



João Nuno Silva Tabar Domingos (26333)

Licenciado em Engenharia Informática

**On the Cloud Deployment of a Session
Abstraction for Service/Data Aggregation**

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientadora: Prof^a. Doutora Maria Cecília Gomes
Co-orientador: Prof. Doutor Hervé Paulino

Júri:

Presidente: Prof. Doutor Adriano Martins Lopes

Vogais: Prof^a. Doutora Ana Paula Pereira Afonso
Prof^a. Doutora Maria Cecília Farias Lorga Gomes



FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

March, 2013

On the Cloud Deployment of a Session Abstraction for Service/- Data Aggregation

Copyright © João Nuno Silva Tabar Domingos (26333), Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

A Faculdade de Ciências e Tecnologia e a Universidade Nova de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objectivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Abstract

The global *cyber-infrastructure* comprehends a growing number of resources, spanning over several abstraction layers. These resources, which can include wireless sensor devices or mobile networks, share common requirements such as richer inter-connection capabilities and increasing data consumption demands. Additionally, the service model is now widely spread, supporting the development and execution of distributed applications. In this context, new challenges are emerging around the “*big data*” topic. These challenges include service access optimizations, such as data-access context sharing, more efficient data filtering/aggregation mechanisms, and adaptable service access models that can respond to context changes. The service access characteristics can be aggregated to capture specific interaction models. Moreover, ubiquitous service access is a growing requirement, particularly regarding mobile clients such as *tablets* and *smartphones*.

The *Session* concept aggregates the service access characteristics, creating specific interaction models, which can then be re-used in similar contexts. Existing *Session* abstraction implementations also allow dynamic reconfigurations of these interaction models, so that the model can adapt to context changes, based on service, client or underlying communication medium variables. *Cloud computing* on the other hand, provides ubiquitous access, along with large data persistence and processing services.

This thesis proposes a *Session* abstraction implementation, deployed on a *Cloud* platform, in the form of a middleware. This middleware captures rich/dynamic interaction models between users with similar interests, and provides a generic mechanism for interacting with datasources based on multiple protocols. Such

an abstraction contextualizes service/users interactions, can be reused by other users in similar contexts. This Session implementation also permits data persistence by saving all data in transit in a *Cloud-based* repository,

The aforementioned middleware delivers richer datasource-access interaction models, dynamic reconfigurations, and allows the integration of heterogenous datasources. The solution also provides ubiquitous access, allowing client connections from standard Web browsers or *Android* based mobile devices.

Keywords: *Big Data, Cloud Computing, Sessions, Dynamic Reconfigurations, Mobile Platforms*

Resumo

Hoje em dia, existe um número crescente de recursos na *ciber-infraestrutura* global, distribuídos em várias camadas de abstracção. Estes recursos, que podem incluir redes de sensores sem fios ou redes móveis, têm como denominador comum a necessidade de mais e melhores mecanismos de interacção, bem como uma crescente necessidade de consumo de dados. Para além destes factores, o modelo de serviços está hoje em dia amplamente divulgado, servindo de suporte para o desenvolvimento e execução de aplicações distribuídas. Neste contexto, novos desafios estão a emergir relacionados com o tópico “big-data”. Estes desafios incluem optimizações ao nível dos serviços, tais como a partilha do contexto de acesso a dados, mecanismos mais eficientes de agregação e filtragem de dados, bem como a criação de modelos de acesso a serviços adaptáveis a mudanças de contexto. Estas características de acesso a serviços podem ser agregadas, de forma a capturar modelos de interacção particulares. Adicionalmente, o acesso ubíquo a serviços é um requisito cada vez mais relevante, particularmente no contexto de clientes móveis, tais como *tablets* e *smartphones*.

O conceito de *Sessão* agrega características de acesso a serviços, criando modelos de interacção específicos, que podem ser reutilizados em contextos semelhantes. Implementações de *Sessão* existentes permitem ainda reconfigurações dinâmicas destes modelos de interacção, para que o modelo se adapte a mudanças no contexto, baseando-se em variáveis relacionadas com o serviço, o cliente, ou o meio de comunicação. O *Cloud computing* por outro lado, facilita o acesso ubíquo, e fornece ainda serviços para persistência e processamento de grandes quantidades de dados.

Esta tese propõe uma implementação da abstracção de *Sessão*, instalada numa

plataforma *Cloud*, sob a forma de um *middleware*. Este *middleware* captura modelos de interação ricos e dinâmicos, entre utilizadores com interesses semelhantes, para além de fornecer um mecanismo genérico para acesso a fontes de dados baseadas em múltiplos protocolos. Esta abstracção cria contextos que englobam as interações entre fontes de dados e utilizadores, de forma a que estes possam ser reutilizados por outros utilizadores em contextos semelhantes. A implementação de *Sessão* referida permite também que os eventos sejam persistidos, ao guardar todos os dados em trânsito num repositório baseado na *Cloud*.

O *middleware* mencionado anteriormente oferece modelos mais ricos para interação com fontes de dados, reconfigurações dinâmicas mais ricas, e permite a integração de fontes de dados heterogéneas. Esta solução disponibiliza também acesso ubíquo, na medida em que permite o acesso a clientes através de *Web browsers* comuns, bem como através de clientes móveis baseados em *Android*.

Palavras-chave: *Big Data, Cloud Computing, Sessões, Reconfigurações Dinâmicas, Plataformas Móveis*

Contents

1	Introduction	1
1.1	Problems	3
1.2	Proposed Solution	5
1.3	Contributions	6
1.4	Document Organization	6
2	State of the Art	9
2.1	Cloud Computing	9
2.1.1	Advantages / Disadvantages	10
2.1.2	Cloud Computing Model	10
2.1.3	Provider Examples	16
2.2	Mobile Platforms	22
2.2.1	iOS	23
2.2.2	Android	24
2.2.3	Mobile Platforms and Cloud Services	24
2.3	Patterns	25
2.3.1	Object-Oriented Patterns	25
2.3.2	Architectural Patterns	26
2.3.3	System Integration Patterns	27
2.3.4	Patterns as Abstractions	29
2.4	Enterprise Integration	31
2.5	Session-Based Dynamic Interaction Models	32
3	A Middleware for Service/Data Aggregation	35
3.1	Requirements	35
3.2	Solution Domain	37

3.3	Session Abstraction	38
3.4	On the Use of a Cloud-Based Approach	43
3.5	Architecture	45
3.5.1	Architecture Modules	45
3.5.2	Architecture Extensibility	47
4	Implementation	49
4.1	Inter-Module Communication	50
4.1.1	Route Specification	52
4.2	Data Source Interface	57
4.3	Middleware Core	59
4.3.1	Data Source Messaging Layer	60
4.3.2	Session Messaging Layer	64
4.3.3	Client Messaging Layer	71
4.3.4	Session Container	74
4.4	Client Interface	75
4.4.1	Services API	79
4.5	Cloud Integration	83
4.6	Web Administration	85
4.7	Mobile Client	89
5	Case-Study	93
5.1	Dynamic Data Driven Applications Systems	93
5.2	Urban Flooding Analysis and Monitoring	94
5.2.1	General Considerations	95
5.2.2	Example Description	97
5.2.3	Scenario Evolution	98
6	Conclusions	121
6.1	Discussion	122
6.2	Contributions	123
6.3	Future Work	125

List of Figures

2.1	Session concept	33
3.1	Session abstraction	38
3.2	The extended session abstraction	40
3.3	Session lifecycle	41
3.4	Solution architecture	45
3.5	Solution architecture modules	47
4.1	Implementation overview	49
4.2	Camel and Esper integration	51
4.3	Camel route lifecycle	53
4.4	Generic route builder	53
4.5	Camel Esper base class diagram	56
4.6	Camel endpoint connectors	58
4.7	Middleware core	60
4.8	Camel data source Esper producer routes	61
4.9	Camel internal event class	62
4.10	Data source data model	62
4.11	Polling data source data model	63
4.12	Event driven data source data model	63
4.13	Camel session Esper consumer routes	64
4.14	Session domain entity	65
4.15	Dynamic Reconfiguration data model	66
4.16	Client routes	72
4.17	UserSessionConnection data model	73
4.18	RoutesWrapper object	75
4.19	WebSocket class hierarchy	76

4.20	WebSocketMessage	78
4.21	BaseBean partial hierarchy	78
4.22	WebSocketServices	80
4.23	SessionEventHandler	81
4.24	ClientWebSocketContext	82
4.25	AWS EC2 management console	83
4.26	AWS RDS management console	84
4.27	Web interface authentication	85
4.28	RSS data source creation screen	86
4.29	Twitter data source creation screen	86
4.30	Session condition creation screen	87
4.31	Dynamic reconfiguration creation screen	87
4.32	Session creation screen	88
4.33	Login screen	89
4.34	Home screen	89
4.35	Echo screen	90
4.36	Listings screen	90
4.37	Data source test screen - selection	91
4.38	Data source test screen - connection	91
5.1	Humidity data source definition	99
5.2	Normal humidity topic definition	99
5.3	Publisher-Subscriber interaction model definition	100
5.4	Esper expression adds esper humidity precipitation data	100
5.5	Alert dynamic reconfiguration	101
5.6	Calculate humidity average every 5 minutes	101
5.7	Session definition	102
5.8	Session operations	102
5.9	Local authority connection definition	103
5.10	Connection detail	103
5.11	Session monitoring screen	104
5.12	Fire department alert dynamic reconfiguration	104
5.13	Fire department connection definition	105
5.14	Normal scenario overview	105
5.15	Mobile session connection	107
5.16	Mobile session events	107
5.17	Fire department monitoring new firemen connections	107
5.18	Wind speed and direction data source addition	108

5.19	Emergency dynamic reconfiguration	108
5.20	Emergency dynamic reconfiguration definition	109
5.21	Alert scenario overview	109
5.22	Local authority adds new data sources	110
5.23	Emergency scenario intermediate status	111
5.24	Disaster dynamic reconfiguration	112
5.25	Emergency scenario overview	113
5.26	Mobile client receives disaster notification	114
5.27	Twitter account setup	114
5.28	Aftermath dynamic reconfiguration	115
5.29	Disaster scenario overview	115
5.30	RSS news feed definition	116
5.31	Ground water level values from the last hour expression	117
5.32	Back to normal reconfiguration	118
5.33	Esper expression	118

List of Tables

2.1	Cloud provider comparison	22
-----	-------------------------------------	----

Listings

4.1	Generic route builder	54
4.2	Esper event producer	55
4.3	Esper event consumer	56
4.4	Session esper consumer internal route	67
4.5	Client esper queue broadcasting	68
4.6	Session interaction retrieval	68
4.7	Session pattern operations	69
4.8	Session dynamic reconfiguration processing	70
4.9	Session event persistence	71
4.10	ServerSideWebSocketHelper	77
5.1	Email sending event handler	106
5.2	Android client low battery event	111



Introduction

In the online world of today, the characteristics of what is called the cyber-infrastructure and of the resulting synergies with its users are changing at a fast pace. The cyber-infrastructure [ADF⁺03] comprehends a large diversity of hardware, software, and information resources, spanning several abstraction layers. These include wireless sensor devices, mobile networks, cloud storage and computing services, etc., with a common denominator - their increasing interconnection and integration being supported by high-speed networks.

In the above context, one major characteristic concerns the "*big data*" problem [JMB11]. Increasingly, there is more data being generated which has to be stored, accessed, processed, and disseminated, potentially to a large number of users, and at a global scale. On one hand, there are more sensing devices of all types, from small cheap wireless sensing devices deployed in large scale areas [GHIGGHPD07], to mobile devices with diverse types of embedded sensors [LML⁺10], or to wide-scale sensors in satellites. Additionally, applications in many areas are also generating and processing large amounts of data like scientific/engineering applications [CFK⁺00], but also business applications (e.g. e-commerce/recommender systems [KR12]) including in the novel area of social networks applications [Gao12]. Likewise, users themselves are demanding more information access, either based on fresh data and/or on accumulated historical data, and with a specific *Quality of Service (QoS)* in terms dependability (e.g. providing availability, reliability, maintainability, and security concerns) but also response times.

On the other hand, the challenges concerning big data mining and processing and large scale data access, require high-performance/high-throughput capabilities' support (which may include co-locating data and processing code), also to respond to peaks, both of data production and on data access interest from users/applications. Forms of timely data delivery, and in a way which is perceived as ubiquitous, are hence required for modern applications and users. Moreover, applications and users demand also novel ways on data access and composition (e.g. [FDFB12, LBC10]), including real-time data incorporation [Dar10]. Additionally, the service concept has been extensively used as an uniform and simple way of providing access to the diverse entities of the cyber-infrastructure and to compose them. The trend on *XaaS* (i.e. providing everything as a service) spans the representation of wireless sensor devices as services [KBLK07], the access to federated computational resources (e.g. Grid computing services [Fos06]), to full-fledged remotely accessed applications (e.g. Software as a Service [TBB03, CC66]).

Cloud computing, provides interesting solutions concerning large scale data and processing capabilities, and provides economies of scale, from resource sharing (which, for instance, is essential for scientific data sharing since it is infeasible to move large amounts of data frequently), to on-demand resource provisioning (e.g. dynamic allocation of computational resources to serve peak requests). Cloud computing can be seen as the on-demand delivery of resources and applications, existing in a Cloud, which are accessed as services, hiding the underlying physical and operational resources [Gro10, JNL10]. A Cloud, in turn, can be perceived as a distributed system, backed by hardware and virtual machines, which can be dynamically allocated or released, and which are presented as a unified resource.

Applications that use Cloud services take advantage of features that are usually only available in large data centers. These features include automatic resource scaling¹, data replication and disaster recovery processes. Moreover, Cloud computing services offer resource elasticity, supporting elastic storage or dynamic resource allocation on application usage. Associated with high availability, these applications are accessed ubiquitously, specifically regarding mobile clients interactions such as *smartphones* and *tablets*. In this context, users are provided with ubiquitous access to resources, and mobile users in particular, benefit from Cloud

¹Resource scaling refers to the process of adding resources when they are needed (up-scaling) or removing them when they are not needed (down-scaling)

computing support as a widely available backend storage and computational infrastructure.

1.1 Problems

Although Cloud computing provides a sound support on large-scale storage and computational capabilities, several open problems can still be identified in terms of data/service management and from the user perspective on service/-data access.

First, data management problems include the need to store data, but also share its access, since it is unfeasible to frequently move large amounts of data (e.g. scientific data on the human genome decoding is of interest to a large number of scientists world-wide). Resource sharing, not only of data storage but also of processing power, is hence essential to up port cost effective applications. Namely, the data processing applications, in particular, can be moved near the data, in order to avoid big data transfers whenever possible. Likewise, the reuse of similar big data mining functions (e.g. over the human genome) which are applicable in similar applications, as well as the sharing of such data processing to users/applications with similar interests, is necessary in order to reduce costs, processing times, and CO₂ emissions.

Second, the large number and diversity of possible data sources, e.g. from wireless sensors, entities in the *Internet of Things* domain, Web applications like *HTML* pages, *RSS* feeds, etc, require a uniform access support that may simplify the development of applications that require access and aggregation of several of such data sources. Moreover, the provision of such uniform access, may allow the application of custom defined filters to the acquired data (e.g. application of co-relation functions to data sensing values on temperature and humidity and the topology of the area where those values are collected).

Third, many of the cited data sources produce data streams, over which continuous filtering has to be performed (e.g. real time data mining on users' interests from tweets), either independently over individual data sources or over different streams. Data streams' clustering may also be of interest to different users, e.g. over the same period of time. However, in case the data sources do not produce a continuous flow of data, for instance if they are based on the *Client/Server* model, user applications have to continuously interrogate those sources if they are to produce a continuous flow of data. Again, different applications with interest in the same data sources would benefit from a common service providing

the same data, without the need to individually interrogate the service continuously. The problem is aggravated if the continuous data access requests degrade in some way the data service, e.g. wireless sensor devices have limited autonomy. Data access sharing is hence beneficial in this case, and for the same reasons, selecting an adequate access model depending on the situations, is also beneficial. For instance, it would be useful to interrogate the service with lower periodicity, if the information provided is no longer urgent. Richer interaction models on service access like *Publish/Subscriber* (for notification of subscribed data) or *Producer/Consumer* (producers and consumers do not have to coexist) are interesting options on adapting data access and dissemination to specific context needs.

Fourth, mobile devices are here to stay, and may be themselves sources of data which also has to be managed, but in spite of their limited processing and storage capabilities, they are increasingly used as a front end to distributed services, including on big data management. Although Cloud computing is the backend solution for mobile devices on ubiquitous data storage and processing capabilities, many wireless connections still experience frequent problems and provide a more reduce bandwidth. Problems on data dissemination optimization, e.g. based on mobile devices location and/or common data interests, still have to be considered.

Fifth, the user perspective/interests on data access, processing/aggregation and sharing have to be taken into account. Users not only want to access large quantities of data and be provided with timely extracted knowledge from it, but they also want to share that data and knowledge. Social applications are changing the way entertainment but also business is perceived, and they are one of the sources of the large amounts of data to be processed/mined. Providing users with filtering data mechanisms that they can tune according to their needs, as well as context sharing among users, it is a pressing need for today.

Finally, dynamic adaptation in the above context is still an open problem. On one hand, it is necessary to allow the dynamic modification of which services/data sources to access at any time, depending on the necessities. Moreover, the aggregation/filtering functions should be also dynamically adaptable both to novel data sources as well to users' requirements. Such modifications should be allowed both on-demand but also be automatically triggered, avoiding an explicit interaction with users/applications. Rule-based systems, for instance, allow the inclusion of new rules tuned for new requirements, and also support an automatic triggering of the dynamic reconfigurations. On the other hand, dynamic adaptation is also necessary in terms of which users/applications may be

interested on which services/data at any time. The interaction models in use to support data dissemination also have to be dynamically reconfigurable according to data sources' values, communication status or specific user related variables.

1.2 Proposed Solution

This work proposes a Cloud-based Session abstraction as a way to capture the interaction of a set of users, with similar interests on the access to a set of data sources. The Session abstraction provides the access to different types of data sources via an uniform access, and supports a rule-based system on events' processing. Data is disseminated to users in the session according to specific interaction models like *Publish/Subscriber*, *Producer/Consumer* or *Streaming*.

The Session abstraction can be reused for similar contexts/applications, and can be shared by different users supporting hence a common environment, as well the reuse of the data access definitions. Additionally, the Session is persistent in time, allowing data received from accessed data sources to be stored in a repository, as well as data pertaining the state of the session itself (e.g. how many users exist at any point in time) and any other generated events (e.g. reconfiguration notifications like a new client joining the session). Namely, Session clients may inspect the saved data at anytime.

Moreover, the Session abstraction captures the available dynamic reconfiguration capabilities. These include the possibility to modify the accessed data sources and the interaction model for data dissemination, as well the rules filtering data and the rules that trigger the dynamic reconfigurations themselves (e.g. switching from a *Publish/Subscriber* model to a *Streaming* model). The implemented interaction models are based on the pattern concept and with well-defined reconfiguration possibilities which are conform to the patterns' semantics. For instance, it is possible to implement intra-pattern modifications like tuning the rate of the *Producer/Consumer* interaction module, and switch the interaction module of the session itself, meaning that all clients will possess the same role within the pattern (e.g. on switching a Session to a *Producer/Consumer* model, all clients become consumers).

Finally, being deployed in the Cloud, standalone and mobile clients may access the Session context, with guarantees on ubiquitous access, storage scalability, and data persistence.

1.3 Contributions

Most of the contributions of this thesis are related to the Session abstraction implementation and a Cloud-based middleware.

- *A Cloud-based middleware for the Session abstraction*: middleware Cloud deployment, with Session abstraction support;
- *Heterogeneous datasources*: integrate datasources using multiple protocols in the Session context;
- *Session/client level interaction models*: interaction model definitions at Session or client level;
- *Rule-based dynamic reconfigurations*: automatic dynamic reconfiguration capabilities via a rule-based system, which can be applied at Session or client level;
- *Repository and session replay*: session events storage and replay functionality;
- *Richer aggregation functions*: aggregation functions that allow further definitions, other than datasource selection;
- *Ubiquitous clients*: allow client access using different approaches, such as Web browsers or mobile devices;

This thesis work offers richer interaction models, by using a Session abstraction implementation, which are accessible through a Cloud-based middleware.

1.4 Document Organization

The document structure is organized in six chapters. A description for each of these chapters follows.

- **Introduction** in Chapter 1 describes the thesis context, motivation and problem. This chapter also describes the proposed solution and enumerates the thesis contributions.
- **State of the art** in Chapter 2 describes the technological areas related to the thesis. Each area will be relevant to one or more steps of the solution implementation.

- **A Middleware for Service/Data Aggregation** in Chapter 3 provides a high level view of the proposed solution. The architecture details are discussed and the major components are analyzed.
- **Implementation** in Chapter 4 provides a detailed description on the solution implementation, including the technologies used and how they are interconnected. All the major modules and components are described in detail as well as some of the most important processes.
- **Case-Study** in Chapter 5 includes a detailed scenario that showcases the more relevant features of the implemented solution.
- **Conclusions** in Chapter 6 describes the case-study results and how they respond to the problems and challenges raised in the introduction. The chapter also recapitulates the thesis contributions and scopes some possible future work in this area.



State of the Art

This chapter describes the current state of the art and work of the areas related with this thesis, mainly, cloud computing, mobile platforms and patterns. The last section, in particular, describes solution considerations about how the state of the art sections relate to the proposed solution.

2.1 Cloud Computing

The expression *cloud computing* gained particular media attention in 2006 when *Amazon* launched the *Amazon Elastic Computing Cloud* [Ser] (EC2) service. From then on, many companies launched their own services, sometimes re-branding existing products to compete in the growing cloud computing market. Many aspects of cloud computing were already in use in large datacenters as a way of assuring scalability and reliability [Gro10, JNL10]. Some companies saw an opportunity to profit from the existing resources and created services allowing clients to access the capabilities of their clouds. Consequently, companies found a way to lower their datacenter costs, and the clients gained access to the capabilities of large datacenters. Cloud computing services are closely related to the *everything as a service* (XaaS) concept [AFG⁺09, Hog11].

2.1.1 Advantages / Disadvantages

One of the main advantages of cloud computing is the possibility to develop and deploy applications quickly, without prohibitive start-up costs, and with minimal logistics [JNL10]. Users do not need to scale their systems for peak situations usage because clouds adapt dynamically either by up-scaling¹ or down-scaling² resources, and billing is proportional to resource usage³. Such storage and processing scalability allows for applications to quickly respond to new business and operational requisites, and therefore, cloud-based applications can be easily accessed from the internet using a browser and can have a higher average up-time.

Cloud computing disadvantages include an increased dependency of network connection availability - since data transfers through the internet are usually slower than transfers in private networks, the data transfers in cloud-based applications can become a bottleneck [AFG⁺09]. Moreover, when subscribing to cloud computing services the organizations migrate internal data and processes to the cloud provider infrastructures which can raise a number of security and privacy issues. Additionally, one of the main aspects still to be improved in cloud computing is standardization and the various providers of cloud computing should in the future follow common standards to promote interoperability and portability [BYV08, Hog11]. Users nowadays cannot easily migrate between providers, specially if the systems were developed using proprietary tools. This causes vendor lock-ins and prevents a free market where users can effortlessly change providers if a more competitive product is available.

2.1.2 Cloud Computing Model

The cloud computing model [Hog11, AFG⁺09, JNL10] involves three major actors (*Providers, Brokers* and *Consumers*), three deployment models (*Private, Public* and *Hybrid* clouds) and three delivery models (*SaaS, PaaS* and *IaaS*). Additionally, the main characteristics of cloud computing can be grouped into three categories : *Non-functional, Technological* and *Commercial*.

¹Up-scaling refers to the dynamic increase of resources.

²Down-scaling refers to the dynamic decrease of resources.

³Cloud billing takes into account impacts all the dynamically allocated resources.

2.1.2.1 Actors

The *provider* owns the datacenters on which clouds are based and offers services to access them. These offers take the form of SaaS, PaaS or IaaS.

The *broker* uses provider services to build its own cloud services. These services can extend the capacities of existing services or aggregate multiple existing clouds and present them as a single interface. The broker acts as an intermediate between the provider and the consumer.

The *consumer* makes use of clouds to optimize internal processes, optimize business needs or develop applications that take advantage of clouds. If the consumer develops applications that provide cloud services, he can become a provider as well.

2.1.2.2 Deployment Models

In *private clouds* an organization installs and maintains its own infrastructures and uses cloud computing technologies to implement a cloud according to their needs. This approach provides maximum flexibility on choosing which infrastructure and levels of service are available. However, the organization must handle all performance and scalability issues having full control over its data and processes.

In *public clouds* providers allow public access to their clouds. Users avoid the costs and logistics of installing and maintaining the infrastructure. This approach does not require the user to handle performance and scalability issues and allows the services to be accessed immediately. Service level agreements (SLAs)⁴ are defined according to business needs. Public cloud users lose some degree of control over their data and applications since they do not directly control the resources involved.

Hybrid clouds combine private clouds and public clouds. Organizations that own private clouds can extend their capacities with existing public clouds. In this approach, the organizations can keep sensitive data and processes in their private cloud and delegate the remaining data and processes to one or more public clouds. Such allows benefiting from the cloud performance and scalability while keeping full control over a selected number of resources.

⁴SLAs are client-provider agreements that define the service delivered.

2.1.2.3 Delivery Models

There are three distinct service levels currently available in cloud computing [Hog11, AFG⁺09, JNL10]: software as a service, platform as a service and infrastructure as a service. The *Software as a Service* model was already familiar before cloud computing⁵ but the *Platform as a Service* and *Infrastructure as a Service* are relatively new and are more closely related to the "utility computing" concept since they provide access to computing resources in a way similar to common utilities like electricity and water.

Software as a Service (SaaS)

The *SaaS* concept and its advantages are well known [JNL10, Lou10]. An organization develops an application and controls the maintenance and versioning process providing only a frontend to the user. The application can be accessed from anywhere as long as a network connection is available and there is no need for application specific installation processes, a simple browser is generally sufficient. The data used by the applications can be safely stored in the cloud. At this level, users access full applications as services and pay for a subscription. Most of these applications do not allow a high level of customization. The customizations available are usually similar to adapting a generic business application like an enterprise resource planning (ERP) application to the specific business needs. However the application itself can be a high level software development platform in which case a much higher level of customization is possible (ex. *Force.com*). Cloud computing did not change the concept of SaaS but allowed for SaaS solutions to be developed and deployed more quickly and with lower costs by using PaaS and IaaS solutions.

Examples of *SaaS* include *Google Apps* [Goob] or *Salesforce CRM* [Sal]. *Google Apps* is an application suite that includes an email client (*Gmail*), office productivity suite (*Google Docs*), or a calendar application (*Google Calendar*). *Salesforce CRM* is a customer relationship management application delivered by *Salesforce.com*, designed for backing sales businesses. The application centralizes contacts, customer accounts, and generates reports and business analysis.

⁵SaaS concept is well known in Service-oriented architectures (SOA).

Platform as a Service (PaaS)

In *PaaS*, users access a development platform on which they can develop and deploy their applications in the cloud. *PaaS* providers deliver a set of development tools or an interface that allows the user to choose the tools from a pre-defined set. These tools can include application servers, database systems, and programming languages. Applications can be quickly developed and automatically benefit from dynamic or programmatic scalability, and the reliability of a robust datacenter. *PaaS* providers usually offer APIs so that the developer can access Cloud resources like databases, message queues, and file storage. Usually there are some development restrictions such as incompatible libraries or limited access to the file system. This approach is best suited for users who want to focus on the solution implementation, ignoring the platform specific issues.

Examples of *PaaS* include *Google App Engine* [GAE] or *Red Hat Openshift* [Hat]. The *Google App Engine* allows developing applications written in Java, Python, or Go. Access to the file system is read only, and Java applications cannot start new threads. These limitations can be overcome to a greater extent by using the *Google App Engine* datastore and *Google Task Queues* services. The *Red Hat Openshift* is a *PaaS* layer on top of the Amazon Web Services. *Openshift* allows to develop applications written in *Ruby, PHP, Java, Perl, or Python*. The user can also select the database between MySQL, SQLite, MongoDB or Membase.

Infrastructure as a Service (IaaS)

At *IaaS* service level, users access computing, networking, and storage resources in order to deploy their applications. These resources can be physical but are mostly virtualized. This approach allows maximum flexibility because in most cases the user accesses a virtual machine on which it can install all required libraries and tools. *IaaS* providers may impose restrictions on the type of VM supported such as imposing a specific operating system. Some providers may or may not provide parallelism for applications running in the cloud. The persistence of data stored in VMs may also vary between providers. The main resources available in *IaaS* are virtual machines and users usually have access to a set of tools for monitoring and resource provisioning. New virtual machines can be assigned on demand and quickly started according to user needs.

Examples of *IaaS* include *Amazon Elastic Compute Cloud (EC2)* [Ser] or *Rackspace Cloud Servers* [Rac]. *Amazon EC2* offers machine instances from several categories according to business needs - from micro instances that are suited for small to

medium websites, to cluster compute instances suited for high performance computing. The instance creation and management process is performed using a specific web interface. The instances can be accessed by connecting to the virtual server using a regular SSH client. *Rackspace Cloud Servers* offers a similar service to *Amazon EC2*.

2.1.2.4 Model Characteristics

There are several *non-functional*, *technological* and *commercial* aspects in the Cloud Computing model [Lou10, Gro10].

Non-Functional Aspects

Agility and elasticity describe the ability to continuously adapt to context changes. These changes can be new performance demands, data storage needs, or any other kind. Clouds offer this adaptability by implementing horizontal and vertical up-scaling and down-scaling. Horizontal scaling changes the number of instances while vertical scaling changes each instance capacities. These characteristics allows clouds to build an illusion of infinite resources.

Clouds offer *reliability and availability* by guaranteeing that systems have a high up-time and high fault tolerance. This is achieved mainly by complex redundancy mechanisms that allow systems to operate regardless of most critical hardware or software errors. Redundancy also allows for most maintenance tasks to occur without disrupting the services.

Cloud providers define a level of *quality of service* (QoS) for their cloud service products. The parameters involved can be response times, data throughput, up-time, concurrent connections, or custom parameters agreed with the clients. All of these parameters can be described in SLAs that describe the service delivered.

Technological Aspects

Virtualization is a fundamental aspect of cloud computing since it provides the ability to deliver multiple virtual resources in a single physical resource. This process is transparent to the user and is responsible for the quick up-scaling and down-scaling provided by clouds. Virtualization also offers users the flexibility to choose the hardware characteristics of their resources in order to fulfill the needed requirements.

Multitenancy is the process of sharing the same instance among different users.

This is an important aspect of cloud computing because, similarly to virtualization, it allows providers to have multiple clients using a single physical resource optimizing usage rate. Using virtualization does not necessarily mean that a single instance of a virtual machine is delivered to each client. In a cloud context, it is possible that a single instance, or a virtual machine, is shared among several users. This process, like virtualization, is transparent to the user. Multitenancy raises some complex data protection and sharing issues that must be efficiently addressed by cloud providers.

Cloud providers supply specific *Application Programming Interfaces* (APIs) and *Software Development Kits* (SDKs). These tools allow developers to access each provider's services like databases, file storage, or message queues. They also provide mechanisms to deploy the applications in the cloud. Since there is no standard for developing these APIs and SDKs, applications developed using these tools are not easily migrated to other cloud provider platforms.

Metering is also one main characteristic in cloud computing because it provides the metrics to bill the service usage and it allows users to better understand the applications behavior. Although metering is mainly a commercial aspect, it also concerns technological aspects, e.g. complex software processes are necessary to efficiently measure the resource usage.

Commercial Aspects

Cloud computing development is commercially driven. Providers reduce costs and profit from their existing infrastructures. Cloud users minimize costs by avoiding resource and maintenance expenses, reducing the time needed to deliver applications, and only paying fees proportional to resource usage.

The *pay per use* approach is one of the main economic characteristics of cloud computing - users do not have to invest on acquiring resources since they can agree with a cloud provider which resource package⁶ they prefer and pay a subscription proportional to resource usage.

Time to market can be a fundamental success factor, specially for small and medium companies. Cloud services shorten time to market by allowing developers to focus on the solutions without spending time and effort on infrastructure and logistics issues.

Service Level Agreements (SLAs) define the relation between the provider and the consumer [Gro10]. SLAs define which service package is delivered, describe

⁶The resource package can include instances of virtual machines, databases, message queues among other resources.

each service, define the provider and consumer responsibilities, lists which metrics should be evaluated or which auditing mechanisms will be used. It should be clear how the SLA will evolve over time and what are the consequences if the agreement is not followed.

