Adérito Herculano Sarmento Baptista

Licenciatura em Engenharia Informática

Dynamic Adaptation of Interaction Models for Stateful Web Services

Dissertação para obtenção do Grau de Mestre em Engenharia Informática

Orientadora: Prof[a]. Doutora Maria Cecília Gomes, D.I da FCT/UNL

Co-orientador: Prof. Doutor Hervé Miguel Cordeiro Paulino, D.I da FCT/UNL

Júri:

Presidente: Prof. Doutor João Moura Pires, D.I da FCT/UNL

Arguentes: Prof. Doutor Francisco Martins, D.I. da FC/UL

Vogais: Prof. Doutor Francisco Martins, D.I. da FC/UL
Prof[a]. Doutora Maria Cecília Gomes, D.I da FCT/UNL

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2011

Dynamic Adaptation of Interaction Models for Stateful Web Services

Dedico o significado da minha tese aos meus pais, que incondicionalmente, sempre me apoiaram, ainda que o conflito entre diferentes gerações tenha originado, por vezes, desentendimentos.
Obrigado por tudo, pai Cinto e mãe Dinha.

Acknowledgements

Agradeço aos meus orientadores Maria Cecília Gomes e Hervé Paulino pela dedicação, esforço, preocupação e paciência revelados ao longo do período que delimitou a realização desta tese. Não posso deixar de mencionar as repreensões, por vezes algo agressivas (e com razão), feitas sobre o atraso no desenvolvimento da mesma. Apenas os amigos são capazes de fazer tais chamadas de atenção, sob pena de gerar ressentimentos, precisamente porque desejam o nosso sucesso.

Aos meus colegas de curso e amigos, pelo companheirismo demonstrado ao longo de mais uma etapa. Em especial, ao Ricardo Neves, João Bolinhas, João Milheiriço, Daniel Pina e à família Bonga, que proporcionaram momentos hilariantes, do qual me recordo e guardo com nostalgia.

Igualmente aos meus pais, às minhas irmãs e sobrinhos, pelo apoio e motivação. Sempre foram e continuam a representar o pilar que me suporta e guia ao longo do meu percurso.

Este trabalho foi parcialmente financiado pelo Centro de Informática e Tecnologias da Informação (CITI), e pela Fundação para a Ciência e a Tecnologia (FCT / MCTES) em projectos de investigação.

Abstract

Wireless Sensor Networks (WSNs) are accepted as one of the fundamental technologies for current and future science in all domains, where WSNs formed from either static or mobile sensor devices allow a low cost high-resolution sensing of the environment. Such opens the possibility of developing new kinds of crucial applications or providing more accurate data to more traditional ones. For instance, examples may range from large-scale WSNs deployed on oceans contributing to weather prediction simulations; to high number of diverse Sensor devices deployed over a geographical area at different heights from the ground for collecting more accurate data for cyclic wildfire spread simulations; or to networks of mobile phone devices contributing to urban traffic management via Participatory Sensing applications.

In order to simplify data access, network parameterisation, and WSNs aggregation, WSNs have been integrated in Web environments, namely through high level standard interfaces like Web services. However, the typical interface access usually supports a restricted number of interaction models and the available mechanisms for their run-time adaptation are still scarce. Nevertheless, applications demand a richer and more flexible control on interface accesses – e.g. such accesses may depend on contextual information and, consequently, may evolve in time.

Additionally, Web services have become increasingly popular in the latest years, and their usage led to the need of aggregating and coordinating them and also to represent state in between Web services invocations. Current standard composition languages for Web services (wsbpel,wsci,bpml) deal with the traditional forms of service aggregation and coordination, while WS-Resource framework (wsrf) deals with accessing services pertaining state concerns (relating both executing applications and the runtime environment).

Subjacent to the notion of service coordination is the need to capture dependencies among them (through the workflow concept, for instance), reuse common interaction

models, e.g. embodied in common behavioural Patterns like Client/Server, Publish/-Subscriber, Stream, and respond to dynamic events in the system (novel user requests, service failures, etc.). Dynamic adaptation, in particular, is a pressing requirement for current service-based systems due to the increasing trend on XaaS ("everything as a service") which promises to reduce costs on application development and infrastructure support, as is already apparent in the Cloud computing domain.

Therefore, the self-adaptive (or dynamic/adaptive) systems present themselves as a solution to the above concerns. However, since they comprise a vast area, this thesis only focus on self-adaptive software. Concretely, we propose a novel model for dynamic interactions, in particular with Stateful Web Services, i.e. services interfacing continued activities. The solution consists on a middleware prototype based on pattern abstractions which may be able to provide (novel) richer interaction models and a few structured dynamic adaptation mechanisms, which are captured in the context of a "Session" abstraction.

The middleware was implemented and uses a pre-existent framework supporting Web enabled access to WSNs, and some evaluation scenarios were tested in this setting. Namely, this area was chosen as the application domain that contextualizes this work as it contributes to the development of increasingly important applications needing high-resolution and low cost sensing of environment. The result is a novel way to specify richer and dynamic modes of accessing and acquiring data generated by WSNs.

Keywords: WSNs, Coordination, Pattern-based Interaction Models, Self-adaptable Software.

Resumo

As Redes de Sensores sem fios (RSSF) são consideradas parte integrante das tecnologias fundamentais da ciência actual e futura em diversos domínios, sendo estas formadas por dispositivos móveis estáticos ou móveis, permitindo uma monitorização ambiente de baixo custo e alta-resolução. Tal possibilita o desenvolvimento de novos tipos de aplicações cruciais, ou a possibilidade de fornecer dados mais precisos a aplicações tradicionais. Tais exemplos constam, RSSF de grande escala residentes em oceanos tendo em vista a simulação de previsões meteorológicas, a instalação de diversos sensores sobre uma área geográfica dispersa, a diferentes pontos de altitude, de modo a coleccionar dados mais exactos para simulações de propagação de incêndios, ou ainda a utilização de redes de dispositivos móveis na gestão de tráfego urbano.

De modo a simplificar o acesso aos seus dados, parametrização de rede e agregação de RSSF, estas têm vindo a ser integradas em ambientes *Web*, nomeadamente através de interfaces de alto nível como os *Web Services*. Contudo, uma interface de acesso típica suporta somente uma quantidade limitada de modelos de interacção, e os mecanismos existentes para a sua adaptação em tempo de execução, são ainda parcos ou limitados. Não obstante, as aplicações tendem a requerer cada vez mais flexibilidade no controlo de acesso às interfaces; por exemplo, tais acessos podem depender de informação contextual e, consequentemente, podem evoluir ao longo do tempo.

Adicionalmente, a utilização de *Web Services* tem se tornado cada vez mais popular ao longo dos últimos anos, e da sua utilização adveio a necessidade de agregá-los e coordená-los devidamente, bem como de representar as transições de estado entre as suas invocações. As actuais linguagens *standard* de composição de *Web Services* (wsbpel,wsci,bpml) lidam com as formas tradicionais de agregação e coordenação de serviços, enquanto a WS-Resource Framework (wsrf) lida com o acesso a serviços que englobam preocupações de transições de estados.

Subjacente à noção de coordenação de serviços está a necessidade de encontrar dependências entre estes (através do fluxo de conceitos, por exemplo), de reutilizar modelos de

interacção comuns, (como por exemplo, os contidos em padrões comportamentais como Cliente/Servidor, Publish/Subscribe ou Stream), e de responder a eventos dinâmicos no sistema (novos pedidos do utilizador, falhas de sistema, etc). A adaptação dinâmica, em particular, é um requisito premente para os sistemas baseados em serviços, dada a crescente utilização de XaaS ("tudo como serviço"), que visa a redução de custos no desenvolvimento de aplicações e manutenção de infra-estruturas, conforme actualmente comprovado no domínio de *Cloud computing*.

Assim sendo, os sistemas auto-adaptáveis (ou dinâmicos/adaptáveis) apresentam-se como uma solução para as preocupações descritas acima. Contudo, eles compreendem uma área bastante mais vasta do que o âmbito desta tese, que se foca particularmente em software auto-adaptável. Concretamente, é proposto um novo modelo para interacções dinâmicas, em particular com *Web Services* com estado, ou seja, serviços que englobam actividades contínuas. Esta solução consiste num protótipo de middleware baseado em abstracções de padrões que fornecem (novos) modelos de interacção e alguns mecanismos estruturados de adaptação dinâmico, que são detectados no contexto de uma "sessão" de abstracção.

Este *middleware* foi implementado utilizando uma *framework* existente de suporte a acesso web a RSSF, e alguns cenários de avaliação foram testados na mesma. Nomeadamente, esta área foi escolhida como o domínio de aplicação que contextualiza este trabalho, que contribui para o desenvolvimento de aplicações relevantes que necessitem de monitorização do ambiente de alta-resolução e baixo custo. O resultado final é uma nova forma de especificar meios mais ricos de aceder e obter dados gerados por RSSF, de uma forma mais dinâmica.

Palavras-chave: RSSFs, Cordenação, Modelos de Interação baseados em Padrões, Software Auto-Adaptável.

Contents

List of Figures

Listings

1

Introduction

This chapter introduces the motivation behind the elaboration of this thesis, namely the dynamic adaptation of interaction models for a specific context - wireless sensor networks (WSNs). The areas concerning this topic will be outlined, as well as, the specific problem we aim to solve, including the proposed solution. Finally we point the aimed contributions of this thesis and the rest of the document outline.

1.1 Overview and Motivation

Service abstraction emphasizes the need to hide as much of the underlying details of a service as possible. It is this principle that allows us to establish services as black boxes, hiding their underlying details from potential customers. Currently, a common and popular approach for service oriented paradigm are Web services. The IT industry, for instance, is the typical area of success where Web services have increasingly been used.

However, the usual single request/response kind of interactions, characteristic of Web services in their primitive form, are not able to cope with the need to provide more complex interactions between the service provider and consumer. One example is the execution of long running HPC (high performance computing) applications, like it was first provided in Grid Computing [FKNT02]. Typical queries would include requests from users about the current running state - is it still running? Has it been suspended or even aborted? These kind of operations are possible if current state is associated with running status, which is translated to the requirement of associating State with Web services.

Such possibility of combining statefulness with Web services, also contributed to the emergence of (business) areas such as Cloud computing [Hay08, FZRL09]. Moreover, the already real-prevalence of ubiquitous computing and the need for interaction with

the real world also justifies selecting stateful Web services as the interaction mechanism. For example, sensor networks, whose importance for environment monitoring is rising, and due to their characteristics are being increasingly interfaced via high-level stateful service-based interfaces [BPRD07, Bac07, PT09]. Another current requisite resulting from today's environment is the necessity to compose different singular Web services into a richer one. For instance, consider an application that provides a booking service comprising flight and hotel reservations. This service, which could perfectly be provided via Web service, would have to combine other two existing different Web services: the first to book a flight and the second for the hotel reservation placed in the destination city. It should also consider that, it may be only in the interest of the client to have both reservations confirmed, and not just one of them.

This example shows that not only would be necessary to compose two services in one, but also to coordinate them (i.e. the booking service may only confirm the reservation to the client, only if both (flight and hotel) were successfully booked. This brings another topic into play, which is coordination. Forms of coordination include not only composition, but also patterns and workflows. The combination of both will helps us to achieve our desired goal - Dynamic Adaptation. Figure 1.1 illustrates how the previous disciplines are related.

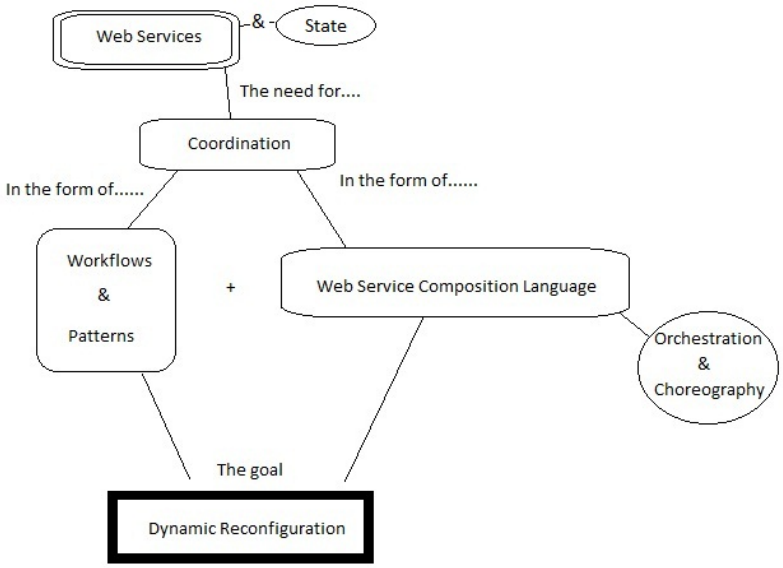


Figure 1.1: Web services, State and Coordination

Considering the current trend on XaaS (*everything as a service*) in the domain of (Web) services in general, users are increasingly demanding dependability garantees from service providers. Service consumers already have lower tolerance on (Web) service unavailabilty or QoS degradation, and on lack of relialibity, consistency or security guarantees. Services do have to become really scalable and flexible enough to provide the contracted QoS. Otherwise, such incapabilities have a negative impact for the enterprises being fundamental for they survival in business.

The problem increases if a (Web) service depends on other (Web) services or if service coordinated aggregation is in order. Such dependencies among distributed services are usually captured in the form of *workflows* for which it is necessary to support adaptation mechanisms. This makes the area of dynamic workflows highly important nowadays, where issues such as dynamic binding to new services have been subject of study by several works [PT09]. The aim is that, the systems, being self-aware of their state, are fault-tolerant and can respond to new user requirements, cope with failure situations, or incorporate new resources without being stopped. Such is included in the area of Self-adaptable Software aiming at building more resilient and perduring but evolvable systems.

For stateful Web services, in particular, it is necessary to guarantee not only consistency in terms of interfacing the service, but also concerning the state of a Web service itself or of a set of interacting stateful Web services. The pressure for fulfilling contracted QoS, as well as the additional interaction models that may be possible in the context of stateful Web services, require new adaptation mechanisms not considered in traditional Web services. For example, it is necessary to guarantee the QoS concerns associated with a stream being received by a client, important notifications have to be effectively delivered and on time, etc. Traditional and novel adaptation approaches like data compression or reducing data quality (e.g. lower resolution), caching, load balancing and replicated servers, processing on transit, etc, are being used to meet such constraints [ST09, BPK+06]. Moreover, it is also necessary to guarantee additional security/authentication levels since long running give malicious users more time to attempt attacks.

Additionally, the dynamic adaptation processes for stateful Web services may depend of, or benefit from, the state of the services themselves or even of the clients (e.g. the server may have a higher throughput capacity than the client). Therefore, the existence of several (and more complex) interaction models may be used as a way to implement adaptation, in order to select the most adequate ones according to the current conditions of either or both the service and the client.

For sure there are many open problems related with the dynamic/self-adaptation of stateful Web services, but this thesis will focus only on adaptation issues related with interaction models among one (or more) service(s) and its (their) users. In particular, the sensor networks area has been chosen as the application domain due to its characteristics and relevance.

Namely, the wireless sensor networks (WSNs) area has been an important growing research field for the last years, as it enables the control and monitorization of the physical world. However, as big as the motivation for its usage may be, there are also challenges that still remain open topics of research. One of them deals with the integration of WSNs in Web environment, which becomes easier [San09, BPRD07], with the help of currently main Web technologies, including Web services. Figure 1.2 shows how such Web technologies may help to overcome integration limitations. Section 2.3 gives detailed information about WSNs.

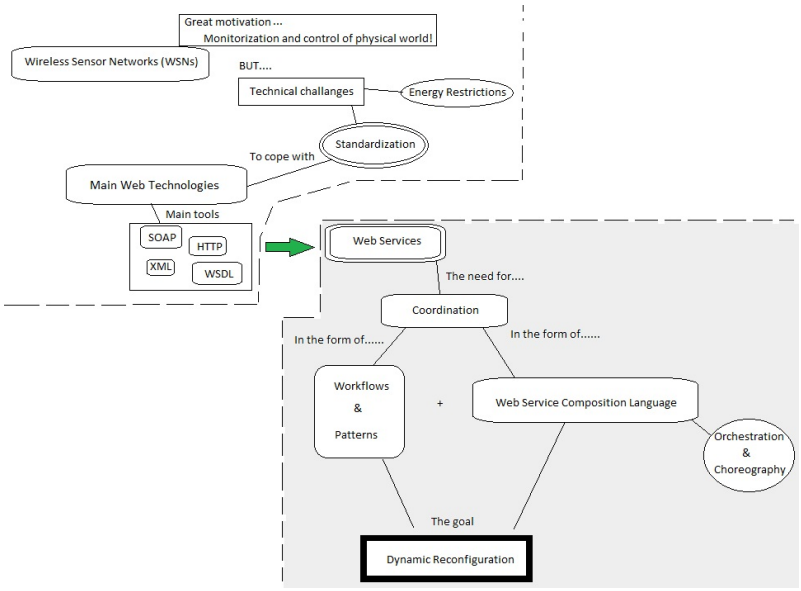


Figure 1.2: Integration of wireless sensor network in Web context

1.1.1 Illustrative example

Consider the following scenario:

Imagine a wireless sensor network (WSN) capturing the temperature of a forest. The WSN is attached to a sink node which represents and identifies the WSN. In turn, the sink node is registered in an application, with a middleware layer that offers operations like streaming of data and notification services.

An application assisting the actions of the fire department for this area subscribes the service to receive a notification should the temperature raise to precaution levels. Periodically, the application may consult the weather information provided by a legitimate source through a Web service. This may be used to keep the fire department personal informed of present and predicted conditions in the area, as well as network parametrisation if such is supported by the cited middleware layer. For instance, in case rain is previsible during the night, the sensors may be instructed to capture the temperature over shorter periods and between larger intervals of time. If the temperature is to rise during the afternoon of the following day, those periods may be defined to become larger. Additionally, if the temperature raises to dangerous level, a notification service may be triggered and a warning is hence sent.

Considering that a fire situation does occur, the application should automatically provide the streaming of the captured temperature to the fire department for its evaluation. If more precise data values are needed, the application should be instructed to acquire and process data from temperature sensors at different heights from the ground making the result readily available. Additionally, it may be necessary to capture data concerning the direction and speed of the wind in the same and adjacent areas, giving additional information about the probable fire propagation direction and speed. It would also be

helpful to (automatically) extend the monitoring of the temperature to a larger area of the forest giving real-time status of the overall fire propagation. Finally, it would be convenient to provide the previous collected data to any additional fire department chosen to help in the fire fighting.

The scenario describes a simple case of a self–adaptive system involving the mentioned disciplines:

- *Web enabled WSN* access, parameterisation and data collection.
- Interaction models in the form of typical *Coordination/Behavioural patterns*, e.g. *Publish/Subscriber* for temperature notification, and *Streaming* for online (almost real-time) temperature and wind data flows.
- Dynamic reconfiguration of the system either automatically or upon user request in case of a fire ignition scenario:
  - The automatic adaptation to the *streaming pattern* to provide online temperature information.
  - Data acquision from other types of WSNs, in this case, capable of capturing wind characteristis, and temperature data acquisition from WSNs in the adjacent locations as a way to monitor a wider area.
  - Dynamically modify the interaction model by switching from a Publish/Subscribe to Streaming, or dynamic aggregation of distinct data stream.

To implement a system like this, the concerns an challenges pertaining diverse areas have to be considered.

Next section will specify the problem and working goals of this thesis.

1.2 Problem Statement and Work Goals

The general problem that this thesis aims to solve is the dynamic adaptation of interaction models for Stateful Web services, with particular application to Web enabled Sensor Networks. This bigger problem includes a few smaller ones that have to be considered in the context of this work, namely:

- Identification of additional interaction models to be considered in the context of Stateful Web Services at large, and Web enabled WSNs in particular.
- Identification of the possible self-adaptive mechanisms to use for dynamically reconfigure within or between diverse interaction models.
- Identification of the possible dynamic reconfiguration scenarios which may be relevant in this context while guaranteeing the consistency of the system.

- Evaluation of application scenarios in the domain of Web enabled WSNs in order to inspect how they may benefit from the above (new) interaction models and adaptation mechanisms for stateful Web services.

To the extent of our knowledge, and answering the previous questions, we consider that:

1. The existent interaction models for stateful Web services are limited, particularly in the client's perspective. For example, there is no interface able to provide high-level, but structured functionalities/interaction models to a user, like aggregating the results from different yet similar interfaces as the result of a single request (e.g. stream aggregation from several services with parameteerisable aggregation function).

2. The existing dynamic adaptation mechanisms are still scarce in this context, and do not take advantage of (new) interaction models nor consider the state of the service and the client. Interesting adaptations in this context may be to switch the interaction model between client and server depending on the state/type of the client and/or the server. For instance, when a client is to receive a large amount of data from a server, and the processing capability of the former is inferior from the latter, the response interaction model should be a producer/consumer instead of streaming, to reduce data loss (and in case real time is not a major concern, implying some form of data caching).

3. Dynamic/self-adaptation issues for stateful (Web) services are applicable and needed for several relevant emergent domains, namely those related with any form of continued activity requiring state knowledge. This is the case of Cloud computing, or domains related with interactions in real world scenarios such as ambient intelligence, participatory sensing and sensor networks.

   Specifically, Web enabled WSNs were selected as the application area, not only because it is a good application domain, but also because it is a novel area with many open problems which may benefit from richer interaction models and (novel) dynamic/self-adaptation mechanisms.

4. In general, in novel domains as the ones mentioned above, it is always good practice to reuse accumulated knowledge which includes state-of-the-art dynamic/self-adaptation mechanisms to promote an agile development of applications and systems. Therefore, such knowledge is best transmited and propragated if well defined forms of reusability employed. This is the case of the pattern concept which is employed as a structuring mechanism defining the scope of the possible dynamic reconfiguration actions.

