Viljo Pilli-Sihvola

Intelligence as a Service

Master's Thesis
in Information Technology
December 16, 2010

University of Jyväskylä

Department of Mathematical Information Technology

Jyväskylä

Abstract

Pilli-Sihvola, Viljo

Intelligence as a Service / Viljo Pilli-Sihvola
Jyväskylä: University of Jyväskylä, 2010
81 p.
Master's Thesis

The problem of providing intelligence functionalities as a service in large, complex software systems is discussed. Intelligence is here considered to be a capability to solve problems by answering questions. The problem is decomposed into sub-problems of communication, service consumer's interface and service provider's interface. For communication, pull type services, mediator pattern and shared, interoperable ontologies are proposed as a solution. Service consumer's and service provider's interfaces are proposed to be implemented with help of adapters, which understand the shared ontologies. A callback mechanism is proposed for providing two-way communication between the consumer and the service provider. One implementation has been devised based on the research, and it is described and analyzed. The implementation is tested with the integration of three different types of AI applications.

Keywords: artificial intelligence, multi-agent systems, service-oriented architecture

Tiivistelmä

Pilli-Sihvola, Viljo
Älykkyys ohjelmistopalveluna / Viljo Pilli-Sihvola
Jyväskylä: Jyväskylän yliopisto, 2010
81 s.
pro gradu -tutkielma

Työ käsittelee älykkyystoiminnallisuuksien tarjoamista palveluna suurissa ja monimutkaisissa ohjelmistoympäristöissä. Älykkyys määritellään kyvyksi ratkoa ongelmia vastaamalla kysymyksiin. Ongelma on hajoitettu pienempiin osiin, jotka ovat kommunikaatio, asiakkaan rajapinta ja palveluntarjoajan rajapinta. Kommunikaation ongelmiin tarjotaan ratkaisuiksi palvelun käyttämistä asiakkaan aloitteesta, Välittäjäsuunnittelumallia ja jaettuja keskenään yhteensopivia ontologioita. Asiakkaan ja palveluntarjoajan rajapinnat ehdotetaan toteutettavaksi jaettujen ontologioiden mukaan toteutetuilla sovittimilla. Kahden suunnan kommunikaation toteuttamiseksi asiakkaan ja palveluntarjoajan välillä ehdotetaan takaisinkutsuja. Yksi tutkimuksen pohjalta toteutettu järjestelmä kuvataan ja analysoidaan. Toteutus on testattu kolmen erityyppisen tekoälysovelluksen järjestelmään liittämisellä.

Avainsanat: tekoäly, moniagenttijärjestelmät, palvelukeskeinen arkkitehtuuri

Glossary

ACL	agent communication language
AI	artificial intelligence
API	application programming interface
FIPA	Foundation for Intelligent Physical Agents
FIPA-ACL	FIPA ACL standard
HTTP	Hypertext Transfer Protocol
LAN	local area network
MAS	multi-agent system
N3	Notation3
OWL	The Web Ontology Language
RAB	reusable atomic behaviour
RDF	Resource Description Framework
RDF/XML	XML serialization format for RDF
S-APL	Semantic Agent Programming Language
SOA	service-oriented architecture
VPN	virtual private network
W3C	The World Wide Web Consortium
WAN	wide area network
WLAN	wireless local area network
XML	Extensible Markup Language

Contents

1 Introduction

As software applications and their complexity grow, so does the need for solutions supporting development of them. With advancing ubiquitousness and interconnectivity of software, distributed architectures such as service-oriented architectures and multi-agent systems become more and more common. One of the basic applications of software is it to function intelligently, and to solve that, various artificial intelligence technologies have been developed. Such applications can often be beneficial when implementing larger software systems. These are the starting points which this work is based on.

1.1 Problem description

The work concentrates on the problem of how to provide intelligence functionalities as a software service effectively, when intelligence functionality is considered to be a capability of solving problems by answering questions. The intelligence functionalities can be either artificial intelligence or human intelligence.

The problem is divided into sub-problems of communication and the implementation of the interfaces of the involved parties (the service consumer and the service provider). The problem of communication is divided further into smaller sub-problems of data synchronization, mutual comprehension, functional representation of knowledge and coupling.

