Lappeenranta University of Technology
Information Technology
Faculty of Technology Management

Bachelor's Thesis

Joona-Pekka Kokko

DISTRIBUTED DOCUMENT MIGRATION BETWEEN ERP SYSTEMS BY MEANS OF MESSAGING AND EVENT SOURCING

Examiner: Doctoral Student Tommi Kähkönen

Abstract

Lappeenranta University of Technology
Information Technology
Faculty of Technology Management

Joona-Pekka Kokko

Distributed document migration between ERP systems by means of messaging and event sourcing

Bachelor's Thesis
2013
34 pages, 9 figures, 6 tables, 1 appendices

Examiner: Doctoral Student Tommi Kähkönen

Keywords: document migration, messaging, event sourcing, eventing, publish/subscribe, masstransit, rabbitmq, mongodb

This bachelor's thesis, written for Lappeenranta University of Technology and implemented in a medium-sized enterprise (SME), examines a distributed document migration system. The system was created to migrate a large number of electronic documents, along with their metadata, from one document management system to another, so as to enable a rapid switchover of an enterprise resource planning systems inside the company. The paper examines, through theoretical analysis, messaging as a possible enabler of distributing applications and how it naturally fits an event based model, whereby system transitions and states are expressed through recorded behaviours. This is put into practice by analysing the implemented migration systems and how the core components, MassTransit, RabbitMQ and MongoDB, were orchestrated together to realize such a system. As a result, the paper presents an architecture for a scalable and distributed system that could migrate hundreds of thousands of documents over weekend, serving its goals in enabling a rapid system switchover.

Tiivistelmä

Lappeenrannan teknillinen yliopisto
Tuotantotalouden tiedekunta
Tietotekniikan koulutusohjelma

Joona-Pekka Kokko

Distributed document migration between ERP systems by means of messaging and event sourcing

Kanditaatintyö
2013
34 sivua, 9 kuvaa, 6 taulukkoa, 1 liite

Tarkastaja: Tohtoriopiskelija Tommi Kähkönen

Hakusanat: document migration, messaging, event sourcing, eventing, publish/subscribe, masstransit, rabbitmq, mongodb

Tämä Lappeenrannan teknillisen yliopiston kanditaatintyö analysoi keskisuurelle yritykselle toteutettua sähköisten dokumenttien migraatiojärjestelmää. Migraatiojärjestelmällä suuri määrä sähköisiä dokumentteja siirrettiin metatietoineen dokumentinhallintajärjestelmästä toiseen, osana yrityksen toiminnanohjausjärjestelmän vaihdosta. Tutkimus käsittelee ohjelmien hajauttamista migraatiojärjestelmän edellytyksenä: se analysoi viestinnän käyttämistä hajauttamiseen, sekä sen luonnollista yhteensopivuutta järjestelmän tilamuutosten tallentamiseen ja kuvaamiseen. Työ esittää analysoidun teorian perusteella rakennetun hajautetun migraatiojärjestelmän arkkitehtuurin, jonka keskeisiä komponentteja olivat MassTransit, RabbitMQ ja MongoDB. Työn lopputulos on hajautettu ja skaalautuva migraatiojärjestelmä, jolla viikonlopun aikana siirrettiin satojatuhansia dokumentteja ja siten mahdollistettiin toiminnanohjausjärjestelmän nopea vaihtaminen.

Contents

Symbols and Abbreviations

AMQP	Advanced Message Queuing Protocol
BSON	Binary JavaScript Object Notation
DDD	Domain Driven Design
DIP	Digital Image Processing
DMS	Document Management System
ESB	Enterprise Service Bus
EDMS	Electronic Document Management Systems
ERP	Enterprise Resource Planning
ES	Event Sourcing
MOM	Message Oriented Middleware
ORM	Object Relational Mapper
RDBMS	Relational Database Management System

1 Introduction

1.1 Background

This paper examines a software system that I created for a medium-sized Finnish company to migrate documents from one document management system (DMS) to another. The migration system was used at the end of the year 2011 as a part of Enterprise Resource Planning (ERP) system migration, whereby the company adopted a new ERP to better support its business processes. This change included switching from the separate DMS employed at the time to the DMS integrated in the new ERP system. This was done to gain integration benefits as well as mitigating licensing and maintenance costs of running two separate systems.

While the document migration was a prerequisite to adopting the new system, it also presented a challenges in scale and integration. The DMS of the time contained over 300000 documents that were to be migrated to the new system. This not only comprised the documents, but the metadata associated with them, as well as multiple versions of individual documents, as enabled by versioning support of the DMS. Meanwhile, the old and new DMS systems had no common surface or bilateral compatibility: they both used a different backend for storing files and different means of encoding and presenting document metadata.

Because of the reasons given above, the document migration was a non-trivial task, requiring a tailored migration system to mediate between the document management systems. In this, I set out to create a migration system that could easily be scaled up or down, so as to control throughput and speed of the migration, ultimately enabling document migration over weekend. This problem in turn seemed to favour distribution, where the migration workload could be split over multiple machines to be executed in parallel. Through theoretical studies, messaging proved to be a viable option to implementing this, as is examined in the following chapters.

This paper is structured to first explore the knowledge required in understanding the migration framework as well as the technological choices that I made. The paper begins by studying messaging as the enabler of distribution and decoupling. It then continues by looking at persistence strategies as well as domain modelling and how messaging comes together with event sourcing in this. The theoretical background is followed by analysis of the various components and technologies that were employed in the migration system: how MassTransit as distributed application framework enabled decoupled software components, how RabbitMQ as a message broker enabled distributed messaging and how MongoDB as a document database served as

persistent store in enabling event sourcing. After looking at the features and roles of the components, I show how they were used together in creating the migration system. This is followed by analysis of the key challenges I encountered and finally, conclusions.

1.2 Brief history to document management systems

The electrification of documents and services has brought an ever increasing need to efficiently manage electronic documents and records. While the first systems to assist in such a task were mere storages for electronic documents, produced through Document Image Processing (DIP), this set a path for Electronic Document Management Systems (EDMS). Rather than just document repositories, EDMS incorporated document management concepts, such as workflows and document versioning. EDMS integration has since been brought into applications, such as word processors, and is an integral part of Enterprise Content Management (ECM) frameworks that encompass, among other things, collaboration tools and business processes. (Adam, 2007)

With the emergence and development of ECM systems and EDMS therein, a variety of standards and specifications have been introduced to harmonize document management. This harmonization ranges from high-level frameworks, applicable both to physical and electronic documents, to implementation guidelines (Adam, 2007). As electronic document management has become an integral part in business process execution, various DMS systems have been created, either as standalone products or contained e.g. in enterprise resource planning software, to solve the problems of managing documents. As such, document management systems also create a technical dependency: changing from one DMS to another can be a non-trivial task that is practically solvable only through customized tools.

2 Decoupling and Distribution Through Messaging

This chapter looks at the fundamentals of messaging as means of decoupling and distributing computer programs into autonomous components. It lays out the theoretical foundation later required to understand the document migration system I implemented and especially the role and of MassTransit, a distributed application framework discussed in chapter 4.

2.1 Message based communication

Computers programs produce information by acting on input and state. In procedural programming, this equates to choosing and calling procedures based on the prevailing program state (Subramaniam & Hunt, 2006). Object-oriented programming on the other hand emphasizes encapsulation of state, thus making it the responsibility of the callee, instead of the caller, to hold the information required to make decisions. This act of invoking behaviour, known as method calls, was originally referred to as “message passing”, as it is known in Smalltalk (Subramaniam & Hunt, 2006). It can thus be said, that object-oriented systems expose their behaviour through messaging (Evans, 2003).

