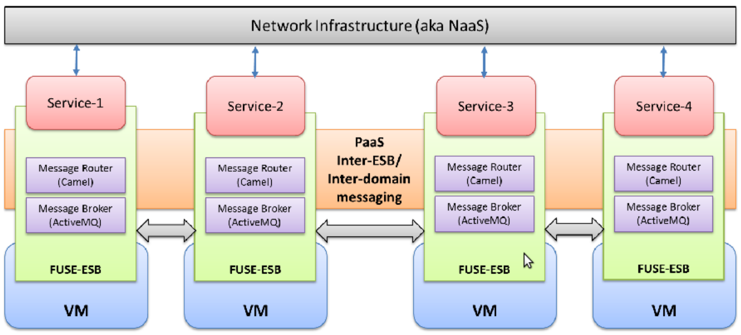

Network Infrastructure (aka NaaS)
Service-1
Service-2
Service-3
Service-4
PaaS Inter-ESB/ Inter-domain messaging
Message Router (Camel)
Message Broker (ActiveMQ)
FUSE-ESB
VM

Figure 1: UvA Testbed, using an ESB based platform

3 Overview Load balancing technologies and practices

In this chapter we will discuss different techniques that can help in building a network of brokers where in the end load can be balanced. The different sections explain the behavior of the named items, their default values and some possible changeable parameters. More and information and specific configuration parameters can be found in the documentation [2] [3] [4].

3.1 Broker network connectors

By default when making a connection, the type for that connection is in “OpenWire”. This also seems to be the fastest connection possible with a setup over a network, without depending on very specific scenarios such that the New I/O (NIO)[3] can be used to improve performance.

Other availible different connection protocols are Stomp, REST, XMPP and VM. Where the Stomp protocol is a simplified messaging protocol that is specially designed for implementing clients using scripting languages. The Representational State Transfer (REST) protocol is a simple HyperText Transfer Protocol (HTTP) based protocol which allows you to interact with the message broker using HyperText Markup Language (HTML) forms and Dynamic HTML (DHTML) scripts, default GET, POST, PUT and DELETE commands can be used. The Extensible Messaging and Presence Protocol (XMPP) can accept connections from for example an Instant Messaging (IM) client. The Virtual Machine (VM) protocol allows clients to connect to each other inside the same Java Virtual Machine (JVM) without the overhead of network communication.

The detailed information and how to implement all of these different protocols is described in detail in the Connectivity Guide [2], in this report we assuming that we will work over a network where the OpenWire protocol is used by default.

3.1.1 Duplex or non Duplex

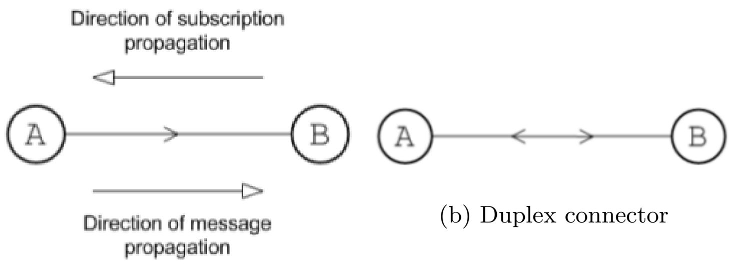
When creating a network connection between brokers the type of duplex can be of essential meaning. By default a network connection has duplex disabled. What means that if broker A creates a network connection to connect to broker B, messages can only propagate from A to B and subscription propagation can only flow in the opposite direction (from B to A). In short, it will only make sense that a producer connects to broker A and a consumer to broker B, else messages and subscriptions wont work over this network connection.

An example of how a non duplex connector looks like is shown in figure 2a and a duplex connector is showed in figure 2b, a configuration example is given in example 11 where the duplex attribute is set to true.

3.1.2 Network TTL

By default when creating a network connector the *networkTTL* attribute has a value of 1. Which means it can only deliver its messages to the next broker.

[3]`http://en.wikipedia.org/wiki/New_I/O`


Direction of subscription propagation
A
B
Direction of message propagation
A
B

(a) Non duplex connector

(b) Duplex connector

Figure 2: Network Connections
Source: Using Networks of Brokers [4]

When a bigger configuration of brokers is used in the network TTL attribute should have a value that satisfies the administrators the best. It has to be taken in mind that this value should be configured prior to deployment to a value that is large enough so that a message can route through the entire network. Configuration example 11 shows how the value can be set.

3.1.3 Conduit subscriptions

When a connection is established between 2 brokers messages can pass from broker A broker B. When a consumer connects to broker B and subscribes to specific topic (t), broker B will inform to broker A he wants messages from topic t because he has an active subscription from a consumer. In this way all messages on topic t delivered to broker A will be send towards the consumer.

When a second consumer connects to broker B, and also informs about his subscription to the same topic t, broker B will *not* pass this information towards broker A. Because there is already a consumer with an active subscription to topic t on broker B. Instead of informing broker A, broker B will receive 1 copy of the message send to topic t, and then copy it to both the consumers.

This behavior is called conduit subscription. Subscriptions to the same queue or topic are automatically consolidated into a single subscription. This example is explained in figure 3.