To sum up, the goal of this work is to provide richer forms of structured interaction models to Stateful Web Services which show context-based dynamic adaptation capabilites, and their validation in the context of Web-enabled WSNs.

This work relies on the pattern concept as a way to specify such structured interaction models and coherent dynamic adaptation actions both within and between them. Interesting patterns in this context include behavioural patterns capturing dependencies among one (or more) stateful (Web) service(s) and its (their) clients. Examples are Streaming, Client/Server, Producer/Consumer, or Publish/Subscriber (see section 2.2.3)

1.3 Contributions of this Thesis

To this extent the contributions of this thesis are:

1. Usage of the pattern abstraction as a mean to represent interactions between clients and stateful Web services.

2. Pattern-based dynamic reconfiguration supporting changes within each interaction model like dynamically increase the number of consumer clients to a Producer Service.

3. System dynamic evolution as a result of context-based dynamic replacement between behavioural/coordination patterns.

4. Usage of the session concept to capture the context of one particular service interfaced through an interaction model, and all its current users, as well as the set of possible dynamic reconfigurations in this context.

5. Specification of the possible evolutions for the system through state-machines.

6. Definition of the system's architecture and middleware prototype implementation, incorporating:

   (a) The implementation of richer interaction models for Web enabled WSNs.

   (b) The implementation of some pattern-based dynamic adaptation mechanisms in this domain.

1.4 Document Outline

For the rest of this document, the remaining sections are organized as follows:

- Chapter 2 introduces the service oriented architecture and study the technical details concerning Web services. It references the problematic issues involving coordination, such as the importance of statefulness in Web services, current popular languages for Web services composition, forms of coordination and reconfiguration

based of patterns operators. Finally gives deeper information about wireless sensor networks and mention some technical challenges that this area faces;

- Chapter 3 presents the adopted approach, which is based in patterns, their combination and dynamic reconfiguration. It also describes the conceptual view of the system;
- Chapter 4 exposes the architecture of the middleware and all the details related to the implementation. It ends with the explanation of the Java user API that was developed;
- Chapter 5 makes the functional analysis of the middleware to the specific scenario described in 1.1.1
- The thesis is concluded in chapter 6 (Conclusion and Future Work) where are presented features that would be interesting to extend on the middleware. Finally, conclusions are drawn on the design and implementation.

2

State of the art

This chapter presents the state-of-art in each of the disciplines concerning this thesis, including web services, coordination languages for orchestration and choreography of stateful web services, patterns capturing behaviour, dependencies in workflows, dynamic reconfiguration, and wireless sensor networks (WSNs) as the specific application area. Due to the large number of the areas involved, one major goal leading to this chapter was the study of the relevant concepts in each one, in order to understand the characteristics pertinent to the problem and how are these areas related with one another.

Namely, as an illustrative guideline of the relation between the areas in the context of this thesis and of the structure for the next sections, consider the figure 2.1. The numbers in it indicate the order in which the areas will be described in this chapter.

The first section describes the relevance of the service paradigm and gives as example the IT area, where it has been extensively used. Additionally, this section gives insight on technical details about web services technology (subsection 2.1.2) and the concrete architecture behind it (subsection2.1.1). In Figure 2.1, the starting point should be number 1 - Service oriented interface.

Furthermore, as IT systems leverage the advantages of using web services, the next natural step comes with the combination of different web services to compose a richer value result. Currently, there are some standard languages for Web services composition and they deal with two important concepts: orchestration and choreography. These are two forms of coordination, each modeling different aspects of Web services composition. Subsection 2.2.1 gives more details about it. Additionally, another important aspect in Web services comes from the need of keeping states between web services invocations. Consequently, section 2.2.2 references the work done concerning the so called *stateful web services*. These two dimensions (composition and statefulness) are represented in Figure

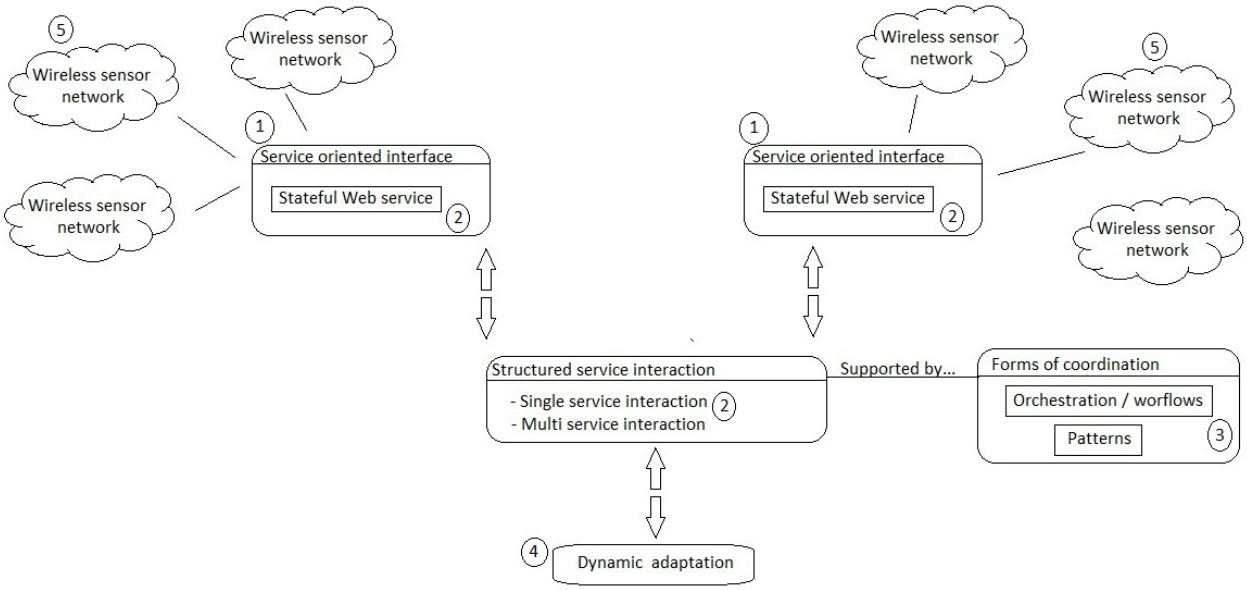


Figure 2.1: Overview

2.1 as number 2, aiming to represent that a *Service oriented interface* may give access, in particular, to a *stateful web service* and that a set of these services can be aggregated in a coordinated way.

In fact, the concept of *coordination* is intrinsic to Web services composition. Coordination is here considered as the way web services should interact in order to achieve the desired goal. Other forms of coordination (besides orchestration and choreography) include *behavioural/coordination patterns* and *workflows*, which are subsequently described in subsection 2.2.3 and are represented in Figure 2.1 by point 3. These forms of coordination are referenced in this work also as a way to support *structured service interaction* models and dynamic adaptation mechanisms. To this extent, patterns and workflows can be used either alone or combined providing the system with some mechanisms to dynamically adapt to its environmental changes - *Dynamic Reconfiguration/Adaptation*, which will be described in subsection 2.2.4 and is represented in the Figure 2.1 by point 4.

Finally, section 2.3 highlights the applicability of the combination of the previous technologies, namely, dynamic adaptation of structured interaction models in the context of stateful web services, to a specific environment: wireless sensor networks (WSN) (Figure 2.1 - point 5). Additionally, some technical challenges in this area are also mentioned as well as the efforts made by the community to overcome them, specifically, in the particular interest of this thesis [San09].

2.1 Service Oriented Architecture (SOA) & Web services

In the next sections, the entities involved in SOA are described, as well as the design principles that it stands for. Next to it, it is also described the Web Services architecture and the concrete technologies involved.

2.1.1 Service Oriented Architecture (SOA)

Service oriented architecture (SOA) is a set of flexible design principles that describe an interaction model between the following parties: a service provider, a service consumer, a service registry, a service contract, a service proxy and a service lease [MTSM03]. Typically, the interactions between these entities happen in the following way:

- The service provider publishes a service description in the service registry. It offers an interface for the service and its implementation. All the information (e.g. location) the consumer needs about the service is obtained and used at runtime. The service interfaces are discovered dynamically, and messages are constructed dynamically. The removal of compile–time dependencies improves maintainability because consumers do not need a new interface binding every time the interface changes.

- The service registry accomplishes the registration of the service. Namely, it is a network-based directory containing the available services and is responsible to map them to their providers. It offers operations of searching, insertion and removal.

- A service consumer wishing to access a service can obtain its description by issuing a query to the service registry to match some criteria. If the service exists, the consumer obtains a service contract that specifies the format of the request and response, its post and preconditions, and the contract may also specify quality of service (QoS) levels. It finally binds to the service over a transport, and executes the service by sending a request formatted according to the contract. This process is known as the "find, bind, execute" paradigm.

- A service proxy is not necessarily required since the consumer could write the necessary software for accessing the service directly (Figure 2.3). However, the proxy may be a convenience entity because it may offer caching and perform some functions locally, avoiding additional hops to the service provider and alleviating some of its work load (Figure 2.2).

- Finally, the service lease specifies the amount of time the contract is valid. Without the notion of a lease, a consumer could bind to a service forever and never rebind to its contract again.

The most important aspect of service-oriented architecture is that it separates the service's implementation from its interface [MTSM03]. In SOA, the user is provided with an abstraction whose pre-conditions must be respected so that the execution can achieve the desirable post-conditions. It is irrelevant how the tasks are processed, as long as the objectives are fulfilled.

The next subsection details the design principles of SOA, which try to cope with all the problematic surrounding the platform independent issues.

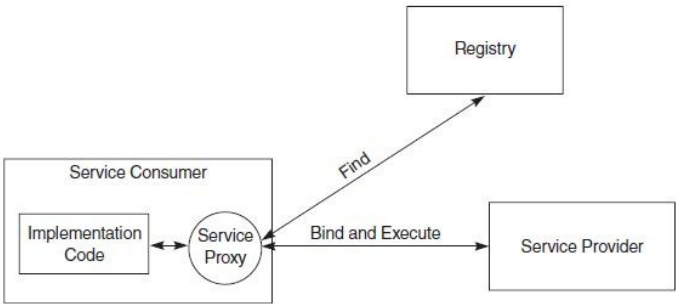


Figure 2.2: "Find, bind, execute" paradigm with proxy [MTSM03]

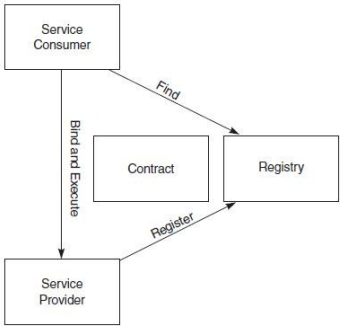


Figure 2.3: "Find, bind, execute" without proxy [MTSM03]

2.1.1.1 Design principles of SOA

- **Discoverable and Dynamically Bound** –"find, bind and execute" paradigm. All information the consumer needs about the service is obtained and used at runtime. The service interfaces are discovered dynamically and messages are constructed as needed. Therefore, consumers do not need a new interface binding every time the interface changes.

- **Self-Contained and Modular**

  Decomposability - also referred as"top-down approach", breaks down bigger problems into smaller modules that solve distinct problems. The goal is to minimize the dependencies between them, in order to be able to reuse the same modules in different problems.

  Composability - states that it should be possible to combine different services and form a more elaborated one.

  Modular understandability - Properly naming services for user readability. E.g. CostumerInfoRequest.

  Modular Continuity - Interfaces should hide internal mechanisms of the service. If not so, changes upon them will cause other applications and user interfaces that benefit from the service having to accompany such changes.

  Modular Protection - Faults in the operation of a service must not impact the operation of a client or other service, nor the state of their internal data, or otherwise

break the contract with service consumers.

Direct Mapping - Map different domain services to distinct problem domain functions. For instance, interfaces that deposit, withdraw, and transfer from a checking account should map to the checking account service.

- **Interoperability** - Service–oriented architecture stresses the ability of systems using different platforms and languages to communicate with each other. SOAP is used as a mediating transport protocol.

- **Coupling** - It refers to the dependencies between modules: loose or tight. The looser the better, since it directly affects the service as more extensible and modifiable.

- **Network-Addressable Interface** - The ability of an application to assemble a set of reusable services on different machines is possible only if the services support a network interface. The network also allows the service to be location–independent, meaning that its physical location is irrelevant.

- **Coarse grained interfaces & Multi-Grained interfaces** - It refers to the aspect of designing interfaces, with methods that return coarse or fine grained information (the service can be implemented by both types). The choice depends of the specific application and the goal is to improve performance by reducing the hops in the network to retrieve the desirable information.

- **Location transparency** - Consumers of a service do not know a service's location until they locate it in the registry. The lookup and dynamic binding to a service at runtime allows the service implementation to move from location to location without the client's knowledge. The ability to move services improves service availability and performance. By employing a load balancer that forwards requests to multiple service instances without the service client's knowledge, it is possible to achieve greater availability and performance.

- **Self –Healing** - With the size and complexity, and longer execution times of modern distributed applications, a system's ability to recover from error is becoming more important. A self-healing system is one that has the ability to recover from errors without human intervention during execution.

  In the next section, it will be clear that many of the concepts for Web services come from the conceptual architecture defined by SOA. For Service–oriented Architectures, Web Services are claimed as state of the art to connect business execution layers as well as networking devices. In fact, Web services achieve some aspects of a true SOA, but also fall short in other areas (e.g. service lease, QoS levels for a service - no official specification provides QoS levels for a service) [MTSM03]. In

addition, service consumers can execute Web services directly if they know the service's address and contract. They do not have to go to the registry to obtain this information (e.g. REST - POST, GET...).

2.1.2 Web services

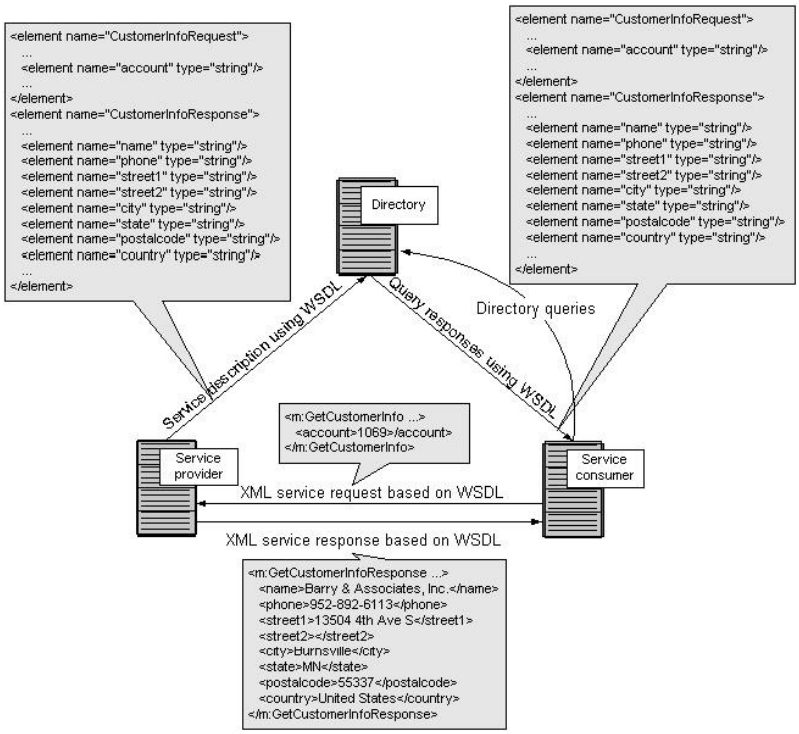


Figure 2.4: WebServices [Bar]

Web services can be viewed as an implementation of SOA. They pretty much follow most of SOA design principles. Figure 2.4 shows Web Services entities and examples of messages used.

Web services follow the "find, bind, execute" paradigm defined in SOA. The interaction between the parties is an instance of the abstract model specified by SOA. In the web services case, the service provider describes its service using the Web Service Description Language (WSDL) [W3Cb] in XML format, and publishes the definition to a directory of services. This directory of services can be a Universal Discovery Description and Integration Registry (UDDI) [OASa]. UDDI is the yellow pages of Web Services. As with traditional yellow pages, the user can search for a company that offers the needed services, read about the offered services and contact someone for more information. A Web Service can be offered without being registered in UDDI, just as someone can open a business in the basement and rely on word–of–mouth advertising. However, to reach a significant market, UDDI is needed in order prospective customers are able to find a service.

Subsequently, a service consumer issues one or more queries to the directory and

receives part of the WSDL earlier published by the provider. The consumer then uses WSDL to send a request to the provider and expects a response from it. All the messages are sent using the Service Object Access Protocol (SOAP) [W3Ca]. SOAP essentially provides the envelope for sending the Web Services messages, and generally uses HTTP, but other means of connection may be used.

Lately, some developers are preferring REST over SOAP. Representational State Transfer (REST) [1] [Fie00] is more of an old philosophy than a new technology, but its realization that came later in technology.

REST is a set of principles that define how Web standards, such as HTTP and URIs, are supposed to be used (which often differs quite a bit from what many people actually do). The promise is that if you adhere to REST principles while designing your application, you will end up with a system that fully exploits the Web's architecture to your benefit.

The key to the REST methodology is to write Web Services using an interface that is already well known and widely used: the URI. It is lighter on bandwidth comparing to SOAP, since SOAP requires an XML wrapper around every request and response. Some may favor REST over SOAP now because those in the agile development camp feel it meets their needs. Namely, REST has been built on the principles of the Web such as HTTP and straight XML, being easy to build without necessarily any toolkit. On the other hand, SOAP proponents argue that strong typing is a necessary feature for distributed applications - e.g. as a way to provide security features.

Neutral developers, however, believe that REST and SOAP do not replace each other. For simple Web Services, developers can use REST. For more complex needs, WS–* services should be used. The WS–* palette of specifications features capabilities in areas such as security and reliable messaging [Kri09].

2.2 Web Service Coordination

As we have come to conclude, a major advantage of the Web Services architecture is that it allows programs written in different languages on different platforms to communicate with each other in a standard base way (using standard Web technologies including HTTP, SOAP, and XML).

The next step is to orchestrate various Web Services to compose a more complex and rich one: a way in which separate Web Services can be brought together in a consistent manner to provide a higher value service. When these services represent related, structured activities or tasks that produce a specific service or product, they are commonly referred to as business processes, and are usually represented as *workflows*.

To elucidate the concept of Orchestration, consider a video-on-demand service (VoD): a customer requests to view a movie at home via a broadband connection to a network

[1] REST was presented in Roy Thomas Fielding's PhD dissertation "Architectural Styles and the Design of Network-based Software Architectures"[Fie00].

operator (NO). The service is operated by a service provider (SP) who has a settlement arrangement with a movie content provider (CP) as well as the NO. The message sequence is shown in the diagram in Figure 2.5:

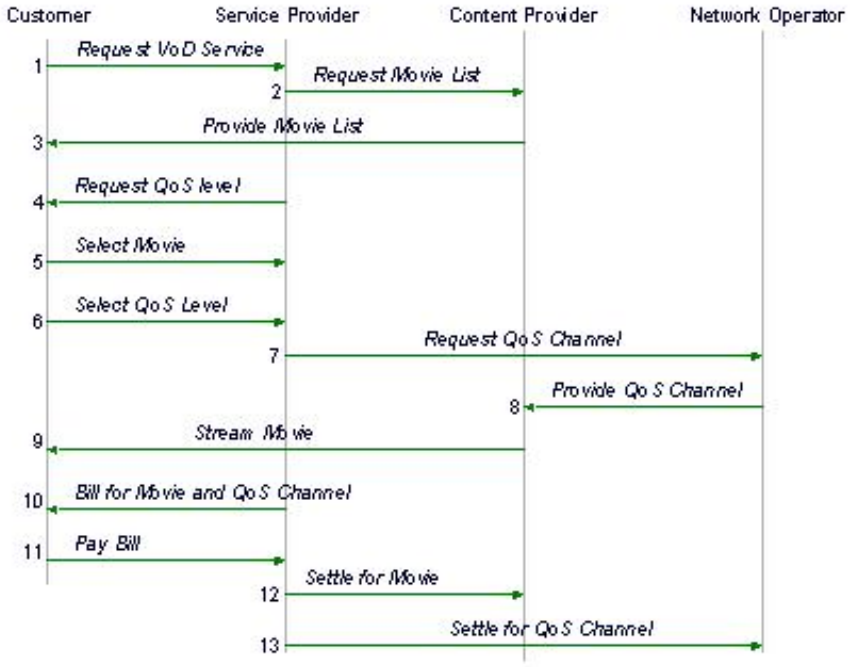


Figure 2.5: VoD [vT]

In this example, and in order for it to succeed, it is essential that the interaction between the Network Operator, the VoD service provider, the content provider, and the end user occurs in a specific way. Many of these interactions can be provided using Web Services in which case there is an immediate requirement for some means with which to tie the overall business process together. This is where Web Services Orchestration comes in. Orchestration in this scenario can be considered to be the means by which the service is automated. Without Orchestration of the interactions between the four actors in the system, they would have to be coordinated manually.

Closely related to the idea of Orchestration is a fundamental concept - Coordination. Coordination is something that intuitively is implicit in many cases of computer science:

Output of a method/function is to be consumed (as input) of other method function - therefore the second function must only execute when the first one has finished; multiple accesses to the same resource –Usage of locks; processes that must occur in parallel/sequential way to meet some goal.

There are many definitions for coordination and the one presented here was selected for its suitability to this context: "The joint efforts of independent communicating actors towards mutually defined goals" [Nat89].