While these in-process method calls are fundamental to object-oriented programming, they couple together in space and time the objects involved in the execution of the program. Ultimately, the program flow becomes sequential, requiring objects to know who to call (coupling in space) and having to wait for the callees to respond (coupling in time). As programs grow in complexity, managing and understanding this sequential flow becomes ever harder. Instead of trying to manage a business process inside a single application, the application can be broken down into autonomous components. This, however, requires means of integration, so as to convey input and output between participants. One such method of integration is messaging, which, as defined by Hophe and Woolf (2003, p. 41) means to “Have each application connect to a common messaging system, and exchange data and invoke behavior using messages”. It should be noted that this differs from remote procedure calls (RPC), which on the surface appear as in-process calls, but are marshalled to a remote implementation, creating a time and space coupling between the caller and the callee (Chapell, 2004).

2.2 Messaging infrastructure

While there are numerous patterns and software systems to implement and facilitate messaging, certain typical features can be extracted from messaging systems. The enabler and provider of these features is called Message Oriented Middleware (MOM). (Chapell, 2004) Before studying these

features, it is however useful to lay out four key definitions central to MOM vocabulary, as defined by Hophe and Woolf:

- **Producers**, also known as senders, produce messages, which in turn are consumed by
- **Consumers**, also known as receivers, consume messages
- **Channels** transmit messages, ultimately connecting producers and consumers
- **Endpoints** connect channels to applications

Figure 1 shows a high-level overview of how two applications communicate in a message based system. In such a system, applications can be both, consumers and producers. The messaging API exposes endpoints to applications, through which they send and receive messages, transmitted in the channels.

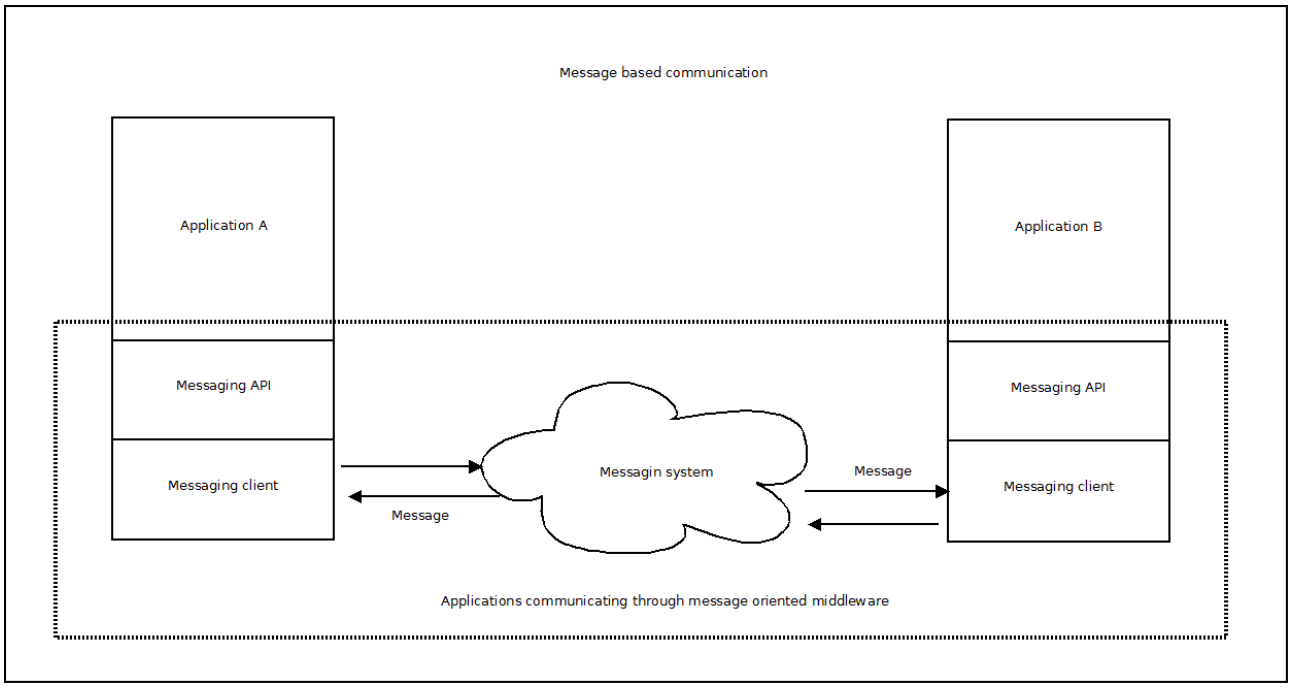

Figure 1 Message based communication.

While not all MOMs implement the same set of features, Table 1 lists common features in MOMs, along with their roles, as researched by Chapell (2004, p. 77 – 100). The software systems that implement and provide these features are commonly referred to as message brokers or message servers (Chapell, 2004).

Table 1 Features of Message Oriented Middlewar as defined by Chapell (2004).

Feature	**Role**	**Example**
Asynchronous messaging	Message senders and receivers are time and space decoupled, i.e. senders and receivers need	Sender places a message in a queue, from which a receiver later picks it up.

	not be aware of each other.	
Loosely coupled interfaces	Components are not dependent on the implementation details of others.	Messages alone convey the required information on which components can decide to act on. Counterexample would be RPC style invocation, where endpoints communicate directly, via endpoint specific interfaces.
Loosely coupled interaction	Components in a process flow only need to ensure they can reach MOM.	A component doing request/response messaging initiates a request, knowing it eventually receives a response through MOM.
Message autonomy	Messages are autonomous and self-contained entities. Once message is sent, MOM guarantees its delivery to any interested parties.	Similarly to a committed database transaction, a message reliably delivered to MOM can be considered safe.
Store and forward	MOM has delivery guarantee semantics, where messages can be delivered to unavailable receivers once they become available.	MOM can guarantee exactly-once delivery or at-least-once delivery.
Message persistence	Messages can reliably be persisted for later retrieval.	In case of crash, MOM can retrieve the non-processed messages from storage, resuming to the state prior the crash.
Message acknowledgment	MOM can determine when a message is successfully produced or consumed.	MOM can guarantee the delivery of messages in a request/response

		conversation.
Messaging models	MOMs can provide one-to-many messaging, as well as one-to-one messaging.	Producer sends a message to MOM, the message is later consumed by two, distinct consumers.

Whereas MOM can be seen as the backbone of messaging, Enterprise Service Bus (ESB) expands on the idea, enriching the feature set that MOMs build on, while increasing manageability as compared to MOMs. This includes a queryable service registry, through which services can be discovered. Key driver of this evolution has been Enterprise Application Integration. (Chapell, 2004) While the development and full capabilities of ESBs are not in the scope of this paper, routing as the central capability of ESBs (Erl, et al., 2014) will be more closely looked, as it is important for later understanding the document migration software. Key features of ESBs, as defined by Daigneau (2011), are shown in Table 2.

Table 2 Features of Enterprise Service Bus as defined by Daigneau (2011).

Feature	**Role**	**Example**
Advanced message routing	The bus in ESB provides full indirection over channels through which the clients connect. Clients communicate with the bus that routes messages between producers and consumers, as defined by a set of rules.	ESB can be configured to route messages based on their content and headers.
Message translation	Translating messages to a Canonical Data Model as they enter the bus and re-translating them as they reach recipients.	Producer sends an XML encoded message, that is first translated to the canonical data model of the bus and then to JSON so as to match the interface of the recipient.

Protocol translation and transport mapping	Mediate protocol transformations so as to hide differences in the client channels	.NET based client sends a message over HTTP that is then transformed to the appropriate format for a service accepting Java Message Service messages.

