jamk.fi

MESSAGE BROKERS AND RABBITMQ IN ACTION

Tsuri Kamppuri

Thesis
May 2014

Degree Programme in Media Engineering
Technology, communication and transport

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s)	Type of publication	Date
Kamppuri, Tsuri	Bachelor's Thesis	15.5.2014
	Pages 40	Language English
		Permission for web publication (X)

Title
MESSAGE BROKERS AND RABBITMQ IN ACTION

Degree Programme
Media Engineering

Tutor(s)
Manninen, Pasi

Assigned by
Paytrail Oyj

Abstract

The objectives of this bachelor's thesis were to study the concept of messaging and messaging systems in the domain of Information Sciences, and to research the applicability of RabbitMQ for Paytrail Oyj as a replacement for pre-existing systems.

The thesis discusses the history of messaging and message queues, and the topologies, patterns, internal operational models, and usable protocols for messaging brokers. Theoretical part also compares RabbitMQ and ZeroMQ messaging solutions. The main focus of message broker applications was in RabbitMQ's suitability for Paytrail's needs as a payment service provider.

The practical section researched the substitutability of a web API example with a message queue solution. The section covered the benefits and different stages of the implementation through command line examples. At the end of the practical part additional examples were given where message queues could be useful from the perspective of Paytrail Oyj.

The results of the thesis were a theoretical information package to support the selection of a message broker solution and instructions for introducing RabbitMQ into an existing system coupled with additional ideas on other suitable applications for message queues. The additional ideas were made from the perspective of the client directly applicable for future development by the client.

The analysis and conclusion stage of the thesis weighed the pros and cons of RabbitMQ and ZeroMQ for different purposes and especially for Paytrail Oyj through their features, strengths and weaknesses, and the experiences from web API replacement implementation. As the final results RabbitMQ was found to be the best message queue solution for the client's purposes. By following the additional application ideas and examples for implementation, the client can begin mapping the use of message queues in their system and working on the implementation.

Keywords

messaging, message queue, message broker, RabbitMQ

Miscellaneous

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

KUVAILULEHTI

Tekijä(t) Kamppuri, Tsuri	Julkaisun laji Opinnäytetyö	Päivämäärä 15.05.2014
	Sivumäärä 40	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty (X)

Työn nimi
MESSAGE BROKERS AND RABBITMQ IN ACTION

Koulutusohjelma
Mediatekniikka

Työn ohjaaja(t)
Manninen, Pasi

Toimeksiantaja(t)
Paytrail Oyj

Tiivistelmä

Opinnäytetyön tavoitteena oli esitellä viestimisen konseptia ja viestijärjestelmiä tietojenkäsittelytieteen viitekehyksessä sekä tutkia RabbitMQ-viestijonototeutuksen soveltuvuutta Paytrail Oyj:n tarkoituksiin korvaavana tekniikkana.

Opinnäytetyössä käytiin läpi viestimisen ja viestijonojen historia, viestijärjestelmien topologioita ja malleja sekä sisäistää toimintaa ja käytössä olevia protokollia ja vertailtiin kahta eri viestijärjestelmätoteutusta. Ohjelmistoissa keskityttiin tutkimaan RabbitMQ:n soveltuvuutta erityisesti Paytrail Oyj:n näkökulmasta maksupalveluntarjoajana.

Toteutusosassa tutkittiin esimerkkisovelluksen korvattavuutta RabbitMQ-viestijonototeutuksella. Toteutuksessa käytiin läpi implementaation hyödyt ja eri vaiheet komentoesimerkein. Toteutusosan lopussa esiteltiin myös muita mahdollisia Paytrail Oyj:n tarkoituksiin soveltuvia kohteita, joissa viestijonoista voisi olla hyötyä.

Opinnäytetyön tuloksina oli viestijonosovelluksen valintaa tukeva teoreettinen osio ja suunnitelma ja ohje olemassa olevan toteutuksen korvaamiseen RabbitMQ-viestijonototeutuksella sekä ideoita viestijonojen muihin käyttökohteisiin. Käyttökohdeideat olivat tilaajan näkökulmasta suunniteltuja ja suoraan hyödynnettävissä jatkokehityksessä.

Pohdintaosassa punnittiin viestijonojen ja tarkemmin RabbitMQ:n ja ZeroMQ:n käytettävyyttä Paytrail Oyj:n tarkoituksiin ominaisuuksien, hyötyjen ja haittojen sekä esimerkki-implementaation kautta. Lopputuloksena todettiin RabbitMQ:n soveltuvan toimeksiantajan tarkoituksiin parhaaksi vaihtoehdoksi. Opinnäytetyön käyttökohdeideoita ja toteutusesimerkkejä seuraamalla tilaaja pystyy kartoittamaan viestijonojen toteutuskohteet järjestelmässään ja aloittamaan niiden toteuttamisen

Avainsanat (asiasanat)

messaging, message queue, message broker, RabbitMQ

Muut tiedot

Contents

FIGURES

TABLES

TERMS AND ABBREVIATIONS

AMQP

Advanced Message Queuing Protocol, a binary-format open messaging protocol

Binding

Entity that creates a relationship between a message queue and an exchange

Broker

Program that provides messaging services

Consumer

Broker user that pulls messages from queues

Erlang/OTP

Erlang is a programming language created to the development of highly concurrent and fault-tolerant applications for telephony applications by Ericsson. Open Telecom Platform (OTP) is the open source distribution of Erlang and an application server written in Erlang

Exchange

Entity within the server receiving messages from producers and optionally routing these to message queues

JMS

Java Message Service API, a messaging standard for communication between components based on Java Enterprise Edition

Key-value store

Data storage that uses a dictionary type data model where data is represented as key-value pairs

Message

Collection of data that is sent between systems

MVC

Architectural software design pattern for segmenting code into Models, Views and Controllers

Queue

Message storage

Producer

Broker user that pushes messages to queues

Round-robin

Algorithm for scheduling processing without prioritising

Routing key

Virtual address that an exchange may use to decide how to route a message

Software Bus

Software architecture model for connecting software modules

Telnet

Protocol for providing text-based communications between machines over a network later inspiring protocols like SSH

UX

User experience

vhost

Virtual host. In RabbitMQ's case these are addresses within the broker server

WAN

Network that covers a large area and large number of smaller networks

1 INTRODUCTION

1.1 Thesis background

The idea of ensuring the integrity of data transmissions has been etched into the very building blocks of the global network that we know today as the Internet. It is so crucial in fact that the concept of acknowledging a packet –an enclosed digital message– transfer is part of multiple transportation layer protocols, such as the Transmission Control Protocol designed already in 1974 (Cerf, Dalal & Sunshine 1974). Transfer acknowledgment and data fidelity also play a key role in Message Oriented Middleware (MOM):

> *A MOM system assures that packets of data or messages are delivered from one destination to another regardless of the state of the network or the recipient of the message. If the message can not be delivered to the receiver at the time the sender sends it, the message will wait in a queue until it can be delivered by channels that connect the MOM services together. The sender and receiver have no knowledge of each other's physical location or platform specifics. (Van de Putte, Adinarayan, Haddon, McCarty, Peltomäki & Quixchan 2005, 12.)*

Messaging middleware has its place also in smaller scenarios, but the nature and exponential growth of the Internet have made message brokers almost a required component in any larger systems. The broad concept of messaging is simple at first glance but the tackled problems are complex in similarly complex environments and choosing and implementing just the right messaging solution for the task can be arduous and require domain expertise to actually utilise effectively.

The principles and patterns of messaging are useful throughout the various fields in the domain of information sciences. This thesis will discuss them mainly in the context of software architecture and Internet applications in particular but also veer towards general notions on messaging for the sake of understanding how various message brokers operate on internal level. It will explain the intricacies behind

message oriented systems and then zoom in on specific message broker products as the solution to decoupling challenges presented in the following chapter.

1.2 Paytrail Oyj

Paytrail Oyj was originally known as Suomen Verkkomaksut (“Finnish Webpayments”) when it was founded in 2007 in Jyväskylä for the purpose of providing businesses a centralised channel for starting to accept payments online. Paytrail was the first of its kind in Finland to receive payment institution license.