Conduit subscriptions prevents against the fact that messages will be duplicated and send twice. When conduit subscriptions would be disabled on the connection between broker A and B, both the consumers C1 and C2 will subscribe to the same topic t. These subscriptions on broker B will then be forwarded to broker A. When broker A receives a message on topic t, it will send 2 copies to broker B, where broker B sees 2 messages on topic t, it will send both messages to consumer C1 and both messages to consumer C2. While 1 message was send to topic t, both the consumers will receive 2 identical messages. This element can produce a lot of problem, see section 4.2 for more information about how avoid these problems.

3.1.4 Filtering

In a multi domain network setup it can be that a particular sub-domain is not interested in messages that are mend for an other sub-domain. Because

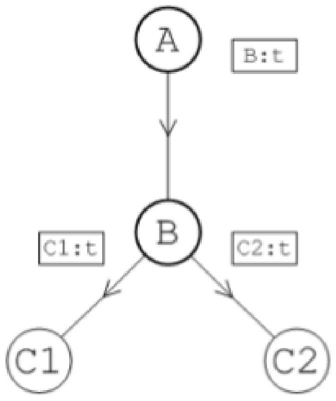


Figure 3: Conduit Subscriptions
Source: Using Networks of Brokers [4]

Wildcard	**Description**
.	Separates segments in a path name
∗	Matches any single segment in a path name
>	Matches any number of segments in a path name

Table 1: Destination name wildcards

the network is inter linked to each other to create one big network of brokers connections still have to exists in order to work properly. Filtering on certain network connectors can then be the solution to avoid extra load on the edge broker of the sub-domain.

A filter can be applied in multiple ways, where for instance destinations are included. This means that every queue or topic matching that rule is allowed to pass the broker connection, all the other topics and queues are not allowed to pass. When applying this configuration it is needed to know that this literally *disables* all other traffic over that network connection *except* from the included filtered destinations.

An other way of applying a filter is to exclude certain destinations. An example combination of including and excluding different destinations is given in example 13. Where broker A is interested in all information about any TRADE.STOCK and PRICE.STOCK as destination, but with excluding the TRADE.STOCK.NYSE.* and PRICE.STOCK.NYSE.* destinations broker A wont receive any information going on the New York Stock Exchange (NYSE).

As shown in the example there are some wildcards used in the filters, in table 1 all possible wildcards are explained.

3.2 Discovery

There are several different techniques which can be used exclusively or combined with each other for the successful discovery of brokers inside a network of brokers. A discovery protocol builds a connection to a message broker in 2 basic steps, where the first step is to gather information of all available broker endpoints in a network of brokers and the second step is to connect to one or more endpoints, according to some selection algorithm. Below is a list of different discovery protocols where i will elaborate on.

- Failover
- Discovery
- Fanout

3.2.1 Failover

The *failover protocol* facilitates quick recovery from network failures [3]. Where there are two ways to setup your failover connection, statically or dynamically. In a static failover the client is configured to use a failover URI, where inside that URI reside multiple message brokers where the client can connect to. If the client tries to establish a connection, he randomly chooses a URI from the list and attempts to connect to it. When this connection fails a seconds connection is tried from this list of URIs.

It is possible to give extra options with the connection URI, with these extra options more specific information about the connection can be defined. For example how many milliseconds to wait for the first reconnect attempt, reconnection timeout in milliseconds, maximal reconnection attempts and to randomize them yes or no. More information about the specifics for these and the rest of the options are given in the manual: Fault Tolerant Messaging [3].

With *dynamic failover* both the clients and the brokers must be configured in such a way that it allows for dynamically updated broker and client lists. On the client side the failover URI must contain at least one active broker connected which participates in a network of brokers. This failover URI can also contain multiple brokers, which can enhance the success rate of the connection. The brokers in this scenario also have to have a adjusted configuration to allow the discovery of them selfs and allow others to connect. On the broker on the connection must have an option which filters the allowed brokers on the discovery. This discovery is done on the brokerId flag, where the filer looks at, wild cards are supported so the wild card * can be applied to allow every broker on the broadcast network. The local network must be configured appropriately for the IP/multicast protocol to work.

In example 6 is a connection URI visible what can be used as failover.

3.2.2 Discovery

In the section above the discovery of brokers is explained, clients can also use this discovery protocol to connect to the brokers. Where the brokers have to configure a transport connector to allow the discovery protocol to work on and a network connector to start the agent on, clients can use this same protocol and connection URI to discover active brokers on the local broadcast network. Where brokers must configure both a transport and a network connector to allow for dynamic discovery, clients only have to do it in their main connection, without any prior knowledge of any broker or brokerId.

When a client receives a list of connected brokers, it can use this list for failover purposes or load balancing. By default when a client has multiple brokers to connect to, it will pick its broker randomly. This be disabled, so that only the first broker will be chooses, until it fails to connect, then the second will be chosen, and so on.

In example 7 is a connection URI visible what can be used to get a list all discovered brokers, from a statically configured broker.

3.2.3 Fanout

The fanout protocol, as the name already implies, is a protocol used by producers to spread their messages to multiple brokers which are not part of a network of brokers [4]. This is not fully relevant in a load balancing environment, since this is based on a network where the brokers are not connected to each other. When brokers are connected to each other and they both receive messages from the producers which used this fanout protocol, there is a change the a consumer ends up with 2 identical the same messages.

This can be useful in a setup where a producer is generating messages meant for more then one domain, which is not connected to an other domain receiving these messages. A good example to visualize this can be a news article, when a new article is created it will spread out with the fanout protocol to every interested network or consumer.

In example 9 is connection URI visible that can be used to spread messages on the default multicast discovery network.

