FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

SOA use in television systems

Francisco José Madureira Brito

Master in Informatics and Computing Engineering

Supervisor: Eurico Manuel Elias Morais Carrapatoso (Professor)

09^{th} February, 2011

SOA use in television systems

Francisco José Madureira Brito

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: Ana Paula Rocha (Professor)
External Examiner: Paulo Sousa (Professor)
Supervisor: Eurico Manuel Elias Morais Carrapatoso (Professor)

09^{th} February, 2011

Abstract

Change is the most constant element in our world. Throughout the ages everything has changed in our world, from continents to dominant species, and even these change throughout time. Mankind is a paradigm of change since our species has evolved for centuries, not only anatomically but also in our way to perceive the world that surrounds us. In fact we evolved to the point where we change what surrounds us at our image and even our creations evolve.

One example of this are computer systems. Computers, which started out as "stand alone machines", were early on put to talk to each other. This, however, is no trivial task. Again, evolution and change comes into play and, along the years, several competing companies come up with their own technologies which mean that developing a standard would be quite the undertaking.

The solution to this problem started to appear in the last decade in the form of a new architecture for distributed computing, the Service Oriented Architecture (SOA). Since SOA is an architectural style and not a technology, it has to rely on a specific technologies in order to be implemented. These technologies are the Web Services. Web Services a series of protocols that provide standards for communication between computers.

The first part of this report starts by studying and describing the whole Service Oriented Architecture paradigm as well as the Web Services that are used in order to implement it. In the second part the development of a news selection system, newsRail, is described. In our ever-growing global village, news will spread faster and in greater number which means that newsrooms will surely need to streamline their workflows as much as possible. This is what newsRail aims to achieve, to provide a "gateway" through which journalists can select news segments from various sources and export them to a video editor. A SOA-compliant architecture was designed and implemented in C# using the .NET framework and Microsoft's IDE, Visual Studio.

Resumo

Mudança é o elemento mais constante no nosso mundo. Ao longo dos tempos tudo tem mudado no nosso mundo, desde os continentes até às espécies que dominam o Mundo, e até mesmos estes últimos se alteram ao longo do tempo. A humanidade é um paradigma de mudança já que a nossa espécie tem evoluído ao longo dos séculos, não só anatomicamente mas também na nossa maneira de percepcionar o mundo que nos rodeia. Na verdade, nós evoluímos ao ponto de mudarmos o que nos rodeia à nossa imagem e até mesmo as nossas criações evoluem.

Um exemplo disso são os sistemas de computadore. Computadores, que começara como "máquinas autónomas", foram, desde cedo, colocados uns com os outros. Isso, no entanto, não é uma tarefa trivial. Mais uma vez, a evolução e mudança entra em jogo e, ao longo dos anos, várias empresas concorrentes apresentaram suas próprias tecnologias, o que significa que o desenvolvimento de um *standard* seria um empreendimento difícil.

A solução para este problema começou a aparecer na última década, sob a forma de uma nova arquitetura para computação distribuída, a Arquitectura Orientada a Serviços (SOA). Como a SOA é um estilo de arquitetura e não uma tecnologia propriamente dita, tem que se basear numa tecnologia específica a fim de ser implementada. Estas tecnologias são os Web Services. Web Services são uma série de protocolos *standard* que definem a comunicação entre computadores.

A primeira parte do presente relatório começa por estudar e descrever todo o paradigma de Arquitectura Orientada a Serviços, bem como os Web Services que são utilizados para a implementar. Na segunda parte o desenvolvimento de um sistema de selecção de notícias, newsRail, é descrito em detalhe. Na nossa, cada vez maior, aldeia global, as notícias vão-se espalhando cada vez mais depressa e em maior número o que significa que as redacções certamente precisarão de agilizar os seus fluxos de trabalho tanto quanto possível. Isto é o que o newsRail pretende alcançar, fornecer uma "porta" através da qual os jornalistas podem selecionar segmentos de notícias a partir de várias fontes e exportá-los para um editor de vídeo. Uma arquitectura que respeita os princípios SOA foi projetada e implementada em C# utilizando a plataforma .NET e o IDE da Microsoft, Visual Studio.

Acknowledgements

Even though, at first sight, it may seem that way, a dissertation is not the work of only one person and, in fact, this project can be considered the sum of a series of people that, directly or indirectly, helped me achieve this goal and gave me all the best working conditions.

I would first like to thank my supervisor Professor Eurico Manuel Elias Morais Carrapatoso who generously accepted my request to be my supervisor when, almost a year ago, I almost barged into his office. I would like to thank Eng. Pedro Ferreira, my supervisor at MOG Solutions for allowing me to be part of a project that helped me learn and grow in every way. Also, I must thank Eng. Rui Ferreira for his patience as he was responsible for introducing me to many of the technologies used within MOG Solutions and also introducing me to the way they do their work.

I would also like to thank my mother, who always supported me in every step of this journey, sometimes, at the cost of great personal sacrifices.

Last, but by no means least, I would like to thank a very special group of friends who, maybe unknowingly, were of supreme importance at the beginning of this project and supported me all the way through. A dissertation, developing an architecture and implementing it, although challenging and extremely rewarding, is intrinsically a solitary task. Without their support and friendship, then it would have been truly lonely. Thank you.

Francisco Brito

"A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away."

Antoine de Saint-Exupery

Contents

List of Figures

Abbreviations

AAF	Advanced Authoring Format
API	Application Programming Interface
COM	Component Object Model
DB	Database
DM	Descriptive Metadata
DRM	Digital Rights Management
CORBA	Common Object Request Broker Architecture
CRUD	Create, Remove, Update, Delete
EBU	European Broadcasting Union
EDA	Event-Driven Architecture
EDSOA	Event-Driven Service Oriented Architecture
ESB	Enterprise Service Bus
GUI	Graphical User Interface
IDL	Interface Definition Language
KLV	Key Length Value
LE	Life Expectancy
MXF	Material eXchange Format
NAS	Network-Attached Storage
OOP	Object Oriented Programming
P2P	Peer-to-Peer
SMPTE	Society of Motion Picture and Television Engineers
SAN	Storage-Area Network
SOA	Service-Oriented Architecture
UL	Universal Label
URI	Uniform Resource Identifier
UMID	Unique Material ID
WADL	Web Application Description Language
WSDL	Web Services Description Language
WWW	*World Wide Web*

Chapter 1

Introduction

With the ever-growing adoption of IT technologies in the media industry came a new reality in the media production workflow, the file-based production. Files are gradually gaining *momentum* over the "old-fashioned" magnetic tape as the main storage and production medium in media facilities across the globe. From this shift in paradigm a single file format has begun to emerge as the "definitive file format" with which all media production should be done, the MXF standard.

Material eXchange Format, or MXF, is a standard file format that allows the combination of two "main components", a video and/or audio component called the essence and the metadata, into a single file [DWBT06]. The understanding that the future brought an age of file-based workflows and the past experiences with the lack of interoperability between the, then analog, systems, made the media industry look for a solution not only to the approaching digital future, but also to prevent some of the past mistakes and problems with formats. MXF was the answer to all these problems. It was custom designed to the media industry's needs for a file format that primarely aimed at exchanging complete programs or segments of programs, a format that carried program metadata as well as video/audio components and that allowed the retrieval of information or editing of incomplete files. But, even though MXF was a major advancement, problems still persisted.

After decades of processes based on magnetic tapes, the use of digital media and, inherently, the IT technologies that support it, in the media industry's production workflow brought with it a series of new challenges. Especially if we think of a media enterprise as an "entity" with production processes that intercept heterogeneous systems not designed to work together. Making these systems work together, integrating all of them, is no easy task. This is where Service-Oriented Architecture (SOA) comes into play. As it will be shown in Chapter 3, defining what is a SOA isn't something straightforward, but one can

find a succinct definition.

> “SOA is an architecture of independent, wrapped services communicating via published interfaces over a common middleware layer." [FF08, p. 73]

What SOA tackles is the inherent problems of lack of interoperability between heterogeneous systems in an enterprise by "cloaking" them as services. These services will then be "plugged" to an infrastructure composed by one or more systems acting as a single entity that orchestrates and enables the interaction between all the services or between the users and the services. That infrastructure is the "middleware layer" mentioned in the previous quote. One of the great aspects of SOA is that all the interactions between services and, consequently, the middleware layer through which they are connected, happen through messages and these messages are based on open standards. By promoting the use of standards in all of the communication happening in the infrastructure, SOA enables true interoperability between systems that, most certainly, are implemented in the most various ways.

What allows these services to "plug" and interact with the middleware layer is the interface, which acts almost as mediator between the application and the rest of the infrastructure. For now, the technical aspects of the interfaces aren’t going to be discussed, but, in short, they encapsulate the applications and "translate" the messages that are coming from the middleware layer into the applications "natural language" and vice-versa.

The media business is one of the most dynamic and volatile businesses that exist and having a big and sluggish IT infrastructure that is very resistant to changes, drastically reduces an enterprise’s ability to adapt to ever changing and unpredictable situations as the media business requires. SOA comes as a bright new possibility to the media enterprises as a way to tackle this.

1.1 Context

As previously mentioned the media business is one of the most dynamic and volatile businesses that exist. But, even within the media business, there is a particular environment that can be considered a paradigm of just how dynamic a media-related environment can be. That particular environment is the newsroom. This dynamism undoubtedly sprouts from the fact that news stories are extremely unpredictable in terms of time and location.

Because of this it is practically impossible for a "normal" broadcaster to actually create many of their own news stories since that would require a TV station to spend a lot of money maintaining office in all major cities in the world. What really happens is that many broadcasters rely on the services of *news agencies* like Reuters or EBU as a way of getting access to global news. Those news agencies gather, prepare and make available to

their customers (namely, broadcasters) a constant feed of "self-service" news stories that are practically ready to go on the air if needed.

The news agencies provide the broadcasters with a "box", henceforth named *basket*,that enables them to receive a constant news feed. Those feeds consist of a news clip and an associated metadata file thoroughly describing the clip and its content. The pair news clip and metadata file will henceforth be known as *asset*.

Therefore, the "generic" TV station newsroom, and its workflows, play an important role and surrounds many aspects of this dissertation since it's the main target of the project proposed by MOG Solutions, the company where this dissertation took place.

MOG Solutions, though a fairly recent company with about twenty-five employees, has become a top player in the "MXF based solutions" market and is the worldwide leader in MXF technology development [Sol10]. It develops software and hardware solutions for file-based television production systems. The focus of this project will fall upon a new product that MOG Solutions wishes to add to their existing line of MXF- related products. Unlike with other existing products, this particular project, newsRail, is aimed specifically at TV station newsrooms.

What newsRail aims to achieve will be briefly discussed in the next section.

1.2 Project

As stated before, newsRail is unlike any other product developed by MOG Solutions since its directed specifically at newsrooms and journalists. It's main objective is to streamline and facilitate newsrooms workflows. What happens nowadays is that, usually, a newsrooms has access to news feeds of several news agencies and a journalist searching for a news segment regarding a particular subject has to manually search within every basket for assets that he finds relevant. After finding the assets he then has to copy the video component of the asset into a video editor.

What newsRail mainly aims to achieve is automate as much as possible these two different workflows, namely, acting as a gateway between a user and any number of baskets so that, when a user searches for assets, he only has to search "within" newsRail and not manually go through every basket available, since the task of going through the baskets is left to newsRail. newsRail also sends, at the users request, one or more assets to an ingest system, namely, MOG Solutions mxfSPEEDRAIL F1000. The F1000 is a file-based ingest solution [Sil10] that allows users to move MXF media from any location and move it to any kind of shared media storage like the AVID Unity^{TM}ISIS [AVI10] or Media Network.

This project also required the study of a framework developed in house, Bobsled, as it will serve as foundation for newsRail. Bobsled was already used as the basis for another of MOG's products, namely, mxfSPEEDRAIL O1000 a file-based outgest solution. Since

newsRail has specific needs, different from the O1000 system, that weren't met by Bobsled at the beginning of this project, the latter had to be modified in order to implement newsRail. All these aspects, such as the Bobsled customization and what real benefits newsRail offers will be discussed in greater detail in the third chapter (3).

1.3 Motivation

The author's interest in multimedia and the will to work in a media-related environment are ultimately behind the motivation to embrace this project. It is a project that provides, at the same time, an engineering challenge and an opportunity. A challenge since, as mentioned before, require a thorough analysis and comprehension of an existing system, Bobsled, and also add some important functionalities required by newsRail. An opportunity because it will allow the author to work on something that will have real business and market value, and to work on the development of a completely new product.

1.4 Report Structure

The present report is divided into five main chapters. In this first chapter (1) the project is introduced as well as the author's motivation. The second chapter (2) is dedicated to the state of the art, where all different technologies involved in the project will discussed. The third chapter (3) is where this project is described in detail as well as all the problematic surrounding it. The fourth (4) talks about all aspects of the implementation of the project and, finally, in the fifth chapter (5) a rundown of all the work done and the degree of satisfaction with the objectives initially proposed and how they were implemented. Also in the fifth chapter, some "directions" where future work might lead are presented.

Chapter 2

State of the Art

2.1 MXF

2.1.1 History

During the mid-90's, the media industry foresaw a shift in technologies used to do their business and since the use of digital media was meant to become a reality due to its potential to enhance and "streamline" all aspects of media. In late 1996 a joint EBU/SMPTE Task Force, that had been born in an informal meeting earlier that year, recognized the need for a standardized file format for TV production and program exchange [DWBT06]. So the EBU/SMPTE Task Force was officially assigned to develop a blueprint for the implementation of new technologies foreseeing a timespan of over a decade. By 1997 the Task Force had already written the User Requirements report. The requirements that the new standard was supposed to meet were:

- it should carry a program-related metadata as well as audio and video components;
- it should allow the users to start working with the file even if the transfer isn't complete;
- it should provide mechanisms that allow the decoding of file information even if the file isn't complete;
- it should be open, standardized and essence compression-format free;
- it should be aimed at exchanging of complete programs or program segments;
- it should be simple and, therefore, allow real-time implementations.

In 1998 the Task Force produced the Final Report and later on, the SMPTE restructured itself to best suit what had been found in the report. To complement the already existing engineering committees like Audio, Video, Interfaces and Recorders, the SMPTE

created several new ones, namely, Systems, Compression, Wrappers and Metadata. MXF, or Material eXchange Format, was later developed based on the foundations created by the SMPTE committees specially the Wrappers and Metadata committees.

MXF emerged, later on, from the joint effort of the Pro-MPEG Forum and the AAF Association even though, initially, just Pro-MPEG proposed MXF as a common file format for the interchanging of files [DWBT06]. Since the AAF format was already in the process of being developed, Pro-MPEG and the AAF Association joined forces to finish defining the MXF format and, at the same time, make it interoperable with AAF. Even though Pro-MPEG was the responsible for the first demonstrations of MXF, the actual process of standardizing the format took place in SMPTE. In fact, after some revisions, all of the process of developing and maintenance of the format was put in charge of the SMPTE. This is easy to see since, structurally speaking, MXF uses several SMPTE technologies/standards in its foundations.

2.1.2 Standards and Specifications

Like it was just mentioned, MXF uses several SMPTE standards such as Unique Material Identifier (UMID), Universal Label (UL) and Key-Lenght-Value (KLV). There are also other related standards that will be mentioned ahead.

A UL is defined by ISO/IEC 8824 as an object identifier and, in its "SMPTE version" is, basically, a sequence with a fixed length of 16 bytes. The SMPTE UL is specified in the SMPTE 298M as a universal labeling mechanism used in identifying the type and enconding of data within a general-purpose data stream.

KLV is a numerically based and language independent coding syntax that has three fields, namely, *Key*, *Length* and *Value*. According to SMPTE 336M, the specification document for KLV, the *Key* field defines the data within the *Value* field. In the same way, the *Length* field, defines the latters. And, finally, the *Value* field, is the data coded as simply a sequence of bytes with no termination subsequence, since its length is previously defined in the *Length* field.

Defined in SMPTE 330M, basic UMIDs are 32-byte numbers that identify a piece of material like TV show or film. There are also extended UMIDs that are 64-byte values that uniquely identify each material unit in a package.

Other related SMPTE standards may include [oMPE10]:

- **S12M** — time and control code;
- **RP210** — metadata dictionary registry of metadata element descriptions;
- **RP224** — SMPTE labels register.

Obviously, the MXF standard also has a specification document, the SMPTE 377M. In Figure 2.1 one can see the structure of all the documents that make up the MXF standard.

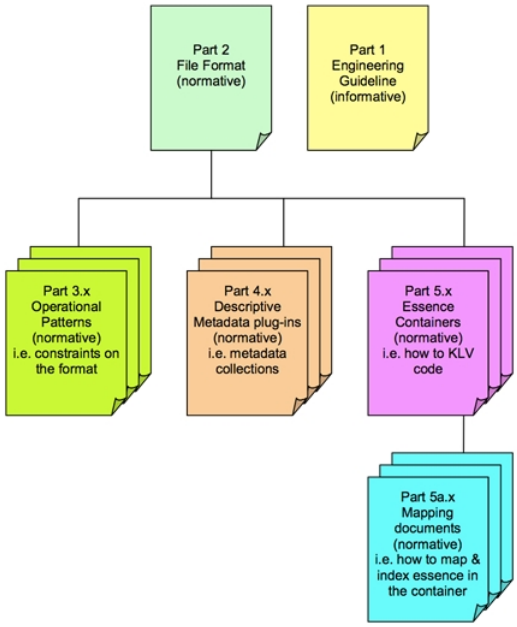

Figure 2.1: Structure of all the documents that make up MXF standard

Dividing the document into different allows new applications to be added in the future. The existing parts are:

- **Engineering Guideline (Part 1)** — guideline that offers an introduction and overall description of MXF;
- **MXF File Format Specification (Part 2)** — this is the fundamental file format specification (SMPTE 377M);
- **Operational Patterns (Part 3)** — describes the operational patterns of the MXF format;
- **MXF Descriptive Metadata Schemes (Part 4)** — defines MXF descriptive metadata schemes that can be optionally plugged in to an MXF file;
- **Essence Containers (Part 5)** — defines the MXF essence containers that "hold inside" audio, video or other kinds of essence;
- **Mapping Essence and Metadata into the Essence Containers (Part 5a)** — individual mappings of essence and metadata into the generic essence container.

2.1.3 MXF File Structure

As it was stated earlier, an MXF file, basically, combines two things: essence and metadata. Even though this is true, the actual structure of, even a basic MXF file, is quite more complicated than that, as it is possible to see in Figure 2.2.

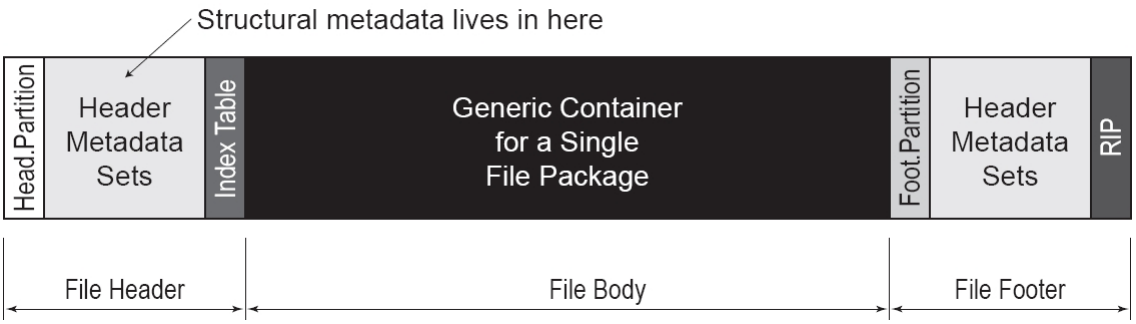

Figure 2.2: Basic Structure of an MXF File [DWBT06]

The basic MXF file is composed by a *file header*, *file body* and *file footer*. In the *file header* lies a very important aspect of MXF, the structural metadata. The structural metadata is nested inside the header metadata which in turn completely describes what the file is intended to represent and so, it doesn't actually contain any video, audio or any other kind of essence, it just describes the synchronization and timing between them. All the the essence elements inside an MXF file that need to be synchronized are always represented by a track and, since there are always several elements there are also several tracks. An example of all these different tracks can be seen in Figure 2.3.

A track is nothing more than a structural metadata component representing time in an MXF file. A track is usually divided in edit units which, normally, corresponds to the video frame rate of the video/audio file ad the typical edit unit will have about $^1/_{25}$ or $^1/_{30}$ of a second. Since there is more than one track, one might begin to think about grouping them. A group, or collection of tracks is called a package and, in an MXF file, there can be more than just one package [DWBT06]. This happens because the goal of the MXF format is not to only playback stored essence but also to easily select/edit material contained within the file. It's easy to see that a single package containing the tracks that represent the synchronization between the elements of the stored material isn't enough. More packages representing the way the material is supposed to output are also required. Figure 2.4 shows a representation of different packages contained inside a MXF. The first package is known as the *file package* and the second is the *material package*. The first, *file package*, represents the content as it is stored within the file itself. On the other hand, the *material package* is the collection of tracks representing the way the content within the MXF is supposed to be played out. The data model for the *material package* can be seen in Figure 2.5. What the model "reveals" is that the *material package* can have one or more *Tracks* and that each *Track* is composed of a sequence made up of one or more *SourceClips*. These *SourceClips* have attributes that directs to the *file package* and correct track where the actual essence can be found.