As the customer base began expanding to other countries, the new name along with a new brand were developed and adopted. Today Paytrail has over 4000 business customers in over 10 countries, more than 350 partners, and a recently launched payment solution by the name of Paytrail Account. Their systems handle tens of millions of euros each month. (Paytrail – Our Story n.d.)

The subject for this thesis was proposed by Paytrail when the company needed a research of potential solutions for separating and isolating applications in order to eliminate service upgrade down-time and increase updatability and fault-tolerance through asynchronic decoupling of individual components.

The separation of services could be achieved through various means but since the company already had previously invested resources in message brokers, they were a natural choice as the subject of this research. The tentative choice of RabbitMQ was made by the company's software architecture team.

This thesis provides Paytrail invaluable insight into the prospects that message broker solutions can provide in various areas, including but not limited to application decoupling, transaction reliability and fault monitoring. The validity of RabbitMQ as the best solution for Paytrail’s case was also assessed.

1.3 Research mission and objectives

The objective of this thesis is to do research on the concept of messaging and to compare different message brokers and message queue solutions. The theoretical part of the thesis studies message queue solutions' maturity and applicability in production environments as a method of separating services to improve availability and to reduce complexity. The practical section of this research will be performed from the viewpoint of existing services so that the results can serve as a direct action plan for Paytrail. Through feature analysis and a case study this thesis determines if the tentatively chosen message broker RabbitMQ is a fit choice for Paytrail.

In the context of this thesis messaging is viewed mostly from the perspective of message brokers that operate over the network. According to a definition by Margaret Rouse (2005), interprocess communication can also be implemented using semaphores, sockets or shared memory.

1.4 Research methodologies

The research is divided into two parts: theoretical background study and practical implementation steps. The theoretical part studies the history, purpose and functionality of messaging on a broad level, and then continues to narrow down on four different message broker products. The practical implementation part discusses actual use cases for message brokers and takes a closer look at a situation where a web API has been created between services and a message queue is brought in to decouple them.

The viability of each message queue application presented in the thesis is assessed by examining the benefits and disadvantages of each product. The applicability of the premeditated product choice for Paytrail, RabbitMQ, is further delved into via a web API replacement scenario.

1.5 Information sources

This thesis uses both digital and analogue sources as references. Digital sources include e-books, electronic product manuals and tutorials, blog articles, forum answers, and presentation slides and recorded videos of presentations. Some of the e-books are privately owned copies, some of them are licensed for educational use via faculty credentials, and some of them are partly or fully accessible to the public. For this thesis, analogue sources refer exclusively to books.

2 MESSAGING

2.1 Messaging in information sciences

Messaging has become a ubiquitous concept since its manifestation in the world of computing in the 1980s. One of the most identifying distinctions among messaging systems is whether they use synchronous or asynchronous message passing.

Synchronous messaging

Synchronous messages are transactions that block further process until the transaction finishes. For software developers the concept of synchronicity is well-known from object oriented paradigms where operations are executed in the specified order unless specifically instructed otherwise. Maryka (2009) notes that historically the World Wide Web has been largely synchronous until the recent shift from websites to web applications that process and display data transported synchronously over HTTP in asynchronous fashion as shown in Figure 1.

Asynchonous messaging

In contrast, asynchronous messages do not require immediate response from the message handler. Asynchronous messaging can provide scalability by delaying data transfers, post-processing capabilities by providing filtering pipelines (see Chapter 2.2), and resistance against connectivity issues by persisting messages until a consumer is able to receive the message. On the negative side asynchronous messaging is more complex to handle and implement requiring additional components to validate transaction statuses. (Janssen n.d.)

Figure 1 reveals the asynchronic nature of today's web through timeline inspection: while each HTTP query in itself is synchronous, the holistic user experience is asynchronous as the user interface is free to update while the data is transferring in the background.

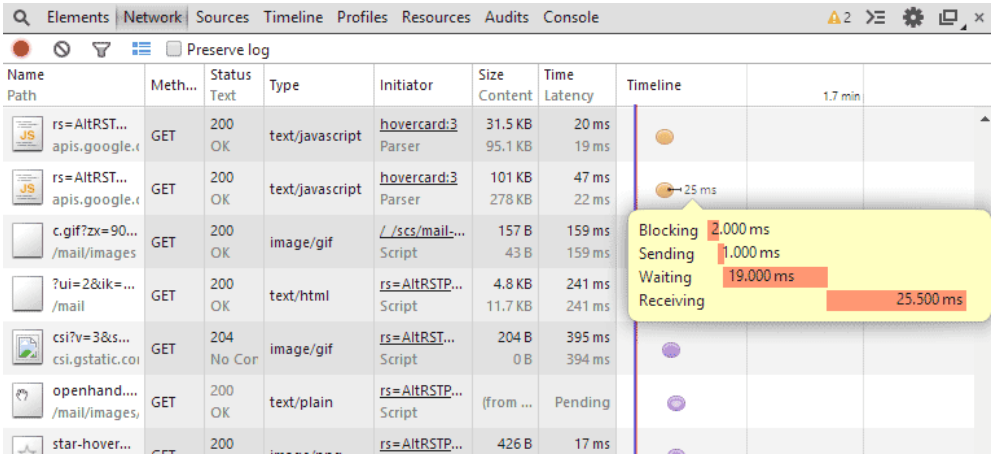


Figure 1. Web inspector showcasing asynchronous UX in Gmail

Messaging over the network

A classic example of a network messaging situation is email: an author sends a message at their desired time, and the recipients fetch and read the message at the time of their choosing. Just like in the real world there are delivery services such as UPS or Fedex, there are digital services in-between stakeholders to store and pass messages on. Sometimes message brokers are referred to as message oriented middleware, a type of intermediary between services or applications (Van de Putte & al. 2005, 12).

2.2 Messaging topologies

In the domain of messaging, a topology describes how connections between nodes are implemented. Understanding these operational differences can help grasp what kind of messaging solutions are best suited for different topics and scenarios.

Client – Server (hub)

Client – Server is a commonly used topology in routing data from single publisher to subscribers (Figure 2), also known as the store-and-forward pattern (Richardson 2008). This topology is the cornerstone of the “publish and subscribe” pattern when scalability is a requirement.

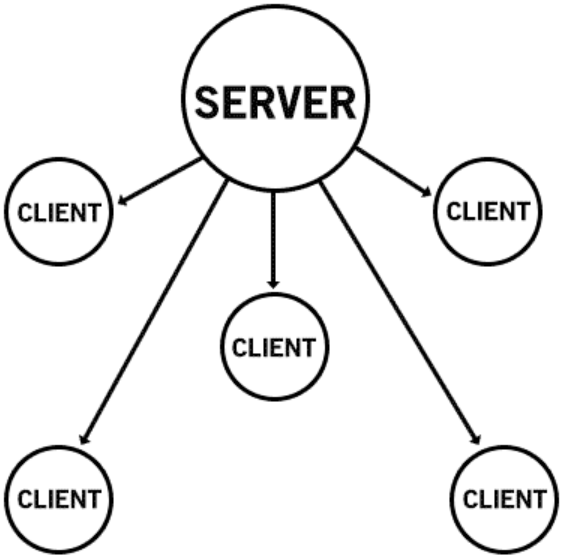


Figure 2. Master server and clients in a hub topology

Service Bus

Service busses are used for example in asynchronous applications in which messages are dropped onto a lane so that consumers of these messages can receive and respond to them at a later time (Figure 3). Service bus architecture provides great flexibility and de-coupling capabilities in the form of message storing and delayed messaging as is required for example in the case of emailing systems.