4 Balancing Load

When talking about load balancing in a network of brokers, there are different methods or solutions used to achieve balancing effect. These mechanisms and techniques will be explained here and their usability will be discussed.

4.1 Propagation

Without any propagation in the network and no active consumers, no matter how big the network is, the messages send by a producer will always remain in the broker where the messages was send to. When a consumer connects to a broker and subscribes to the topic where the producer just send his message to, the messages will get forwarded to the consumer. An example of this behavior is visualized in figure 4. Where P is the producer, A,B,C,D and E are brokers in a network of brokers and C1 is a consumer. M represents the message producer (P) sends to the queue of broker A. Without any consumer subscriptions the message will remain in broker A. When consumer C1 connects to the network and subscribes (S) to the topic where producer (P) send its message (M) to, the message will propagate to the network until the consumer consumes it from broker E.

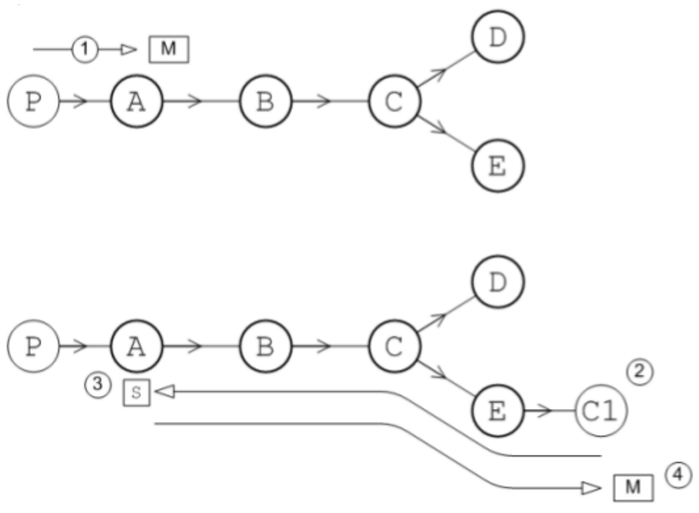


Figure 4: Dynamic propagation of queue messages
Source: Using Networks of Brokers [4]

With static propagation traffic can already be shaped before any active subscription is present on a broker, this can prevent spike network usage when after a long period a consumer connects to the network and all messages have to be send at once over all the brokers. Static propagation is implemented on the broker side, where in the network connector to an other broker `staticallyIncludedDestinations` are configured. Where such a destination can be a single topic or an entire queue, filtering and wildcards are allowed here. This has to be configured the same way on all the brokers in the path. To use the same topology as in figure 4 broker A,B,C,D and E have be configured on their network connectors to statically propagate messages from a certain queue. This is better visualized in figure 5. Where a producer sends 10 messages to Broker A, these messages are then forwarded to broker B, then broker C and there they will be divided over broker D and E. When now a consumer connects to broker E there is no need to send all messages through the chain of brokers because it is already at broker E.

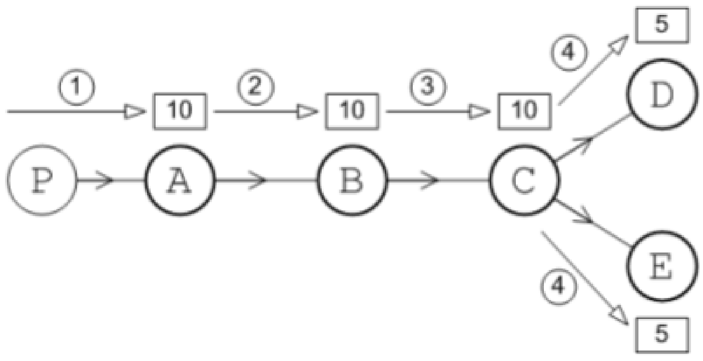


Figure 5: Static propagation f queue messages
Source: Using Networks of Brokers [4]

As seen in this example, when a consumer connects to broker E, he can only see 5 messages of the 10 send by the producers. This way there is a potential risk of stalled messages on brokers. When there are no consumers connecting for a long time to broker D messages will get stuck there. This configuration is in good use when there are known consumers on the brokers which have statically propagation enabled.

4.2 Balancing consumer load

With the default configuration of both the brokers and the consumers there is already a load balance feature present. That is when a broker has 10 messages in his queue and 2 consumers subscribe to that queue they both will get an even number of messages. If the both consumers are connected at the same time and can handle messages in the same speed. So with default configuration this can have its benefits, just trow more consumers to a broker if the other consumers can not handle all the messages from the producers. How ever when you change the topology, where you connect another broker to the broker where already a consumer was connected balancing the load over consumers does not always works the way it should. Because topology and settings are important things to take care of in setting up the right environment this also works here.

When a producer and a consumer are connected to a broker, the consumer will receive every message, assuming they use the same queue. When a second consumer connects to the broker, both the consumers will get an even number of messages, assuming that they can process each message with the same speed. So when the producer sends 10 messages, consumer 1 will receive 5 messages, and consumer 2 will also receive 5 messages.

In an other environment where we have a producer and a consumer connected to broker A, in line broker B also connects to broker A and broker B then again has 2 consumers (see figure 6). When the producer sends 12 messages, consumer

1 will receive 6 of them, and the other 6 will be send to broker B, where the messages get dived over consumer 2 and 3. This default behavior is explained in more detail in section 3.1.3, conduit subscriptions.