Another type of metadata contained within an MXF file is the *descriptive metadata*

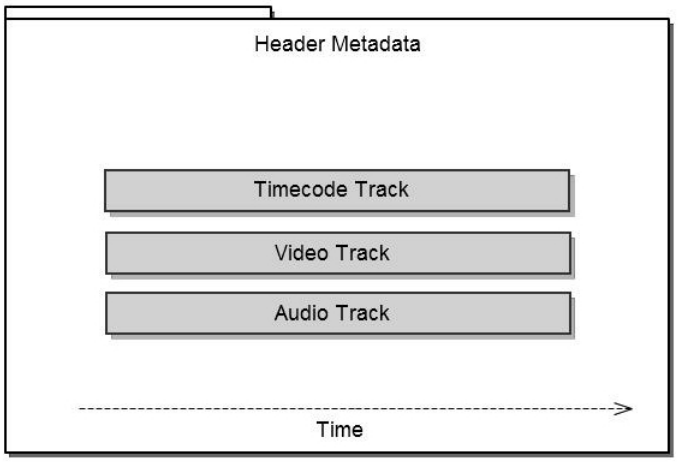

Figure 2.3: Different track representation/synchronization in header metadata

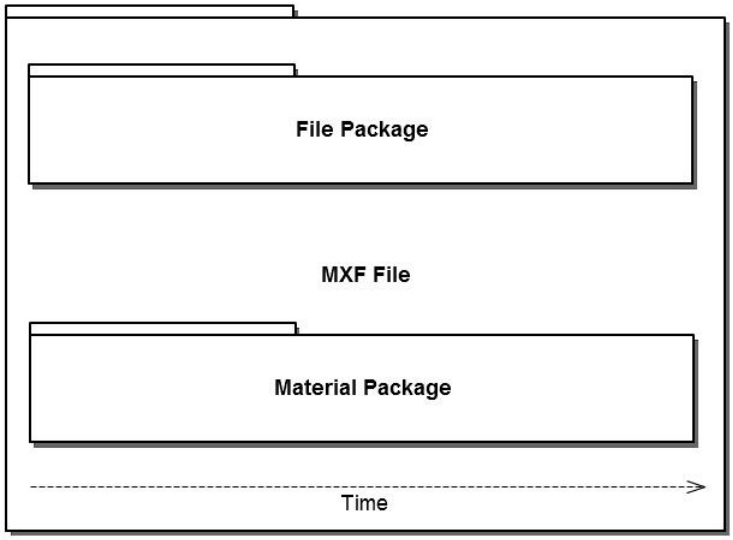

Figure 2.4: Different packages in an MXF file

(DM). Unlike the structural metadata which is "machine-generated", DM is created by humans for human use but, regardless of this aspect, DM is also represented by a track just as the structural metadata. In order to achieve true DM interchangeability, a standardized structure had to be defined. The answer to this came in the form of the SMPTE standard 380M - Descriptive Metadata Scheme 1 (DMS-1) which divided DM into three separate categories:

- **Production Framework** — production-related descriptions;
- **Scene Framework** — scene/content related descriptions;
- **Clip Framework** — descriptions related to the way the clip was shot.

Unlike structural metadata, DM doesn't need to be continuous like the way video and audio need to be and it can actually describe more than one essence track. For example, a DM can describe Picture, Audio and Timecode Essence tracks simultaneously.

Since content description, as mentioned before, was always an important aspect during the whole creation process of the MXF format standard, describing not only what the content was but also where the content came from is a very important aspect of MXF metadata. This is where the linking mechanism between packages, named *source reference chain*, comes along. The *source reference chain* links every package in an MXF file, from the top-level *material package* to the lowest-level *file packages*. As mentioned earlier, an MXF file, has one *material package* and can have one or more *file packages*. The *material package* synchronizes the stored content during playout and the *file packages* describe the content stored in the file. However, due to the fact that there might be several *file packages*, things aren't so linear. The reality is that, in the case that there is more than one *file package*, the top-level packages describe the actual stored content and the lower-level ones describe what has happened to that same content in the past. This allows the creation of "footprints" of past processes that the content was subjected to. Each process is identified by a Unique Material ID (UMID) and even for past processes the UMIDs are kept along with a description of what the content of the package was at the time. That description of the essence inside a package is done with the help of *essence descriptors* who describe the parameters of the stored essence. Since essence descriptors are "linked" to the file packages, even to the lowest-level ones, it's possible to have a complete history of the essence in its various "forms". An analogy could be thinking of these essence descriptors, especially in low level file packages, like "snapshots of the essence's past lives". The essence descriptor, however, allows for other useful tasks like mining the MXF file for information. For example, it's possible for an MXF application to know if its capable of handling the essence contained inside the file. That same MXF application could also "discover" in a low level file package that the content was described as DV and, in the highest level file package, the application could "see" that the current

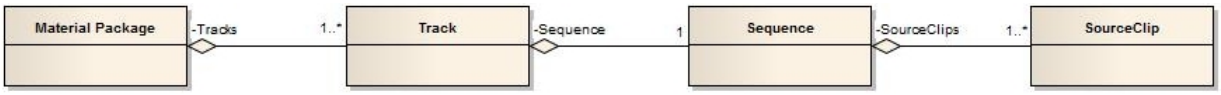

Figure 2.5: Material package basic data model

content is described as MPEG. This would indicate that the essence was subjected to a transcode process at some stage. Essence descriptors fall in two categories: file descriptors and physical descriptors. File descriptors, basically, describe the stored content inside an MXF file and can include parameters like resolution, sample rate, compression format and many others. Physical descriptors, however, describe how the content was obtained. An example if content that originated from a tape digitalization. in this case there should be a tape descriptor inside the file. If, for example, the content was the result of an audio file conversion, then an audio physical descriptor would also be in the MXF file.

Another important aspect of the MXF file are the *operational patterns*. The complexity of the source reference chain and the complexity of the MXF encoder/decoder (required to generate of play an MXF file) are directly linked, since the first controls the latter two. A file that has the material package and a single file package has a completely different complexity of a file that has 3 file packages and a material package synchronizing them all. And, since, the source reference chain is the link between all the packages in an MXF file, it's easy to understand how its complexity directly affects/controls the complexity of the MXF encoder/decoder required to generate/play the file. Figure 2.6 shows the MXF operational pattern matrix. The columns differentiate the complexity of the time axis in an MXF file. This is represented by the way the material package plays out the file package(s). In the first column, everything is played out in its entirety. In the second, every file package is also played out in its entirety but according to a defined playlist. And, finally, in the third column, the material package plays out according to an *edit decision list* (EDL), randomly accessing the MXF file and playing out only selected portions of the file packages. On the other hand, the matrix rows, differentiate the package complexity of the MXF file. In the first row, only a single file package is active at any given time in the output timeline. The second row represents the the material package-enabled synchronization between two or more file packages. Finally, the third row, allows for more than one material package to exist in the file. This is especially useful when dealing with versioning and multi-language capabilities. The complexity of the MXF encoder/decoder required to generate/play an MXF file increases as one moves left and/or downward in the operational matrix.

Until this point, it has been referred many times that the "essence is contained within the MXF file". This is actually done in the *essence container*, a diminutive from the "original" naming, *essence in a generic container*. MXF allows for several essence containers

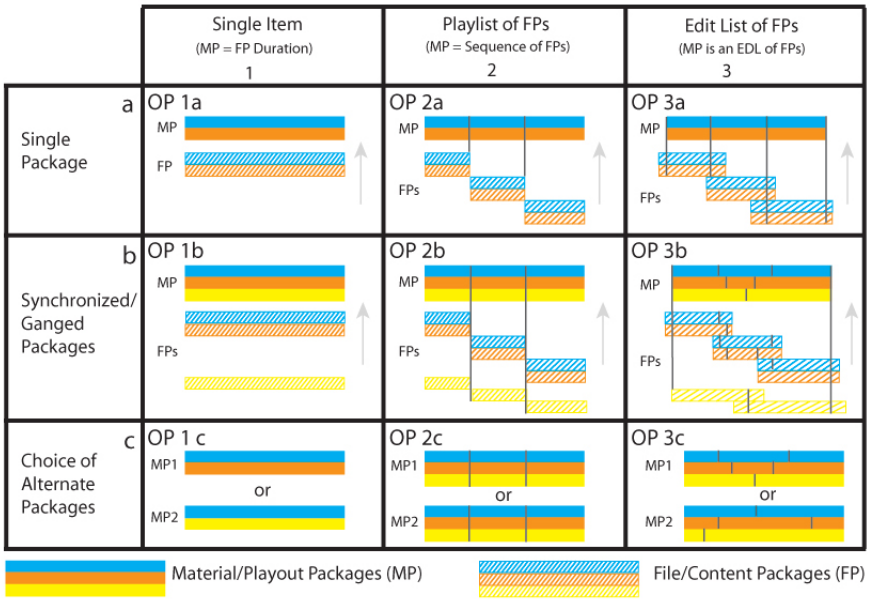

Figure 2.6: MXF operational patterns [Ive06]

since a single file can hold more than one essence inside. The generic essence container actually divides the essence into content packages (CP) of equal duration. There are two different ways an MXF file is divided into content packages. The first are the *frame-wrapped files*. In this case the file divided in such a way that all the content packages have one video frame of duration. The second are the *clip-wrapped files* and in this case the content package has the same duration as the file itself.

With this multitude of elements inside, it's not difficult to see why identification is a key feature in the structure of an MXF file. In this "chapter", nothing stands out more than the already mentioned UMID. This number, specified in SMPTE 330M, uniquely identifies a piece of material. For example, even in a OP1a file, a file package would have a different ID from the material package ID, even if the played out material in the latter package is almost equal to the material in the file package. Inside a package, tracks have also have a unique ID, the *TrackID*. This, however, is only unique within the scope of a specific package. Different tracks on different packages can very well have the exact same TrackID. This means that, to pinpoint a single track within a file, not only the TrackID is required, but also the UMID that identifies the package where the track resides. There are more identifications within an MXF file, like the *StreamID* that identifies a separate stream of bytes, but these two, UMID and TrackID, are two of the most important ones since they belong to the structural metadata holding the file together.

2.2 Service Oriented Architecture

According to Andrew Tanenbaum and Marteen von Steen's book, *Distributed Systems - Principles and Paradigms*, the definition of a distributed system states that:

> "A distributed system is a collection of independent computers that appears to its users as a single coherent system." [TS02]

At its core, a Service-Oriented Architecture (SOA), is nothing more than a distributed system since, and this will be studied in greater depth over the next sections, it's composed by heterogeneous elements "wrapped" as services that communicate over a common layer known as middleware [FF08]. Since SOA is just a "set of design principles", the word *software* isn't necessarily "linked" to SOA, because the latter can be implemented using anything from software to people exchanging paper forms around. But, before, studying SOA in greater depths, one must know how and why it came to be.

2.2.1 History

As with many of the architectures and principles in computer science, SOA didn't just spontaneously appear out of nowhere. Instead, SOA, was born out of the "aggregation" of some of the principles of existing architectures/methodologies. One of these methodologies that greatly contributed to SOA and its "philosophy", was Object Oriented Programming (OOP). When OOP first appeared it was a sort of revolution in the software world since it introduced a completely new programming paradigm. It introduced us to the *object*, an encapsulated software entity that could perform operations and that could be reused. This is very similar and, in fact, is behind one of SOA's cornerstones, *service re-usability*. OOP programming languages, however, had one major downside, they're platform-dependent. In other words, a program written in C++ only finds itself to be really useful in another C++ platform. This fact lead to the research of architectures that allowed multi-platform applications integration. The first answer to this problem came from Microsoft's Component Object Model (COM). This architecture allowed, otherwise incompatible applications, to communicate with each other through the exchange of objects. DCOM, or Distributed Component Model, was COM's natural successor. The latter used the Interface Definition Language (IDL) which allowed the different COM component to "tell" each other what sort of objects/operations they could offer. Therefore, IDL provided a common standard interface language so every application knew what to expect when contacting another application. This principle of published and standardized interfaces is similar in SOA, especially through Web services implementation.

Another architecture, the last in fact, that directly, and greatly, influenced SOA was the Common Object Request Broker Architecture, or CORBA. This became even more popular than Microsoft's COM and DCOM mainly because it introduced the idea of "wrapping" a software component with a standard interface, like IDL. The notion of using interfaces to wrap and, therefore, hide software components is actually what one sees in a modern SOA implementation as well. Even today, many enterprises implement have a CORBA implementation, because, even though, it uses different standards from those that SOA uses, both concepts are very similar, to the point that, it is possible to implement SOA in an IT infrastructure that already has a CORBA implementation.

2.2.2 Definition and Benefits

After an introduction to SOA's history, it is important to clearly define it. This report began with a short and concise definition of SOA, but a proper definition is in order. The Open Group, a vendor and technology-neutral consortium [Gro10a], has a good example of a SOA definition. It states that:

> "Service-Oriented Architecture (SOA) is an architectural style that supports service orientation.
>
> Service orientation is a way of thinking in terms of services and service-based development and the outcomes of services.
>
> A service:
>
> - Is a logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit; provide weather data, consolidate drilling reports);
> - Is self-contained;
> - May be composed of other services;
> - Is a "black box" to consumers of the service.
>
> An *architectural style* is the combination of distinctive features in which architecture is performed or expressed. The SOA architectural style has the following distinctive features:
>
> - It is based on the design of the services – which mirror real-world business activities – comprising the enterprise (or inter-enterprise) business processes;
> - Service representation utilizes business descriptions to provide context (i.e., business process, goal, rule, policy, service interface, and service component) and implements services using service orchestration;

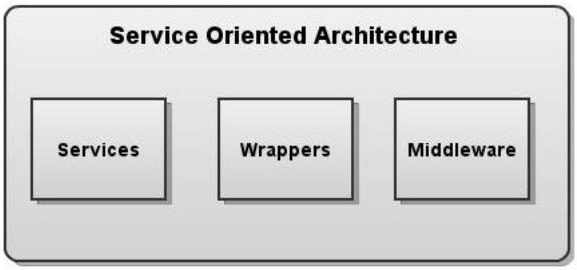

Figure 2.7: Service Oriented Architecture three main "components"

- It places unique requirements on the infrastructure – it is recommended that implementations use open standards to realize interoperability and location transparency;
- Implementations are environment-specific – they are constrained or enabled by context and must be described within that context;
- It requires strong governance of service representation and implementation;
- It requires a “Litmus Test", which determines a “good service”." [Gro10b]

A Service-Oriented Architecture is composed by three main components (seen in Figure 2.7): *services, wrappers* and *middleware layer*. *Services*, are a collection of software components/applications which individually carry out business processes [SHM08] and that provide business value to the enterprise [FF08], also, a service should have a well defined platform independent interface and it should have self-contained functionality.As for *wrappers* they "sit between" a service and the middleware layer, and its "main job" is to transform or, in other words, "translate" messages that pass through them. Finally, the third component , is also the one that, among other things, enables message exchange between services. That component is known as the *middleware layer*. The middleware layer is, in fact, just a term used to describe all the underlying communication infrastructure built around a network of services.

These three components will be discussed in greater detail in the following sections but there’s more to SOA than just these three main components. In fact, just "having" these is no guarantee of successfully implementing a SOA infrastructure. To do this there are four aspects that one should keep in mind [PH07]:

- *Service enablement* — each discrete application needs to be exposed as a service;
- *Service orchestration* — distributed services should be configured and orchestrated in a unified and clearly define distributed process;

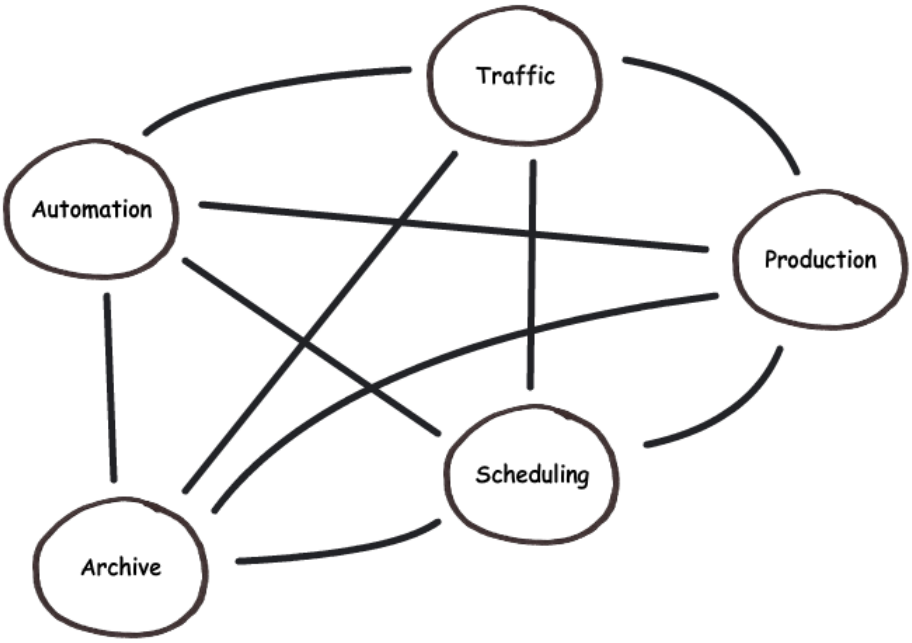

Figure 2.8: Example of an accidental and tightly-coupled architecture

- *Deployment* — emphasis should be shifted from test to the production environment, addressing security, reliability, and scalability concerns;

- *Management* — services must be audited, maintained and reconfigured. The latter requirements requires that corresponding changes in processes must be made without rewriting the services or underlying application.

A SOA is easily recognizable just by looking at the architectural diagram of an IT infrastructure. Figure 2.8 represents an example of a tightly-coupled architecture. Tight-coupling is the exact opposite of loose-coupling, and by loose-coupling one refers to the concept of allowing a user or application to use a service to the fullest of its functionalities without any knowledge of the underlying technical details. Normally, tight-coupling it's a "natural side-effect" of an accidental architecture. Unlike loose-coupling, in tight-coupling one refers to the fact the services are so intricately connected that changing just one service causes a "chain reaction" where all other services that are directly or indirectly connected must also be changed. This happens because implementation isn't separate from interface. On the other hand, in Figure 2.9 one can see an example of a typical Service Oriented Architecture. In that example it's clearly visible how all the three main components mentioned earlier connect to each other. One can clearly see, for example, how a wrapper serves a gateway between the service and the middleware layer and that one can easily add more services to the infrastructure without affecting previously

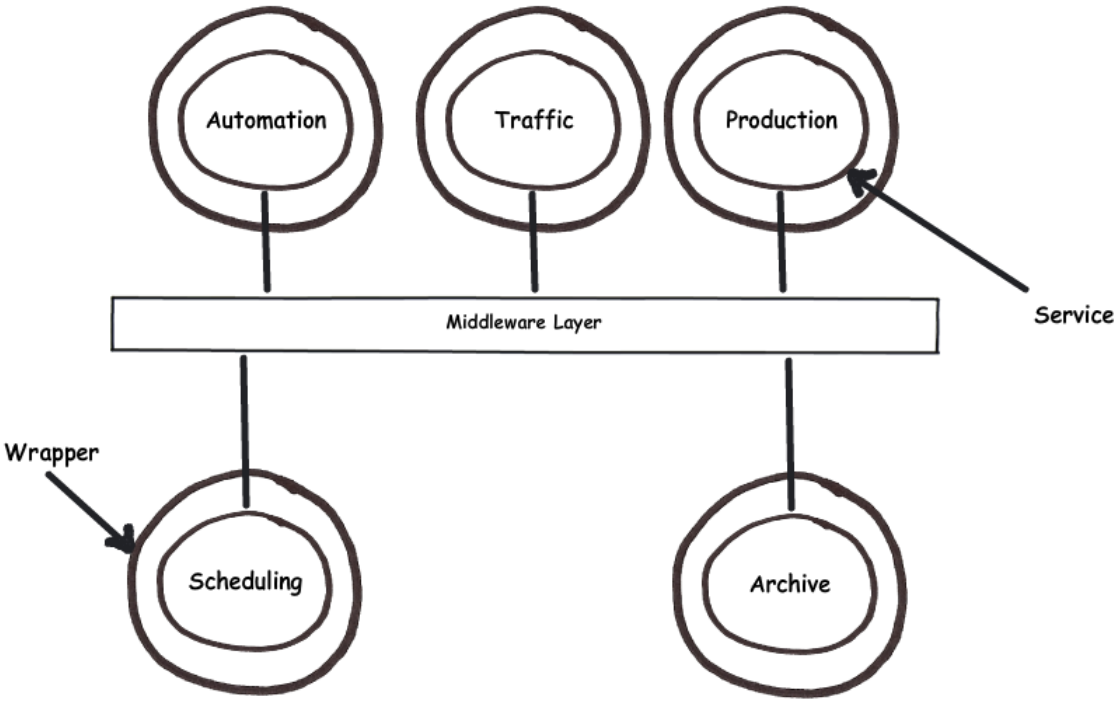

Figure 2.9: Example of a Service Oriented Architecture

existing services, since none of them is directly connected and all communication is done through the middleware layer.

Service Oriented Architecture aims to provide an enterprise or, in the scope of this report, a media enterprise with three main benefits:

- *Business agility* benefits
- *Business visibility* benefits
- *Organizational benefits*

Business agility is, in a way, a direct consequence of some of the SOA characteristics just mentioned. Especially, the layer of abstraction provided by the wrappers and standardized interfaces. This layer of abstraction will enable independence between the SOA components so that their implementations do not affect each other, or in other words, loose-coupling. As stated before, loose-coupling allows changing one component, like a specific service, without affecting the of the components. This is something that is practically impossible in a tightly-coupled architecture and, since the media business is a very volatile business,this ability to easily change components in the infrastructure is a valuable quality for any media enterprise.

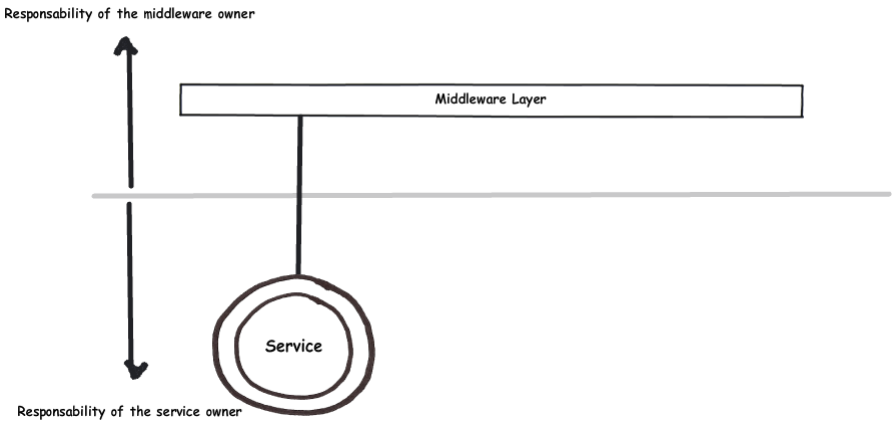

Figure 2.10: Boundary between vendor responsibility and integrator responsibility

Despite its "name", *business visibility* refers not only to actual visibility into the business but also visibility into actual data. What this means is that a SOA allows the aggregation of data from across the enterprise for analysis and monitoring purposes. In order to achieve this, one must gather and aggregate data from multiple systems in the infrastructure. In a SOA, services measure and provide this data. For example, in a news corporation a graphics, asset management and/or newsroom computer services may be able to provide data about story popularity from their systems. The "trick" is to be able to identify and move that data from the services into the only place in a SOA that has visibility to all services and that can collect and analyze their data, the middleware layer. Service will then have to externalize their data in a real-time or a as-needed basis. Since the details of what the data may be and the way it's formatted is something that is business-function specific, differing from system to system, wrappers are the key to successfully expose that data to the middleware layer. That way, the middleware layer, just need to make a business level call in order to obtain the data is needs, leaving the rest in charge of the wrapper. Since all inter-service communication flows through this layer, the integrator can "listen" to all the messages passing through the middleware and collect data from them. All this data is extremely useful as it allows someone, a system administrator, for example, to " peek inside" and assess the status of the enterprises architecture.