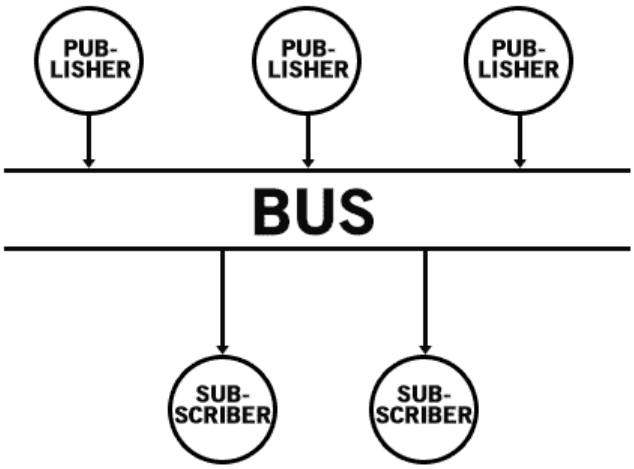


Figure 3. Publishes and subscribers connected to a service bus

Pipeline

In the article *Messaging Patterns in Service-Oriented Architecture, Part 1* Soumen Chatterjee (2004) states that a pipeline topology is used in systems where a message may pass through multiple messaging middleware servers that all run an action on the message be it consuming, redistributing, or filtering it (Figure 4). This kind of system is sometimes referred to as *flat system*. Pipes are an effective way to sequence a task into smaller sub-tasks for independent, organized processing.

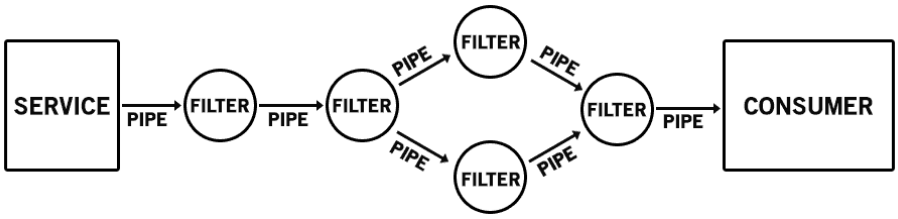


Figure 4. Pipeline topology

Peer Network

A peer network is, as the name suggests, a network of equally operating clients that share messages between them (Figure 5). Peer messaging is commonly encountered in peer-to-peer file sharing and private chat connections.

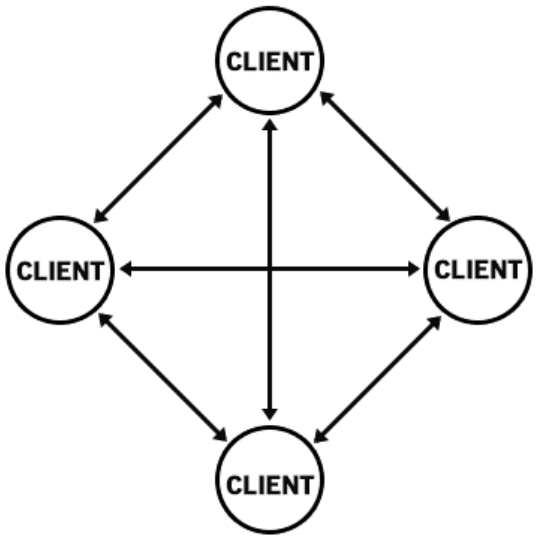


Figure 5. Clients connected to each other in a peer-network topology

Federation and Cloud

Federation and clouds step into the picture when one topology is not enough (Figure 6). Broyer (2011) defines a federated cloud effectively as a union of clouds that act as black boxes around architectures and services joining together various topologies. Clouds are generally an effective way of abstracting implementation details between two different systems, as could happen for example between systems using IBM WebSphere MQ and Microsoft Azure AppFabric Service Bus (Cloud Platforms & Integration n.d.).

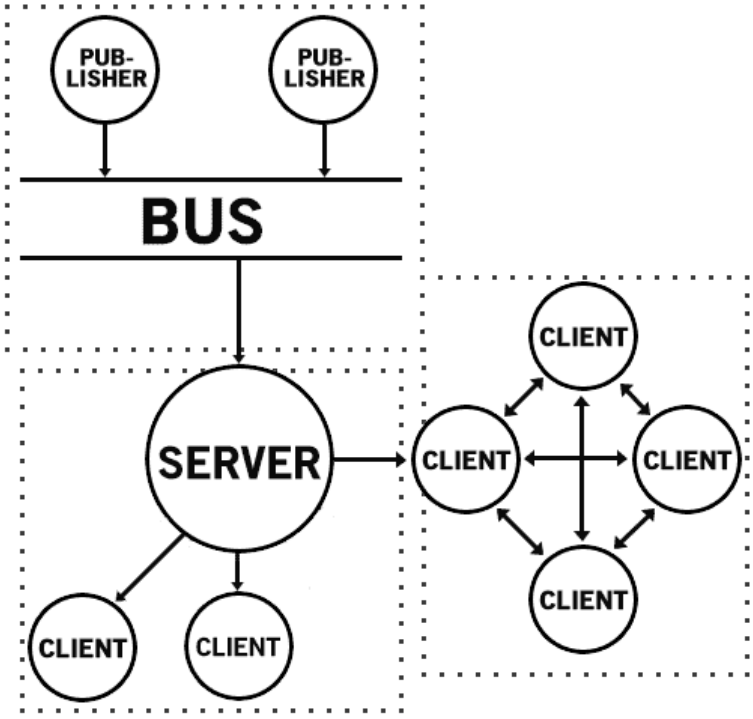


Figure 6. Federated service bus, hub and peer-network topologies

2.3 Messaging patterns

Patterns are the rules, the behavioural models of message brokers. They describe how a message broker functions: who can send messages, who receives what, in which format and when, and what happens before, after and during when a message is transported. There are some common operational patterns such as publish–subscribe and request–reply but message broker applications tweak and adapt patterns to best service their purposes.

Message type and channel patterns describe different varieties of messages and the attributes of the message delivery systems. Routing and service consumer patterns describe addressing mechanisms and behaviours for servers and clients. Contract patterns define interactional specifications between clients and servers, and construction and transformation patterns dictate what format goes in and what format comes out of the other end. (Chatterjee 2004.)

Publish-subscribe

Chatterjee (2004) defines publishers as service providers pushing messages to all interested consumers that have subscribed to the bus or channel. According to him a copy of the original event is replayed to each recipient, and the message is considered published only when all of the subscribers have been notified. Publish–subscribe model is based on the Observer Pattern and it can be useful for testing, system management and error solving.

Request-response

The pattern of requests and responses, or replies, is one of the basic methods of message exchange between information systems: a client sends in a request, and the server sends back a response based on the received request. This is largely how for example web service calls over HTTP operate. Pieter Hintjens (n.d.) states in ZeroMQ's official guide that request-reply pattern is likely the simplest way to use ZeroMQ where it is used for remote procedure calls and heavy load scenarios.

No-subscribers

Technically it is possible to use queues as final destinations without traditional consumers attached to them for example for the purposes of monitoring and gathering statistics. This pattern is not strictly without subscribers as in order to read the queues, broker plugins and add-ons or ephemeral one-time connections are required to act as message consumers. Instead of reading the actual messages themselves, the subscribers may consume messaging metadata like processing rate, and message size and message destination distribution for example.

3 MESSAGE BROKERS

3.1 History

Message broker products are the middleware that embody different messaging solutions. In 1983 Vivek Ranadivé from Teknekron Software Systems began working on an idea based on the ideals of a Software Bus to enable applications to share data in a standard fashion. That work would later be referred to as “The Information Bus”. Already in 1986 his ideas were put to use when Goldman Sachs, an American investment banking firm, launched cooperation with Teknekron to find solutions for the trading floor of the future. (Vivek Ranadivé – Teknekron Software Systems 2014.)

For nearly two decades the domain of message exchange was left for proprietary vendors and proprietary formats. Throughout the 1980s and 1990s message queuing kept evolving but in isolation since commercial message queue vendors strived for interoperability between client applications rather than worked on standardised interfaces for their message queuing products to utilise. In their book *RabbitMQ in Action* Videla and Williams (2012, 4) say that vendor lock-ins were largely the reason why message queuing did not find its way into the public’s favour during those years because they made it unviable option for smaller companies.