However, forms of coordination include not only orchestration, but also dependencies in Workflows, behaviour/coordination captured in patterns and, ultimately, in the

interest of this thesis –Dynamic Reconfiguration. But before jumping directly into those topics, some currently standard Orchestration languages are described in the next section, due to their relevance on representing coordination in Web Services. Another form of coordination will also be mentioned –Choreography. The term choreography is closely related to orchestration, and as we will see in the next section, both are intimately addressed in Web Services composition.

2.2.1 Web Services Orchestration and Choreography

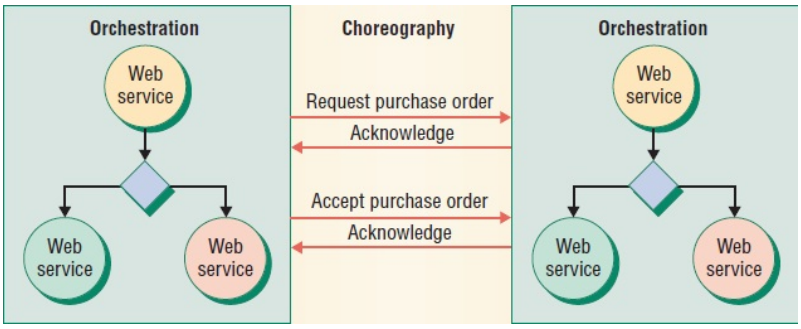


Figure 2.6: Orchestration & Choreography [Pel03]

The terms Orchestration and Choreography describe two similar, but distinct aspects of coordination when creating composite Web Services. The two terms overlap somewhat, but Figure 2.6 illustrates their relationship to each other at a high level. Orchestration refers to an executable business process that can interact with both internal and external Web Services. The interactions occur at the message level. Orchestration always represents control from one party's perspective.

This differs from choreography, which is more collaborative and allows each involved party to describe its part in the interaction. Choreography tracks the message sequences among multiple parties and sources–typically the public message exchanges that occur between Web Services –rather than a specific business process that a single party executes.

Currently, some of the most popular composition languages for Web Services are BPEL, WSCI and BPML:

- **BPEL4WS or BPEL**

  Business Process Execution Language for Web Services (BPEL4WS or BPEL for short) is the successor of Microsoft XLANG and WSFL, that combined with WS–Transaction [OASd] and WS–Coordination[OAS07], form a trio designed to tackle Web Services composition problem. Microsoft, IBM, Siebel Systems, BEA, and SAP coauthored version 1.1 of the BPEL4WS specification, which they released in May 2003.

  BPEL is a workflow–like definition language that allows to describe sophisticated business processes. It provides an XML–based grammar for describing the control

logic required to coordinate Web Services participating in a process flow. WS–Transaction and WS–Coordination complement it, by providing mechanisms to define specific standard protocols to be used by transaction processing systems, workflow systems or other applications that wish to coordinate multiple services [Pel03].

- **WSCI**

  The Web Service Choreography Interface (WSCI) is an XML–based interface description language that describes the flow of messages exchanged by a Web Service interacting with other Web Services. It has been jointly developed by BEA Systems, Intalio, SAP AG, and Sun Microsystems. The specification was issued in May 2002.

  WSCI defines the overall choreography or message exchange between Web Services. The specification supports message correlation, sequencing rules, exception handling, transactions, and dynamic collaboration. As Figure 2.7 shows, WSCI describes only the observable behavior between Web Services. It does not address the definition of executable business processes as BPEL does [Pel03].

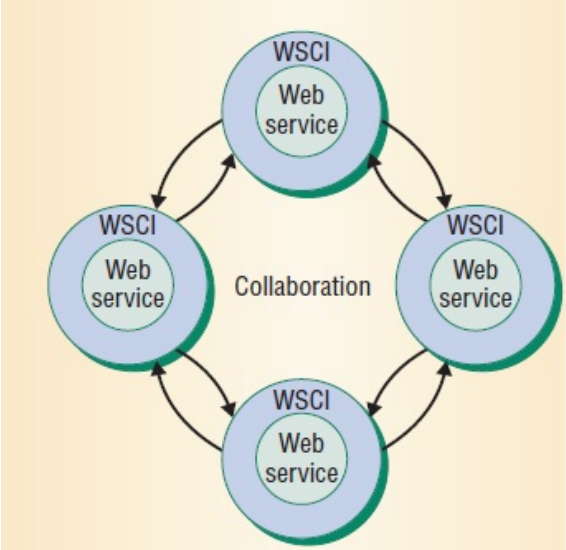


Figure 2.7: WSCI collaboration. The interface specification addresses only the observable behavior between Web Services and not the definition of an executable business process [Pel03].

- **BPML**

  BPML (Business Process Management Language) is a specification from the BPMI.org (Business Process Management Initiative Organization). BPML aims at providing a comprehensive means of specifying the process of an enterprise. BPMI.org and ebXML are addressing complementary aspects of e–business process management. While ebXML Business Process Specification Schema (BPSS) provides a standard way for describing the public interface of an e–business process (choreography), BPML provides a standard way for describing their private implementation (orchestration).

All BPEL, WSCI, and BPML take somehow different approaches to orchestration and choreography. For instance, BPEL provides support for both executable and abstract business processes. An executable process models the behavior of participants in a specific business interaction, and an abstract process or business protocol specifies the public message exchanges between parties.

Essentially, executable processes model orchestration while abstract processes model the choreography of services.

2.2.2 Statefull Web Services

The evolution that IT industry has experienced in the last years, has resulted in new demanding features that Stateless Web Services are not capable to cope with (or at least, not only by themselves). Nowadays, most real–world scenarios involve multiple parties and different organizations that interact with each other throught negotiations, commitments or cancel operations, to name a few. Typically, these kind of interactions are far more complex and longer than the traditional request/reply invocations. Sometimes these processes invoke other long services, or contain steps that may be dependent on external events or even on human interaction.

For example, consider a business process that implements the processing of a purchase order. It is desirable to be able to not only initiate a purchase order, but also modify or cancel a running order in certain situations. These situations usually need to consult the current state. State in this context is basically any piece of information outside the contents of a Web service's request message, that needs to be taken into account, in order to properly formulate the reply.

To deal with these new demands concerning the coordination of stateful Web Services, it will be described how BPEL deals with them. Namely, how it deals with the so called "long–lived transactions" and with exceptions or errors. It will be also mentioned the work done in this matter by the Organization for the Advancement of Structured Information Standards (OASIS) –a global consortium that drives the development, convergence and adoption of e–business and Web Service standards. OASIS defined the Web Services Resource Framework (WS–Resource Framework) [OASc] in conjunction with Globus Alliance, IBM and HP, to relate web services and states.

2.2.2.1 BPEL case study

Web transactions occur between organizations and they can commonly be long–running, in the sense that there may be dependencies among steps that lead to undetermined periods of interaction (e.g. how long before the contract is accepted by the client?). In these cases, traditional ACID (atomicity, consistency, isolation, and durability) transactions are not sufficient because it is simply not feasible to maintain isolation for such periods of time, due to performance issues [WV01]. This brings the need for Web services orchestration environments to provide a compensation mechanism which can be executed when

the effects of a transaction must be canceled. "Compensations" are actions which attempt to reverse the effects of previous actions; in the BPEL case, the notion of compensation is achieved in the following way: a process designer defines the compensating actions to be performed should an error occur during the execution of the primary process [CKM+03].

BPEL also provides a robust mechanism for handling transactions and exceptions, building on top of the WS–Coordination and WS–Transaction specifications. To group a set of activities in a single transaction, BPEL uses a scope tag. The tag signifies that the enclosed steps should either all complete or all fail. Within this scope, a developer can specify compensation handlers that the BPEL orchestration engine can invoke in case of error. Like the Java programming language, BPEL handles exceptions through throw and catch clauses [Pel03].

Trying to leverage BPEL4WS mechanisms, OASIS continued the work started on the business process language published in the Business Process Execution Language for Web Services (BPEL4WS) 1.0 specification. The result is the WS–BPEL [OASb]. WS–BPEL version 2.0 was accepted as an OASIS standard, in 2007 by its Technical Committee.

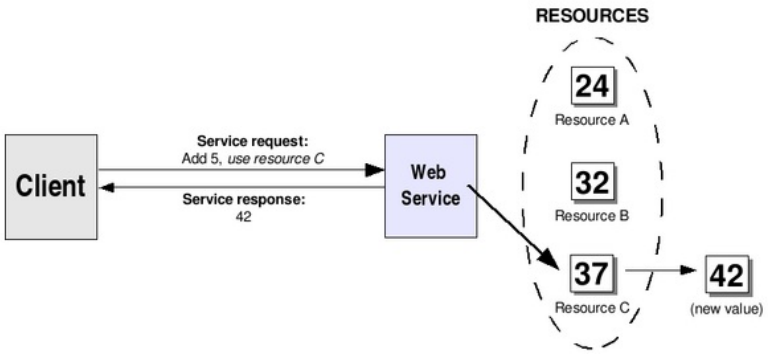


Figure 2.8: Resource approach to statefulness [Sot04]

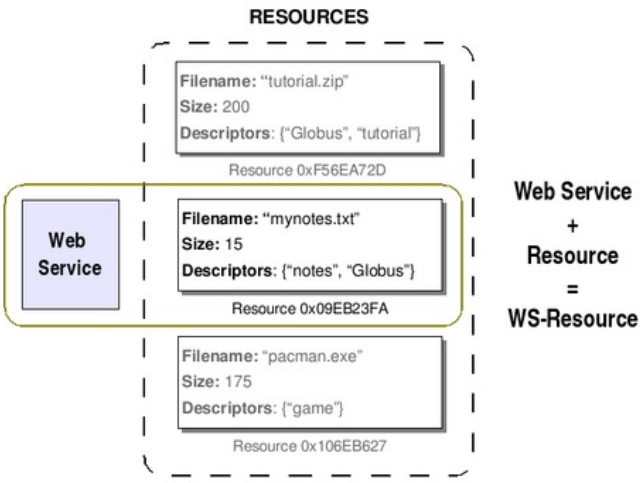


Figure 2.9: WS–Resource [Sot04]

Also concerning the same line of work, OASIS defined WS–Resource Framework

[OASc]. These are a set of specifications that characterize the relationship between stateful resources and Web services. The fundamental concept behind it is straightforward: simply keep the Web service and the state information completely separate. Consider an integer accumulator example. Figure 2.8 shows a resource approach to statefulness. The client invokes the Web service to manipulate the counter value. This is the kind of approach followed by WS–Resource Framework specification –Web service can describe the stateful resources to which it provides access, and a service requestor can discover the type of that WS–Resource. A WS–Resource (Figure 2.9) is the composition of a Web service and a stateful resource.

In the next sections we will finally talk about Dynamic Reconfiguration, but begin by describing Workflows and Patterns, due to their relevance for that subject.

2.2.3 Patterns & Workflows

Workflow is a term used to describe the tasks and procedural steps, to achieve automation (totally or in parts) of a business process, decurrent in organizations and tasked by people or machines [Ple02].

The designing of a workflow forces to examine and understand the business processes, helping to find areas that can be improved. Among other advantages, Workflows help to get the right information to the relevant person, and the relevant information to the right person. The great impact this has in a company in the cost-efficiency of an employee, is obvious [Ple02].

The evolution of the tasks in the workflow is related with the dependencies between them, and since these dependencies are highly recurrent, they were captured as workflow patterns. Namely, the relevance of workflows in business modeling and the observation of several common types of dependencies across different workflows has lead to the sound work by Wil Van Der Aalst [RtHvdAM06, vdAtH10], where those different types of workflows patterns have been identified (and described in a formal way).They include data patterns, control patterns, resource patterns and exception handling patterns [RtHvdAM06, RHEA04, RvdAtHE05, RAH06]

In the following some usual control workflow patterns are described:

- **Sequence** –A task in a process is enabled after the completion of the preceding task in the same process. (figure:2.10)
- **Synchronization** –The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled. (figure:2.13)
- **Exclusive Choice** –The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on a mechanism that can select one of the outgoing branches.(figure: 2.12)

- **Simple Merge** –The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch. (figure:2.11)
- **Structured Loop** –The ability to execute a task or sub-process repeatedly. The loop has either a pre-test (figure:2.14) or post-test (figure:2.15) condition associated with it that is either evaluated at the beginning or end of the loop to determine whether it should continue. The looping structure has a single entry and exit point.

Figure 2.10: Sequence Pattern [vdAtH10]

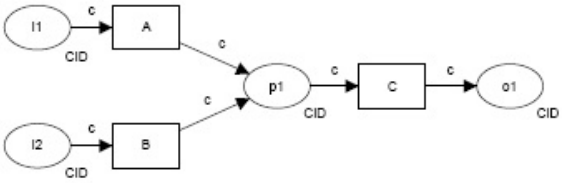


Figure 2.11: Simple Merge Pattern [vdAtH10]

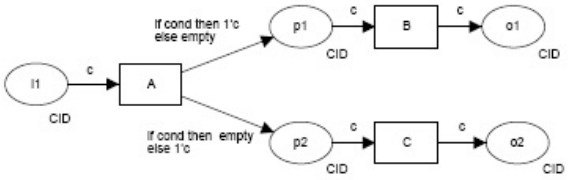


Figure 2.12: Exclusive Choice Pattern [vdAtH10]

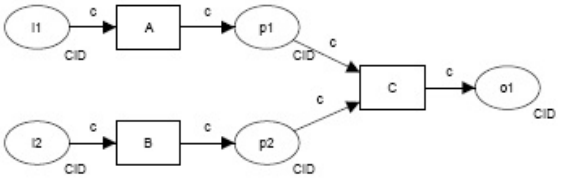


Figure 2.13: Synchronization Pattern [vdAtH10]

Recall the VoD example (Figure 2.5). From the point of view of the client, and considering the timeline, every task he does is accomplished in a sequential manner (figure:2.16).

Workflow Patterns allow to represent similar ways of passing the control flow among processes, in an abstract manner. But the concept of capturing typical recurrences does not only exists in Workflows. In Software Engineering, design patterns represent such an important matter that it has deserved a discipline of its own [GHJV94].

A design pattern does not describe code; instead it allows developers to communicate the mechanism by which the problem they are discussing can be solved. The design pattern is a re–usable solution to a common design problem, that has proven its strength over time.

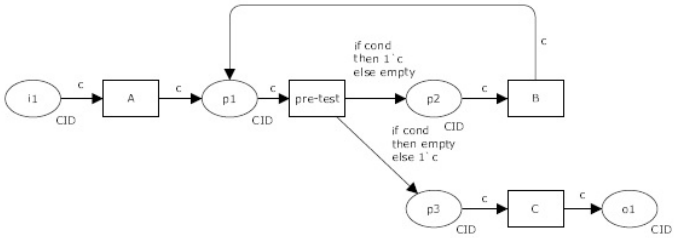


Figure 2.14: Structured Loop Pattern (While) [vdAtH10]

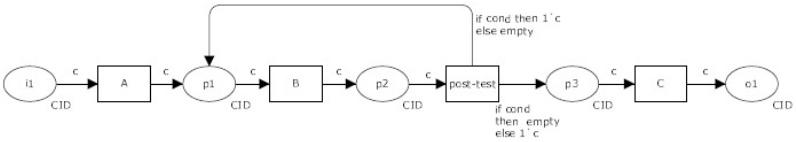


Figure 2.15: Structured Loop Pattern (Repeat) [vdAtH10]

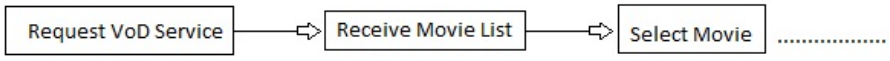


Figure 2.16: VoD Sequence Flow

Design Patterns can be divided into 3 main types: Behavioural, Structural and Creational [GHJV94]. The first one, identify common communication patterns among entities. The second type simplifies the design by identifying an intelligible way to realize relationships between entities. And the last one deals with creational mechanisms.

Some relevant patterns for this work are described next, grouped by type:

- **Behavioural Patterns**

  Client/Server - Almost any service involves a party that provides the service to a client. In the async version of the Cl/Sv pattern, the client invokes the service, but only gets the reply later. There should be an identifier relating the request with the client.

  Peer2Peer - Each peer is seen as client and a server, at the same time. A peer provides and requests services to other peers.

  Producer/Consumer & Streaming - Data flow is unidirectional from producer to consumer; Streaming is a specific case of consumer/producer where the production of data is continuous.

  Publish/Subscriber & Observer - One or more entities subscribe to a certain event of interest. When the event occurs, the subscribers are notified. The observer pattern can be seen as a simplified version of the Pub/Subs pattern. There is no need to register in the event to receive the notification.

  Parameter-Sweeper - Repeated invocations of a component with a unique set of parameters.

Master-Slave - The master distributes tasks to be performed by the slaves and reassembles the results from each one.

- **Structural Patterns**

  Layers - It helps to structure applications into independent (as possible) layers, so that changes in one layer does not affect other ones - extensibility.

  Pipes & Filters - Since it may not be interesting to store all data (for instance captured by sensors), one can filter only certain data, saving storage space.

  Proxy - It can provide location transparency, fault tolerance or load balancing. The proxy serves as an intermediator between the source and the destination entities.

  Facade - The Facade pattern is useful when a system may be divided into several subsystems, and the access to communication/entry-point into the system needs to be restricted.

- **Creational Patterns**

  Abstract Factory - Provide and interface for creating families of related or dependent objects, without specifying their concrete classes.

  Singleton - Ensure that a class only has one instance and provides a global access to it.

  Builder - Separate the construction of a complex object from its representation, so that the same construction process, can create different representations.

Next section, introduces the topic of dynamic reconfiguration.

2.2.4 Dynamic Reconfiguration

Dynamic Reconfiguration is a kind of self–adaptive and management behaviour that allows a system to evolve at run–time [JJ85], as opposed to design–time, while introducing little (or ideally no) impact on the system's execution. In this way, systems do not have to be taken off–line, rebooted or restarted to accommodate changes. These kind of systems address adaptivity in various concerns including performance, security, and fault management [KM03].

Self–adaptive systems, or also referred to as autonomic computing or even self–managing, are systems capable of dynamic reconfiguration. In turn, self–adaptive software is a more limited domain and falls under the umbrella of autonomic computing. Among several existing definitions for self–adaptive software, one is provided in a DARPA Broad Agency Announcement (BAA): "*Self–adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible.*". To do so, such software should

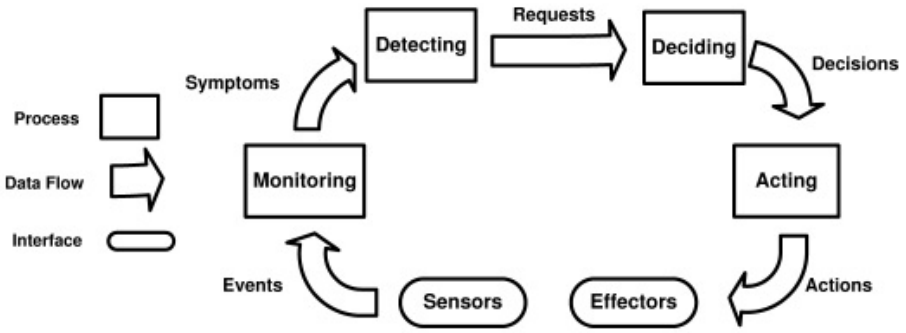


Figure 2.17: Adaptation loop [ST09]

include functionality for evaluating its behavior and performance, as well as the ability to replan and reconfigure its operations in order to improve its operation.

A similar definition is given in [OGT+99]: *"Self–adaptive software modifies its own behaviour in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end–user input, external hardware devices and sensors, or program instrumentation"* [ST09].

Given these two definitions, recall the behaviour of our famous example of the fire detector system, and match the similarities.

Hierarchically, we can consider a layered model for a software–intensive system, consisting of application(s), middleware, network, operating system, hardware and sublayers of middleware [Sch02]. According to this view, the scope of this thesis (self–adaptive software) primarily covers the application and the middleware layers. On the other hand, autonomic computing covers lower layers too, down to even the network and operating system [ST09].

The key point in self–adaptive software is that it should be capable of dealing with a continuously changing environment and emerging requirements that may be unknown at design–time. Its life cycle should not be stopped after its development and initial setup. This cycle should be continued in an appropriate form after installation in order to evaluate the system and respond to changes at all time. Such a *closed loop* deals with different changes in user requirements, configuration, security, and a number of other issues[ST09]. In the following it is described an approach for such control loops.

2.2.4.1 Adaptation loop

The **adaptation loop** in Figure 2.17, also known as MAPE-K (monitor-analyze-plan-execute over a knowledge base) [KM03] consists of several processes assisted by sensors and effectors, and comprising the phases of monitoring, analyzing, planning and executing:

1. The monitoring process is responsible for collecting and correlating data from sensors and converting them to behavioral patterns and symptoms. This process is supported by disciplines such as WSN's environments or distributed systems.

2. The detecting process is responsible for analyzing the symptoms provided by the monitoring process and the history of the system, in order to detect when a change (response) is required. It also helps to identify where the is source of a transition to a new state (deviation from desired states or goals). It is supported by disciplines such as Game Theory, Economic models, Inference and Reasoning.

3. The deciding process determines what needs to be changed, and how to change it to achieve the best outcome. This relies on certain criteria to compare different ways of applying the change, for instance by different courses of action. It is supported by Risk analysis, Planning and Decision theory as cited in [ST09].

4. And finally, the acting process is responsible for applying the actions determined by the deciding process. This includes managing non primitive actions through predefined workflows, or mapping actions to what is provided by effectors and their underlying dynamic adaptation techniques.

Each of these phases includes their own areas of research and challenges, as illustrated in [ST09], which can be summarized next for information purposes:

1. A significant challenge for monitoring different attributes in adaptable software is the cost/load of the sensors. In most cases, a number of in vivo methods collect various information, which may not be required. Either by collecting details that are not needed , or by not adapting the collection process should the software change its behaviour from the original one. For example in wireless sensor networks (WSN's), this feature is so important for sensor nodes, as an energy–safe measure (e.g. low duty cycle).