Lastly, the Service Oriented Architecture allows some *organizational benefits*. Thanks to services, and the way they hide implementation from functionality, and to middleware, SOA provides a set of organizational benefits that are especially useful in an enterprise with multiple departments all wanting interoperability. Ownership, for example, is an area that is made clearer in a loosely-coupled service-based architecture. Figure 2.10 clearly illustrates this situation and how SOA helps separate vendor responsibility from integrator responsibility. That said, the "line" that actually separates vendor and integrator responsibility is the service interface. The vendor, or owner of the service is responsible

for writing the appropriate interface for that particular service and it's the integrator's responsibility to adequately write processes and integrate that interface into the middleware layer.

In the next subsections all the three main components, *services, wrappers* and *middleware* of a SOA will be described in greater detail.

2.2.3 Services

As the name Service Oriented Architecture suggests, services play a crucial role in the whole service-oriented methodology. However, defining what a service is, isn't a trivial task as first it might seem. One can compare a service to a mini program that does a fundamental unit of work that's designed to be reused and recombined with other mini programs to build something known as a composite application. According to Jim Adamczyk, senior executive at Accenture in New York City, "services really are like today's Legos with multiple custom shapes" [Xin09]. In Service Oriented Architecture, a service is a black box entity that has inputs and outputs and that provides business value. It has, associated with it, a service description and an interface, so that the user knows what inputs to give the service and what kind of outputs it should be expected. This is know as "calling" the service. Since the services are like black boxes, it doesn't really matter if "inside" the black box is a person or an application doing all the work when the service is called. As mentioned before, this happens because the service interface is independent from its implementation. That interface is know as the *wrapper*.

The act of wrapping is what turns a generic application into a service, or business service, since it provides encapsulation. A service, or business service, must always provide some business value. That means that not everything with inputs and outputs and that interface independent from implementation can, or should, be considered a business service. That is why defining a service is one the hardest things to do when implementing a SOA or when developing SOA-enabled products. Functions and/or operations are not, necessarily, services. An object that has the ability to turn integers into strings isn't, and probably shouldn't be, a service. What this can be is part of a larger service. It is important not to confuse functions/operations with services. Also, services shouldn't be conversational which means that in some businesses, like the media business where some processes require a high degree of interaction, some business processes, like producing a movie, shouldn't be "transformed" into business services. This leads to another difficulty when dealing with services, knowing when an application should expose a single service, when should it expose multiple services and when should it be tightly-coupled with other applications in order to expose a loosely-coupled service.

According to C. Cauvet and G. Guzelian's article [CG08], where they propose a way to adopt a service-oriented approach to business process modeling, a business service,

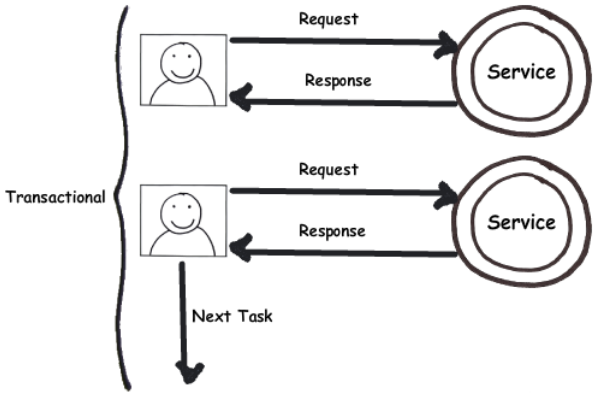

Figure 2.11: The two types of transactional services

which delivers a process to achieve a certain goal by using resources, can be described by dividing it into three parts:

- *Profile*
- *Structure*
- *Process*

The first part, profile, defines the business service's the purpose of the service, it emphasizes the business problem solved by the service. The profile contains contextual knowledge on why a "potential customer" would gain advantage by using the service.

The structure part describes the process organization used in order to achieve the service's goal. The structure part is divided in three elements: an initial situation, a final situation and a process structure. The initial situation indicates the pre-conditions and the resources necessary to process realization. On the other hand, the final situation specifies the results and the post-conditions of the process. Finally, the process, can be one of four types of processes:

- **Atomic Processes** — realize elementary goals.
- **Composite Processes** — correspond to complex goals.
- **Simple Processes** — allow differing process realization in other services.
- **Decisional Processes** — are a specific case of composite processes, since they propose several alternative decompositions of a goal.

Lastly, the process part defines the solution offered by the service and, by solution, the author's mean an executable process described in terms activities and business objects.

As one can see, properly defining a service while trying to "maximize" its business value is not a trivial task. But not all services have the same "degree" of business value. In fact, there are different kinds of services and also several ways to "expose" and to implement them. There are two main types of services: transactional and one-directional. The first category, as seen in Figure 2.11, transactional, is divided into two groups, synchronous and asynchronous. In an asynchronous service there are four basic steps:

- **1.** — Request is sent to the service.
- **2.** — Service works on request.
- **3.** — After completing requested task, service sends a response back to the service requester.
- **4.** — The service requester carries on to perform other tasks.

As for one-directional services, there is only one real step that is sending the request to the service. After sending the request, the service requester proceeds with another task and doesn't wait for a reply from the service. Figure 2.12 portrays this kind of service.

As for service implementation, within an enterprise, the most common one is a software application. Wrapping one application as a service and exposing a service interface to the rest of the enterprise's SOA is the most usual procedure. In the media industry there are several examples of this like transcoding servers or automation systems. The individual applications are wrapped exposing one or more services providing business value to the enterprise. Since services need to be encapsulated, abstracted and provide business value, applications that belong to a larger suite of applications, and therefore, are tightly-coupled aren't a good candidate to expose services. In this case, when dealing with a tightly-coupled suite of applications or multiple component system, a better architecture is to configure a single system component to serve as a service gateway and only have the wrapper communicate with that single component. That component will then deal with all of the "communication and handling" with the rest of the systems in the suite.

Finally, another type of service implementation is exposing entire departments, facilities or even, whole enterprises as services. In a way, this is, basically, exposing a SOA as a service. This is particularly useful when companies have a department that outsources since it allows that particular department to be accessible to the "outside world". Freelancing, a common fact in the media business, also benefits from this type of service implementation. The way to implement this kind of services is similar to exposing a large suite of systems as a service. The best option is to implement a service gateway that will ensure that only the desired functionality is made available to other enterprises and, at the same time, it also ensures that the inter-enterprise communication is safe and secure.

Another important subject when defining and designing services is defining it's class. Services can be divided into different classes according to it level of abstraction. In their

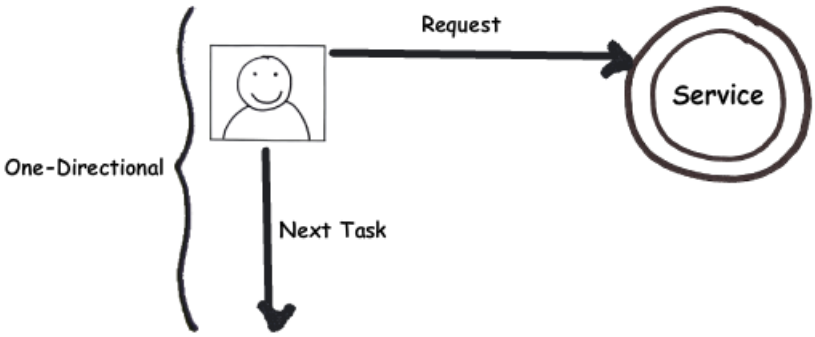

Figure 2.12: A one-directional service

book, Footen and Faust [FF08], consider three major hierarchical layers of service abstraction. The business value of the services within those layers grows as one "climbs" the hierarchical structure. These three layers, or domains, are the application domain, the service domain and the business domain. This SOA model is based on the fact that integrators first wrap the business services almost at application the application level and then wrap suites of those lower level infrastructure services into larger services with real business functionality that do business specific tasks.

The lowest of the domains is the *application domain*. Here, there are three main components: the application itself, the application APIs and the wrappers that transform those APIs into low level services, also know as, technology-level services. Applications, obviously, do all the work and present APIs as a way to expose itself to the "outer-world. Even so, the most important component in the application domain is the wrapper. This happens because it's the wrapper's job to transform the application APIs into technology-level services required by the enterprise. In a way, the wrapper shows an idealization of the application's functionality. Metaphorically speaking, it's almost like squeezing the juice out of an orange, where the orange would be the application and its APIs and the hand squeezing the orange would be the wrapper.

In the middle of the "hierarchical ladder" sits the *service domain*. This layer, even though it can contain also the technology-level services as the previous one, is, also, the "home" to infrastructure-level services. These services, comprised only by the wrapper itself, have no application API "behind them", since they can be considered almost like a mash-up of technology-level services. They're implemented as way to expose technical tasks that none of the existing technology-level services in the SOA expose, since none of the existing applications has that functionality.

Finally, the topmost layer is the *business domain*. This is the layer that an enterprise middle-management and even, executives better understand because this domain contains notions that are familiar to their line of work. Notions like business logic, business rules and business processes. All these components are designed to meet the business

requirements as that is the main goal of this domain, cater to the business. At the same time those components are completely independent from the technology that, "behind the scenes", actually meets those requirements. Also, in this layer of the SOA model, people can orchestrate business events known as business-level events into business processes. The purpose of business-level services is to represent fundamental business activities that the SOA can support. As for the business processes, they're orchestrated workflows that connect various services and, even, other processes in order to achieve a business goal.

All this layering is rendered useless if one cannot "service-orient" a system in order to fit that layered SOA model. This process is known as *service decomposition*. Decomposing a service is the act of breaking down application components and identifying the important details of each service and it fits the layered SOA model mentioned before by defining technology-level services and designing a wrapper that will turn existing APIs into those technology-level services. This is a crucial step in service orientation that must be undertaken with special concern toward how these technology-level services will fit into the upper-layers of a SOA. In order to successfully decompose a service, three things need to fully understood, namely:

- The application being decomposed and it's APIs;
- The form the service interface should take;
- The way the completed service will participate in the overall architecture.

Decomposition is hard to master and an integrator should be careful when decomposing applications in order to not copy the vendor's APIs at the service interface level. This aspect cripples reusability, a cornerstone benefit of the SOA design methodology. In fact, service interfaces should be customized in order to meet the specific needs of the enterprise or any department to which that interface is "aimed". Another important aspect to avoid crippling service reusability is, before decomposing an application and figuring out which services to expose, one should look at all available services and see if any of the existing services could be reused. Also, SOA aims at having a structured, componentized data model. Data should be defined, discrete and focused on the business functionality of the various services. This data that is digested by services is known as *business objects*. In the application decomposing process, one should know what business objects that application requires to do its job and what business objects it will create or provide back to the SOA/enterprise.

2.2.4 Wrappers

As stated earlier in this report, *a service equals an application plus a wrapper* and the main function of the wrapper is to transform the applications APIs into technology-level

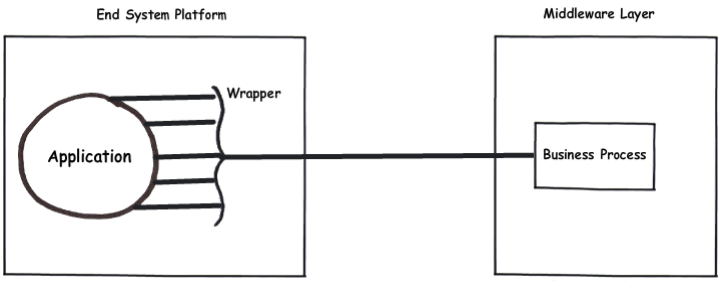

Figure 2.13: Example of a wrapper implementation: wrapper built in the application

services. After an application is wrapped, there is no need to call its APIs directly. In architectures predating SOA, wrappers might have been known as adapters. This causes some misunderstandings between what really is a wrapper and an adapter. A wrapper's job was already mentioned, as for the adapter, even though it might also be used in the context of a SOA, it refers to a component in the middleware layer that transforms one standards-based method of communication into another. Basically, adapters, allow services that use different interface technologies to all connect to the same middleware layer. The middleware layer uses adapters in order to convert one technology into another and, therefore, facilitating interoperability.

Apart from the main function that a wrappers must perform, there are two "sub-functions" that "derive" from that main function and that are crucial in order for the wrapper to properly perform its job. First is must encapsulate and abstract the application to create a service interface. Second, it must transform the technology used in the application to whatever is the chosen SOA standard.

As for implementation, there are three possible options of implementing wrappers, also known as, *wrapper models*. Namely:

- Build the wrapper in the application using custom workflow or extensible API capabilities.

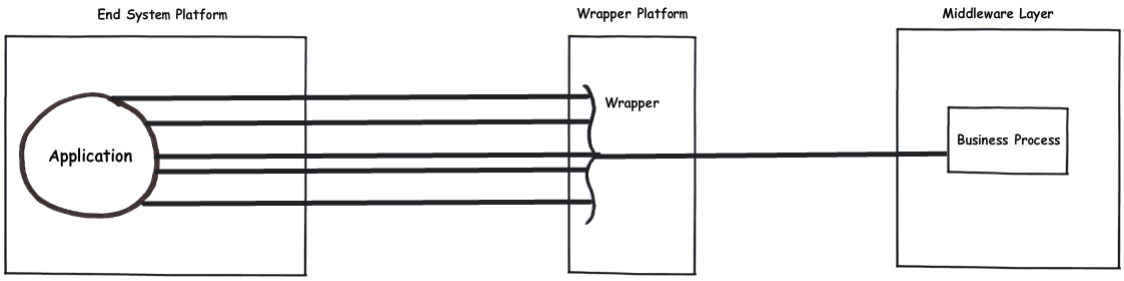

Figure 2.14: Example of a wrapper implementation: wrapper built in a separate environment

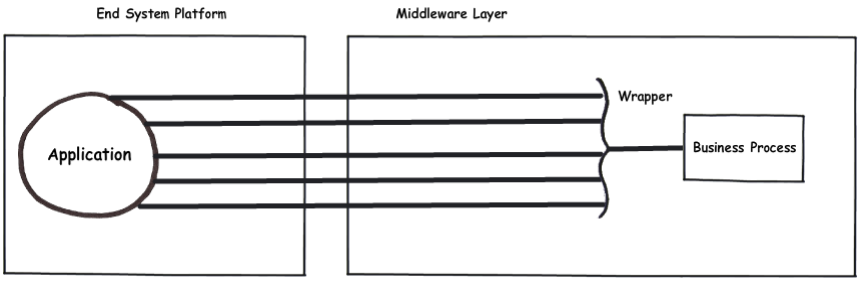

Figure 2.15: Example of a wrapper implementation: wrapper built in the middleware layer

- Build the wrapper on external platform running in a separate environment (either on the same or different physical hardware).
- Build the wrapper in the middleware layer using SOA tools and technologies.

In the second wrapper model, as seen in Figure 2.14, the wrapper is built in a completely separate environment which could mean that it's even on a completely separate server from the end system and the rest of the middleware layer. This implementation model only requires that the wrapper is capable of calling the application API and expose technology-level services using the chosen SOA standard. Even though this implementation model means that there is a lot more programming to do, it also means that an integrator is given complete freedom about how the wrapper functions. It's a wrapper implementation model particularly useful when integrating applications that are particularly difficulty to integrate or when one needs a very complex wrapper design that, for example, needs to call the APIs of several applications.

The third and final wrapper implementation model, shown in Figure 2.15, implements the wrapper in the middleware layer. The middleware, studied in greater detail in the next chapter, is home to several components ranging from an enterprise service bus, to messaging utilities and data handling. In a way that can be seen as "similar" to what is done in the first wrapper model, an integrator can use all those components in order to properly encapsulate and transform data in order to change the application interface into a technology-level service. If, for example, an application already uses modern and standardized communication protocols but that aren't the ones chosen for service communication, the integrator can find or build a component that just has to transform one protocol into another. Data transformation can also be done in the middleware layer using the same principles. An integrator should, however, be careful using this implementation model since there's always the possibility of "clogging" the middleware layer with custom wrappers and/or data transformation components. Also, an integrator should always be careful not to accidentally "tight-couple" those components to other services or processes or processes in the middleware.

This isn't the only danger or problem related to wrapper design. One of the most common problems is the "Center of the Universe" (COTU) problem. This problems occurs when a vendor system must control functionality that has wide applicability in an enterprise. This will, oftentimes, force an architecture to be far more complex that it really needs to be since there will have to many more components handling data consistency and transformation. Another serious problem is the existence of fundamental data representation differences between applications and middleware. An example of this problem in a media enterprise, would be an application that logs temporal metadata as a list in a text file and the enterprise chooses MXF as its media wrapper format for the infrastructure. This forces an integrator to "manually" parse that text file into the MXF model which is a very complex process. Finally, other common wrapper problem is when it comes the time to upgrade or replace the wrapped applications. The differences between old and new versions can be so great that it forces the integrator to re-write the whole wrapper from scratch.

Even, with these potential problems, all these wrapper models provide a fundamental advantage that is, in the end, it all comes down to an integrator's choice. One might use each model separately or use a combination of two or even the tree models.

2.2.5 Middleware

The *middleware* is also known as *middleware layer*. This last term is more accurate since instead of being just one single component in the enterprise SOA infrastructure, it is a "layer of systems" that are, in fact, the backbone of the SOA infrastructure since it connects everything. Since the middleware is a layer of software one can add more software in order to meet business or technical requirements. So, just like a growing plant, its very important to take care of the middleware layer from the start. The fact that is possible to add more components to the middleware layer, allows it to be extremely customizable. Every one of those components in the middleware layer should provide some sort of unique value and so, before discussing the components of the middleware, one must discuss what is expected of the middleware itself.

The first, and probably most important thing a middleware layer should do is to act as a communication layer between all the services in the infrastructure. The middleware should provide a way to send messages between services back in forth so, when a service drops a message in the middleware it knows how to deliver it to the appropriate service. This is done by implementing routing schemes much like an Ethernet protocol but for business messages instead of bits. This functionality, also known as enterprise messaging, is the fundamental functionality of the middleware layer.

Another feature of the middleware layer that is directly related with the previous feature is the ability to "install" in the middleware layer some messaging "extensions". These

may include extensions that ensure that a message is always delivered even if the recipient is offline (ex. broken). In order to do this, the middleware may use commonly used techniques in other mail delivery systems like providing receipts, attempting delivery several times or even saving the messages for some period of time.

Since all the messages travel through the same channel within the middleware layer, this enables a big feature of the middleware, *data aggregation*. Components that allow data aggregation by "analyzing" the messages that travel the common channel in the middleware, can easily be added. And if the enterprise wants specific messages to be secure from scrutiny while it's on route to its destination, encryption components can be added also. This way, messages can easily be sent securely from service A to service B and, just as easily, messages can be sent from service A to all of the recipients in the infrastructure.

Even though, like it was mentioned in the previous chapter, data transformation is a "wrapper's job" it can also be done, to some extent, by the middleware layer. By transforming data, like changing message formats or correcting message content, the middleware layer makes communication flow more seamlessly.

Messaging related features aren't the only features that a middleware has. The middleware can play an active role in the business processes that ultimately manage message flow between services. Process orchestration is one of the most important features of the middleware layer as it allows it to become the *de facto* technical and business center within an enterprise by providing it with a common place process instances and definitions. All the features presented before all lead to a feature whose importance is only surpassed by the "basic" messaging feature mentioned in the beginning. That feature is data visualization and manipulation. As the middleware includes not only message routing but also process orchestration, data visualization means that, through a portal into the middleware, one might visualize and manage data that can range from errors to process status, also panning through business conditions and rules. The downside to this feature is that it may require a very complex portal in order to view and manipulate all that data.

In order to provide all these features, some components are required as depicted in Figure 2.16. All these features and components that enable them, have actual technologies that implement them. There two major systems that implement all those components just mentioned: the *application server* and the *enterprise service bus* (ESB).

If the middleware layer can be considered the backbone of a SOA, then an application server should be considered the foundation of the middleware layer. Actually, an application server plays a key role not only in SOA but also within every enterprise since an application server's main function is to help build and host applications to its users. Also, many of the additional components just mentioned that extend the middleware's features are, usually, directly plugged to an application server. In a typical three-tier architecture, an application server sits right in the middle tier, the business logic tier, and at the same time, it also connects the storage tier and the presentation tier [FFVM08]. It allows both

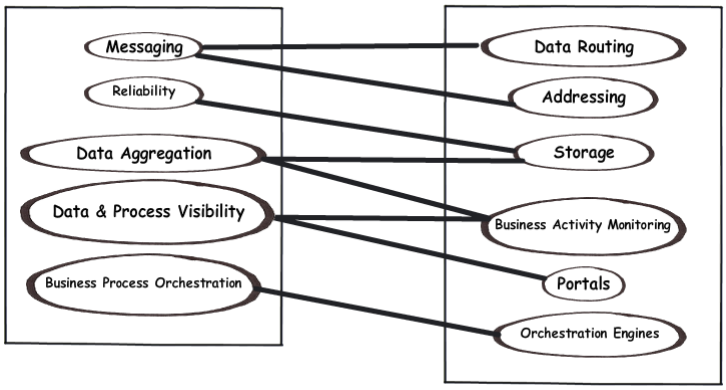

Figure 2.16: Relationship between several middleware layer features and the components that enable them

messages and server state to be securely saved which makes the application server a very reliable system and a fundamental aspect, specially in enterprises in the media business.

Finally, there is one other feature that application servers "lend" to the middleware layer, visibility. From what has just been talked about, especially the central role that application servers play within an enterprise and in the overall SOA, it's easy to see that the an application server, working alone or as a part of a cluster, cam serve as a basis for the portals mentioned earlier that allow business data and processes management/visualization. An application server enables the centralized administration feature in a middleware layer even when one deals with a cluster, since many vendor products already come with some sort of central console that commands and monitors every device in the cluster.