2.3 Messaging models and routing in the bus

A bus provides a standard way for consumers and producers to plug into the messaging system, so as to post and receive data. Pertaining to message autonomy, once messages enter the bus they carry enough information to determine their flow through the messaging system. Two such, fundamental flows are publish-and-subscribe messaging, also known as pub/sub, and point-to-point messaging. Whereas point-to-point model is used to deliver messages from one producer to one consumer, pub/sub allows for consumers to register their interest in messages. Thus, in publish-and-subscribe messaging, a message sent by producer is copied and delivered to any interested consumers. (Chapell, 2004) Finally, it should be noted that messages in the pub/sub system are often referred to as events, meaning they signal "a discrete state transition that has occurred", as defined by Tarkoma (2012).

To allow for the one-to-many messaging in pub/sub, consumers must have a way of registering their interest in messages. This functionality is realized in routing, which implements a scheme to make delivery decisions. While routing itself is a broad and advanced topic, it can be divided into five general categories, as detailed in Table 3. (Tarkoma, 2012) Of the presented five routing schemes, type based routing, as implemented in MassTransit, remains of interest to this paper.

Table 3 Routing scheme, as categorized by Tarkoma (2012).

Routing scheme	**Description**
Channel/topic-based routing	A named channel, known as topic, is used to determine recipients. Subscribers register their interest in topics. This can include support for hierarchical topics, where recipients register their interest to certain level of the topic hierarchy. For example, a consumer might register interest in the topic "Buyorder.#.Equities", which would satisfy both, the topic "Buyorder.Foreign.Equities and the topic "Buyorder.Domestic.Equities", # acting as a wildcard.
Subject-based routing	Similar to topic-based routing, but the routing decision is made based on the subject of the message, instead of the channel to which the message was published.
Type-based routing	Producers and consumers share a message type hierarchy, which is used to determine recipients.
Header-based routing	Messages are enriched with headers that contain information used to determine recipients.
Content-based routing	Delivery decisions are made by inspecting the body of the message.

3 Presenting and Persisting System State

This chapter deals with the notion of system state and how it can be persisted as a series of messages. This serves as a prerequisite to understanding the persistence model that I used in the migration system to keep track of individual documents and their versions.

3.1 Structural data models

Unless programs hold all their input and output in memory, they must at some point persist their state to a persistent media, e.g. a hard drive. This not only requires the persistent medium, but also a model according to which the data can be stored and later retrieved. A common way of doing so, is to have a structural data model, where relationships between pieces of datum are normalized, after which the data can be written out into a relational database as rows and columns. Loading back the data then becomes an act of finding the connected data and transforming it back into an in-memory representation. In object-oriented programming, this task can be handled by an Object Relational Mapper (ORM), which manage the act of converting data between its object representation and structural model (Daigneau, 2011).

To demonstrate object persistence and structural data models, Figure 1 below presents an imaginary and trivialized case of a shopping cart, with an in-memory instance of a specific cart (above) and its class diagram (below).

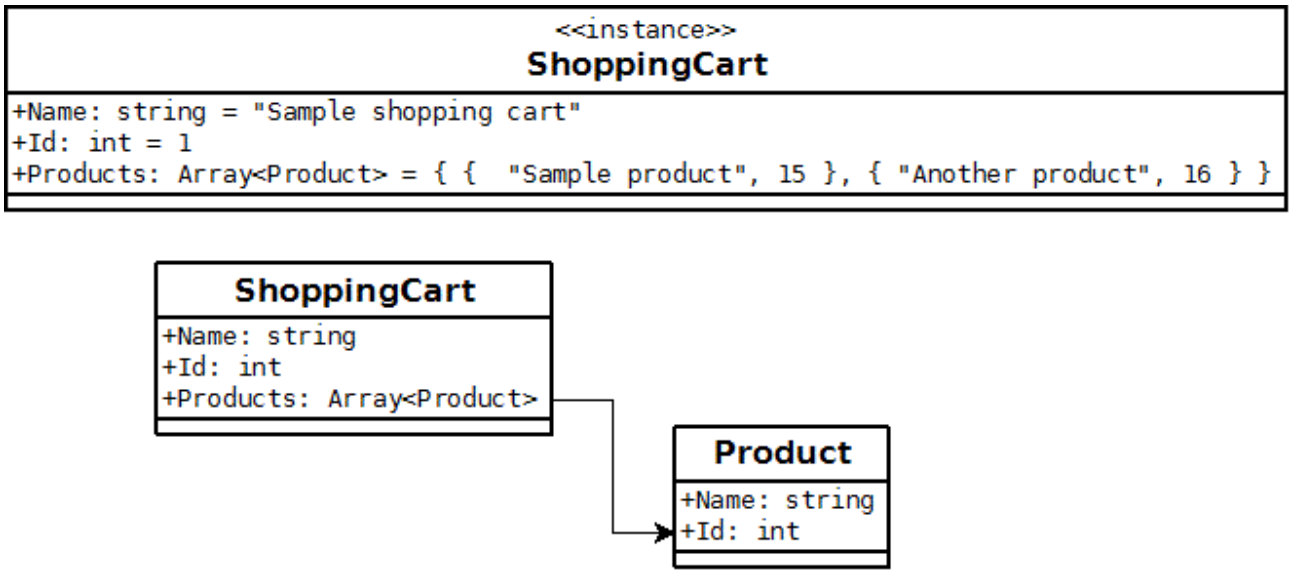

Figure 2 Shopping cart instance and its class diagram.

To store such an item in a relational database, Figure 3 below depicts a normalized, relational model, where the shopping cart data is split amongst three different tables.

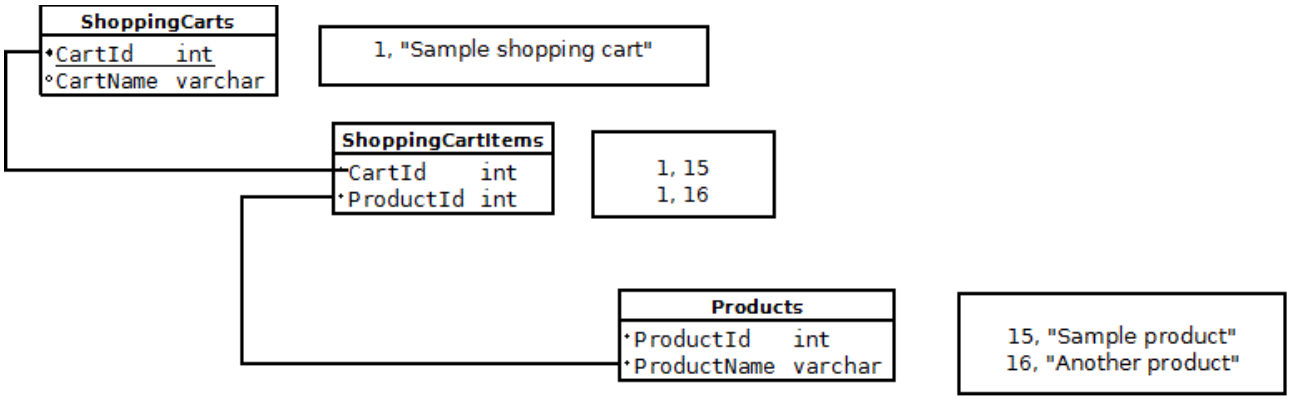

Figure 3 Structural data model of a shopping cart

3.2 Events as persistence model

Another, alternative persistence model to structural data modelling is Event Sourcing (ES), as applied and made popular in Domain Driven Design (DDD) by Greg Young (Vernon, 2013). Event sourcing makes use of events, which, as described in chapter 2, capture transitions in the program state. Whereas object-relational mappers hydrate objects from database records, Event Sourcing uses an ordered sequence of events, recorded since certain point of time, to hydrate an object back to its certain, observed state in time. Thus, restoring state becomes an act of replaying history (Vernon, 2013).