As the demand for interoperability grew alarmingly beyond the capabilities of existing solutions at the time, new contenders began emerging. The market witnessed the rise of open standard specifications like Java Message Service API (JMS) and Advanced Message Queuing Protocol (AMQP). The field of messaging had rapidly evolved into one with options for multiple vendors across different applications inside and across systems. (Videla & Williams 2012, 5)

Figure 7 shows a brief timeline of message queuing.

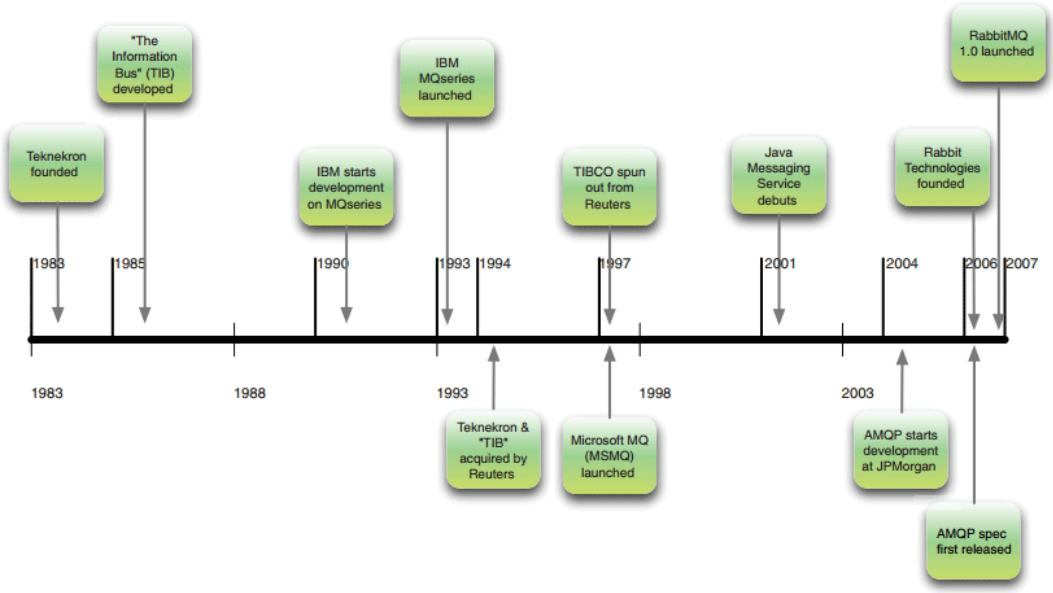


Figure 7. Short timeline of message queuing (Videla & Williams 2012, 5)

3.2 Features and benefits

3.2.1 Synchronous and asynchronous operations

Sometimes immediate action is required, sometimes the timing is unimportant, and sometimes a delayed execution is desired. Message queues can cater both ends of the spectrum with synchronous and asynchronous configurations.

Paddy Foran lists in his blog article *Top 10 Uses For A Message Queue* (2012) asynchronous communication as one of the top ten reasons to use message queues:

> *A lot of times, you don't want to or need to process a message immediately. Message queues enable asynchronous processing, which allows you to put a message on the queue without processing it immediately. Queue up as many messages as you like, then process them at your leisure.*

Synchronous message queues based on time serve well as task schedulers because brokers can be set to ensure that queue items are handled in given order (Foran 2012). The benefit of doing this over something like traditional cron jobs is not only guaranteed execution order but also the configurable built-in logic of resuming execution of a task if errors do occur.

3.2.2 Efficiency

Message queues can be very useful in creating scenarios based on batches. In situations where connections are limited or costly, or the data to transfer comes in such small portions that the overhead of transferring the data as single items would be too much, a message queue can be created to store and forward the data in time or amount based intervals, enabling the optimisation of data exchanges in regards to bandwidth.

Table 1 shows the processing speeds of different message broker solutions using AMQP and STOMP protocols to simultaneously enqueue and dequeue 20 000 messages each of which 1 KB in size. The benchmark was performed on a consumer-grade laptop with brokers running on default configurations. The table uses the following abbreviations:

- PET: Persistent Enqueue Time
- PDT: Persistent Dequeue Time
- TET: Transient Enqueue Time
- TDT: Transient Dequeue Time. (Salvan 2013.)

Table 1. High volume processing speeds of messaging brokers (Salvan 2013)

Engine	PET (s)	PDT (s)	TET (s)	TDT (s)
ZeroMQ_Transient	–	–	0.411023	6.957397
RabbitMQ_AMQP	9.184524	10.715612	7.266415	7.896451
ActiveMQ_STOMP	32.055833	35.928055	31.936827	34.554975
RabbitMQ_STOMP	34.955998	38.692212	32.89588	36.465085
Apollo_STOMP	40.542319	48.187756	34.44997	39.242244
QPID_AMQP	36.771103	51.162925	6.826391	6.949398
HornetQ_STOMP	65.988774	66.11078	64.582694	64.614696

3.2.3 Scalability

Scaling is divided into three different categories in an article *Scaling the Message Queue Service* by Oracle (2010):

- **Vertical scaling**

  Processing power is increased through hardware additions or system changes that expand available resources

- **Stateless horizontal scaling**

  Brokers and queues are added and load is redistributed over the new network

- **Stateful horizontal scaling**

  Brokers are clustered either through a master broker or shared state database. Brokers can occupy shared location or they can spread over a network.

In a blog article *Spikability - An Application's Ability to Handle Unknown and/or Inconsistent Load* Travis Reeder (2012) concludes that message queues are a simple and effective way of dealing with spiky behaviour. Queue processors operate at the maximum rate they are able to while being restrained by a set of boundaries from their host system so traffic spikes will not befall on systems and tip them over. When a queue fills up, its load can be distributed horizontally over multiple new queues. This balancing can be done manually as needed, or automatically on a cloud based solution such as IronMQ.

3.2.4 Abstraction

A message broker can hide away not only message destinations but also message formats providing a layer of abstraction on top of the application logic. This allows the broker's publishers to leave the responsibility of formatting and routing to the broker. For consumers the static front enables future flexibility in regards to application location and data formats, meaning that applications using data from the queue can be moved around and changed with ease.

For administrators solutions by third-party vendors can bring a sense of relief because as Richardson, Radestock and Garnock-Jones (2008, 16) put it in their

presentation *Introduction to RabbitMQ*, maintenance is taken care of by the vendors. If there is a known issue in the message broker –a bug or performance flaw– the vendor will fix it. Or, if the product is open source and the vendor is too slow to act or reluctant to fix the issue, users can take the source code and fix the problem themselves.

Clustered messaging is extremely complex to implement from scratch. Using a message broker by a known vendor with an existing community lets its users focus on application logic and let vendors and the powerful communities think of message transfers. (Richardson & al. 2008, 31.)

3.2.5 Monitoring and fault tolerance

Message queues expose bottlenecks within a system through message processing rate. If the application has been architecturally dissected into granular pieces, snapshots of the messages' states throughout different queue processes can be used to identify and analyse performance issues in business logics. (Foran 2012)

With message queues systems are resilient against partial system failures as a direct effect of decoupling. Even if a process within message queues fails, like the process of handling received messages for example, the message broker will still be able to receive messages and process them later when the system recovers. (Foran 2012)

In a distributed architecture, should the message broker itself fail, the failure state is universal to the whole system. This prevents discrepancies over application states as nothing is moving forward before the issues are resolved and application's consumers begin receiving new tasks from the queues.

3.2.6 Isolation and extendibility

By introducing abstraction layers to the system through message queues, it can be grown organically and extended as new requirements are discovered and the system

specification matures. Foran (2012) suggests that message queues are a great way to decouple applications and even data processing stages within applications.

Especially during development for the purpose of mocking values and ensuring system stability in a range of different cases, it is crucial to be able to isolate not only applications from each other but also data sources from their applications. Messaging can provide just that: with other parts of the application being independent from the data it receives, messages are easy to switch around, replay for testing and analyse after processing (Pasker 2008, 8).