The other major system mentioned was the *enterprise service bus* (ESB). As stated earlier, the application server might serve as the foundation for the middleware, but its real structure lies within the ESB. As stated earlier, some consider the ESB the glue that holds the SOA together by enabling the communications between all the enterprise applications [Ort07]. This means that the ESB provides connectivity between the service requesters and the service providers. Basically, it's a distributed, service-oriented messaging channel providing business communication capabilities between all the different systems connected to the SOA. The ESB also has business-level capabilities as the communication since the communication rules and methods, that deal not only with the format and technology of the messages but also with their content, established in the ESB are applicable in all aspects and departments of an enterprise, from the IT department all the way up to the higher echelons.

The ESB implements this business communication by implementing one of the many available communication models. The several communication models ensure that, for example, messages are sent to many recipients, that a message is delivered in the most extreme conditions and also give the control of what messages one might or might not

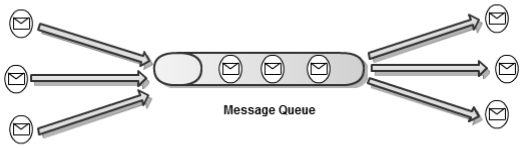

Figure 2.17: Communication Model: Message Queue

receive.

The simplest communication model, as seen in Figure 2.17, is the message queue. This communication model its based on the idea of queue placed "within" the middleware, where a sender system can place a message, that is later retrieved by a recipient system. Also, the message queue can share information with other applications. This means that, since the message queue is an asynchronous communication model, the message queue supports asynchronous communications between components regardless of heterogeneous operating systems and programming languages [YLN00] and, the fact that it's asynchronous, also means that it doesn't matter if the recipient is down/broken or busy when the message is sent as it can be "stored" within the message queue until the recipient is available. Thus the message is guaranteed to reach its destination.

Other important communication model is the publish/subscribe model, seen in Figure 2.18. In this model, the first move is actually done by the recipients as they subscribe to a desired message channel. After subscribing to that particular channel, they'll receive all messages sent to that channel by the sender or publisher. This model, although having the ability to be synchronous, it doesn't have to be so, as it is perfectly possible to persist messages in the middleware. That way, when a message is published, the middleware will store the message before forwarding it to the subscribers. When a subscriber isn't reachable and hasn't received it's message, the middleware will keep on trying to send it.

Finally, another common communication model used by ESB is the process orchestration, as seen in Figure **??**. Process orchestration can be considered as the most important

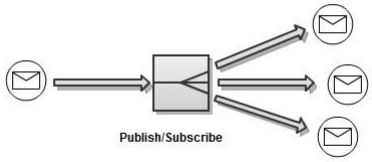

Figure 2.18: Communication Model: Publish/Subscribe

use case in ESB's. In fact, management and execution of business processes in the middleware layer has become one of the most features in a SOA. Process orchestration is based on the idea that the middleware layer can store a workflow dictating the next destination of any given message. A sender just has to release the message in the middleware layer and the process will determine where it will be sent.

Apart from those two major middleware components, application servers and ESB's, there are also some optional components, namely, *business data*, *monitoring*, *business rules* and *identity management* components.

At first sight, the middleware layer might seem such a big and monolithic structure, and a sort of contradiction in the SOA methodology, but after studying it, one can easily see that the fact that the middleware layer is actually composed by several components,it makes it highly customizable which allows an enterprise to actually "tailor" a middleware layer in order to best suit its business needs.

2.3 Protocols

As it was stated, SOA is design methodology not a technology, but one uses a certain technology to implement it. To implement service-oriented architecture, the most used technology nowadays are Web Services. Web Services allow, not only the communication between services, but also allow previously existing services to discover new services. The World Wide Web Consortium (W3C) defines Web Service sin the following way:

> "*A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards*." [Con04]

Web services are designed to bring together distributed and heterogeneous applications in a large scale and provide interoperability between them [YLBM06]. They're an XML-based communication protocol for exchanging messages between loosely coupled systems. In fact, XML is the perfect match for Web Services. Even though XML doesn't directly enable all the characteristics that Web Services require, it does serve as a foundation on which one can correctly build Web Services systems. Figure 2.19 clearly shows this relationship between XML and Web Services.

Web Services and SOA go along very well just because Web Service were first designed with service-orientation in mind. In fact, SOA proposes a Web Services model based on different roles that Web Services play. There are three roles in that model, see in Figure 2.20, namely: the service provider, the service consumer and the service registry.

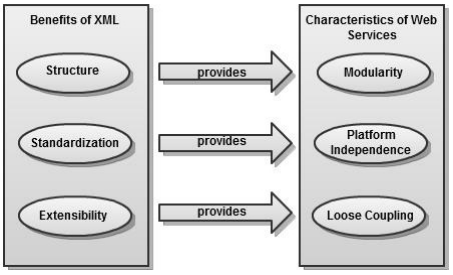

Figure 2.19: How XML benefits enable Web Services characteristics

The way they all interact is, in a way, almost orchestrated since there some steps that have to be taken in order for a service to interact with another. Basically, the steps are:

- **1.** The service provider publishes information about its interface to the service registry.
- **2.** The service consumer then "consults" the service registry to find that interface in the registry.
- **3.** The service binds directly to the service provider.

The service provider is the system that offers a service to the rest of the Web Services network. As for the service consumer, to define this, one must understand its two responsibilities. Firstly, a service consumer should acquire and understand the service provider's interface (using the service registry). Secondly, the service consumer should be able to generate message that conform to that interface in order to correctly contact and bind to the service provider. Finally, the service registry is the "place" where service providers put their interface information and where service consumers locate information about service providers.

All the interactions between the different Web Services are enabled by the Web Services standards. Also in Figure 2.20 one can see the three major standards that enable all the binding/publishing actions of the services namely SOAP, WSDL and UDDI. There's also a fourth standard, that has been gaining relevance in the last few years, REST. This standard is a direct "competitor" to the SOAP standard and it will be studied later on.

So, resuming, and before proceeding to study these protocols in detail, the basic Web Services protocols are:

- **XML** — The standardized data representation format.
- **HTTP** — The standardized data transport protocol.
- **SOAP** — Web Service standard message format.

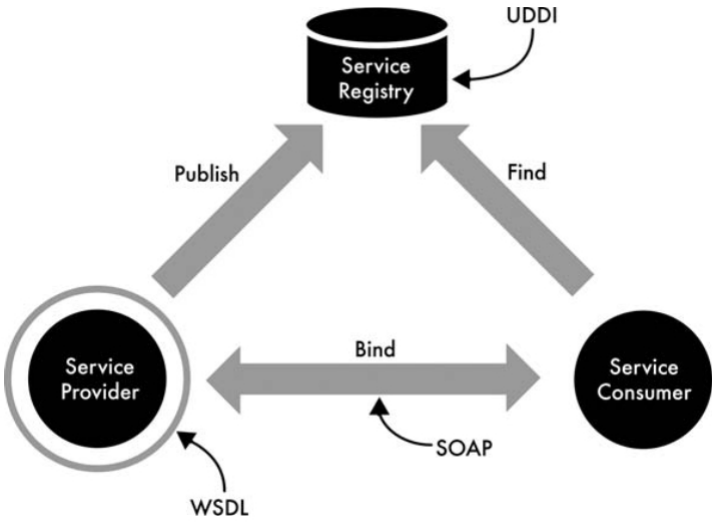

Figure 2.20: The Web Service model [DWBT06]

- **REST** — Web Service standard message format.
- **WSDL** — Web Service standard service description format.
- **WADL** — Web Service standard service description format.
- **UDDI** — Standard that describes a way to implement a service registry in a Web Services infrastructure.

2.3.1 SOAP

Originally the acronym for Simple Object Access Protocol, nowadays, SOAP doesn't mean absolutely nothing. As seen before, SOAP, which is in version 1.2, is a protocol for representing the Web Services messages that go back and forth in a Web Services based SOA. An example of a SOAP message can be seen in Figure 2.21 and the actual W3C definition is:

> "*SOAP is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment.*" [Con07]

Even though it can use several protocols such as SMTP and FTP, SOAP primarily uses HTTP as its transport protocol (*SOAP over HTTP*) or it can use a language-specific remote procedure call (*SOAP RPC*). In the SOAP over HTTP case, which is a "document style", the body of the SOAP message, or envelope, will contain an XML document that matches the input specified by an operation in a service provider's WSDL file. As for the SOAP RPC case, the SOAP body contains a function call which can used to refer

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

Figure 2.21: Example of a SOAP message

to a function written in Java, C#, or some other programming language. This last case, isn't very recommended as it's not very SOA-friendly because is does not promote loose-coupling between the service interface and its implementation.

A SOAP message has two compulsory components and one optional component. The compulsory components are the SOAP envelope and SOAP body components and the optional component is the SOAP header. All XML tags associated with SOAP use the prefix *soap* so, the previously mentioned SOAP components are:

- soap:Envelope
- soap:Header
- soap:Body.

In Figure 2.22 one can see how these components "organize themselves" in the overall SOAP envelope. The soap:Envelope component contains global namespaces in order to avoid conflicts with the names of the elements in the message. Even though an envelope can only contain one message, it can contain several header elements. One of the pieces of information contained in the soap:Envelope component is the SOAP version used to encode the message. That way, the receiver of the message knows if it is capable of handling the message in the displayed version.

The next component is the optional component, soap:Header. When present in a SOAP message, it's the first child element right after the soap:envelope element. The header element is normally used when one wishes to transmit security credentials like a username and password. In fact in the specification clearly defines two attributes, soap:mustUnderstand and soap:actor, which help in the transmitting credentials. The first, soap:mustUnderstand, when defined to the value "1", indicates that an error message should be generated in case the message receiver isn't prepared to handle message headers. As for for the soap:actor element is indicates who are the receivers of that particular message.

The final component is soap:Body. This component clearly indicates the beginning of the message and it's a compulsory as it delivers the payload to the receiver of the message. By payload one refers to all the information that has to be carried until its final destination.

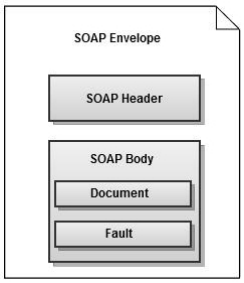

Figure 2.22: A SOAP Envelope example

Obviously, this payload can be anything ranging from a remote method invocation to a simple XML document.

If one looks at Figure 2.22, within the soap:Body component there's one "extra" element not mentioned until now, the soap:fault element. Just like the soap:Header element, soap:fault is optional albeit extremely useful. Useful because the soap:fault element is defined by the SOAP specification as the error processing element in the messages. In such a distributed and heterogeneous environment such as that in which Web Services "live", errors are likely to happen when a remote method is invoked. Common errors are, for example, missing/non-existing methods or lack of parameters. Since errors like these can't be corrected in the remote computer, a message must be sent back to the message sender in order to notify it that the invoke process has failed, and at the same time, providing enough in order to correct the error. The soap:fault was "born" to handle situations like these and the fact that it's an optional element lies in the fact that only response messages would require this element. The soap:fault has, in turn, four elements, namely:

- *soap:faultcode* — It's a specification required element. It contains the code that indicates what or where is the problem. In all here are four possible codes:*server, client, versionMismatch, mustUnderstand*.
- *soap:faultstring* — A text that describes the error represented by the code presented in the soap:faultcode element.
- *soap:faultfactor* — An optional element that informs about the method that was responsible for the fault.
- *soap:faultdetail* — This element contains as much information as it can about the state of the server when the fault occurred.

The SOAP protocol wasn't developed with every possible option of possibility in mind and that's why some additions have been made to the SOAP standard. Until now one always assumed that two services only needed to exchange simple data in an XML representation, but they can also need to exchange things like security keys or even .JPEG

images. However, one should be careful not to attach large files to the messages as it will surely slow down the relay time. A solution for this is including in the SOAP message, information where a service can "get" a certain attachment.

There are also two common security-related additions to the SOAP standard. In large enterprises where all kinds of sensitive data exists and goes back and forth security can, and it usually is, a big issue. SOAP itself doesn't include a method to encrypt messages as this is done by the XML Encryption standard. In SOAP messages, the usual procedure is encrypting the body and leaving the header, along with useful and necessary routing information, in plaintext. Encryption can also done at the XML element level which means that a service consumer/provider can selectively choose to encrypt any element at the sub-document level. Also, this standard allows several encryption format like the 128-bit AES encryption.

The second security-related standard is the XML Signature. This standard is used to to authenticate a message from a service provider. An XML signature consists of an encrypted signature element and a key to authenticate the signature. And, just like the XML Encryption standard, it supports a number of encryption standards and can also authenticate an entire message or just a part of it.

2.3.2 REST

REST, which stands for *REpresentational State Transfer*, as mentioned before can be considered a direct competitor of the SOAP standard as both are standards that define the communication between services even though they rely on different paradigms. REST is an architectural style based on the HTTP protocol that describes a navigational, resource-oriented style of design [FT02]. The first edition of REST was developed between October 1994 and August 1995, as a means for communicating Web concepts while developing the HTTP/1.0 specification and the initial HTTP/1.1 proposal. Actually, REST is so "connected" to HTTP that is also known as the "HTTP object model".

Web services that adopt the REST "style" are known as RESTful services and are nothing more than simple resource-oriented lightweight web services that comply with REST design principles [LK10]. The REST architectural style defines everything on the Web as a resource that may have different representations such as HTML, JSON or XML and that is addressed by a URI which is built according to a uniform scheme of */resource/id/verb*. Resources support simple verbs like Create, Read, Update, Delete (these are known as CRUD). The *verb* is sent to the URI as an HTTP method, namely, POST (create), GET (read), PUT (update) or DELETE (Delete) to realize the requested CRUD function on the resource identified by *id*.

REST isn't a standard per se, but a set of constraints to be used in the construction of distributed web applications. As a high-level concept, REST is not attached to a specific

technology or implementation just like SOAP [GS09]. REST's constraints are [FT02]:

- **Client-Server Constraint** — Clients are separated from servers by a uniform interface. Separating the user interface concerns from the data storage concerns improves the portability of the user interface across multiple platforms and improves scalability by simplifying the server components.
- **State Constraint** — Communication between client and server must be stateless. This means that each request from a client should contain all the information necessary for the service to correctly process the request. This also means that all aspect regarding session state are stored in the client as the server stores no session state whatsoever.
- **Cache Constraint** — By cache constraints one means that the data within a response to a request has to be, implicitly or explicitly, labeled as cacheable or non-cacheable. This means that a client can locally cache a representation of a resource which greatly boosts an application performance.
- **Layered System Constraint** — This constraint refers to the fact that, within an architecture composed of hierarchical layers, each components behavior is "controlled" in such a way that each component cannot "see" beyond the immediate layer with which they are interacting.
- **Code-On-Demand (COD)** — This constraint allows a client's functionality to be extended by downloading and executing code in the form of applets or scripts, which greatly simplifies clients by reducing the number of features required to be implemented beforehand.
- **Uniform Interface Constraint** — REST places a special emphasis on a uniform interface between components which improves interaction visibility and simplifies the overall architecture. It also means that the components can evolve independently from each other.

Unlike SOAP, XML markup which is quite abundant in SOAP messages, in a RESTful web service there is none XML markup whatsoever since it directly uses HTTP as the invocation protocol and this avoids unnecessary XML. The response is a representation of the resource itself, and does not involve any extra encapsulation or envelopes. As a result, RESTful web services are much easier to develop and consume than SOAP-base ones. URIs and the representation of resources are self-descriptive and make RESTful web services easily accessible. Just like SOAP, REST also has a description language, namely, WADL [Had09] (*Web Application Description Language*). It describes the resource in simpler way that its "SOAP counterpart", WSDL. WSDL and WADL will be studied in greater detail in the following subsections.

2.3.3 WSDL

WSDL stands for Web Service Description Language. It can be considered the most important of the three major Web Services standards (SOAP, WSDL, UDDI) because it's the WSDL's job to communicate all of the critical information regarding a specific Web service. A service's WSDL document (an example of a WSDL file can be seen in Figure 2.23) is designed to communicate information about how to connect with the service it describes, but that's not the only piece of information it describes. A WSDL document is designed to provide the service consumer with all the minimum information possible to allow the consumer to effectively engage in service communication. This is information is:

- What are the operations available in a service.
- What data these operations require.
- What data these operations return.
- Where and how to connect to the service provider.

WSDL files have specific elements that transmit all of this information. In fact, WSDL file's structure can be divided into two major parts: the abstract descriptions and the concrete descriptions. There are four abstract XML elements that can be described in a WSDL file, namely:

- <*wsdl:types*>
- <*wsdl:message*>
- <*wsdl:porType*>
- <*wsdl:operation*>

These elements data and operations supported by the Web service. As for the concrete descriptions:

- <*wsdl:service*>
- <*wsdl:port*>
- <*wsdl:binding*>
- <*wsdl:operation*>

```
- <wsdl:definitions targetNamespace="urn:oasis:names:tc:wsrp:v1:wsdl">
    <wsdl:import namespace="urn:oasis:names:tc:wsrp:v1:bind"
    location="http://www.oasis-open.org/committees/wsrp/specifications/version1/wsrp_v1_bindings.wsdl"/>
    <wsdl:import namespace="urn:bea:wsrp:ext:v1:bind" location="wlp_wsrp_v1_bindings.wsdl"/>
  - <wsdl:service name="WSRPService">
    - <wsdl:port name="WSRPBaseService" binding="urn:WSRP_v1_Markup_Binding_SOAP">
        <soap:address location="http://localhost:7001/strutsHello/producer"/>
      </wsdl:port>
    - <wsdl:port name="WSRPServiceDescriptionService"
      binding="urn:WSRP_v1_ServiceDescription_Binding_SOAP">
        <soap:address location="http://localhost:7001/strutsHello/producer"/>
      </wsdl:port>
    - <wsdl:port name="WLP_WSRP_Ext_Service" binding="urn1:WLP_WSRP_v1_Markup_Ext_Binding_SOAP">
        <soap:address location="http://localhost:7001/strutsHello/producer"/>
      </wsdl:port>
    </wsdl:service>
  </wsdl:definitions>
```

Figure 2.23: A service's WSDL document

These elements describe how a consumer can connect to the Web service and are placed as children after the root element, *definitions*, by the presented order. Also, apart from these elements a WSDL document may contain SOAP messages and/or other XML elements.

The *types* elements define the data structured used in the Web Service. As for the *message* elements, these define the format of the data exchanged between client and server, or in the other words the format of the inputs and outputs, by using the structures defined in the *types* element. The elements *porType* define the structure of the methods by using the *operation* elements which, in turn, define the names of the methods and gather all kinds of messages between client and server through parameters define in the *message* elements. The *binding* elements establish the connection between the methods structure, represented by the *porType* elements, with concrete actions that the Web service supports. Finally, the *service* element defines the exact location of the Web service, as well as establishing connections with the operations represented by the *binding* elements.

2.3.4 WADL

Web Application Description Language, or WADL, is an XML-based file format that provides a machine-readable description of HTTP-based web applications [PML09]. In its article [Had06], Marc Hadley, describes WADL as a language meant to describe the interface of a service on the web with the following intents:

- Provide support for the development of modeling and visualization tools.
- Support the generation of stub and skeleton code.
- Provide a common piece of configuration for client and server.

A WADL consists of the following elements:

```
<application xmlns="http://research.sun.com/wadl/2006/10">
  <doc jersey:generatedBy="Jersey: 1.0.1 12/01/2008 01:44 PM" xmlns:jersey="http://jersey.dev.java.net/"/>
  <resources base="/helloworld/rest">
    <resource path="/HelloService">
      <resource path="/getGreeting">
        <method name="GET" id="getGreeting">
          <request>
            <param type="xs:string" style="query" name="userName" xmlns:xs="http://www.w3.org/2001/XMLSchema"/>
          </request>
          <response>
            <representation mediaType="*/*"/>
          </response>
        </method>
      </resource>
      <resource path="/createInstanceAndReturnReference">
        <method name="GET" id="createInstanceAndReturnReference">
          <response>
            <representation mediaType="*/*"/>
          </response>
        </method>
      </resource>
    </resource>
  </resources>
</application>
```

Figure 2.24: A web application's WSDL document

- **Grammar** — This includes all external schemas for data-types, which is similar to types in WSDL.
- **Resource** — This references the base path for accessing a tree of resources.

In Figure 2.23 one can see and exmaple of the structure and contents of a WADL document.

2.3.5 XML and HTTP

Of all the Web Service protocols studied so far all of them had one thing in common: they're XML-based! That means that XML, *eXtensible Markup Language*, is a very important element in the whole Web Service standards. XML was born out of the need of exchanging information between computers. The easiest way is, obviously, text but a computer doesn't have a way of knowing what the text means and that's where XML comes in to play. XML provides a way to describe and structure data in order to be exchanged between computers. An XML document is composed by data that's "surrounded" by tags. These tags "dictate" what the data that they surround means.

The XML specification actually doesn't define tags, or any other element for that matter. All the tags are user generated according to its needs. If a user has a need to describe an address most likely he will create the <*name*>, <*street*>, <*city*> and <*country*> tags in order to correctly describe the address. So, all in all, XML has the following characteristics:

- It separates content and format.
- Readable by humans and machines.
- Limitless tags creation.
- Allows the creation of structure validation documents.

- Allows to connect distinct databases.

The main components that compose the XML standard are:

- *XML Document* — a file that complies with the rules of the XML standard.
- *XML Parser* — software that interprets the content of the XML document by transforming the document into data structures that can easily be processed.
- *Document Type Definition (DTD)* — the tag structure allowed within an XML file and the relation between them. It allows validation of an XML file.
- *XML Schema* — the tag structure allowed in an XML file and the relation between them. The validation of the XML document occurs by comparing it with the corresponding XML schema.
- *Namespaces* — an exclusive name that should be used in order to avoid conflicts between tags names.

"Going down the protocol ladder" one arrives at the protocol that, in a way, transports all the other protocols, HTTP (or HyperText Transfer Protocol). All the transfers that occur on the Web are based on a client-server architecture and on the server side there is a specific software that just "sits and waits" for any request from the client. These are know as *web servers*.

As it was already stated, HTTP isn't the only transport protocol supported by Web Services. Actually there are several other protocol which can be used such as message queues, SMTP, FTP and instant messaging.

2.3.6 UDDI

UDDI, or Universal Discovery, Description and Integration, is a standard that describes a way to implement a service registry in a Web Services architecture. In fact, UDDI consists in a series of public directories that can be sustained by any enterprise.Since every public directory replicates other public directories information, UDDI can be seen as one big replicated collection of Web Services. UDDI can be described as a transparent, normalized mechanism used to describe services, specifying a central service registry whilst also describing simple methods in order to call a service.

The UDDI registries are based on a confederated model as they copy each other registries. Even though the registries might be physically distant, they're logically centralized as they're periodically synchronized. The informations on a UDDI registry can be classified in three main categories, namely:

- **White pages** — include general information about a company, such as contacts, business description, address, etc...