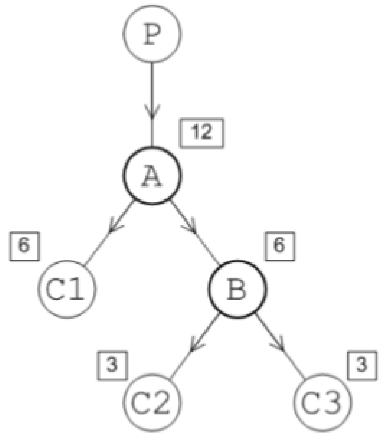


Figure 6: Message flow with conduit subscriptions enabled
Source: Using Networks of Brokers [4]

There is also an option which can be configured to disable conduit subscription over broker links. As explained in section 3.1.3 the conduit subscriptions protect against duplicate topic messages. So turning this option off requires some extra configuration to work. When we look at the same example given in figure 6 but now we disable conduit subscription on the broker link between A and B. Queue messages will get dived equally over all the consumers connected to the network. This is because every consumer lets broker A know that it is interested in messages from the example queue (see figure 7).

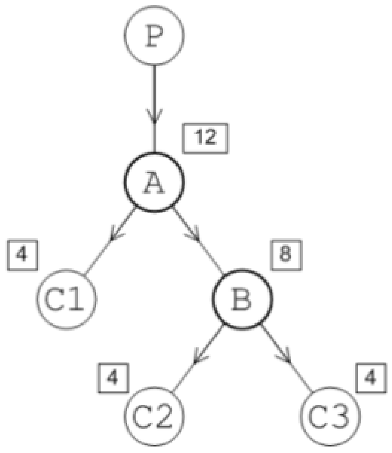


Figure 7: Message flow with conduit subscriptions disabled
Source: Using Networks of Brokers [4]

Because in this situation we are talking about queue messages only the conduit subscriptions can be disabled, when it also involves topic messages, things can go wrong and duplicated messages will be delivered. When a load balance needs to handle both queues and topics, you might want to disable conduit subscriptions for optimizing queue messages, but you want to enable conduit subscriptions for topics to avoid duplicated topic messages and to lower the usage of a network connector where a lot of consumer subscribe to the same topic.

Since it is not possible to enable and disable the conduit subscriptions on one single network connector, you can create multiple network connectors and apply filtering (see section 3.1.4) for queue or topic messages only on each of the network connector.

4.3 Managing producer load

Managing producer load has little to do with the configuration in the process. There are not many options which can help reducing the load producers and consumers can produce. When you want to manage the load in a better way for producers it is better to change the topology style in the network you have build or are building.

One way of doing this can be in layers, where the first layer only handles connections from the producers (can be many) and there are no connections from consumers coming in. The second layer of brokers is a smaller set of brokers where each broker from layer 2 connects to each broker from layer 1. The brokers on the second layer can then function as point where the consumers can connect to. This way the brokers from the first layer can be optimized to only handle incoming producer messages and the brokers on the second layer can be optimized to only serve consumers with messages. See figure 8 how this looks like. Notice that in this figure the network connectors are all non duplex. This means that messages will pass in the way the arrows point and subscriptions of topics and queues only propagate in the opposite direction.

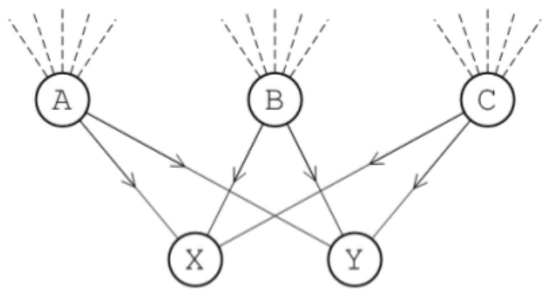


Figure 8: Concentrator topology
Source: Using Networks of Brokers [4]

4.4 Filtering

Load can also be balanced by applying filters on different network connectors. This can be useful in certain situations where for example an end point in the broker topology does not need to know information about all topics and queues, but only a specific topic or queue. A nice example for implementing such a filter is discussed in Using Networks of Brokers [4], where stock quotes are handled.

If we assume that we run a big exchange where multiple cities are connected to our system, messages to every city and every exchange will come by. When we for example are in a end broker in New York, maybe we are only interested in all the stock prices of New York. In the New York broker we can configure that we only allow messages in a certain destination. Destinations can be created in different ways as explained in section 3.1.4. This way the load on a broker can be reduced by only allowing certain destinations to pass through.

There is also a way of excluding different destination, while allowing the rest, when we are interested in everything on the stock marked, but we in New York are not interested in the stock exchanges that are going on the NASDAQ, we can exclude those from our broker connection. This example is already discussed in section 3.1.4 and in example 13.

4.5 Topology

Not only configuration is a big issue when you want to create a topology with brokers, producers and consumers where you want to balance load, the creation of the topology also plays a very big role.

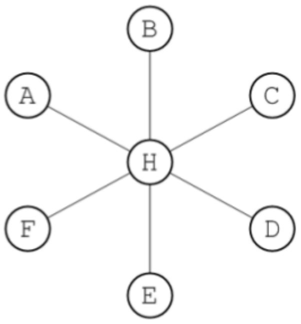
When you create a topology with only 1 broker, 10 producers and 50 consumers, its most likely that the broker will have to struggle a lot with performance when the producers are sending a lot of messages around. Adding multiple brokers can already reduce the load where the producers are responsible for. Dedicating one or multiple brokers for consumer connections, this already reduces the initial load for the one broker mentioned before.