3.2.7 Security

Security features depend on the message broker product. In RabbitMQ security is exercised through user accounts and access control lists. This allows handing over some parts of access control to the message broker, especially in systems that talk over publicly accessible services.

3.3 Protocols

For nearly two decades since the concept of service buses for applications was introduced vendors tied their customers closely to their solution stack by using proprietary, closed formats. If two vendors' applications were able to talk to each other, the bridging was made on application level. (Videla & Williams 2012, 4.)

In the beginning of 2000s a few projects –most notably JMS and XMPP which would later become a notable contender among instant messaging protocols– could provide openly accessible messaging but the holy grail, an open messaging standard, was yet to be discovered. The arrival of Advanced Message Queuing Protocol marked a new era for open messaging, leading the way to other comparable specifications like STOMP, Microsoft's MQ Telemetry Transport (MQTT) and Apache's OpenWire.

The following chapters will introduce one binary-based protocol, AMQP, and one text-based protocol, STOMP. The main focus will be on AMQP due to its importance both in the domain of message brokers but also in the founding of RabbitMQ.

3.3.1 Advanced Message Queuing Protocol

The computing world was in dire need of an open protocol for messaging because messaging middleware market was stuck on outdated products that provided little to no solutions for interoperability, service architecturing or virtualisation (Richardson & al. 2008, 13–14). Advanced Message Queuing Protocol is one such direct result of that need, and it was thus designed with portability in mind.

The driving force behind an open message exchange protocol was the bank industry: "AMQP got started because a bank, JPMorgan Chase, realised that they were losing money by locking themselves in in this way to multiple different vendors and thought 'well wouldn't it be easier if messaging was just plug and play like TCP and HTTP?'" When they began working on AMQP in 2004, JP Morgan looked into open Internet protocols like SMTP, HTTP and TCP for reference (RabbitMQ: An Open Source Messaging Broker That Just Works 2008).

Today AMQP is developed by a joint AMQP Working Group consisting of financial service providers Bank of America, Bloomberg, Credit Suisse, Deutsche Börse and JPMorgan Chase, and technology providers Informatica, LogMeIn, Microsoft, Red Hat, SITA, Software AG, US Department of Homeland Security and WSO2 (AMQP Members n.d.). This division of responsibility has been the key to AMQP's success because it ensures that any message queue solution built on AMQP outlasts any single supplier (RabbitMQ: An Open Source Messaging Broker That Just Works 2008).

The finalised version 1.0 specification of AMQP was released in October, 2012. It is worth noting that while version 0.8 and 0.9 of AMQP are similar to each other, version 1.0 is completely different. RabbitMQ implements version 1.0 through an experimental plugin. (RabbitMQ - Protocol Compatibility n.d.)

3.3.2 STOMP

STOMP (stylised *Stomp*), or Simple/Streaming Text Oriented Messaging Protocol, is an open interoperable wire format for message exchange with same driving principles as AMQP. Like AMQP, it draws the roots of its design from simple and popular protocols like HTTP and FTP. As is true for AMQP implementations, likewise all Stomp clients can communicate with all brokers that implement the Stomp protocol as per the specification, regardless of client technology – one can even connect to a Stomp broker with a telnet client. Stomp currently stands at version 1.2, released in October, 2012. (Stomp - The Simple Text Oriented Messaging Protocol 2012.)

Unlike AMQP, Stomp is fully text based format making it more analogous to their common ancestor HTTP in regards to appearance. Textual representation increases readability and simplicity at the expense of larger transfer payload due to verbosity. Instead of queues, Stomp uses an FTP-like SEND semantic with a destination to which consumers can subscribe to. Because destinations are not strictly mandated in the documentation, Stomp brokers may have differently defined and supported destinations. This makes porting code between brokers difficult. Stomp is, however, relatively lightweight and well supported by an active community. (Piper 2013) Stomp supports transactions that allow messaging to occur atomically (STOMP Protocol Specification, Version 1.2, 2012).

4 MESSAGE QUEUE SOLUTIONS

Message brokers each come with their own set of features, formats and protocol support. When asked how to choose a message queue solution, Dan Echlin (2013a) reminds that knowing the strengths and weaknesses of at least a few different messaging technologies is important because different products are best suited for different scenarios. Messaging layer can be implemented using also other than conventional message queue solutions, like Redis key-value store that is trending as of May 2014 (DB-Engines Ranking of Key-Value Systems 2014), however, they can require additional manual implementations that traditional message brokers would handle automatically (Echlin 2013b). The following chapters will introduce two message broker solutions.

4.1 RabbitMQ

RabbitMQ is a message broker solution by Pivotal Software written in Erlang/OTP. It is fully open source and has a large open source community behind it. At the time of writing RabbitMQ is at version 3.3.1 which was released on 29th of April, 2014.

4.1.1 History

The sparks that eventually led to the birth of RabbitMQ and the foundation of Rabbit Technologies were ignited in the early 2000s. While working on caching Java objects at Metalogic, Richardson came to learn what works in distributed computing environments, and what companies want for those environments. In 2004 Richardson began working on the building blocks that would later manifest themselves as RabbitMQ. (Videla & Williams 2012, 4–7.)

4.1.2 Anatomy and features

Each RabbitMQ server contains at least one node, and more frequently multiple in a cluster configuration. Exchanges and queues comprise virtual hosts that are used to

partition broker data. Connections are made to virtual hosts on a RabbitMQ node (see Figure 8). (Edelson 2011.)

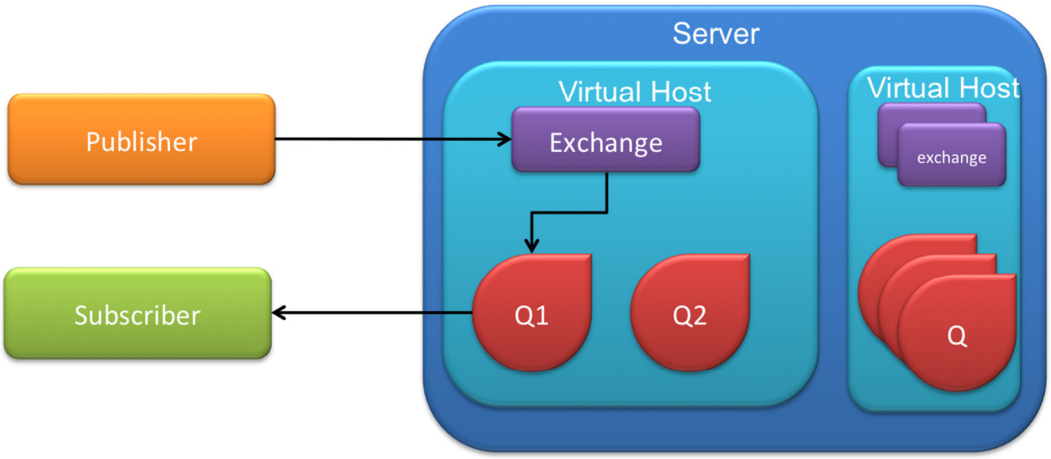


Figure 8. The main concepts of RabbitMQ (Rotem-gal-oz 2012)

In an introductory article *RabbitMQ and a Short Intro to AMQP* Arnon Rotem-gal-oz (2012) states that RabbitMQ exchanges can be temporary, auto-deleted or durable. Exchanges support direct and fan-out, and header and topic-based routing schemes for exchanges. Rotem-gal-oz continues that "queues, like exchanges cab *[sic]* be temporary, auto-deleted or durable."

In a blog article *The Merit of AMQP (part I)* Martin Sústrik (2013) states that almost all messaging protocols have two functionally distinct layers: connection and broker. Connection layer comprises any and all details related to connections between the nodes in a messaging system, such as handshake and authentication procedures. Broker layer or broker model defines what occurs to the messages in the queue between endpoints.

4.2 ZeroMQ

ZeroMQ (often stylised as ØMQ) is an asynchronous messaging library or concurrency framework geared towards performance. According to a definition by a Stack Overflow forum member under the alias Julien as quoted by Pronschinske

(2013), “ZeroMQ is a very lightweight messaging system specially designed for high throughput/low latency scenarios like the one you can find in the financial world.”