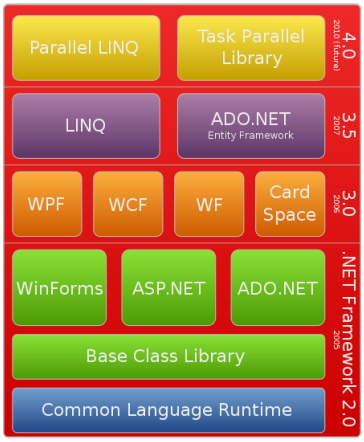

Figure 2.25: The .NET Framework

- **Yellow pages** — include general classification data to every company or available service. This identifying an industry to which a certain company belongs or a product a company manufactures.
- **Green pages** — contain technical information about a service on the Web. That usually includes an address to invoke the service.

Another characteristic of UDDI is that is "built" using the same standards as SOAP which means that the act of looking up a service in a registry is completely "programming language independent", hardware independent and environment independent. Finally, it's important to clarify that UDDI registries are accessed exactly like any other regular Web Service.

2.4 Framework

2.4.1 Microsoft .NET

The .NET framework, which can be seen in Figure 2.25, is Microsoft's approach to web services. It runs on a single platform, namely Windows even though it natively supports several programming languages like VB.NET and C#. This last one, however, is the most widely used in order to implement Web Services. Also, the development environment of choice is Microsoft Visual Studio has tools specially designed to Web Services development. According to the official website [Mic10], the .NET framework has been built with the following objectives in mind:

- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.

- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.
- To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

The .NET Framework has two main components: the common language runtime and the .NET Framework class library. The first, the common language runtime, can be considered as the foundation of the .NET Framework. It manages code at execution time, providing core services such as memory management, thread management, and remoting, while also enforcing strict type safety and other forms of code accuracy. As for the class library, it's a comprehensive, object-oriented collection of reusable types that one can use to develop applications ranging from traditional applications (command-line or GUI) to XML Web services.

2.5 Conclusion

After reviewing all that was studied so far, one should starting questioning if the whole SOA methodology and even its most commonly used implementation technology, Web Services, are really "perfect". As everything else in life, both SOA and Web Services have their share of problems.

Starting with SOA, the first thing one notices is its shear "size" and complexity. This means that implementing it requires a lot of planning beforehand which, in a big enterprise, usually means not only lots of time but also lots of money. Implementing a SOA shouldn't be done with immediate gains in mind since its main benefits, already mentioned, only start to really "appear" in the long run. A way to prevent errors/difficulties while implementing a SOA is to start small. It's a very bad idea to try and implement "all at once". On the other, the downside of implementing a SOA step-by-step is that it takes a long time to fully implement a SOA to a point where it can start to "pay off" and when a company starts to really feel its benefits. Other SOA disadvantages were found [WL09], namely:

- A big problem of SOA is organization, culture and politics as people usually are unwilling to accept change.
- Core problem of SOA is control, quality and management as it is destined to fail without control.
- No two SOA implementations are alike since different businesses require different implementations.

Services also provide some challenges, namely, when one tries to define them. Correctly defining services can be a very difficult task. Since the fundamental characteristic of a service is its business value, if one defines a service too "low-level" is has little or no business value at all and if a service is "all-encompassing" it has a low re-usability rate, something that goes against SOA's principles. This makes the act of defining services/decomposing applications something quite difficult to master.

As far as middleware goes, it's not entirely free from "criticism" either since it poses some challenges. When talking about disadvantages of middleware one should differentiate its two most important components: applications servers and ESB's. The application servers, even though they're a proven technology, they are extremely complex systems which means that they're not something that can be just plugged in and start working immediately. They require fine tunning and maintenance in order to correctly perform and being as complex as they are, it will require a fair amount of time and training. The ESBs are equally complex and installing them only adds another layer to the already stacked application server. This means, added complexity and probably a steeper learning curve to any IT administrator. Finally, middleware as whole has one big disadvantage, especially in the context of a media enterprise: middleware is terrible at handling media! Middleware technologies were designed to carry messages around, not big media files. Obviously, this is a big problem for media enterprises and forces them to find ways of dealing with this problem. Normally this is solved by having media transfered on an bus external to the middleware layer.

Web Services also have their share of disadvantages. A first problem with Web Services would be their most commonly used transport protocol, HTTP. This protocol doesn't guarantee delivery of the messages which obviously, can pose a problem. In order to solve this, Web Services have to use other kinds of transport protocols. Web Services might also have performance issues mainly because HTTP is based on a request/response method which makes the connection between client and server non-persistent. This means that a web service wastes a lot of time just reconnecting to web servers. Also, since Web Services use the Web's infrastructure, which means that a certain location in the network might be not be always accessible just like a normal website on the Internet can, sometimes, be "down".

Even "within" Web Services protocols one can find some advantages and disadvantages between them. The most notorious protocol "pro and cons discussions" would be the SOAP vs. REST discussion and WSDL vs WADL.

Even though REST and SOAP can "work side-by-side", the REST vs SOAP "dispute" is one that is here to stay as the two architectural styles greatly differ even though both have their share of disadvantages. One of the most used arguments in favor of the REST style is that it's lightweight especially when compared to SOAP. This happens because REST, unlike SOAP, doesn't have that extra XML markup in its messages which also makes REST more straightforward than SOAP. REST also has some disadvantages and some even are "born" from some of its advantages. One good example of this is the fact that REST is stateless which means, like it was already mentioned, that it's the clients "job" to store a session state. This improves visibility, reliability, and scalability but it also has its drawbacks, namely the fact that this decreases network efficiency by increasing repetitive that is sent by the client. This lead to, already mentioned, cache constraint which also lead to another issue, namely, the fact that cached data decreases reliability if the stored data differs from the data that one would obtain if the request had been sent to the server. Another current argument in favor of SOAP is that, since it has such widespread use, there is much more support behind it. This means that, even though REST is starting to get used more by big companies (like Twitter's API [Twi10]), it's still not a well established technology, especially when comparing it with SOAP.

This also happens with WADL and WSDL. Since the latter has been around longer there's is also a widespread support and any major framework (like Microsoft's Visual Studio) has extensive support. WADL, much like REST aims to be simpler than SOAP, aims to be a simpler alternative to WSDL but it doesn't have such and extensive support which slows down its adoption. Version 2.0 of WSDL also supports REST, but until WADL rose to the scene, there was no real way to describe RESTful services. WADL and WSDL 2.0 each have their respective pros and cons. WADL is simple, but limited to describing HTTP applications. On the other hand, WSDL 2.0 is more feature rich, yet still lacks a true resource-centric model.

Chapter 3

Project Specification

3.1 The Media Business

Before knowing how the newsroom environment actually works, one must know what the media industry really is and how does it do its business. Media is the plural of the word *medium* which in Latin means means or instruments [Dic10]. The media industry (in this particular case the main focus is on broadcasters) can be seen, in fact, as an instrument for various things such as broadcasting information through things like news bulletins or just to entertain through movies or TV shows. However, the way the media and entertainment industry "works" has been changing.

Like it was mentioned at the beginning of Chapter 1, for some years now, the media industry has seen a migration from the old physical media to digital media as the main medium for storage and production. This has been caused, partly, by the appearance of new media forms that have spun thanks to the Internet and, therefore, changing the paradigm through which people access media content. This transition to digital provides benefits such as an increase in effectiveness and efficiency of the business processes and, also, a reduction in travel and administrative expenses [LDG+09]. On the other hand, this new medium meant a restructuring of the IT infrastructure of media enterprises everywhere, to accommodate the new file-based media systems. In Figure 3.1 one can see what a traditional media system would look like a decade ago. Thanks to the new digital media, a media system nowadays, most certainly looks like what is shown in Figure 3.2.

In their book, *The Service-Oriented Media Enterprise: SOA, BPM and Web Services in Professional Media Systems*, Footen and Faust consider the media enterprise as a craft-oriented business because it involves a certain amount of creativity in some of its processes. The typical media enterprise is also highly collaborative, widely dispersed and, has a "high resistance to change" since people in the media business have been doing

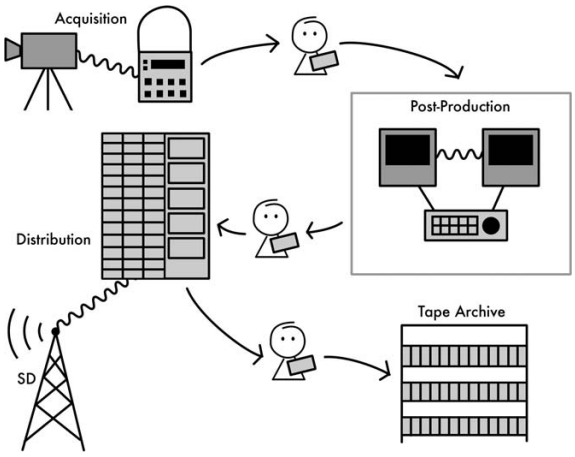

Figure 3.1: A media system a decade ago [DWBT06]

things a certain way for a long time. Since there is a fairly high amount of creativity and innovation in media processes, implementing a SOA will pose a challenge because there will be parts of the enterprise that will clearly benefit from the service/business orchestration, studied in greater detail in Chapter 2, that the SOA accomplishes and other are very "humanized" and, as such, won't require any orchestration at all. This is particularly true in a newsroom environment has it will be described in greater detail in the next subsection.

Time is money. Although an old cliché, it's especially true when it comes to the media business and especially if one thinks that for just one second lost due to server problems, a TV station can lose millions in revenue. This makes using SOA in television systems something not to be taken lightly. This is just of some "unique concerns" that the media industry has. But not only reliability is seen as something fundamental, quality is also seen as something crucial in this business. In a media enterprise not only reliability when delivering the content must assured, it's quality must also be assured and maintained throughout its "path" through the media facility's IT infrastructure. Usually this path is punctuated by the occasional transcoding needed so that the asset is "processed". Since an asset can go through various acts of transcoding before reaching its "final destination", it's not unusual to witness a loss in the asset's quality. In Figure 3.3 it's possible to see the differences in quality of a picture that has been handled hundreds of times and has suffered a severe quality degradation, something that is called in the media business as *generational loss*. Quality is a subject just as sensitive as time in a media enterprise since the media content must, not only, be delivered uninterrupted but also with perfect picture quality according to the format that is used. This means that one must be careful when using a certain asset since it can be in the wrong format/quality, especially when considering the business processes automation that a SOA can provide.

The media business has a series of unique challenges that set it apart from the rest of

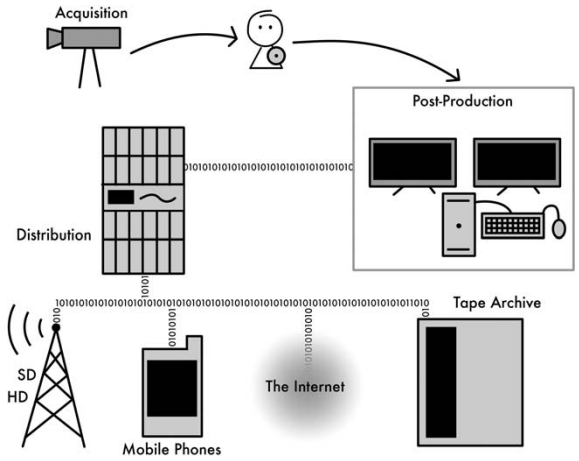

Figure 3.2: A typical media system nowadays [DWBT06]

the rest of the business world. If, on one hand, it has the same challenges of many IT based businesses out there, on the other hand, it also has a set a particularities that pose a challenge to the more conventional approaches. All this blends and creates a unique industry, the media industry. But, as it was already stated in the beginning of this report, within the media business there's an environment that might just be a paradigm of, not only all the challenges a media business faces, but also of just how dynamic and volatile a media business can be. That particular environment is the newsroom environment and it will be described in detail in the next section.

3.1.1 The Newsroom Workflow

As already mentioned, due to the unpredictable nature of news, a newsroom can be (and usually is) an extremely dynamic environment. To better assess how a typical newsroom works and how that workflow can be improved, the newsroom in RTP's [RTP11] Porto headquarters was studied. Typically, what happens nowadays is that a newsroom has paid access to several news agencies's news feeds. Each news agency provides a broadcaster with a "box", henceforth named *basket*, where these news feeds are placed in what might or bot be regular intervals. These feeds are nothing more than a media file containing one or more news clip and some associated metadata that describes the content of the media file. The pair composed by the video file plus the metadata file will, henceforth, be known as *asset*.

As stated in the beginning of this report, currently, the task journalist undergo when searching for assets can be quite cumbersome as they have to individually search within each news agency website. The process in detail can be summarized in these steps:

- **1.** A journalist accesses a news agency website where it searches for all assets regarding a particular subject.

Figure 3.3: Example of generational loss

- **2.** For each asset that the journalist finds relevant to its work, he has to take note of the reference of each asset.
- **3.** After repeating steps 1 and 2 for each news agency it wishes to search, a journalist opens a video editor (in RTP's case, Quantel's sQ Edit and sQ Cut [Qua11] are used) and using the references it took note in step 2 and manually searches the different baskets for the asset in order to import it to the editor.
- **4.** After the editing is done, a journalist can do one of two things: save the finished news piece in a storage server or it can "send it" to a specific news production software like ENPS [Pre11] (Essential News Production System).

It's easy to see how this series of steps can become quite cumbersome with just a couple of baskets, but the usual "scenario" is a newsroom with access to several baskets.

3.2 newsRail

From what was just described one can easily see that there's improvements that can be made in a typical newsroom workflow and this is where newsRail comes into play. The newsRail news selection system has been idealized as a product that centralizes all the news selection process. Within the workflow mentioned in the previous section, newsRail "sits" between the journalists and their sources, namely, the baskets (steps. In that sense, newsRail, acts as a gateway through which journalists can search and preview assets as well as visualize the metadata associated with them, without any regard to the basket whence that particular asset came from. On an infrastructural level, newsRail can easily be integrated into an existing infrastructure, as newsRails SOAP and REST interfaces

make a SOA-compliant system. This means that an integrator, even though newsRail is mostly meant to be used through its GUI, can use its interfaces to create a customized wrapper and connect newsRail to an existing infrastructure, preferably a SOA.

In order to perform some of its tasks, newsRail is aided by and external transcoder that is commonly used in the broadcasting community, Rhozet's Carbon Coder [Rho11b]. This means that newsRail doesn't need to directly handle the assets, or at lest their video component, and only needs to issue orders to Carbon Coder when a transcode is required. Over the course of the next sections, the Carbon Coder transcoding system as well as all of the other components that make up the newsRail system will be described in greater detail.

3.2.1 Requisites

In order to achieve its goals, namely, facilitate journalists news selection process, newsRail must "hide" all the, already mentioned, cumbersome tasks that journalists must undergo everyday in order to select the news that are relevant to them. With that in mind, one can define newsRail three main functionalities as being:

- **Automatic Discovery & Proxy/Keyframe Generation**
- **Browsing**
- **User-Requested Ingest to Destination**

The first functionality, *Automatic Discovery & Proxy/Keyframe Generation*, means that the newsRail system, after being correctly configured, can connect to any number of baskets and "listen" to any new asset that arrives in each and every one of them which, consequently, will trigger newsRail to issue an order to Carbon Coder to generate a proxy version of the asset's video component and also extract a keyframe from the same file. A proxy is a lower resolution (and smaller) version of the original media file, mainly suited for video previews and edits. The specific contents of the message that is sent to Carbon Coder will be described in detail in the next chapter (Chapter 4), but basically, newsRail sends Carbon Coder three vital pieces of information, namely:

- the location of the asset (and, consequently, its video component);
- what Carbon Coder should do with it (in this case, generating a proxy version of the video component and extracting a keyframe from the same component);
- the location where the proxy and keyframe should be sent to.

Even though newsRail is originally thought out to have profiles and different "types" of users, in its current implementation that feature is not supported (see following chapters

for further explanations) and so there is only one user, namely the administrator. The administrator is supposed to see all this process taking place in order to supervise but, to "normal" users, this process is completely "silent" and all they see are the assets, in the GUI's asset list, after every step in this process is complete. The proxy and keyframe that were generated will then be used in the GUI and presented to the users. How that is done, however, will not be discussed as it steers away from the context of this dissertation which, as it was already mentioned before, focuses only on developing the newsRail system middleware layer with particular emphasis on defining and implementing its interfaces. Also, upon a new asset detection, newsRail inserts into its database (DB) a series of elements regarding the newly discovered asset. Some are read from the asset's metadata component while other are "generated" at the time the data is inserted in the DB (for example, current date and time). This process, unlike the proxy/keyframe generation process, is completely hidden from every user regardless being administrator or not. The asset data that is stored will described in detail in the next chapter.

The asset related data that is stored leads to the second mentioned functionality, *Browsing*. As already mentioned, currently, journalists have to individually search each basket for assets they find relevant. As assets "fall" into the baskets and newsRail detects them, it will also populate its DB. This amount of data is extremely important for two particular reasons, firstly because some of that data comes directly from the asset's metadata and it will be presented to the users in the GUI, secondly because that data will also provide the means through which the users may browse the assets. This means that, instead of having to go through all the assets in the asset list (which can quickly grow in size), a user can search, filter and order the assets in order to quickly exclude the assets that are irrelevant to them. Users can search by keyword and order by ascending or descending date. Again, more details about this will discussed over the next chapter.

Finally, the third main functionality, *User-Requested Ingest to Destination*, covers the part of journalists workflow where, after they have selected the assets that interest them, they individually export the assets to a video editor of choice. The newsRail system also facilitates this aspect of the workflow since it allows users to simultaneously select multiple assets and export them to a previously configured F1000's ingest system hot folder. This functionality streamlines the already mentioned "asset export process" in different ways because not only it allows the users to export multiple assets at the same time but also because it saves the users the troublesome task of having to transcode different assets from different baskets as these always have different formats. Again, to perform the transcoding, newsRail is aided by Carbon Coder. In a similar way to it was done when generating the proxy, newsRail issues an order to the Carbon Coder so that it transcodes the original asset's video component into an appropriate format so that the F1000 system can ingest the resulting media file and it also "tells" Carbon Coder where to place that file (namely, into a pre-configured F1000 hot folder).

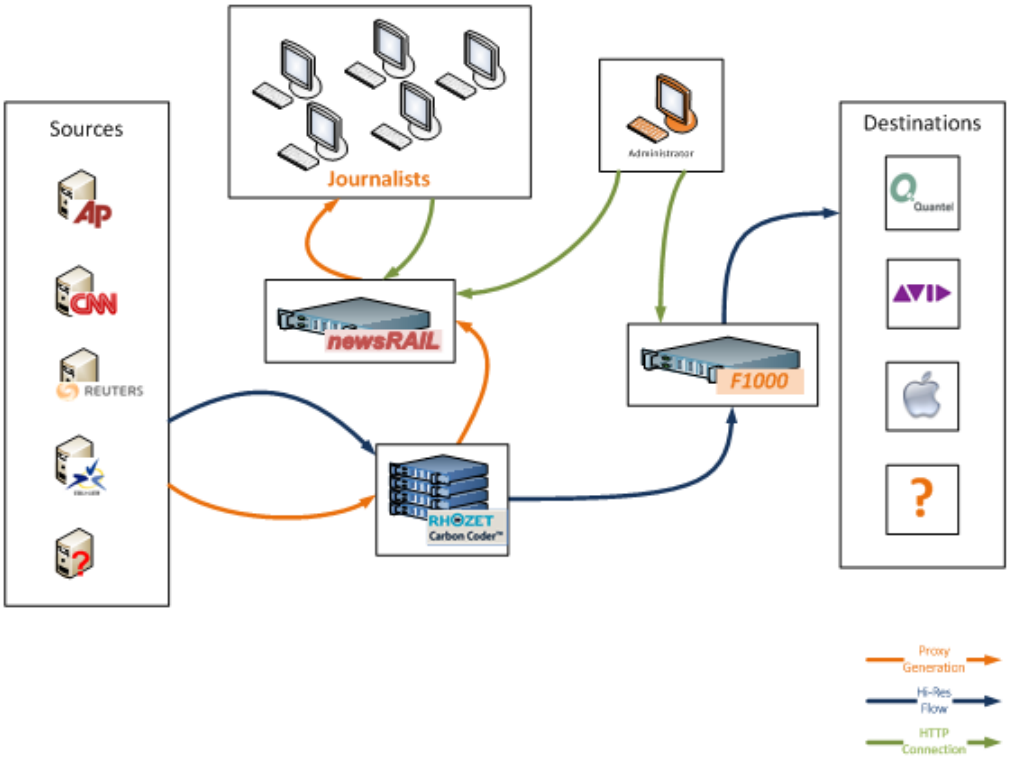

Figure 3.4: The complete newsRail workflow

In Figure 3.4 one can see the complete newsRail workflow. This particular workflow is the final idealization of what the newsRail system will be. One should bear in mind that in its current state of implementation, newsRail does not support user profiles and there is only an administrator account that has access to all the newsRail functionalities. Apart from that, every aspect of that workflow has been implemented and it will be discussed later on. In that diagram one can see that the users connect to the newsRail via HTTP connection, which means that, even though a user can obviously access the system locally, the most common usage should be accessing it through a web browser. Also, the connections are meant to show the "flow" of data within the worflow, namely, the different paths that the proxy and high-resolution versions of the asset take.

And how is a complete workflow processed from start to finish? The newsRail system's workflow can be summarized in a series of steps, namely:

- After correctly configured, newsRail will connect to every basket that has been previously configured and will start "listening" to new assets that might appear.
- If a new asset appears (a new asset is every asset detected by newsRail that doesn't have a DB entry) then the newsRail system will automatically read the assets metadata and store it and, shortly after, will issue an order to the Carbon Coder to generate a proxy version and extract a keyframe from the assets video component.