   As another example, consider a system that is monitored to determine when a particular level of Quality of Service (QoS) is not satisfied, which then initiates the adaptation process. However, monitoring a system, especially when there are several different QoS properties of interest, has an overhead. In fact, the amount of degradation in QoS due to monitoring could outweigh the benefits of improvements in QoS to adaptation.

2. Challenges in the detecting process include the ability of the system to determine which behaviours/states are considered as healthy/normal. Answering this question often requires a time-consuming static and dynamic analysis of the system, which may also be strongly affected by the underlying random variables (i.e., users' requests arrival times, and faults in different components). Although there have been efforts to apply statistical and data mining techniques to address this issue, the existing realizations of this process are still mostly ad hoc and incomplete.

3. A self–adaptive software system often needs to perform a trade–off analysis between several potentially conflicting goals. Current state–of–the–art techniques leverage a utility function to map the trade–offs among several conflicting goals

to a scalar value, which is then used for making decisions about adaptation. However, in practice, defining such a utility function is a challenging task. Practical techniques for specifying and generating utility functions, potentially based on the user's requirements, are needed.

On the other hand, in the presence of multiple objectives, in addition to the necessity of deciding online and dynamically, one faces the following additional challenges: (i) finding approximately or partially optimal solutions for multiobjective decision–making problems, (ii) dealing with uncertainty and incompleteness of events/information from the system's self and context, (iii) correlating local and global decision–making mechanisms, and (iv) addressing the scalability and fault-proneness of the decision–making mechanism using centralized or decentralized models.

4. One important acting challenge is how to assure that the adaptation is going to be stable and have a predictable impact on the functional and non–functional aspects of the underlying software system. It is important to know: (i) whether the adaptation actions follow the contracts and the architectural styles of the system, (ii) whether they impact the safety/integrity of the application, and (iii) what will happen if the action fails to complete, or if preemption is required in order to suspend the current action and deal with a higher priority action. These are important issues to be considered, specially in safety–critical software systems, since adaptation itself should still ensure the safety requirements. There is a need for verification and validation techniques that guarantee safe and sound adaptation of safety–critical systems, under all foreseen and unforeseen events.

5. There are also other challenges aside from the adaptation process, such as requirements analysis, design, implementation, self–evaluation, testing and assurance, evaluation and quality of adaptation.

2.2.5 Pattern-based reconfigurations

In respect to self-adaptive software previously mentioned, another tool that can help to achieve a dynamic reconfigurable system is *Pattern-based reconfiguration*. The idea is that patterns as first class entities, become manipulable and thereby reconfigurable, for example through patterns operators. Related work concerning reconfigurations based on patterns operators is presented in [GRC08]. The authors propose an approach that makes use of design patterns at the compositional level (referred to as "structural patterns") and at the behaviour/run time level (referred to as "behavioural patterns"). These patterns can then be manipulated using scripting tools through the use of "structural" and "behavioural operators". Some other works that use the pattern concept for systems' self-adaptation mechanisms make use of reconfigurable Architectural Patterns on system

definition [HM08]. For example, the inherent architecture of the Publish/Subscriber pattern allows it to be reconfigurable in terms of the number of publishers, subscribers and the events they subscribe to; the Master/Slave pattern allows the addition of new slaves to optimise task execution [ADK09].

Another work concerning the same subject is presented in [BPK+06]. The authors present one solution for self-adaptability of service-generated data streams targeting problems such as data loss or delays associated with communication networks disruptions. The solution uses a distributed hierarchical structure of controllers/actuators for a) adjusting data flow according to the detected dynamic variations; b) saving these data in buffers whenever necessary in order to avoid data loss; c) in-transit data processing at the hierarchy nodes hence reducing the execution time of the application generating the data.

So up until now, we have discussed forms of coordination in web services and the way they can provide automation of processes through dynamic reconfiguration (DR). Some of its supporting disciplines still don't quite manage to answer to all the challenges that comes with DR.

Next sections will present a specific domain area - wireless sensor networks (WSNs) and the usefulness of the previous discussed technologies, in this emerging topic.

2.3 Wireless Sensor Networks

The current advent of nano-technology has made it technologically feasible and economically viable to develop low-power devices that integrate general-purpose computing with multi-purpose sensing and wireless communications capabilities. It is expected that these small devices, referred to as sensor nodes, will be mass-produced and deployed, making their production cost negligible. Examples of interesting applications using these networks are habitat monitoring, infrastructure security, traffic control, controlling homes, military applications (e.g. submarine detection), industry and manufacturing automation, weather information (e.g. temperature, wind, moistness).

2.3.1 Sensor Node

WSNs have been the subject of intense research for nearly a decade. The applicability of this technology happens as a consequence of the technological trend we have experienced in the last century or so and described by Moore's law –"Doubling of transistors every two years", affecting the capabilities of electronic devices such as processing speed, memory capacity, sensors and even the number of pixels in digital cameras [Myh06].

Related to this subject is a wireless sensor node (Figure2.18) that typically has embedded processing and storage units, potentially multiple onboard sensors (each containing an ADC - analog to digital converter), a power unit, a location finding system, an antenna, to name a few. These sensor nodes form a sensor network and they usually are

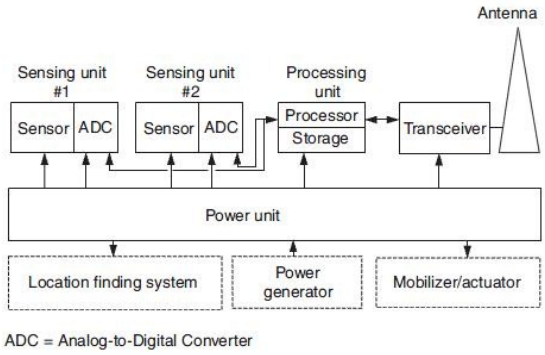


Figure 2.18: Typical Sensor Node [SMZ07]

attached to a sink node responsible for aggregating/summarizing significant data captured by the sensor nodes. The position of the nodes does not necessarily have to be predetermined, allowing random deployment in inaccessible terrains or dynamic situations (e.g., a dangerous location or an area that might be contaminated with toxins or be subject to high temperatures). Meaning that sensor network protocols and algorithms must possess self-organizing capabilities, making them versatile [ASSC02, RSZ04].

2.3.2 Technical Challenges

In this section it will be described some relevant technical challenges, again for information purposes.

2.3.2.1 Ad Hoc Network Discovery

It is necessary for each node to know its neighbors to support processing and collaboration. In ad hoc networks, the topology needs to be constructed in real-time, and updated periodically as sensors fail or new ones are deployed - environmental dynamics. Survivability and adaptation to the environment are ensured through deploying an adequate number of nodes to provide redundancy in paths, and algorithms to find the right ones. On the other hand, fine-grained time synchronization and localization are needed to detect events of interest in the environment under observation. (E.g. On what floor and in which quadrant is the smoke detected? What is the temperature of the atmosphere at a given height h?).

2.3.2.2 Communication

The collection of data can be done by transmitting it directly from sensor nodes to the centralized data collection. It is, however, undesirable because the required transmission power increases as the square of the distance between source and destination. In fact, if the distance between source and destination is R, the power required for single-hop transmission is proportional to R^2. If nodes between source and destination are taken advantage of to transmit n short hops instead, the power required by each node is

proportional to R^2/n^2 [CDW04]. Multiple short message transmission hops require less power than one long hop. Each node in the sensor network can act as a repeater, thereby reducing the link range coverage required and, in turn, the transmission power.

2.3.2.3 Information Processing

It is desirable to communicate via short distances messages, instead of long ones. Therefore, distributed (local) algorithms are attractive because they are robust to network changes and node failures. However, these are difficult to design because of the potentially complicated relationship between local behavior and global behavior. The challenge comes from the fact that, there is no entity with exactly knowledge of the whole network. Consequently, even though each node only has a partial/local view of the network, all of them behave in a way that the system eventually converges to a global optimum. Intrinsic to this is the use of in-network processing as way of reducing, aggregating, and summarizing the data, and so the communication throughput. Furthermore, because the data which is being collected by multiple sensors is based on common phenomena, there is potentially a degree of redundancy (duplications) in the data being communicated by the various sources in WSNs. Care must be taken in order to properly aggregate that data into useful information (e.g. hierarchical super nodes are responsible for fusing the data).

2.3.2.4 Power Management

Many applications do not constantly require data capture by sensing nodes, meaning that they do not need to always remain in an active state. It is desirable that in those periods of time, the node can switch into a sort of a 'sleepy' state in order to save power consumption. This can be achieved using low duty cycle protocols. The term duty cycle describes the proportion of 'on' time to a regular interval or period of time; a low duty cycle corresponds to a duty cycle in which most of the time, the power is 'off'. Duty cycle is expressed in percent, 100% being fully on. Say for example, that one node has a duty cycle of 30% in a period of 10 minutes; it means that in 7 of those 10 minutes, it is in a 'resting' mode.

2.3.2.5 Standardization

There are more than 3,000 global sensor manufacturers. Trying to use a variety of sensors from a number of different manufacturers together in a data acquisition system can be very complex and require expensive customization by the integration team. For that reason, in 1993 the IEEE and the National Institute of Standards and Technology (NIST) began working on a standard: IEEE 1451 –standard for Smart Sensor Networks. The objective is to make it easier for different manufacturers to develop smart sensors and to interface those devices to networks. Among other specifications is the wireless interface that meets the IEEE Standard 1451.5-2007 (WiFi, Bluetooth, and ZigBee). Particularly,

ZigBee is focused on providing reliable, cost-effective, low-power, wirelessly networked communications.

On a higher architectural level, challenges are, for instance, related with exporting data captured by WSNs and providing it as a service to the 'outside'. Preferably, WSNs would be seen as just another easily integrable software component and which is accessed via Web Technologies, consequently extending the Internet into the actual physical world. In order to meet these goals, efforts have been made by the community to allow remote (Web) access and management of WSNs, as well as, to provide standards that turn the integration process smoother. Respecting this concern there are the Sensor Web Enablement (SWE) specifications [BPRD07] of the Open Geospatial Consortium. SWE is a set of models and Web Service interfaces for the Internet integration of sensor systems. The goal is to provide a specification for the integration of sensor networks in a Web of sensors accessible via Internet technologies. The standards described by SWE are focused (at the time this thesis was written) on the following functionalities:

- Discovery of sensor systems, observations, and observation processes that meet the immediate needs of an application or user
- Determination of the capabilities and quality of measurements of a sensor.
- Access to sensor parameters that automatically allow software to process and geolocate observations.
- Retrieval of real-time or time-series observations and coverage in standard encodings.
- Tasking of sensors to acquire observations of interest.
- Subscription to and publishing of alerts to be issued by sensors or sensor services based upon certain criteria.

To meet these functionalities, the SWE initiative established several encodings to describe sensors and sensor observations through web technologies currently in vogue. Mainly, Web Services Technology and Extensible Markup Language (XML). To some extent, such usage of widely accepted technologies can be seen, in itself, a way of enforcing standardization in the area of SWE.

SWE has been adopted by several relevant enterprises on their projects. Some examples include NASA, the German organization 52North, the Northrop Grumman Corporation (NGC), the European Space Agency and various partner organizations in Europe [Bac07].

```
1 interface WSNService {
2  String[] getNetworks();
3  String[] getSinks();
4  String[] getSinks(String networkId);
5  String[] query(String snId, String query, Date targetDate);
6  String[] requestStream(String snId, String condition, Date startDate, Date endDate, int rate);
```

```
7   String[] requestNotification(String condition, Date validityDate);
8   boolean subscribe(String topic);
9   boolean unsubscribe(String topic);
10  String[] getSensorNetworksOperations(String networkId);
11  String[] getOperationParameters(String networkId, String operation);
12  String[] executeOperation(String snId, String operation, String[] params);
13 }
```

Listing 2.1: Interface WSNService

```
1  interface WSNAdminService {
2   String[] getNetworks();
3   String[] getSinks();
4   String[] getSinks(String networkId);
5   void registerSink(String sinkId, String args);
6   void unregisterSink(String sinkId);
7   String[] getPropertyList();
8   String getPropertyValue(String property);
9   void setPropertyValue(String property, String value);
10  String[] getNetworkPropertyList(String networkId);
11  String getNetworkPropertyValue(String networkId, String property);
12  void setNetworkPropertyValue(String networkId,String property,String value);
13  String[] install(String snId, String program);
14  void setFilter(String snId, String filter);
15 }
```

Listing 2.2: Interface WSNAdminService.

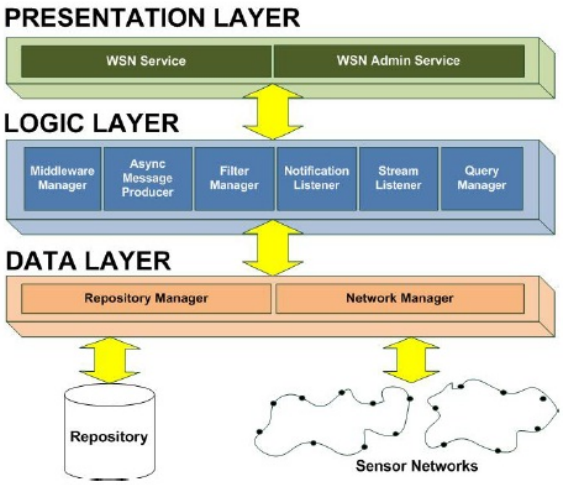


Figure 2.19: SenSer Architecture [San09]

Related work in this area include [AHS06, HlFcRL06] and SenSer [San09] - A Middleware Framework for the Remote Access and Management of Sensor Networks. Senser provides operations that are divided in two categories: regular user and administrator. The former may access the target network to post queries, require data-streams, subscribe notifications, perform operations, among others (2.1). The latter may perform management operations, such as register and unregister sinks, and reprogram a network (2.2).

Concerning user operations, conditions are used to parameterize stream and notification services. In the first, the condition, along with the specification of a maximum data rate, operates as a filter on the data to be received. In the second, the condition specifies the scenarios on which a notification event is to be raised.Notifications may encompass more than a single sink or network type, thus no explicit target sink argument is included.

Senser's architecture follows a three-tier model, decoupling the presentation, logic and data layer (Figure 2.19), the latter comprising the two possible data sources: registered sensor networks and a history repository.

Now that we have studied the tools that will support the elaboration of this thesis, next chapters will describe the proposed solution and its implementation, as well as, the results from the evaluation process.

3

Proposed Solution

This Chapter presents the conceptual view of the system and the adopted approach, which is based in patterns. Afterwards, we will present a '*top-level*' view of the system and describe each component.

3.1 Conceptual view of the system

The goals of this thesis previously identified in Section 1.2, can be summarized as follows: *a)* to provide additional interaction models useful for stateful Web services, in particular in the context of Web enabled WSNs; *b)* to define a set of dynamic reconfigurations that may focus in a specific interaction model, or on the replacement of it for another. Since these reconfigurations may depend on the context of the interaction model, it is also necessary to *c)* represent such context, as means of capturing characteristics of those interactions, at run-time. The conceptual view of the proposed solution is illustrated in Figure

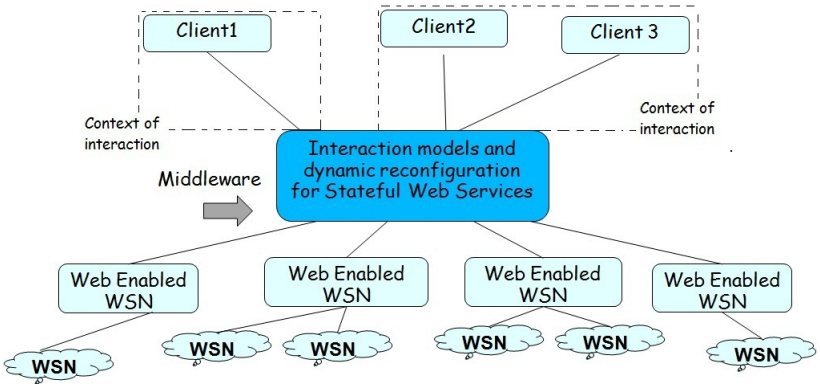


Figure 3.1: Conceptual view

3.1. It contemplates the development of a middleware layer that provides *a), b)* and *c)*, which hides the details inherent to the access of Web enabled WSNs, and provides an uniform interface. From our point of view, the clients that interact with the service according to the same interaction model and characteristics (e.g data source), should be able (but are not obliged) to share the same context of interaction. As a consequence, the dynamic reconfigurations performed upon that context, affect all the clients present in it (the justification for this purpose is given in the next Section). Despite, a client should also be able to independently define its own context of interaction. Thus, it is next described the representation of a context.

3.2 The session concept

The context of interaction between the service and the clients is defined through the notion of session. One Session (Figure 3.2) has associated an interaction model to be guar-

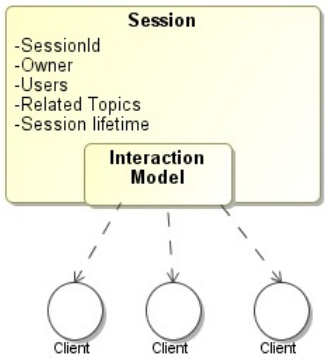


Figure 3.2: Session

anteed for all users within that particular session (same context of interaction model). It also includes information related to its state, such as:

- An identifier that allows, for example, new clients to join the session;
- The identifier of the "owner of the Session" (the older user that created the Session);
- Identifiers of users participating in it;
- The topic(s) related to the session (in practise, the topic(s) related to a session, identify the data source(s) of that session);
- The lifetime of a session, after which the session will be destroyed. A session with an unbound time limit is destroyed either by explicit request from its owner, or when he leaves it.

Figure 3.3 illustrates two different instances of a Session. The first is related to a Publish/Subscribe interaction model, whose owner is *Client 1*, it has no time constraints

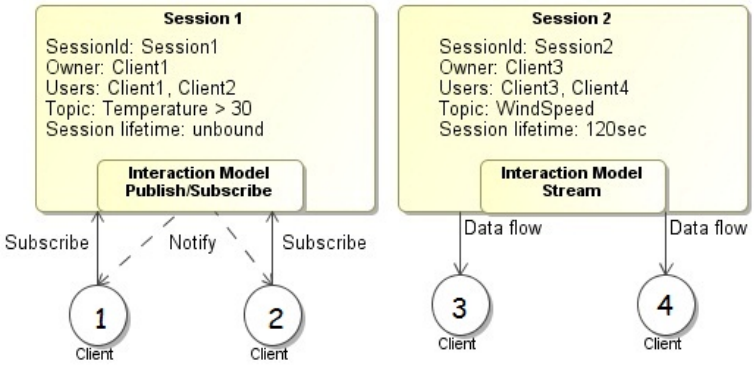


Figure 3.3: Sessions

and the clients are notified when the event related to the topic "Temperature > 30" takes place. The second session belongs to *Client 3*, it has a lifetime of 120 seconds and the clients receive a stream of data related to the topic "WindSpeed".

All clients participating in the same session, comply to the following: *1*) they share the same interaction model defined by the owner in the moment of its creation. *2*) the reconfigurations that may alter the state of a session, can only be explicitly requested by its owner. *3*)the reconfigurations that affect the session as a whole, result in a state transition of the session that is notified to the other participating clients. The need for the previous three, is exhibited by the fire detection example described in 1.1.1, since any additional fire departments chosen to help the first, must be able to access the same data. The fire detection example, falls into a class of problems where it is useful to delegate the control of a session to a main entity and whose participant clients are subject to the reconfigurations made by the owner. Another examples could include: an university that provides the streaming of a video conference to be accessed by its students, a notification service to notify the users about subscribed topics (e.g notification service of a discipline to disseminate important dates, such as, delivery deadlines, exam dates, etc), among others.

3.3 Pattern-based Dynamic Interaction Models

The adopted approach in the implementation of the interaction models is based on the *Design Patterns* concept. The literature identifies three main types of patterns: the creational, the structural and the behavioural patterns [GHJV94]. Our focus will reside on the last two in order to capture the static and dynamic relations among a session's members (e.g. one service and its clients).

3.3.1 Structural and Behavioural Design Patterns

The *structural patterns* capture a session's static view in terms of the structural dependencies/relations among its members (e.g. pipeline). Namely, they define the links among

their elements, without however, specifying any restrictions in terms of data or control flows. These are defined by the *behavioural patterns* which capture a session's dynamic view (e.g the Producer/Consumer pattern). Specifically, they characterize the dependencies in terms of data and control flows among a session's members as well as their role concerning the the behavioural patterns' semantics (e.g. roles of "producer" and "consumers" when considering the Producer/Consumer).

Furthermore, the proposed solution considers both structural and behavioural patterns as abstractions in the form of *pattern templates*, as specified in [GRC08]. A pattern template aggregates a set of "component place-holders" according to the corresponding pattern semantics, where the place-holders may be instantiated to Web services, for instance. Additionally, the composition of a structural pattern template with a behavioural pattern template, as explained in section 3.3.2 ahead, results in a set of elements tagged with both structural and behavioural semantic annotations (e.g. an element in pipeline is a producer and the next element is a consumer).

On one hand, the considered structural patterns were the facade and the pipeline. The facade pattern, in particular, represents the possibility of interfacing a set of heterogeneous systems (i.e. which may show distinct interfaces). Namely, the facade aggregates a set of elements where one particular element (the "façade") provides an uniform access interface to the other elements in the pattern (named as "subsystems") [GHJV94], hiding the communication details to the sub-systems from the external entities accessing the pattern.