Like RabbitMQ, also ZeroMQ has lived through a major change between its versions: while most parts of the API are compatible between versions 2.2 and 3.2, some breaking changes were made (Hintjens 2013b). Its latest version is 4.0.3 that was released in November, 2013. ZeroMQ is an LPGLv3 licensed product of iMatix.

4.2.1 Features

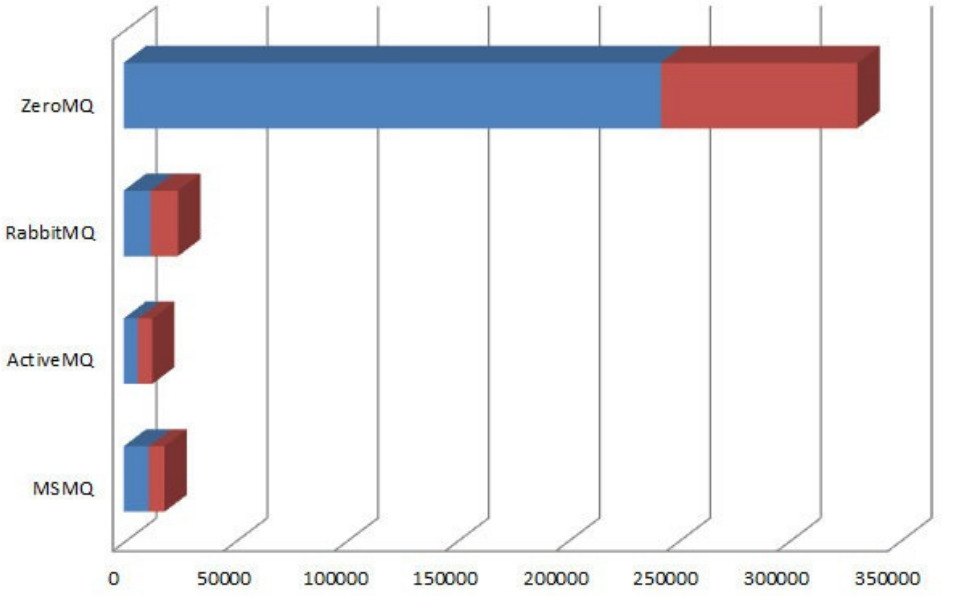
ZeroMQ is aimed at solving issues that occur in massive scale in the ballpark of hundreds of millions of messages per second. It is a performance-oriented, socket-based messaging library. ZeroMQ is driven by an open source expert community as opposed to RabbitMQ (ØMQ - Welcome to ØMQ for AMQP users n.d.)

In ZeroMQ, messages are carried in sockets that have types. The socket type defines its semantics, its policies for routing, queues etc. Different types of sockets can be connected together, such as publisher and subscriber sockets. The patterns in messaging manifest themselves through these sockets. (Hintjens 2013b.)

ZeroMQ derives its power from connections which are somewhat different to traditional TCP connections. In a traditional scenario, the server would have to be alive when a socket is created. In ZeroMQ, however, bits and pieces can be started autonomously: socket can be created prematurely and it’ll start delivering messages only when the server comes online. One ZeroMQ socket can also have many incoming and outgoing connections attached to it, and connections are automatically renewed after disconnects. (Hintjens 2013b.)

4.2.2 Strengths and weaknesses

Without a doubt ZeroMQ’s strength lies in its processing speed – it is staggeringly fast as shown in benchmarks by Mike Hadlow in the article *Message Queue Shootout!* (2011) (see Figure 9).



	MSMQ	ActiveMQ	RabbitMQ	ZeroMQ
Send	11406	6452	12278	242659
Receive	7021	6453	12281	88699

Figure 9. Processing benchmarks of messaging brokers (Hadlow 2011)

Pieter Hintjens (2013a), a veteran in the software industry, also commends the possibilities ZeroMQ has to offer:

> *It takes a few days of coding and then you will experience a kind of enlightenment, and see new dimensions in your software. Not only can you make much more ambitious architectures, but you can make them easily, and the results are fast and solid.*

The biggest issue of ZeroMQ is its level of abstraction. Instead of having a predefined, built-in broker, ZeroMQ is a library of broker-like functionality that enables, or in this case necessitates, users to write their own brokers (ØMQ - Welcome to ØMQ for AMQP users n.d.). This lightweight approach and abstraction of ZeroMQ also means that it does not do security – security has to be layered on top. In addition to security, many other pieces in the messaging system need to be implemented manually as well, thus in general ZeroMQ requires more development resources to tackle than RabbitMQ. Hintjens would also like to see ZeroMQ support threading. (Hintjens 2013a.)

5 INTRODUCING RABBITMQ INTO EXISTING SYSTEM

The following sub-chapters will explain step by step in practical fashion how RabbitMQ can be introduced as a replacement to a web API connecting two applications. RabbitMQ supports other protocols besides AMQP but according to measurements by Salvan (2013), AMQP as binary-based format performs very well. Using AMQP now also enables the simple switching of RabbitMQ to another broker solution that supports AMQP protocol at a later time.

5.1 Existing system

The existing system consists of two separate applications on separate servers each with their own database. Upon user data updates in the first application, the second application must reflect these changes. Both of their databases include a user table but each with a different structure. To propagate changes from the first application to the database of the second application a web API has been defined in the latter application to receive user data and save it to the database. Figure 10 shows component locations and data flows in current application architecture.

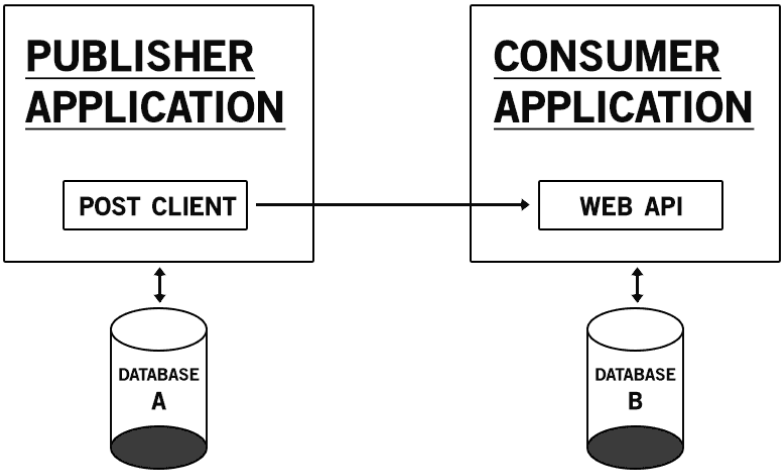


Figure 10. Existing application architecture

The applications reside under different domains on different machines and the data is passed over to a publicly available but restricted location. When user data is

updated in the first application, the updates are sent through the web API using POST queries. The API is protected against eavesdropping and unauthorised entries by connection encryption, firewall access rules and message tokenisation.

Figure 10 also shows that the applications are tightly coupled – they must both be online and offline at the same time to prevent data discrepancies from occurring. With a message broker between the applications, the second application can be offline when the first application updates user data, and the changes will propagate when the second application comes back online. The benefits of introducing a message broker to the system in this case are not limited only to the decoupling itself but include also its simplification: the maintenance of both applications is easier with reduced amount of application code to manage.

5.2 Replacement steps

In general, the very first step in bringing in a message broker to replace existing functionality in a system is firstly to define business requirements for the broker, and secondly to recognise data endpoints, i.e. to find where the data exchange joints of the system. These steps must be taken to determine the best suited message broker and the right operational pattern within the broker to match the task at hand.

Here the choice to investigate RabbitMQ was already made, so the next step was to identify the sections where common data is used and exchanged. When decoupling a larger system that could be a time-consuming task requiring thorough architectural inspection. Here, however, this was easy since it is already known what data the two applications share and how.