2.1.3 Provider Examples

Two of the most established cloud service providers today are *Amazon* with their *Amazon Web Services* (AWS) [AWS], and *Google* with the *Google App Engine* (GAE) [GAE]. This chapter focuses on these two solutions and provides an overview of the main services available in each one. The analysis for each provider will be organised in the following topics: Storage, Messaging, Security, and Monitoring.

2.1.3.1 Amazon Web Services

Amazon was one of the first cloud computing providers with the *Amazon Elastic Compute Cloud* (EC2) service. Since then *Amazon* has added many services to the *Amazon Web Services* solution. As the name implies, *Amazon Web Services* [AWS] provides most of the services through web services. Most of the services can be used independently in a standard application simply by creating an AWS account and using the service web service interface. This allows the user to choose which degree of commitment that his application will have with the AWS. Even though this flexibility exists, most of these services are optimized to work with other AWS services.

Examples of these optimizations are near LAN speeds guaranteed between EC2 instances and other services like SimpleDB or Simple Queue Service, or the integration of most AWS services with the Amazon user authentication system (used on Amazon itself). The base of the AWS products is the *Amazon EC2* which provides virtual instances with multiple configurations. This product fits the *Infrastructure as a Service* (IaaS) paradigm. More recently Amazon launched the *Amazon Elastic Beanstalk* which provides a *Platform as a Service* (PaaS) layer on top of *Amazon EC2* instances and the other AWS services. Many AWS services allow the user to select in which region the service will be deployed. This can be used to improve application latency or to address legal requirements regarding data location.

Storage

Amazon SimpleDB is a distributed, highly available non-relational database, based on key-value pairs that provides automatic indexation, and secure https connections. **Amazon Simple Storage Service (S3)** is a cloud file storage service that allows file sizes from 1 byte to 5 terabytes and offers SOAP or REST interfaces. **Amazon Relational Database Service (RDS)** is a relational database service that allows access to cloud-based MySQL and Oracle database instances. **Amazon Elastic Block Storage (EBS)** is a virtual storage volumes system that offers unformatted virtual drives from can be formatted and can support any type of file system. The volume size can vary from 1 gigabyte to 1 terabyte. **Amazon ElastiCache** is a in-memory cache service compliant with Memcached⁷.

Messaging

Amazon Simple Queue Service (SQS) is a highly available, and reliable message queue system in which the queues can be anonymous, shared with another AWS accounts, or restricted to certain ip ranges. **Amazon Simple Notification Service (SNS)** is a publisher-subscriber based message delivery service that uses a "push" based notification system and supports many protocols such as HTTP, EMAIL, or SMS. **Amazon Simple Email Service (SES)** is a transactional mailing system that allows the basic send and receive operations with multiple recipients. Users can send raw or SMTP mails as well as define the mail headers and MIME types.

Security

AWS Identity and Access Management (IAM) is the Amazon user authentication system. Most of the AWS services offer integration with IAM. The user can control how the application resources are accessed based on several conditions that include the user account, the access time or ip address. The access can also be restricted to https. The users can be arranged into groups to easily centralize the security policy.

Monitoring

Amazon CloudWatch is the AWS application monitoring service. CloudWatch can be used in the application directly using the API or can be used through

⁷Memcached is a widely used free, open source, high-performance, distributed memory object caching system.

the Administration Console screen in the AWS website. In this screen, the user can visualize graphics for the selected metrics and statistical information. The user can define alarms based on conditions involving certain metrics. These alarms can trigger actions like up-scaling or down-scaling resources. The user can obtain values and statistical information on a variety of metrics involving each one of the AWS services used.

Development

When developing applications using the *Amazon Web Services*, the user can choose different approaches with a variable degree of AWS coupling. One loosely coupled method is to develop a standard application, just as if no cloud services would be used, but using only certain AWS services such as a cloud database or message queue services. This can be accomplished by using the web service interface for the specified services. Another method, more coupled to AWS, is to install a development environment on a Amazon EC2 instance and develop the application using this environment. In this approach the application is fully deployed in the cloud and the other AWS resources can be accessed with optimal performance.

The user can also choose a PaaS approach by using the Elastic Beanstalk to deploy a pre-configured development platform. In this approach it is not necessary to configure a development environment but the user has to choose from predefined environment configurations. Amazon Elastic Beanstalk is a PaaS solution based on EC2 instances that guarantees automatic load balancing, scaling, and monitoring. Amazon provides a SDK for developing application using Java, Ruby, Python, PHP and .NET programming languages. There is also an SDK for developing mobile application for Android or iOS.

2.1.3.2 Google App Engine

Google App Engine (GAE) [GAE] is a Platform as a Service (PaaS) provided by Google. The underlying resources are transparent to the user which only needs to care about the development. Issues such as resource management, platform configuration, application scaling, and backups are automatically managed by the service.

Storage

Memcache is a high performance, distributed memory caching system, that supports JCache⁸. **Datastore** is a non-relational, transactional, distributed database service built on top of BigTable⁹, that supports JDO¹⁰, and JPA¹¹. **Blobstore** is a large object storage service, where the maximum size of each object is much larger than for Datastore.

Messaging

Task queues is an asynchronous processing queues service. Tasks can be scheduled to run at a specific time or can run in user defined intervals. Task queues can be *Push queues* (default) or *Pull queues*.

- *Push queues* tasks are executed at a predefined rate defined by the user. In push queues, when a task is added, a user defined action is executed (usually a http request), preventing periodic polling for new tasks. In push queues, GAE automatically scales processing according to the processing volume and deletes the tasks after processing.
- *Pull queues* allow the user to define workers that periodically monitor the queue and consume new tasks. With pull queues it is possible to interact with non GAE based applications using the experimental Task Queue REST API. In pull queues the application itself is required to handle resource scaling and task deletion tasks.

Mail is a mail messaging service. It is possible to send emails on behalf of the application administrators or on behalf of google account users. **XMPP** is a chat message processing service. It allows applications to send and receive chat messages, send chat invitations and request user status, among other features. **URL Fetch** is a http communication service that allows GAE applications to communicate with other web resources through http and https requests and responses. **Channel** is a server to client communication service. When using Channel the application creates a persistent connection with the Google servers which allows to send messages to javascript based clients.

⁸Java Specification Request 107 (JSR 107): JCACHE - Java Temporary Caching API

⁹BigTable is a compressed, high performance, and proprietary database system built on Google File System (GFS)

¹⁰Java Specification Request 243 (JSR 243): Java™ Data Objects 2.0 - An Extension to the JDO specification

¹¹Java Specification Request 317 (JSR 317): Java Persistence 2.0

Security

A GAE application can be accessed by anonymous users or can have restricted access. For restricted access the applications can use the Users API. **Users API** is an access management service which provides three distinct authentication methods. Users can be authenticated by their global google account, by a domain specific account, or by an OpenID identifier.

Monitoring

Administration Console is a web interface for application management. From this interface it is possible to create new applications, configure domain names, manage application versions, select the active version, go through application logs, or consult the application datastore. **AppStats** is a API call monitoring tool. By monitoring all API calls it is possible to have statistical information regarding all of GAE services. This tool can be accessed from within the application or by using the provided rich web interface which allows users to visualize statistical information. **Datastore Statistics** allows applications to check the amount of data stored in the Datastore, check the total number of items, and review the most recent updates. **Prospective search** allows applications to register a number of queries that are simultaneously matched against a set of input documents. The matched documents are defined by the user.

Development

Google App Engine currently allows developers to use Java, Ruby, Python, and Go programming languages. Java based GAE applications are based on Servlets and JSPs. Developers can use their own domains or choose a free name from the "appspot.com" domain to host their apps. GAE provides an SDK for each specified language which includes a sandbox for local development and testing that simulates the real GAE application environment. GAE applications run in an isolated executing environment where the underlying resources and execution detail are transparent to the user. However the above execution environment has some limitations. Namely, applications can only contact other applications through the internet by using URL Fetch or Email services; Additionally, applications can only be contacted by receiving http and https requests on standard ports; Moreover, processing can only be triggered by web requests or executing tasks and each request must be answered within 30 seconds.

GAE applications can include source files, static files, deployment descriptors, and other configuration files in a standard WAR structure. GAE uses a customized JVM that allows developing in Java 5 or 6. However, GAE JVM imposes the some restrictions: sockets are not allowed; it's not possible to write in the filesystem; threads cannot be explicitly launched by the application; each request must be answered within 60 seconds; and JNDI access is not available.

2.1.3.3 Comparison

In the **storage** section, AWS offers a relational database service (Amazon RDS) that can be very useful when migrating existing applications to the cloud. Both AWS and GAE offer a non-relational database service (Amazon SimpleDB and Google Datastore), file storage services (Amazon S3 and Google Blobstore), and both include a memory caching services (Amazon ElastiCache and Google Memcache).

In the **messaging** section, both cloud platforms offer an asynchronous queue service (Amazon SQS and Google Task Queues) and an email service (Amazon SES and Google Mail). Amazon provides a generic notification service (Amazon SNS) supporting multiple protocols such as HTTP or SMS, while GAE provides the XMPP service that is specific to the XMPP protocol as the name implies. Additionally GAE includes Channel, a specific server to client communication service which is useful for rich javascript client applications that need data refreshing.

In the **security** section, both solutions allow the applications to use the providers underlying authentication system (Amazon IAM and Google Users).

In the **monitoring** section, Amazon offers CloudWatch which is a comprehensive metric based monitoring service that provides fine grained statistical information on each AWS resource. CloudWatch can be used for global statistics and quota usage information, as well as real time application usage. GAE includes AppStats which is an API calls resource monitoring system that allows developers to identify which calls are using most of the application resources. GAE also includes Datastore Statistics which allows developers to access current and global database statistics. However, GAE does not provide a specific quota status service to monitor each resource quota individually. GAE offers Prospective Search which can be used for monitoring purposes, nevertheless, this is not a pure monitoring feature, .

In the **development** section, both services include SDKs for multiple programming languages and mobile development both in Android and iOS. *Amazon Web*

Services allows developers to use only specific services without a full commitment to AWS. *Google App Engine* requires the application to be deployed in the GAE environment to take advantage of the services.

In Table 2.1 we can find an overview of the main services and features for both providers in each category:

	<i>Amazon Web Services</i>	<i>Google App Engine</i>
Storage	SimpleDB; Simple Storage Service (S3); Relational Database Service(RDS); Elastic Block Storage (EBS); ElastiCache	Datastore; Blobstore; memcache
Messaging	Simple Queue Service (SQS); Simple Notification Service (SNS); Simple Email Service (SES)	Task Queues; Mail; XMPP; URL Fetch; Channel
Security	Identity and Access Management(IAM)	Users API
Monitoring	CloudWatch	AppStats; Datastore Statistics; Prospective Search
Development	Possible to use only specific services	Deploy needed to use GAE services

Table 2.1: Cloud provider comparison

2.2 Mobile Platforms

Mobile devices and tablets are becoming common in modern society in what can be seen as a manifestation of ubiquitous computing, with mobile device owners demanding internet connectivity everywhere and as many associated related services as possible. In this context of global connectivity mobile devices are becoming prominent computing service consumers in general, and of cloud computing service clients in particular.

Mobile devices have evolved largely in recent years. Hardware advances such as faster processors, better screen resolutions, or larger memories, augmented the functionality of these devices to a point where, nowadays, their capabilities are comparable to many laptop computers.

Parallel to these hardware advances, the operating systems have also evolved and the struggle to gain advantage in this growing market has distinguished two companies: Apple with iOS [Appb] and Google with Android [Gooa]. Each of these companies has developed its own mobile operating system and this is,

nowadays, one of the main criterias consumers look for when selecting a mobile device.

2.2.1 iOS

Apple introduced iOS [Appb] with the release of the first iPhone in January 2007. The operating system was built based on the existing Mac OS X that Apple used in its laptop and desktop computers, and it is currently used in the iPhone, iPod Touch, and iPad devices. The operating system includes a number of core applications such as the Phone, Mail, and Safari browser applications. The most recent iOS version is iOS 5.0.1 (9A406) and was released in December 12, 2011.

Native applications are created using the iOS system frameworks and the Objective-C language. Native applications are installed physically on a device and they reside next to other system applications. Applications and user data can be synced through iTunes. The iOS Software Development Kit contains all the tools and interfaces needed to develop, install, run, and test native iOS applications.

Similarly to Mac OS X, iOS acts as an intermediary between the underlying hardware and the applications that appear on the screen. Native applications communicate with the hardware through a set of well-defined system interfaces that protect native applications from hardware changes. iOS architecture is divided in four layers: Cocoa Touch layer, Media layer, Core Services layer, and Core OS layer. When developing iOS applications the top layer should be analyzed first and only if it does not provide the required functionality should the developer use the other layers.

The Cocoa Touch layer provides the key frameworks for building iOS applications such as multitasking, touch based input, and push notifications. The Media layer includes the graphics, and the audio and video technologies needed to develop multimedia applications. The Core Services layer contains the fundamental system services transversal to all applications. Usually the developer does not need to use this layer directly but the top layers use the features it provides. The Core OS layer includes low-level features that most other technologies are built upon. Developers only need to use this layer directly, in situations where they explicitly deal with security issues or when communicating with an external hardware accessory.

2.2.2 Android

Android [Gooa, SRS⁺09] was first announced in November 2007 and is nowadays the leading mobile operating system in terms of market share [c111, com11]. The commercial breakthrough happened in October 2008 when the T-Mobile G1 smartphone was launched, in the United States, with the Android 1.0 mobile operating system. Google was the initial developer of Android and was later backed by the Open Handset Alliance (OHA)¹².

Android is a highly optimized Linux based mobile operating system that uses the Dalvik virtual machine (DVM), which is a register-based Java Virtual Machine optimized for Android that focused on memory efficiency and application runtime performance. Android runs a DVM per application, and these applications communicate with each other by using the Android Interface Definition Language (AIDL). To be precise, Dalvik is not exactly a Java virtual machine because Dalvik can not read standard Java classes. An additional step is need to convert Java classes to DEX, the DVM custom byte code.

A drawback of the Android mobile operating system it is the incompatibility with Java SE or Java ME class libraries, because DVM does not handle standard Java bytecode. Since android Java libraries are based on Apache Harmony runtime¹³ implementation and not Sun (Oracle) runtime, standard Java SE libraries cannot be directly used in Android. However, there are equivalent Android classes for most Java SE libraries.

Android applications are developed using a set of Java based libraries developed by Google. The libraries along with the core applications and tools compose the Android Software Development Kit. The Android SDK includes a mobile device emulator which is a virtual mobile device that runs in the developer workspace.

2.2.3 Mobile Platforms and Cloud Services

Both iOS and Android offer cloud interaction capabilities. In iOS users can access iCloud [Appa] which is data storage cloud backup service. iCloud offers 5GB of free storage for backing up contacts, music, photos, documents and other data. Android offers native applications for interacting with SaaS Google Apps [Goob] such as Gmail, GDocs, Calendar and contacts. In both mobile platforms this data can be accessed by other devices by using the same account. This allows

¹²Open Handset Alliance (OHA) a confederation of 50 Telecom companies, mobile hardware and software companies headed by Google.

¹³Apache Harmony is a modular Java runtime with class libraries and associated tools

users to change devices without worrying about losing contacts, documents or other previously backed up data.

2.3 Patterns

Patterns describe behaviors and processes which can be reused in different contexts. They are particularly useful in software development as a way to capture knowledge regarding component and service interactions. A user may partially or completely design an application by combining the most adequate patterns for each functionality.

Patterns were initially by Christopher Alexander [AIS77], which wrote “*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*”. Patterns have since then evolved and today assume different forms that include: object-oriented patterns [GHJV95], architectural patterns [BMR+96], system integration patterns [HW03] or more recently pattern templates [GRC08].

2.3.1 Object-Oriented Patterns

Object oriented design makes extensive usage of design patterns. The main object-oriented patterns described in the *Gang of Four* book [GHJV95] can be divided into three major groups : creational, structural and behavioral:

- **Creational design patterns** describe forms by which an object can be created, and constraints in creation process. Examples of creational design patterns include the *Builder* and *Singleton* patterns:
 - *Builder*: This pattern describes how a complex object creation should be separated from its representation thus allowing different object representations to use the same creation mechanism.
 - *Singleton*: This pattern describes the process by which a single instance can exist for a given class.
- **Structural design patterns** describe common object relationship contexts. Examples of structural design patterns include the *Composite* and *Decorator* patterns:
 - *Composite*: This pattern describes a tree like object hierarchy where each group of object implements a specific interface.

- *Decorator*: This pattern describes the process of adding functionality to an object at runtime by wrapping the original class.
- **Behavioral design patterns** describe common object communication contexts. Examples of behavioral design patterns include the *Iterator* and *Observer* patterns:
 - *Iterator*: This pattern describes how to sequentially access elements, without exposing their representation.
 - *Observer*: This pattern describes how object can register as listeners to a certain event that should be raised by another object. This pattern is also know as Event Listener.

2.3.2 Architectural Patterns

Architectural patterns describe fundamental structural organization schemas for software systems and provide the means to specify the fundamental structure of applications [BMR⁺96]. These patterns help to identify a system subsystems, specify their responsibilities and define how the subsystems should relate and communicate with each other. Each architectural pattern helps to achieve a specific global system property, such as the adaptability of the user interface. Patterns that help to support similar properties can be grouped into four categories:

- **Structure Patterns** define the system architecture. They provide a method to divide the system into its constituent parts. Examples of these patterns include the *Layers* pattern and the *Pipes and Filters* pattern:
 - *Layers*: This pattern groups the system components with a similar level of abstraction into groups, and each of these groups defines a system layer. These layers should be loosely coupled.
 - *Pipes and Filters*: This pattern structures systems in terms of data streams and processing steps. By encapsulating each step in a filter component, and passing data through the pipes that connect adjacent filters, it's possible to build families of related systems.
- **Distributed System Patterns** describe common distributed system scenarios. An example of such patterns is the *Broker* pattern:

- *Broker*: This pattern structures distributed software systems with decoupled components that interact with each other using remote service invocations. Broker components are responsible for coordinating communication.
- **Interactive Systems Patterns** describe common user interaction scenarios using graphical user interfaces. Examples of these patterns include the *Model-View-Controller* pattern and the *Presentation-Abstraction-Control* pattern:
 - *Model-View-Controller*: This pattern divides an interactive application in three layers. The model contains core functionality and data, the view displays information to the user and the controller handles all user inputs.
 - *Presentation-Abstraction-Control*: This pattern structures interactive applications in a hierarchy of cooperating agents, where each agent is responsible for a specific functionality.
- **Adaptable System Patterns**: describe common system evolution context and mechanisms for adapting to these evolutions. Examples of these patterns are the *Reflection* and *Microkernel* patterns:
 - *Reflection*: This pattern allows for dynamic structure and behavior changes by dividing the application into two levels: a meta level and a base level.
 - *Microkernel*: This pattern isolates a minimal functionality core from peripheral functionality. Additional functional extensions are coupled to the microkernel.

2.3.3 System Integration Patterns

System integration deals with the heterogeneous system communication, and for capturing these scenarios various integration patterns have been identified. In many contexts the system involved can be abstracted as services and the majority of these integration patterns can be applied as service interaction patterns. System integration patterns [HW03] deal primarily with messaging systems since messages are a generic form of data transportation. These patterns can describe operations on the messaging system or on the message itself. System integration patterns can be divided in the following types:

- **Messaging System** patterns describe high level interactions between the applications and the messaging system. Examples of system patterns include the *Router* and *Translator* patterns:
 - *Message Router* pattern decouples individual processing steps so that messages can be passed to different filters depending on a set of conditions. The Message Router differs from the most basic notion of Pipes and Filters [BMR⁺96] in that it connects to multiple output channels.
 - *Message Translator*: describes how systems that use different data formats can communicate with each other using messages. The Message Translator a messaging equivalent to the Adapter pattern [GHJV95].
- **Messaging Channel** patterns describe how messages are delivered or broadcasted to the receivers. Examples of channel patterns include the *Message-Bus* and *Publish-Subscribe*:
 - *Message Bus*: describes how separate applications can work together in a decoupled fashion so that each application can be easily added or removed without affecting the others.
 - *Publish-Subscribe Channel*: describes how a sender can broadcast an event to all registered receivers. The pattern has one input channel that splits into several output channels, one for each subscriber.
- **Messaging Endpoint** patterns describe different contexts regarding endpoint message consumption. Examples of endpoint patterns include the *Polling Consumer* and *Selective Consumer* patterns:
 - *Polling Consumer*: describes the process by which receivers monitor senders to consume messages when they are ready. This pattern is also known as synchronous receiver since the receiver blocks until a new message is received.
 - *Selective Consumer*: describes how consumers filter which messages they would like to receive. The consumer only receives messages that are available on a registered channel and pass a certain filter.
- **Message Construction** patterns describe message building behaviors. Examples of these patterns include the *Correlation Identifier* and *Return Address* patterns:

- *Correlation Identifier*: describes how the requestor knows which request does a received reply relates to. Each reply message should contain a unique identifier that indicates which request message this reply is for.
- *Return Address*: describes how repliers know where they should send the replies. A return address should be part of the request so that the replier can ask the request where it should send the reply.
- **Message Routing** patterns describe how messages can move between components. Examples of these patterns include the *Splitter* and *Aggregator* patterns:
 - *Splitter*: describes how to process messages that contain multiple elements and each element can be processed in a different way. The Splitter breaks composite messages into individual messages.
 - *Aggregator*: describes the process of combining results of related messages so that they can be processed as a single message. By using a stateful filter it's possible to collect and store each message until a complete set of related messages has been received.
- **Message Transformation** patterns describe operations that change the message. Examples of these patterns include the *Normalizer* and *Content Filter* patterns:
 - *Normalizer*: describes the process of handling semantically equivalent semantically messages that are received with different format. The Normalizer applies a Message Translator to each message so that they are transformed to a common format if they are equivalent.
 - *Content Filter*: describes how to deal with large messages when only certain parts of the messages are relevant. The Content Filter removes non relevant data from the message leaving only the relevant items.

2.3.4 Patterns as Abstractions

Patterns capture structured solutions for common problems based on the experience of experts. Patterns also capture commonly recurring aspects of component and services interaction. These patterns can be grouped into two distinct groups [GRC08, GRC03] : structural and behavioral pattern templates.

- **Structural patterns** describe components and services connectivity. They represent how these components can be grouped, creating distinct structures to perform a certain operation or processing task. Furthermore, structural patterns be embedded in other structural patterns, forming hierarchies of patterns. Patterns in this group describe topologies like a *Ring*, a *Star* or a *Pipeline* or structural design patterns such as *Facade*, *Proxy* or *Adapter*. Examples of structural patterns include:
 - *Star*: This pattern consists of a nucleus that communicates through simple connectors to a number of satellite components.
 - *Pipeline*: This pattern represents a sequence of ordered stages, where one stage produces data to the next.
 - *Ring*: This pattern can be seen as an extension to the pipeline where the last stage is connected to the first.
 - *Facade*: This pattern is used to restrict access to a set of sub-systems, through a common interface.
 - *Proxy*: This pattern allows the local presence of an entity's surrogate which transparently supports access to the remote entity.
 - *Adapter*: This pattern allows communication between two elements when they do follow the same interface.

- **Behavioral patterns** describe dependencies among a set of component and services at runtime, by defining the aggregated runtime behavior. Behavioral patterns do not necessarily specify the exact components and services involved. Patterns in this group capture temporal control and data flow dependencies between components such as loops and execution order. Behavioral patterns can also describe synchronization and interaction constraints between the components. Examples of behavioral patterns include *Client-Server*, *Publish-Subscriber*, *Producer-Consumer*, *Streaming* or *Master-Slave*:
 - *Client-Server*: This pattern involves a server application that is accessed by multiple clients. Clients issue requests and synchronously wait for a server response.
 - *Publisher-Subscriber*: This pattern describes how message senders (Publishers) publish messages without explicitly stating the message destination. The published messages belong to a type and only the message receivers interested in this message type (Subscribers) will receive them.

- *Producer-Consumer*: This pattern describe the message senders (Producers) send messages to a message block, and how message receivers (Consumers) read these messages from that message block.
- *Streaming*: This patterns describes how a continuous flow of data is broadcasted and captured.
- *Master-Slave*: This pattern describes how a main component (Master) distributes work to identical components (Slaves) and computes a final result from the results returned by slaves.

This separation between structure and behavior [GRC03, GRC08] allows to compose systems in which the behavior is captured by behavioral templates, and component connections are captured by structural patterns. This specification allows more flexibility in system definition because the system components can switch behavior or structure patterns independently, allowing the system to adapt to context changes.

2.4 Enterprise Integration

Integration tools allow the creation of networks where each node can be a heterogeneous sub-system or application and each of these nodes can use a specific protocol/technology. By using these integration tools users can abstract from each node specific characteristics and define routing rules the control how the information is transferred between nodes. Usually these rules are based on some of the patterns described in Section 2.3.3. In a cloud computing context, these nodes may involve some of the cloud computing services described in Section 2.1. Examples of integration tools are *Spring Integration* [Spr] and *Apache Camel* [Cam]:

- *Spring Integration* is a lightweight messaging system that supports system integration via declarative adapters. These adapters provide higher-level abstractions that support messaging, and scheduling operations. *Spring Integration* focuses on providing simple models for building enterprise integration solutions, while maintaining the separation of concerns required for producing maintainable and testable code. *Camel* also supports well known *enterprise integration patterns* [HW03] that can be used for workflow composition.

- *Apache Camel* is an open source integration framework that allows the use of patterns in system composition. Users can implement routing and mediation rules using *XML* configuration files or a specific *Java based Domain Specific Language (DSL)*. Furthermore, Camel can work with a number of transport messaging models such as *HTTP*, *ActiveMQ* or *JMS*, and the same API is used for all transport models. Camel allows workflow composition using a number of *enterprise integration patterns* [HW03], and provides cloud service integration for *Google App Engine Task Queues* [GAE] and *Amazon Simple Queue Service* [AWS].

2.5 Session-Based Dynamic Interaction Models

The concept of Session-based dynamic interaction models [BGP12] applies the notion of session to capture client-service interaction configurations. The interaction context can be shared by multiple clients and each clients can join or abandon an active session. The client that created the session is the session owner, and the owner can explicitly change the session interaction model. If the owner modifies the session, all clients connected to that session will be affected by these changes. If a client other than the owner requests a change in the session configuration, he will be moved to a session with these characteristics, or to a new session if such a session does not exist. The session maintains a context for these interactions, and this context specifies: the session owner; the services being accessed by clients; the current interaction model, and optional quality of service properties.

By decoupling services and interaction model, it is possible to apply dynamic changes to the interaction model at runtime, according to eventual context changes in the client, services, or communication medium. This flexibility allows to create dynamic reconfiguration behaviors. Figure 2.1 describes the session structure:

In this work the session concept is used to capture interactions context between clients and cloud services that serve as interfaces to multiple datasources. In this context the session should also be able to define the possible dynamic reconfigurations, either based on the mobile device state, cloud metrics or user defined reconfiguration functions.

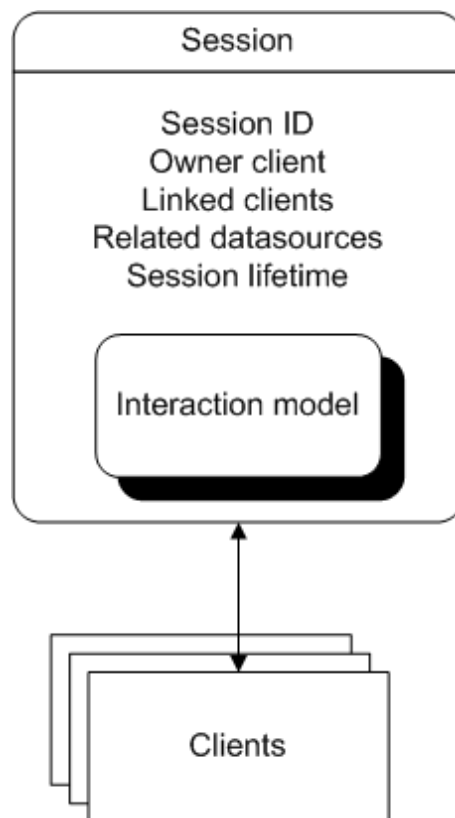


Figure 2.1: Session concept



A Middleware for Service/Data Aggregation

This chapter offers an overview of the solution implemented in this thesis. This overview includes the initial requirements list, the key domains that relate to these requirements and a high-level description of the solution architecture.

3.1 Requirements

The goal of this work is to develop a middleware for supporting supporting the access to heterogeneous data sources via richer and dynamically adaptable interaction models. This access can be reused among clients with similar interests hence supporting a shared context. The key characteristics identified to build such a system are described in the following.

Access to heterogeneous data sources: Clients should be able to access and aggregate several heterogeneous data sources, and apply custom defined filters to the acquired data. It should be possible to integrate data sources that use different protocols and belong to different domains, so the solution should not be tightly coupled to any specific protocol or domain;

Richer interaction models on service access: The solution should provide additional interaction models besides the standard client/service model, in order

to better capture the characteristics of the accessed service and its clients' requirements. Examples may be a streaming interaction model with a constant pre-defined rate; a producer/consumer model supporting a true decoupling between data generation and data acquisition; or the possibility to select a subset of the overall generated data, in order to reduce data processing requirements. Moreover, the solution should also allow the definition of the particular interaction model to use in a particular situation.

Dynamic reconfiguration mechanisms of interaction models: The solution should allow dynamic reconfiguration mechanisms, either on-demand or automatic / pre-defined. These mechanisms should include: changing the number of data sources accessed at some point in time, adapting the parameterization of the current interaction model (e.g. data rate modification on a streaming interaction model triggered by a pre-defined alert threshold) and changing the interaction model in use (e.g. switching to a producer/consumer interaction model to reduce data loss); These dynamic reconfigurations should be "rule-based", i.e. triggered according to a set of rules applicable at some point in time. These rules should relate to:

- *Data source* related information (e.g. wireless sensor generated data);
- *Session* characteristics (e.g. number of clients in the session);
- *Client* characteristics (e.g. location or battery autonomy in case of mobile clients).

Context sharing among clients: The solution should allow that multiple clients with similar interests may share the same data source access context. This improves reusability since each individual client does not have to set up the access to different services/data sources if a similar access already exists. This reusability offers some performance optimization since one data source connection may be shared by multiple clients. Moreover, allowing multiple clients to access the same data under the same conditions creates common information views which could be used to make decisions dependent on that common information (e.g. a single interaction model modification can trigger similar actions on all clients sharing the same context).

Large data storage and processing: The solution should be scalable so that large quantities of data can be accessed, stored, and processed. This is necessary

in domains where the data to be processed is generated in large quantities (eg. weather simulation applications);

Ubiquitous access and service reliability: The middleware should have high availability and fault tolerance. The solution should also allow mobile clients to connect and offer some optimization for these devices;

3.2 Solution Domain

In order to satisfy the above requirements, the proposed solution incorporates concepts/technologies from the domains described in the following.

Session abstraction: The *session concept* is commonly used to contextualize the interaction between a service and its clients, and the work in [BGP12], in particular, defines a session abstraction that provides dynamic reconfiguration mechanisms on service/data access, aggregation, and dissemination of results to clients with similar interests. These mechanisms include the dynamic addition of services/data sources, as well as pattern-based interaction models for data dissemination which can be dynamically adapted according to their suitability for several scenarios (eg. streaming for constant data flows or publisher-subscriber to broadcast data that relates to a specific topic). This session abstraction offers richer/dynamic interaction models on service/data access and aggregation, defines a common reusable interaction context, and can hence be used to support some of the requirements previously described;

Enterprise integration tools: Existing *enterprise integration tools* [Cam, Spr] allow the connection of heterogeneous endpoints in message oriented contexts. Using a tool specific notation it is possible to create routes that connect these endpoints. This ability allows integration tools to aggregate heterogeneous data sources by implementing custom endpoints that support the necessary protocols;

Complex event processing: By using *complex event processing (CEP)* tools like *Esper* [Esp] clients can define complex aggregation functions that operate on multiple sources. Complex event processing tools also allow clients to apply processing logic so that meaningful events are filtered and processed (e.g. detect if the values generated by a sensor decreased in a certain period of time);

Cloud service providers: *Cloud providers* [AWS, GAE] offer services that allow clients to upscale resources according to the context. This ability enables cloud services to respond to large data storage and processing requirements. Cloud service providers offer high availability, uptime and *Quality of Service (QoS)*. Most cloud providers also offer support and optimized services for mobile devices operating systems [Gooa, Appb]. These characteristics can answer ubiquitous access and service reliability requirements;

Since this work extends the session abstraction described in [BGP12], the following section describes the dimensions of this abstraction in more detail, as well as how it was extended in order to respond to the requirements identified in this section.