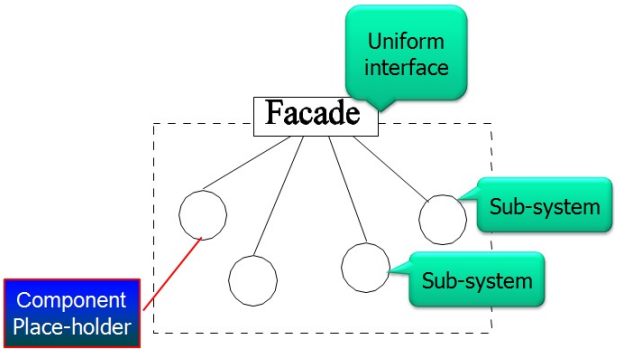


Figure 3.4: Structural pattern template - Facade

Figure 3.4 shows a particular example of a facade structural pattern template containing place-holders for the "facade" and four sub-systems. The facade pattern is used in our solution as a way to capture the structural relations concerning a set of clients accessing the same service in the context of a particular session. Additionally, it is also used to represent the structural dependencies underlying data aggregation acquired from diverse services.

The pipeline structural pattern, in turn, captures the structural relations between a sequence of stages, where the first is connected to the second, the second is also connected to the third, etc.

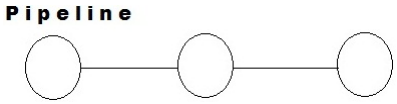


Figure 3.5: Structural pattern template - Pipeline

Figure 3.5 represents a particular pipeline pattern template with three component place-holders. The relevance of this pattern in terms of capturing the structural dependencies among the members of a session will be described in more detail in the implementation of the Aggregation interaction model in Chapter 4.

On the other hand, the selected *behavioural patterns* to capture the data and control flow dependencies among a session's members were the Client/Server, Publish/Subscribe, Streaming, and Producer/Consumer. Therefore, these patterns represent possible interaction models for client access to services which we identified as being important in the context of WSNs. Additionally, the Aggregation interaction model is defined in terms of one of those selected pattern. Specifically, data is acquired from a set of data sources according to a particular behavioural pattern, say Streaming, and the aggregated data is sent to all clients in the session according to that same behavioural pattern, i.e. streaming. Therefore, the aggregation interaction model may represent an aggregation of data streams, an aggregation of subscription notifications in the case of the Publish/Subscriber pattern, etc.

The justification for the choice of the cited interaction models lies in the fact that they represent data flow with distinct quality services, allowing enough diversity on the access to data generated by the networks. For instance, if a client wishes to evaluate a certain condition over a registered value by the WSNs, say '*Temperature* > *50*', he could constantly request for readouts upon the source temperature and perform the evaluation himself (similar to a *busy waiting* technique). This is, however, inefficient and it is not guaranteed that the client notices the event at the moment that it took place. For this purpose, the Publish/Subscribe interaction model is more appropriate. As another example, and recalling the fire detection scenario in 1.1.1, it would be interesting from the point of view of the client, to be able to acquire data from multiple sources (e.g temperature, wind speed, humidity), to aggregate that data according to a criteria of his choice, and finally to disseminate the aggregated data according to a behaviour. These features are assured by the Aggregation interaction model.

3.3.2 Combination of a structural pattern with a behaviour

The implementation of a particular interaction model is based on the composition of one or more structural patterns with a behavioural pattern.

Figure 3.6 shows two examples of such composition, namely the composition of a facade structural pattern with two different behavioural patterns. In the first case, the "façade" element in the structural pattern will behave as "producer" at runtime, to the "sub-systems" in the structural pattern, i.e. the clients, which will behave as consumers.

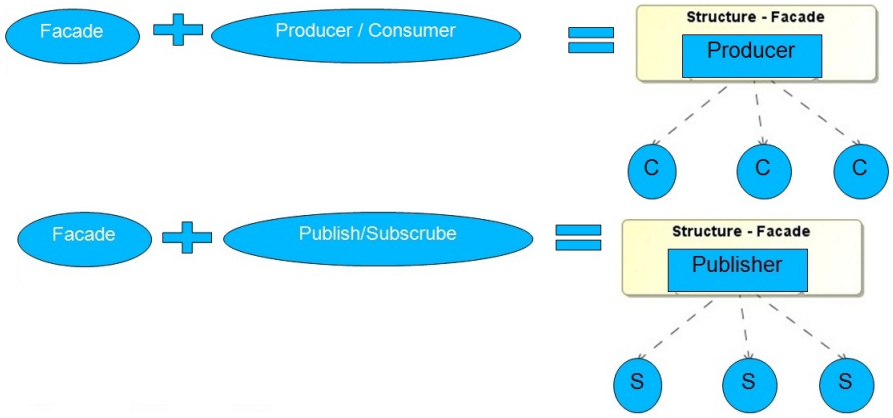


Figure 3.6: Composition of the same structural pattern with different behavioural patterns.

Therefore, it is possible to identify the following structural and behavioural annotations, capturing, respectively, the static and dynamic views of four related elements in the system:

- there is one element annotated both as a "façade", i.e. the structural annotation defining that the element is the "common entry point" to the other elements in the façade pattern, and as a "producer", i.e. a behavioural annotation defining that this element produces data (with some delivery guarantees in terms of minimizing data loss) in a decoupled away from the entities which will consume this data;

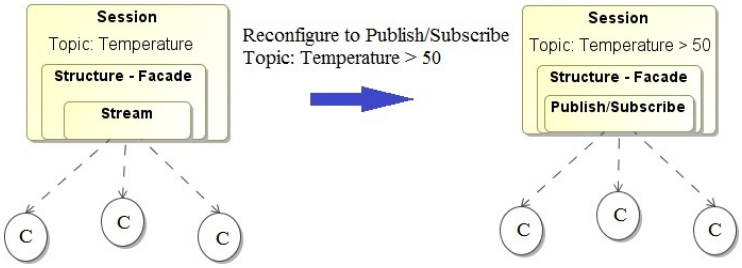
- there are three elements each one annotated both as a "sub-system" (structural annotation concerning the facade's semantics) and as "consumer", i.e. a behavioural annotation conform to the Producer/consumer semantics defining that the element will retrieve/consume data from a repository (i.e. decoupled in time from the data source).

The second case is similar to first in the structural annotations, but distinct in the behavioural semantics which is defined by the Publish/subscribe behaviour.

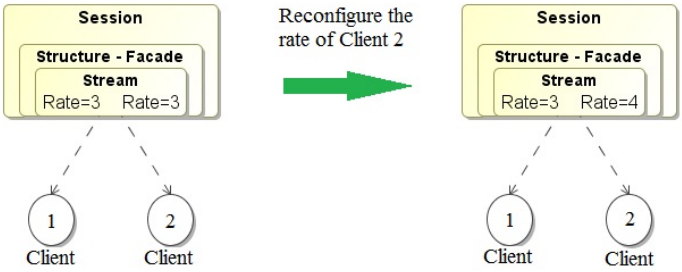
The usage of such structural and behavioural pattern compositions in the context of the middleware's implementation will be described in detail in chapter 4. For instance, the facade pattern composed with a behavioural pattern is used to capture service data dissemination to service clients using a particular interaction model, being the latter based on that behavioural pattern. Likewise, data acquisition and aggregation generated by different services is also captured as a facade composed with a particular behavioural pattern.

3.3.3 Pattern-based dynamic reconfiguration

The dynamic reconfigurations considered in this thesis are based in pattern operators as presented in [GRC08]. We find this approach particularly appropriated to deal with the



(a) Reconfiguration from Stream to Publish/Subscribe



(b) Reconfiguration of the rate of dissemination of a client

Figure 3.7: Reconfigurations

dynamicity of the system intended to implement. Once a Session is first combined with a particular structure and behaviour, it can be modified/reconfigured at run-time through pattern operators, and thereby, enabling the service to adapt to changes in the computational environment. Moreover, the separation of concepts of Session, Structure and Behaviour, and the way they were structured, serve the purpose of limiting the impact of reconfigurations. Consider the two Figures in 3.7. They illustrate possible reconfigurations that focus on different entities. The first (3.7a) respects the reconfiguration from a Stream session to Publish/Subscribe. It is made by replacing the former behavioural pattern and updating the related topic of the Session. The second is requested by *Client 2* to change his data transfer rate. A data transfer rate concerns interaction models with a continuous data flow from the middleware to the clients (for example, it would not make sense to define a rate in a Client/Server behaviour). Therefore, such reconfiguration should be restricted to the scope of the respective behaviour, as illustrated in 3.7b.

3.4 Overview of the system

The '*top-level*' view of the proposed solution is illustrated in Figure 3.8. The pattern-based middleware operates over a framework - Senser [San09], that allows for the remote (Web) access and management of sensor networks by virtualizing their functionalities as services. Senser offers services of subscription and notifications and streaming of data collected by the sensor nodes. Despite, we believe that a) it would be beneficial to provide other type of services as interaction models, implemented by patterns; b) to offer dynamic reconfigurations mechanisms, based in pattern operators, to capacitate the system to evolve at run-time; c) and present an user API to enable an easier access by the

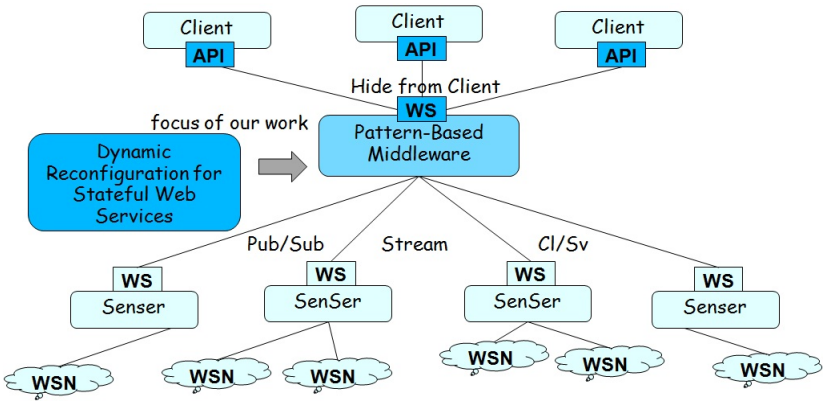


Figure 3.8: 'Top-level' of the system

client. Therefore, we propose a middleware that offers, in one hand, a set of interaction models represented as the combination of a structural pattern with behaviour, in the context of a Session. And, in the other hand, dynamic reconfiguration mechanisms of those models (section 4.4 identify those mechanisms). Finally, it was developed a Java user API that hides the details of communication intrinsic to Web services technology (section 4.5 explains its usage).

4

Implementation

This Chapter exposes the architecture of the middleware. It will be described the relationship between the logical components, the details of their implementation, and the dynamic reconfigurations that we considered to be relevant, as well as, the mechanisms to support them. Finally, the last section presents a Java user API developed to ease the interaction between the client and the service.

4.1 Architecture

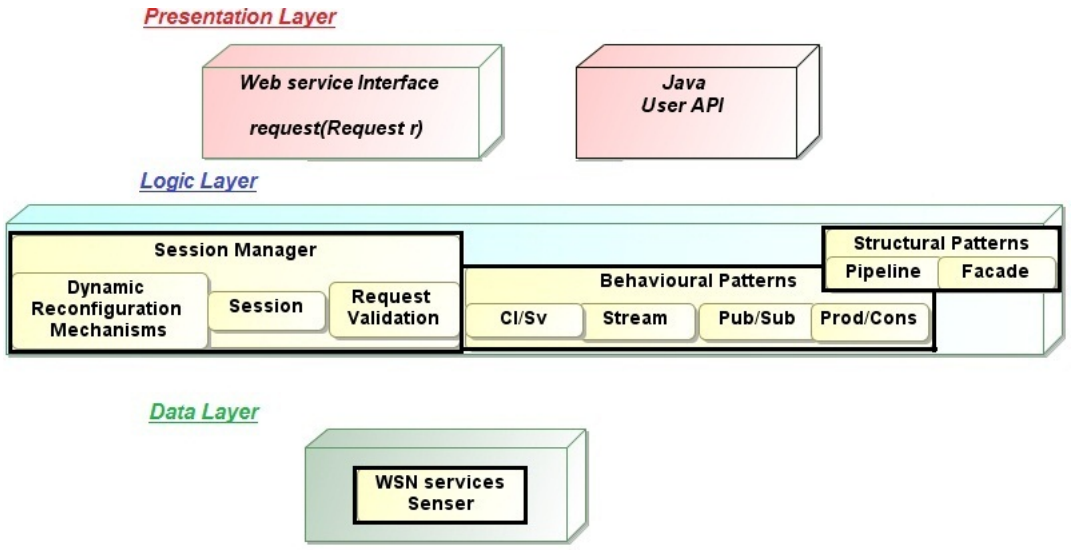


Figure 4.1: Three-tier model

The architecture of the middleware comprises several components, organized in a

three-tier model (Figure4.1), in which the presentation, logic and data layers are decoupled.

The presentation layer exposes an interface accessible via Web service, with a single method request(*Request)*, whose argument must be properly parametrized when invoked. Its simplicity is justified by extensibility issues: in case the middleware should be extended with new components that could conceivably add new functionalities (e.g. new interaction models), there would be no need to re-publish the service interface.
However, as one may notice, this interface forces the client to know the details inherent to the Web services communication, as well as, to explicitly manage the interaction models. For this cause, we developed a more friendly Java user API (section 4.5) that hides those details.

The logic layer grants the implementation of the necessary components to provide the interaction models and their dynamic reconfiguration. The ones that we consider relevant in the context of WSN are, the already stated, Client/Server, Publish/Subscribe, Producer/Consumer, Stream, and also the Aggregation of the previous ones. The Aggregation interaction model, is not exactly implemented as a behavioural pattern, but instead as the combination of several patterns. The details about its implementation are given in the next chapter 4. As for the components of the logic layer, in particular, the role of each one is:

- *Session Manager* - Represents the entity that manages and maintains the consistency of the existing sessions; maps a data source to the related sessions. In addition, it is responsible for the validation of requests and the dynamic reconfiguration mechanisms;
- *Structural Patterns* - creation of structural patterns used in the context of a session;
- *Behavioural Patterns* - creation of behavioural patterns used in the context of a session, setting the model of interaction between the service and its users in this context;

The aggregation interaction model allows a client to combine multiple data sources, and the criteria of aggregation is defined by him. For instance, it is possible to the client to define a *racio* between the sources or to calculate the average of the registered values in the incoming data. At last, the data layer access other systems (Senser) via Web services and transfer the data the logical layer.

4.2 Relationship between the logical components

The relationship between the logical components previously cited, is illustrated in Figure 4.2.

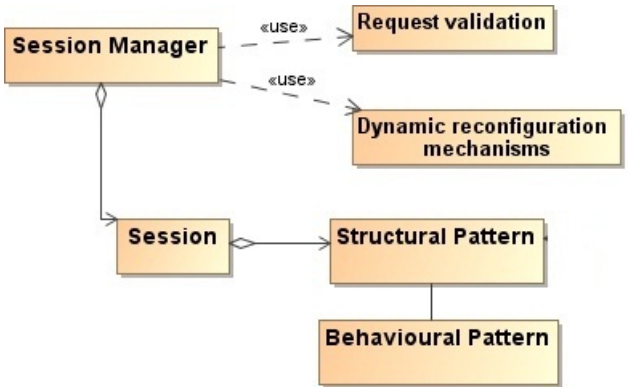


Figure 4.2: Relationship between the components

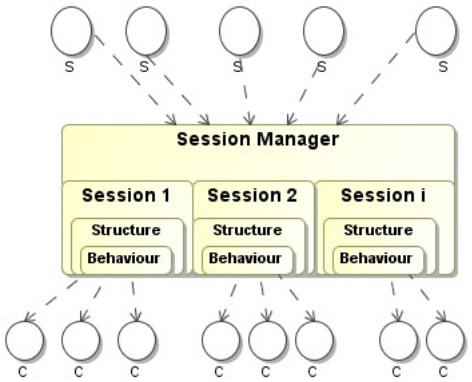


Figure 4.3: Session Manager

- The Session Manager (Figure 4.3) is responsible to maintain the consistency of the existing sessions, and to map the data from a given source to the related sessions. It also initiates the process of validation and dispatch of a request sent by a client, and updates the necessary data that may result from a reconfiguration;

- The validation of a request sent by a client is made through the authentication of the Session to which the request is addressed, the identifier of the client in that session and the (requested) operation in the context of that Session;

- The dynamic reconfigurations mechanisms are automatically triggered in situations where it is necessary to transition from the current state to another. Section 4.4 gives deeper information about those reconfigurations and the mechanisms used to support them;

- A structural pattern is combined with a behaviour and both define the interaction model to be assured between the middleware and the clients;

- The considered behavioural patterns were:

  Client/Server - This is the default interaction model between the users and the middleware. Through it, the clients can request for the current existing sessions, the

available sources (topics), obtain the registered value in a topic, or even to create a session, join another or reconfigure the current one;

Publish/Subscribe - The clients subscribe to a topic , e.g. *"Windspeed > 40"*, and they are notified of such event. In addition to the topic, the owner of the session (who created it) may also indicate another interaction model to which the session is automatically reconfigured when the event occurs;

Stream - The clients involved in a stream session continuously receive data related to a source;

Producer/Consumer - Similar to Stream behaviour, with the difference that the buffer on the client side detects when its storage levels are high, meaning that the client is not capable of consuming the data at the same rate that the middleware sends them. In this situation, the buffer itself sends a request to the middleware to lower the specific rate of this client;

- A Session is composed by the combination of a Structural Pattern with a Behavioural, and has associated a topic(s) that identifies the source(s) related to it.

Although the interaction model related to the aggregation scenario is not explicitly presented in the components, we will see in the next section 4.3 that an Aggregation results from the combination of two structural patterns (facade and pipeline) along with a behaviour (responsible for the dissemination of the aggregate data).

4.3 Implementation of the logical components

In this section we will describe in detail the implementation of the logical components. It is reserved a single subsection to each one of them. In every subsection, it will be described the data structures and the variables that compose the component and how they are involved in two different processes: the validation and the dispatch of a request sent from a client, and the dissemination of data from a source to the respective sessions.

4.3.1 Session Manager

The Session Manager is the main entity in the middleware. The entire data flow occurring at a given time between the middleware, the clients and the data sources is monitored by him. He contains two main data structures to directly support his two main roles: a) the maintenance of sessions which includes the creation, the destruction and the reconfiguration of sessions. b) The dissemination of data from Senser networks to the related sessions.

The first structure 'sessions' is an HashMap<String,Session> and contains the current existing sessions. The key of an entry is a String (sessionId) that univocally identifies the session in the middleware, and the value of that entry is an instance of the object Session.

Every new session that is created by a client request, origins a new entry in this structure whose Id is generated by the Session manager. A running session has always, at least, one client. As soon as the last client leaves it, the session is destroyed and removed from the structure.

The second structure 'notifications' is an HashMap<String,DataFlowDissemination> that maps a source (topic) to the related sessions. The key of an entry is a String (topic) that identifies a source, and the value of that entry is an instance of the object DataFlowDissemination. This object comprehends a collection of sessions and possesses the following methods:

- public void addSession(Session s) - adds a new session to receive the incoming messages from the source identified by the topic;
- public void removeSession(Session s) - removes the session.
- public void disseminate(ClientNotificationMessage message) - Disseminates the message for all sessions. When the Session Manager receives a message from a source, maps it to the related sessions and disseminates the message:
  notifications.get(message.getTopic()).disseminate(message);

4.3.1.1 Dissemination process in the Session Manager

The structures 'sessions' and 'notifications' evolves in accordance to each other, in the sense that an instance of a Session is always referenced by both. Figure (4.4) illustrates a situation where there are 3 running sessions. One related to the behaviour Publish/Subscribe with the topic "Temperature > 30". The second related to the behaviour Stream with the topic "WindSpeed". And the last related to the Aggregation interaction model with topics "Temperature" and "WindSpeed". Notice that the Session with the aggregation scenario will be referenced in two entries of the structure notifications, in order for it to receive data from both sources. Also notice that, the DataFlowDissemination3 instance has two entries - Session2 and Session3.