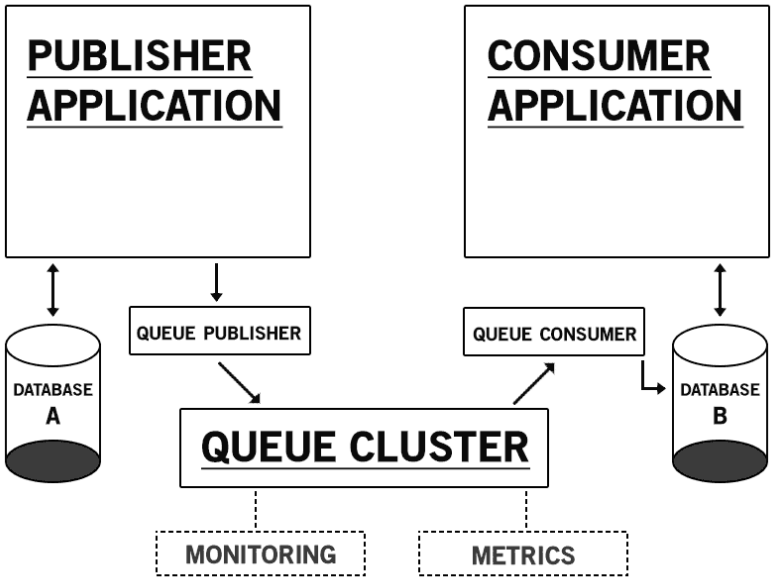


Figure 11. Future system architecture design with RabbitMQ

As seen in Figure 11, applications no longer have a direct connection and can thus be managed separately. Queue publisher and consumer are not part of the applications and can thus be implemented using more performant technologies. If user data traffic grows enough to slow down the operations, more queue consumers can be added to process the messages faster.

Figure 11 also portrays potential future additions enabled by the message broker solution. In order to make RabbitMQ resilient to failure, a cluster should be used instead of a standalone RabbitMQ node (Videla & Williams 2012, 89). This is important especially in production environments where data integrity is an absolute requirement. A Rabbit message queue also inherently enables monitoring and metrics through RabbitMQ's built-in management plugin.

5.3 Implementation

In the existing applications the publisher application initiates data synchronisation from a controller class coded using a PHP MVC framework. The data is received by a web API coded into a few classes on top of another PHP MVC framework. In the message queue solution, queue consumer replaces the web API completely.

Publishing logic could be moved outside the publishing application for example by using database triggers instead, but that would surface another set of complexity, reliability and integrity challenges. Publishing is therefore left here inside the existing application with POST query replaced by a queue connection.

To start working with RabbitMQ message queues, follow downloading and installation instructions on RabbitMQ's website. Once installed, open the installation directory into terminal for the next steps.

The commands used here in the examples are specific to Linux systems but they are very similar in OS X Windows environments, the difference being the way environment variables are inputted.

5.3.1 Clustering

Starting with a clustered solution differs little from single node solution, yet it reaps significant benefits in the future. To create a queue cluster, start a few queue nodes by commanding:

```
$ RABBITMQ_NODE_PORT=5672 RABBITMQ_NODENAME=rabbit \
  ./rabbitmq-server -detached

$ RABBITMQ_NODE_PORT=5673 RABBITMQ_NODENAME=rabbit_1 \
  ./rabbitmq-server -detached
```

Next stop and reset nodes to prepare them for clustering:

```
$ ./rabbitmqctl -n rabbit@myhostname stop_app
$ ./rabbitmqctl -n rabbit_1@myhostname stop_app

$ ./rabbitmqctl -n rabbit@myhostname reset
$ ./rabbitmqctl -n rabbit_1@myhostname reset
```

Now the nodes can be clustered:

```
$ ./sbin/rabbitmqctl -n rabbit_1@myhostname \
  join_cluster rabbit@myhostname
```

Noteworthy here is that the built-in clustering was not designed to be used over WAN – instead, shovel and federation plugins should be used for networked clustering.

5.3.2 Load balancing

Because RabbitMQ will distribute traffic evenly over all consumers using a round-robin method, loads are balanced by default (Videla & Williams 2012, 63). If a separate load balancer like HAProxy already exists in the system, it may make sense to centralise all load balancing efforts there. The configuration for load balancing a cluster with three nodes through HAProxy may look something like this:

```
global
    log 127.0.0.1 local0 info
    maxconn 4096
    stats socket /tmp/haproxy.socket uid haproxy mode 770 level admin
    daemon
defaults
    log global
    mode tcp
    option tcplog
    option dontlognull
    retries 3
    option redispatch
    maxconn 2000
    timeout connect 5s
    timeout client 120s
    timeout server 120s
listen rabbitmq_local_cluster 127.0.0.1:5670
    mode tcp
    balance roundrobin
    server rabbit 127.0.0.1:5672 check inter 5000 rise 2 fall 3
    server rabbit_1 127.0.0.1:5673 check inter 5000 rise 2 fall 3
    server rabbit_2 127.0.0.1:5674 check inter 5000 rise 2 fall 3
listen private_monitoring :8100
    mode http
    option httplog
    stats enable
    stats uri /stats
    stats refresh 5s
```

This imports cluster nodes into HAProxy which begins to balance load across nodes using the round-robin algorithm.

5.3.3 Monitoring

Queue traffic can be monitored easily through the built-in management plugin. To enable the plugin command:

```
$ rabbitmq-plugins enable rabbitmq_management
```

Next restart the server by commanding:

```
$ rabbitmq-service stop
$ rabbitmq-service install
$ rabbitmq-service start
```

Note that plugins such as the management plugin here may interfere with clustering or multiple simultaneous nodes in general if they use constant ports. Each new node will try to occupy the same port used by a plugin, which will cause the node not to be able to start up properly. The error message when attempting to set up the cluster when a node is unavailable does not convey the source of the problem very clearly.

5.3.4 Security

By default RabbitMQ comes with a guest user enabled. The user should be disabled and replaced by a user with more secure password:

```
$ ./rabbitmqctl add_user secure_user mySecurePassword!
```

Access list permissions should be applied for the new user account to allow it access to vhosts inside the broker:

```
$ ./rabbitmqctl set_permissions -p my_vhost \
  secure_user ".*" ".*" ".*"
```

This will give the secure user all privileges to configure, read and write in the realm of `my_vhost`.

RabbitMQ message transfers can be encrypted using SSL certificates. Assuming the certificate files already exist, RabbitMQ can be given the following configuration to enable listening for SSL connections on port 5671:

```
[
    {rabbit, [
        {ssl_listeners, [5671]},
        {ssl_options,
            [{cacertfile,"/path/to/rmqca/cacert.pem"},
            {certfile,"/path/to/server/cert.pem"},
            {keyfile,"/path/to/server/key.pem"},
            {verify,verify_peer},
            {fail_if_no_peer_cert,false}]}
     ]}
]
```

5.4 Other scenarios

The many features of RabbitMQ make it viable for other scenarios for payment service providers and businesses dealing with monetary transactions in general.

5.4.1 Task scheduling

Businesses with any larger systems often have tasks that need to be taken care of at regular intervals. One popular is scheduled tasking through cron jobs but they do not provide any error handling capabilities: if a task fails, the next task is none the wiser of its failure unless additional logic has been added to the cron job. A message queue can be used to make sure that each task is executed in the correct order and only after and if the previous task has finished.

5.4.2 Batch processing

As part of financial transactions many businesses have to do batch processing. In order to move money between business customers and consumers, payment service providers often opt to do calculate totals first and do settlements in batches in order to cut down transactional costs between financial institutions.

Processing thousands and thousands of entries synchronously on daily basis can be slow and tedious task. An error in the process halts progress for the whole settlement process. Instead of processing transactions through single application, the application could publish transaction data to a queue cluster that would crunch the data through smaller applications and pass it on to a financial transaction application that would consume the reports once all transactions have been handled.

6 RESULTS AND ANALYSIS

6.1 Messaging in production

Looking into the history of messaging it is safe to say that the concept is at a mature age. Open messaging standards are relatively new but they are being developed and trusted upon by an impressive congregation of vendors and large communities.