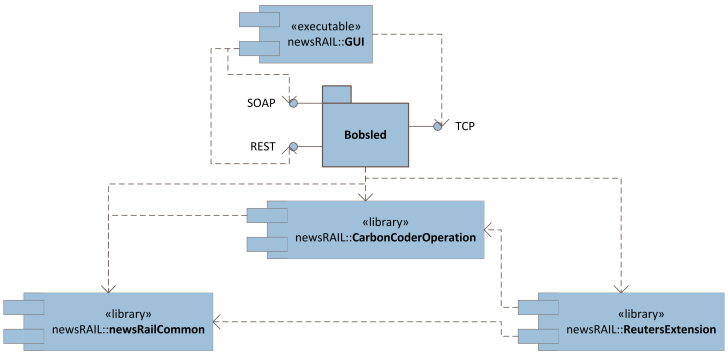

Figure 3.5: The newsRail architecture

- After the previous process terminated successfully, then the assets will be accessible to the users. By accessible one means that they'll be shown to the user in the GUI's asset list. In fact what the users will have access to is the proxy (for previewing the asset), the keyframe and the metadata associated with the asset.
- After a user has selected at least one asset that it wishes to export and issues that command, the newsRail system will again issue and order to the Carbon Coder in order to generate an high-resolution version (MXF IMX format) of the the assets video component and send it to a F1000 hot folder so that the file is ingested.

In the next section the overall architecture of the newsRail will be studied in order to understand what "lies beneath" and how it all works together.

3.2.2 Architecture

The first aspect of the newsRail development, and consequently its architecture, is that is was developed using as its foundation a framework developed by MOG Solutions, *Bobsled*. This framework and its architecture will be studied in detail in the next section but in a very succinct way, one can say that Bobsled has provides a set of concepts/tools, namely, Rules, Flows and Operations, that enable a developer to create any number of customized workflows using those three concepts. This is done by loading extensions, or source code files that contain specific implementations of those concepts. So, again, in a very succinct way a system that is developed using Bobsled as its foundation is, at its core, "just" Bobsled plus a set of extensions. And that is what newsRail really is, a modified version of Bobsled (see next section for details) as its foundation plus a set of extensions that implement the newsRail's already mentioned functionalities.

In Figure 3.5 one can see the newsRail architecture and the main "components" that make up the system. Those components are:

- **Bobsled**

- **newsRailCommon**
- **ReutersExtension**
- **CarbonCoderOperation**
- **GUI** (this component will not be mentioned/studied as it is not included in this dissertation's context)

The first component, *Bobsled*, as already mentioned is the framework upon which newsRail was built. Next there are a set of extensions, namely, *newsRailCommon* and *ReutersExtension*. The *CarbonCoderOperation* component, isn't really an extension per se, but instead an implementation of one the concepts introduced by Bobsled, namely, Operation. Bobsled loads all these three components when booting up, even though different components within Bobsled itself will actually "use" them as it will be shown in over the course of the following chapters

Bobsled, despite having numerous advantages that will be discussed in the next section, places a sort of restraint on how one designs an architecture "around it" (like any framwework, actually), which isn't, necessarily, a disadvantage. What this does is "force" developers to keep in mind that, when working with the Bobsled framework, one must always look at software as a set of extensions (basically a set of DLL files) that will contain actual implementations of Rules, Flows and Operations that will, in the end, be "loaded" into Bobsled. When designing newsRail's architecture that was, obviously, kept in mind and it was decided that each basket that newsRail would support would also correspond to an extension. In its current implementation, newsRail only supports one kind of basket, Reuters, hence the ReutersExtension.

The newsRailCommon is also an extension but it's a sort of "special" extension as it doesn't represent any kind of basket, what it does is contain all the components that are common to newsRail. In theory, this didn't even really have to be considered or implemented as an extension since it could have been considered a simple library common to every basket extension, but for reasons that will be discussed later on, it was decided that newsRailCommon would also be implemented as another Bobsled extension.

3.2.2.1 Bobsled

Bobsled, as mentioned, is a framework developed by MOG Solutions to be used as a basis for all its products. It introduces a set of concepts that should be implemented by extensions that are loaded by Bobsled (when booting up). Bobsled's architecture is shown in Figure 3.6.

From the diagram one can clearly that there are three main components:

- **Core Service**

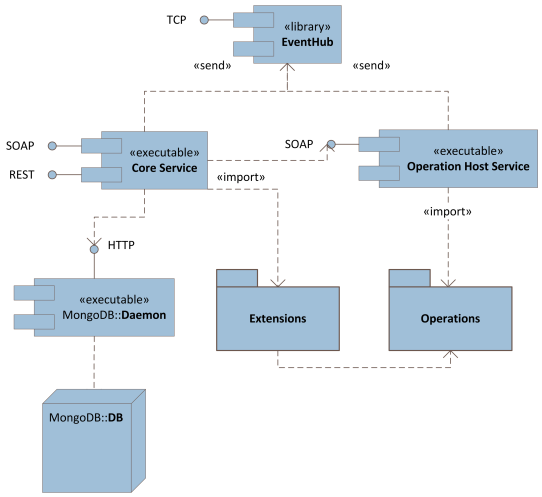

Figure 3.6: The Bobsled architecture

- **Operation Host Service**

- **EventHub** (even though this one is a "special" case)

The Core Service, or just Core, contains all the components that implement the base definitions of Rules and Flows and it also contains all the interfaces and its implementations. As already mentioned, Bobsled was subjected to some modifications in order to accommodate REST interfaces along side the "old" SOAP interfaces and be able to load customized interfaces as it would any regular extension. The details of how this was done are left for the next chapter, but in Figure 3.7 one can see the result of those modifications and how they affect the architecture. The ability to load customized interfaces doesn't change much of the architecture, but, the REST interfaces are clearly seen in the diagram. The previously existing SOAP interfaces are paired with their REST equivalent. In order to maintain a certain degree of consistency the names of the interfaces were maintained but, in order to differentiate the SOAP from the REST interfaces, the latter have the prefix "IWeb" appended in their name.

Also in that diagram, it's visible how the concepts of Rule and Flow fit in the overall Bobsled Core architecture. The actual interactions and how they are processed will be discussed in detail in the next section but for know one should know that a Rule uses a Detector in order to "listen" to new assets and when a new asset is discovered, an event is sent (see EvenHub description) that triggers that particular Rule which will, in turn, start a Flow which is nothing more than a class that encapsulates a state-machine that specifies how operations collaborate in order to perform a task. The Flow then uses the Dispatcher in order to contact the Operation Host Service to start the required operations to perform the task.

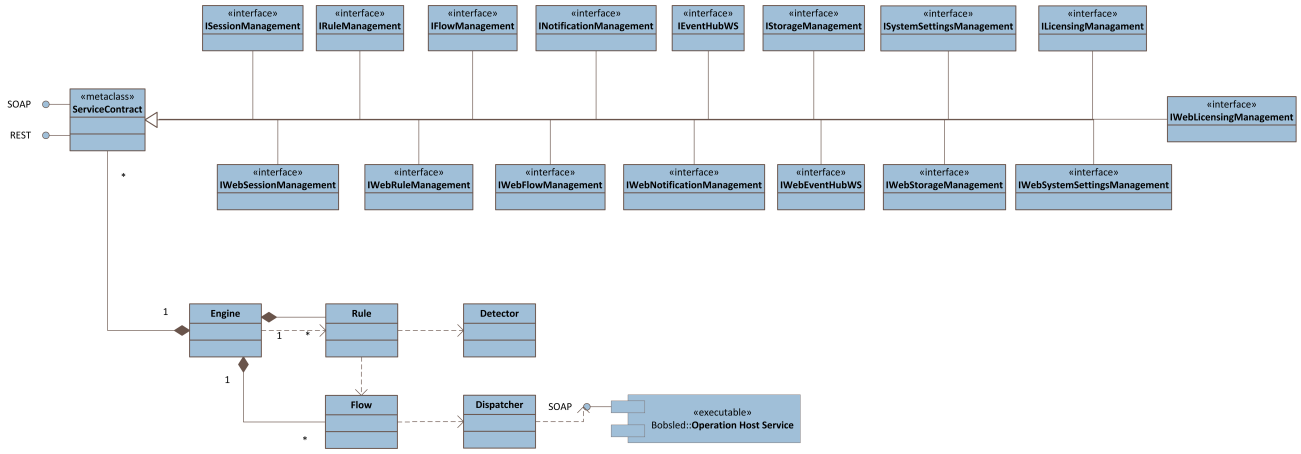

Figure 3.7: The Core Service architecture

The Operation Host Service is Bobsled's Core component where all the implementation regarding operations is. As it was just mentioned, a flow uses the Dispatcher class in order to contact the Operation Host to start or stop an operation. Since Bobsled Core and Operation Host are not "directly connected", all communication with the latter is done via its SOAP interfaces. Unlike Bobsled Core, which has SOAP and REST interfaces, in the current Bobsled implementation, Operation Host only has SOAP interfaces.

As for Event Hub, one should see it as a sort of "messaging component". One of the aspects that Bobsled aimed to improve was the notification process that was previously done through polling which is a very inefficient process. The solution to that problem was EventHub. This component allows other components like Rules or Flows to register a type of event that they're interested in so that, whenever an event of that type is sent, the interested parties are notified. EventHub is the main facilitator through which that whole process happens. All events are sent to EventHub who has the job of "spreading the word" to all of the interested parties in that particular event. This way one avoids the constant polling process to check whether or not some task has been completed.

3.2.2.2 Extensions

As already stated, the newsRail is composed by Bobsled plus a set of extensions. These extensions, that were also already mentioned, are the newsRailCommon extension and the ReutersExtension.

When designing the newsRail architecture, the decision was to separate every components specific to a particular basket which means that every basket has its own extension. This allows newsRail to be as modular as possible which brings several advantages not only when implementing but also for the end user, since, in theory, it's possible to "customize" according to each customer the number of baskets newsRail supports.

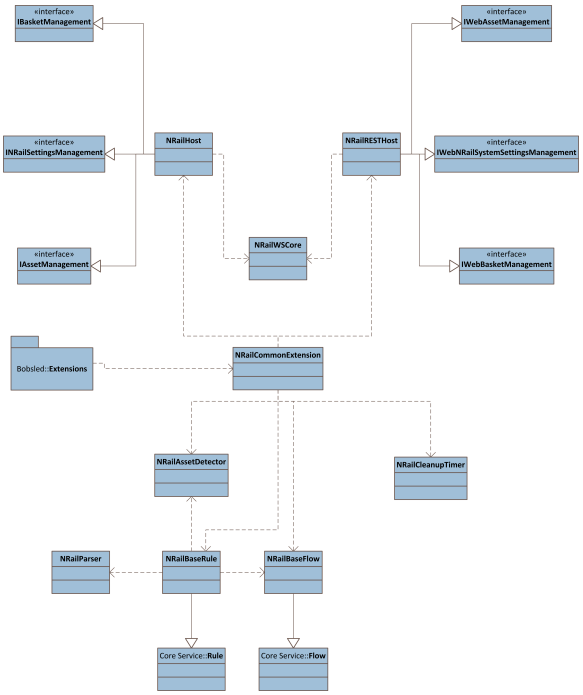

Figure 3.8: newsRailCommon architecture

There is obviously many more components that are common to every possible basket extension that may be added. In order to avoid code replication, every common component was "placed" in the newsRailCommon extension. In theory this component did not need to be an extension, however, some of the components that are common to every basket are the SOAP/REST interfaces and these need to be loaded into Bobsled. One of the modifications that were done on Bobsled was that it would load "foreign" interfaces as a regular extension. So, in order to have Bobsled load these interfaces, newsRailCommon had to be "converted" into an extension so that it could be loaded by loaded and, therefore, also load the interfaces. A component that is specific to newsRail is the NRailParser which is the asset metadata component. Since every basket has its own type metadata, which means that every basket extension should contain a specific implementation of the NRailParser. The component in the newsRailCommon extension is just an abstract class with some method declarations. The way the methods are implemented is different for each basket. In Figure 3.8 one can see the overall architecture of the newsRailCommon extension.

As one can clearly see newsRailCommon contains specific implementations of Bobsled concepts, namely, Rule throught the NRailBaseRule class, and the Flow through the NRailBaseFlow class. There is also the NRailAssetDetector which is class that inherits functionality from Bobsled's FileDetector class and, consequently, from Detector (Figure 3.9). Again, NRailAssetDetector is an implementation that is specific to newsRail

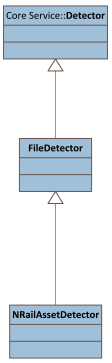

Figure 3.9: NRailAssetDetector inheritances

since the "normal" Bobsled's FileDetector would not meet the requirements.

The architecture of ReutersExtension can be seen in Figure 3.10. In this case, there is a specific implementation of NRailRule and NRailParser, ReutersRule and ReutersParser respectively. The flow used is the NRailBaseFlow and, in the next chapter this will be explained in detail, but succinctly one can say that the flow implemented in NRailBaseFlow covers the majority of cases when generating proxies/keyframes and/or exporting the asset to F1000. However it's perfectly possible for every basket extension to have a customized NRailBaseFlow implementation.

3.2.2.3 Operations

Operations can also be considered as extensions, or plug-ins, that are loaded, not by Bobsled Core, but by Operation Host (Figure 3.11). The current newsRail implementation only has one operation, *CarbonCoderOperation*. The reason why there is only one operation is quite simple: it's the only one that is needed in order to perform all the required tasks, namely, proxy/keyframe generation and export to F1000 hot folder. What CarbonCoderOperation actually does is connect to a Carbon Coder and send it messages

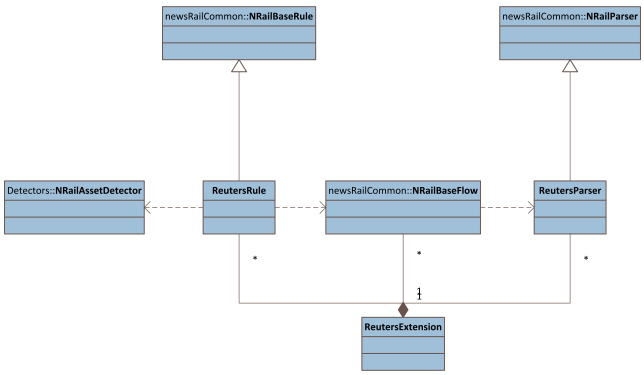

Figure 3.10: ReutersExtension architecture

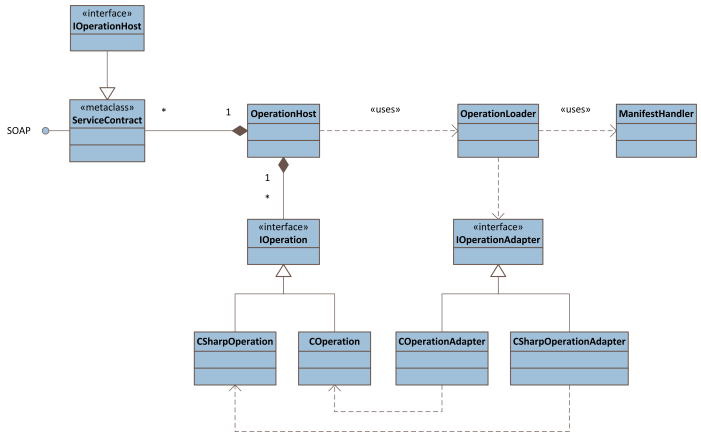

Figure 3.11: OperationHost architecture

containing information about the transcoding jobs it should perform. The implemented operation is versatile enough so that it possible to issue all these different job orders to Carbon Coder. The implementation details will be discussed in the next chapter.

3.2.3 Interfaces

The newsRail system has two types of interfaces, namely, SOAP and REST (except OperationService which only supports SOAP). Even though they were "included" in the previous Bobsled Core and newsRailCommon sections they are an important subject in this project and dissertation and so, they should be discussed separately from the other components.

Even though to an "outsider" it may seem that the existing interfaces are all in the same component, in reality there are two "families" of interfaces in the overall newsRail system, namely, the Bobsled Core interfaces and the newsRailCommon interfaces.

In the first case, these interfaces are "native" to Bobsled and were already implemented when this project began. The interfaces in question are:

- **IRuleManagement**
- **ISessionManagement**
- **IFlowManagement**
- **INotifcationManagement**
- **IEventHubWS**
- **IStorageManagement**
- **ISystemSettingsManagement**

- **ILicensingManagement**

Since the goal of MOG Solutions is to use Bobsled as the foundation for all product lines these interfaces were designed and implemented as being something that would be transversal to every product currently developed by MOG Solutions. These interfaces allow a client (like the GUI) to connect to Bobsled and control (via SOAP/REST messages) it as this is, in fact, the API that Bobsled exposes to the "outside world". This means that allows Bobsled, or a Bobsled-bases product can be easily wrapped and plugged into a SOA.

However, at the beginning of this project, one thing was perfectly clear: these interfaces lacked "bandwidth" in order to accommodate some of newsRail's concepts like Assets and Baskets. That is the main reason why it was decided to allow Bobsled to load "foreign" interfaces as regular extensions. In fact, this decision benefits not only the newsRail system but all other MOG Solutions products that still haven't been ported to a "Bobsled-based architecture" as it's unpractical to add functionalities to Bobsled, to meet every products requirements and needs and to be exposed to the "outside" as services through "product-specific" interfaces without it becoming a "monolithic behemoth" full of components that might be important in the context of one particular product but completely irrelevant in another.

As mentioned before, the "extensible API" was not the only modification that Bobsled underwent. Adding REST interfaces (similar in name to the SOAP interfaces but with prefix "IWeb") alongside the existing SOAP interfaces was also a specific requirement at the beginning of the project. In the next chapter one will see that that implementation process was not totally without its share of unexpected difficulties which lead to some modifications in Bobsled Core's architecture, which were, later on, replicated in newsRailCommon's architecture. Those modifications can be seen in Figure 3.7 and Figure 3.8. In each of those diagrams, there is a component, in the first case, it's *BobsledWSCore* and in the latter, *NRailWSCore*, that hold all the implementation regarding the exposed services (for both SOAP and REST). In Bobsled, the components that host the services, BobsledWSHost (SOAP) and BobsledRESTHost (REST), are in fact just a "door" to the real implementation that lies in BobsledWSCore. The reason why this has been designed this way and a single can't host (and implement) both SOAP and REST interfaces will be discussed in detail in the next chapter. In meantime, one can think of BobsledWSHost and BobsledRESTHost as being "two different doors to the same room", namely, BobsledWSCore.

In newsRailCommon the process was similar to this, as the interactions between NRailHost, NRailRESTHost and NRailWSCore are completely analogous to those just mentioned between BobsledWSHost, BobsledRESTHost and BobsledWSCore.

As for the interfaces that are "autochthonous" to the newsRail system, there are three of them, namely:

- **IAssetManagement**
- **IBasketManagement**
- **INRailSystemSettingsManagement**

They were conceived designed to meet specific newsRail requirements as it was necessary to provide ways for "outsiders" to have control over assets, baskets and some newsRail specific system configurations which were concepts that Bobsled lacked.

With IAssetManagement a client has access to the main functionalities regarding assets, namely list assets (i.e., search assets) and remove asset. As for the IBasketManagement interface, it "holds" all methods that allow the basic CRUD control over baskets. Finally, the INRailSystemSettingsManagement interface, has a "global" name since it was thought to hold all methods regarding specific configurations that may not need their own interface like Assets and Baskets. In the current newsRail implementation, this interface allows a client to add/configure/remove a Carbon Coder and set the NRailCleanupTimer. The latter, will discusses in greater detail in the next chapter, but for now one should that NRailCleanupTimer is the component responsible for automatically cleaning all assets that are over a pre-determinate "age". That "age" is what is a client, currently, can change through the INRailSystemSettingsManagement interface.

3.2.4 Externals

So far all "internal" components of the newsRail system have been discussed. In this section, the object of study will be all the components that, even though are part of newsRail, are external tools. One refers to the Carbon Coder transcoding system and to MongoDB.

3.2.4.1 MongoDB

MongoDB [mon11] is a "scalable, high-performance, open source, document-oriented database". At the beginning of this project, MongoDB was already Bobsled's database and used to store all kinds of information regarding many of Bobsled's components. In order to keep architecture simple and clean, it was decided that newsRail would also use Bobsled's database to store information about assets, baskets and coders.

MongoDB has several advantages over more "traditional" databases, namely:

- it's an open-source project with strong and active community as well as efficient support;

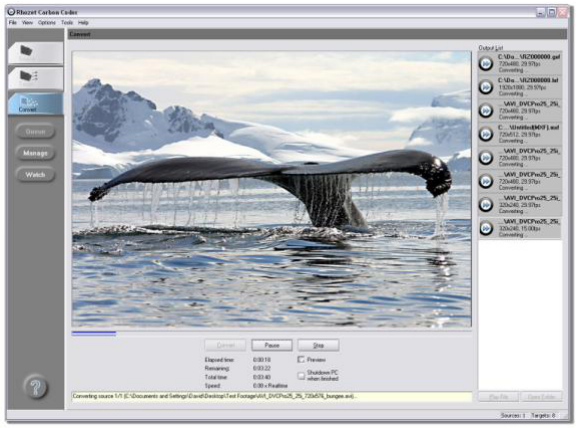

Figure 3.12: Carbon Coder GUI [Rho11b]

- it supports automatic replication of data between servers for failover and redundancy (which also makes it extremely scalable);

- since it's a "document-oriented" database, it doesn't use a strict data scheme (unlike traditional RDBMS) which allows for easier software updates and, again, easier scalability;

- it's free.

A document in MongoDB corresponds, in a way, to a row in a traditional database's table. Also, in MongoDB collections are the natural equivalents to tables, but unlike the latter, collections can contain documents that are all different from each other. This is possible because MongoDB is a *schema-free* database. This allows a collection of documents to keep completely different documents such as the ones shown below:

```
1  {"greeting" : "Hello, world!"}
2  {"foo" : 5}
```

As one can see, a document in MongoDB can be seen as an ordered set of keys with some associated values. In the previous example, the key "foo" has an associated value of "5". Again, this is a completely different paradigm from traditional databases where the schema in each table is quite "strict" and every entry must contain the same kind of information.

How data is actually introduced into MongoDB and how newsRail interacts with it will be studied in the next chapter.

3.2.4.2 Carbon Coder

Carbon Coder [Rho11b] is a file-based transcoding application that supports all major formats. It can work as a stand-alone application (Figure 3.12) or as part of a multi-node, fully-automated rendering farm under the control of Carbon Server [Rho11a] or the Rhozet Workflow System (WFS) [Rho11c].

In a typical three-tier architecture the CarbonCoder would correspond to the lowest of the three tiers as this is the "layer" that actually handles the data, namely, when transcoding the video files submitted by CarbonCoderOperation.

Carbon Coder's API provides three separate interfaces through which one can submit jobs:

- XML file watch folder
- Sockets
- Command Line

The choice for job submission was using sockets. Unlike the "XML file watch folder method", where one just sends an XML file to a designated folder that Carbon Coder is monitoring and one gets no feedback whatsoever about the current status of the current job, using sockets to submit jobs to Carbon Coder, one receives constant feedback about the status of the job (like the completion percentage). The third option, command line, required the use of a small shell-like application to submit jobs.