4.3.1.2 Validation and dispatch of a request in the Session Manager

A request sent by a client is treated in the following way:

- Creation of a new session - if the client asks for a new session, the session is created with a behaviour and related to the topics defined by him. He becomes the owner of the session and this new instance is added to both 'sessions' and 'notifications' data structures;

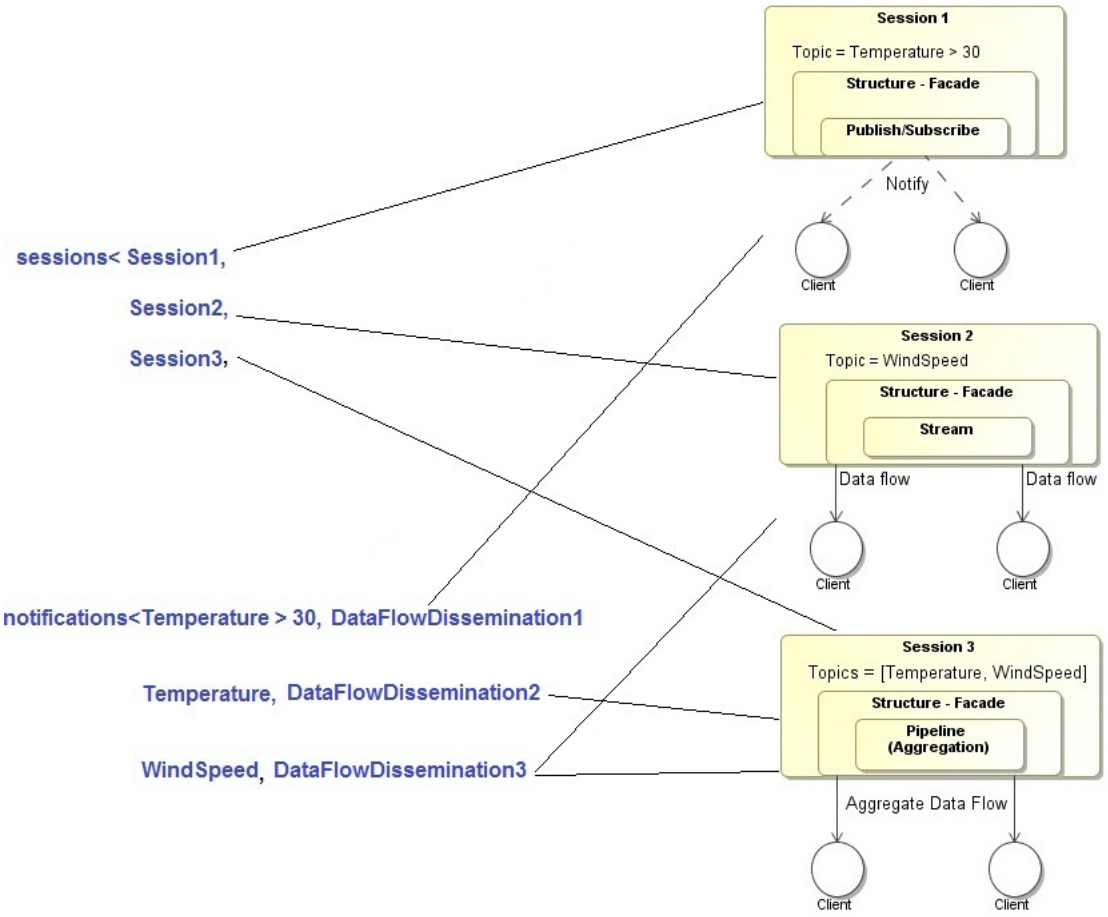


Figure 4.4: Data structures

- Join a session - if the client asks to join a session (by indicating the sessionId), the Session Manager verifies the existence of it in the structure 'sessions'. If the session exists, the client is added, if not, the request is denied;
- Request addressed to a session - if the client requests to perform an operation or a reconfiguration in a specific session, the Session Manager verifies its existence, and if the session is valid, the request is forwarded to session to be dispatched: session.get(Request.getSessionId().dispatchRequest(request));

4.3.2 Session

The entity Session comprises several variables that characterizes its state. Those include:

- The sessionId - this variable is a String that identifies the session in the middleware and it is assigned by the Session Manager in the moment of its creation;
- The owner of the session - this variable identifies the owner of the session which is the client that requested the creation of it. The type of this variable is IComponent and it represents a reference to the client Web service;

- The clients contracted to the session - an HashMap<String,IComponent> components, that contains the clients participating in this session. The key of an entry is a String (userId) that univocally identifies the user in this session. A part from the owner, the clients present in this session are added to it, by requesting the Session Manager to join this session giving the sessionId;
- The topics - a List<String> containing the topics related to this session.
- Session Manager - a reference to the Session Manager;
- A timer - the owner of the session may limit the session lifetime by indicating its duration in moment of creation;
- A Structural pattern combined with Behaviour that will define the interaction model.

4.3.2.1 Dissemination process in the Session

Continuing the dissemination process started by the Session Manager in 4.3.1.1, the next step is to simply forward it to the structure: structure.messageNotification(message).

4.3.2.2 Validation and dispatch of a request in the Session

When the dispatch of a request reaches the 'Session level' (preceded by the Session Manager 4.3.1.2), it means that the operation is within the scope of the session. The first validation to be made is to authenticate the client in this session, by verifying its existence in the structure 'components'. If the client is not a valid user, the request is denied. Otherwise, the valid operations at this level are:

- Finish the interaction - if the request is made by the owner, the session becomes invalid and the other clients are informed through an invalid session notification. The entity Session then notifies the Session Manager to remove it from sessions and notifications structure. If the request was made by a participant client, he is simply removed from the 'components' structure;
- Reconfigure the interaction to another - this reconfiguration request may focus on the topic(s) of the session, in the replacement of the current behaviour to another, or in both. If the request is made by the owner, the session is reconfigured and the other clients are notified of such modification(s) in the interaction model.
  If the request was made by a participant client, he is removed from the session and the reconfiguration request is forwarded to the Session Manager. Once in the Session Manager, it can happen one of two things: if there is already a session with the features asked by the client, he is added to it (similar to the join process). If not, it is created a new session and the client becomes the owner of it. In any of the previous cases, the Session Manager is notified to update the necessary changes in the 'notifications' structure;

- Add interaction - From the point of view of the client, an 'add interaction' request results in the addition of another source to the session. The interaction model that results from this type of request is an Aggregation. In order to do it correctly, the client must provide the additional topic(s) (source(s)), and the aggregation function that will be applied to the data. Subsection 4.3.5 gives more precise information about the implementation of the aggregation interaction model. And again, the semantics of this reconfiguration varies with the type of client that requested it;

- Reconfigure current behaviour - If the requested operation did not match any of the previous ones, than the request is specific to the behaviour. The dispatch is then forward to it: structure.getBehaviour().dispatchRequest(Request);

4.3.3 Structural Patterns

The predominant Structural Pattern that was implemented was the facade. The facade pattern extends the abstract class AbstractStructuralPattern and overrides the definition of three methods (applyBehaviour(), messageNotification(ClientNotificationMessage msg) and public Response dispatchRequest(Request r)). The AbstractStructuralPattern class has two main variables which are the AbstractBehaviouralPattern (the behaviour) and an Map<String,IComponent> componentPlaceHolders, that in practise will be instantiated with the clients. It also includes the definition of a list of methods that are used by the entity Session to manage the interaction with the clients. We will only describe the relevant ones:

- public boolean defineBehaviour(AbstractBehaviouralPattern behaviour) - defines the behaviour;

- public boolean replaceBehaviour(AbstractBehaviouralPattern behaviour) - destroys the current behaviour and replaces with the new 'behaviour';

- public boolean instanciate(IComponent component) - adds a new entry in the componentPlaceHolders structure and the key of that entry is given by component.getUserId();

- public IComponent decrease(String componentId) - removes the component identified by 'componentId';

- public void applyBehaviour() - this is one of the methods that is overrided. Conceptually, this is arguably one the most important methods in the AbstractStructuralPattern class. It is through this method that one can define the relationship (links) between elements. In the case of the facade pattern, its definition is straightforward: to provide (all) the componentPlaceHolders on which the behaviour will be applied: : behaviour.apply(componentPlaceHolders);

- public Response dispatchRequest(Request r) - this method simply forwards the request to the behaviour behaviour.dispatchRequest(r); it continues the dispatch process to the behaviour.

- public void messageNotification(ClientNotificationMessage msg) - this method forwards the request to the behaviour behaviour.dispatchRequest(r); it continues the dissemination process to the behaviour.

In the next subsection we described the implementation of the behavioural patterns.

4.3.4 Behavioural Patterns

Before jumping directly into the implementation, we should first notice that the Client/Server interaction model does not require a Session. Ultimately, this model is constantly present in the interactions between the client and the middleware, in the sense that, every requests, including the ones related to reconfigurations, are based in this direct request/response kind of interaction. As such, we did not found the necessity to reserve its implementation in a behavioural pattern. The user API 4.5 identifies some methods that can be categorized as Client/Server ones.

4.3.4.1 Stream and Producer/Consumer behaviours

The Stream and Producer/Consumer classes extend the abstract class AbstractDataFlowBehaPattern. This abstract class extends another abstract class AbstractBehaviouralPattern, and captures the common characteristics between the two behaviours. It has three protected variables that are:

- Map<Long,Dispatcher> dispatchers - this map contains the dispatchers that are declared during the lifetime of the behaviour. They key of an entry in this structure is a Long that identifies the rate with which the Dispatcher disseminates the messages. This rate is initially defined by the owner of the session, but each client can adjust its own (user API 4.5 demonstrates how). The class Dispatcher contains a reference to the clients (IComponent) that consume at that same rate, and a buffer to save the incoming messages;
- List<String> topics - the list of topics identifying the sources;
- TimerTask ping - a timer that is reset every time the behaviour receives an incoming message from a source. Section 4.4 justifies the use of this timer as a reconfiguration mechanism.

Dissemination process in Stream and Producer/Consumer

When a message arrives in the Stream or Producer/Consumer behaviours, it is diffused to all the dispatchers, that in turn, buffers it for later dissemination in accordance to the specific rate of that Dispatcher.

Validation and dispatch of a request in Stream and Producer/Consumer

The only possible operation that the client may perform in the Stream and Producer/-Consumer behaviour is to change his the rate of dissemination. If the request sent by the client happens to be valid, his rate is updated by removing his reference in the dispatchers structure from the old rate, and add him into the new one (identified by the new rate).

4.3.4.2 Publish/Subscribe behaviour

The Publish/Subscribe behaviour extends the abstract class AbstractBehaviouralPattern and basically has one variable worthwhile mentioning - a private Pattern newBehaviour that represents the interaction model to which the session is automatically reconfigured, once the event related to topic of subscription is triggered.

Dissemination process in Publish/Subscribe behaviour

When the message of notification arrives the Publish/Subscribe behaviour, the session clients are notified about the subscribed event. Subsequently, if the 'newBehaviour' variable is defined (different from null), the Publish/Subscribe behaviour notifies the Session entity that it should be reconfigured to the interaction model defined in 'newBehaviour'. Then the Session entity proceeds to the reconfiguration exactly in same way as stated in 4.3.2.2, concerning the operation of reconfiguring the current interaction to another one.

Validation and dispatch of a request in Publish/Subscribe

It is only possible to change the topic of subscription related to a Publish/Subscribe interaction model. This operation is made by requesting the entity Session to update the subscription (related topic), and by notifying the Session Manager entity to update the notifications structure.

4.3.5 Aggregation

Finally, we explain the architecture of the Aggregation interaction model. We can first acknowledge that an aggregation scenario has several sources. However, the behaviour related to this *'upside down'* facade is not exactly any of the previous ones already identified. The intention is to give to the client, the freedom to confine the criteria of aggregation to be considered in the aggregation process. Therefore, the entity responsible for it is an aggregation function[1] that should be defined by the client (Figure 4.5). Thus, the aggregate data resultant from the application of the aggregation function upon the original

[1]The reader may be mislead to think that the aggregation function is, here, pictured as a behaviour. Instead, it should be seen as a member that constitutes the aggregation process.

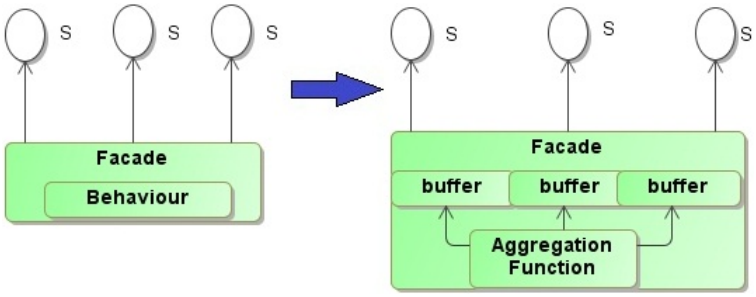


Figure 4.5: Facade with several sources

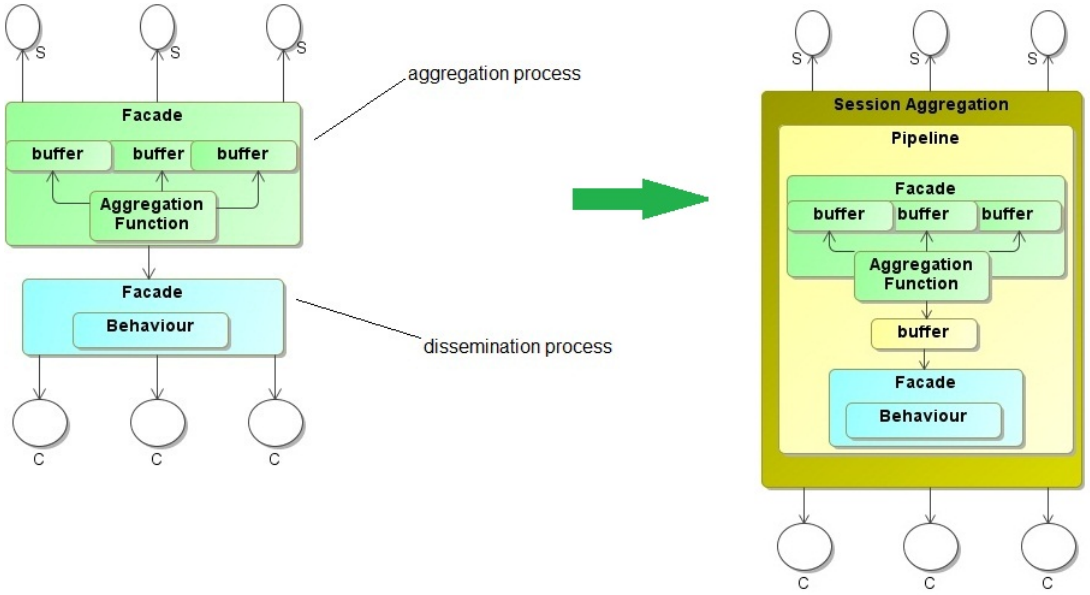


Figure 4.6: Aggregation Session

data, can now be disseminated through the combination of the already known facade, with a behaviour. At last, the context of the interaction is given by the Session entity and transition of data between both facades is done using a pipeline (Figure 4.6).

The previous aggregation architecture was indeed the first model that we considered to implement. Nonetheless, and for code efficiency purposes, considered the following:

- The facade with multiple sources, is already available in the middleware, if we notice that it is simulated from the fact that a session with multiple sources, is added to the respective entries in the 'notifications' structure (recall the Session Manager entity 4.3.1). So, both facades (aggregation and dissemination) can be merged into one;

- The pipeline only adds an indirection level in the dissemination process.

Taking into consideration the previous, the implemented aggregation architecture is pictured in Figure 4.7. The facade with multiple sources (AggregationStructure), extends

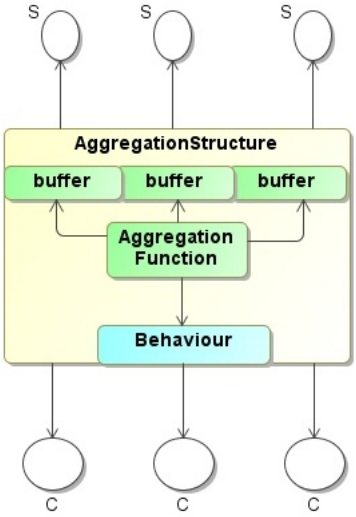


Figure 4.7: Implemented aggregation architecture

the abstract class AbstractStructuralPattern and additionally contains a variable IFunction representing the aggregation function. Also, overrides two methods dispatchRequest(request) and messageNotification(message).

4.3.5.1 Dissemination process in Aggregation

When a message arrives in an AggregationStructure(Figure 4.7), it is buffered in the correspondent queue. Then periodically (with a rate superior to the data transmission from the sources), a dispatcher (*Thread*) applies the aggregation function (IFunction) upon the data and forwards the result to the behaviour responsible for the dissemination process.

4.3.5.2 Validation and dispatch of a request in Aggregation

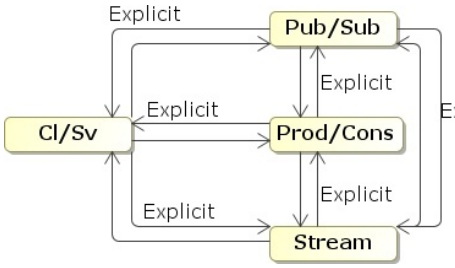
The only possible operation in an aggregation interaction model is to replace the current aggregation function. The dispatcher responsible for the invocation of the aggregation function is put *'on standby'*, the data are kept in the buffers, the aggregation function is replaced, and then the dispatcher is resumed.

4.4 Mechanisms of dynamic reconfiguration

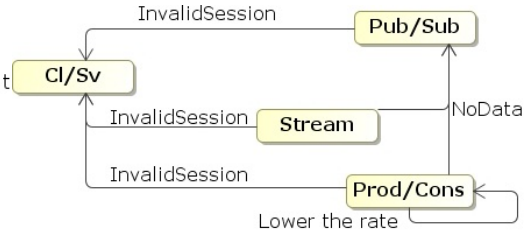
This section introduces the dynamic reconfigurations that we considered to be important, and the mechanisms to implement them.

Figure 4.8 illustrates the state machine that represents the dynamic reconfigurations. For the sake of simplicity, we subdivided it in five to facilitate their explanation.

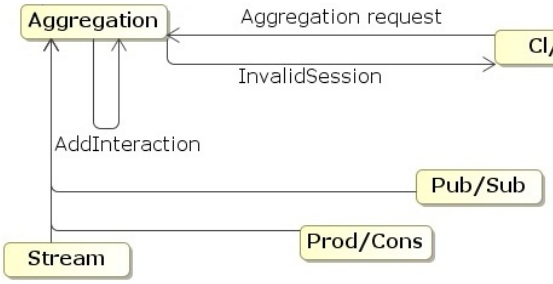
The first one (4.8a) is related to the explicit requests made by an user, to replace the current behaviour. It is possible to reconfigure from any former interaction to another



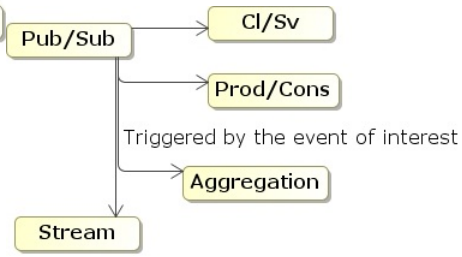
(a) Explicit reconfigurations - Replacement



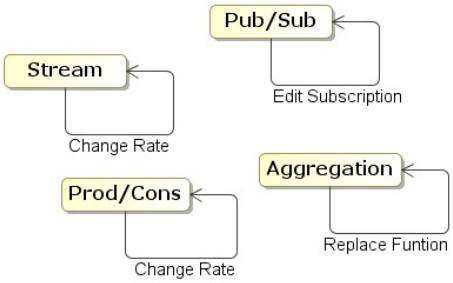
(b) automatic reconfigurations



(c) Reconfigurations - Aggregation



(d) Reconfigurations - Publish/subscribe



(e) Reconfiguration on the current pattern

Figure 4.8: State machine

one. Again, if the reconfiguration request is made by the owner, the session is reconfigured along with the participant users. If it is made by another user, he is reconfigured to another session with those features.

The second state machine 4.8b identifies the reconfigurations that are triggered by the middleware, as a response to the changes in the context of the service, users, or the communication between them. A session becomes invalid if the owner of the session leaves it, if the lifetime of the session expires (recall the Session timer in 4.3.2), or even if the data sources related to a Stream and Producer/Consumer behaviours are unavailable. In respect to the two previous interaction models, if the session related to them suddenly stops receiving incoming data (recall the AbstractDataFlowBehaPattern timer in 4.3.4.1), the middleware 'pings' its source and if it happens to be alive, the Session is reconfigured to Publish/Subscribe related to same topic. If resumes the data transmission, the users are notified.

The transition identified as 'Lower the rate' occurs when the buffer on the client side detects when its storage levels are high, meaning that the client is not capable of consuming the data at the same rate that the middleware sends them. In this situation, the buffer itself sends a request to the middleware to lower the specific rate of this client.

The third state machine is related to the Aggregation interaction model. An user may request for an aggregation session indicating the sources, the aggregation function and the behaviour responsible for the dissemination of the aggregate data. Once in a Stream, Producer/Consumer or Publish/Subscribe behaviours, the user may add additional sources to it and the resultant interaction model is an Aggregation. He should also define the aggregation function and the additional sources to be added.

The fourth state machine (4.8d) respects the automatic reconfigurations related to the Publish/Subscribe interaction model. When a client starts a session with a Publish/Subscribe behaviour, he can also define an interaction model to which the session is automatically reconfigured, when the event related to the topic of subscription is triggered.

The last state machine concerns the specific operations that are possible to perform in a certain interaction model. They include the change of the rate of dissemination in Stream and Producer/Consumer behaviours. The replacement of the aggregation function in the aggregation interaction model and finally the edit of the subscription of a Publish/Subscribe behaviour.

4.5 User API

This Section is devoted to the description of the implemented user API, giving some examples to help the reader to understand its usage. We will subdivide it in three sub sections. The first will give special focus the creation of a session, the second on the reconfiguration and the third will be dedicated to the methods that do not fall in any of the previous categories. The two main entities that are represented in this API, are the Session and the interaction model related to it. The combination of both will characterize the way the user interacts with the service. The diagram class of the user API is shown in Figure 4.9.

4.5.1 The creation of a session

The class ClientSession represents a session on which the user may access the services provided by the middleware and can be parametrized with one of the interaction models (the behavioural patterns). It has three alternative constructors for different purposes. The main constructor ClientSession(int sessionDuration, IPattern p), receives as arguments the duration of the session and the behaviour. Another two variants are provided: one that does not impose any time constraints public ClientSession(IPattern p); and the other

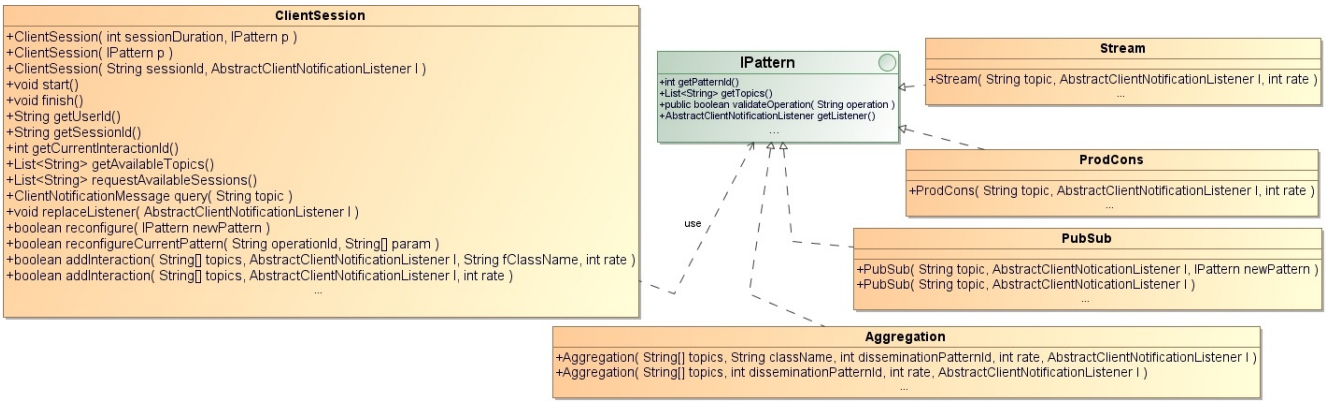


Figure 4.9: Diagram class of the user API

one ClientSession(String sessionId, AbstractClientNotificationListener l), that receives as argument the identifier of a session and a listener. The sessionId argument identifies the session, to which the user pretends to participate in. The l argument defines a listener responsible for the process of incoming data. At the end of this subsection, we will give further information about the programming of a listener.