3.3 Session Abstraction

The server/clients interactions can be characterised by how the clients perform requests, how the services respond to these requests, or how the communication between clients and the services' platform is managed. The sum of all these characteristics define a specific *interaction model*. In this context, a the *Session concept* [MP95, HKLP05] can be used as an abstraction that encapsulates these characteristics and describes interaction model contexts, as described in previous works [BGP12, GPBdSA12].

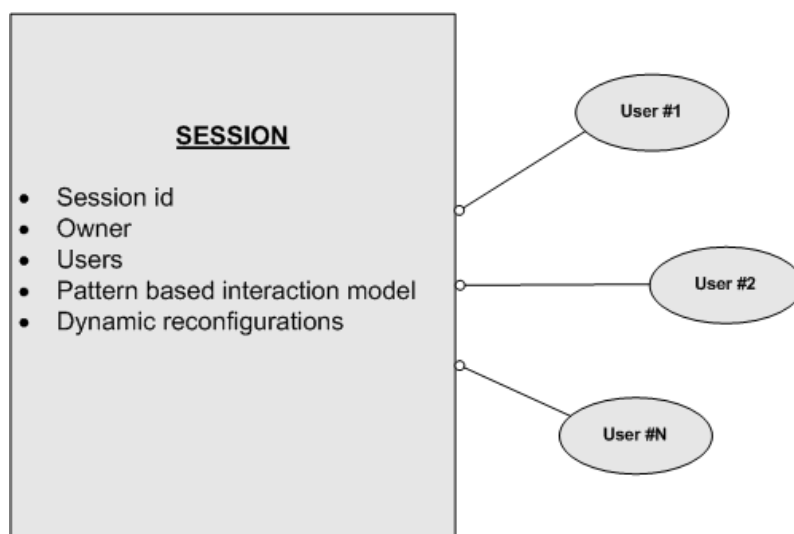


Figure 3.1: Session abstraction

This existing session abstraction is used as a guideline, and the main characteristics are described in the following. The extended session abstraction proposed in this work is described subsequently. The main characteristics of a session as proposed in [BGP12] are:

- A **unique id** identifies a session in the system and thus can be used by clients to reference the session they want to connect to;
- Each session has an **owner** which is the user that created it. The owner has special privileges and is the only user allowed to execute administrative session operations;
- The session may include an **aggregation function** that will operate on the data generated by the session related data sources. This allows message filtering from multiple sources according to what is defined in the aggregation function;
- **Pattern-based interaction models** are implemented between the middleware and the session clients. These patterns are the *Publisher-Subscriber*, *Producer-Consumer* or *Streaming*;
 - **Publisher-Subscriber:** This interaction model is used in case the clients are interested only in a specific topic, and therefore only a subset of events are delivered to the session clients;
 - **Producer-Consumer:** in case the clients want to consume events at their own rate with guarantees on event delivery, this interaction model supports a true decoupling from producers and consumers. The clients can connect / reconnect to the session when they see fit, and from that moment on they will be able to receive all produced events;
 - **Streaming:** this model is used when the clients want to receive a continuous stream of events. There is no guarantee of event delivery when using this interaction model pattern;
- The session supports a set of **dynamic reconfigurations**. These reconfiguration definitions trigger modifications in the session interaction model when specific events are generated. An example of such an event is when a data source, that produces numeric values, generates a value above a certain threshold pre-defined in the dynamic reconfiguration rule.

For example, an existing client is receiving data from a humidity sensor using a *Producer-Consumer* interaction model. The same client specifies a dynamic reconfiguration to be triggered when the humidity passes 70%, and alter the current interaction model to a *Streaming* based interaction model.

This particular take on the concept of session answers some of the requirements identified for this solution, by providing rich interaction models and dynamic reconfiguration mechanisms. In order to fulfill the enumerated requirements, in this work we have extended this sessions abstraction described above by adding some characteristics such as the possibility of integrating domain independent data sources or storing large amounts of data. The extended session abstraction defined in this work is illustrated in Figure 3.2 and the added characteristics description follows.

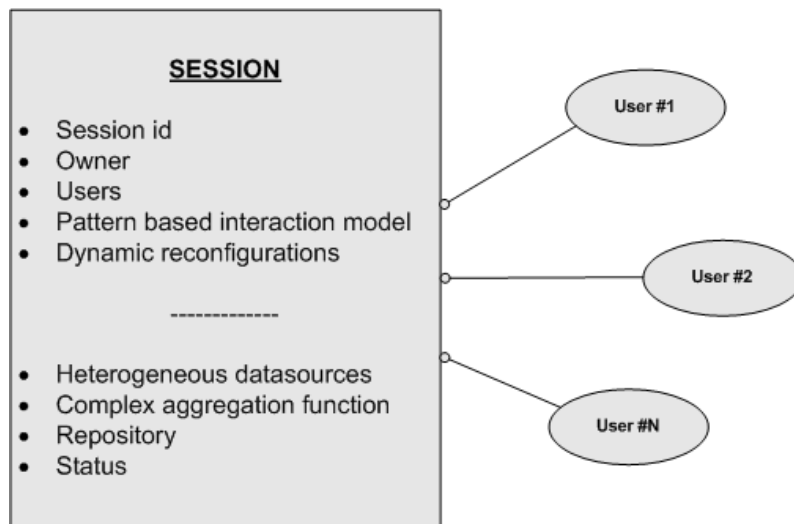


Figure 3.2: The extended session abstraction

The implementation of the extended session abstraction proposed in this thesis admits a set of **heterogeneous data sources**, meaning that they are not tied to a specific protocol like web services. This is opposed to the available implementation of the session abstraction previously described, which was focused on the domain of web-enabled wireless sensors network. Therefore, the web services protocol was the only data source type supported by that implementation;

A **complex event processing tool** is used to define the aggregation functions associated with the session. This allows clients to define data source aggregations

and specific events' detection expressions either on the session definition stage (inactive session) or in real-time (session is active);

The session is now persistent and stateful. The notion of **session repository** is added so that all generated events produced in the context of a particular session can be stored. All the stored events can be completely or partially reproduced at any given moment, thus allowing the replay of session events for posterior analysis (e.g. *offline* weather data analysis for simulation purposes). Stored events can be external events or internally generated events:

- *External events* are mainly data sources' generated events that are included in the session (e.g. events generated by a wireless humidity sensor, acting as a session data source);
- *Internally generated events* include all session related events such as dynamic reconfigurations, client connection/disconnection or session status modifications. For example, when a dynamic reconfiguration that changes the interaction model from *Streaming* to *Producer-Consumer* is triggered, an internal event is generated. This generated event can also be stored and later reproduced;

The session is equipped with a **notion of lifecycle**. Between session creation and termination there are intermediate non-final statuses that allow sessions to be paused/resumed while maintaining the session context (e.g. connected clients). All clients that can connect to the session can consult the session status, but only the owner can perform operations that modify the status (e.g. pause/resume). The possible session statuses are *New*, *Starting*, *Started*, *Stopping*, *Paused* and *Ended*. The session lifecycle is illustrated in Figure 3.3.

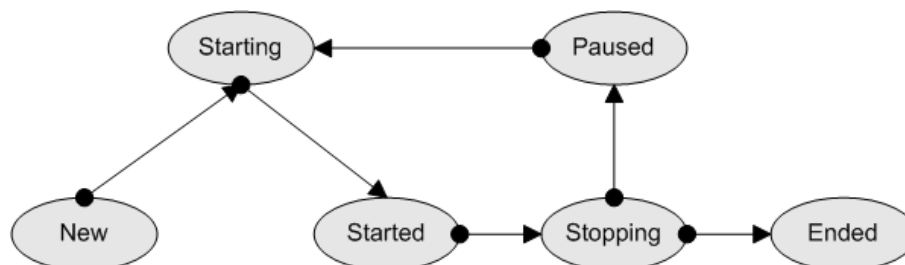


Figure 3.3: Session lifecycle

A description for each possible session status follows.

- **New:** This is the initial status, when the session is created but was never started. At this point the only possible session operation is to start it;
- **Starting:** This is an intermediate status when the session is performing all necessary initializations. This status can occur initially, on starting the session (i.e. *New*) or when resuming from a previously paused session (from status *Paused*). No session status operations are possible at this stage (e.g. stop or pause the session);
- **Started:** This status indicates that the session is fully functional. There are two operations possible at this stage, pause the session (to status *Paused*) or end the session (automatic process on session lifetime expiration);
- **Stopping:** This is an intermediate status indicating that all the necessary processes are being stopped. This status can occur when manually pausing a session or when the session lifetime has expired while the session is running (while being at *Started* status). No session status modification operations are possible at this stage (e.g. start or pause the session);
- **Paused:** this status indicates that the session has been explicitly paused (from status *Started*). The session can be restarted at this stage;
- **Ended:** this status occurs when the session lifecycle has expired (from any status). The session cannot be restarted at this point. However the session definition is maintained all the events generated by the session are kept in the session repository. This allows to perform session replay operations even after the session has ended;

3.4 On the Use of a Cloud-Based Approach

Due the cloud platforms capabilities discussed in 2.1 the deployment of a session abstraction in one of these platforms, provides an easier ubiquitous access to session clients, as well as the provision of a large/persistent storage capacity for the session's state. These characteristics are described below.

Databases are easily scalable when deployed in a cloud service system since in theory it is possible to have almost unlimited storage space, being the associated cost its only limitation. Data sources generating large amounts of data to be persisted may benefit from this storage scalability characteristics. For example, if a certain running simulation produces an amount of data that is much larger that it was initially expected, thus causing the available storage to reduce dramatically, it is possible to request an increase of the database storage space without interrupting the running processes. This can be performed on demand or based on pre-defined conditions (e.g. when data limit reaches 90%, duplicate the storage data);

Performance issues can also be handled by upscaling the hardware where the middleware is deployed. It is possible to provide faster processors and more memory as a response to increased performance demands. Clients that perform complex operations on large portions of data (e.g. simulation clients) may benefit from the performance scalability available in cloud deployed applications. For example, if the hardware requirements for a certain simulation were underestimated and the total time prediction is much larger than required, it is possible to *upgrade* the current hardware to higher-end specifications so that the simulation can be completed more quickly;

Applications deployed in the cloud (ou in a cloud platform) offer high *service reliability* by offering end-users advanced error recovery mechanisms, that previously were only accessible in large data centers. Cloud providers also use complex redundancy mechanisms so that most of the hardware maintenance operations are performed transparently, without compromising cloud deployed application. By combining these features, cloud-based applications offer high *QoS* levels such as fast response times and above average application up times. These characteristics are particularly important in the context of critical real-time applications (e.g. health or aeronautical applications);

Cloud service providers usually offer *mobile device optimized services*. Combined with the *service reliability* factors describe before, these services are favorable to ubiquitous contexts where there is an increased dependency of service connectivity. Scenarios where mobile clients are required to interact with the middleware and take part in sessions will benefit from these features. For example, a session involving a fire department, with firemen deployed on the field that need to receive real-time information regarding the weather conditions and fire progression;

In the context of session concept described before, the following cloud-based features will be used:

- The session repository will be implemented using a cloud deployed database instance, thus taking advantage of all the database features previously described. The repository will benefit from cloud scalability by using high performance storage services that can scale and take into account large amounts of data while maintaining data access reliability and access speed;
- By upscaling/downscaling the hardware specifications, cloud deployed session will be able to perform complex hardware demanding operations, such as long running weather simulations.
- The middleware cloud deployment will enable sessions to be used in real-time scenarios where the *QoS* level are a key factor;
- Session ubiquitous access will be improved since mobile clients will benefit from mobile optimized services and the *service reliability* characteristics previously described;

The next section will provide an overview of the solution architecture. The main modules will be enumerated and each modules' scope will be defined and explained in detail.

3.5 Architecture

Based on the requirements highlighted in section 3.1 the architecture of the proposed solution is composed of three main modules: the *datasource interface* that handles the heterogeneous data sources, the *middleware core* that handles session management operations and the *client interface* that handles clients' session connections. The middleware is deployed in a cloud platform and hence uses cloud-based services. This solution overview is illustrated in Figure 3.4.

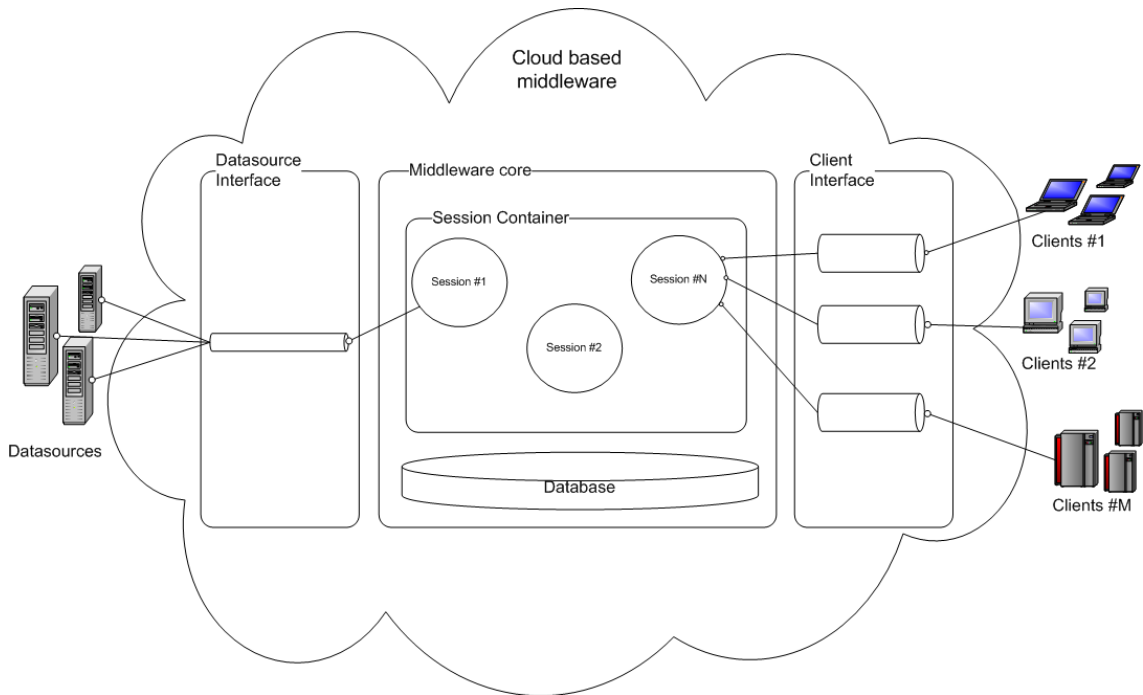


Figure 3.4: Solution architecture

3.5.1 Architecture Modules

The middleware architecture can be seen as a three tiered architecture where the *data source interface* represents the data tier, the *middleware core* represents the business logic tier and the *client interface* represents the presentation tier.

The **datasource interface** module is responsible for handling multiple protocols (e.g. *XMPP* or *RSS*) and forwarding the data events to the *middleware core*. This interface performs preliminary data source events' processing related to the data source definition (e.g. applying a regular expression to filter the generated data). This filtering is perceived by all clients in each session that receive input from this data source, thus creating an additional data source validation layer.

This interface decouples protocol specific operations from the underlying session operations;

The **middleware core** encapsulates all session specific operations such as dynamic reconfigurations, interaction model modifications or event persistence. The module consumes and processes the events generated by the *data source interface*, and forwards these events to the *client interface*. To perform these operations the module follows a message oriented approach composed of three layers: the *data source messaging layer*, the *session messaging layer* and the *client messaging layer*:

- **Data source messaging layer:** handles all data source data events and send these events to the session messaging layer. These events represent data sources' generated data such as wireless sensors' sensing data (e.g. humidity, temperature). Each of these data source may use a specific protocol as described in the *datasource interface*;
- **Session messaging layer:** handles the data sources' generated events according to each session's context. This includes processing events according to the session's aggregation function or modifying the interaction model according to a pre-defined dynamic reconfigurations. These dynamic reconfigurations impact all clients that take part in the session. Afterwards, the processed events are forwarded to the *client messaging layer*;
- **Client messaging layer:** performs any client related operations, such as client side dynamic reconfigurations and delivers the events to the appropriate clients through the client interface. These dynamic reconfigurations are only related the specific client that created the reconfiguration, no other session client is impacted;

The **client interface** module is responsible for delivering the session processed events to heterogeneous clients such as standalone, web or mobile clients. The events to be disseminated can include data sources' generated events and session context events such as dynamic reconfiguration notifications or new client connection notifications. The client interface is flexible enough to allow for mobile clients to interact with the middleware and take part in sessions. The client interface should provides a web based interface that allows administrative tasks such as data source definition. This web interface allows clients to set up sessions and control their lifecycle;

The middleware architecture can be seen as a three tiered architecture where the *data source interface* represents the data tier, the *middleware core* represents the business logic tier and the *client interface* represents the presentation tier. Figure 3.5 represents the three tiered organization described before.

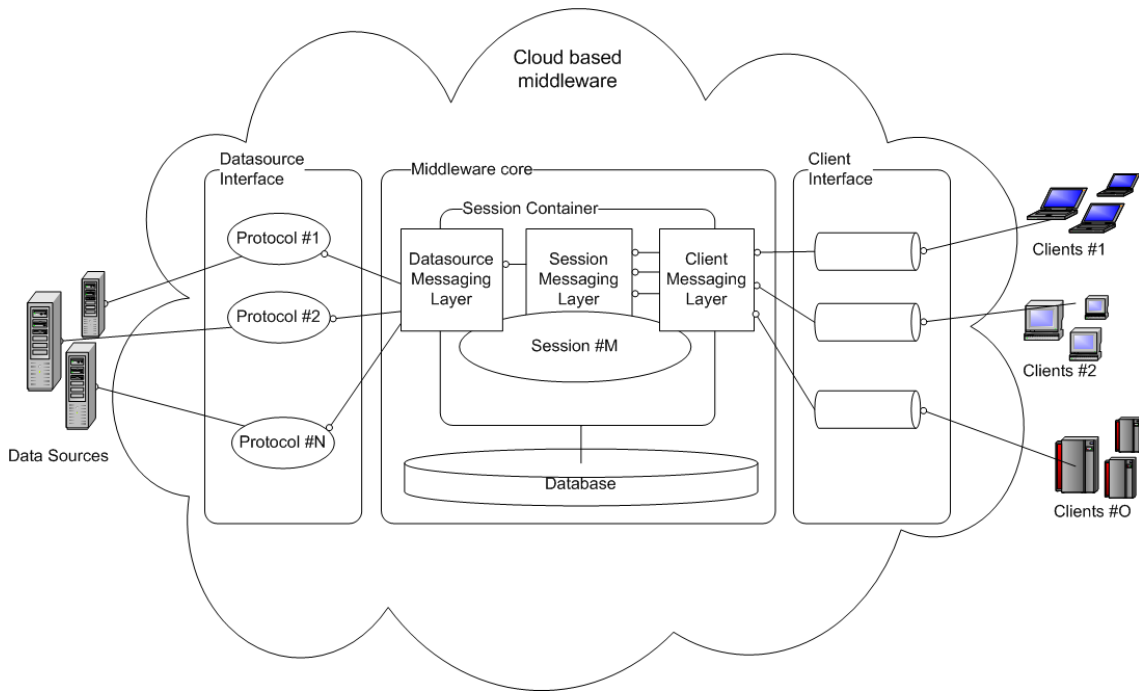


Figure 3.5: Solution architecture modules

3.5.2 Architecture Extensibility

One possible extension to this architecture might be to allow clients to input data to the session context, acting like specialized data sources. This extension would allow clients to create session conditions related to other clients characteristics or actions, thus widening the scope of possible applications that could take advantage of this solution. One example could be to develop a framework for collective stock trading data access by clients. In this scenario all clients would access stock quotations data sources and send their buy/sell order to the session. All other clients could then create some dynamic reconfiguration rules that would be triggered by these events. For instance, user B could define a dynamic reconfiguration rule describing that if user A buys stocks of type X, then user B subsequently starts receiving live information on values of type X.

The architecture organization and module's description included in this chapter are intended to be a high level view of the solution developed in this thesis. The specific development choices and technologies used for each of the architecture components are described in more detail in the next chapter.

4

Implementation

The proposed solution for the cloud-based session abstraction is composed of three main modules, as discussed on Chapter 3: the *data source interface*, the *middleware core* and the *client interface*. In this three tiered architecture the data source interface represents the data access, the middleware core controls the business logic and the client interface represents the presentation layer. Figure 4.1 provides an precise overview of the solution implementation.

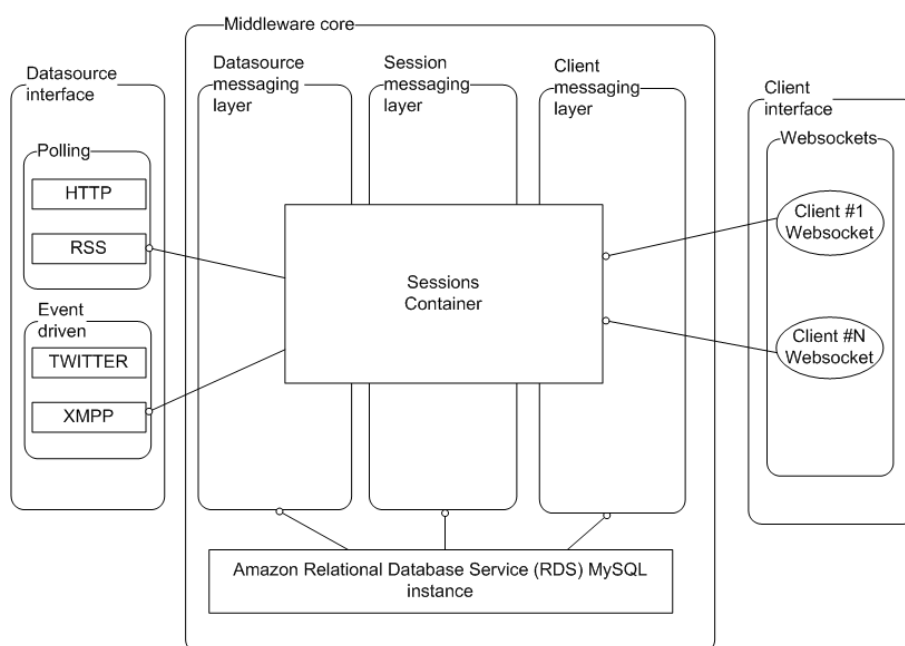


Figure 4.1: Implementation overview

4.1 Inter-Module Communication

Data events are generated by the data sources and pass through each of these modules until they are delivered to one or more clients. In order to move these messages between layers some routing system is necessary. Enterprise integration tools like *Apache Camel* [Cam] and *Spring Integration* [Spr] allow the definition of routing systems that can be used for this effect. Integration tools can also be used to satisfy the requirement on heterogeneous data sources, since they provide support for end points that use different protocols. Additionally, these tools have built in support for some pattern based interaction models as described in Section 2.4, which is useful for the session interaction models' definition.

On the possibility of choosing between *Apache Camel* and *Spring Integration*, the choice fell on *Apache Camel* mainly because it supports more useful technologies for the data source interface (e.g. the *WebSocket* endpoint is not supported by *Spring Integration*), and its expression language is more intuitive. This expression language is used for defining the routes that inter-connect each endpoint node.

The *Apache Camel* tool was used to implement all necessary routes from the data source interface to the client interface. These routes are used in the middleware core for each of the messaging layers, namely, the data source, session, and client messaging layers. Dedicated routes were implemented for each layer to layer communication, and each session possesses one group of such routes associated with it. Apart from the characteristics that define a session, which were described in Section 2.5, each session is associated with the following *Camel* routes:

- **Data source routes:** one route for each data source is linked to the session. These routes are defined in the *data source messaging layer* (Figure 4.1) and act as the bridge between the data source interface and the session messaging layer. Each of these routes applies data source's specific logic, such as regular expressions (i.e. client these regular expression serve as a pre-validation filter that is applied on all data sources' generated values), and delivers the processed messages to the session messaging layer;
- **Session routes:** one internal session route is linked to each session. These routes are defined in the *session messaging layer* and act as the bridge between the *data source messaging layer* and the *client messaging layer*. All session specific operations such as the session's dynamic reconfigurations and the definition of the session's interaction models are supported by these routes.

These operations are defined using *Camel* processors that are executed on each message;

- **Client routes:** each client is linked to a session through a route. These are defined in the *client messaging layer* and act as a bridge between the *session messaging layer* and the *client interface*. Each of these routes supports client specific operations such as client level dynamic reconfigurations or client interaction model modifications. These operations are defined using custom *Camel* processors that are executed on each message;

To allow for rich aggregation function definition while maintaining a message oriented approach that would be coherent with the *Apache Camel* philosophy, a complex event processing tool was used: *Esper* [Esp]. *Esper* provides special message queues upon which it is possible to apply complex expressions. These expressions can aggregate different types of data sources or apply other operations such as calculating averages or sums in specified time slots. *Apache Camel* supports *Esper* endpoints so this integration was performed by using the *Camel* domain language. Session or client level aggregation operation are performed when the messages are consumed from the *Esper* queues. Figure 4.2 illustrates the detailed implementation with *Apache Camel* routes and *Esper* queues.

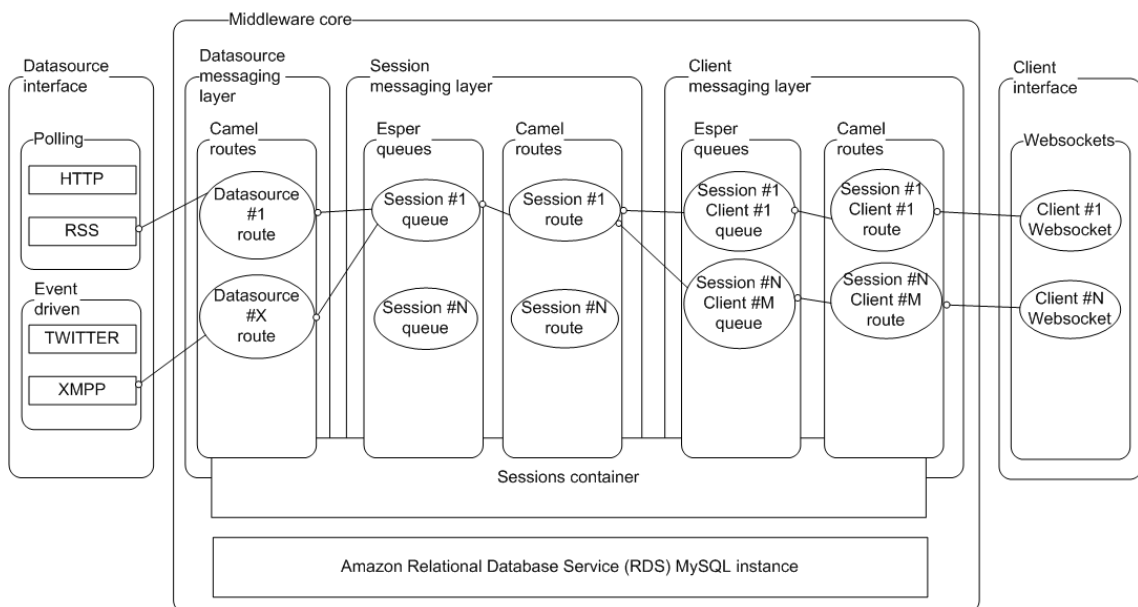


Figure 4.2: Camel and Esper integration

There are two types of *Esper* queues defined in this architecture:

- **Session queues:** there is one *Esper session queue* for each session. These queues are the entry point for messages in the *session messaging layer*. The previously described *Camel data source routes* will add messages to these queues coming from the *data source interface*.

The previously described *Camel session routes* will consume messages from these queues and deliver them to the correspondent *Esper client queues*. Session level aggregation operations are performed by applying *Esper* expressions when consuming events from these queues;

- **Client queues:** there is one *Esper client queue* for each client connected to each session. The previously described *Camel client routes* will consume messages from these queues and deliver them to the clients through the client interface.

Client level aggregation operations are performed by applying *Esper* expressions when consuming events from these queues;

4.1.1 Route Specification

Apache Camel philosophy is based on the concept of routes that connect distinct endpoints. Several routes can be combined to form complex networks where each node is an endpoint. A route complexity can range from a simple pass through style route to complex routing that involve patterns such as *Publisher-Subscriber*, dynamic endpoint definitions or custom *Camel processor classes* that are applied on each message.

These routes can either be defined in static file descriptors or entirely in the *Java* source code. The second approach was chosen to allow for runtime flexibility and route definition.

The base class used by *Apache Camel* to define these routes is `RouteBuilder`. This class allows to create one or more routes, using the *Camel* expression language to define the endpoints and more complex routing options.

The `RouteBuilder` class also allows to control the route lifecycle which is described in Figure 4.3.

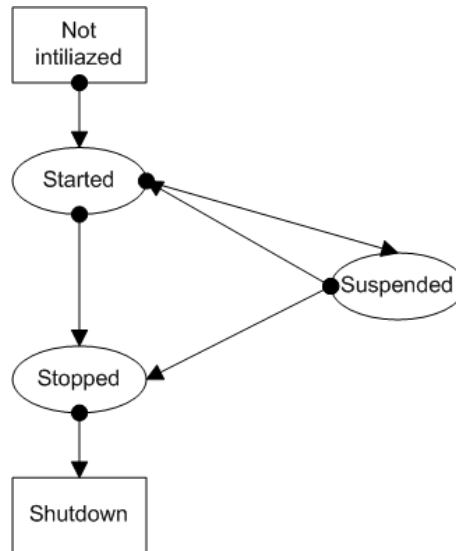


Figure 4.3: Camel route lifecycle

The base class used by the middleware to integrate *Apache Camel* base functionalities is `GenericRouteBuilder`, that extends the `RouteBuilder` class included in the *Apache Camel* libraries. The constructor forces implementations to associate a session id with the route, which will later be used to indicate which routing endpoint should be used.

The abstract class described before defined routes with the layout illustrated in Figure 4.4.

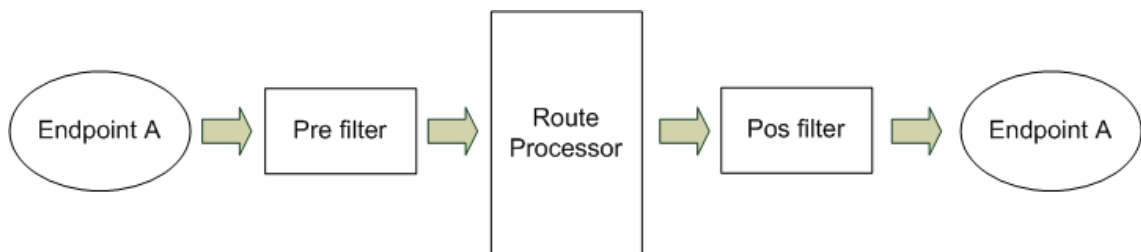


Figure 4.4: Generic route builder

The main `GenericRouteBuilder` method signatures are displayed in Listing 4.1.

Listing 4.1: Generic route builder

```
1 public abstract class GenericRouteBuilder
2     extends RouteBuilder {
3     ...
4     /**
5      * The constructor receives a session id
6      */
7     public EsperEventProducer(... Long sessionId ...) { ... }
8
9     /**
10    * Define a filter to be applied before the Camel processor
11    */
12    protected Class<?> getPreFilterClass() { ... }
13
14    /**
15    * Define a filter to be applied after the Camel processor
16    */
17    protected Class<?> getPosFilterClass() { ... }
18
19    /**
20    * Define the Camel processor class to be applied
21    */
22    protected abstract Processor getProcessor();
23
24    /**
25    * Start the route
26    */
27    public final boolean start() { ... }
28
29    /**
30    * Stop the route
31    */
32    public final boolean stop() { ... }
33
34    /**
35    * Suspend the route
36    */
37    public final boolean suspend() { ... }
38    ...
39 }
```


From the previously described *Apache Camel* routes and *Esper* queues two high level routes were defined to implement this integration: *Esper* event producers and *Esper* event consumers. The description for each type follows.

`EsperEventProducer` routes send events to a specified *Esper* queue. These routes are used by the *data source interface* to send data to *session queues* through the *data source messaging layer*. `EsperEventProducer` is an abstract class that forces implementations to associate a data source with the route. This data source will later be used as the input endpoint.

Listing 4.2 depicts the abstract methods that `EsperEventProducer` adds to its superclass.