The shopping cart example of chapter 3.1 can also be modelled and persisted using events, given that two state transition messages, i.e. events, are introduced:

- Event “ShoppingCartCreated” that records the name of the cart
- Event “ProductAdded” that records a product in the cart

By recording such events for a specific shopping cart, the act of persisting the in-memory object then means persisting the events themselves. Rehydrating the shopping cart from storage then becomes the act of replaying the events. In other words, the shopping cart is a projection, calculated from the replayed events. Figure 4 below demonstrates how a shopping cart, similar to that in example above, is projected from three persisted events.

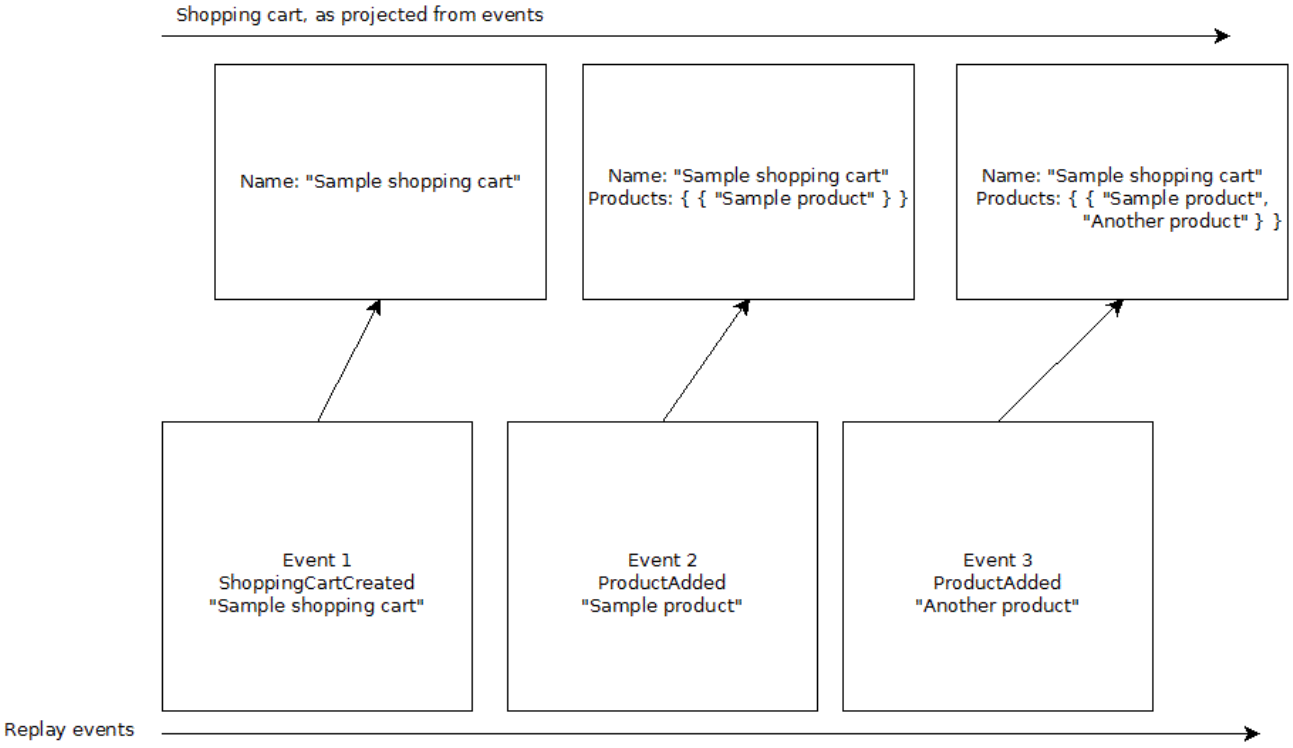

Figure 4 Projecting shopping cart from stored events

3.3 From storing structure to storing behaviour

Based on the elementary analysis above, it can be concluded that event sourcing as a data and persistence model is vastly different from structural data modelling: whereas the former focuses on relationships and structural representation of data, the latter ultimately stores behaviour, i.e. messages that record state changes. This fundamental difference comes with several implications that can best be clarified through an example: if the shopping cart in the structural data model were to be updated by removing a product, then the knowledge of the removed item ever being in the cart would be lost. In contrast, an event sourced model would record the removal as a "ProductRemoved" event. While this inherent feature to maintain history can be valuable in itself, it also affects the usage of the persistent store: as no data needs to be deleted, an event sourced data model can be append-only, which in turn removes the complexities of update and delete operations. For these reason, there are also specific storage solutions, called Event Stores, for managing event data (Vernon, 2013).

4 Building a Distributed Document Migration System

This chapter, while relying on the theoretical background explored in the previous chapters, examines how messaging was applied to build a distributed, loosely coupled document migration system. The chapter first looks at the individual components that were used in the migration system and how they implement and serve their roles. In this, the focus is set on MassTransit, a distributed application framework that formed the integral part in enabling the use of messaging in the project. Furthermore, MassTransit has not been analysed in scientific studies before, giving value to analysis of its features in implementation. After MassTransit, the chapter looks at RabbitMQ, the message broker that was paired with MassTransit in the migration system. This is followed by analysis of an event store implementation, then MongoDB as the persistence backend. Lastly, the chapter puts together the various components in building and enabling a distributed document migration system.

4.1 Messaging with MassTransit

MassTransit, which served as service bus in the migration system, is an open source service bus implementation, started by Chris Patterson and Dru Sellers in the year 2007, to enable distributed messaging in the .NET ecosystem (What is MassTransit?, 2013). For the purpose of this paper, I analysed the version 2.0.1 code base of MassTransit, as it was used in the document migration system. This also applies to the documentation, which was built, using Sphinx, for the version 2.0.1. Finally, the analysis is contained to the MassTransit main assembly and the assembly implementing RabbitMQ integration, leaving out e.g. unit and integration tests, inversion of control container integration and the Microsoft Message Queuing (MSMQ) transport implementation. The source code, documentation included, is publicly available from GitHub git repository of MassTransit (MassTransit commit 29e8cf9058e9833203e5ce61f211a00064b489d4, 2013). Details of this chapter assume elementary knowledge of the .NET infrastructure and C# language.

The core of MassTransit, contained in the assembly MassTransit, implements a service bus that allows for sending and receiving of messages, effectively supporting the point-to-point and publish-and-subscribe message models introduced in chapter 2. While implementing a service bus, MassTransit requires a Message Oriented Middleware to distribute messages, implementing only a loopback channel itself. Thus, the MOM specific code, namely MSMQ and RabbitMQ support, is abstracted away and contained in separate assemblies. This can be seen from Figure 1 that visualizes the dependency graph of MassTransit. Ultimately, the dependency graph shows that the core of MassTransit has few dependencies, most of which are .NET base class libraries. In addition,

it uses Magnum for utility classes and a reflective visitor pattern implementation, Stact for actor implementation, Newtonsoft.JSON for JSON serialization and log4net for logging. Finally, MassTransit.Transports.RabbitMq has a dependency on the RabbitMQ AMQP client implementation.