Understanding the paradigms behind messaging and knowing message broker's features help design and implement architectures that best suit operational needs at hand. The many benefits of message queues over other solutions not only validate but encourage their use. As shown in the implementation chapter of this thesis, restructuring system architecture to include messaging queues can be relatively simple given the original design has a clear data flow structure.

In addition to handling the obvious, message delivery, message brokers come packed with many features like routing and access control lists that help centralise business logics irrelevant to immediate business code from applications to the message broker. Features and benefits related to scaling, monitoring and security make messaging brokers enticing option for businesses requiring solid performance. As shown in the web API replacement scenario, RabbitMQ covers all of the previously mentioned features well.

Sometimes the choice of message broker is limited by pre-existing technological choices but often, like in the cases of RabbitMQ or Apache ActiveMQ, technologically inclined brokers may be used across endpoints that have been implemented in different technologies.

6.2 RabbitMQ from Paytrail's perspective

Based on both theoretical and practical study of messaging and RabbitMQ with AMQP in particular, RabbitMQ with AMQP can easily be recommended as the perfect

candidate for anyone to begin gradual transformation toward de-coupled systems. Should the choice need revisiting at a later time, changing to another message broker is simple due to the open messaging standard.

If it was well known from the beginning of a project that superior performance was required, ZeroMQ would be an obvious choice as evident from the dominance displayed in the measurements by Salvan (2013) and Hadlow (2011) (see Table 1 and Figure 9 respectively).

Weighing the strengths and weaknesses of both products and reflecting on the experiences during the web API replacement implementation, RabbitMQ's ease of use and reasonably good performance overcome the benefit of ZeroMQ's brute performance. Overall, RabbitMQ matches Paytrail's current purposes and use cases better.

REFERENCES

Broyer 2011. A Federation of Clouds. Blog article. Accessed on 4 May 2014. Retrieved from http://blog.venyu.com/2011/12/08/a-federation-of-clouds/

Cerf, V., Dalal, Y., Sunshine, C. 1974. SPECIFICATION OF INTERNET TRANSMISSION CONTROL PROGRAM. Accessed on 1 May 2014. Retrieved from http://tools.ietf.org/html/rfc675

Chatterjee, S. 2004. Messaging Patterns in Service-Oriented Architecture, Part 1. Tech article, Microsoft Developer Network. Accessed on 4 May 2014. Retrieved from http://msdn.microsoft.com/en-us/library/aa480027.aspx

DB-Engines Ranking of Key-Value Systems May 2014. Accessed on 11 May 2014. Retrieved from http://db-engines.com/en/ranking/key-value+store

Echlin, D. 12 July 2013. Stack Exchange - Designing an scalable message queue architecture. Forum post. Accessed on 11 May 2014. Retrieved from http://programmers.stackexchange.com/a/204626

Echlin, D. 15 July 2013. Stack Exchange - How to implement a message queue over Redis?. Forum post. Accessed on 11 May 2014. Retrieved from http://programmers.stackexchange.com/a/204922

Enable Consulting - Cloud Platforms & Integration n.d. Accessed on 5 May 2014. Retrieved from http://www.enableconsulting.com/#!cloud-platforms/c7zg

Foran, P. 2012. Top 10 Uses For A Message Queue. Blog article. Accessed on 1 May 2014. Retrieved from http://blog.iron.io/2012/12/top-10-uses-for-message-queue.html

Hadlow, M. 2011. Message Queue Shootout! Blog article. Accessed on 12 May 2014. Retrieved from http://mikehadlow.blogspot.fi/2011/04/message-queue-shootout.html

Hintjens, P. 2013. ØMQ - Welcome to ØMQ for AMQP users. n.d. Accessed 14 May 2014. Retrieved from http://zeromq.org/docs:welcome-from-amqp

Hintjens, P. 2013. ZeroMQ. California: O'Reilly Media, Inc.

Hintjens, P. 2014. ØMQ - The Guide. Accessed on 5 May 2014. Retrieved from http://zguide.zeromq.org/page:all#Chapter-Four-Reliable-Request-Reply

Janssen, C. n.d. Techopedia - Asynchronous Messaging. Online IT Dictionary definition. Accessed on 12 May 2014. Retrieved from http://www.techopedia.com/definition/26454/asynchronous-messaging

Maryka, S. 2009. What is the Asynchronous Web, and How is it Revolutionary?. Tech article. Accessed on 12 May 2014. Retrieved from

http://www.theserverside.com/news/1363576/What-is-the-Asynchronous-Web-and-How-is-it-Revolutionary

Pasker, B. 2008. "You Might Need Messaging If...". Presentation slideshow. Accessed on 4 May 2014. Retrieved from http://blog.pasker.net/2008/06/16/you-might-need-messaging-if/

Paytrail - Our Story n.d. Company information page. Accessed on 10 May 2014. Retrieved from http://www.paytrail.com/en/our-story

Piper, A. 2013. VMware vFabric Blog - Choosing Your Messaging Protocol: AMQP, MQTT, or STOMP. Blog article. Accessed on 6 May 2014. Retrieved from http://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html

Pronschinske, M. 2013. A Concise Comparison of RabbitMQ, ActiveMQ, and ZeroMQ Message Brokers. Tech article. Accessed on 13 May 2014. Retrieved from http://java.dzone.com/articles/concise-comparison-rabbitmq

RabbitMQ - Protocol Compatibility n.d. Technological documentation. Accessed on 12 May 2014. Retrieved from https://www.rabbitmq.com/specification.html

Reeder, T. 2012. Spikability - An Application's Ability to Handle Unknown and/or Inconsistent Load. Blog article. Accessed on 5 May 2014. Retrieved from Http://blog.iron.io/2012/06/spikability-applications-ability-to.html

Richardson, A. 2008. RabbitMQ: An Open Source Messaging Broker That Just Works. Video recorded presentation, Google Tech Talks. Accessed on 3 May 2014. Retrieved from https://www.youtube.com/watch?v=ZQogoEVXBSA

Richardson, A., Radestock, M., Garnock-Jones, T. 2008. Introduction to RabbitMQ - An open source message broker that just works. Presentation slides. Accessed on 4 May 2014. Retrieved from http://www.rabbitmq.com/resources/google-tech-talk-final/alexis-google-rabbitmq-talk.pdf

Rouse, M. 2005. Techtarget.com - Definition of 'message queueing' [sic]. Online IT dictionary definition. Accessed on 1 May 2014. Retrieved from http://searchsoa.techtarget.com/definition/message-queueing

Rotem-gal-oz, A. 2012. RabbitMQ and a Short Intro to AMQP. Tech article. Accessed on 12 May 2014. Retrieved from http://java.dzone.com/articles/rabbitmq-and-short-intro-amqp

Salvan, M. 2013. A quick message queue benchmark: ActiveMQ, RabbitMQ, HornetQ, QPID, Apollo. Blog article. Accessed on 12 May 2014. Http://blog.x-aeon.com/2013/04/10/a-quick-message-queue-benchmark-activemq-rabbitmq-hornetq-qpid-apollo/

Sústrik, M. 2012. The Merit of AMQP (part I). Blog article. Accessed 1 May 2014. Retrieved from http://250bpm.com/blog:11

Scaling the Message Queue Service 2010. Technological documentation. Oracle corporation. Accessed 11 May 2014. Retrieved from http://docs.oracle.com/cd/E19717-01/819-7759/aerbc/index.html

Stomp n.d. Product website front page. Accessed on 5 May 2014. Retrieved from http://stomp.github.io/

Van de Putte, G., Adinarayan, G., Haddon, R., McCarty, B., Peltomäki, M., Quixchan, O. 2005. Messaging Solutions in a Linux Environment. Accessed 1 May 2014. Retrieved from http://www.jamk.fi/kirjasto, Nelli portal, IBM Redbooks.

Videla, A., Williams. J.J.W. 2012. RabbitMQ in Action. New York: Manning Publications Co.

Vivek Ranadivé – Teknekron Software Systems 2014. Accessed on 5 May 2014. Retrieved from http://emereo.net/success/vivek-ranadive-teknekron-software-systems/