The job submission consists of a message with all the details necessary to execute the job as well as options regarding the transcoding process. That message is written in an "customized" (and documented) XML format that Carbon Coder can read. The job of CarbonCoderOperation is to assemble the message according to the task that should be performed.

3.3 Conclusion

In this chapter all of newsRail's architecture was studied. Reviewing all that has been said, one must highlight certain facts.

First, the "customization" that Bobsled underwent in order to have it load product specific interfaces as it would with any normal extension and the implementation of REST interfaces alongside the SOAP interfaces. The first case, at an architectural level isn't something that truly affected Bobsled, even though it's something that affects any application developed using Bobsled as its framework. In the second case, however, the impact in Bobsled's architecture was truly felt. Without going into any implementation details, there was unexpected difficulties, namely when pairing SOAP and REST interfaces in the

same service host. After some time studying the subject and after finding out that, due to some constraints regarding the contents of the SOAP messages that were sent to the GUI and how they were incompatible with REST, it was unpractical to implement REST and SOAP in the same host, a decision had to be made. That decision was to completely separate both interfaces in separate hosts. Those hosts instead of implementing (and, obviously, hosting) the interfaces they would only "point" to another component were the actual interface implementation. With this any code replication in both interfaces was avoided and, if one wishes to update the interface implementation, the interface itself can be left "untouched" as its completely separated from implementation.

Second, the decision that was made regarding the "separation" of every basket in different extensions. This architecture design option was made in order to keep newsRail as modular and flexible as possible. By separating every basket into its own extension, later on, one can deploy newsRail with a "custom set" of baskets according to each client's needs. Another reason why this decision was made has to do with the fact that not every basket is the same. A Basket 'X' may need a different Rule implementation than basket 'Z'. In the newsRailCommon extension there is a rule implementation that should "fit" every basket (and the same happens with the flow), but if needed, one can always implement a custom rule for a specific basket. Basically, by separating baskets into their own extensions one can think of the newsRailCommon extension as a foundation of a house and the basket extensions the bricks which can be placed on top of the foundation.

In the next chapter, the actual implementation of what has just been mentioned will be discussed in detail.

Chapter 4

Project Implementation

In this chapter all the implementation details will be discussed, what were the main issues and how they were solved.

4.1 Implementation Context

In order to implement this project, the chosen framework was Microsoft .NET. This framework has been designed from the beginning to support web services (creating and consuming) and it's the main development environment within MOG Solutions, as well as Microsoft's integrated development environment (IDE), Visual Studio .NET, which contains a set of tools that allow to create a number of application with special emphasis on web services web services-based applications. Even though the .NET framework supports a number of programming languages, the implementation language of choice was C# as this is Bobsled's "native" language.

4.2 Bobsled

As already mentioned, when this project started there was already an implemented version of Bobsled, that was being used to develop the O1000 outgest system. The Bobsled framework aims to centralize common features to all MOG Solutions products and it provides the developer with a set of "concepts" which can be used as "building blocks" for applications which, in turn, allow a complete control of a program's workflow.These concepts are:

- **Rules** — Uses events from detectors to trigger flows.

- **Flows** — Encapsulates a state-machine that specifies how operations collaborate in order to perform a task.
- **Detectors** — Monitors media sources and generates events on changes.
- **Extensions** — A plugin for Bobsled, loaded at runtime
- **Operations** — Basic unit of a workflow, loaded at runtime, but independent from "regular" extensions.

Also, Bobsled already had SOAP interfaces implemented, namely:

- **IRuleManagement**
- **ISessionManagement**
- **IFlowManagement**
- **INotifcationManagement**
- **IEventHubWS**
- **IStorageManagement**
- **ISystemSettingsManagement**
- **ILicensingManagement**

However, when designing newsRail, one thing became clear: the current implementation of Bobsled didn't cover some concepts that were unique to newsRail! These concepts were:

- **Assets** — "Building block" of a news segment, composed by a video file and a metadata file.
- **Baskets** — "Box" that receives news feeds from a news agency.
- **Coders** — The Carbon Coder transcoding system.

Since Bobsled didn't covered this concepts, it's only natural that it also didn't have interfaces to allow "outside interaction" with those concepts, namely:

- **IAssetManagement**
- **IBasketmanagement**
- **INRailSystemSettingsManagement** (allows, among other things, configuration of one or more Carbon Coders)

This meant that Bobsled had to be modified in order to perform two things: load interfaces as if they were "regular" extensions, which in a way means extending Bobsled's own API, and add REST interfaces to Bobsled (and, consequently, to newsRail).

The first step towards enabling the extensible API feature in Bobsled was to study how did it load extensions. When booting up, Bobsled loads an XML configuration file that contains Core's global settings. One should quickly mention that, Core and Operation Host are two completely different and independent processes, which means that they both are booted up independently. So, when stating that when Core botts up it loads an XML file containing global settings that it needs in order to properly function, the same thing applies to Operation Host.

The loaded settings in the XML configuration file will be used in the creation an object of type Engine (see Figure 3.7). This class, Engine, is responsible for "loading" several objects of these types:

- **IEventHub**
- **ISessionManager**
- **INotificationManager**
- **IDetectorManager**
- **IDeviceManager**
- **IRuleManager**
- **IFlowManager**
- **IWSHostManager**

The mentioned classes provide methods to control several aspects like Sessions, Flows, Rules and, the one that was added to Bobsled specifically for the extensible API feature, WSHostManager. They allow components within the application to access important aspects like using the SessionManager to access the current session's *Id*.

The WSHostManager class implements the IWSHostManager interface which declares one method:

- **void AddWSHost(IHostInfo host)** — This method adds an object of type IHostInfo into a list of IHostInfo.

During the booting process a class, BobsledWSServer, that in the "original" Bobsled version loaded the single service host that Bobsled had, was modified in order to load all hosts, which means, extending Bobsled's API. This is done taking advantage of the mechanism that Bobsled already used to load extensions. In a class named ExtensionLoader,

the method TryLoadAssembly. This method loads extensions by "searching" in a specific folder for files that implement the IExtension interface and, more precisely, the Initialized method. This method allows extensions to "add" their custom Rules, Flows, Detectors and, now, WSHosts (service hosts) to the Bobsled engine. Every extensions that "wishes" to add their own API to Bobsled, just needs to add a WSHost:

```
1 public void Initialize(IEngine engine)
2 {
3     if (engine == null) throw new ArgumentNullException("engine");
4
5   engine.WSHostManager.AddWSHost(new NRailHost(engine));
6   engine.WSHostManager.AddWSHost(new NRailRESTHost(engine));
7                                           ...
8 }
```

This will add the hosts NRailHost (SOAP) and NRailRESTHost (REST) to the already mentioned list in WSHostManager. Adding the Bobsled hosts, namely, BobsledWSHost (SOAP) and BobsledRESTHost (REST), is a process that, unlike the process done with the extensions, is done "manually":

```
1   BobsledWSHost wsHost_ = new BobsledWSHost(engine_);
2   engine_.WSHostManager.AddWSHost(wsHost_);
3   engine_.WSHostManager.AddWSHost(new BobsledRESTHost(engine_));
```

The next step is loading the SOAP and REST interfaces from each of the stored hosts in order to create the endpoints through which the clients can communicate with the application. Since an extension can have many more interfaces that aren't necessarily SOAP or REST endpoints, a solution had to be found to make those two kind of interfaces "stand out in the crowd". The answer to that was to create a custom attribute class, ServiceHostInfo. This attribute class allows BobsledWSServer to differentiate the SOAP and REST interfaces. One example of how one would use the ServiceHostInfo class:

```
1   [ServiceHostInfo(typeof(IAssetManagement), "AssetManagement", BindingType.
        Soap)]
2   interface IAssetManagement
3   {...}
```

The ServiceHostInfo constructor accepts three arguments: the first is the type of the respective interface, the second is the name one wishes to give to that particular endpoint (as it will appear in the URI) and finally, the third argument defines the type of binding, or in other, if it's a SOAP or REST interface.

All this information will be of vital importance in the BobsledWSServer when loading the hosts in the HostInfo list. The process is as follows:

```
1
2   server_ = AddHost(engine_.WSHostManager.Hosts());
3
```

```
4   foreach (var serviceHost_ in server_.ServiceHosts)
5           {
6               Type[] interfaces_ = serviceHost_.SingletonInstance.GetType().
                    GetInterfaces();
7
8               foreach (var interface_ in interfaces_)
9               {
10                  System.Attribute[] attrs = System.Attribute.
                        GetCustomAttributes(interface_);
11
12                  foreach (System.Attribute attr in attrs)
13                  {
14                      if (attr is ServiceHostInfo)
15                      {
16                          ServiceHostInfo info_ = (ServiceHostInfo)attr;
17
18                          try
19                          {
20                              BindingType binding = info_.GetBinding();
21
22                              switch (binding)
23                              {
24                                  case BindingType.Soap:
25                                      server_.AddServiceEndpoint(info_.
                                            GetInterfaceType(), info_.GetName()
                                            , serviceHost_);
26                                      break;
27                                  case BindingType.Rest:
28                                      server_.AddRestServiceEndpoint(info_.
                                            GetInterfaceType(), info_.GetName()
                                            , serviceHost_);
29                                      break;
30                                  case BindingType.All:
31                                      server_.AddServiceEndpoint(info_.
                                            GetInterfaceType(), "SOAP/" + info_
                                            .GetName(), serviceHost_);
32                                      server_.AddRestServiceEndpoint(info_.
                                            GetInterfaceType(), "REST/" + info_
                                            .GetName(), serviceHost_);
33                                      break;
34                                  default:
35                                      break;
36                              }
37                          }
38                          catch (System.Exception e)
39                          {
40                              log.ErrorFormat("Interface attributes cannot be
                                    null: {0}", e.ToString());
41                          }
```

```
42                          }
43                      }
44                  }
45
46                  server_.AddMetadataExchangeEndpoint("MEX", serviceHost_);
47              }
48  ...
49  public WSServer AddHost(List<IHostInfo> hosts)
50  {
51              WSServer wsServer_ = new WSServer(hosts, "Bobsled");
52              return wsServer_;
53  }
```

The WSServer class has methods that create and add the endpoints. Initially it only had two methods, the AddMetadataExchangeEndpoint method which provides an endpoint through which clients can connect in order to "read" the WSDL file of the endpoint, and the AddServiceEndpoint which adds the previously loaded endpoint. In order to accommodate REST a third method was created:

```
1  public void AddRestServiceEndpoint(Type serviceType, string restServiceUrlPath,
       ServiceHost serviceHost_)
2  {
3    if (ServiceHosts == null) SetupServiceHost(hosts_);
4
5    WebHttpBehavior endpoint_behavior = new WebHttpBehavior();
6
7    Uri endpointAddress = new Uri(serviceHost_.BaseAddresses[0] +
         restServiceUrlPath);
8    ServiceEndpoint restEndpoint = serviceHost_.AddServiceEndpoint(serviceType,
         restBinding_, endpointAddress);
9    restEndpoint.Behaviors.Add(endpoint_behavior);
10 }
```

After all the process has been finished, each and every endpoint will have a URI that identifies it.

- http://<address>:8731/Bobsled/Core/**SOAP**/<endpoint>/
- http://<address>:8731/Bobsled/Core/**REST**/<endpoint>/

In those URIs there are two highlighted items, SOAP and REST. When adding a service endpoint, SOAP and REST endpoints are differentiated and separated. Also the, the "Core" element in those URI's is something that is not "static". That element is "loaded" from each Host since host has a property, Name, that returns a string naming the "service" to which it belongs. With that in mind, one arrives at the obvious conclusion those two URIs are for hosts "within" the Core Service. In newsRail case, the URIs would look like this:

- http://<address>:8731/Bobsled/newsRail/**SOAP**/<endpoint>/

- http://<address>:8731/Bobsled/newsRail/**REST**/<endpoint>/

This situation arose from the fact that, unlike initially thought, pairing SOAP and REST interfaces in the same host proved to be, in this project's case, impossible. This happens because REST has some serious deserialization issues with a SOAP's message header. In Bobsled's case the header in a SOAP message transports the *sessionId* which is a "key" that allows users to interact with the application. If the *sessionId* is not valid, an exception ins immediately thrown. The way that Bobsled and the GUI (developed in Flex) interact "forces" one to use a message header containing the *sessionId*.

Another problem that clearly shows the difficulty of using REST alongside SOAP is that, REST, doesn't deserialize complex data (i.e., objects) within its messages.

These two difficulties were the main reason why it was chosen to separate both types of interfaces.

4.3 newsRailCommon

This extension, in newsRail's current implementation, houses most of its code.

First, the Initialize() method to this extensions which allows it to be "discovered" by Bobsled's ExtensionLoader class:

```
1  public void Initialize(IEngine engine)
2  {
3      if (engine == null) throw new ArgumentNullException("engine");
4
5    engine.WSHostManager.AddWSHost(new NRailHost(engine));
6    engine.WSHostManager.AddWSHost(new NRailRESTHost(engine));
7    engine.DetectorManager.RegisterDetectorType(NRailAssetDetector.DetectorName,
         typeof(NRailAssetDetector));
8    engine.FlowManager.RegisterFlowType(NRailBaseFlow.FlowName, typeof(
         NRailBaseFlow));
9    LoadNRailConfig();
10   NRailCleanupTimer cleanupTimer = new NRailCleanupTimer(assetAge_);
11   NRailCleanupTimer.StartCleanupTimer();
12 }
```

As we can see, the newsRailCommon extension adds two hosts (NRailHost and NRailRESTHost), one detector (NRailAssetDetector) and one flow (NRailBaseFlow). This means that this extension doesn't have any true implementation of a Rule. The NRailBaseRule is an abstract class, which means that it's only meant to be used as a base class for others and so, every basket extension should implement it's own Rule.

4.3.1 NRailBaseRule

As stated NRailBaseRule is an abstract class which acts as a basis for others, namely, rules implemented in basket extensions. The main function of a Rule is to create a flow according after receiving a specific event (NRailBaseRule creates and instance of NRailAssetDetector), in this case, every time a new asset is detected it should create the appropriate flow, but not without first inserting into the DB all the data from the asset metadata file. To do this, the NRailBaseRule creates an object of type NRailParser and uses it in order to read all metadata from the asset. This is seen in the code excerpt that follows:

```
1  Type parserType = this.GetParser;
2  NRailParser parser = (NRailParser)Activator.CreateInstance(parserType);
3  parser.SetMetadata(fe.Asset.AssetMetadata.Uri);
```

The *SetMetadata* method will load in the metadata file to be loaded and then it will be accessible to be read and stored in the DB.

The "flow" of a Rule is quite simple and it can be summarized in three steps:

- **1.** When the rule is created the first thing it does is create a NRailAssetDetector object that will monitor a basket.
- **2.** When a new asset event is received the rule creates a NRailParser object in order to have acess to all the assets metadata and stores that metadata into the BD.
- **3.** The rule creates a flow according to the extension from which the actual code it is running. Since NRailBaseRule is an abstract class, a basket extension can have a specific rule that may not need to implement all of the methods of NRailBaseRule.

4.3.2 NRailParser

Just like NRailBaseRule, NRailParser, is an abstract class as it serves as basis for other classes, namely, the Parser classes that every basket should have. This happens because every metadata schema is differente from basket to basket which means that every parser must also be different. This also means that, unlike NRailBaseRule that had some implemented methods that could be used by every extension's rule, the NRailParser only declares the methods and its up to the extension's parsers to implement each and every one of them. The methods are:

```
1  public abstract void SetMetadata(UriEx assetMetadataComponent);
2  public abstract string GetAssetTitle();
3  public abstract DateTime GetAssetDate();
4  public abstract string GetAssetDuration();
5  public abstract string GetAssetDescription();
```

4.3.3 NRailAssetDetector

The NRailAssetDetector "uses" the FildeDetector class from Bobsled's Core in order to implement its main functionality, detect new assets. As already stated an asset is made up of a pair of files, a metadata file and a video file. Apart from the file extension, both files share the same name. Whenever FileDetector detects a new file (metadata file or video file) in a basket, it launches an event. This event is "picked up" by NRailAssetDetector which will then check if the that particular asset name was already detected. If so, it means that there's a "semi-complete asset" waiting for completion and it will check if the detected file that is the missing half of the asset. If that is true, then the asset is completed and and event (containing all the information regarding that asset) will be launched in order to be picked up by a Rule. If the file that was detected is not the missing half, it means that it's the same component that had already been detected and, therefore, it will be ignored. On the other hand, if a detected file has a name that has never been detected before, it mens that that file is a video or metadata component of a new asset.

The NRailAssetDetector also handles removed assets, again "using" Bobsled's Core which also detects files that have been removed. If a asset component is removed that asset, even though still stored in the DB, will automatically become "invisible" and will no longer appear in any asset list. if the final half of the asset is removed then all of the asset is cleaned from the DB as well as any proxy/keyframe that might have been generated.

4.3.4 NRailBaseFlow

As already stated, a flow encapsulates a state-machine that specifies how operations collaborate in order to perform a task. In newsRail's case, as it will be shown in the next section, only one operation, CarbonCoderOperation, is needed and is, in fact, started.

NRailBaseFlow altough not an abstract class, like NRailBaseRule, has some virtual methods which means that, a basket extension can implement its own specific flow that will inherit from NRailBaseFlow and override any method to meet is particular needs.

These methods are:

```
1  protected virtual XElement createOperationConfig(OutputKind outputKind, UriEx
       videoSource_, string assetId) {...}
2
3  protected virtual void OnOperationEvent(object sender, BobsledEventArgs e)
       {...}
4
5  protected virtual void FinishComplete() {...}
```

The first, *createOperationConfig*, creates the configuration string that will be sent to the operation so it performs its task. The *OnOperationEvent* method can be called the "heart" of the flow as this is the "place" where the state-machine is located. This method

"reacts" to events sent by operations and acts accordingly to it. One example is the proxy/keyframe generation flow. In this case the first operation that is invoked is the proxy generation. Again, as we will see in the next section, the actual code that implements the operation is the same, the only difference lies in the configuration that is sent to the operation. That said, when invoking the proxy generation operation one really means that a configuration specific to that was created and sent to the operation.

After being invoked, and regarding the fact that newsRail submits jobs to Carbon Coder using sockets, constant feedback of the job status is being sent back to the operation which, in turn, sends events that are caught by the *OnOperationEvent* method. The event (of type BobsledEventArgs) contains information about the current status of the job and it completion percentage. Using that information the *OnOperationEvent* method can know when the operation finished and, as soon as this happens it creates a new configuration (for keyframe generation) and invokes the operation for the second time.

Finally, the *FinishComplete* method "springs to action" right after a flow ends its workflow. What it does is insert some data into the DB regarding the asset that was just processed, like setting a flag that makes the asset "visible" (in the proxy/keyframe generation workflow).

4.3.5 Interfaces

When designing the newsRail interfaces the requisites of the project were taken into serious consideration. During the designing process some aspects were considered of supreme importance and therefore, namely, a way to handle assets, baskets and coders. These three elements are of key elements within the newsRail workflow and so, three SOAP interfaces and its three REST "counterparts" were designed:

- **IAssetManagement**
- **IBasketManagement**
- **INRailSystemSettingsManagement**
- **IWebAssetManagement**
- **IWebBasketManagement**
- **IWebNRailSystemSettingsManagement**

The implementation paradigm of these interfaces follows what has been done with the Bobsled interfaces. The SOAP and REST interfaces are hosted in different hosts, namely, NRailHost (SOAP interfaces) and NRailRESTHost (REST interfaces), but the actual implementation code for both is in a separate classe, NRailWSCore. Since the process

through which the interfaces were designed is similar to the process already described in the Bobsled section, only the methods of the SOAP interfaces will used as examples.

The first, IAssetManagement, defines the following methods:

```
1  [ServiceContract(Namespace = "http://mog-solutions.com/schemas/Bobsled")]
2  [ServiceHostInfo(typeof(IAssetManagement), "AssetManagement", BindingType.
       Soap)]
3      interface IAssetManagement
4      {
5              /// <summary>
6              /// Returns a list of all assets in a basket
7              /// </summary>
8              /// <param name="message"></param>
9              /// <returns></returns>
10             [OperationContract]
11             IAssetManagement_ListAssets_OutputMessage ListAssets(
                   IAssetManagement_ListAssets_InputMessage message);
12
13             /// <summary>
14             /// Removes a specific asset
15             /// </summary>
16             /// <param name="message"></param>
17             /// <returns></returns>
18             [OperationContract]
19             IAssetManagement_RemoveAsset_OutputMessage RemoveAsset(
                   IAssetManagement_RemoveAsset_InputMessage message);
20     }
```

The first method, *ListAssets*, can be considered the core functionality of the newsRail news selection system. This method allows a client receive a list of assets according to some search criteria. Searching is done using a keyword that is input by the user in the GUI and going through the asset's metadata stored in the DB. In order to do so, a regular expression was built to search the metadata:

```
1  string regex = String.Format(".*({0}).*", string.Join("|", keywords));
```

The search can be done according to basket and it can be filtered according to two parameters:

- **AssetDate** — The asset creation date which is present in the asset's metadata.
- **SystemDate** — The date which in which the asset was inserted into the DB.

Also, in both cases, one can sort the search by ascending or descending order.

As for IBasketManagement, the following methods are defined:

```
1  [ServiceContract(Namespace = "http://mog-solutions.com/schemas/Bobsled")]
2  [ServiceHostInfo(typeof(IBasketManagement), "BasketManagement", BindingType.
       Soap)]
```

```
3   interface IBasketManagement
4   {
5               /// <summary>
6               /// Adds a basket
7               /// </summary>
8               /// <param name="message"></param>
9               /// <returns></returns>
10              [OperationContract]
11              IBasketManagement_EditBasket_OutputMessage EditBasket(
                    IBasketManagement_EditBasket_InputMessage message);
12
13              /// <summary>
14              /// Returns a list of all assets in a basket
15              /// </summary>
16              /// <param name="message"></param>
17              /// <returns></returns>
18              [OperationContract]
19              IBasketManagement_ListBaskets_OutputMessage ListBaskets(
                    IBasketManagement_ListBaskets_InputMessage message);
20
21             /// <summary>
22              /// Deletes a basket
23              /// </summary>
24              /// <param name="message"></param>
25              /// <returns></returns>
26              [OperationContract]
27              IBasketManagement_RemoveBasket_OutputMessage RemoveBasket(
                    IBasketManagement_RemoveBasket_InputMessage message);
28      }
```

The methods names are self-explanatory, with one exception, the EditBasket method. This method allows not only to edit a basket but also to add it. It does this by checking if in the message received the basket Id is null. If so, then it the method will create a new entry in the DB. If not, it does a search on the Basket collection in order to find the basket with that Id and update it with the data.