Listing 4.2: Esper event producer

```
1 public abstract class EsperEventProducer
2     extends GenericRouteBuilder {
3     ...
4     /**
5      * Retrieve the data source associated with this route
6      */
7     public abstract DSSDataSource getDataSource();
8     ...
9 }
```

`EsperEventConsumer` routes consumes events from a specified *Esper* queue. These routes are used by the *session messaging layer* to read data from the *session queues* and send the data to the respective *client queues*. This is actually a special route since it consumes from *Esper* queues and sends messages to *Esper* queues. `EsperEventConsumer` routes are also used by the *client messaging layer* to read messages from each client's *Esper* queue and send the data to the actual clients though the *Client interface*. `EsperEventConsumer` is an abstract class that forces implementations to provide the *Esper* queue name from where messages should be consumed. Implementing classes must also provide the *Esper* expression that should be used to consume events.

Listing 4.3 depicts the abstract methods that `EsperEventConsumer` adds to its superclass.

Listing 4.3: Esper event consumer

```

1 public abstract class EsperEventConsumer
2     extends GenericRouteBuilder {
3     ...
4     /**
5      * Specify the Esper queue name from which this route
6      * will consume events
7      */
8     protected abstract String getEsperName();
9
10    /**
11     * Specify the Esper expression used to consume events
12     */
13    protected abstract String getEsperExpressionQuery();
14    ...
15 }

```

The EsperEventProducer and EsperEventConsumer class organization described in this section is illustrated in the Figure 4.5.



Figure 4.5: Camel Esper base class diagram

4.2 Data Source Interface

To incorporate heterogeneous data sources a thin connector layer was built on top of *Apache Camel*. The `datasource` interface is the data events' entry point to the middleware. A set of custom generic classes that facilitate the creation of *Apache Camel* based endpoints.

The base connector class is `ComponentHelper` which is an abstract class. All implementations are forced to implement three methods: `buildComponent()` to perform any connector specific initializations, `getFromUri()` to provide the *Camel* routing expression, should this connector act as a input endpoint, and `getToUri()` to provide the *Camel* expression, should this connector act as a output endpoint.

Connector classes abstract from the *Camel* specific syntax and provide a more intuitive object oriented data source creation process and act as connectors for protocols supported by *Camel* and that will be used to incorporate heterogeneous data sources. To showcase this functionality four connectors were developed: *HTTP*, *RSS*, *Twitter* and *XMPP*. These data source connectors are organised in two main groups according to the nature of the data event generation process:

- **Polling data sources** refer to data sources that need to be interrogated with a certain periodicity. From the implemented connectors, both *HTTP* and *RSS* fall into this category. This implies that one of the characteristics necessary to define the data source is a polling period;
- **Event driven data sources** refer to data sources that send data to the middleware without the need to be interrogated. This fits into the push notification definition and implies that the middleware must be aware that events can be pushed at any given moment. From the implemented connectors, both *Twitter* and *XMPP* fall into this category;

These connectors serve mainly as protocol specific interfaces, the actual separation between *Polling data sources* and *Event driven data sources* in the source code is implemented in *Camel* routes and in the data model. This organization will be described in more detail in Section 4.3.1.

Figure 4.6 describes the connector organization.

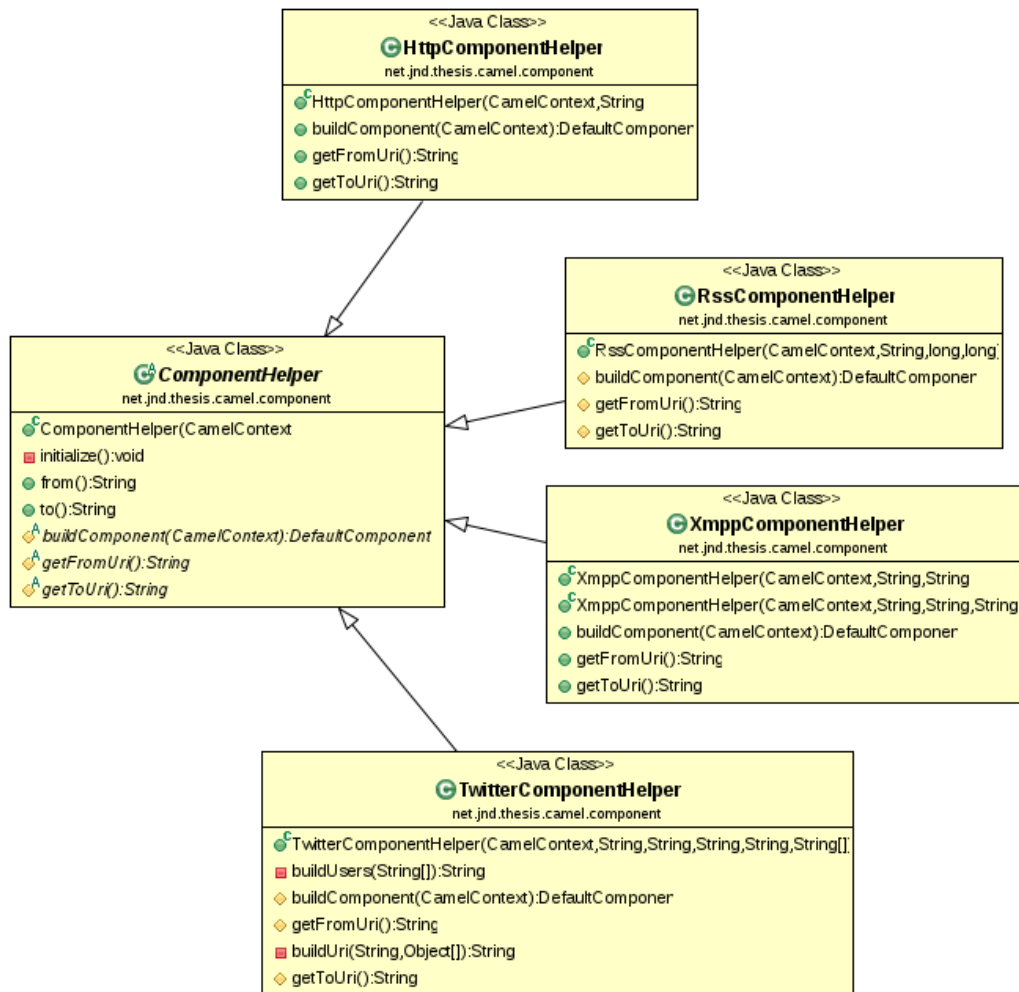


Figure 4.6: Camel endpoint connectors

The implemented connectors are the following:

- **HTTP connector:** for this connector the main attribute to be defined is the URL that should be interrogated. The “from URI” which defines the *Camel* expression that should be used to use HTTP as an input endpoint is very simple and takes the form of “*http://url.of.some.site.com*” in *Camel* notation;
- **RSS connector:** for this connector the main attribute to be defined is the URL that should be interrogated. The “from URI” which defines the *Camel* expression that should be used to use RSS as an input endpoint takes the form of “*rss://url.of.some.site.com?splitEntries=true*” in *Camel* notation. The *splitEntries* parameter indicates that the rss feed should be consumed one by one instead of being consumed in batches;

- **Twitter connector:** for this connector the main attributes to be defined are the consumer key and access tokens that indicate the twitter developer account that should be used to connect to the Twitter services and the user ids that should be followed. The “from URI” which defines the *Camel* expression that should be used to use Twitter as an input endpoint takes the form of “*twitter://streaming/filter?type=event &userIds=userA,userB &consumerKey=A &consumerSecret=B &accessToken=C &accessTokenSecret=D*”;
- **XMPP connector:** for this connector the main attributes to be defined are the user account information, username and password. The “from URI” which defines the *Camel* expression that should be used to use XMPP as an input endpoint takes the form of “*xmpp://talk.google.com:5222/?serviceName=gmail.com &user=A &password=B*” in *Camel* notation;

The connection between the *client interface* connectors and the *middleware core* that allows events produced by the data sources to be consumed by the middleware sessions is performed by a specific set of *Camel* routes from the *data source messaging layer* present in the *middleware core*. This messaging mechanism will be described in the next chapter.

4.3 Middleware Core

Internally the middleware uses a message based routing system composed of three layers: the *data source messaging layer*, the *session messaging layer* and the *client messaging layer*. These messaging layers consist mainly of specific sets of *Esper* queues, *Camel* routes and *Camel* processors.

The *middleware core* also contains a *sessions container* which is transversal to the messaging layers since each session is associated with routes from the three layers.

The database support necessary to all middleware components is provided by a *MySQL* database instance deployed in the cloud. Each of these elements will be described in detail in the next chapters.

Figure 4.7 displays the middleware's core organization.

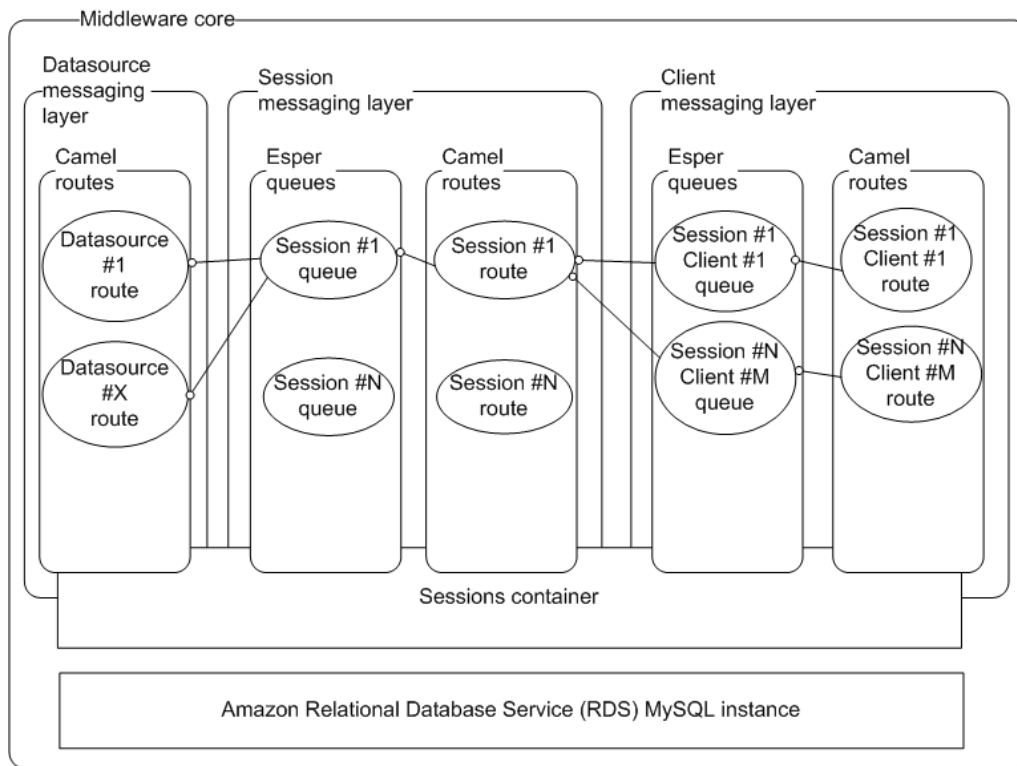


Figure 4.7: Middleware core

4.3.1 Data Source Messaging Layer

The *data source messaging layer* provides a set of *Camel* routes that forward the messages from the data sources connectors defined in the *data source interface* to a specific *Esper* session queue present in the *session messaging layer*. As described before in Section 4.1, the base class for defining routes that send events to an *Esper* queue is the `EsperEventProducer` class.

Since these routes will send data source events to a session *Esper* queue, the concrete implementations will extend the `EsperEventProducer` class and implement/override the necessary methods so that the functionality for each connector type is available.

The `EsperEventProducer` class organization is displayed in Figure 4.8.

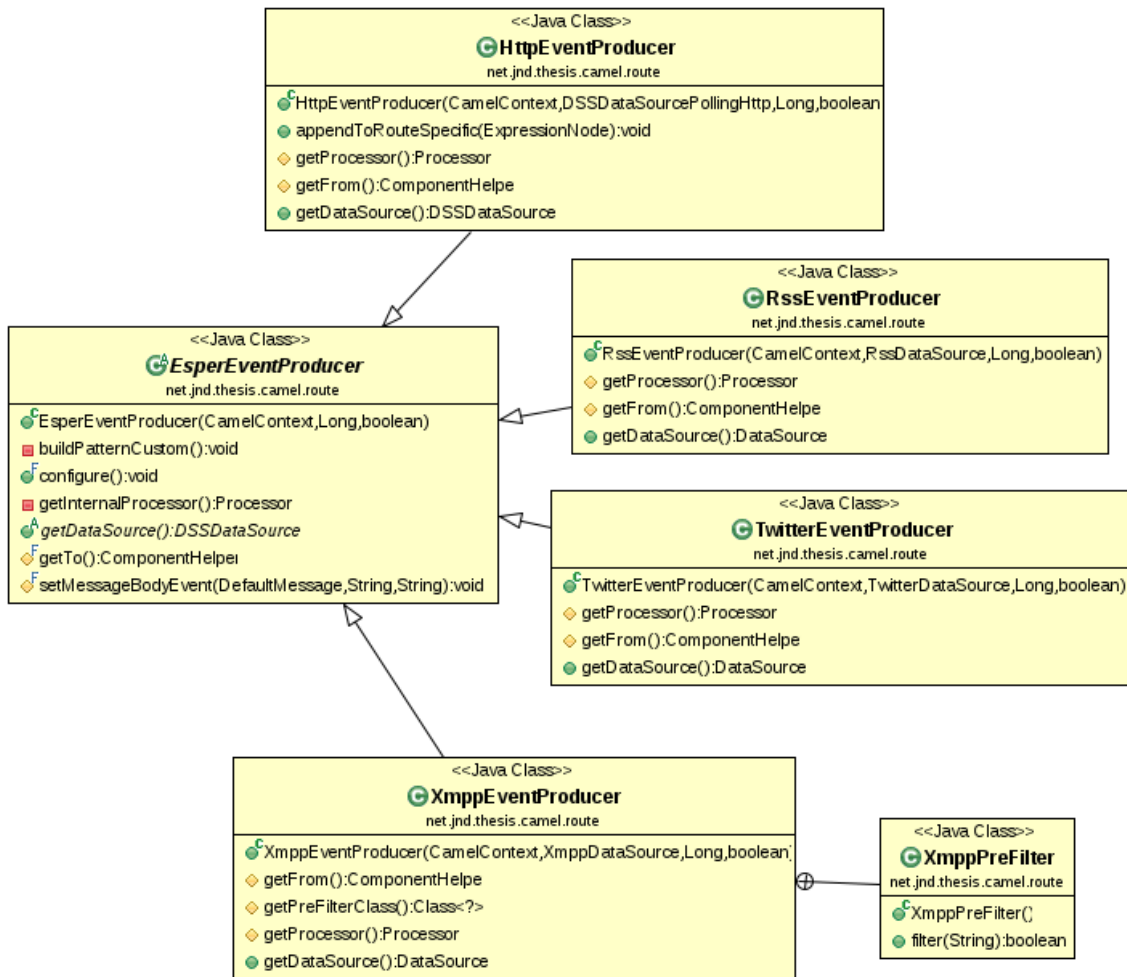


Figure 4.8: Camel data source Esper producer routes

`EsperEventProducer` classes are directly linked with a data model entity which contains the attributes necessary to build the route. The constructor for these routes receives one of these entities as a parameter as illustrated in Figure 4.8.

The `EsperEventProducer` routes use data source connectors to generate the necessary *Camel* routing expressions, used to obtain events from *Camel* endpoints. Internally *Camel* may generate an arbitrary event class according to the type of endpoint. In order to transparently integrate all endpoints and re-use the same processes, the same event class must be generated, independently of the connector type. To accomplish this, the events produced by *Camel* are normalized to a middleware specific format, specified by an internal class, that is used to broadcast events between layers.

The internal event class `CamelInternalEvent` is illustrated in Figure 4.9.

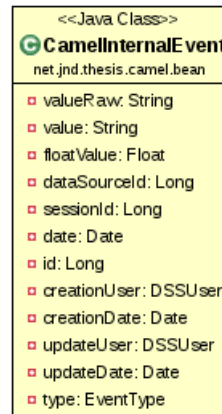


Figure 4.9: Camel internal event class

When a client creates a data source, a correspondent domain entity is generated and persisted in the database. The data model reflects the data source types described before, and the separation between polling data sources and event driven data sources. Apart from the class hierarchy separation the main characteristic that separates these two data source types is that the `PollingDataSource` entity defines a polling interval. This value will be used in the routes to define the periodicity which will be used to interrogate the adjacent polling data source. The class diagram in Figure 4.10 reflects the `EsperEventProducer` base hierarchy.

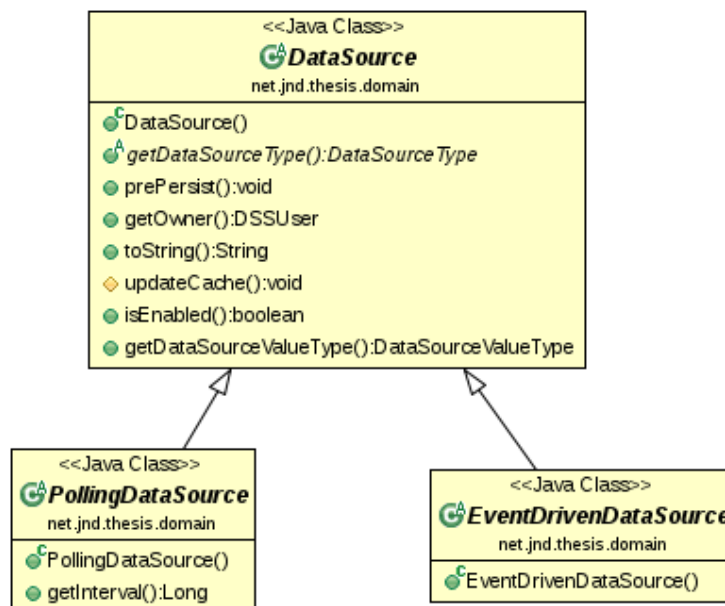


Figure 4.10: Data source data model

The class diagram in Figure 4.11 reflects the `PollingDataSource` hierarchy.

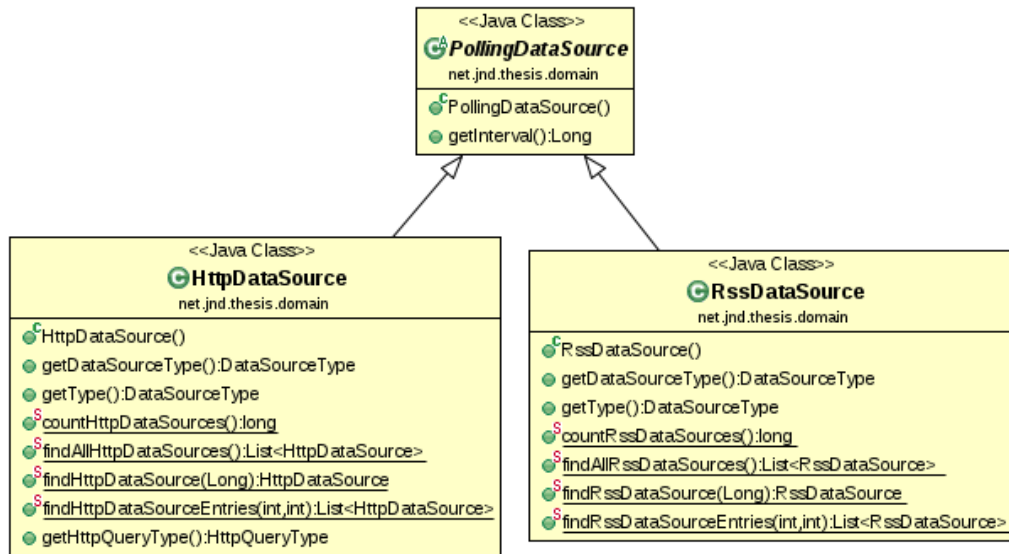


Figure 4.11: Polling data source data model

The class diagram in Figure 4.12 reflects the `EventDrivenDataSource` hierarchy.

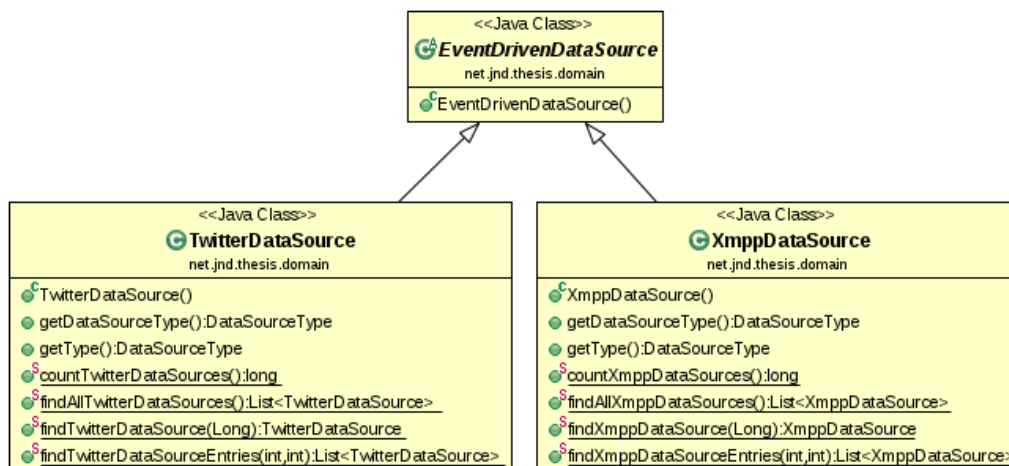


Figure 4.12: Event driven data source data model

The constructor for each data source route also receives a unique session id and this id will be used to send the generated events to the appropriate session *Esper* queue in the *session messaging layer*. If the *Esper* queue was not yet created the queue is created automatically transparently by *Camel*. This process will be described in more detail in the next chapter.

4.3.2 Session Messaging Layer

The *session messaging layer* consumes messages from each session *Esper* queue and forwards them to the *client messaging layer*, more specifically, to every client *Esper* queue that is linked with this session. Operations such as dynamic recon-figurations at session level are handled in this layer by using specific *Camel* processors that operate on the data source events. The session aggregation function is also handled in this layer by applying a specific *Esper* expression in the session *Esper* queue.

The class that implements these routes is `EsperEventConsumerInternal`, and since these routes consume events from *Esper* queues, the class extends the `EsperEventConsumer` class. The constructor for each of these routes receives a session id that will be used to fetch the appropriate `Session` object from the data model whenever it is necessary to inspect it's attributes. A session has one interaction model, a set of dynamic reconfigurations, a set of allowed users and a session status among other attributes. The class diagram in Figure 4.13 represents the session route hierarchy.

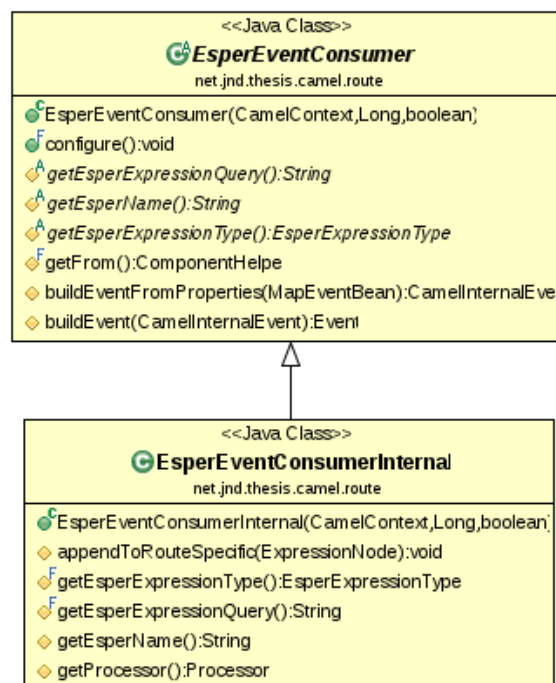


Figure 4.13: Camel session Esper consumer routes

The `Session` object contains one `SessionStatus` enum that represents the current status, one `EsperExpression` object that represents the aggregation function, one `InteractionModel` object that represents the current pattern, a list of `DynamicReconfiguration` objects and a list of `AuthUser` objects that represents the clients that are allowed to connect to the session. The `Session` object is illustrated in Figure 4.14.

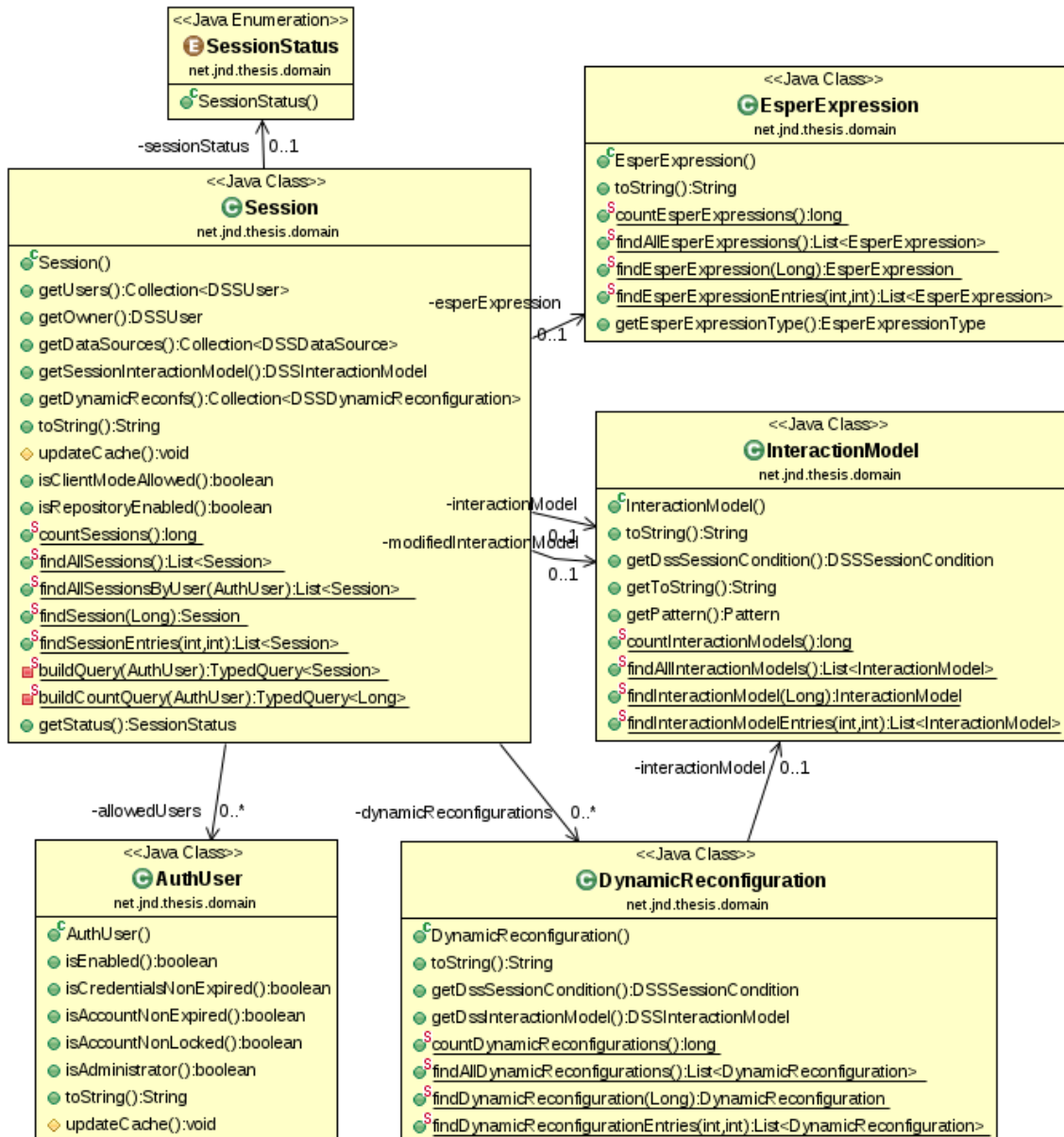


Figure 4.14: Session domain entity

There are three key attributes in the `Session` object that describe the interaction model. The description for each attribute follows.

The `EsperExpression` entity allows the definition of an aggregation function using *Esper* notation. Each expression contains one `EsperExpressionType` enum which indicate if this is a *Pattern* or *EQL* expression (*Esper* specific expression types). The default is a simple pass through *Pattern* expression: “select * from pattern [every e=Event]”;

The `InteractionModel` entity allows to describe which pattern will be applied to the session by default. Each `InteractionModel` contains a single `PatternTemplate` enum that identifies the pattern in use. There are three available patterns: *Streaming*, *Publisher-Subscriber* and *Producer-Consumer*;

The `DynamicReconfiguration` entity list allows clients to specify which reconfigurations will be included in the session context. These reconfigurations are defined three objects: a `SessionCondition`, an `InteractionModel` and an `EsperExpression`. Each `SessionCondition` contains a value, a single `DataSource` object and a `ConditionOperator` enum that indicates which operator to use (e.g. greater / smaller than). When each reconfiguration is triggered, the associated `InteractionModel` or `EsperExpression` can be applied to the running session.

This data model mapping is illustrated by the class diagram in Figure 4.15.

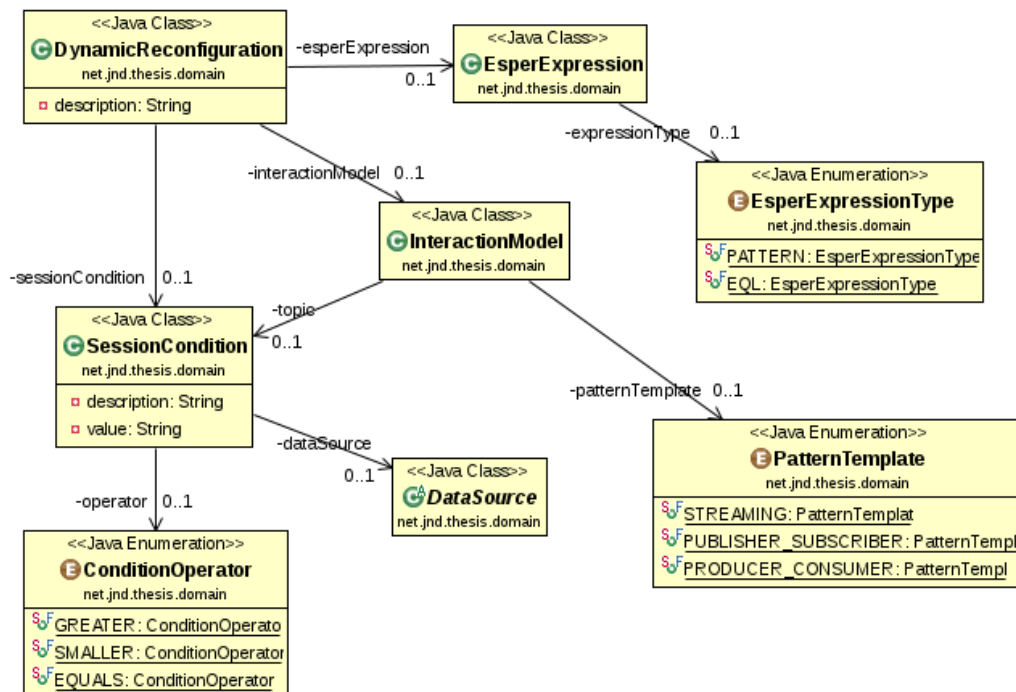


Figure 4.15: Dynamic Reconfiguration data model

Apart from the routing characteristics described before the core functionality is implemented in the *Camel* processor class returned by the `getProcessor()` method. This code organization is described in Listing 4.4.

Listing 4.4: Session esper consumer internal route

```
1 class EsperEventConsumerInternal extends EsperEventConsumer {
2     ...
3     /**
4      * Fetch the Camel processor that encapsules the required behavior
5      */
6     @Override
7     protected Processor getProcessor() {
8         return new Processor() {
9             @Override
10            public void process(Exchange exchange) {
11                ...
12            };
13        }
14        ...
15    }
```

This *Camel* processor is executed on each event that is consumed from the session *Esper* queue and the first operation it performs is to fetch the session domain entity that describes the session.

The pattern based interaction model is applied on the `process()` method of the *Camel* processor class. The most relevant operations performed for each pattern are: for *publisher-subscriber* all events are matched with the associated condition and will be discarded if the match fails, for the *producer-consumer* the associated polling interval will be set.