There are different techniques to create a topology. When creating a topology it is good to take in mind the default behavior of all the network connectors and their settings.

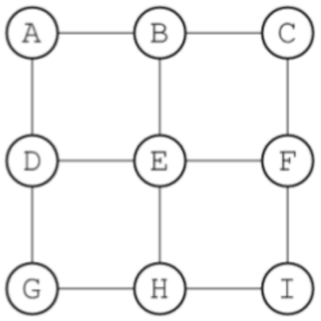
The hub and spoke topology in figure 9a is one of the easiest way of setting up a network of brokers. The central hub can be used to transfer all messages around, all connections with duplex enabled, so that every broker can reach every other broker, the networkTTL is always 2 in this entire network. It is easy to maintain, the hub only enables one or multiple transport connectors and every connecting broker initializes the connection to the hub. The downside of this approach is that if something happens to the central hub, overloading or has downtime, the entire network suffers from it.

The mesh topology from figure 9b is a network that naturally arises when you connect brokers geographically with each other and you want to connect to your neighbors. This topology needs more configuration then the hub and spoke, but is more robust against failing nodes. The networkTTl increases with every node which is attached to the outside of the topology, so when configuring this network, the networkTTL value has special attention if all brokers need to be reached. Creating 1 management platform where for example the networkTTL can be configured drastically lowers management overhead.

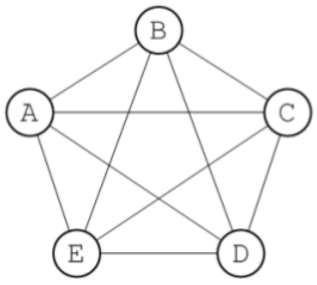
The complete graph topology from 9c is an example where messages always will take the shortest route, the networkTTL is always 1 over the entire network. When manually configuring this topology it can create a lot of problems when you want to add more brokers, every new broker brings, number of current brokers + 1 configuration changes. This topology is a nice example how the



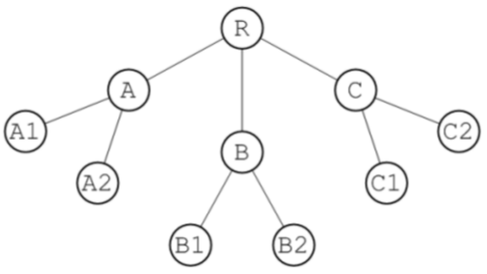
(a) Hub and spoke topology



(b) Mesh topology



(c) Complete Graph topology



(d) Tree topology

Figure 9: Different topology styles
Source: Using Networks of Brokers [4]

network will be build when discovery is enabled on all of the brokers in the same broadcast domain. Every broker only configures their own transport connector to allow discovery and their own network connector to act as a discovery agent.

The tree topology from figure 9d is a typically example ow how a network of brokers shall look like after a certain amount of time, where also the physical topology. First the brokers are setup from multiple domains to connect to root broker, then after some time other sub brokers are begin connected in side the local domain to dived the load.

The concentrator topology as already shown in figure 8 can be very useful when you anticipate that your network will receive a large number of incoming connections, which can overwhelm a single broker.

5 Design and development

In this section the used testbed will be explained in different configurations and some pieces will be enlightened with configuration examples or pieces of code.

5.1 Test Setup

To start with I created a very small test setup, consisting off 3 brokers connected to each other with duplex connection. There was no special filtering applied and no other big changed applied to the setup. The 3 brokers all active on the same broadcast network and to start with no extra producers or consumers where inserted in the network.

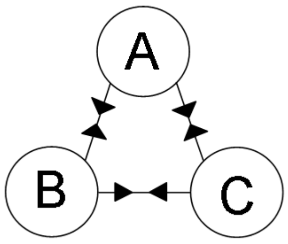


Figure 10: Initial test setup

```
<blueprint ... >
  ...
  <broker brokerName="BrokerA" brokerId="A" ... >
  ...
    <!-- The transport connectors ActiveMQ will listen to -->
    <transportConnectors>
      <transportConnector name="openwire1" uri="tcp://0.0.0.0:60001" />
    </transportConnectors>
  ...
  </broker>
</blueprint>
```

Listing 1: Simple Configuration Broker A

```
<blueprint ... >
  ...
  <broker brokerName="BrokerC" brokerId="C" ... >
  ...
    <networkConnectors>
      <networkConnector name="linkCtoA" uri="static:(tcp://...105:60001)"
        duplex="true" networkTTL="5"></networkConnector>
      <networkConnector name="linkCtoB" uri="static:(tcp://...106:60001)"
        duplex="true" networkTTL="5"></networkConnector>
    </networkConnectors>

    <!-- The transport connectors ActiveMQ will listen to -->
    <transportConnectors>
      <transportConnector name="openwire1" uri="tcp://0.0.0.0:60001" />
    </transportConnectors>
  ...
  </broker>
</blueprint>
```

Listing 2: Simple Configuration Broker C

As we can see in figure 10 it is a very simple setup with every node attached to every node in the network, no special configuration on the “main” broker A, only a simple open transport controller as we can see in listing 1. In listing 2 is visible how the network connectors are setup and broker C creates his connections to A and B. Broker B has similar configuration only 1 connection to broker A, since the connection from broker C is in duplex mode.