Next, we briefly describe the constructors of each of the behavioural patterns.

- Stream(String topic, AbstractClientNotificationsListener l, int rate) - Defines a Stream interaction model related to the source topic, with listener l and constant dissemination rate rate;
- ProdCons(String topic, AbstractClientNotificationsListener l, int rate) - Similar to Stream, with the difference that, the interaction model is dictated by a Producer/-Consumer behaviour;
- PubSub(String topic, AbstractClientNotificationsListener l, IPattern p) - Defines a Publish/Subscribe interaction model, related with the subscription topic and with a pattern of automatic reconfiguration p. The PubSub class has another constructor (String topic, AbstractClientNotificationsListener l, that does not define any automatic reconfiguration, once the event related to the subscription is triggered;
- Aggregation(String[] topics, String className, int disseminationPatternId, int rate, AbstractClientNotificationsListener l) - Defines an aggregation interaction model, and receives as arguments: an array of strings - topics, the name of the aggregation function - className, the rate - rate and the listener - l. The aggregation function is a Java class, and the location from where its file is read, is parametrizable. One should notice that, there must be a match between the requested sources, with the ones referenced in the aggregate function. This class must implement the type IFunction, and consequently define the method NotificationMessage[] apply(), that is invoked to proceed the aggregation process (recall the aggregation architecture process in 4.6). To simplify the programmer's effort, the API features an abstract class

AbstractAggregationFunction that has already defined several protected methods to be called in the middleware, for initialization purposes. Also, enables him to access the messages of a certain source, by using the method protected NotificationMessage getNextMessage(String topic). If the user invokes this method with an absent topic, the method returns null.

We give an example of a possible aggregate function related to the data sources of "Temperature" and "Humidity". In the example, it is determined that for each two messages of "Temperature", it should be return one of "Humidity".

```
1  public class AggregationFunction extends AbstractAggregationFunction{
2    public  AggregationFunction(){}
3
4    public NotificationMessage[] apply(){
5      NotificationMessage[] msgs = new NotificationMessage[3];
6      msgs[0] = getNextMessage("Temperature");
7      msgs[1] = getNextMessage("Temperature");
8      msgs[2] = getNextMessage("Humidity");
9      return msgs;
10   }
11 }
```

One final note concerning the Aggregation class: if the user does not specify an aggregation function (by using the constructor Aggregation(String[] topics, int disseminationPatternId, int rate, AbstractClientNotificationsListener l)), the criteria of data aggregation will be, by default, to simply forward the data;

Finally, the data processing is delegated to the listener declared in the definition of the behaviour. The user should extend his listener from the AbstractClientNotificationListener class, and must implement the method processMessage(ClientNotificationMsg msg). This method will be invoked to treat the incoming messages. Apart from it, the user must also define a list of methods related to notifications of reconfiguration, that are sent from the middleware. Therefore, the complete list of methods that he must define, and the reasons behind the invocation of each one of them, are the following:

- public void processMessage(ClientNotificationMsg msg); - invoked by the fact that a message has arrived;
- public void InvalidSessionNotification(InvalidSessionException e);- The session has become invalid;
- public void NoDataNotification(NoDataException e);- There is no available data;
- public void SameFlowNotification(OwnerSameFlowException e);- The session has been reconfigured but the topic remains the same;
- public void DiffFlowNotification(OwnerDiffFlowException e);- The session has been reconfigured and the topic has changed;

- public void PubSubNotification(PubSubReconfException e);- The session has been automatically reconfigured to the reconfiguration pattern associated to publish/subscribe behaviour;

- public void OwnerAddInteractionNotification(OwnerAddInteractionException e);- The session has been reconfigured because it was added a new source of data do the interaction;

In addition, the method void replaceListener(AbstractClientNotificationListener l) replaces the current listener.

Following a common approach in Java, the creation of a ClientSession only builds the object, but does not start its execution. For that reason, we supplied the method start. This method initiates a session that was previously declared. It should be used together with one of the previous constructors. Consider the following examples:

```
1  //Definition of the Session with behaviour Stream
2  ClientSession s1 = new ClientSession(10,new Stream("WindSpeed", listener, 4));
3  //Iniciates the Session and the interaction
4  s1.start();
5
6  //Definition of the Session without time limit.
7  ClientSession s2 = new ClientSession(new Stream("Temperature", listener, 4));
8  //Iniciates the Session and the interaction
9  s2.start();
10
11 //Declares a session, identified by "SessionId"
12 ClientSession s = new ClientSession("SessionId",new ClientNotificationListener());
13 //Requests the middleware, to join the session
14 s.start();
```

When the user attempts to join a session that does not exist, the start() method throws an InvalidSessionException. The method finish() ends the current interaction model between the user and the middleware and the user is removed from the respective session (in the middleware).

4.5.2 Reconfiguration methods

The methods boolean reconfigure(IPattern p) and public boolean addInteraction(String[] topics, AbstractClientNotificationListener l, String className, int rate) perform reconfiguration at the current behaviour. They are intimately related with both state machines 4.8a and 4.8c, described in Section 4.4. The first one enables de client to explicitly reconfigure the current behaviour to another (p). The second one, and from the point of view of the user, adds more sources do the current interaction. By doing so, the resulting interaction model is an Aggregation. Again, it is possible for the user, to not define an aggregate function, by using the variation of this method public boolean addInteraction(String[] topics, AbstractClientNotificationListener l, int rate)).

The method boolean reconfigureCurrentPattern(String operation, String[] params) reconfigures the current pattern (recall the state machine 4.8e), but the permitted reconfiguration operations are limited to the behaviour. It is possible to change the dissemination rate of a session with behaviour Stream or Producer/Consumer. The user may also change the topic of subscription related to a Publish/Subscribe pattern or even replace the aggregation function in a aggregation interaction model. Each of these operations can be made in the following way:

```
1  //Change the rate of a session related do Stream or Producer/Consumer
2  ClientSession s1 = new ClientSession(new Stream(topic,listener,rate));
3  s1.start();
4  s1.reconfigureCurrentPattern("ChangeRate",newRate);
5
6  //Edit the topic of subscription related to a Publish/Subscribe session
7  ClientSession s2 = new ClientSession(new PubSub(topic,listener,newPattern));
8  s2.reconfigureCurrentPattern("EditSubscription",newTopic);
9
10 //Replace the aggregation function
11 Aggregation agg = new Aggregation(topics, className, disseminationPatternId, rate,
12  listener);
13 ClientSession s3 = new ClientSession(agg);
14 s3.start();
15 s3.reconfigureCurrentPattern("ReplaceFunction",newClassName);
```

4.5.3 The static methods and session state methods

The remaining methods of the class Session can be categorized as static, or related to the its state. The static methods, can be seen as being related to the Client/Server interaction model, since its execution occur in a direct request/response fashion with the middleware. The static methods are:

- List<String> requestAvailableSession() - returns a list of identifiers of the sessions that currently exist;
- List<String> requestAvailableTopics() - returns a list of the currently available topics;
- ClientMessage query(String topic) - it permits the user to query the value registered on certain topic;

The state methods are:

- String getUserId() - retrieves the current identifier of the user in the session;
- String getSessionId - returns the identifier of the session;
- public int getCurrentInteractionId() - return the current identifier of the behaviour;

5

Evaluation

In this chapter, we will make the functional analysis of the middleware. Firstly, we will deepen the effects of the reconfiguration process in the session and listener objects. Secondly, we will describe the fire department example and how it is translated into the user API. We will also detail the transitions that occur during this scenario, both at the middleware and the clients, to give a better understanding of how the system evolves. Finally, we present the output of execution of the example.

5.1 Effects of the reconfiguration process

To fully understand, how the reconfiguration process should be properly handled (either by an owner of a session, or by a participative user), we should first give more details about its effects. When a client receives a notification of reconfiguration, two objects are subject to such modification - the ClientSession object and the Listener object.

5.1.1 The ClientSession object

On one hand, the notification of reconfiguration activates a flag in the ClientSession, indicating that the session has been reconfigured. When this reconfiguration flag is activated, there is a method on the ClientSession object, that the user should be careful with: boolean reconfigureCurrentPattern(String operationId, String[] param). The invocation of this method in a reconfigured session may fail, due to the lack of precaution by the user, concerning the current state of the session. Consider the following example:

```
1  Stream str = new Stream("Temperature", listener, rate);
2  PubSub ps = new PubSub("Temperature > 80",listener,str);
3  ClientSession s = new ClientSession(ps);
4  s.start();
5
6  //Moments later
7  ....
8  ..
9  try{
10   s.reconfigureCurrentPattern("EditSubscription", "Temperature > 50");
11 }catch(ReconfiguredSessionException e){
12   //For example, add another stream
13   if (s.getCurrentInteractionId() == stream)
14     s.addInteraction("WindSpeed", listener, aggregationFunction, rate);
15 }
```

The example demonstrates that, after starting the session (line 4), the user decides that instead of reconfiguring to stream when "Temperature > 80", he opts to reconfigure it when "Temperature > 50", by editing the subscription (line 10).
This reconfiguration may fail or succeed, depending on the current state of the session. If it has already been reconfigured to stream, the edit of the subscription would not make sense and consequently fail, because the middleware would invalidate the operation.
In order to circumvent this miss perception by the user, we decided that this method throws an exception (ReconfiguredSessionException), when the reconfiguration flag is activated and if the operation is not coherent with the current behaviour. It is the responsibility of the user, to treat this exception accordingly, so that his operation may succeed.
In the previous code sample, if the session was already related to Stream behaviour, the ReconfiguredSessionException is thrown and the user should only perform reconfigurations that make sense in that behaviour. In this case, he added another stream related to "WindSpeed" (line 14). If still, he insisted to change the subscription, the request would be made to the middleware, the operation would be properly invalidated, and the method would return false.

5.1.2 The Listener object

On the other hand, the notification of reconfiguration is notified in the Listener object, by invoking the properly method that represents the kind of transition that was made. For example, if a session becomes invalid for whatever reason, it is invoked the method:public void InvalidSessionNotification(InvalidSessionException e). It is up to the user, to program how he wishes to proceed in this situation.

Consider a scenario, where the user starts a session of stream about temperature and, suddenly, there is no available data from the temperature sensors. In that case, the middleware would send a notification of reconfiguration to the client, informing that the

session would be reconfigured to Publish/Subscribe. If the source was to resume the data transmission, the client would be notified.
But imagine that, the user chooses to reconfigure the session, to a stream upon the topic "Humidity", instead of waiting for temperature data to be available. He can program this reconfiguration in the Listener, as follows:

```
public class MyTemperatureListener extends AbstractClientNotificationListener{
  public myTemperatureListener(){}
  public void processMessage(ClientNotificationMsg msg) {..}
  public void DiffFlowNotification(OwnerDiffFlowException e) {..}
  public void InvalidSessionNotification(InvalidSessionException e){..}
  public void OwnerAddInteractionNotification(OwnerAddInteractionException e){..}
  public void PubSubNotification(PubSubReconfException e){..}
  public void SameFlowNotification(OwnerSameFlowException e){..}
  public void NoDataNotification(NoDataException e){
    getSession().reconfigure(new Stream("Humidity",new MyHumidityListener(),rate));
  }
}
```

The method NoDataNotification(NoDataException e) is invoked when the event about the unavailability of the data occurs. The method protected ClientSession getSession() returns the reference of the ClientSession object, related to this listener. Also, the user must point this listener, in the definition of the Stream behaviour interaction. Finally, and if he wishes, he can define another Listener object (MyHumidityListener in the code sample), responsible to process the data related to humidity:

```
Stream str = new Stream("Temperature",new MyTemperatureListener(),3);
ClientSession s = new ClientSession(str);
s.start();
```

Taking into consideration, the effects of the notifications previously explained, on the objects of ClientSession and Listener, we describe next, the motivating example of this thesis.

5.2 The fire detection example

Recalling the fire example described in 1.1.1, consider now the concrete scenario:
The main fire department responsible for a forest, is interested in receiving the notification, of the event that the temperature raised to values above 50. (5.1 - line 10)
If this notification is received, they would automatically want to reconfigure from this alert state, to a critical one, that contemplates the raise of the temperature above 80. (5.1 - line 9)
Based on this last notification, which probably indicates an imminent scenario of fire, the next logical step is to obtain in real time, the data about the registered temperature. Beyond this, it would be also helpful to receive data about the "WindSpeed" and "WindDirection"; and preferentially, the data would be aggregated according to their criteria.(5.1 - lines 6 and 5)
We may also remember that, a secondary fire department chosen to help the first one, would benefit

from the access to the same data. (5.2)
Furthermore, during the fire fighting, the first fire department decides to add another source of data - "Humidity", to gain a more precise information about the conditions in the terrain.(5.1 - line 18)
The stated example can be directly translated to java code, of the implemented user API, as follows:

Listing 5.1: The main fire department

```
1    AbstractClientNotificationListener ownerListener1 = new OwnerClientListener();
2    AbstractClientNotificationListener ownerListener2 = new OwnerClientListener2();
3    AbstractClientNotificationListener ownerListener3 = new OwnerClientListener3();
4    //Main fire department
5    String aggregationF = "reflection.AggregationFunction";
6    String[] topics = new String[]{"Temperature","WindSpeed","WindDirection"};
7    Aggregation aggregate = new Aggregation(topics,aggregationF,
8        PatternID.Stream,3,ownerListener3);
9    PubSub critical = new PubSub("Temperature␣>␣80",ownerListener2,aggregate);
10   PubSub alert = new PubSub("Temperature␣>␣50",ownerListener1,critical);
11   ClientSession s = new ClientSession(alert);
12   s.start();
13   ...
14   ..
15   .
16   //Moments later, during the fire fighting
17   AbstractClientNotificationListener ownerListener4 = new OwnerClientListener4();
18   s.addInteraction(new String[]{"Humidity"}, ownerListener4, 3);
```

Listing 5.2: The secondary fire department

```
1    //Secondary fire department
2    ClientSession s = new ClientSession("Session1",new ClientNotificationListener());
3    s.start();
```

In the next subsections we will describe the transitions that happen during the fire scenario, from the point of view of the three entities involved (middleware, first and second fire departments). We will see that, each of the departments will have to adapt to the transitions in different ways.

5.2.1 Transitions in the middleware

The entities that are created, and the automatic reconfigurations that take place in the middleware, are illustrated in figure 5.1. The Session Manager creates a new Session, combining the Facade structure with the Publish/Subscribe behaviour, and defines the main fire department as the owner. The topic related to this session is "Temperature > 50". Meanwhile, the second fire department joins the same session.
Once the temperature raises to values above 50, the middleware sends notifications of the event to the respective subscribers. Likewise, are sent other notifications informing that

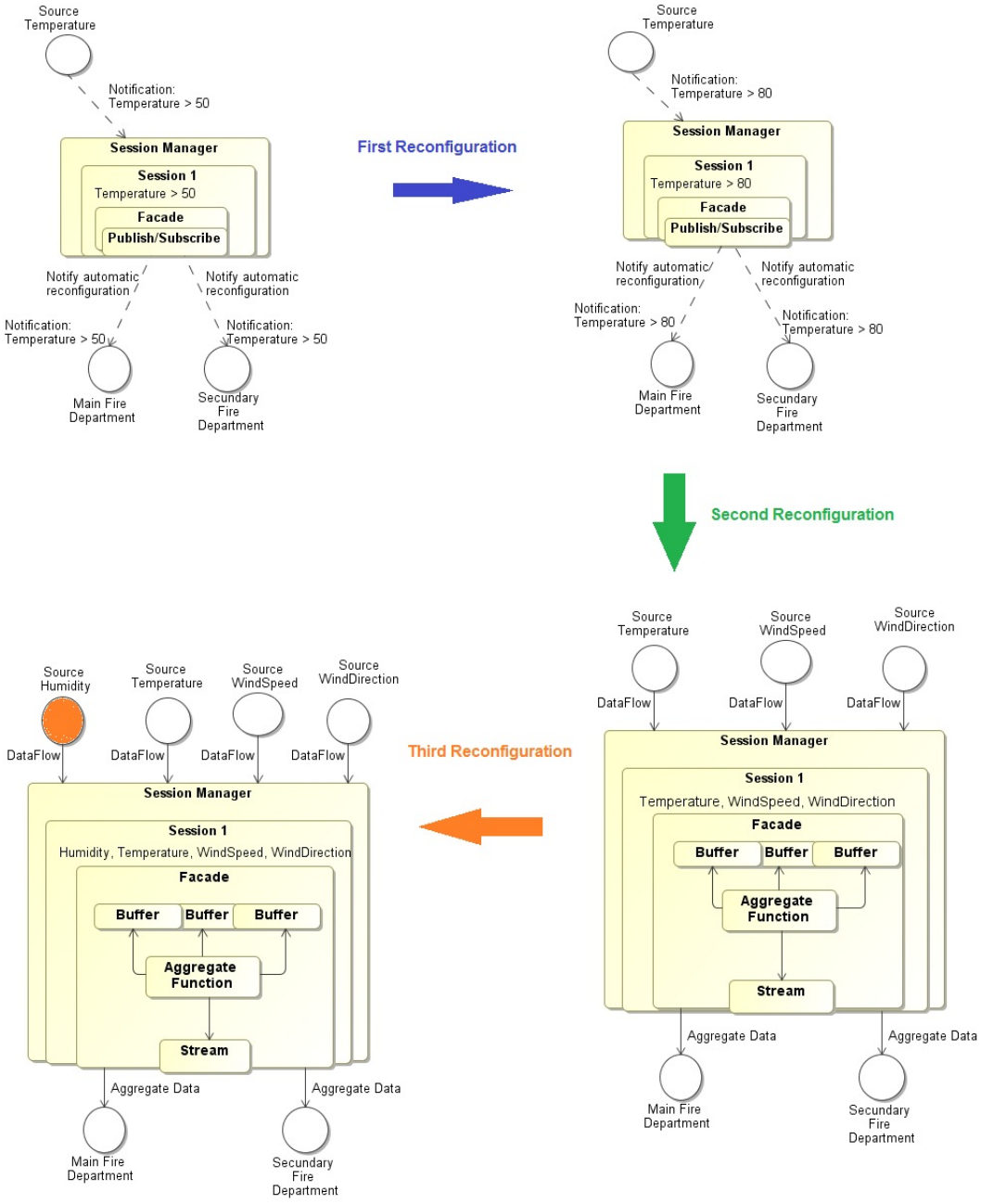


Figure 5.1: Entities created by the Session Manager

the current session is going to be reconfigured to another instance of a Publish/Subscribe behaviour, with the new topic "Temperature > 80" (first reconfiguration).
The second reconfiguration happens in a similar process to the first, with the difference that the new interaction model to which the session will be reconfigured, is now the Aggregation of streams upon "Temperature", "WindSpeed" and "WindDirection". On the other hand, the event that triggered it was the fact that the temperature raised above 80.
Finally, the primary fire department requests an additional stream of "Humidity", resulting in a new source related to the session (third reconfiguration).

5.2.2 Transitions in the fire departments

The first fire department, who created the session (and thereby, becomes the owner of it), defines the way the incoming messages are processed, by pointing the correspondent listeners, in every declaration of the behaviours. In the java code of the user API, the declared listeners are the OwnerClientListener 1 to 4, to distinctively treat the data respecting the different interaction models.
The second fire department adapts to those transitions, by consecutively replacing the current listener of the session, in each notification of reconfiguration sent by the middleware. Analysing these reconfigurations, we notice that:

- The first transition is made, from an automatic reconfiguration related to Publish/-Subscribe pattern;
- The second transition is also made, from an automatic reconfiguration related to Publish/Subscribe pattern;
- The third transition happens, from the fact that, the owner of the session added another stream interaction;

So the respective methods invoked upon the present listener, in each of the reconfigurations are:

- *public void PubSubNotification(PubSubReconfException e)* - the transition from publish/subscribe on the topic "Temperature > 50";
- *public void PubSubNotification(PubSubReconfException e)* - the transition from publish/subscribe on the topic "Temperature > 80";
- *public void OwnerAddInteractionNotification(OwnerAddInteractionException e)* - the transition that occurred during the aggregation session, when the primary fire department (owner) added another stream of humidity;

This results in the declaration of different Listener objects as we illustrate in figure 5.2. At last, the first listener (*ClientNotificationListener*) is used as argument of the method *join(sessionId,listener)*, as previously demonstrated in 5.2 - line 2.