Finally, INRailSystemSettingsManagement:

```
1   [ServiceContract(Namespace = "http://mog-solutions.com/schemas/Bobsled")]
2       [ServiceHostInfo(typeof(INRailSystemSettingsManagement), "
            NRailSystemSettingsManagement", BindingType.Soap)]
3       interface INRailSystemSettingsManagement
4       {
5               /// <summary>
6               /// Adds a transcoder
7               /// </summary>
8               /// <param name="message"></param>
9               /// <returns></returns>
10              [OperationContract]
```

```
11              INRailSystemSettingsManagement_EditCoder_OutputMessage EditCoder(
                    INRailSystemSettingsManagement_EditCoder_InputMessage message);
12
13              /// <summary>
14              /// Removes a transcoder
15              /// </summary>
16              /// <param name="message"></param>
17              /// <returns></returns>
18              [OperationContract]
19              INRailSystemSettingsManagement_RemoveCoder_OutputMessage
                    RemoveCoder(
                    INRailSystemSettingsManagement_RemoveCoder_InputMessage message
                    );
20
21              /// <summary>
22              /// Lists all transcoders
23              /// </summary>
24              /// <param name="message"></param>
25              /// <returns></returns>
26              [OperationContract]
27              INRailSystemSettingsManagement_ListCoders_OutputMessage ListCoders(
                    INRailSystemSettingsManagement_ListCoders_InputMessage message)
                    ;
28
29              /// <summary>
30              /// Sets old proxies cleanup period
31              /// </summary>
32              /// <param name="message"></param>
33              /// <returns></returns>
34              [OperationContract]
35              INRailSystemSettingsManagement_EditSystemConfiguration_OutputMessage
                     EditSystemConfiguration(
                    INRailSystemSettingsManagement_EditSystemConfiguration_InputMessage
                     message);
36
37              /// <summary>
38              /// Sets old proxies cleanup period
39              /// </summary>
40              /// <param name="message"></param>
41              /// <returns></returns>
42              [OperationContract]
43              INRailSystemSettingsManagement_ListSystemConfiguration_OutputMessage
                     ListSystemConfiguration(
                    INRailSystemSettingsManagement_ListSystemConfiguration_InputMessage
                     message);
44          }
```

The first three methods are CRUD functions regarding Carbon Coders and they're functionally similar to what was done in the IBasketManagement interface. As for the

EditSystemConfiguration, in this prototype version of newsRail, its only function is to set the NRailCleanupTimer with a new asset age.

The NRailCleanupTimer is a class that is launched right at the same time the NRailCommonExtension Initialized method is invoked and is in charge of "cleaning up" old assets daily. Every 24 hours, NRailCleanupTimer searches the DB for old assets. By default, any asset that is stored for more than 10 days is automatically erased. In the EditSystemConfiguration it's possible to set a new asset age and by asset age one means the maximum number of days an asset can be stored in the system.

4.4 ReutersExtension

Since one of the objectives of the NRailCommon extension was to group as much common code as possible and, therefore, have "lighter" basket extensions, this extensions, ReutersExtension, has "only" three classes, namely:

- **ReutersExtension**
- **ReutersRule**
- **ReutersParser**

In the ReutersExtension, just like NRailCommonExtension, allows this basket extension to be "found" and loaded by Bobsled. However, since this extension only needs to add a rule to the Bobsled engine, the method is smaller:

```
1  public void Initialize(IEngine engine)
2  {
3              if (engine == null) throw new ArgumentNullException("engine");
4
5              engine.RuleManager.RegisterRuleType(ReutersRule.RuleName, typeof(
                   ReutersRule));
6  }
```

As for the ReutersParser class it implementes all the already mentioned methods the NRailParser abstract class. It is custom built to "read" Reuters metadata files and extracts the following information:

- **AssetTitle** — the asset's title.
- **AssetDate** — the asset's creation date.
- **AssetDuration** — the asset's video duration.
- **AssetDescription** — a small description of the news segment contained in the asset's video file.

As for ReutersRule, as one can see has little implementation code:

```
1  class ReutersRule : NRailBaseRule
2  {
3    public static string RuleName { get { return "mog.newsrail.rules.
       reutersrule"; } }
4
5    public override string Name { get { return ReutersRule.RuleName; } }
6
7    protected override Type GetParser { get { return typeof(ReutersParser); } }
8
9    protected override string GetFlow { get { return NRailBaseFlow.FlowName; }
       }
10 }
```

Since most of the functional code is located in NRailBaseRule class, the ReutersRule class, as all future basket extensions, only has two main functions, namely, implementing methods GetParser and GetFlow. This will allow in the NRailBaseRule portion of the code to know what parser and what flow it should be invoking. This implementation option allows NRailBaseRule to be "used" by many basket extensions since every one of them mught have a customized parser/flow.

4.5 CarbonCoderOperation

As already stated earlier, operations are also plugins for Bobsled, but they are different and independent from extensions. This is especially true if one notes that extensions are loaded by Bosled's Core Service and operations are loaded by Bobsled's Operation Host Service.

CarbonCoderOperation is the only operation in newsRail as no more are needed since proxy/keyframe generation and exporting video file as OP1a file to F1000 hot folder all depend on the configuration this class receives from the Flow. In reality, CarbonCoder-Operation, doesnt' actually submit the jobs to the Carbon Coder system. That task is the responsibility of an already existing module (developed by MOG Solutions) that is used to submit jobs to Carbon Coder. What CarbonCoderOperation actually does is receive the configuration from the Flow, break it down and extract information from it. That information will then be used to construct the XML request string that will be sent to the Carbon Coder. In order to do so, CarbonCoderOperation loads one of two templates (one for proxy/keyframe and another for OP1a exporting) files that contain the XML configuration that should be sent to Carbon Coder. This and other elements that were "extracted" from the configuration received from the flow, are sent to the TranscodeOperation class which is located within the already mentioned developed module used to submit jobs to Carbon Coder.

```
1  void AssetToCoder(object state)
2  {
3          ...
4    transcode_ = new TranscodeOperation(carbonCoderUri_,
         assetVideoComponentPath_.LocalPath, targetDirectory_, coderConfigFile_)
         ;
5    transcode_.Start();
6    started_ = true;
7    transcode_.OnStatusUpdate += new EventHandler<StatusUpdateEventArgs>(
         transcode_OnStatusUpdate);
8    transcode_.OnComplete += new EventHandler<EventArgs>(transcode_OnComplete);
9    transcode_.OnFailed += new EventHandler<FailedEventArgs>(transcode_OnFailed
         );
10 }
```

Another task that is the responsibility of the CarbonCoderOperation class is receiving the events from TranscodeOperation which in turn receives them from the Carbon Coder system itself. CarbonCoderOperation then redirects those events to EventHub where, after being launched will be, eventually, caught by the flow that invoked this operation.

4.6 newsRail Data

As already mentioned in the previous chapter, it was decided that newsRail would use Bobsled's own DB, MongoDB. What this, in fact, really means is that, alongside the already mentioned Bobsled collections, there would be new ones, namely:

- **Assets**
- **Baskets**
- **Coders**
- **SystemConfig**

As already mentioned in the previous chapter, one of the advantages of MongoDB is that instead of storing data in table rows like more "traditional" DBs, it stores them in *Documents*. These are, in way, just a representation of an object which means that what in reality MongoDB stores are objects. Also as mentioned before, MongoDB is a *schema-free* database which means that, in theory, one could choose to create a single newsRail collection where all of the newsRail data would stored. However, this hypothesis was quickly discarded as it would be completely impractical and there would be no effective of searching through all of the documents and no feasible way of retrieving only documents of a certain kind. This means that classes had to be implemented in order to parametrize the information regarding the several collections just mentioned that should be stored. The answer to this was to create four different classes:

- **AssetInfo**
- **BasketInfo**
- **CoderInfo**
- **SystemConfigInfo**

The AssetInfo class stores the following data:

```
1  public string Name = "";
2  public string Title = "";
3  public string Id = "";
4  public string AssetDate;
5  public string SystemEntryDate;
6  public string Duration  = "";
7  public string Headline = "";
8  public string Description = "";
9  public string OriginalPath = "";
10 public string ProxyPath = "";
11 public string KeyframePath = "";
12 public string BasketId;
13 public string ParentRuleId = "";
14 public bool AlreadyExported  = false;
15 public bool Visible = false;
```

As for BasketInfo:

```
1  public string Id = "";
2  public string Name = "";
3  public BasketKind Kind = BasketKind.Reuters;
4  public string Address = "";
5  public string Username = "";
6  public string Password = "";
```

Next is CoderInfo:

```
1  public string Id = "";
2  public string Name = "";
3  public string Address = "";
4  public string Port = "";
```

Finally, SystemConfigInfo:

```
1  public string Id = "";
2  public string Name = "";
3  public bool Default = false;
4  public int assetAge = 10;
```

These classes are mimicked by the following classes:

- **AssetInfoDAO**

- **BasketInfoDAO**
- **CoderInfoDAO**
- **SystemConfigInfoDAO**

The DAO suffix stands for *Data Access Object* and these classes provide an abstract interface that allows the system to access the underlying database but, at the same time, keeping both "worlds" separated as this mechanism doesn't expose details of the database. All the CRUD operations that are performed in the newsRail system (and also in Bobsled) are done through the corresponding DAO classes.

4.7 Conclusion

In this chapter the most relevant aspects of the newsRail implementation were discussed. Some unexpected difficulties slowed down this process right when it was starting, namely, the SOAP and REST interfaces feature which took longer to expected to implement. But, after the solution was found, the process became straightforward and, in a way, almost mechanized. We can use Bobsled's and newsRail's cases as example. In th first case, implementing the interfaces was almost a month-long process. In the latter it was implemented during the course of a morning.

And the same applies to basket extensions since after implementing the first, ReutersExtension, all future extensions will become easier and faster to implement.

Chapter 5

Final Conclusions and Future Work

This project can be divided into two distinct "segments": the first, consisted of a more theoretical approach to the project, where all the concepts and characteristics behind a Service Oriented Architecture were studied. Next, and on a more practical level, all the main technologies used to implement a SOA and Web Services were studied in order to perceive what were their main advantages and limitations, as well as the way they "fit" into the whole SOA paradigm. Also, a study of the media business was undertaken, with special emphasis on the newsroom environment in order to better comprehend the way journalists do their work and to better understand in what ways could a journalist benefit from this project, as this is a project that was originally conceived as being custom made to "fit" into the newsroom workflows. Still regarding the media, a study of a main technology, namely, the MXF file format, was done. This format has become, in recent years, a major player within the media world, and it also plays an important role in this project. This is easily explained as the company where this project took place, MOG Solutions, is the worldwide leader in MXF technology development and so, it's only natural that a project proposed by MOG Solutions relies on their area of expertise.

The second "segment" was dedicated to actual development and implementation of this project. Using what had been previously studied regarding SOA and newsrooms workflows, the objective was to implement a news selection system, newsRail, that would be SOA-compliant and fit in the newsrooms workflow by facilitating the job a journalist usually does when selecting news which is something that, currently can be quite cumbersome. Newsrooms usually have access to worldwide news agencies feeds through a device, *basket*. For each news agency there is a basket and each basket has it own kind of feeds. Those feeds, named *assets*, consist of a pair of files, namely, a video file that contains a news clip and a metadata file that describes all the content of the video file. Based on what has been studied, and how requirements that were established, the next step was

to design the overall architecture. As basis for newsRail's architecture the Bobsled framework would be used. Bobsled is a framework developed by MOG Solutions that will be used as the basis for all their line of products. After studying the framework, one thing became obvious, namely, the fact that Bobsled needed to modified in order to meet some of newsRail's needs and to accommodate REST interfaces alongside the SOAP interfaces Bobsled already had.

After all the modifications had been done, the next task was to implement newsRail. When designing the architecture, it was decided to implement newsRail as a set of extensions that would "plug" into Bobsled. Apart from the a "common extension" that contained common entities to all the other extensions and the operation (which is different and independent from normal extensions), there would be one extension for each of the baskets that newsRail would support. Ending this prototype implementation phase, three unit tests were developed in order to assert that all the required functionalities were correctly implemented and actually functional.

5.1 Objective Completion

When the author set out on to work on this project he had one clear goal: deliver a functional prototype of the newsRail system. By functional one means that it should, at least, support one basket and it should complete its two main worflows correctly. These workflows are:

- **Proxy/Keyframe Generation** — newsRail should be able to connect to a basket, detect new assets, submit job to Carbon Coder in order to generate a proxy and a keyframe from the asset's video file.
- **Export OP1a to F1000** — At the users request, newsRail should be able to submit one or more requests in order to export an asset's video file as a OP1a file to a F1000 hot folder.

One can say that regarding workflows, all objectives were completed. The newsRail system interfaces (SOAP and REST) are completely functional and, if one so wishes, newsRail could be plugged into a SOA infrastructure. Also, newsRail runs both workflows using Reuters assets as test subjects. Regarding baskets, the primary objective was to have the newsRail prototype to support two baskets, namely, Reuters and EBU. However, due to external circumstances, it was impossible to implement the EBU extension. The reason to this was due to the fact that the test EBU assets that were being used had incomplete metadata files and there was no way to have access to complete metadata files. The Reuters extension, as mentioned, has been completely implemented and is completely functional.

All this is supported by the unit tests that were developed after the newsRail prototype was implemented in order to asset that the system behaved as expected. The tests covered three "cornerstones" of the newsRail system, namely:

- **NRailAssetDetector**
- **NRailBaseRule/ReutersRule and NRailBaseFlow**
- **CarbonCoderOperation**

The tests asserted all aspects of implementation, from the overall system workflow to the correctness of the data that was inserted into the DB as well as the events that were thrown/caught and if they were thrown/caught by the correct entities.

5.2 Future Work

As stated in this chapter, the main objective of this project was to deliver a SOA-compliant prototype of the newsRail news selection system and, as mentioned in the previous section, that goal was accomplished. The very definition of prototype states that it's not a finished product, which means that this prototype, even though fully functional, has still some road ahead before becoming a fully developed commercial product. There are some directions where one can take newsRail along. The most obvious is the support of multiple users and user sessions. This functionality, however, is something that wouldn't be used only on newsRail as it would have to implemented in Bobsled itself. This is a decision that affects all of MOG Solutions products and so, it was decided that, in the meantime, this functionality would not be implemented in order to be carefully planned.

Another obvious direction that newsRail can take is the ability to support more baskets. Since the objective of this prototype was to be used almost like a proof-of-concept, one basket would be enough, but in a commercial version newsRail should be able to support several baskets.

Also, there is work that can be done regarding the REST interfaces. As REST becomes more and more widespread, it is only natural that also more and more applications will support it. The biggest advance would be the generation of WADL files just like SOAP's WSDL files. When Microsoft's Visual Studio (the chosen IDE within MOG Solutions) supports REST in the same way it supports SOAP, then it will start to have a more relevant role within newsRail. However, this first step, of supporting REST interfaces was an important one.

As for the SOAP interfaces, the work that remains to be done in this field might be adding more interfaces as newsRail's functionalities also grow. This will allow a better control of the system from any client that wishes to access it via SOAP interfaces. As already mentioned before, the most common way to access in a real newsroom will

probably be through its GUI, however, it might be interesting to see a newsRail system plugged into a SOA infrastructure as this would provide some very important feedback on the currently implemented interfaces as well as providing a way to develop new ones that may better meet the requirements that such an infrastructure usually has.

5.3 Final Conclusion

With the ever growing adoption of IT technologies in the media industry it's only natural that, not only the *medium* enters the digital age, but also the workflows, that were traditionally done "manually" (even if it was using a computer), will inevitably enter the digital age and be as automated as possible. The newsRail system intends to be just that, a first step towards automation which, normally, is something very difficult to achieve in the media industry because many of the processes depend heavily on human interaction. Obviously, one does not want to imply that an ultimate (and utterly Utopian) goal would be to develop a "sentient newsRail" that would automatically select news for the journalists, but it's always important to have a clear view of what the future might hold and act accordingly.

SOA, which has already proven itself in other types of business/enterprise, became the chosen methodology to be used to organize media enterprise's IT infrastructure. The fact that is scalable and, thanks to loose-coupling, extremely agile means that a company that implements it gains an agility and capacity to easily adapt to change which is something extremely important. Especially in the media business, where the only certainty about it is that it's a very uncertain business. SOA, even with some limitations like those mentioned earlier, has spread out into all kinds of enterprises and it's everywhere (badly or correctly implemented), so one can state that the SOA paradigm, that is here to stay and will "grow even stronger", is more than certainty, it's a reality. That said, one can easily see that it's better to board the train as early as possible than to miss at all.

References

[AVI10] AVID. Unity isis, 2010. Available at http://www.avid.com/US/products/Unity-ISIS/.

[CG08] C. Cauvet and G. Guzelian. Business process modeling: A service-oriented approach. In *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, pages 98 –98, 7-10 2008.

[Con04] World Wide Web Consortium. Definition of web service, 2004. Available at http://www.w3.org/TR/ws-arch/#whatis.

[Con07] World Wide Web Consortium. Soap version 1.2 part 1: Messaging framework (second edition), 2007. Available at http://www.w3.org/TR/soap12-part1/.

[Dic10] Dictionary.com. Definition of media, 2010. Available at http://dictionary.reference.com/browse/media.

[DWBT06] Bruce Devlin, Jim Wilkinson, Matt Beard, and Phil Tudor. *The MXF Book: Introduction to the Material eXchange Format*. Focal Press, 2006.

[FF08] John Footen and Joey Faust. *The Service-Oriented Media Enterprise: SOA, BPM and Web Services in Professional Media Systems*. Focal Press, First edition, 2008.

[FFVM08] E.B. Fernandez, M. Fonoage, M. VanHilst, and M. Marta. The secure three-tier architecture pattern. In *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, pages 555 –560, 4-7 2008.

[FT02] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2:115–150, May 2002.

[Gro10a] The Open Group. About the open group, 2010. Available at http://opengroup.org/overview/what-we-do.htm.

[Gro10b] The Open Group. The soa work group : Definition of soa, 2010. Available at http://opengroup.org/soa/soa/def.htm#_Definition_of_SOA.

[GS09] G.G.R. Gomes and P.N.M. Sampaio. A specification and tool for the configuration of rest applications. In *Advanced Information Networking and Applications Workshops, 2009. WAINA '09. International Conference on*, pages 500 –505, May 2009.

[Had06] Marc J. Hadley. Web application description language (wadl). 2006.

[Had09] Marc J. Hadley. Web application description language (wadl), 2009. Available at http://wadl.java.net/.

[Ive06] John Ive. Mxf, 2006. Available at http://broadcastengineering.com/news/broadcasting_mxf_file.

[LDG+09] Martin Lister, Jon Dovey, Seth Giddings, Kieran Kelly, and Iain Grant. *New Media : a critical introduction*. Routledge, second edition, 2009.

[LK10] Yong-Ju Lee and Chang-Su Kim. Building semantic ontologies for restful web services. In *Computer Information Systems and Industrial Management Applications (CISIM), 2010 International Conference on*, pages 383 –386, 2010.

[Mic10] Microsoft. Overview of the .net framework, 2010. Available at http://msdn.microsoft.com/en-us/library/zw4w595w(v=VS.71).aspx.

[mon11] mongoDB. Mongodb, 2011. Available at http://www.mongodb.org/.

[oMPE10] Society of Motion Picture and Television Engineers. Numerical index of standards, recommended practices, engineering guidelines and registered disclosure documents, 2010. Available at http://www.smpte.org/standards/NumberIndex.pdf.

[Ort07] S. Ortiz. Getting on board the enterprise service bus. *Computer*, 40(4):15 –17, april 2007.

[PH07] Mike P. Papazoglou and Willem-Jan Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, 2007.

[PML09] Yu-Yen Peng, Shang-Pin Ma, and J. Lee. Rest2soap: A framework to integrate soap services and restful services. In *Service-Oriented Computing and Applications (SOCA), 2009 IEEE International Conference on*, pages 1 –4, 2009.

[Pre11] Associated Press. Enps - the essential news production system, 2011. Available at http://www.enps.com/.

[Qua11] Quantel. Quantel sq editors, 2011. Available at http://www.quantel.com/page.php?u=de7fa7245fa0e03b1d51ed71c0e1dbab.

[Rho11a] Rhozet. Carbon server, 2011. Available at http://www.rhozet.com/carbon_server.html.

[Rho11b] Rhozet. Rhozet: Carbon coder, 2011. Available at http://www.rhozet.com/carbon_coder.html.

[Rho11c] Rhozet. Rhozet workflow system, 2011. Available at http://www.rhozet.com/rhozet_wfs.html.

[RTP11] RTP. Rádio e televisão de portugal, 2011. Available at http://www0.rtp.pt/homepage/.

[SHM08] Derek T. Sanders, J. A. Hamilton, Jr., and Richard A. MacDonald. Supporting a service-oriented architecture. In *SpringSim '08: Proceedings of the 2008 Spring simulation multiconference*, pages 325–334, San Diego, CA, USA, 2008. Society for Computer Simulation International.

[Sil10] André Silva. mxfspeedrail f1000 - user manual. Technical report, MOG Technologies, February 2010.

[Sol10] MOG Solutions. Mog solutions - the mxf experts, 2010. Available at http://www.mog-solutions.com/.

[TS02] Andrew Tanenbaum and Marteen von Steen. *Distributed Systems - Principles and Paradigms*. Prentice Hall, 2002.

[Twi10] Twitter. Rest api documentation, 2010. Available at http://dev.twitter.com/doc#rest-api.

[WL09] Ying-Hong Wang and Jingo Chenghorng Liao. Why or why not service oriented architecture. In *Services Science, Management and Engineering, 2009. SSME '09. IITA International Conference on*, pages 65 –68, 11-12 2009.

[Xin09] Chen Xin. Service-oriented architecture in business. In *Computing, Communication, Control, and Management, 2009. CCCM 2009. ISECS International Colloquium on*, volume 4, pages 521 –524, 8-9 2009.

[YLBM06] Qi Yu, Xumin Liu, Athman Bouguettaya, and Brahim Medjahed. Deploying and managing Web services: issues, solutions, and directions. *The VLDB Journal*, 17(3):537–572, November 2006.

[YLN00] L. Ye, Y. Luo, and M. Nagata. Xml based message queuing. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 3, pages 2034 –2039 vol.3, 2000.