5.2 Producers and Consumers

Because this configuration was working as the manual said it should do, there was nothing much to test from this perspective. Because creating some load, sending some messages to the system seemed to be a good solution, i created a simple producer based on the example given[4]. The producer had a simple connection to a secondary openwire transport connector on broker A 3.

```
ActiveMQConnectionFactory connectionFactory = new
  ActiveMQConnectionFactory("tcp://...105:60002");
```

Listing 3: Simple Producer URI configuration

The producer was sending 25 messages on the queue “TEST.NICK” and with a very simple consumer these messages where consumed from a different broker. This example showed that the all the messages send from the producer to broker A, nicely stayed on broker A until a consumer joined the network and subscribed itself to the test queue. When the consumer subscribed to the test queue it was visible that the messages left broker A, to be forwarded to the broker where the consumer was connected to. When only a few messages where consumed the rest stayed on that broker. This behavior is already been discussed in this report and is a nice example of stalled messages.

A consumer has almost the identical connection setup as the producer, as showed in listing 3. But for testing and better read-ability the consumer connects to an other broker and again a different transport connector on that broker. When executing tests with multiple thousands of messages, it was nice to see incoming and out going connections on different port numbers to categorize them.

5.3 Changing Settings

As shown there are only small changes made to the default configurations, where duplex is set to true and the networkTTL has been increased. The duplex mode was done to be able to connect any consumer or producer anywhere in the network and that all the messages on the queues where spread around. The networkTTL has to be high enough to reach any point in the network, taking any route. Because the maximum route possible to take consumes at least 2 brokers, the networkTTL has be 2 or higher, I set it to 5 so that the configuration could change over time.

[4]Websource: `http://activemq.apache.org/hello-world.html`

5.4 Tests

To test out of the options from the default and adjusted configurations where working as expected i applied some tests to my setup. The three brokers from the start are still there, only the connections to them are removed. Every broker is in its own little network. The producer has a failover connection string, so that when a broker is not answering in time or tells the producer to back off a little, the producer can send it to the other brokers. By default when creating a failover URI the connections are made randomly, meaning every broker in the URI will get around the same amount of messages. Because that was not something that i wanted to test out, that value was changed to false. This way the first broker is always the preferred one, when that one is failing the second is chosen or at last the third.

```
 ...
public static void main(String[] args) throws Exception {
  for (int i=0;i<3000;i++){
    thread(new HelloWorldProducer(), false);
  }
  Thread.sleep(15000);
  System.out.println("Done");
}
 ...
public static class HelloWorldProducer implements Runnable {
  public void run() {
    try {
 ...
      ActiveMQConnectionFactory connectionFactory = new
        ActiveMQConnectionFactory("failover:" +
        "(tcp://...105:60002,tcp://...106:60002," +
        "tcp://...107:60002)?randomize=false");
 ...
      producer.send(message);
 ...
```

Listing 4: Simple spam Producer

When executing this code on a single broker this already overloads the broker. Because there is no timeout in the first for loop this creates a lot of messages within a short time frame. The 3000 is also because when we go higher, 5000 for example there where some difficulties with the connection states. The reason why the brokers are not connected to each other in this test is to see how many messages will end up on each broker. When the brokers are connected, there is no good way of telling how many messages there are in each broker. For that I also used a slightly modified consumer, which connects to 1 broker, counts the messages, and then rerun on the other brokers.

When executing the code from the producer 4 it was already visible in the brokers that it cost a lot of load to handle all these requests. The same also holds for the consumer 5 where it only connects to one broker. Because the test was more in the sending then the retrieving of the messages, i added a small delay in the consumer so that i could collect all messages without overloading the broker non stop.

When executing the test it already became quite clear that the mechanism of failover works quite well, when sending all message in the same speed to the brokers, most of them ended up on the first broker in the connection URI (Broker A), and the rest of the messages ended up on both broker B and C. After running this test twice it became clear that this was default behavior and the results where about the same. Then for testing with not only the load of the producer, I also used the program stress [5]in combination with nice [6]to create load on one of the brokers time for time.

```
 ...
public static void main(String[] args) throws Exception {

  for (int i=0; i<3000;i++){
    thread(new HelloWorldConsumer(), false);
    Thread.sleep(25);
  }
  Thread.sleep(10000);

  System.out.println("Done:␣" + count);
}
 ...
public static class HelloWorldConsumer implements Runnable,
    ExceptionListener {

  public void run() {
    try {
      ActiveMQConnectionFactory connectionFactory = new
        ActiveMQConnectionFactory("failover:(tcp://...107:60004," +
        "tcp://...107:60003)?randomize=true");

 ...
if ( message != null ) count++;
 ...
```

Listing 5: Messages counting consumer

Because the broker was running under the same user as the stress command and both the programs had the same priority, test results did not came out as expected. The broker still used some CPU power while the stress command should have consumed them all. There fore i used the tool nice to decrease the nice value (Increase the priority) of the stress command will executing the next tests. Using the stress command with all options enabled, CPU, HDD, IO and VM, made sure that the broker was busy enough to create some different results.

As shown in table 2 the first 2 runs there was no extra load generated on one of the brokers, messages mostly go to A but also some to B and C. When we stress broker A it is visible that only a very small amount of messages gets there and the rest goes to B and C. Where broker B has almost all messages. When we stress broker B it is visible that broker A gets almost all messages B almost none and the rest is for C. Almost the same holds for stressing broker C.