The event broadcasting to each client queue is handled using *Camel* expression language. This route class implements the `appendToRouteSpecific()` method inherited from the `EsperEventConsumer` class and adds a parallel processing that will disseminate the event to a client queue for each client that is allowed to connect to the session. This client list is included in the `Session` domain entity obtained by the processor as described before.

Listing 4.5 illustrates the event broadcasting process

Listing 4.5: Client esper queue broadcasting

```

1 class EsperEventConsumerInternal extends EsperEventConsumer {
2     ...
3     /**
4      * Specific route rules are added to use Camel broadcast
5      */
6     @Override
7     protected void appendToRouteSpecific(ExpressionNode route) {
8         // Camel built-in recipient list pattern support
9         route.recipientList(header(
10
11         // Recipient list is fetched from the message header
12         EsperEventConsumer.HEADER_RECIPIENT_LIST),
13         EsperEventConsumer.DELIMITER_RECIPIENT_LIST)
14
15         // Camel built-in parallel processing improves performance
16         .parallelProcessing().ignoreInvalidEndpoints();
17     }
18     ...
19 }

```

The `EsperEventConsumerInternal` processor first retrieves the latest session `InteractionModel`, then applies all necessary pattern related operations according to this `InteractionModel`. Listing 4.6 demonstrates the interaction model retrieval process.

Listing 4.6: Session interaction retrieval

```

1 class EsperEventConsumerInternal extends EsperEventConsumer {
2     ...
3     protected Processor getProcessor() {
4         return new Processor() {
5             ...
6             // Retrieve session interaction model
7             InteractionModel interactionModel = session
8                 .getInteractionModel();
9             Long oldInteractionModel = interactionModel.getId();
10            Collection<DynamicReconfiguration> reconfs = session
11                .getDynamicReconfigurations();
12            ...
13        };
14    }
15    ...
16 }

```

Listing 4.7 demonstrates the pattern operations.

Listing 4.7: Session pattern operations

```

1 class EsperEventConsumerInternal extends EsperEventConsumer {
2     ...
3     protected Processor getProcessor() {
4         return new Processor() {
5             ...
6             // Process pattern related operations
7             switch (interactionModel.getPattern()) {
8                 case PUBLISHER_SUBSCRIBER:
9                     DSSSessionCondition condition = interactionModel
10                        .getDssSessionCondition();
11                     eventOk = EsperEventConsumerHelper.checkCondition(
12                        condition, event);
13                     // if not ok, clean recipient list
14                     if (!eventOk) {
15                         msg.setHeader(HEADER_RECIPIENT_LIST,
16                            org.apache.commons.lang3.StringUtils.EMPTY);
17                     }
18                     break;
19                 case PRODUCER_CONSUMER:
20                     msg.setHeader(HEADER_DELAY, interactionModel.getEventDelay());
21                     break;
22                 ...
23             };
24         }
25     }
26 }

```

The dynamic reconfigurations are applied in the `process()` method of the *Camel* processor class. For each event received all existing dynamic reconfiguration conditions are checked and if any condition is verified that modified the current interaction model, then the dynamic reconfiguration is applied. First, all the dynamic `DynamicReconfiguration` objects related to this `Session` are fetched. Then these reconfiguration objects are iterated and for each one, the associated `SessionCondition` is matched with the current event. If the condition is verified then the `InteractionModel` associated with the `Session` is replaced with the new one.

The most relevant code that performs these dynamic reconfiguration operations can be in Listing 4.8.

Listing 4.8: Session dynamic reconfiguration processing

```

1 class EsperEventConsumerInternal extends EsperEventConsumer {
2     ...
3     protected Processor getProcessor() {
4         return new Processor() {
5             ...
6             // Iterate all session DynamicReconfiguration objects
7             for (DSSDynamicReconfiguration rec : reconfs) {
8
9                 // Fetch the DynamicReconfiguration related condition
10                DSSSessionCondition condition = rec.getDssSessionCondition();
11
12                // If the condition is verified apply new interaction model
13                if (EsperEventConsumerHelper.checkCondition(condition,
14                    event)) {
15                    Long newModel = rec.getDssInteractionModel().getId();
16
17                    // only apply if this is a different interaction model
18                    if (!oldInteractionModel.equals(newInteractionModel)) {
19                        updateSessionInteractionModel(session, newModel);
20                        ...
21                    }
22                }
23            }
24            ...
25 }

```

The cloud repository data storage is also implemented on the `process()` method of the *Camel* processor class. The object can be persisted using standard *Hibernate* / *JPA*.

For each processed event the custom *Camel* processor checks if the repository flag is enabled for this session, and if this is the case, the `Event` object is persisted in the cloud-based data base instance.

The most relevant code that performs these repository persistence operations can be found in Listing 4.9.

Listing 4.9: Session event persistence

```

1 class EsperEventConsumerInternal extends EsperEventConsumer {
2     ...
3     @Override
4     protected Processor getProcessor() {
5         return new Processor() {
6             ...
7
8             // If the session repository is enables, store the event
9             // in the cloud-based repository
10            if (... && session.isRepositoryEnabled()) {
11                Event evt = buildEvent(event);
12                if(evt != null) {
13                    ...
14
15                    // Persist the object using Hibernate/JPA
16                    evt.persist();
17                    ...
18                }
19            }
20        };
21    }
22    ...
23 }

```

The *session Esper consumer* routes described in this chapter are simultaneously *Esper event producers* since they broadcast events to *client Esper queues* that are present in the *client messaging layer* that will be described in detail on the next chapter.

4.3.3 Client Messaging Layer

This layer consumes events from each client *Esper* queue and forwards these messages to the appropriate client through the Client interface. Client level dynamic reconfigurations are handled in this layer by applying an *Esper* expression to the client *Esper* queue.

Unlike session level reconfigurations, client level reconfigurations are only applied to this specific client. Client dynamic reconfigurations are also handled in this layer.

The class diagram in Figure 4.16 represents the client routes hierarchy.

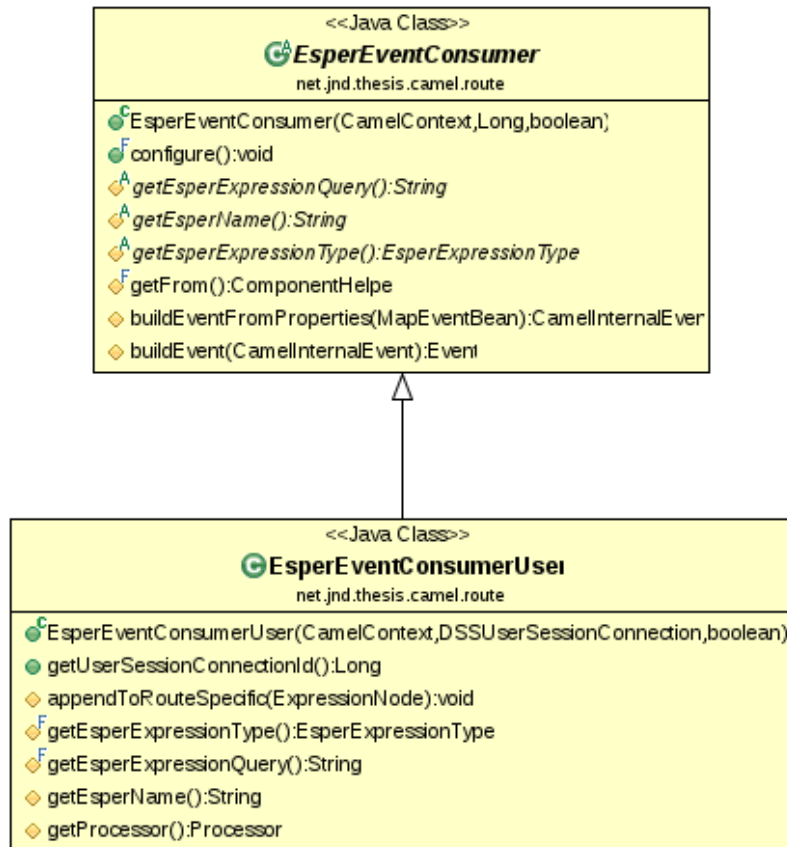


Figure 4.16: Client routes

These routes are closely linked with the `UserSessionConnection` domain entity. The `UserSessionConnection` describes to which session should the client connect to, and how this connections should be performed. Apart from the session id, there are three main attributes used to describe a connection: an `EsperExpression`, a `InteractionModel` and a `DynamicReconfiguration` list. This organization is similar to what was described in the Section 4.3.2 for the `Session` object.

When a client connects to the middleware through the client interface, one single `UserSessionConnection` object reference is passed and this reference is used to create one `EsperEventConsumerUser` route. This route will then consume events from the appropriate client *Esper* queue according to the user defined `EsperExpression`, `InteractionModel` and also using all reconfigurations defined in the `DynamicReconfiguration` list.

The constructor for each of these routes receives a `UserSessionConnection` that contains all the necessary attributes to manage client routes and perform client level operations. Similarly to the session routes, client level core functionality is implemented by a *Camel* processor that is consumed from the client *Esper* queue. This processor behavior is defined by the anonymous inner class returned by the method `getProcessor()`.

Event broadcasting to the clients is performed by calling the `broadcast()` method from helper class `UserSessionConnectionHelper`. This class is a middleware entry point to the client interface that manages the client connections. In the source code, the interaction model specific event processing and the dynamic reconfigurations are implemented using the same approach used in the session routes *Camel* processors. These operations are described in detail in Section 4.3.2. The `UserSessionConnection` object relevant data model is described in Figure 4.17.

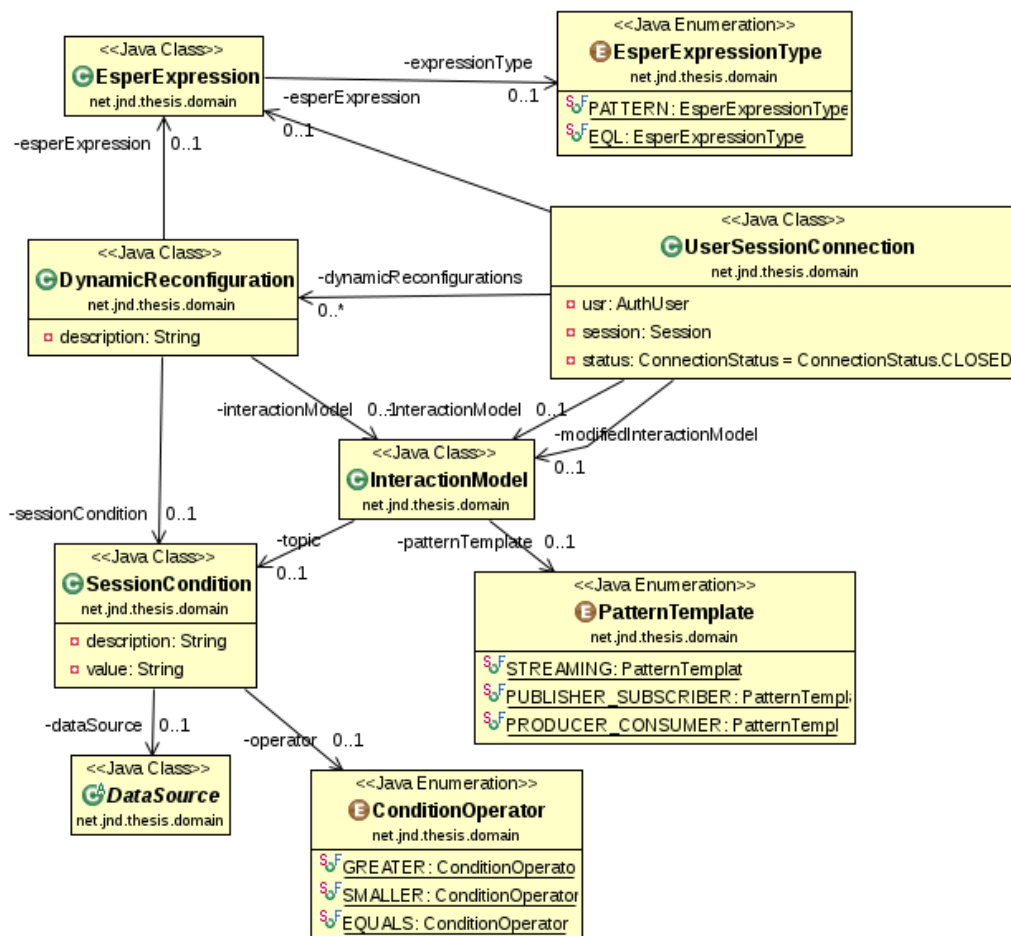


Figure 4.17: `UserSessionConnection` data model

The *client Esper consumer* routes described in this chapter are consume events from the *client Esper queues* associated with each session. As described before there is a specific *Camel* processor that applies client level operations. This same processor is responsible for sending the events to the actual clients, through the client interface, by invoking the `broadcast()` method from the helper class `UserSessionConnectionHelper`. The details of client connection management will be described in Section 4.4

4.3.4 Session Container

The session container component is a group of classes that manage the session creation and life cycle. The user performs session operations by manipulating the data model associated with the session, specifically the `Session` entity. When a `Session` entity object is modified, an event is triggered and if the route is active the modifications performed by the client will be propagated to the associated routes or client connections. These modifications can include adding/removing dynamic reconfigurations, adding/removing data sources, explicitly modifying the interaction model or modifying the allowed users set.

When a `Session` is created and the session is started all the routes associated with this session are also created and encapsulated in a `RoutesWrapper` object. The main attributes are the following:

- A map that stores the `EsperEventProducer` objects associated with the session. There is one of these objects for each data source linked to the session. These objects represent data source routes present in the *data source messaging layer*;
- One session `EsperEventConsumer` that references the single session route associated with the session. This route is present in the *session messaging layer*;
- A map that stores `EsperEventConsumer` objects associated with the session. There is one of these objects for each client currently connected to the session. These objects represent client routes present in the *client messaging layer*

Figure 4.18 represents the `RoutesWrapper` object main attributes and operations.

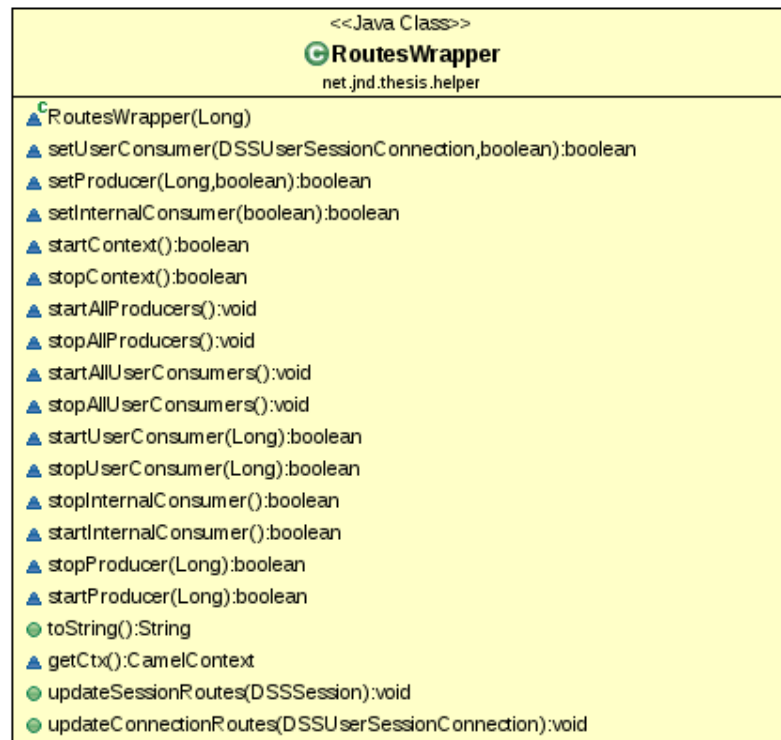


Figure 4.18: `RoutesWrapper` object

4.4 Client Interface

The *client interface* is responsible for the direct connections between clients and middleware. Clients connect to the middleware using the *WebSockets* [IET11] protocol. *Websockets* protocol allows client/server bi-directional communication using an client application that implements the specification or using a standard web-browser.

The middleware is listening on a specific port for *WebSockets* connections. This is performed using a specialized *Java Servlet* class that implements the *WebSockets* protocol. There are several server side implementations of the protocol but for this thesis the *Jetty* server implementation was used. The *Servlet* class created on server side is `ServerSideWebSocketServlet` and this class extends the *Servlet* class `WebSocketServlet` provided by *Jetty*. Extending classes must override the method `doWebSocketConnect()`. This method returns an object that implements the *WebSocket* interface, which is also provided by *Jetty*.

A new `WebSocket` object is created for each connection and this object is internally called whenever an operation is required. In this implementation the concrete `WebSocket` class used is `ServerSideWebSocket`. This class implements the interface `OnTextMessage` provided by *Jetty* which extends the interface `WebSocket`. The `ServerSideWebSocket` class hierarchy is illustrated in Figure 4.19.

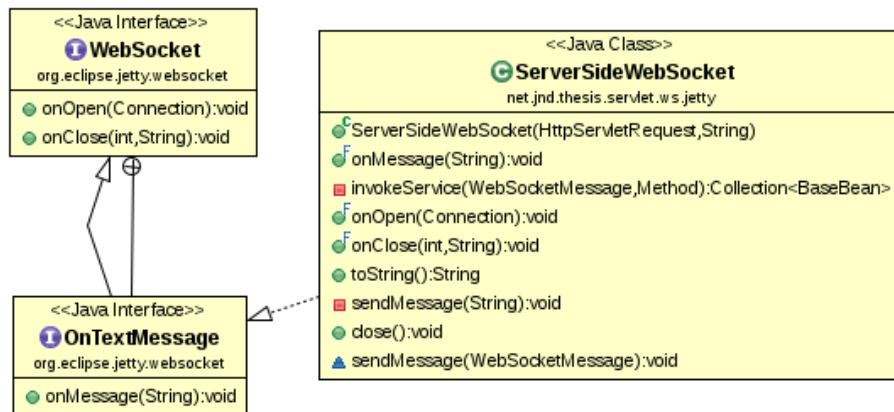


Figure 4.19: WebSocket class hierarchy

By implementing the `OnTextMessage` interface, the previously mentioned `ServerSideWebSocket` classes must define three methods:

- `onOpen(Connection)`: This method handles the initial handshake. Information about the client trying to establish a connection is included in the `Connection` object. The reference to this `Connection` object is maintained to send messages back to clients.
- `onMessage(String)`: This method handles all incoming messages. After the initial handshake is established, the actual communication is performed by exchanging custom *JSON* serialized messages. The required marshalling/unmarshalling mechanisms are implemented in the helper class `WebSocketMessageHelper` which is common to both server and client applications. To access any of the services a authentication request message must sent by the client and validated by the server. After the authentication is successful, the `WebSocket` is linked to a client and is ready to accept further service requests. This association is implemented in class `ServerSideWebSocketHelper` which maintains a map that links user ids with the matching `WebSocket`

Listing 4.10 illustrates the client - *WebSocket* mapping:

Listing 4.10: `ServerSideWebSocketHelper`

```

1 public class ServerSideWebSocketHelper {
2     ...
3     /**
4      * Map the links user ids with ServerSideWebSocket objects
5      */
6     private static Map<Long, ServerSideWebSocket> connections =
7         Collections.synchronizedMap(
8             new HashMap<Long, ServerSideWebSocket>());
9     ...
10 }

```

At this point the middleware is also able to spontaneously send messages to this client (e.g. broadcast some relevant middleware context modification). If the authentication is not validated, the association between the user and this `ServerSideWebSocket` is not stored, but the connection itself is not automatically closed to allow further authentication attempts.

- `onClose(int, String)`: This method handles all connection closing and resource cleanup operations. This closing operation can occur explicitly by invoking the `close()` method in the `Connection` object associated with this *WebSocket* or as a result of a connection timeout. The main resource cleanup operation is to remove any association between the user and this *WebSocket* in the previously referred `ServerSideWebSocketHelper` helper class.

Additionally, if the client is taking part in any session, then all related *client Camel routes* are closed (Section 4.4 describes the *client Camel routes*). All route cleanup operations are handled in class `RouteHelper` by calling the method `closeUserSessionConnection(Long)`.

All the exchanged message classes must extend from a base `WebSocketMessage` class. As described before, the `ServerSideWebSocketHelper` class is responsible for marshalling/unmarshalling all messages, and the methods in this class expect message that extend `WebSocketMessage`.

Figure 4.20 illustrates a the `WebSocketMessage` class:

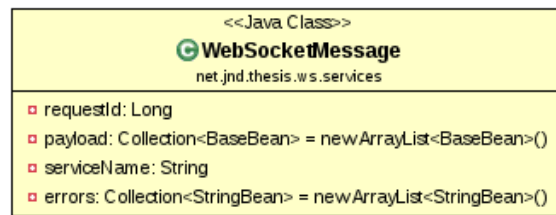


Figure 4.20: `WebSocketMessage`

There are four main attributes in each `WebSocketMessage`:

- *requestId*: This attribute is used whenever a synchronous request is performed. The associated response must include the same request id so that the caller can relate the message to a specific previously issued request.
- *payload*: This attribute can include either all objects passed in a request or all the objects returned by a service call.
- *serviceName*: This attribute described which service is being requested. The list of available services will be described in detail in Section 4.4.
- *errors*: This attribute contains all error message that may result of a service request.

The *payload* attribute contains the actual exchanged data in a `BaseBean` collection. All marshalling/unmarshalling operations expect that the payload beans extend this class. Figure 4.21 illustrates a partial `BaseBean` class hierarchy.

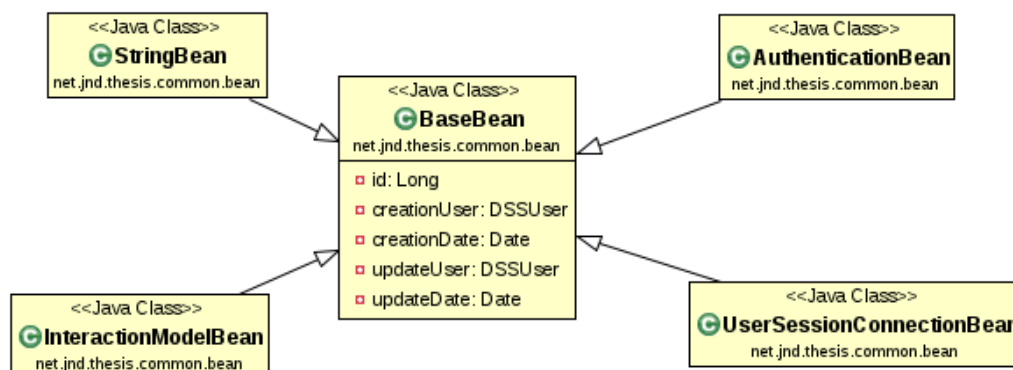


Figure 4.21: `BaseBean` partial hierarchy

As described in Section 4.3.3, when a session generates events that need to be delivered to specific clients, the method `broadcast (UserSessionConnection, Event)` from class `UserSessionConnectionHelper` is called. The necessary session id and user id are extracted from the `UserSessionConnection` object and then the previously described class `ServerSideWebSocketHelper`, more specifically the method `broadcastEvent (Long, Event)`.

The next section will enumerate the available services provided by the middleware, and describe the base client side services implementation on top of which new clients can be developed.

4.4.1 Services API

The *services API* is an interface that provides access to the all the middleware functionalities. Any *Java* based application that implements the API can be a middleware client and thus create or take part in existing sessions. There are four service groups defined:

- **Core services:** This group includes services that do not require client authentication. These services include the testing method `echo (String)` and the `authenticate (AuthenticationBean)` method. All the remaining services require a previous successful authentication;
- **Listing services:** This group includes all services that allow clients to fetch data model information that will be used to display and to serve as input to another services. Mostly these are methods that fetch lists of a certain domain entity or return a single entity with a specified id. These services only return the entities created by this client. Examples of such services are `listDataSources ()` or `getSession (Long)`. The `listSessions ()` is a particular service because it returns not only the client owned sessions but also the session to which the client can connect to. There is a specific `listOwnedSessions ()` that only returns the sessions created by the client;
- **Updating services:** This group includes all services that explicitly modify the data model, either by creating a new entity or by updating an existing entity. Similarly to the listing services, the updating operations are only available for entities created by this client. Examples of such services include `createEsperExpression (DSSesperExpression)` or the equivalent `update updateEsperExpression (DSSInteractionModel)`;

- **Session services:** This group includes all session related services. These services include the `connectToSession(DSSUserSessionConnection)` or the `replaySession(DSSReplaySession)` methods. There is a convenient service defined by method `startTestDataSourceSession()` that allows clients to quickly setup a session with a single data source and a specified interaction model by providing the ids for each one;

The class diagram in figure 4.22 illustrates the services hierarchy.

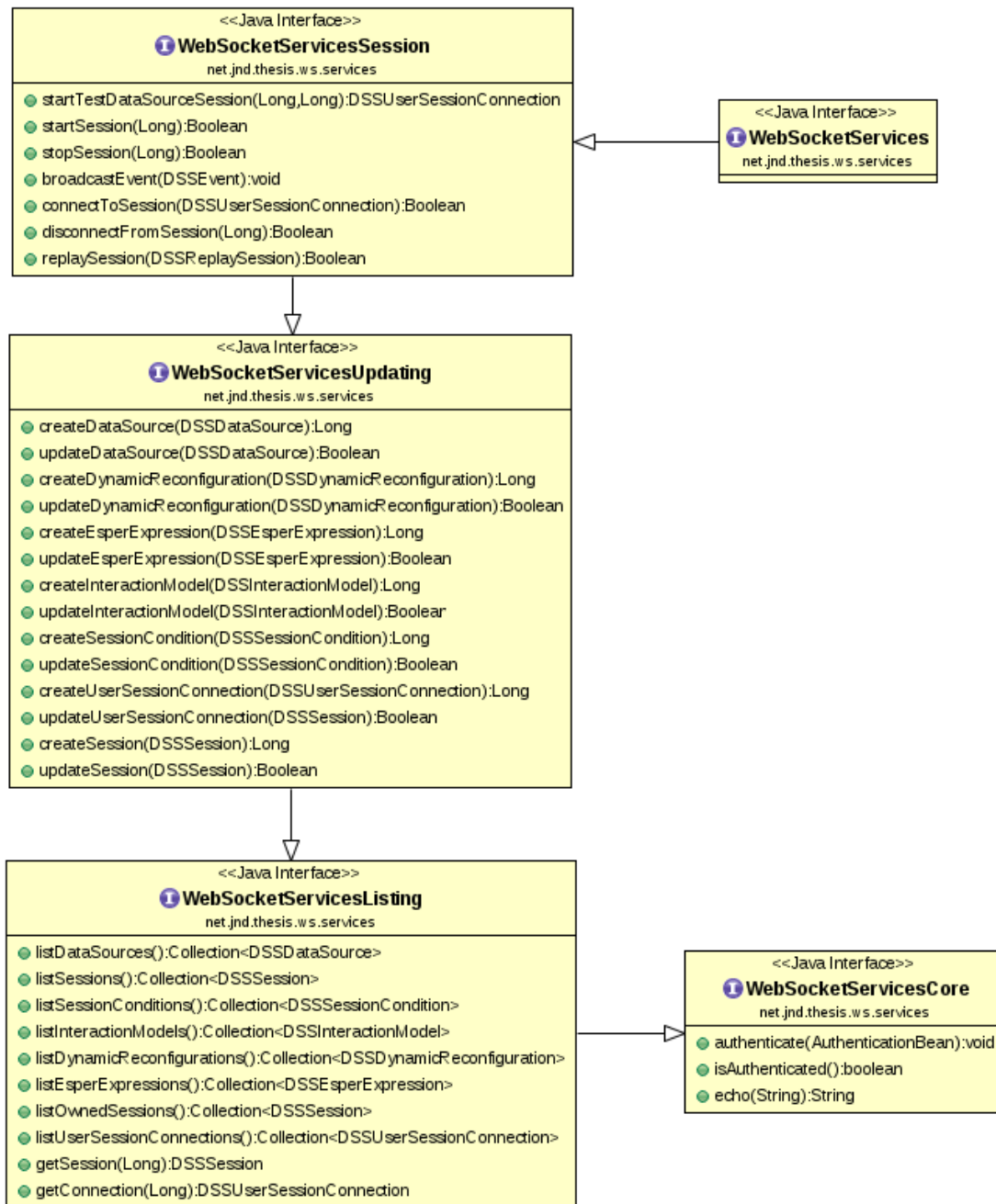


Figure 4.22: WebSocketServices

This services interface is implemented on server and client side. On server side the operations requested by the server are performed and the result of these operations is returned via the previously described *WebSocket* channels. On client side the implementation involves creating the appropriate `WebSocketMessage` object with the all the necessary attributes as described in Section 4.4. These attributes include the service name and the arguments in the form of `BaseBean` objects.

The base client side services implementation is `ClientSideServices`. This class is used by all implementing clients to handle all the base functionalities. When creating new clients to interact with the middleware it is necessary to integrate these services with the client specific graphical user interface (*GUI*). One of the key aspects of the (*GUI*) integration is the event handling operations. Each client may have different requirements and to handle these specific requirements, the base client side services implementation includes a `SessionEventHandler` object.

`SessionEventHandler` is an interface whose contract defines a single method `handleEvent(EventBean)`. In the base implementation the concrete event handler is the `DefaultSessionEventHandler` class which simply writes the events to the default output stream "System.out". Figure 4.23 illustrates this hierarchy.

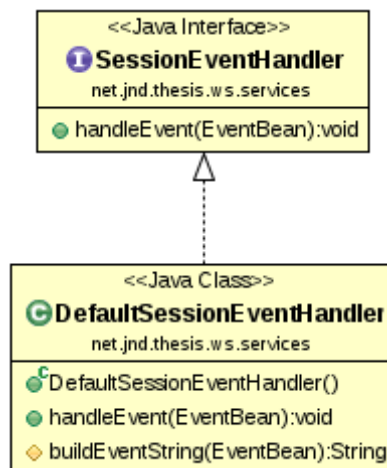


Figure 4.23: `SessionEventHandler`

To implement client specific event handling each client must define it's own event handler class that implements the `SessionEventHandler` interface. This event handler will then be used instead of the default event handler by calling the `setEventHandler(SessionEventHandler)` method included in helper class `ClientWebSocketContext`.

Figure 4.24 illustrates the `ClientWebSocketContext` helper.

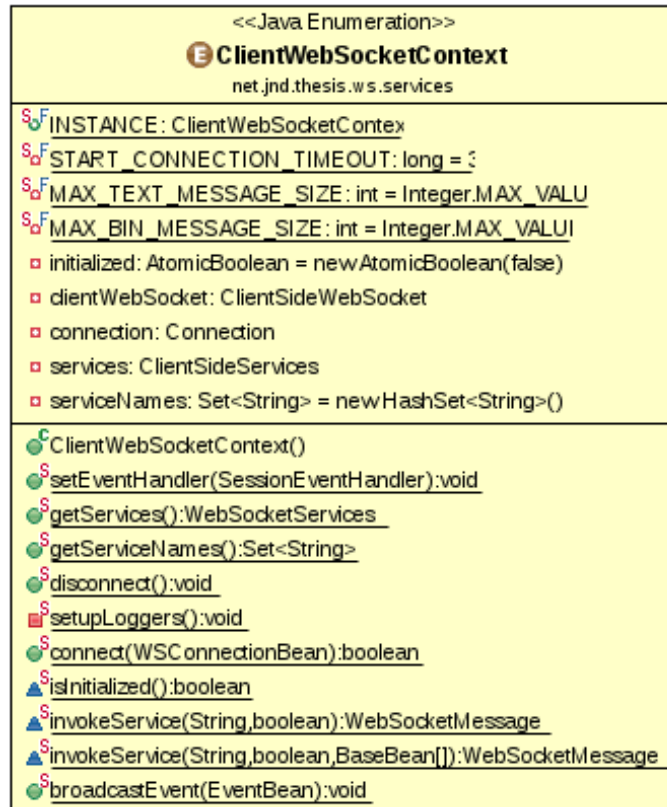


Figure 4.24: `ClientWebSocketContext`

`ClientWebSocketContext` is a *Singleton* helper that maintains client side context data such as the authentication status. This helper is also responsible for creating the `WebSocket` connection with the middleware. This operation is defined in the method `connect(WSCConnectionBean)`. The object `WSCConnectionBean` defines all connection characteristics such as the connection *URL*, the connection timeout or the maximum message size.

The next section describes the middleware cloud integration details.

4.5 Cloud Integration

Amazon Web Services [AWS] was the chosen cloud services provider for implementing this solution. The main reasons for choosing *Amazon Web Services* over *Google App Engine [GAE]* are described below.