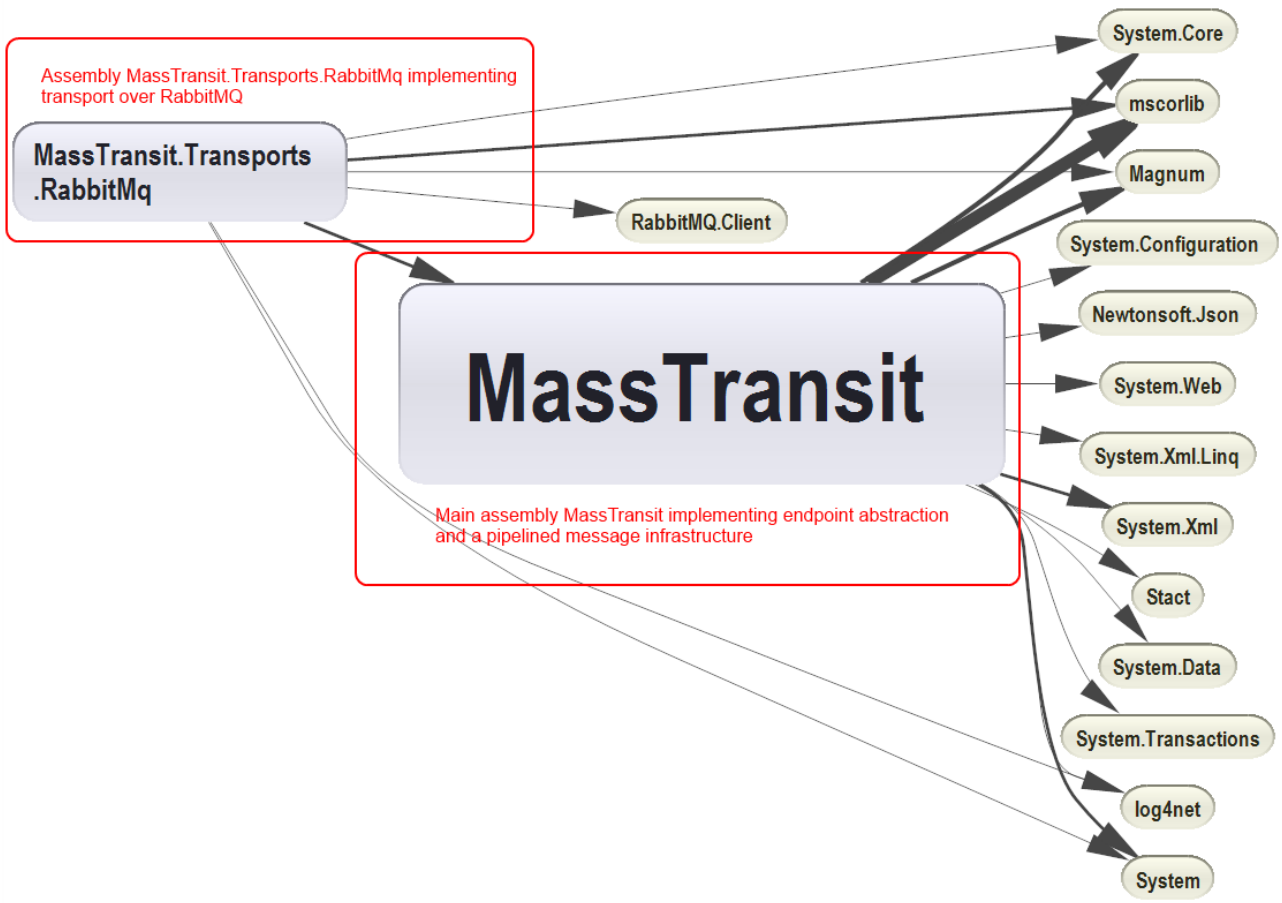

Figure 5 Annotated MassTransit dependency graph, as obtained by NDepend.

In abstracting away the message oriented middleware, MassTransit uses a concept of pipelines, through which messages, wrapped in context, flow. While the pipelines are concerned with in-process flow of messages, they provide means for message interception, which serves as an extension point to the system. Finally, MassTransit hides the implementation details of endpoints and message channels through use of the interface ITransport, which provides the surface for receiving and sending out-of-process messages.

As the core implementation of MassTransit is MOM agnostic, MassTransit makes heavy use of extension methods and configuration callbacks to wire up the runtime behaviour of the service bus. A concrete example of this is the static class ServiceBusFactory that provides a shorthand for instantiating a service bus, as configured through the ServiceBusConfigurator callback, passed in System.Action parameter. Once the service bus is started, it repeatedly queues, through the use of a thread pool, a receive call on the transport. When endpoint receives a message, it gets deserialized

(to an envelope format), wrapped in a ReceiveContext class and passed down to the inbound pipeline of the service bus. The inbound pipeline may then have zero or more interceptors that provide consumers for the message. Publishing a message works in a similar fashion, whereby the message, first wrapped in PublishContext class, is passed down the outbound pipeline of the service bus. The outbound pipeline may then have attached behaviour that e.g. further publishes the message to a broker.

Finally, to make use of MassTransit, it is important to understand its routing model, which adheres to type-based routing, first examined in chapter 2.2. As a .NET product, MassTransit uses the .NET type system to allow for the definition of a message hierarchy. Thus, subscribers express interest in messages by their types. Pertaining to message autonomy, explained in chapter 2.1, messages must also carry this type information once published, to allow for their processing at a later time. In practice, MassTransit moves around messages wrapped in an Envelope class that encapsulates message types in MessageUrn class, which, in turn, builds a string representation from the full name of the .NET type. This is illustrated in Figure 6, where a subscriber subscribes to messages of type "MessageContracts.SampleMessage". Once such message is published, it is routed to the subscriber.

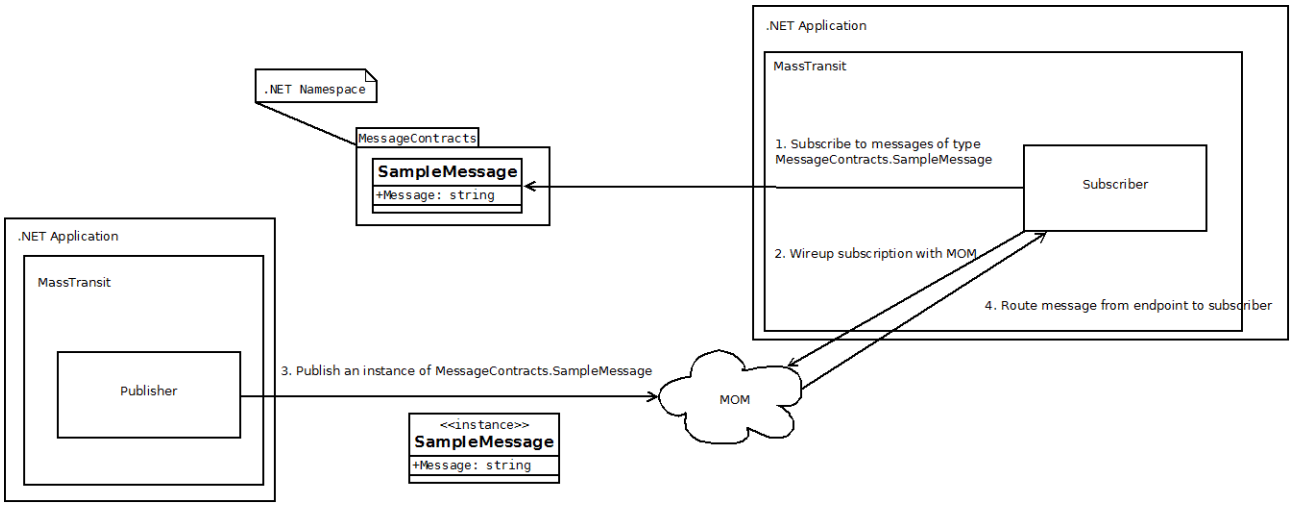

Figure 6 Pub/sub and type-based routing in MassTransit.

4.2 MassTransit with RabbitMQ

As examined in chapter 4.1, the core of MassTransit is MOM agnostic. To allow for out-of-process messaging, MassTransit needs to be wired up with a message oriented middleware. Out of the box, it supports Microsoft Message Queuing (MSMQ) and RabbitMQ. This chapter looks at core details of RabbitMQ, which I used in the migration system to serve as a messaging backend.