When you look at the first test and the second, you wont expect that much of a difference, but actually there is. When we take a closer look at the first test we see that broker A has around 1400 messages and both broker B and C around 750. All messages where send in a very short time frame, where initially they have to go to broker A, when broker A is busy or overloaded by the requests back off messages will get send back from time to time. This is the time where the next broker in line is contacted to deliver to. All initial connections go to broker A, when after some time out period there is no answer, or there is back off message, other brokers will get contacted.

[6]`http://weather.ou.edu/~apw/projects/stress`

[6]`http://en.wikipedia.org/wiki/Nice_(Unix)`

Stressed:	None	None	A	A	B	B	C	C
Broker A	1442	1370	122	156	2298	2236	2054	1753
Broker B	795	953	2111	2038	94	167	808	1096
Broker C	763	677	767	806	608	597	138	151

Table 2: Test Results

When we look at the third test you would suspect that a lot more messages will end up on broker C then there are now. But when we think back how this mechanism works it makes more sense, because broker A is so overloaded it will reject a lot of messages and a lot of messages will receive a time out. Because this tests also takes a lot longer to complete, the rate at which the messages ends up on broker B is a lot slower then they arrived at broker A in test one and two. This way broker B can handle a lot more messages then broker A in the firsts tests.

Also because of this behavior it is visible that in test five and six (Broker B stressed) A gets a lot more messages. While in tests seven and eight (broker C stressed) broker A will receive again less messages because B can handle more then previous tests.

5.5 Extra

As shown in table 2 I only conducted 2 test runs on each stress test on a broker, nothing where I change the parameters of the servers to tweak or work together to balance load in other ways. This was due to the fact that these tests where only conducted in the last day of my research. Due to the failed initial installation and the more theoretical approach of this research, the test are only seen as extra and are there to proof in the smallest sense how and that the load balancing works. Because these tests where conducted on a single server running multiple Virtual Machines, the stress command also had effect on the entire machine. Therefore testing on different kinds of hardware, different locations and over different kind of connections can show other test results then displayed here.

6 Conclusion

When looking back on the entire research and the included research question, the goal is not ultimately met. The goal of the research was to investigate in a real time load balancing mechanism, to automatically balance load in real time services operations. Because the configuration options within the FuseESB platform won't allow values which can make this happen.

Reading into everything because the lack of knowledge when starting the research took quite some time. After reading all matrial for a week, implementing the demo source code from the UvA testbed seemed a good start to setup and investigate into the possibilities within FuseESB software. The first try to recreate the test bed did not worked out because the wrong module was used. After discussion an other angle of investigation was used where the configuration options within the FuseESB software are investigated and figured out if these are useful within a big implementation. Creating a test setup where the load balancing options could be tested was successful, however implementing the source code from the earlier testbed did not worked out well.

There are inside the FuseESB environment a lot of configuration options available which result in various ways of balancing load, none of them really reflect to that goal we are trying to achieve. All the options rely most of the time on link basis, this means based on the network connectors that are created too and from brokers. When configuring options for these network connectors most of the time the network connection is setup on a static way. This does not work within an environment where most of the network connections have to be created on a dynamic way.

This reflects to the option where filtering is used, when a topology is made in advance filtering can have a lot of benefits on the load of the links between brokers or sites. But when filtering is applied on every new network connection which will be made dynamically this can work against the goal of that newly created broker.

Creating a dynamic network connection is only possible with discovery or updating the configuration file on every broker and update the running configuration. When discovery is enabled, which has its pros and cons, all settings to the discovery connector applies to all the connections made with the discovery protocol.

When setting up a network of brokers, the topology plays a big role in the success of the created network. When creating a bigger network which consists out of multiple sites, which are or are not geographically totally different, setting up static brokers on the edges to communicate with each site can be very beneficial. Inside a site there can be taken advantage of the discovery protocol to discover all the brokers, but setting up one or more static brokers to always rely on can be very useful too. The discovery protocol can be useful to automatically create a full mesh topology so that in a site every broker has a direct connection to any other broker. So that producers and consumers can attach every where and be connected inside this network of brokers.

When there is a certain situation where there are a lot of producers, as explained earlier, the previous topology is also something that you do not want. So thinking about how the network is going to look like can be the biggest gain in such a load balancing ESB platform. Create solid end to end point through multiple sites, setup discovery for a lot of joining and leaving of consumers/producers, when a lot of traffic is expected plan for it.

7 Summary and future work

7.1 Results achieved

The presented report provided analysis of load balancing methods in applications to ESB platform for services composition and proposed that creating a good topology already contributes in balancing load and setting up predefined paths can help in some cases to increase performance. Setting up basic fail over connectors showed that that could be accomplished quickly and broker independently.

Due to the time limits for the presented research, it was not possible to fully design and test a module which could dynamically informs other brokers of the current load and status of the running service. Future work will include creating such a module, which informs other brokers of current load statuses. Based on these assumptions creation of new brokers can be redirected or placed on totally different server or other underlying created service, other routes can be used to transfer messages to reduce the load on the overloaded service.

Creating a setup with all normal settings on a small environment already showed that configuring fail over and connectors was a feasible option. Using this knowledge it can help in future research based on this.

There is no research been done in this field, no publications are made where balancing load in any way is researched within an ESB based platform, finding references to papers therefore was not easy and hard to start with.

7.2 Lesson learned

Having prior knowledge of the ESB platform and all of the functions and implementations, can have great advantages before starting to investigate further on this matter. Because this already took away a week of the research time before I could continue investigating in the documentation found.