IaaS over PaaS: As previously described in Section 2.1.3, AWS offers mainly with *Infrastructure as a Service (IaaS)* services offers a *Google App Engine* is a *Platform as a Service (PaaS)* service. For example, when configuring the deployment environment necessary to deploy the middleware a specific application service was used (*Jetty*) and this kind of configuration is not possible in GAE. There are also some limitations in the *Java Virtual Machine (JVM)* used by GAE that were an important constraint on this particular development (e.g. no explicit thread launching allowed). The *Amazon Elastic Computing Cloud (EC2)* was used so that a virtual machine could be configured from scratch with the necessary environment. The middleware was deployed in a *Amazon EC2* instance that was configured with a *Ubuntu* based operating system and a *Jetty* application server where the middleware application was deployed. The *Amazon EC2* virtual instance can be managed from the AWS EC2 management console illustrated in Figure 4.25.

The screenshot shows the AWS EC2 management console. At the top, there are buttons for 'Launch Instance' and 'Actions'. Below that, there are filters for 'Viewing: All Instances' and 'All Instance Types'. A table lists the instances:

Name	Instance	AMI ID	Root Device	Type	State	Status Checks	Alarm Status	Monitoring	Security Groups
DSS	i-aba592e0	ami-c1aaabb5	ebs	t1.micro	running	2/2 checks passed	no data	basic	quick-start-1

Below the table, the details for the selected instance 'DSS (i-aba592e0)' are shown. The 'Description' tab is active, displaying the following information:

- AMI:** ubuntu/images/ebs/ubuntu-precise-12.04-amd64-server-20121001 (ami-c1aaabb5)
- Zone:** eu-west-1a
- Type:** t1.micro
- Scheduled Events:**
- VPC ID:** -
- Source/Dest. Check:**
- Placement Group:**
- RAM Disk ID:** -
- Key Pair Name:** dss
- Monitoring:** basic
- Elastic ID:** -
- Alarm Status:** 1 of 1 in INSUFFICIENT DATA
- Security Groups:** quick-start-1. [view rules](#)
- State:** running
- Owner:** 464613434710
- Subnet ID:** -
- Virtualization:** paravirtual
- Reservation:** r-18999150
- Platform:** -
- Kernel ID:** aki-62695816
- AMI Launch Index:** 0
- Root Device:** ebs1

Figure 4.25: AWS EC2 management console

Relational over Non-relational database: The *Amazon Relational Database Service (RDS)* allowed the middleware to take advantage of cloud-based storage without resorting to a non-relational based solution which would require a greater learning curve and a larger effort for integrating the necessary tools such as *Apache Camel*. The middleware requires database access in several contexts and several data model sections were already described in the previous chapters. The *RDBMS* used is *MySQL* and the database instance is deployed in the cloud supported by *Amazon Relational Database Service*. The database communication is implemented using *Hibernate/JPA*.

Apart from supporting the entire middleware data model, one key advantage of having a cloud-based database instance is when it is used to store data events. Since *Amazon RDS* service provides scalability transparently, it is possible to use the database using the same mechanisms used in standard databases. The database instance can be managed from the *AWS RDS* management console illustrated in [Figure 4.26](#).

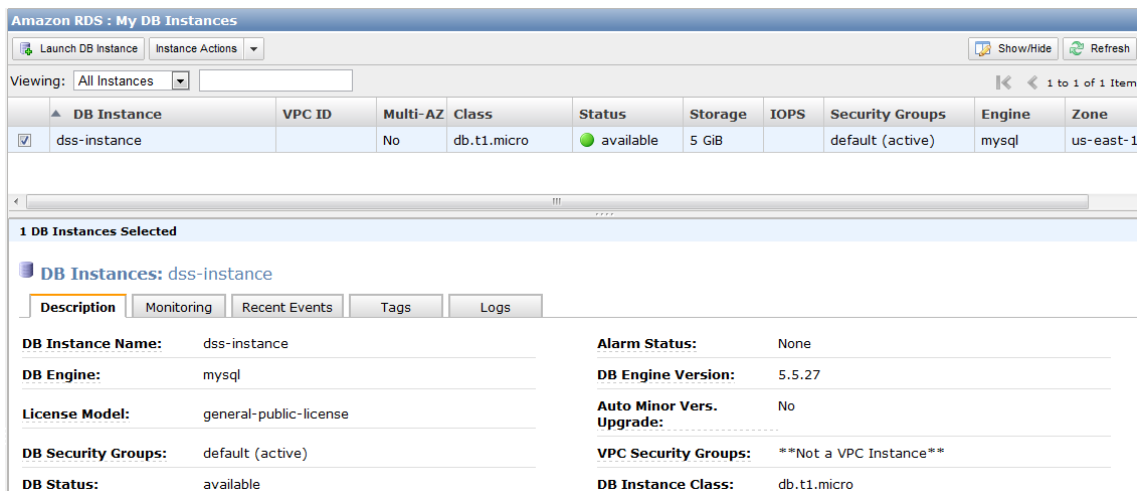


Figure 4.26: AWS RDS management console

Cloud provider maturity: Overall *AWS* services are more complete and mature than the *GAE* offer. In particular when referring to cloud deployed virtual machines, *Amazon* were pioneers with the *EC2* service and nowadays many cloud providers based their own offer on *Amazon EC2* virtual machines.

The next section will describe the web administration interface that was developed in this thesis to demonstrate how a client can manage the middleware using a standard web browser.

4.6 Web Administration

The web administration interface is a middleware management web site that clients can use to perform operations using a standard web browser. This interface allows to perform administrative tasks such as datasource creation, session definition or creating dynamic reconfigurations. There is a *CRUD* (*Create, Read, Update, Delete*) based approach that allows clients to manipulate the data model on which the middleware is based upon. Certain detail screens allow for more operations such as session start/stop on the session detail screen or session connection open/close on the user connection detail screen. The web interface also allows for clients to connect to sessions using a standard web browser. The functionalities offered by the web interface are also possible for any client by using the services API. This interface is implemented mainly using web technologies (*HTML, JavaScript*) and *Java* based technologies (*JSP, Spring MVC, Spring Security, AspectJ*). The base source code used was created using *Spring Roo* and customized to address the needs of the project. When connecting to the web interface a client must be authenticated. The authentication screen is displayed in Figure 4.27.



Datasource Sessions

Authentication

Please enter your credentials to access the web interface.

Username

Password

Figure 4.27: Web interface authentication

The home screen menu organization described the main feature categories allowed in the web interface: *data source, interaction model* and *session management*.

Data source management: this section allows clients to access the screens used for managing the data source definitions through the related `DataSource` entities. There is one menu entry for each data source type: *HTTP, RSS, Twitter* and *XMPP*.

Figure 4.28 illustrates the RssDataSource creation screen and Figure 4.29 illustrates the TwitterDataSource creation screen.

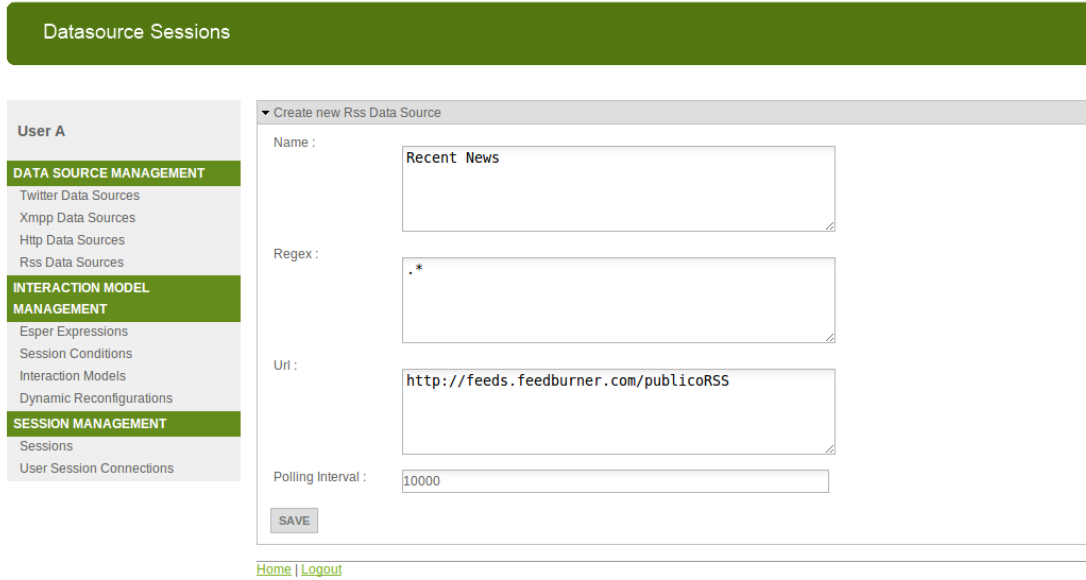


Figure 4.28: RSS data source creation screen

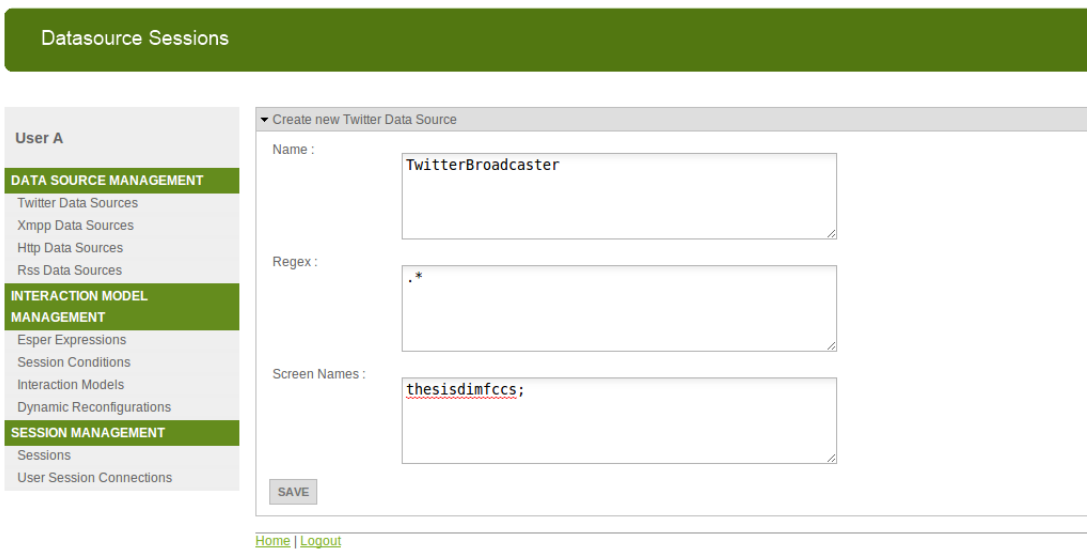


Figure 4.29: Twitter data source creation screen

Interaction model management: this section allows clients to access the screens used for managing interaction model related entities such as the `SessionCondition`, `DynamicReconfiguration`, `EsperExpression` or `InteractionModel` domain entities. Figure 4.31 illustrates the `SessionCondition` creation screen and 4.31 illustrates the `DynamicReconfiguration` creation screen.

Figure 4.30: Session condition creation screen

Figure 4.31: Dynamic reconfiguration creation screen

Session management: this section allows clients to access the screens used for managing session related entities. These entities include the `Session` and the `UserSessionConnection` objects.

Figure 4.32 illustrates the Session creation screen.

The screenshot displays the 'Datasource Sessions' web administration interface. On the left is a navigation menu for 'User A' with sections: DATA SOURCE MANAGEMENT (Twitter Data Sources, Xmpp Data Sources, Http Data Sources, Rss Data Sources), INTERACTION MODEL MANAGEMENT (Esper Expressions, Session Conditions, Interaction Models, Dynamic Reconfigurations), and SESSION MANAGEMENT (Sessions, User Session Connections). The main area is titled 'Create new Session' and contains the following fields:

- Name:** A text input field containing 'Weather Monitoring Session'.
- Allowed Users:** A list box containing 'Administrator (admin)', 'User A (usera)', 'User B (userb)', and 'User C (userc)'.
- Session Data Sources:** A list box containing 'TwitterBroadcaster', 'HumiditySensor', 'PSI 20 Index', and 'Latest News'.
- Client Model Allowed:** A checked checkbox.
- Repository Enabled:** A checked checkbox.
- Esper Expression:** A dropdown menu with 'SimplePassThrough;' selected.
- Interaction Model:** A dropdown menu with 'STREAMING' selected.
- Dynamic Reconfigurations:** A text area containing the rule 'if (Above average humidity value) then apply STREAMING'.

At the bottom of the form is a 'SAVE' button. Below the form are links for 'Home' and 'Logout'.

Figure 4.32: Session creation screen

The next section will describe the base mobile client developed in this thesis. This mobile client demonstrates some of the middleware functionalities and serves as a base implementation for other mobile applications that need to interact with middleware.

4.7 Mobile Client

The mobile client is an *Android* based application that implements the services API. Base functionalities are offered such as authentication, session listing and session connections. This mobile client is also meant to be used as a platform for richer mobile clients with specific behavior (e.g. generate actions on some specific middleware event). The main reason for choosing *Android* over *iOS* was the *Java* based programming language that allowed for a quicker learning curve. Another element that contributed to this choice was the unavailability of *iOS* supported hardware. The *Android* application developed in this thesis serves as a prototype to showcase the core middleware functionality from a mobile device client.

The initial screen is the login screen, which is linked with the `LoginActivity`. This activity collects user input from the username and password fields, connects to the middleware using the method `connect` (`WSConnectionBean`) from the previously described `ClientWebSocketContext`.

If the connection is successful, the `authenticate` (`AuthenticateBean`) service is called with the collected user input. If the authentication is not successful the user is redirected to the login screen with an error message, otherwise the user is redirected to the application home screen. Figure 4.33 illustrates the login screen and Figure 4.34 illustrates the home screen.

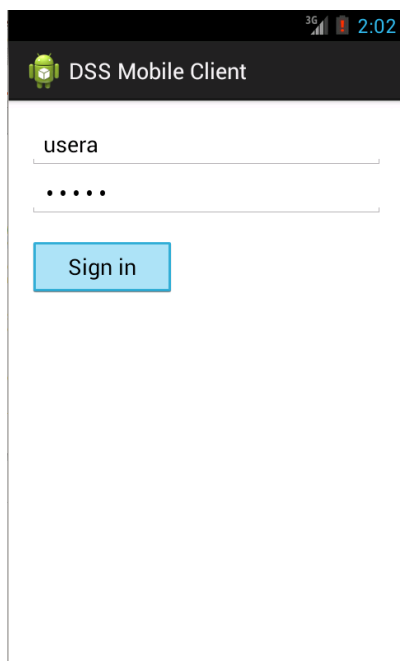


Figure 4.33: Login screen

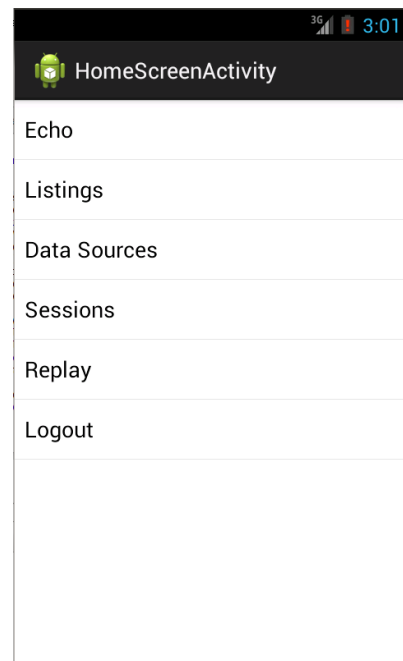


Figure 4.34: Home screen

The home screen allows clients to access six operations:

- **Echo:** This activity provides a quick connectivity test with the classic echo functionality. This operations calls the `echo()` service from the API interface `WebSocketServicesSession`. Serves merely as a bi-directional communication test. Figure 4.35 illustrates the echo screen;
- **Listings:** This menu allows clients to consult the domain entities that are accessible to them. These can include data sources, dynamic reconfigurations or sessions among others. This operations relates to the services described in the API interface `WebSocketServicesListings`. Figure 4.36 illustrates the listings screen;

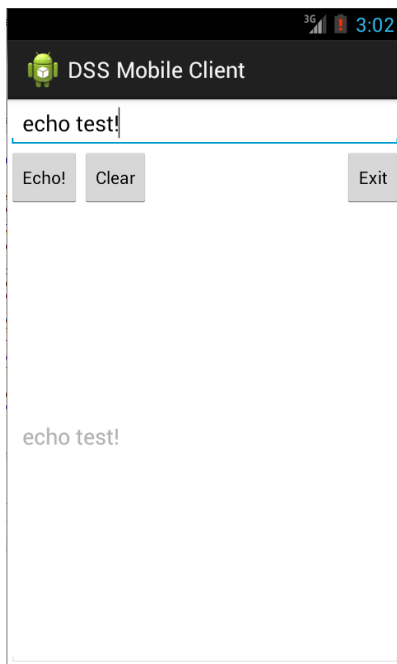


Figure 4.35: Echo screen

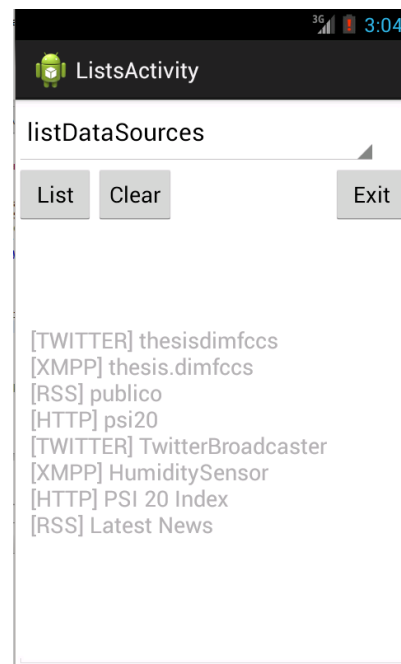


Figure 4.36: Listings screen

- **Data Sources:** This activity allows users to quickly create a test session with a single data source. On server side a temporary session will be created with a Streaming interaction model and a simple pass-through Esper expression. This should be used to test if a certain data source is working correctly. This operations calls the `startTestDataSourceSession()` service from the API interface `WebSocketServicesSession`;

Figures 4.37 and 4.38 illustrate the data source test session screen.

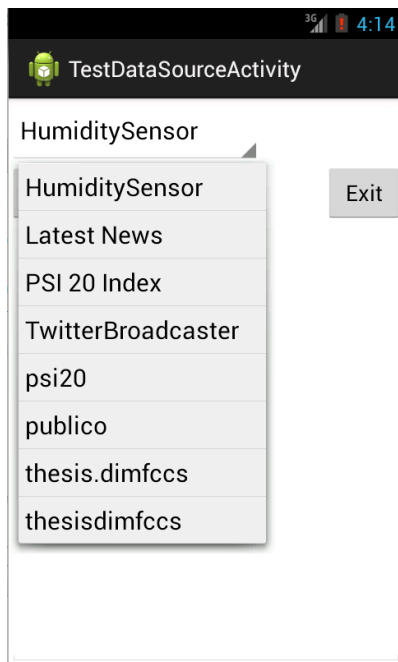


Figure 4.37: Data source test screen - selection

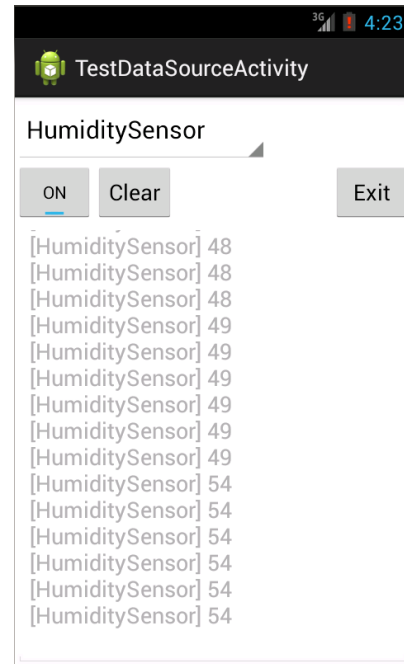


Figure 4.38: Data source test screen - connection

- Sessions:** The screen access by this menu displays a list of possible sessions from which he can choose to which session he wishes to connect. This operations the collects the chosen session and calls the `connectToSession()` service from the API interface `WebSocketServicesSession`. The screen layout is similar to the data source test screen described before, but in this screen the user selects a session instead of a data source;
- Replay:** This menu allows users to access the replay session functionality. The replay screens lists all possible sessions from which the user can selected one. This operations calls the `replaySession()` service from the API interface `WebSocketServicesSession`. The screen layout is similar to the session connection screen;
- Logout:** This menu allows users to logout from the application. This operation calls the `disconnect()` method from `ClientWebSocketContext` to close the `WebSocket` connection. On server side, all *user Camel routes* are closed as described in the previous chapters;



Case-Study

This chapter describes an application scenario that demonstrates the features of the proposed middleware. Some of the implemented features, such as the dynamic modification of the data sources, or the ability to replay and analyse previously stored data, are common to the domain of *Dynamic Data Driven Applications Systems* (as described in [Dar05]). Given these similarities, the example scenario described in this chapter will be also be contextualized in the domain of *DDDAS*.

The chapter will start by identifying the *DDDAS* domains and its dimensions, followed by a detailed description of a simulated natural disaster scenario.

5.1 Dynamic Data Driven Applications Systems

The *DDDAS* concept was first proposed by Frederica Darema [Dar05] and “entails the ability to incorporate dynamically data into an executing simulation applications, and in reverse, the ability of applications to dynamically steer measurement processes”. These dynamic data inputs include real-time data but also data that was previously stored for posterior analysis. In [Dar10] the author discusses the main characteristics of *DDDAS* and its requirements, some of which are mentioned in the following.

The *DDDAS* concept aims to improve the “modelling methods, augmenting the analysis and prediction capabilities of simulations applications” [Dar05], so that a more

accurate view of an evolving event is available. This may be achieved by integrating *fresh/live* application related data, or by dynamically selecting only the relevant data at some point in time (e.g. restricting data collection from a forest only from a specific sub-area already on fire). This selective data gathering mechanism promotes hence the “*efficiency of simulations and the effectiveness of measurement systems*” [Dar05] by reducing the amount of data that needs to be handled (e.g. using data only from the affected area, instead of from the whole forest). Since only the selected devices (e.g. wireless sensors) have to be interrogated, this contributes to reducing the energy consumption of the remainder, hence increasing their autonomy. (e.g. restricting data sensing to sensors in a sub-area allows saving other sensors autonomy, which typically is limited).

On the other hand, the dynamic inclusion of previously stored data into an simulation application also promotes this efficiency. Instead of only consuming and processing real-time data, an simulation application may filter already processed data from a repository, to extract useful information related to the evolving event (e.g. data from the same area in a previous fire event). By applying these mechanisms, the processing power demands are reduced, the sensing devices autonomy is extended, and the simulation can produce quicker results.

An additional dimension in *DDDAS* is the possibility to dynamically acquire or upscale hardware resources from the infrastructure if required [Dar10]. This is useful in areas such as simulation applications where the hardware requirements may need to adjust according to the simulation evolution. As described in Section 2.1, *cloud-based* systems allow the dynamic scaling of resources. This may include incrementing the storage space, upscaling the underlying machines to a higher-end processor or adding extra virtual machine instances as a response to usage peaks. These characteristics allow these *cloud-based* systems to respond some to some of the requirements of the *DDDAS* domain.

5.2 Urban Flooding Analysis and Monitoring

This section describes the modelling of an application scenario of our proposed solution in the domain of *DDDAS* [Dar05]. The example describes a support system for natural disaster monitoring, analysis and simulation.

5.2.1 General Considerations

The specific context for the example is an urban *flash flooding* [IBM96] emergency event resulting from unexpected heavy rain. This is a recurrent situation in certain urban areas, where sudden high levels of precipitation require a coordinated effort from local authorities. The example application that supports this scenario will be referred to as “*Urban Flooding Analysis and Monitoring*” (UFAM). The next sections describe the data acquisition process and how this example relates to the DDDAS domain. Afterwards, a simulated *flash flooding* scenario is described in detail.

5.2.1.1 Data acquisition

Meteorological data is required when *flash flooding* incidents occur, in order to build statistical information, and help to define the probability of new events. Data such as the air temperature, humidity, precipitation or wind strength can be used by human agents (or simulation applications) during the course of the event, to support all required operations. In particular, data concerning the water levels of nearby rivers and water saturation levels on the affected provide crucial information on how the incident may evolve.

Geographical data collection is also important in these scenarios to help determine how the water may flow in the affected areas. For example, hydrological models can provide terrain analysis information that can be used to determine the soil types, terrain roughness, area flatness or if there are any nearby sloped areas. Likewise, information on the presence of streams, rivers, lakes, and dams in the vicinities also should to be taken into account. Typically, geographical data is accessed from repositories, both for off-line and on-line processing. This data can be accessed by simulation applications, in contexts such as the scientific modelling of natural systems [Sys]. These simulations may be executed after a critical event to gather further data on how the event evolved, or they may be executed in real-time (e.g. during a flooding incident) to provide an insight on current status, and help determine alternate course of action.

In the context of this example, the data acquisition is performed by accessing a group of data sources, using two models: client-server requests (*pull data*), and topic subscription (*push data*). The middleware will interpret these services as data sources, and in turn, deliver this data with a different QoS to the clients that take part in a session, depending on pre-defined conditions or on their explicit request. This allows, for instance, that live data sources (e.g. humidity sensing

devices) may be accessed infrequently during a severe drought period, and in short intervals if it's more likely to rain.

Moreover, the previously mentioned data accesses may have direct or indirect costs associated with it. For instance, meteorology services that provide accurate weather predictions may be charged, and a frequent acquisition of sensing data reduces the sensor devices' autonomy, which is usually low. Therefore, in the context of a session, the variables that determine how the data is delivered to all interested entities, might not only be scenario related, but may also depend on cost assessment factors.

The Case-Study environment used simulated wireless sensors, since no real sensors were used in the course of this thesis. The example described in this chapter assumes that the live data sources (e.g. humidity or precipitation sensors) are wireless sensing devices [GPBdSA12], that use the XMPP communication protocol [HBD09]. All these sensors were simulated by using a small customized application, which injects data into sessions, so that the scenarios described in Chapter 4 could be reproduced and tested. This approach allows full control of the testing data and helped the course of the implementation and testing processes.

5.2.1.2 Characteristics related with DDDAS

As discussed in [Dar05, Dar10], a few distinctive characteristics are common to DDDAS applications. Some of these characteristics are also present in this UFAM example, and are described in the following.

This example takes into account entities such as *local authority agents*, *emergency team* personnel deployed in the affected area or an *flood simulation*. These entities are able to dynamically include additional data sources, according to pre-defined dynamic reconfiguration rules. Additionally, the session owner may explicitly include new data source definitions in the session context, while the session is running. These actions are accomplished in the context of a session that monitors an area prone to flash flooding events. In this session abstraction's implementation, these dynamic data sources can be selected from a list, which is defined while creating the session (but can later be explicitly modified).

The notion of *people in the loop* is present, since all session members observe/are notified of session events (e.g. dynamic reconfiguration events). For example, the *local authority* could define a dynamic reconfiguration rule describing an alert situation, and all session clients would be notified when this reconfiguration was triggered. This would allow all subordinated entities to include some pre-defined

action, coherent with the emergency plan, that would be triggered automatically on an alert situation (e.g. start receiving weather data more frequently). In some situations, these pre-defined behaviors can aid session clients to act based only on local information (e.g. real-time data or saved session events), without having to request new instructions. Session clients may include a *local authority* in charge of the affected area, or subordinated entities following a pre-defined emergency plan (e.g. firemen, medical emergency teams or city hall workers).

Off-line data, including all relevant session's events and data sources' processed data, can be consumed and processed by interested entities by using the *session replay* functionality. In this session abstraction's implementation, off-line data access is mutually exclusive to live-data acquisition, which implies that in order to access off-line data (i.e. data stored in the repository), the session must evolve to a status where no live-data is consumed (i.e. the session is paused). There are two off-line data types that can be stored in the repository: session events (e.g. dynamic reconfigurations, client connections) and data sources' processed data (e.g. values produced by a sensing device). Session events replaying may be used by quality assurance entities who, for instance, have to evaluate if there were enough emergency teams deployed in the area at some point in time, or if some entity joined the session too late. Processed data may be used by simulation applications to create an event evolution knowledge base, which can be offer valuable information in future similar events.

5.2.2 Example Description

A critical urban area is being monitored via wireless sensor networks (*WSN*). The Web integration of these networks is built on top of the *XMPP* protocol [HBD09], that collect data such as water levels, precipitation, humidity values and wind speed. Some of these sensors may be located at different heights from the ground (e.g. like humidity and wind values) in order to calculate more accurate values. The collected data is persisted for data mining and analysis purposes.

Having experienced flash flooding events in recent years, the area requires surveillance by the local authority department. The *fire department* may also connect to the session to monitor relevant data. In the field, firemen or emergency teams' personnel are also involved in certain phases. At some point, a flood simulation application connects to the session to perform some analysis. The next section is composed by a sequential list of scenarios that capture one possible evolution of a flash flooding event. A brief description for each scenario follows.

- A *normal scenario* represents all the periods when the probability of an emergency is low (e.g. in summer period);
- An *alert scenario* represents a higher probability of a *flash flooding* event (e.g. on fall/winter period, when significant precipitation values can occur). This assessment is based on a weather forecast warnings or on fresh data collected from the critical area (e.g. an abnormal rise of humidity values);
- An *emergency situation* occurs when a sequence of meteorological occurrences results in a actual *flash flooding* event;
- A *disaster scenario* occurs when the *flash flooding* conditions are aggravated;
- An *aftermath scenario* is reached when the situation is normalized, but it is necessary to assist the population. At this point it is necessary to perform a post-event analysis while maintaining some monitoring (e.g. the event may yet worsen);
- A *return to normal scenario* occurs when the situation is stabilized and all rescue operations are finished, hence it is possible to return to one of the initial scenarios;

5.2.3 Scenario Evolution

This section provides a detailed description of all the scenarios listed in Section 5.2.2. For each scenario the involved actors will be identified, as well as the key actions performed by each one. All relevant interactions between the actors and the middleware will also be described.

5.2.3.1 Normal Scenario

Under normal weather conditions the *local authority* agent monitors the critical area using a previously created session. This session is configured for a default scenario, where no abnormal events were recorded. In this context the *local authority* is only interested in receiving notifications regarding above average humidity levels. This is accomplished by setting up the session with a *Publisher-Subscriber* interaction model. This model allows the *local authority* to subscribe to events generated by a humidity *XMPP WSN* data source, that are above 60%. Figure 5.1 illustrates the humidity *XMPP* data source definition.



▼ Create new Xmpp Data Source

Name : Humidity

Regex : [0-9]*

Username : thesis.dimfccs@gmail.com

Password :

From User :

SAVE

Figure 5.1: Humidity data source definition

To define the interaction model there are two steps involved, first it is necessary to define a condition that will serve as a subscription topic, then it is necessary to define an interaction model object, of type *Publisher-Subscriber*, that will use the previously created condition. The interaction model will be based on the session condition that represents the subscription topic. All these operations are performed using the middleware's *Web administration interface*. Figure 5.2 illustrates the topic's definition and Figure 5.3 illustrates the *Publisher-Subscriber* interaction model definition.



▼ Create new Session Condition

Description : HumidityNormalScenario

Data Source : Humidity

Operator : GREATER

Value : 60

SAVE

Figure 5.2: Normal humidity topic definition

Figure 5.3: Publisher-Subscriber interaction model definition

The *local authority* also defines a dynamic reconfiguration rule so that whenever the humidity value increases above 70%, a new data source on precipitation values for the same area is added to the session. This is accomplished by creating a new dynamic reconfiguration that modifies the *Esper* expression associated with the session. Note that the definition of the precipitation data source must be performed in advance, similarly to the humidity data source definition. Figure 5.4 illustrates this new *Esper* expression definition.

Figure 5.4: Esper expression adds esper humidity precipitation data

This **ALERT** dynamic reconfiguration is based on a new session condition that describes the referred humidity conditions. Note that the *local authority* could have added the data source manually by updating the session definition directly. Figure 5.5 illustrates the dynamic reconfiguration creation process.

When the condition linked to this dynamic reconfiguration is matched, the associated *Esper* expression is applied and a notification is broadcasted to all connected clients. This reconfiguration modifies the session to an *aggregation session* in which notifications of humidity and precipitation values may be also combined/filtered according to a rule. In this situation, no rule is applied, i.e. both values on humidity and precipitation are sent to all clients without transformations, but a rule can be applied by the session's owner at any time.