RabbitMQ is an Erlang-based, open source message broker that implements the Advanced Message Queuing Protocol (AMQP) open standard and supports both, one-to-one and one-to-many

messaging. In implementing AMQP, RabbitMQ builds around the concept of exchanges and queues, where the former receive messages, routing them to the latter, to which consumers bind. The routing decisions are affected by the type of the exchange, whereby each of the four available exchange types implements a different routing scheme (Videla & Williams, 2012). Of the implemented exchange types, the fanout exchange is of interest to this paper, as it is used by MassTransit, when paired with RabbitMQ. In effect, a fanout exchange multicasts all the messages it receives to all the queues that have been bound to the exchange (Videla & Williams, 2012).

As examined in the chapter 4.1, MassTransit relies on .NET type information to make routing decisions. When paired with RabbitMQ, MassTransit does this by creating RabbitMQ exchanges named after the .NET types. Messages can then be addressed to a single exchange, to which the consumers in turn are bound. In this way, MassTransit uses the underlying topic-based routing of RabbitMQ to implement the type-based routing visible to its client surface. I have illustrated this process of exchange declaration, queue binding and publish/subscribe messaging in Figure 1. It's noteworthy that when configured to use the routing capabilities of RabbitMQ, MassTransit manages exchange declarations and queue bindings, using its inbound and outbound pipelines to intercept messages so as to ensure that they have the respective exchanges and queues.

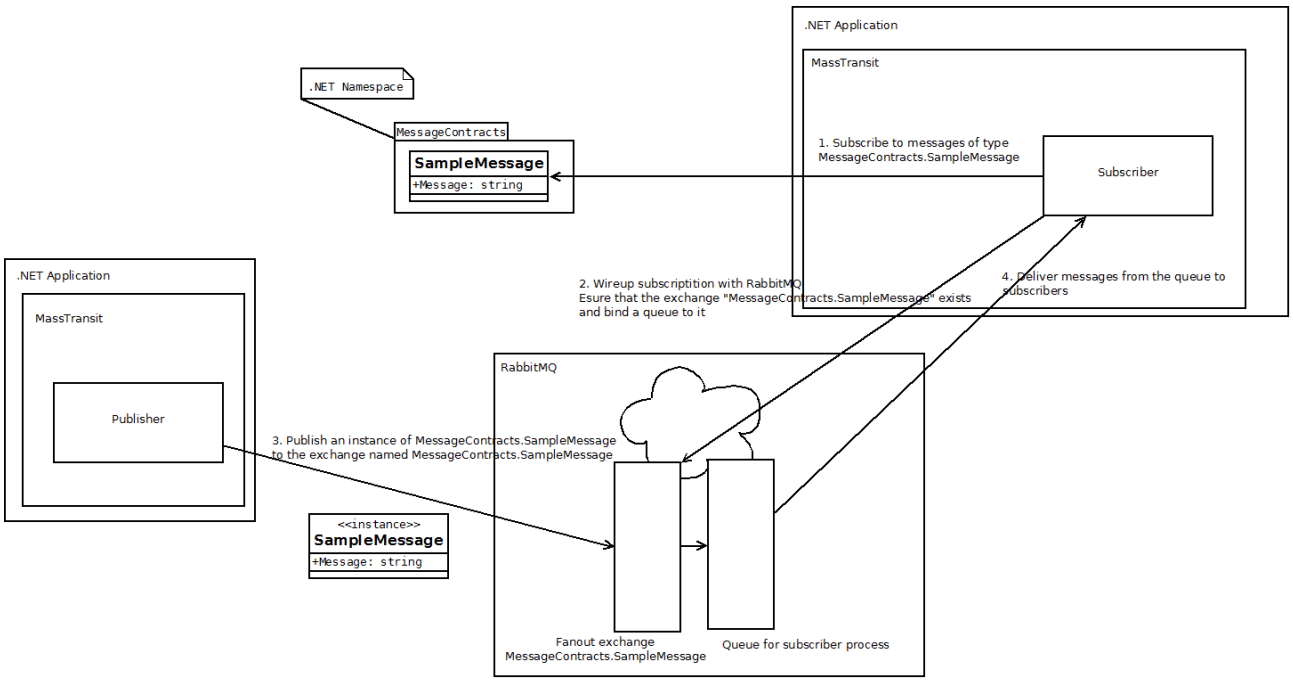

Figure 7 Using MassTransit with RabbitMQ.

4.3 Event sourcing and event store

As looked in chapter 3, events can be used to record and signal behaviours that manipulate program state. As such, they can be used to project and persist the state of a program in time. Event store, the component implementing fetching and persisting of events, for the migration system was built using

the .NET based *Simple CQRS example*, written in C# by Greg Young (Young, 2013). The example, as is available from its GitHub Git repository, demonstrates the Command Query Responsibility (CQRS) pattern and central concepts of Domain Driven Design (DDD). It contains a trivial in-memory event store that supports loading and storing of events whereby a set of related events is uniquely identified by a .NET System.Guid. This, in turn, is used by a repository to store and populate aggregates that implement the abstract base class AggregateRoot. As such, aggregates are uniquely identified by System.Guid and are capable of replaying their state from events, as well as providing the set of events that have been applied to them. Figure 8 below lists the central interfaces and classes of the Simple CQRS example.

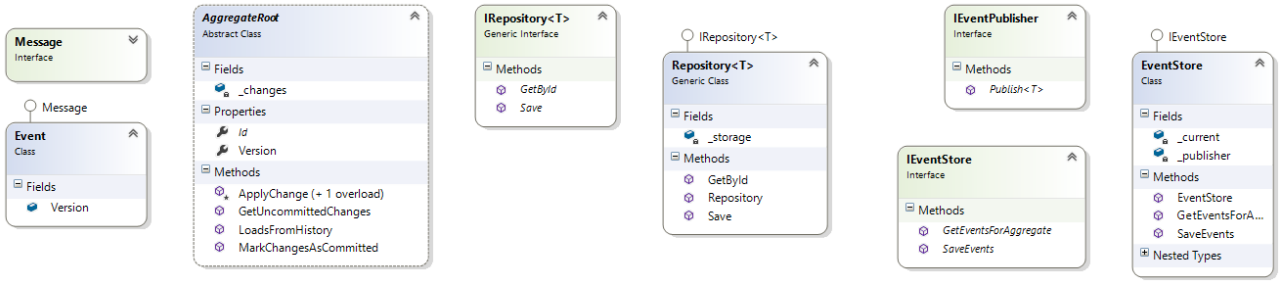

Figure 8 Central interfaces and classes in the Simple CQRS example, as drawn by Visual Studio 2012.

To understand how the classes, presented in Figure 8, play together, the listings below shows a high-level overview of the actions resulting from creating an aggregate, modifying its state and finally persisting and reloading it entirely. The listing assumes that any dependencies on interfaces are implemented and supplied by the concrete classes visible in Figure 8.