The demo source code worked with a different underlying system of creating and deploying virtual machines, recreating this setup to test newly created modules can proof its work better then the setup created this research without the working source code. When such a setup is created, document it so that recreation takes less time to set up. Loosing time setting up a system that in the end will not work with the provided tools is not a nice starting point.

7.3 Future research

Future work will include creating the load monitoring module, which could inform the network of brokers about current load status. In wider scope, the future research can look into creating a larger test scenario where multiple machines and different sites are used. This should create a better real life example that reflects the typical topology (multi domain / multiple sites) that should allow wider and more targeted experimentation.

8 References

References

[1] Yuri Demchenko Canh Ngo Pedro Martinez-Julia Elena Torroglosa Mary Grammatikou Jordi Jofre Steluta Gheorghiu Cees de Laat. Gembus based services composition platform for cloud paas. 2011.

[2] FuseSource. *Fuse MQ Enterprise - Connectivity Guide*, 7.0 edition, Apr 2012.

[3] FuseSource. *Fuse MQ Enterprise - Fault Tolerant Messaging*, 7.0 edition, Apr 2012.

[4] FuseSource. *Fuse MQ Enterprise - Using Networks of Brokers*, 7.0 edition, Apr 2012.

[5] M. Grammatikou C. Marinos S. Kafetzoglou P. Martinez-Julia Y. Demchecko R. Hedberg J. Jofre S. Gheorghiu A. Perez-Morales E. Torroglosa M. Debowiak L. Dolata K. Dombek M. Gorecka-Wolniewicz T. Wolniewicz S. Milsom. Gembus cookbook. march 2012.

[6] Canh Ngo Yuri Demchenko. Joint gembus/esb testbed at uva. 2011.

A Appendix

A.1 Client Configuration

A.1.1 Failover connection URI

```
failover:(tcp://host-a.com:61616,
   tcp://host-b.com:61616)?initialReconnectDelay=100
```

Listing 6: Client failover connection URI

A.1.2 Discovery connection URI

```
discovery://(static://(tcp://host-a.com:61616))
```

Listing 7: Client static discovery connection URI

A.1.3 Discovery multicast connection URI

```
discovery://(multicast://default)
```

Listing 8: Client multicast discovery connection URI

A.1.4 Fanout URI

```
fanout://(multicast://default)
```

Listing 9: Fanout multicast URI

A.2 Broker Configuration

A.2.1 Default transport connector

```
<transportConnectors>
   <transportConnector name="openwire"
      uri="tcp://0.0.0.0:61001"/>
</transportConnectors>
```

Listing 10: Default broker transport connection

A.2.2 Default network connector

```
<networkConnectors>
   <networkConnector name="linkToBrokerB"
     uri="static:(tcp://localhost:61002)"
     networkTTL="5" duplex="true" />
</networkConnectors>
```

Listing 11: Default broker network connection, duplex enabled

A.2.3 Static propagation

```
<networkConnectors>
    <networkConnector name="linkToBrokerB"
        uri="static:(tcp://localhost:61002)"
        networkTTL="3">
        <staticallyIncludedDestinations>
            <queue physicalName="TEST.FOO"/>
        </staticallyIncludedDestinations>
    </networkConnector>
</networkConnectors>
```

Listing 12: Static propagation

A.2.4 Destination filtering

```
<networkConnectors>
    <networkConnector name="linkToBrokerB"
        uri="static:(tcp://localhost:61002)"
        networkTTL="3">
        <dynamicallyIncludedDestinations>
            <queue physicalName="TRADE.STOCK.>"/>
            <topic physicalName="PRICE.STOCK.>"/>
        </dynamicallyIncludedDestinations>
        <excludedDestinations>
            <queue physicalName="TRADE.STOCK.NYSE.*"/>
            <topic physicalName="PRICE.STOCK.NYSE.*"/>
        </excludedDestinations>
    </networkConnector>
</networkConnectors>
```

Listing 13: Destination filters, included and excluded

A.2.5 Shortest route

```
<networkConnectors>
    <networkConnector name="linkToBrokerB"
        uri="static:(tcp://localhost:61002)"
        networkTTL="3"
        decreaseNetworkConsumerPriority="true" />
</networkConnectors>
```

Listing 14: Network connector for choosing the shortest route

A.2.6 Multicast discovery Agent

```
<transportConnectors>
    <transportConnector name="openwire"
        uri="tcp://localhost:61001"
        discoveryUri="multicast://default" />
</transportConnectors>
```

Listing 15: Enabling a Discovery Agent on a Broker

A.2.7 Fanout protocol URI

```
fanout://(multicast://default)?initialReconnectDelay=100
```

Listing 16: Fanout protocol URI

A.2.8 Seperate topics and queues

```
<networkConnectors>
   <networkConnector name="queuesOnly"
      uri="static:(tcp://localhost:61002)"
      networkTTL="3" conduitSubscriptions="false">
      <dynamicallyIncludedDestinations>
         <queue physicalName=">"/>
      </dynamicallyIncludedDestinations>
   </networkConnector>
   <networkConnector name="topicsOnly"
      uri="static:(tcp://localhost:61002)"
      networkTTL="3">
      <dynamicallyIncludedDestinations>
         <topic physicalName=">"/>
      </dynamicallyIncludedDestinations>
   </networkConnector>
</networkConnectors>
```

Listing 17: Separate configuration of topics and queues