▼ Create new Dynamic Reconfiguration

Description :

Session Condition Enabled:

Session Condition :

Event Name Enabled:

Interaction Model Enabled:

Esper Expression Enabled:

Esper Expression :

Figure 5.5: Alert dynamic reconfiguration

An example of such a transformation is to calculate the average humidity every five minutes. This is accomplished by applying the *Esper* expression illustrated in Figure 5.6.

▼ Create new Esper Expression

Description :

Expression Type :

Expression Query :

```
select avg(value), 'Humidity' as datasource,
from net.jnd.thesis.domain.Event(name='Humidity')
.win:time(5 min)
```

Figure 5.6: Calculate humidity average every 5 minutes

Moreover, the session is configured with a session repository, meaning that all session's data, including filtered data from the data sources, is saved in the repository associated to this session. To accomplish this, the flag "*Repository enabled*" must be checked when creating the session definition, as illustrated in Figure 5.7.

The session is started by pressing the "*Start session*" button present in the session detail screen. In the same screen it is also possible to *pause*, *end* or *replay* the session, by pressing the correspondent operation buttons. Figure 5.8 illustrates the session detail screen.

The client connection is performed by creating a user connection definition. The *local authority* will define client level connection characteristics, such as the client level interaction model or dynamic reconfigurations. These attributes will only be applied to this specific client and have no influence at session level. In this case, the *local authority* is responsible for the session, hence it wants to receive

▼ Create new Session

Name :

Allowed Users :

- LocalAuthority
- FireDepartment
- Fireman
- EmergencyTeam

Session Data Sources :

- Humidity
- Precipitation

Client Model Allowed:

Esper Expression :

Interaction Model :

Dynamic Reconfigurations :

`if (HumidityAboveNormal) then apply Humidity_Precipitation;`

Enable Repository:

Figure 5.7: Session definition

Enable Repository : true

Session Status : NEW

Figure 5.8: Session operations

the events exactly as they are broadcasted. To ignore client level definitions a *Streaming* interaction model and a pass-through *Esper* expression are used. The *Streaming* interaction model, applied at client level, allows all events broadcasted by the session to be delivered to this client. The pass-through *Esper* will eliminate any aggregations, so that events from all data sources are considered, again, at client level. Figure 5.9 illustrates the *local authority* connection definition. The connection itself is established by pressing the “*Open connection*” button present in the connection detail screen. In the same screen it is also possible to *close the connection*. Figure 5.10 illustrates the connection detail screen.

▼ Create new User Session Connection

Session : Urban Flooding Analysis and Modelling (owner: LocalAuthority)

Dynamic Reconfigurations :
if (HumidityAboveNormal) then apply Humidity_Precipitation;

Interaction Model : STREAMING

Esper Expression : PassThrough;

SAVE

Figure 5.9: Local authority connection definition

▼ Show User Session Connection

Session : Urban Flooding Analysis and Modelling (owner: LocalAuthority)

Status : CLOSED

Esper Expression : PassThrough;

Interaction Model : STREAMING

Dynamic Reconfigurations :

Open connection

Close connection

Figure 5.10: Connection detail

After the connection is established, the client is redirected to the session monitoring screen, where all the events are displayed. On the background, the necessary routes are created, and a client *WebSocket* connection is established, so that the middleware can push the events to the client, that in this case is a Web browser. Figure 5.11 illustrates the session monitoring screen.

Typically, the *local authority* department is the only session client at this stage, but other clients may join as well, such as the *fire department* responsible for the this area. This client configures a *Publisher/Subscriber* that subscribes to humidity values above 65%. To create this configuration, the *fire department* client must create its own interaction model object, similar to the one created before by the *local authority*. This client also creates a client side dynamic reconfiguration, so that when an **ALERT** event is received, the interaction model is modified to start receiving all the events generated in the context of the session. Figure 5.12 illustrates this dynamic reconfiguration definition.

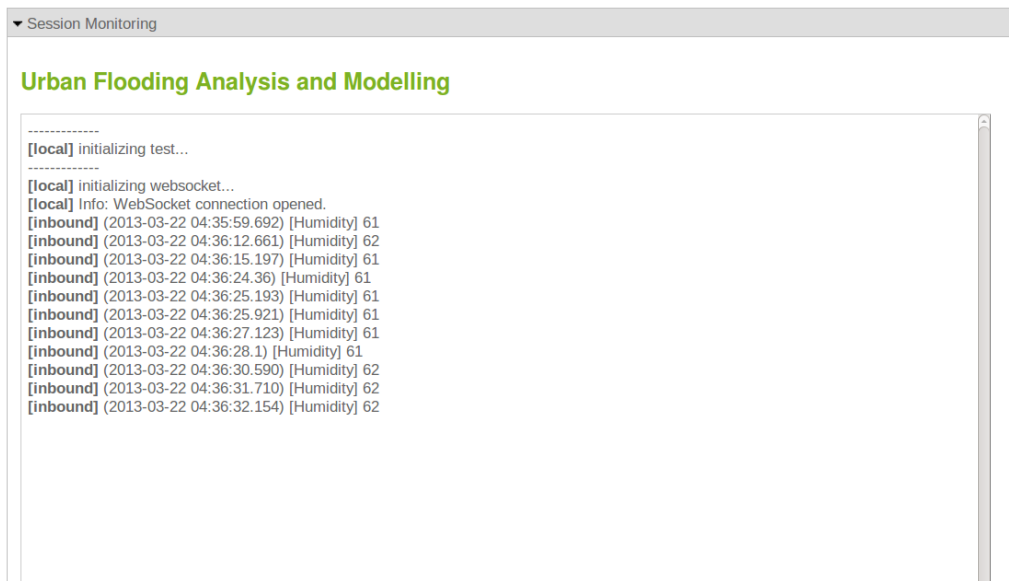


Figure 5.11: Session monitoring screen

Description :	FireDepartmentAlert
Session Condition Enabled:	<input type="checkbox"/>
Event Name Enabled:	<input checked="" type="checkbox"/>
Event Name :	ALERT
Interaction Model Enabled:	<input checked="" type="checkbox"/>
Interaction Model :	STREAMING
Esper Expression Enabled:	<input checked="" type="checkbox"/>
Esper Expression :	PassThrough;
<input type="button" value="SAVE"/>	

Figure 5.12: Fire department alert dynamic reconfiguration

To connect to the session, the process is similar to the one described above for the *local authority* connection, but since this client cannot modify the session (the *fire department* is not the owner), the interaction model will have to be defined at client level, as illustrated in Figure 5.13.

When the **ALERT** dynamic reconfiguration is triggered, all clients are notified of this particular dynamic reconfiguration event within the session context, and may respond with a specific action. This specific event represents the transition to an *Alert scenario*, which is described in the next section. The normal scenario described in this section is illustrated in Figure 5.14.

▼ Create new User Session Connection

Session : Urban Flooding Analysis and Modelling (owner: LocalAuthority)

Dynamic Reconfigurations :

Interaction Model : PUBLISHER_SUBSCRIBER (topic: Normal Humidity Fire Department)

Esper Expression : OnlyHumidity;

Figure 5.13: Fire department connection definition

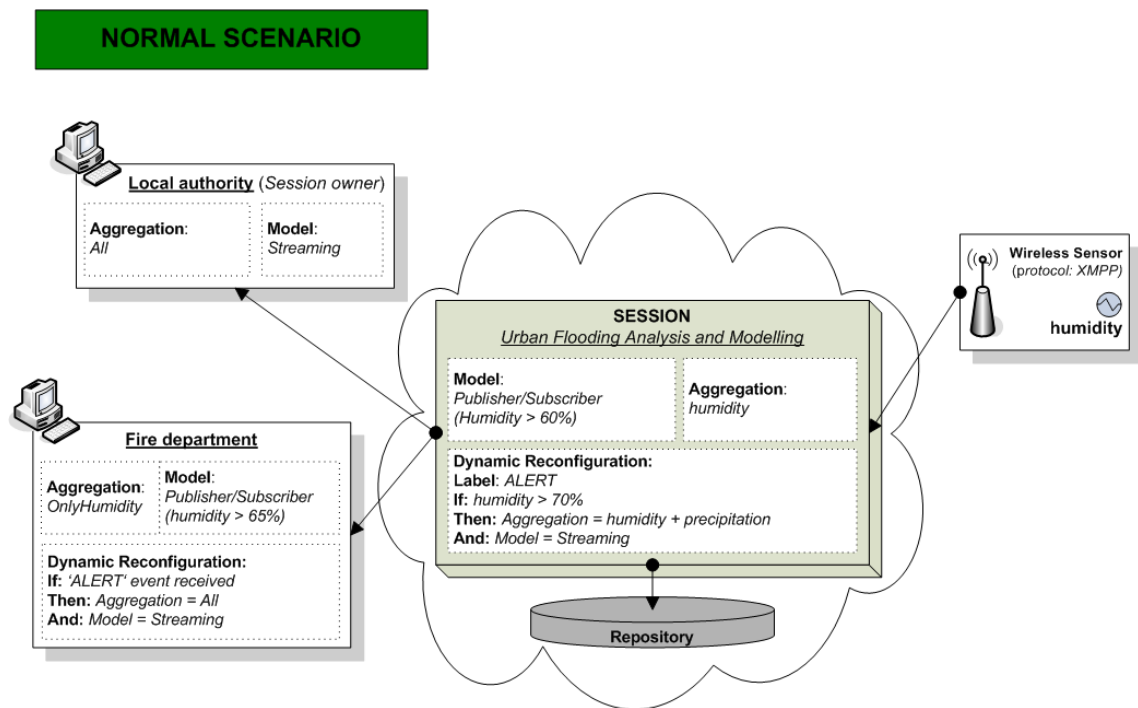


Figure 5.14: Normal scenario overview

5.2.3.2 Alert Scenario

After the **ALERT** dynamic reconfiguration is triggered, the session starts receiving precipitation values together with the humidity values already received. Additionally, each connected client can execute some specific behavior as a response to this event. In particular, due to the dynamic reconfiguration created by the *fire department* (Section 5.2.3.1), this client automatically starts receiving all the events generated in the context of a session.

Custom client applications that implement the *Services API* may implement more complex behaviors (e.g. such as sending emails or SMS messages) as a response to these events. To accomplish this kind of customized behavior, the custom client application must implement a custom session event handler class, and

integrate this event handler as described in Section 4.4.1. Listing 5.1 illustrates a custom handler that invokes a simulated email sending service.

Listing 5.1: Email sending event handler

```

1 public class MessagingEventHandler implements SessionEventHandler {
2
3     private enum EventName { ALERT, EMERGENCY, DISASTER, AFTERMATH }
4
5     /**
6      * This method would broadcast messages based on the received event
7      * using an existing email service to broadcast messages
8      */
9     @Override
10    public void handleEvent(EventBean evt) {
11        ...
12        // if the event is of type SESSION_DYNAMIC_RECONFIGURATION
13        // and the event name is ALERT, then broadcast alert message
14        switch (evt.getEventType()) {
15            case SESSION_DYNAMIC_RECONFIGURATION:
16                if (EventName.ALERT.name().equals(evt.getValue())) {
17                    DummyEmailSendingService.broadcastAlerSignal();
18                }
19                ...
20        }
21    }
22 }

```

At this stage some *firemen* join the session to monitor the situation in the field. These clients are constantly in motion, covering the entire area in danger to check for signs that the situation may worsen. To be able to connect to the session the *firemen* require a mobile device, such as a laptop or a smartphone. In this case they connect to the session by using an *Android* based smartphone that includes the mobile client application described in Section 4.7. The connection's characteristics were previously configured by the *fire department*, so that the *firemen* only have to choose an existing connection configuration, thus speeding up the session connection process. The existing connection definition assumes that the only relevant events at the time are related to the precipitation values. Figures 5.15 and 5.16 demonstrate how each *fireman* connects to the session, using the *Android* based mobile client.



Figure 5.15: Mobile session connection

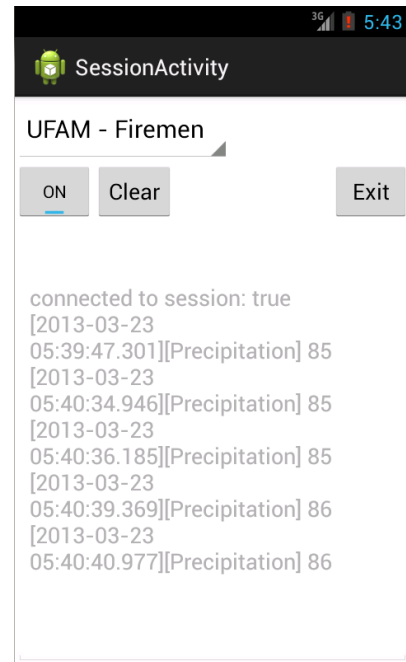


Figure 5.16: Mobile session events

These *firemen* connection notifications can be followed by the *fire department*, to verify the operational responsiveness. Figure 5.17 illustrates a broadcasted *fireman* connection event, as perceived by the *fire department*.

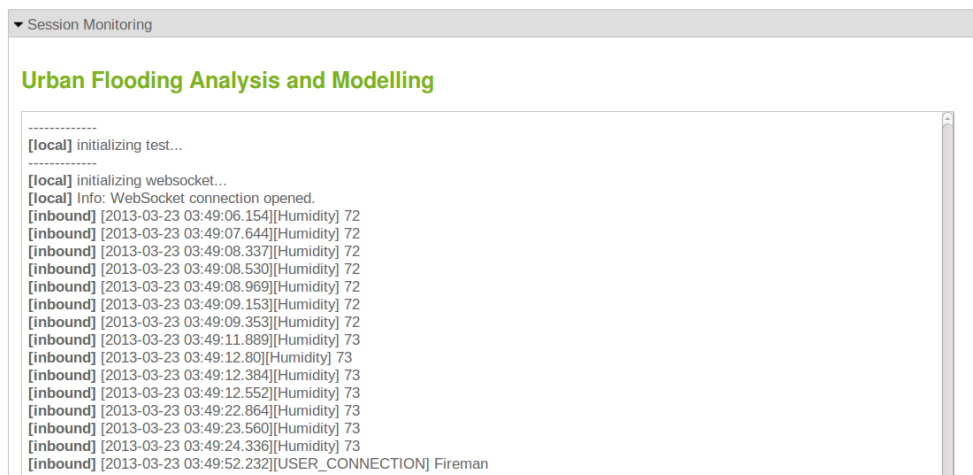


Figure 5.17: Fire department monitoring new firemen connections

An alert scenario represents a high probability of a flash flooding event. As a response to this situation some *emergency teams* are instructed to move into the

area and join the same session. The connection definition used by these teams is not client customized, hence they will be notified of the same events as all the clients that did not define a custom client interaction model (e.g. the *local authority*). The *emergency team* personnel connection process is similar to what was previously described for the *firemen*.

In order to obtain more data on the possible evolution of the situation, the *local authority* explicitly adds new data sources providing information from the area on wind speed and direction. These data sources are defined by following the procedure established for the remainder in Section Section 5.2.3.1. Figure 5.18 illustrates this explicit data source addition.

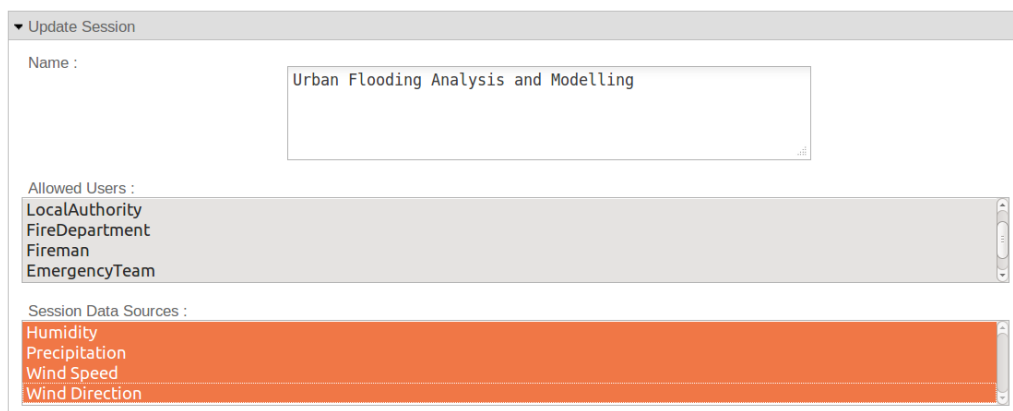


Figure 5.18: Wind speed and direction data source addition

Additionally, the *local authority* creates a **EMERGENCY** dynamic reconfiguration rule, so that when the precipitation values rise above 130mm an emergency scenario is set. In this case, a new data source on ground water level values for the same area is added to the session. To implement this behavior, the *local authority* creates a new dynamic reconfiguration, as described in Section 5.2.3.1. Figure 5.19 illustrates the new reconfiguration being added to the session, and Figure 5.20 illustrates this reconfiguration definition.

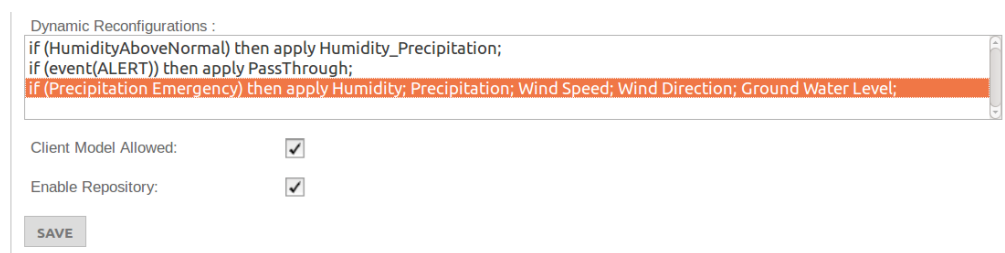


Figure 5.19: Emergency dynamic reconfiguration

▼ Create new Dynamic Reconfiguration

Description :

Session Condition Enabled:

Session Condition :

Event Name Enabled:

Interaction Model Enabled:

Interaction Model :

Esper Expression Enabled:

Esper Expression :

Figure 5.20: Emergency dynamic reconfiguration definition

Figure 5.21 illustrates the alert scenario described in this section.

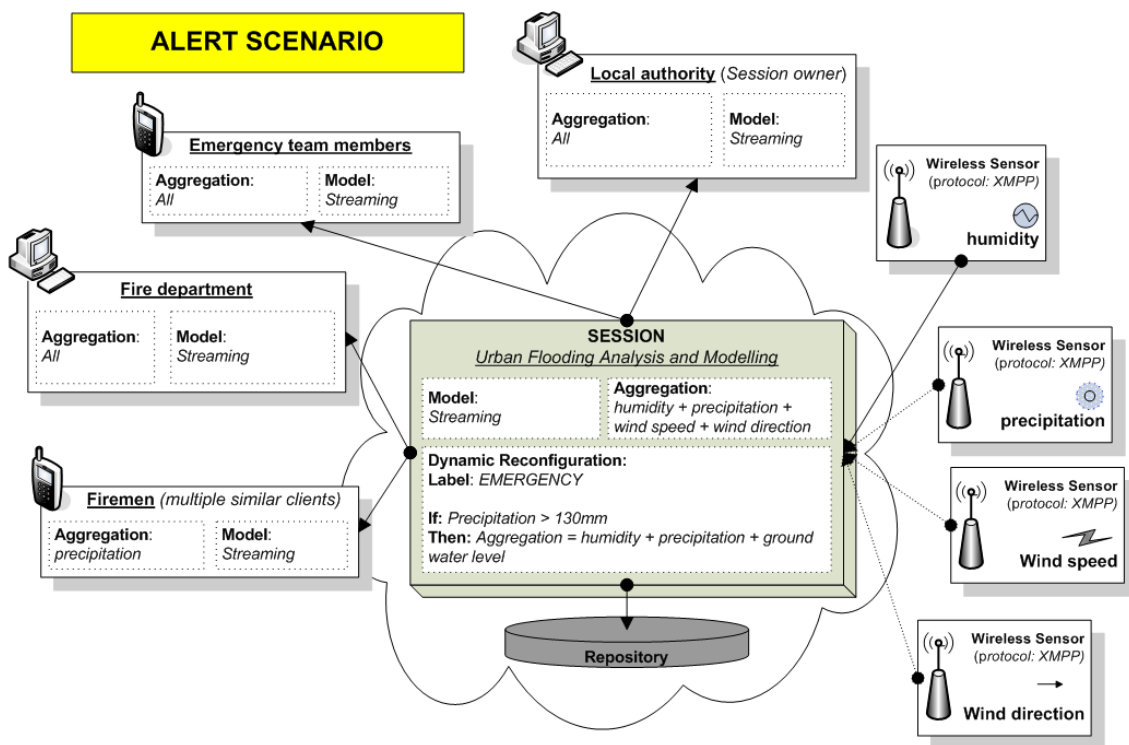


Figure 5.21: Alert scenario overview

This **EMERGENCY** dynamic reconfiguration to an aggregation session is then notified to all session members, and the novel aggregated data is automatically disseminated to all clients. Note that all the data generated in the context of the session, including this aggregated data, is being persisted for later processing and can be monitored by additional agents, as will be described ahead.

5.2.3.3 Emergency Scenario

An emergency situation is reached when a sudden rise on the precipitation values results in a *flash flooding* event. In this scenario the *local authority* adds new data sources that provide water levels of dams, lakes and rivers, located in the vicinities. These have been identified as a possible risks, so it is necessary to monitor them. These data sources are defined using similar processes as in the previously referred data sources (Section 5.2.3.1), and they are explicitly added to the session by the session owner (the *local authority* client). Figure 5.22 illustrates how the *local authority* includes this new data sources in the session context.

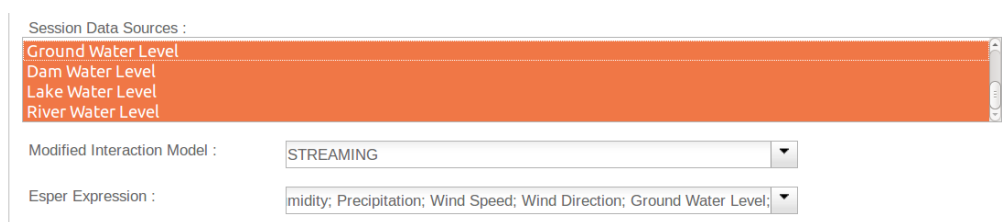


Figure 5.22: Local authority adds new data sources

To provide additional information regarding these possible risks, the *local authority* starts a *flood simulation* to analyse and predict how the situation may evolve. This application also becomes a session's client so that it can inspect and process all relevant events. This simulation will help *local authority* agents on the evaluation of this scenario. The simulation application is assumed to execute on a *cloud* platform, so that it is possible to dynamically provision additional storage resources as well as computational ones. Also, the simulation application could be organised in modules which would be executed on-demand as necessary (e.g. first an analysis simulation would be launched, then a prediction tool).

This *simulation application* was not implemented in this work, and the goal of this description is to refer that such an application could be integrated in the session context. In this example, the simulation application client would be previously configured in the *Web administration* interface. This includes setting up the simulation client credentials, including the simulation client in the existing session, and setup a new client connection. Afterwards, the simulation would be launched (e.g. from the command line), using the login credentials and connection identifier as arguments. This simulation application would then performs calculations to help the *local authority* to predict the evolution of this situation.

The mobile devices used by the some clients may experience low battery levels during the course of the event. If such situations occur, the situation is handled transparently by the *Android* mobile client, by triggering a pre-defined rule

that automatically adjusts the interaction model in use. This rule states that if the mobile client is currently connected to a session, and the battery levels drop dramatically, then the client interaction model is explicitly modified to a *Producer-Consumer* model, that will consume data once every five minutes. This modification is transparent to the client. Listing 5.2 illustrates the *Android* listener class that requests this interaction model modification.

Listing 5.2: Android client low battery event

```

1 BroadcastReceiver batteryLevelReceiver = new BroadcastReceiver() {
2   public void onReceive(Context context, Intent intent) {
3     // if the battery low
4     if(intent.getIntExtra("level", 0) < LOW_BATTERY_THRESHOLD) {
5       InteractionModelBean currentModel = getCurrentInteractionModel();
6       // modify the model to a PRODUCER-CONSUMER (every 5 minutes)
7       currentModel.setPattern(Pattern.PRODUCER_CONSUMER);
8       currentModel.setEventDelay(5L*60*1000);
9       // update the interaction model object on server side
10      ClientWebSocketContext.getServices().updateInteractionModel(model);
11      ...

```

Figure 5.23 illustrates an overview of the scenario at this moment.

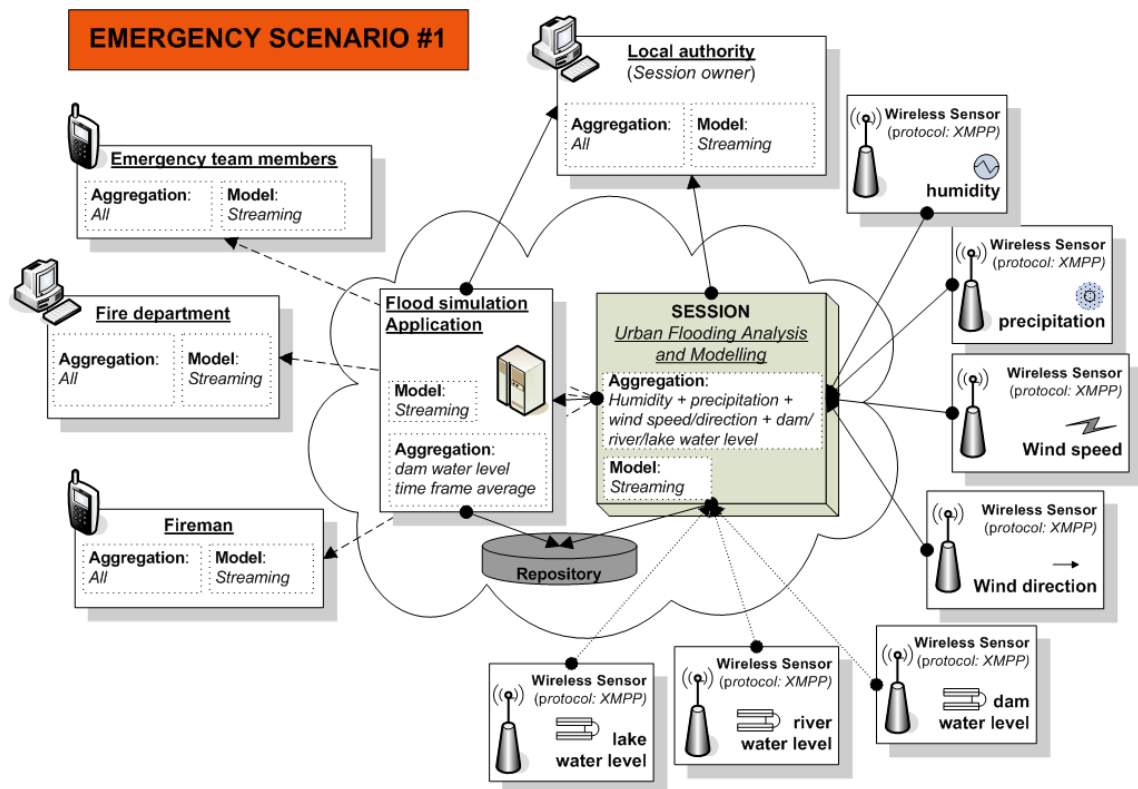


Figure 5.23: Emergency scenario intermediate status

Based on the simulation results, and in order to reduce the amount of data that has to be processed, the *local authority* tunes the data sources used by the simulation in order to restrict the data acquisition to a certain sub-area. This characteristic of live data influencing a running simulation and, in turn, the simulation results, is a major characteristic of the *DDDAS* area, i.e. “*closing the loop*”. This is accomplished by explicitly removing some of the previously added data sources from the session *Esper* expression.

During this phase, the *local authority* creates a dynamic reconfiguration rule to be triggered if the scenario worsens. The label for this dynamic reconfiguration is **DISASTER**. This dynamic reconfiguration will be based on the ground water level data source defined in Section 5.2.3.2. When the ground water level raises above 200mm, it means that the situation has aggravated dramatically. The dynamic reconfiguration is then triggered, modifying the session interaction model to a *Producer-Consumer* model that consumes events every ten seconds. This will ensure that no data is lost, even in the presence of a bad/intermittent network connection, while delivering data with a high frequency. Figure 5.24 illustrates this dynamic reconfiguration definition.

Figure 5.24: Disaster dynamic reconfiguration

This **DISASTER** dynamic reconfiguration is then broadcasted to all session clients, allowing each client to perform some specific behavior if necessary. This specific event represents the transition to a *Disaster scenario*, which is described in the next section. Figure 5.25 illustrates the alert scenario described in this section.

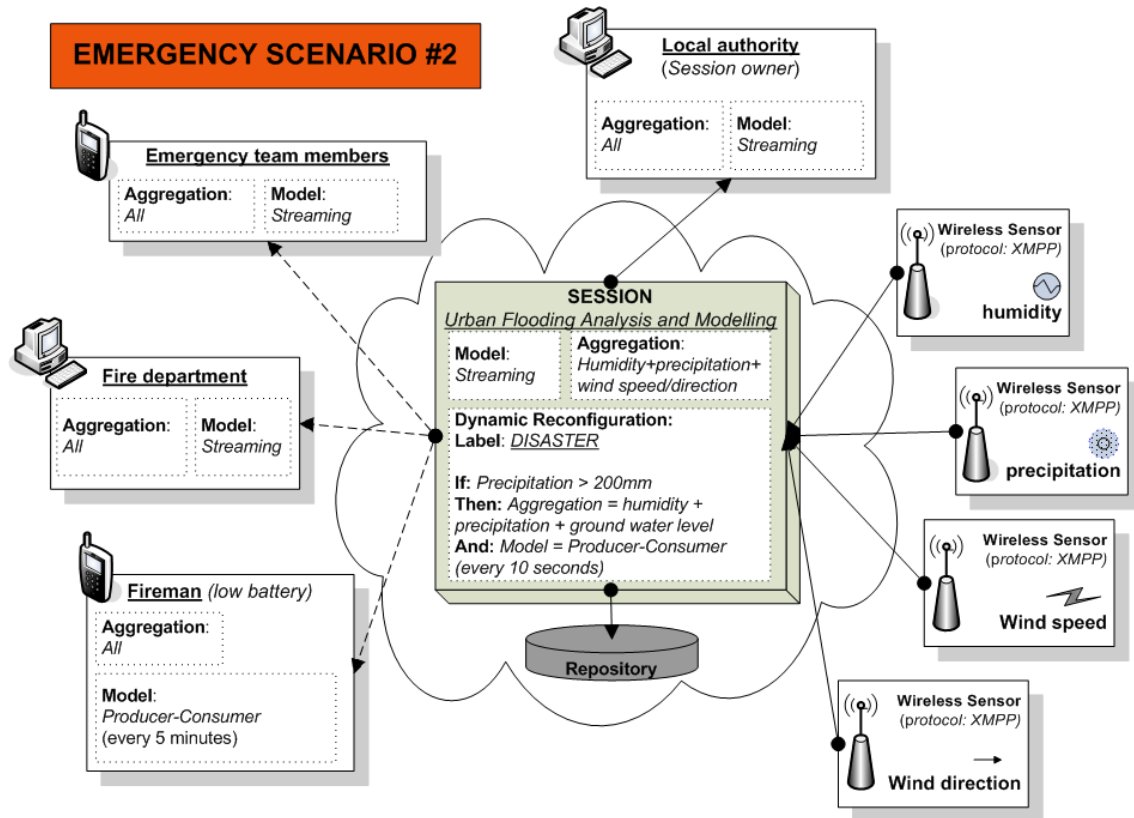


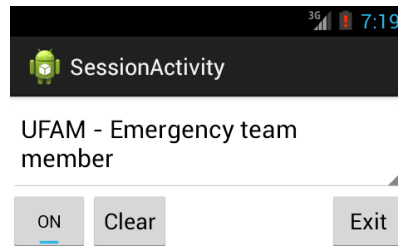
Figure 5.25: Emergency scenario overview

5.2.3.4 Disaster Scenario

When a *DISASTER* scenario is reached, new *firemen* and *emergency staff* personnel are deployed and join the session. These new clients start receiving the same information as the other clients in the session. Since the previous dynamic reconfiguration described in Section 5.2.3.3 modified the interaction model to a *Producer-Consumer* model, all new clients that joins the session from this point on, may receive the produced data at their own pace, by connecting/disconnecting from the session when necessary (e.g. to preserve battery life). Whenever they join the session, they will consume all produced data and, in this way, receive the same context information as all the other clients. The session concept provides hence a simple way for context sharing under such difficult circumstances.