Creating, modifying and persisting an aggregate:

1. An aggregate, inheriting from AggregateRoot, is instantiated and given a unique identifier
2. A behaviour is applied on the aggregate, changing its state
    1. The aggregate instantiates an event, inheriting from Event, that captures the state change
    2. The aggregate applies the event to itself, i.e. invokes behaviour associated with the event
    3. The aggregate records the event in its change history
3. The aggregate is passed on to a generic, typed repository implementing the IRepository interface, via the Save method
    1. The repository asks the aggregate for its change history, i.e. the events that it has recorded
    2. The repository passes the events, along with the unique identifier of the aggregate to the IEventStore implementation that it holds by calling SaveEvents on it
        1. EventStore wraps the events with the unique identifier of the aggregate

2. EventStore stores the wrapped events in memory
3. EventStore publishes the wrapped events to any interested subscribers via the IEventPublisher interface

Loading the previously saved aggregate from event store:

1. GetById is called on a typed, generic implementation of IRepository and the unique identifier of the aggregate from previous listing is passed as a parameter
    1. The repository creates a new instance of the aggregate type
    2. The repository delegates to its IEventStore implementation, calling its GetEventsForAggregate and supplies the given unique identifier
        1. EventStore fetches the events associated with the identifier and then unwraps and returns them
    3. The repository calls LoadsFromHistory on the newly constructed aggregate, supplying the events received from IEventStore
        1. The aggregate applies all the events to itself, invoking the behaviours that are associated with each event

Once the loading is complete, the aggregate has replayed its behavioural history, i.e. events, and recovered to a known state. It is important to note that once events are replayed in the aggregate, they are not recorded in its change history, as it would end up duplicating the behavioural history once the aggregate gets saved again.

4.4 Persisting events to MongoDB

While chapter 4.3 demonstrated an in-memory event store, it is not sufficient for running distributed components that need the same source of truth as well as recoverability. For this reason, I adapted the previously described event store to store events in a database, namely MongoDB. MongoDB, written in C++, is an open source document database, exhibiting features commonly associated with the NoSQL movement (Banker, 2011). Unlike traditional relational databases, MongoDB has no enforced schema and instead of columns, rows and tables models data as documents, stored and transmitted as Binary JavaScript Object Notation (BSON). As such, it has traditional database features like indexing, but instead of SQL like query language, has a language fit for documents. (Banker, 2011)

As a document store, MongoDB treats documents as autonomous entities – while documents can have references to other documents, MongoDB has no notion of joins as known in the relational database management systems (RDBMS). (Banker, 2011) This, however has a natural fit with events, which, as discussed in chapter 2.1, are also autonomous entities. Because of this, events can

be modelled one-to-one with documents, i.e. a single event is a single document. This benefit is further enhanced because of the schemaless nature of MongoDB: events, while themselves conform to a schema, can be stored without normalizing them to an explicit schema in the database. Thus, the complexities of object-relational mapping, explored in chapter 3.1, can be avoided. Ultimately, the act of persisting events only required to storing of the wrapped event, as described in the previous chapter, as a document in MongoDB, from which it could be fetched back by using the unique identifier of the aggregate.

4.5 Bringing together the migration system

Given the nature and incompatibilities of the source and target systems of the migration, data could not directly be transferred between the systems. Because of the metadata associated with documents, data structures internal to the systems and the different storage mechanisms, the migration was neither a trivial batching of file system operations. Lastly, given the large number of documents and document versions, the goal was set to create migration system that could easily be scaled and distributed, so as to avoid bottlenecking resources of a single process or computer. By applying the theory presented in chapters 2 and 3, I chose the components suitable to implementing the explored architectural styles and patterns, namely messaging and event sourcing. These components, together with their roles, are summarized in Table 4 below.

Table 4 Components of the migration system.

Component	**Role**	**Setup**
MassTransit	Enable publish/subscribe messaging, where producers publish messages, to which subscribers subscribe, using type-based routing.	MassTransit 2.0.1, transport over RabbitMQ 2.6.1.
RabbitMQ	Act as MOM for MassTransit, providing message channels through exchanges and queues	RabbitMQ 2.6.1 running on Erlang R14B04.
Event Store, as adapted from *Simple CQRS Example*	Provide implementation basis for aggregates, repositories and event store to store events passing through the system, so as to capture and later project the system state.	Event persistence in MongoDB with numeric aggregate identifiers
MongoDB	Persist events to disk as documents.	Replicated MongoDB 1.8.2 with master-slave-arbiter setup

In choosing the components, the first criteria was to estimate their suitability out-of-the-box, i.e. how well they could fulfil their role without customization. To supplement this criteria, I wanted to use open source software, with permissive enough license to allow for commercial use, so as to allow for full transparency and customizability while ensuring usage free of charge. The following table, Table 5, summarizes the technology choices I made.

Table 5 Component choices in the migration system.

The component chosen	**Alternative components**	**Reasoning**
MassTransit	NServiceBus (Particular Software, 2013)	While both .NET products, supporting the creation of distributed applications, the free version of NServiceBus had severe performance limitations. NServiceBus at the time only supported MSMQ, whereas MassTransit had support for RabbitMQ.
RabbitMQ	MSMQ	MSMQ would have to be installed on each machine running the migration system, whereas RabbitMQ is a centralized broker, requiring only the client library to be copied with the product harnessing its power. RabbitMQ supports pub/sub and allows for competing consumers, where multiple consumers read from the same queue, whose alternative with MassTransit and MSMQ would have required the use of MassTransit distributor pattern, further complicating the implementation.

Simple CQRS Example	EventStore by Jonathan Oliver (Oliver, 2013)	While both .NET (C#) code, Simple CQRS Example had significantly smaller code base than Jonathan Oliver's event store, as well as trivial aggregate support, making it easier to understand and refactor. While both libraries identify aggregates by System.Guid, Simple CQRS Example, due to its smaller surface, was easier to refactor to use identifiers of other type.

MongoDB	Sql Server 2008	While Sql Server was already deployed at the premise, MongoDB had the attraction of storing documents, which, as shown in the previous chapter, are a natural fit for events, thus eliminating the need for ORMs and customized serialization strategies (e.g. serializing events to XML in the database and then deserializing them to their object representation). As the migration was going to stress the SQL Server, harnessed by the new DMS, MongoDB helped to distribute load away from the new document storage.

By analysing the problem domain, I could identify a clear case for producer/consumer pattern, where a producer would extract information from the old system to then be given to consumer to process. In other words, a single message would correspond to the intent to migrate a single, versioned document from the old document management system. By this notion, such a message can be called command: unlike events that signify a state change, commands express intent to invoke a behaviour and as such, can also be rejected. More generally, a command is a serialized function call (Evans, 2003). Yet, given the large number of documents, processing such commands sequentially would have bottlenecked a consumer. To work around this, I adopted the competing consumers pattern, where a producer uses a point-to-point channel to publish messages to multiple consumers so that each message is given to a single consumer for processing (Hohpe & Woolf, 2003).

MassTransit, when paired with RabbitMQ, supports competing consumers out-of-the-box, allowing for the distribution of consumers. Furthermore, as MassTransit uses a threaded consumer model, consumers could be configured to consume multiple messages at a time. Relying on this, I built a

single producer that extracted information from the old document management system, harvesting information of the documents that would need to be migrated. The producer then built a command for each document, signifying the intent of its migration, and sent the command to a consumer endpoint. To consume these commands, I built a command handler, which subscribed to the type of the migration commands. Once the handler received a command, containing information for a document to be migrated, it set out to populate an aggregate corresponding to the document. As the documents were all identified by a unique number, it was also a natural choice for the aggregate identifier. To enable this, I modified the event store, based on the Simple CQRS Example, to use numeric identifiers. By then examining the document aggregate, the command handler could then decide whether the document, along with all its version, had been migrated. For non-migrated documents, the handler would proceed to download the document from the old DMS and store it, along with its metadata, in the new DMS. After successfully doing so, the handler would then modify the state of the aggregate, i.e. produce events, and persist it in the repository, ultimately storing the events in MongoDB, through the event store, as well as publishing them through MassTransit. This described migration system is illustrated in Figure 9.