Figure 5.2: Declaration of listeners by the secondary fire department

5.2.3 Output of the execution of the example

```
.............
.......
....
Thread.sleep(60000);
//Moments later, during the fire fighting
AbstractClientNotificationListener ownerListener4 = new OwnerClientListener4();
s.addInteraction(new String[]{"Humidity"}, ownerListener4, 3);
```

Figure 5.3: Thread.sleep(60000)

We must first enlighten the reader, of the necessary changes in the code for the fire scenario. First, we had to introduce a line of code into the first fire department example, to simulate the period of time between the request for the notification and the imminent fire situation - figure 5.3.
Second, the programming of the listeners was to consecutively replace them, as already illustrated in figure 5.2; and also, to print in the console the correspondent notification method that was invoked. Finally, we printed the content of the messages in the method *processMessage(Msg)*, along with the listener that processed it.
And last, the programming of the aggregate function was simply to merge the content of each of the messages of the different sources, into a single one.

From the output in figure 5.5, we can see that both departments receive the notifications of reconfiguration, related to the Publish/Subscribe pattern. It is strictly necessary that also the owner receives it, because he does not control the instant that those reconfigurations happened. He needs to be notified, so that he can replace his listener, at the appropriate time. For the remaining reconfigurations explicitly requested by the owner, he does not need to receive the corresponding notifications, from the middleware. Finally, we can notice that, when the owner requested for another stream of humidity, the incoming data was not aggregated. That is justified from the fact that, he did not define any aggregation function, when he invoked the method *addInteraction(....)*. The result from that, is the forwarding of the messages.

Concluding the evaluation section, we are aware that in the fire detection example, the second fire department (as a participative user) will need to know, the exactly type and order of the reconfigurations, that occur at execution time. This is strictly necessary, if he wants to affect different listeners, to each data type. However, one alternative to tackle this problem, is to define a sufficiently capable listener, of treating all the different kind of messages that he may receive, during the lifetime of the session.

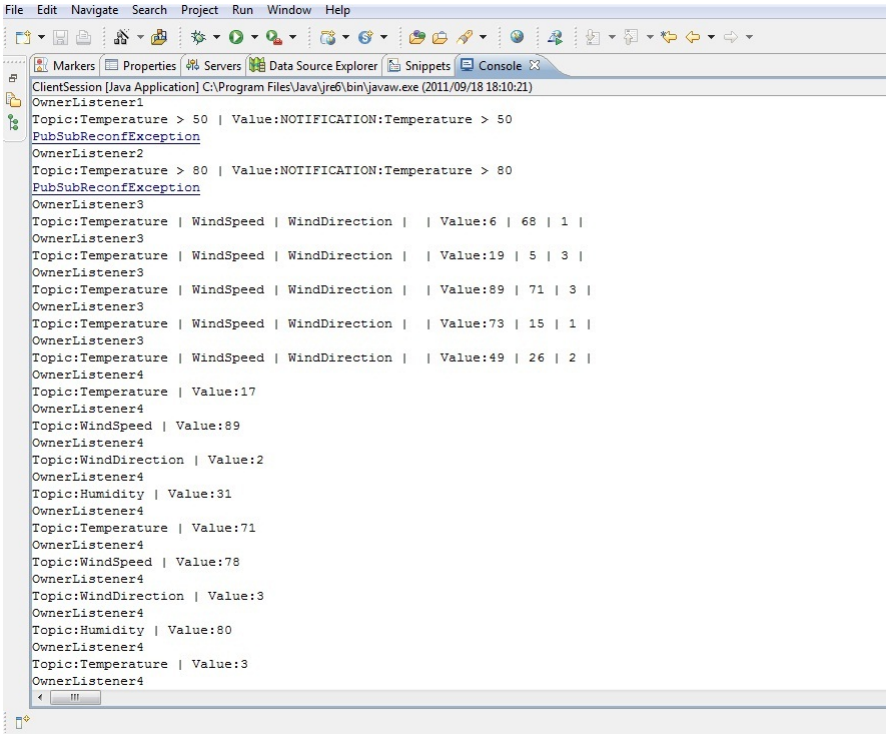


Figure 5.4: Output of the main fire department - owner

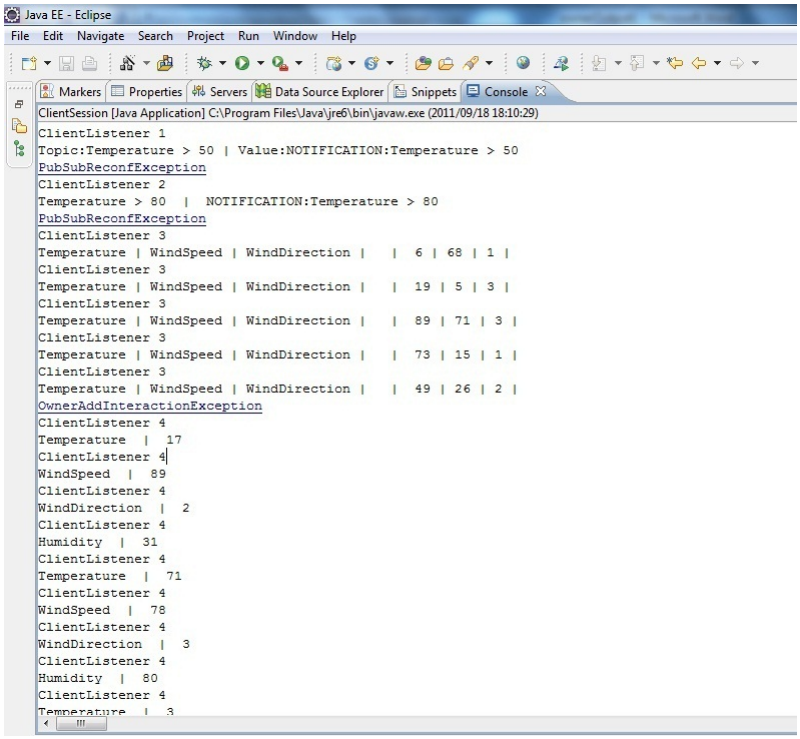


Figure 5.5: Output of the second fire department - participative user

6

Conclusions and Future Work

The current high pressure on real-time and precise large-scale environmental monitoring relates to, among other concerns, the need to inspect and predict extreme meteorological events. Namely, one major challenge for science today is predicting green-house effect consequences like large-scale flooding [clc], hurricanes, severe droughts, etc. Likewise, simulations for fire evolution predictions are of utmost relevance due to their social and economical impact, and frequent occurrence. Simulation applications for these kind of events demand hence the access to different types of data collected across wide scale geographic areas, and for long periods of time, which may include real time data. Only large amounts of diverse data support more precise information extraction and knowledge concerning a better evaluation of such complex events.

To this extent, Wireless Sensor Networks (WSNs) offer a good low cost solution for such large-scale environmental monitoring. WSNs allow hence the development of more or less complex applications [GHIGGHPD07] to which the interaction with the real world is a pressing requirement. Such includes more traditional scientific applications, e.g. like the cited meteorology or earthquake prediction simulations, but also the support to new types of applications. This is the case of Participatory Sensing applications, e.g. urban traffic management or for virtual communities' support, which rely on data acquisition and dissemination through mobile devices [CEL+08].

Nevertheless, one disadvantage of WSNs is still their low-level limited interfaces. To this extent, abstracting WSNs as Web services [BPRD08, PS11] support their inclusion in Web environments, e.g. in the context of business processes. Namely, the service paradigm via standard Web technologies supports an uniform and simple access to WSNs, their parametrization and aggregation, with the possibility of systematising the access to the collected data.

Simultaneously, the service paradigm in general is becoming increasingly relevant across different domains, either commercial, technological or scientific. For instance, the perceived current trend on making everything accessible as a service (*XaaS*)) builds on the concept of service as a simple but powerful abstraction for systems' interaction and integration. Examples range from, a) the *Internet of Things* [ITU05] using services as the main abstraction for the simple integration of objects and entities, either real or virtual, within intelligent contexts; to b) *Cloud computing* aiming at providing standard solutions for different types of service access (example of a contribution in this domain [BGPCV11]).

The service paradigm represents accordingly a way to provide uniform access and aggregation of entities with different characteristics and at different levels of the cyber-infrastructure. Particularly, this paradigm may contribute to the building of an *integrated middleware vision*, e.g. supporting diverse complex networked embedded applications.

Considering the WSNs context in particular, network accesses are typically best represented through stateful services implying more complex interactions between a service and its users, besides the traditional response request access. Examples of such interactions are event subscription and notification models or stream-based data dissemination.

Consequently, new models and solutions for service interaction are required which include the support for dynamic adaptation mechanisms. These may comprehend, for instance, support for fault-tolerance concerns (e.g. related with service or interaction medium failures) or allowing users to directly and dynamically control and modify such interactions (e.g. dynamic selection/replacement of the used interaction model with a service).

6.0.4 Work Contributions

In order to allow richer and dynamic interaction models to WSNs interfaced via Web services, this work described a solution based on the concept of *Design Patterns* to build a middleware supporting such interactions. These include data access via Behavioural Patterns like Producer/Consumer, Streaming, Publish/Subscriber, or dynamic aggregation of such collected data from distinct networks. Moreover, dynamic changes assisted by the middleware may take into account

a) the context of the provided service (e.g. service availability, or application specific characteristics like the current subscribed topics to particular WSNs);

b) the context of the interaction medium among the service and its users (e.g. the current quality of service of the communication); or

c) the context of these users (e.g. users accessing the service through a mobile device with limited autonomy or processing power).

To this extent, the solution uses the concept of a *Session* to capture the characteristics relating a set of users accessing the same particular service with the same interaction model. On one hand, the owner of a session may, at run-time, explicitly change the

interaction model to the service, being this automatically perceivable by the other users in the same session. On the other hand, the dynamic modification of the used interaction model within a session may be triggered by the middleware itself and according to a state machine following a set of pre-defined rules. This provides limited forms of coping with failures or guaranteing some consistency characteristics as a result of changes in the context of a session (which comprises the service, its users, and the communication medium, as described above).

Aiming at building such environment the contributions of this thesis were,

1. Usage of the pattern abstraction as a mean to implement richer interactions between clients and stateful Web services like

   - rate configuration of both the Streaming and Publish/Subscriber patterns. This allows different clients (also within the same session) to process data at their own adequate pace, reducing data loss;
   - implementation of the Producer/Consumer pattern allowing an extra level on data availability and delivery, whenever the rates between the service provider and its users are different;
   - implementation of data stream aggregation with flexible user parameterization on the aggregation function to be applied upon such acquired data.

2. Pattern-based dynamic reconfiguration supporting changes within each interaction model. The possible dynamic reconfigurations are conform to each interaction model's semantics, e.g. increasing the number of consumers or adding more data streams to a stream aggregation or replacing the aggregation function.

3. System dynamic evolution as a result of context-based dynamic replacement between behavioural/coordination patterns. The possible substituitions are captured in state-machines guaranteing that only the adequate transitions are performed. This contributes hence to system consistency.

4. Implementation of the session abstraction which includes all service clients sharing the usage of the same interaction model to the same service. This simplifies adding new clients requiring the same interaction characteristics to this service by simply requesting the join to an identified session. Moreover, all users are notified in the same way whenever events are generated in the context of the session they all belong to. Finally, whenever the interaction model changes automatically or is requested to change (by the session owner) within the context of that session, all clients are notified and can adequately respond to this dynamic reconfiguration.

5. Definition of the system's architecture and middleware prototype implementation incorporating the above characteristics, in the context Web enabled WSNs represented by the Senser system [PS11].

6.0.5 Critical Evaluation

The performance evaluation in terms of the overhead of one additional middleware layer between a service and its users is one point that unfortunately is missing in this dissertation. Likewise, it would have been interesting to implement more application scenarios in order to evaluate the expressiveness of the model.

However, due to significant complexity of the proposed model and its implementation, major validations like the above were not possible in the limited frame time of a typical MsC dissertation, and are deferred to future work. Nevertheless, the major ideas were already validated in a publication in Inforum, on September 2011 [BPG11], with a positive feedback about the proposed solution. Also, recently the submission of another paper entitled: "Session-based Dynamic Interaction Models for Stateful Web Services", was accepted in the International Conference on Exploring Services Sciences (IESS 2012), contributing towards statefuless of today's services paradigm.

6.0.6 Future Work

The novel work described in this thesis concerning richer interaction models for Web enabled WSNs opens several interesting further developments concerning this context. Additionally, the resolution of several open problems which was not possible to cover in this thesis will provide a richer API for application construction in this domain. Namely,

- Possibility of accessing other Web services beside the ones supported by the Senser platform.
- Aggregation of several service interactions with the characteristics described in this thesis in the form of workflows. Such allows capturing dependencies for instance in terms of data flowing among different services.
- Additional dynamic adaptation capabilities concerning QoS issues like security and reliability.
- Adaptation of the model to Stateful Web Services in other domains like Cloud computing.
- Dynamic change of the interface access to each pattern's specific methods
- Support the dynamic definition of the additional classes which can parameterise the aggregation function in the aggregation interaction model.
- Enable the client to upload the classes, which do not exist in the service side (middleware), and are used in the aggregation function.

These are just a few of the possible extensions to this work which, in our opinion, pave the way for an interesting approach towards dynamic adaptation on service access.

Bibliography

[ADK09] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Towards hierarchical management of autonomic components: A case study. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2009*, pages 3–10. IEEE Computer Society, 2009.

[AHS06] Karl Aberer, Manfred Hauswirth, and Ali Salehi. A middleware for fast and flexible sensor network deployment. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1199–1202. VLDB Endowment, 2006.

[ASSC02] Lan F. Akyildiz, Welljan Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A survey on sensor networks. *Communications Magazine, IEEE*, 40(8):102 – 114, aug 2002.

[Bac07] Sam Bacharach. New implementations of ogc sensor web enablement standards, December 2007.

[Bar] Douglas K. Barry. Web services explained. http://www.service-architecture.com/web-services/articles/web_services_explained.html.

[BGPCV11] Lee Badger, Tim Grance, Robert Patt-Corner, and Jeff Voas. Cloud computing synopsis and recommendations (draft), nist special publication 800-146. Technical report, Recommendations of the National Institute of Standards and Technology, 2011.

[BPG11] Adérito Baptista, Hervé Paulino, and Cecília Gomes. Reconfiguraçao dinâmica de redes de sensores. *INFORUM 2011, Coimbra*, September 2011.

[BPK+06] Viraj Bhat, Manish Parashar, Mohit Kh, Nagarajan K, and Scott Klasky. A self-managing wide-area data streaming service using model-based

online control. In *in Proc. 7 th IEEE Int. Conf. on Grid Computing*, pages 176–183, 2006.

[BPRD07] Mike Botts, George Percivall, Carl Reed, and John Davidson. Sensor web enablement:overview and high level architecture. Technical Report OGC 07-165, Open Geospatial Consortium Inc. (OGC), December 2007. version 3.

[BPRD08] Mike E. Botts, George Percivall, Carl Reed, and John Davidson. OGC sensor web enablement: Overview and high level architecture. In Silvia Nittel, Alexandros Labrinidis, and Anthony Stefanidis, editors, *GeoSensor Networks, Second International Conference, GSN 2006, Boston, MA, USA, October 1-3, 2006, Revised Selected and Invited Papers*, volume 4540 of *Lecture Notes in Computer Science*, pages 175–190. Springer, 2008.

[CDW04] Diane Cook, Sajal Das, and John Wiley. *Smart Environments: Technology, Protocols and Applications*. Wiley-Interscience, 2004.

[CEL+08] Andrew T. Campbell, Shane B. Eisenman, Nicholas D. Lane, Emiliano Miluzzo, Ronald A. Peterson, Hong Lu, Xiao Zheng, Mirco Musolesi, Kristóf Fodor, and Gahng-Seop Ahn. The rise of people-centric sensing. *IEEE Internet Computing*, 12:12–21, July 2008.

[CKM+03] Francisco Curbera, Rania Khalaf, Nirmal Mukhi, Stefan Tai, and Sanjiva Weerawarana. The next step in web services. *Commun. ACM*, 46(10):29–34, 2003.

[clc] Bbc news: "climate change raises flood risk, researchers say". http://www.bbc.co.uk/news/science-environment-12484314.

[Fie00] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[FKNT02] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, June 28 2002.

[FZRL09] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud Computing and Grid Computing 360-Degree Compared. *ArXiv e-prints*, December 2009.

[GHIGGHPD07] Carlos F. García-Hernández, Pablo H. Ibargüengoytia-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. Survey.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison-Wesley Professional, 1994.

[GRC08] Cecilia Gomes, Omer F. Rana, and Jose Cunha. Extending grid-based workflow tools with patterns/operators. *Int. J. High Perform. Comput. Appl.*, 22:301–318, August 2008.

[Hay08] Brian Hayes. Cloud computing. *Commun. ACM*, 51(7):9–11, 2008.

[HlFcRL06] Gregory Hackmann, Chien liang Fok, Gruia catalin Roman, and Chenyang Lu. Agimone: Middleware support for seamless integration of sensor and ip networks. In Phillip B. Gibbons, Tarek F. Abdelzaher, James Aspnes, and Ramesh Rao, editors, *Distributed Computing in Sensor Systems, Second IEEE International Conference, DCOSS 2006, Proceedings*, volume 4026 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[HM08] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing degrees, models, and applications. *ACM Comput. Surv.*, 40:2–25, August 2008.

[ITU05] ITU. Itu internet report 2005: The internet of things. Technical report, International Telecommunication Union, 2005.

[JJ85] J.Kramer and J.Magge. Dynamic configuration for distributed systems. *IEEE Trans. Softw. Eng.*, 11(4):424–436, 1985.

[KM03] Jeffrey O. Kephart and David M.Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[Kri09] Paul Krill. REST vs. SOAP-based WS-* Web services duel dismissed. *InfoWorld*, 2009. http://www.infoworld.com/d/developer-world/rest-vs-soap-based-ws-web-services-duel-dismissed-432.

[MTSM03] James McGovern, Sameer Tyagi, Michael Stevens, and Sunil Mathew. Chapter 2: Service oriented architecture. In *Java Web Services Architecture*, pages 35–63. Elsevier Science, May 2003.

[Myh06] Nathan Myhrvold. Moore's law corollary: Pixel power. *New York Times*, 2006.

[Nat89] National Science Foundation. A report by the NSF-IRIS Review Panel for Research on Coordination Theory and Technology, 1989.

[OASa] OASIS. OASIS UDDI Specification TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uddi-spec.

[OASb] OASIS. Web Services Business Process Execution Language (WS-BPEL) TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

[OASc] OASIS. Web Services Resource Framework (WSRF) TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf.

[OASd] OASIS. Web Services Transaction (WS-TX) TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx.

[OAS07] OASIS. Web Services Coordination (WS-Coordination) Version 1.1. http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.1-spec-os/wstx-wscoor-1.1-spec-os.html, 2007.

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[Pel03] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[Ple02] Charles Plesums. Introduction to workflow. Computer Sciences Corporation, Financial Services Group, http://www.plesums.com/image/introworkflow.html, 2002.

[PS11] Hervé Paulino and João Ruivo Santos. A middleware framework for the web integration of sensor networks. In *Sensor Systems and Software - Second International ICST Conference, S-CUBE 2010, Revised Selected Papers*, LNICST, pages 75–90. Springer-Verlag, 2011.

[PT09] Hervé Paulino and Carlos Tavares. Sedeuse: A model for service-oriented computing in dynamic environments. In Jean-Marie Bonnin, Carlo Giannelli, and Thomas Magedanz, editors, *Mobile Wireless Middleware, Operating Systems and Applications. Second International Conference, Mobilware 2009, Berlin, Germany, April 28-29, 2009*, number 7 in Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering. Springer-Verlag, 04 2009.

[RAH06] N. Russell, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Workflow exception patterns. In E. Dubois and K. Pohl, editors, *Proceedings of the*

18th International Conference on Advanced Information Systems Engineering (CAiSE'06), volume 4001 of *Lecture Notes in Computer Science*, pages 288–302. Springer-Verlag, Berlin, 2006.

[RHEA04] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow data patterns. QUT Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.

[RSZ04] C. S. Raghavendra, K. Sivalingam, and T. Znati. Communication protocols for sensor networks. In *Wireless sensor networks*, pages 21–50. Kluwer Academic Publishers, 2004.

[RtHvdAM06] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow control-flow patterns : A revised view. (pdf, 1.04mb). Technical report, BPMcenter.org, 2006. http://www.workflowpatterns.com/documentation/documents/BPM-06-22.pdf.

[RvdAtHE05] Nick Russell, Wil M. P. van der Aalst, Arthur H.M. ter Hofstede, and David Edmond. Workflow resource patterns: Identification, representation and tool support. In Oscar Pastor and João Falcão e Cunha, editors, *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings*, volume 3520 of *Lecture Notes in Computer Science*, pages 216–232. Springer, 2005.

[San09] Joao Ruivo Santos. A middleware framework for the remote access and management of sensor networks. Master's thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2009.

[Sch02] Douglas C. Schmidt. Middleware for real-time and embedded systems. *Commun. ACM*, 45(6):43–48, 2002.

[SMZ07] Kazem Sohraby, Daniel Minoli, and Taieb Znati. *Wireless Sensor Networks: Technology, Protocols, and Applications.* Wiley-Interscience, 2007.

[Sot04] Borja Sotomayor. *The Globus Toolkit 4 Programmer's Tutorial*. Globus Documentation Project, July 2004.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.

[vdAtH10] Wil van der Aalst and Arthur ter Hofstede. Workflow patterns home page. http://www.workflowpatterns.com/, 2010.

[vT] Do van Than. Web service orchestration - an open and standardised approach for creating advanced services. http:

//www.eurescom.eu/message/messageJun2003/Web_Service_Orchestration.asp.

[W3Ca] W3C. SOAP Specifications. http://www.w3.org/TR/soap/.

[W3Cb] W3C. Web Services Description Language (WSDL). http://www.w3.org/TR/wsdl.

[WV01] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.