The establishing of a disaster scenario also results in a message that is broadcasted to all clients connected to the session. Figure 5.26 illustrates a *emergency team* member receiving this notification, using a mobile client.

At this point, the population of the affected area is in danger, and several rescue operations are on course. Several civilian volunteers are cooperating with the rescue teams in these operations. In this context, the *local authority* decides to



[2013-03-23 07:09:39.137][EVENT]
DISASTER

Figure 5.26: Mobile client receives disaster notification

add a new *Twitter* data source to the session. This data source will allow the *local authority* to monitor *tweets* from several accounts. These accounts may belong to volunteers, helping in the field, to enable them to communicate and input information to the session context. This data source can also be used by any of the clients already connected to the session, providing an extra information input. Figure 5.27 illustrates how this *Twitter* data source is defined.

Figure 5.27: Twitter account setup

Meanwhile, the *local authority* creates a new dynamic reconfiguration to be triggered when the flood conditions are again normalized. This dynamic reconfiguration will be based on the ground water level defined in Section 5.2.3.2. When the ground water level drops below 50mm, this means that the water levels are now at standard levels. The dynamic reconfiguration is then triggered, removing unnecessary data sources from the session. From the field sensors,

only the ground water level and precipitation data sources are maintained after this reconfiguration. The *Twitter* data source is also maintained so that the field personnel may continue to provide feedback. The interaction model remains as *Publisher/Subscriber*, but the interval will be increased so that information is received once every thirty minutes. The label for this dynamic reconfiguration is **AFTERMATH**. Figure 5.28 illustrates this dynamic reconfiguration definition.

▼ Create new Dynamic Reconfiguration

Description :

Session Condition Enabled:

Session Condition :

Event Name Enabled:

Interaction Model Enabled:

Interaction Model :

Esper Expression Enabled:

Esper Expression :

Figure 5.28: Aftermath dynamic reconfiguration

Figure 5.29 illustrates the disaster scenario.

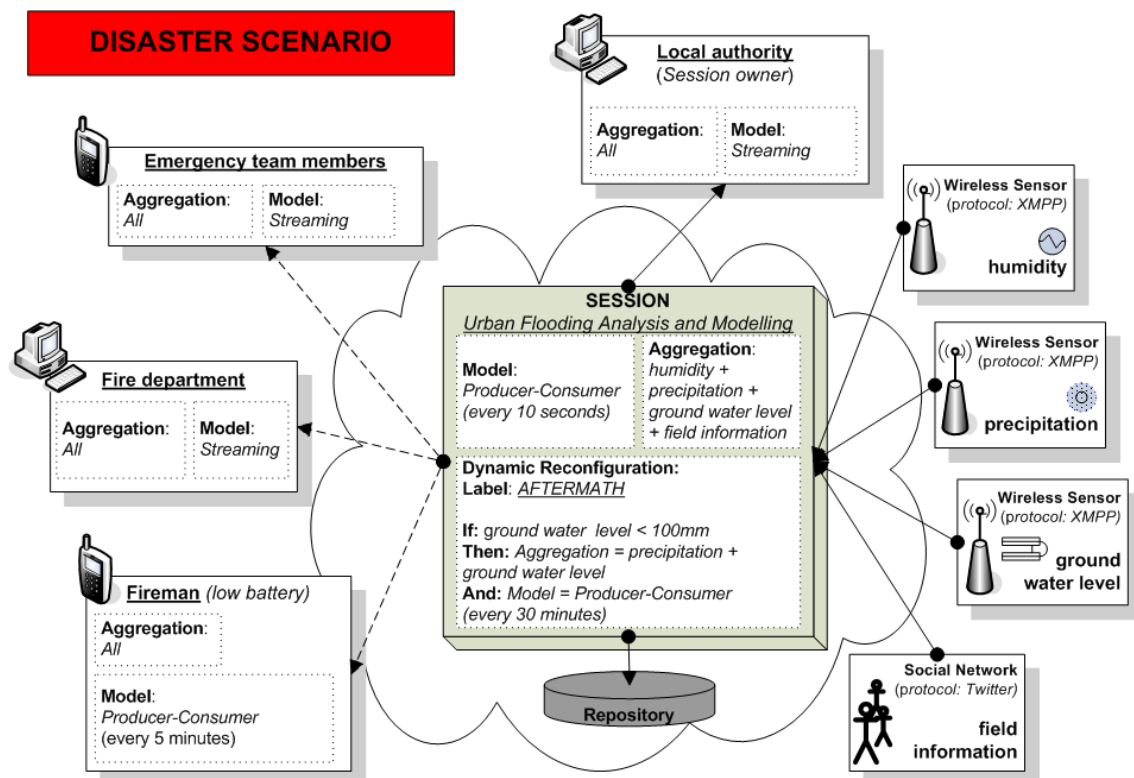
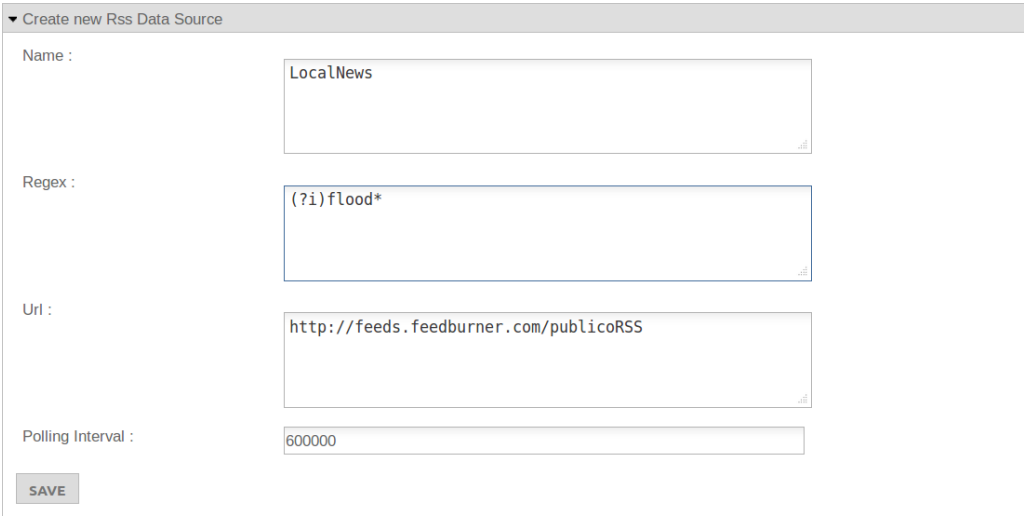


Figure 5.29: Disaster scenario overview

The **AFTERMATH** dynamic reconfiguration is then broadcasted to all connected clients, and each client may trigger any specific behavior. This specific event represents the transition to an *Aftermath scenario*, which is described in the next section.

5.2.3.5 Aftermath Scenario

When an aftermath scenario is established it is necessary to perform an after-event analysis and still remain in a prevention situation. In a first phase, it is necessary to answer all help emitted by the population. To assist this process an additional news feed is added to the session, in the form of a *RSS* data source that generates messages from the local newspaper. The session will interrogate this data source every ten minutes for recent news that include the term “*flood*”. The news filtering is accomplished by applying a regular expression on the data source generated events. This news feed, together with the *Twitter* data source described in the previous section, will help field personnel to direct the operations according to the broadcasted information. Figure 5.30 illustrates the news feed data source definition.



▼ Create new Rss Data Source

Name : LocalNews

Regex : (?i)flood*

Url : http://feeds.feedburner.com/publicoRSS

Polling Interval : 600000

SAVE

Figure 5.30: RSS news feed definition

Upon the conclusion of the rescue operations, the *fire department*, *firemen* and *emergency team* members leave the session. At this point, the session is “*paused*” by the *local authority*, so that all the data stored by the session can be reviewed.

This may be of interest to scientific experts who post-analyse the flash flood evolution, or to quality assurance entities who, for instance, have to evaluate if there were enough emergency teams deployed in the area at some point in time or if some entity joined the session too late. In particular a *quality assurance* responsible joins the session at this point. In order to allow all connected clients to review the sessions' events, the *local authority* requests a *session replay* operation, by pressing the respective button in the session detail screen. The output for the session replay will be similar to what the standard session monitoring, as described in section 5.2.3.1, the only difference being that the events are now past event.

During this "off-line" session, new clients may join the session as well. For this analysis, each client can also define which data should be received, based on date/time of datasource related constraints. These definitions are accomplished by modifying the user connection interaction model and setting up a new *Esper* expression. For instance, figure 5.31 illustrates the configuration for on client who is only interested on the groundwater levels collected on the last hour.

▼ Create new Esper Expression

Description :

Expression Type :

Expression Query :

```
select *
from net.jnd.thesis.domain.Event as e
where e.name = 'GroundWaterLevel'
and e.date > current_timestamp.minus(60 minutes)
```

Figure 5.31: Ground water level values from the last hour expression

In case of the situation of the "session replay" described above, new clients will only receive the events sent to all clients from the point in time when they join the session. For simplification reasons, clients do not have access to live data at this point (i.e. fresh data collected from the data sources). The *local authority* can, however, resume the session at any time, so that the *session replay* is interrupted and live data is again delivered to all connected clients. This process should be well handled with criteria, since while the session is being replayed, the area is not being monitored.

After the analysis is finished, the *local authority* defines a new **NORMAL** dynamic reconfiguration, that will reconfigure the session back to a normal scenario. The reconfiguration will be triggered when there is nearly zero precipitation. This dynamic reconfiguration is illustrated in figure 5.32

▼ Create new Dynamic Reconfiguration

Description :

Session Condition Enabled:

Session Condition :

Event Name Enabled:

Interaction Model Enabled:

Interaction Model :

Esper Expression Enabled:

Esper Expression :

Figure 5.32: Back to normal reconfiguration

Figure 5.33 illustrates the aftermath scenario, with the main actors and data sources involved.

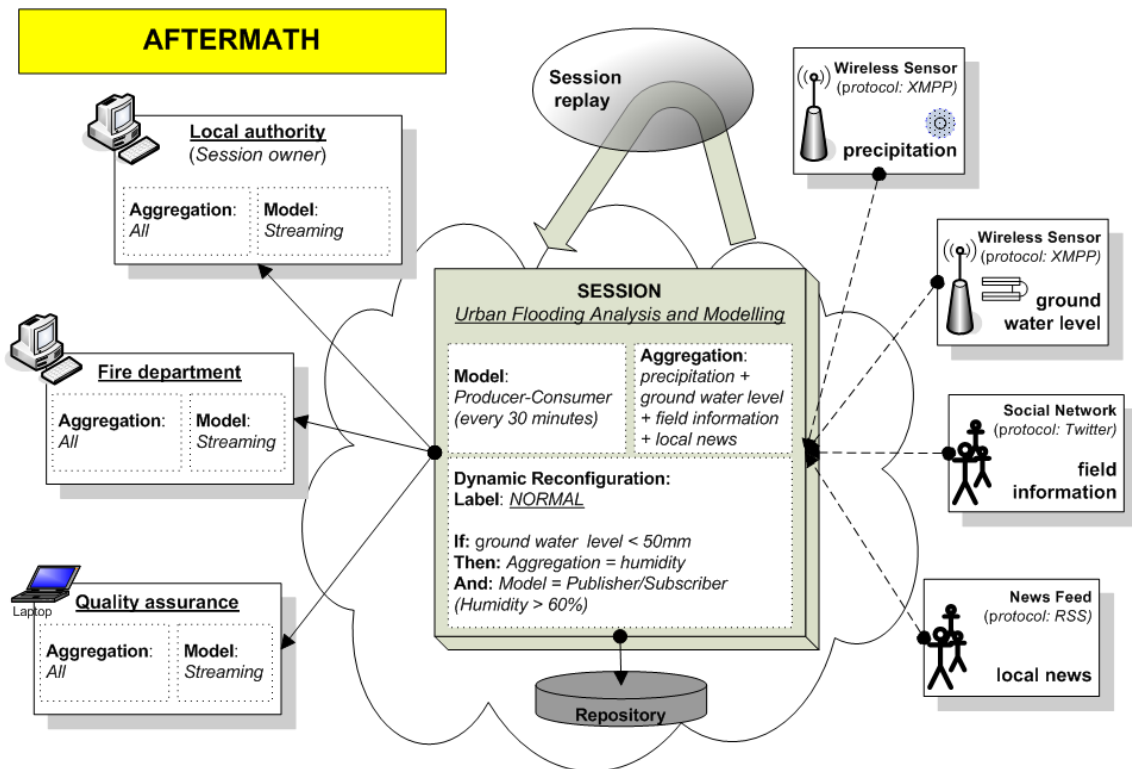


Figure 5.33: Esper expression

5.2.3.6 Return to a Normal Scenario

When all operations are finished, the session is configured back to a normal configuration, as a response to the **NORMAL** dynamic reconfiguration. The *local authority* could have also explicitly modified the session to match the initial configuration, instead of relying on a dynamic reconfiguration. The *local authority* could also “*end*” the session and create a new session with the same characteristics, thus ensuring a clean configuration, and fresh repository data, which is only linked with this new session. In this scenario, the configuration is the same as described for the normal scenario in Section 5.2.3.1.

6

Conclusions

As described in Chapter 1, the computing resources delivery model is shifting towards *XaaS* (everything as a service) providing a true decoupling between service access and service implementation, may this be elastic storage capabilities, high-performance computing power, aggregation of small entities in the context of the *Internet of Things (IoT)*, or widely used applications like *Twitter*.

Moreover, the service access context is becoming increasingly ubiquitous as a result of the mass usage of mobile devices such as *smartphones* or *tablet* devices.

Cloud computing answers some of the challenges in these environments by offering distinct service delivery models (e.g. *IaaS*, *PaaS*, *SaaS*) with simplified resource management and ubiquitous access, and by offering optimized services for mobile devices.

Considering service-based platforms, several relevant characteristics can be identified when describing client-service interactions. Namely, they can relate to the service status (e.g. low storage space, high *CPU* usage, the rate at which a stateful resource interfaced by a service is generating data, etc.); the client status (e.g. low mobile device battery level); or the communication medium (e.g. communication protocol changes from *2G* to *3G*).

By aggregating all these service access characteristics, it is possible to define interaction models, which in turn describe how each service is being accessed by a particular client or set of clients, at a particular point in time.

In the *XaaS* context described before, there is a growing demand for richer and more adaptable interaction models so that the service/data access characteristics

can be adjusted to each user's requirements or may reflect dynamic modifications in the provided service.

This adaptability should ideally take into account context variables from the three dimensions previously referred: the service status, the communication medium or user context variables. The Session concept defined and implemented in this work responds to some of these requirements by capturing these interaction models and by allowing their dynamic reconfiguration according to particular context variables.

The work in this thesis combines the concepts of Session and a Cloud-based middleware that supports the implementation of a Session abstraction. This Session captures the access of a set of clients with similar interests in a set of services/generic data sources, building a common context where all relevant dynamic events are disseminated to the interested session clients. Therefore, on one hand the implemented middleware offers Session abstraction's characteristics (e.g. the dynamic reconfiguration of the session's interaction models in use at some point in time, and the possibility to share the service access context to novel clients with minimal effort). On the other hand, the middleware enriches the Session abstraction with Cloud deployed applications characteristics (e.g. large data storage and wide accessibility).

In the following, the thesis work that led to the implemented middleware is discussed, as well as the work contributions followed by some guidelines regarding possible future work extensions.

6.1 Discussion

The proposed solution in Section 1.2 describes characteristics such as application ubiquitous access or the existence of persistent Sessions, so that the events produced in the context of a Session could be stored. To this extent, the Session abstraction implemented in this thesis was deployed in a Cloud platform (*Amazon EC2*), to take advantage of Cloud characteristics such as the elastic storage. The implemented Session abstraction characteristics are extensions to an existing Session abstraction implementation [BGP12], as described in Section 3.3. They include the notion of session repository (to store the events produced by the session) and simplified support for heterogeneous data sources (for integrating data sources based on multiple protocols).

However, one characteristic of the existing Session was not included in this implementation: the separation between behavioural and structural patterns and

the associated validation mechanisms needed to prevent incoherent operations on them. For simplification, due to time constraints, the Session implementation discussed in this work considers only behavioural patterns (e.g. *Publisher/Subscriber*, *Producer/Consumer*) which are validated by the underlying data model, preventing thus incoherent operations.

Nevertheless, the Cloud-based middleware together with the Session implementation developed in this work, respond to many of the problems/limitations identified in Section 1.1, as described in the following.

- By using a Cloud-based storage service, it is possible to store large amounts of data without compromising the overall performance, by using elastic storage characteristics. It's also possible to apply database replication, or scale the underlying hardware transparently, by using a specific Web administration interface offered by the Cloud provider;
- The usage of protocol connectors, based on the *Apache Camel* system, allows the integration of heterogeneous data sources, using a wide array of protocols. Implemented connectors include *XMPP*, *Twitter*, *HTTP* and *RSS*, but it is possible to develop and integrate further connectors if necessary. All the data produced by these data sources are handled and delivered in uniformly;
- The Session implementation allows dynamic reconfigurations of the interaction models. These reconfigurations can be performed explicitly/on-demand or they can be automatic, i.e. triggered by pre-defined rules which can themselves be modified dynamically;
- Ubiquitous access is improved by using the base *Android* mobile client developed in this thesis work. This client can be extended if necessary.

Overall, the main goals defined for this thesis were achieved. The next section describes this work's main contributions.

6.2 Contributions

As described in Section 1.3, this thesis work's contributions are mostly related to the Cloud-based middleware and the Session abstraction implementation. Namely,

- **Cloud-based middleware for the Session abstraction:** As described in Sections 3 and 4, a middleware was deployed in the Cloud using the *Elastic Computing Cloud (EC2)* service, offered by *Amazon Web Services (AWS)*. This middleware instantiates the Session abstraction for service/data aggregation and filtering. The middleware also includes a Web based administration that can be used to perform administrative tasks (e.g. new client profile definitions), and to execute operations such as to start/pause one Session or connecting users to existing sessions. The middleware is also used for creating dynamic reconfigurations, session conditions, *Esper* expressions, and all the remaining operations that modify the data model;
- **Heterogeneous Data Sources:** The middleware allows data sources' access using a wide array of protocols, by using the *Apache Camel* based connectors developed in this thesis. This feature allows a unified access to heterogeneous data sources. Moreover, the implemented architecture allows to integrate further connectors if necessary, as long as the necessary protocols are supported by *Apache Camel* (e.g. *Web Services, JMS*);
- **Session/client level interaction models:** The Session implementation allow interaction model definition at Session level or at client level. At Session level, the interaction model impacts all the clients that are connected to the session, while at client level, the interaction model modifications impact only that specific client. Session level modifications can only be performed by the Session owner, while client level modifications can only be performed by that specific client;
- **Rule-based dynamic reconfigurations:** This session implementation allows dynamic reconfiguration based on rules. These rules can also be defined at Session or client level. When a Session level dynamic reconfiguration is triggered, a specific event is broadcasted to all clients connected to that session, stating that has occurred the dynamic reconfiguration rule labeled X (each rule has an associated label). These rules can be based on the values produced by the data sources (e.g. if humidity > 60% then modify pattern to *Streaming*), or they can be triggered by a Session level dynamic reconfiguration that was triggered (e.g. if rule X was triggered, then modify pattern to *Streaming*). The later approach allows the composition of chained rules where one rule triggers additional rules consecutively, thus triggering complex modifications that can impact each Session client specifically (i.e. if each client defines a client level rule based on that event);

- **Repository and session replay:** The notion of an events' repository is also included in this Session implementation. The stored events include data source generated values and internal Session events, such as the triggering of dynamic reconfigurations or new client connection notifications. These stored events can then be reviewed by using the session replay operation, which allows all connected clients to receive all previously generated data;
- **Richer aggregation functions:** By using *Esper*, it is possible to use richer function constructions, which can include operations like calculating averages on time periods. Moreover, these aggregations can be linked with the previously mentioned dynamic reconfigurations, since each dynamic reconfiguration rule can also modify the aggregation function whenever is triggered;
- **Ubiquitous clients:** The *Android* based mobile client developed in this thesis showcases the middleware's functionalities by implementing the service API in a mobile device. This client can be used to connect clients to sessions and visualize the received session data. Clients can also connect to the middleware by using the Web administration interface;

6.3 Future Work

There are several extensions and improvements of this thesis work that could be addressed in future work, as described in the following.

- **Instance parellization:** Optimizing the middleware's Cloud integration to allow parallelization of multiple middleware instances, hence improving overall performance. This would allow the middleware to take advantage of the virtual machine instance auto scaling service, provided by *Amazon Web Services*. By using these service it would be possible to define rules based on Cloud service metrics (e.g. CPU usage, number of disk accesses). These rules would allow to automatically add virtual machine instances to face high processing peaks, or remove existing instances in periods of low usage, so that the underlying costs could be minimized;
- **Improving repository access:** The repository access could be improved, both in functionality and performance. Some of these repository improvements possibilities are described in the following.

- Extend the existing implementation so that it is possible to access repository data, simultaneously with the live data. Clients would then be able to filter or aggregate historical data, together with current data (e.g. verify if the current temperature is greater than the average temperature, from the session's beginning);
- Perform experiments using repository specific, Cloud-based *NoSQL* instances. Given the nature of the repository data (e.g. one single relation), the usage of this kind of non-relational key-value databases could result in improved performance. *Amazon DynamoDB* could be used for this purpose;
- **Elastic sessions:** The aforementioned performance improvements would allow the middleware to support a large number of data sources, producing large amounts of data, as well as support a large number of clients. Nonetheless, data delivery should be optimized whenever possible, e.g. if data is to be disseminated to a set of session clients using the same interaction model, and with the same parameterization (e.g. the same data rate). These extensions are to be investigated in the context of a new MSc which is starting;
- **Simplified user interface:** Provide a simpler interface for performing middleware operations, such as complex rule definition. For instance, clients could be presented with a wizard, or simplified *GUI* that offers visual guidelines for creating *Esper* expressions intuitively (e.g. aggregate all received events from different data sources at the same interval in time);
- **Richer dynamic reconfigurations:** As described before, the existing dynamic reconfiguration rules can be based on the data generated by data sources, or based on other dynamic reconfigurations with a specific label. The rule set could be enriched by allowing to create rules based on other factors, such as the Cloud service metrics. This would allow users to inspect the service state, and define dynamic reconfigurations based on metrics such as CPU usage, data usage, or service usage associated costs;
- **More protocol connectors:** By developing and integrating new connectors based on *Apache Camel*, the middleware would broaden the possible data sources that could be integrated in Session. Examples of useful connectors would be a *WebServices connector*, or a *Java Messaging Services (JMS) connector*;

Bibliography

- [ADF⁺03] Daniel E. Atkins, Kelvin K. Droegemeier, Stuart I. Feldman, Hector Garcia-Molina, Michael L. Klein, David G. Messerschmitt, Paul Messina, Jeremiah P. Ostriker, and Margaret H. Wright. Revolutionizing Science and Engineering Through Cyberinfrastructure: Report of the National Science Foundation Blue-Ribbon Advisory Panel on Cyberinfrastructure, 2003.
- [AFG⁺09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.
- [AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.
- [Ama] Amazon. Amazon cloud player. <http://www.amazon.com/b?ie=UTF8&node=2658409011>.
- [Appa] Apple. Apple icloud. <https://www.icloud.com/>.
- [Appb] Apple. ios technology overview. <http://developer.apple.com/library/ios/>.
- [AWS] Amazon Web Services AWS. Amazon web services - aws). <http://aws.amazon.com/>.

- [BGP12] Adérito Baptista, M. Cecilia Gomes, and Hervé Paulino. Session-based dynamic interaction models for stateful web services. In *Exploring Services Science - Third International Conference, IESS 2012, Geneva, Switzerland, February 15-17, 2012*, Lecture Notes in Business Information Processing. Springer-Verlag, 2012.
- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: a system of patterns*, volume 1. John Wiley and Sons, 1996.
- [BYV08] Rajkumar Buyya, Chee S. Yeo, and Srikumar Venugopal. Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities. In *HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 5–13, Washington, DC, USA, September 2008. IEEE Computer Society.
- [Cam] Apache Camel. Apache camel. <http://camel.apache.org>.
- [CC66] Gianpaolo Carraro and Fred Chong. Software as a service (saas): An enterprise perspective. Technical report, Microsoft, 2066.
- [CFK⁺00] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of network and computer applications*, 23(3):187–200, 2000.
- [cI11] comScore (Information Technology Analytics). U.s. mobile subscriber market share. http://www.comscore.com/Press_Events/Press_Releases/2011/8/comScore_Reports_July_2011_U.S._Mobile_Subscriber_Market_Share, 2011.
- [com11] comScore. Market share analysis: Mobile devices, worldwide, 1q11. <http://www.gartner.com/it/page.jsp?id=1689814>, 2011.
- [CRPN08] Manuel Caeiro-Rodriguez, Thierry Priol, and Zsolt Németh. Dynamicity in scientific workflows. Technical Report TR-0162,

- Institute on Grid Information, Resource and Workflow Monitoring Services , CoreGRID - Network of Excellence, August 2008.
- [Dar05] Frederica Darema. Dynamic data driven applications systems: New capabilities for application simulations and measurements. In *International Conference on Computational Science (2)*, pages 610–615, 2005.
- [Dar10] Frederica Darema. Cyberinfrastructures of cyber-applications-systems. *Procedia CS*, 1(1):1287–1296, 2010.
- [Esp] Esper. Esper: Complex event processing. <http://esper.codehaus.org/>.
- [FDFB12] E. Ferrara, P. De Meo, G. Fiumara, and R. Baumgartner. Web Data Extraction, Applications and Techniques: A Survey. *ArXiv e-prints*, July 2012.
- [Fos06] Ian Foster. Service-oriented science: Scaling escience impact. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence, WI '06*, pages 9–10, Washington, DC, USA, 2006. IEEE Computer Society.
- [GAE] Google App Engine GAE. Google app engine - gae. <http://code.google.com/appengine/>.
- [Gao12] Dehong Gao. Opinion influence and diffusion in social network. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval, SIGIR '12*, pages 997–997, New York, NY, USA, 2012. ACM.
- [GCR05] M. Cecília Gomes, Jose C. Cunha, and Omer Rana. *A Pattern-based Software Engineering Tool for Grid Environments*, pages 213–222. NATO Science Series III: Computer and Systems Sciences. IOS PRESS, 05 2005.
- [GHIGGHPD07] Carlos F. García-Hernández, Pablo H. Ibarguengoytia-González, Joaquín García-Hernández, and Jesús A. Pérez-Díaz. Wireless sensor networks and applications: a survey. *International Journal of Computer Science and Network Security*, 17(3):264–273, 2007.

- [GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [GHS95] Diimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. In *DISTRIBUTED AND PARALLEL DATABASES*, pages 119–153, 1995.
- [Gooa] Google. Android developers. <http://developer.android.com>.
- [Goob] Google. Google apps. <http://www.google.com/apps/intl/en/business/index.html>.
- [Gooc] Google. Google reader. <http://www.google.com/reader>.
- [GPBdSA12] M. Cecília Gomes, Hervé Paulino, Adérito Baptista, and Filipe Jorge da Silva Araújo. Accessing wireless sensor networks via dynamically reconfigurable interaction models. *International Journal of Interactive Multimedia and Artificial Intelligence*, 1(7):52–61, 12 2012.
- [GRC03] M. Cecilia Gomes, O. Rana, and Jose C. Cunha. Pattern operators for grid environments. *Scientific Programming*, 11(3):237–261, 2003.
- [GRC08] M. Cecilia Gomes, O. Rana, and Jose C. Cunha. Extending grid-based workflow tools with patterns/operators. *IJHPCA - The International Journal of High Performance Computing Applications*, 22(3):301–318, 08 2008.
- [Gro10] Cloud Computing Use Case Group. Cloud Computing Use Cases Whitepaper, 2010.
- [Hat] Red Hat. Red hat openshift. <https://openshift.redhat.com/app/>.
- [HBD09] A. Hornsby, P. Belimpasakis, and I. Defee. Xmpp-based wireless sensor network and its integration into the extended home environment. In *Consumer Electronics, 2009. ISCE '09. IEEE 13th International Symposium on*, pages 794–797, 2009.

- [HKLP05] Hal Hildebrand, Anish Karmarkar, Mark Little, and Greg Pavlik. Session modeling for web services. In *Third European Conference on Web Services (ECOWS 2005), 14-16 November 2005, Växjö, Sweden*. IEEE Computer Society, 2005.
- [Hog11] Liu F. Sokol A. W. Jin T. Hogan, M. D. Nist-sp 500-291, nist cloud computing standards roadmap. Technical report, NIST, National Institute of Standards and Technology, U.S. Department of Commerce, Aug 2011.
- [HW03] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [IBM96] Charles A. Doswell III, Harold E. Brooks, and Robert A. Maddox. Flash flood forecasting: An ingredients-based methodology. *American Meterological Society*, 11(2), 12 1996.
- [IET11] IETF. The websocket protocol. <http://tools.ietf.org/html/rfc6455>, 2011.
- [JMB11] Brad Brown Jacques Bughin Richard Dobbs Charles Roxburgh James Manyika, Michael Chui and A.H. Byers. Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute, 2011.
- [JNL10] Keith J. and Burkhard Neidecker-Lutz, editors. *The Future Of Cloud Computing, Opportunities for European Cloud Computing Beyond 2010*. EUROPA > CORDIS > FP7, January 2010.
- [KBLK07] Tomasz Kobialka, Rajkumar Buyya, Christopher Leckie, and Ramamohanarao Kotagiri. A sensor web middleware with stateful services for heterogeneous sensor networks. In *Intelligent Sensors, Sensor Networks and Information, 2007. ISSNIP 2007. 3rd International Conference*, pages 491–496, 2007.
- [KR12] JosephA. Konstan and John Riedl. Recommender systems: from algorithms to user experience. *User Modeling and User-Adapted Interaction*, 22:101–123, 2012.

- [LBC10] N. Laga, E. Bertin, and N. Crespi. Composition at the frontend: The user centric approach. In *Intelligence in Next Generation Networks (ICIN), 2010 14th International Conference on*, pages 1–6, 2010.
- [LML⁺10] Nicholas D. Lane, Emiliano Miluzzo, Hong Lu, Daniel Peebles, Tanzeem Choudhury, and Andrew T. Campbell. A survey of mobile phone sensing. *Comm. Mag.*, 48(9):140–150, September 2010.
- [Lou10] Panagiotis Louridas. Up in the air: Moving your applications to the cloud. *IEEE Software*, 27(4):6–11, 2010.
- [MP95] Nelson R. Manohar and Atul Prakash. The session capture and replay paradigm for asynchronous collaboration. In *In Proc. of European Conference on Computer Supported Cooperative Work (ECSCW)'95*, pages 149–164. Kluwer Academic Publishers, 1995.
- [Rac] Rackspace. Rackspace cloud servers. <http://www.rackspace.co.uk/>.
- [Sal] Salesforce. Salesforce crm. <http://www.salesforce.com/crm/>.
- [Ser] Amazon Web Services. Amazon elastic compute cloud (EC2). <http://aws.amazon.com/ec2/>.
- [Spr] Spring. Spring integration. <http://www.springsource.org/spring-integration>.
- [SRS⁺09] Hans-Gunther Schmidt, Karsten Raddatz, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. Google android - a comprehensive introduction. Technical Report TUB-DAI 03/09-01, Technische Universitat Berlin - DAI-Labor, 2009. <http://www.dai-labor.de/fileadmin/files/publications/GoogleAndroid.pdf>.
- [SWJ08] Christoph Stasch, Alexander C. Walkowski, and Simon Jirka. A geosensor network architecture for disaster management based on open standards. In *Digital Earth Summit on Geoinformatics 2008: Tools for Climate Change Research.*, pages 54–59, 2008.

- [Sys] South Florida Water Management System. Natural system model. <http://www.sfwmd.gov/portal/page/portal/xweb%20-%20release%202/natural%20system%20model>.
- [TBB03] Mark Turner, David Budgen, and Pearl Brereton. Turning software into a service. *IEEE Computer*, 36(10):38–44, 2003.
- [vdAHW03] Wil van der Aalst, Arthur Ter Hofstede, and Mathias Weske. Business process management: A survey. In *Proceedings of the 1st International Conference on Business Process Management, volume 2678 of LNCS*, pages 1–12. Springer-Verlag, 2003.
- [YB05] Jia Yu and Rajkumar Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34:44–49, September 2005.