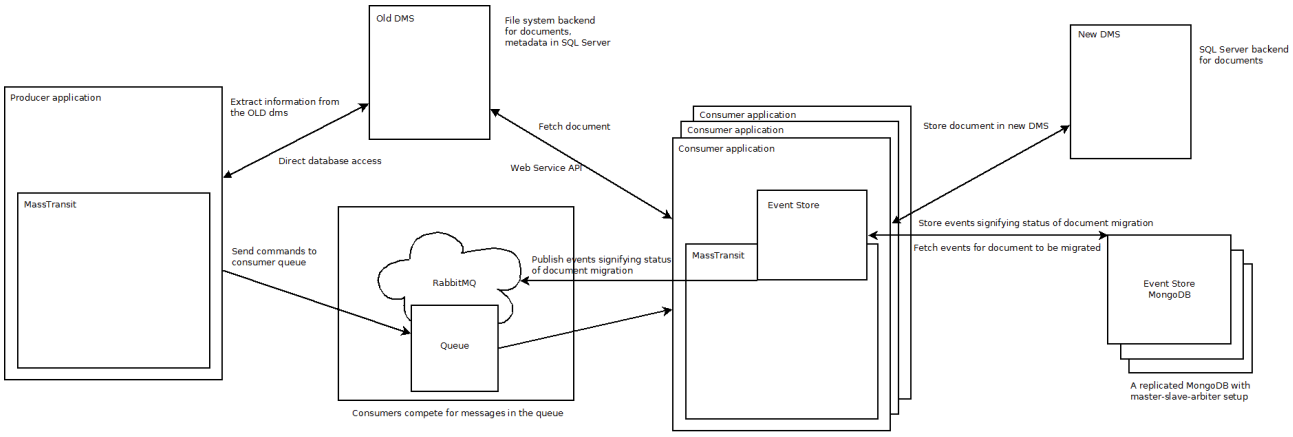

Figure 9 Components of the migration system (see appendix Components of the migration system for larger image).

With the producer and consumer components implemented, completing the migration then became a matter of running the producer and a number of consumers, distributed across machines, until no migration events were published, signifying that all the documents had been migrated. By using MassTransit, observing this event flow from the migration only required subscribing to the types of events created by the document aggregate (e.g. “ExportedNode”, signalling successfully migrated document) and projecting them to a dashboard-like view. In case of failures in the command handlers themselves, the consumers would move the command to an error queue, a form of Invalid Message Channel (Hohpe & Woolf, 2003) and behaviour provided by MassTransit, from which they could later be moved back to the consumer queue, after examining the case of failure and

possibly applying a fix for it. Finally, the production migration was run over a weekend, migrating over 300000 documents and producing over one million events

4.6 On suitability and problems of the migration system

The migration could be run over a weekend, signifying that the strategy to distribute the system was successful, as sequential processing of the documents would have taken longer. This, in turn allowed for the immediate adoption of the new ERP system following the migration weekend. Some architectural decisions in the migration system can however be questioned, as they have far-reaching impacts.

The first far-reaching impact results from the use of event sourcing to capture system state. Because of this, aggregates cannot be queried by their state, as the state is ultimately a projection over a stream of events. Thus, queries such as select document aggregates that were migrated between certain dates are not possible. This can, however, be mitigated through the use of read models, which are an integral part of Command Query Responsibility Segregation. By subscribing to the events that flow in the messaging system, a consumer can build and update projections to capture various states of the system. Whereas commands then are issued to command handlers, queries are executed against read models (Evans, 2003). As the migration system used no persistent read models, the migration had to be run until no new events, signalling document migrations, were produced.

The second far-reaching, as well as somewhat complex problem was to operate without distributed transactions, as MassTransit, RabbitMQ and MongoDB could not be fit inside the same transaction boundary. Because of this, a document could, for example, be successfully migrated, but persisting the events to signify that might fail. This, in turn could result in duplicate documents being stored in the new DMS, as the state of the document aggregate, built from events, would not be in sync with reality. Besides using technologies where all the components can participate in a distributed transaction, a common solution, especially with at-least-once delivery, is to use idempotent message handlers (Tarkoma, 2012). This guarantees that repeated delivery of the same message does not change the state of the system. The migration system did not adhere to message idempotency, as there was no actual risk to having a duplicate document.

The presented design decisions, along with their impacts are summarised in Table 6 below.

Table 6 Significant design decisions, along with their impacts and alternatives.

Design decision	**Impacts**	**Alternative design**
System state through event sourcing	Aggregates cannot be queried by state from the event stream.	Maintain separate read models
No distributed transactions	The components mutating the system state cannot be fit inside the same transaction boundary	Adhere to message idempotency

5 Conclusion

Messaging is a viable architectural style to implementing software systems that need to be distributed over multiple processes and machines. While messaging can be implemented in many forms, its ultimate value comes from decoupling software components in space and time. This was shown to be true with migration system that I created: it harnessed the messaging pattern of competing consumers to distribute the task of migrating documents from one system to another across multiple machines. Event sourcing, used to load and persist the system sate, was shown to be a good companion with messaging, where state changes could be modelled as events and then used as behavioural history to project the state of the system in time. Finally, in implementing the event store and event persistence, a document database proved to be a natural fit for events, which could then be modelled as autonomous, individual documents.

In exploring the components to satisfy the different roles found in messaging and event sourcing, MassTransit was shown to be a fit backbone for enabling the distribution of applications in the .NET space. While MOM agnostic, it could be paired with RabbitMQ, a message broker that enabled, amongst other things, publish/subscribe and competing/consumers styles of messaging. Lastly, in combining messaging with event sourcing, MongoDB allowed for the persistence of events as individual documents, freeing the system from the burden of explicit database schema design or object-relational mapping. With this understanding of messaging and event sourcing, I could put together a migration system that achieved its goals, i.e. migrated the documents from a decommissioned document management system to a new document management system, while allowing for the distribution of the migration load so as to run the migration over weekend, enabling instant adaptation of the new system.

6 References

Adam, A. (2007). *Implementing Electronic Document and Record Management Systems.* Auerbach Publications.

Banker, K. (2011). *MongoDB in Action.* Manning Publications.

Chapell, D. (2004). *Enterprise Service Bus.* O'Reilly Media.

Daigneau, R. (2011). *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services.* Addison-Wesley Professional.

Erl, T., Little, M., Arnaud, S., Rischbeck, T., Assi, A., Chapell, D., . . . Liu, A. (2014). *Service-Oriented Infrastructure: On-Premise and in the Cloud.* Prentice Hall.

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional.

Hohpe, G.;& Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.*

MassTransit commit 29e8cf9058e9833203e5ce61f211a00064b489d4. (2013, 8 19). Retrieved from MassTransit: https://github.com/MassTransit/MassTransit/commit/29e8cf9058e9833203e5ce61f211a00064b489d4

Oliver, J. (2013, 8 21). *EventStore v3.0*. Retrieved from GitHub: https://github.com/NEventStore/NEventStore/tree/a339412da008dd18e862c0f1e33009923c297e8c

Particular Software. (2013, 8 21). *NServiceBus*. Retrieved from GitHub: https://github.com/Particular/NServiceBus/tree/5621b1df9dc97769e689fdc898d613ec29d24985

Subramaniam, V.;& Hunt, A. (2006). *Practices of an Agile Developer: Working in the Real World.* Pragmatic Bookshelf.

Tarkoma, S. (2012). *Publish / Subscribe Systems: Design and Principles.* Wiley.

Vernon, V. (2013). *Implementing Domain-Driven Design.* Addison-Wesley Professional.

What is MassTransit? (2013, 8 21). Retrieved from MassTransit 2.0 documentation: http://docs.masstransit-project.com/en/latest/overview/backstory.html

Videla, A.;& Williams, J. J. (2012). *RabbitMQ in Action: Distributed Messaging for Everyone.* Manning Publications.

Young, G. (2013, 8 21). *Simple CQRS example*. Retrieved from GitHub: https://github.com/gregoryyoung/m-r/commit/493705faa6079e6146f8b8e1d7ff5fd3d2d568c5

7 Appendices

Components of the migration system