The motivation for providing intelligence as a service is in the general benefits of implementing software functionalities as a service. Implementing software functionalities as services naturally separates the functionalities from each other. That makes the reuse of software functionalities easier in various contexts by increasing the level of cohesion [1].

The abstract analysis is done on a high enough level to possibly give insights on various design domains. The solution is, however, designed keeping in mind the possible practical implementations, which lie in complex and logically distributed systems, most notably in multi-agent systems and service-oriented architectures. It is presented as a general software architecture that could be used in actual implementations. Many of the design solutions are derived from common design patterns that can be found in the literature.

On a practical level, the work discusses how to implement a multi-agent system in which the usage of artificial intelligence technologies is as easy as possible. The ease here translates to requirements of ease of implementation, scalability and tolerance for changes, which are all important issues in various software development scenarios.

The main approach taken in this work for providing intelligence functionalities as a service is to specify an interface which is implemented by all the services, and to make the interface as useful as possible. The purpose of the interface is to help in larger scale integration of artificial intelligence software applications to complex and logically distributed systems. The interface achieves this by lowering the level of coupling of the functionalities.

The scope of the solution is limited to artificial intelligence capable of answering questions somehow. However, intelligence provided by human users is not ruled out, and it is also discussed. The scope of the entire solution is dynamic large systems with a large amount of different artificial intelligence software applications in use, where a high level of coupling and a low level of cohesion would be impractical.

1.2 Structure of the work

The work is divided into a theoretical part and a practical part. The theoretical part describes, analyzes and proposes solutions for the research problem, and also includes short descriptions of the relevant technologies and concepts. The practical part describes implementation of one possible solution and testing and an analysis of it. Finally, a summary is presented.

2 Term definitions

For the sake of clarity, this section defines terms important for this work. Terms such as *"intelligence"* could otherwise be understood in various ways.

- Intelligence: Intelligence is understood within the context of this work as the ability to solve problems [2]. The definition is wide, but works to its purpose, as the purpose of it is to function as a term to cover seemingly intelligent decision making either software or human beings can perform. The ability to solve problems does not, however, need to be general in any way. An ability to solve a single problem is enough to be considered as intelligence. A problem, in the context of the definition of intelligence here, is defined as something that can be formulated as a question and solved by a reply to it.

- Intelligence software application: Refers to a software application of some artificial intelligence technology or a software application with its intelligence provided by a human user.

- Intelligence service: Refers to an intelligence software application provided as a service.

- Service environment: Refers to an environment, where intelligence services are located and provided as a service for other components. Service environment can refer to a web service environment, a multi-agent system or possibly some other environment hosting services. The service environment is presumed to be dynamic, meaning that services and other components can be added and removed during run-time.

- Consumer: Refers to a potential user of a service. User within this context is a software process, either human guided or not.

- Provider: Refers to a service provider, a unique invokable service, usually an intelligence service within the context of this work.
- Caller: Refers to a consumer invoking a pull type intelligence service.
- Callee: an invoked pull type intelligence service
- Query: a single message sent by a consumer, containing a question for the intelligence services to solve

3 Key concepts

The scope of this work covers several technologies and concepts seen essential to describe here in order avoid ambiguity. The focus of the descriptions is on what is essential for this work.

3.1 Service

According to [4], *"a service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description."* The definition comes from a reference model of service- oriented architectures, and is well applicable to all the usage scenarios of the term within this work. The same source also considers that the concept of service combines the following ideas:

- *"The capability to perform work for another"*
- *"The specification of the work offered for another"*
- *"The offer to perform work for another"*

Another, more practice-oriented definition with proposed benefits is as: *"Services are self-describing, open components that support rapid, low-cost composition of distributed applications"* [5]. The same source promotes services as suitable infrastructure components in distributed computing application integration. The source discusses web service style services, but this work is not limited only to them.

An important aspect with services regarding this work is that which party (the consumer or the provider) is in control of the delivery of the service. The party in control defines if the service is either of pull type or push type. A pull type service works by the consumer initiating the service use every time, while a push type service works like a

subscription. Using a push type service typically would mean the consumer registering as a user of some service, and receiving the benefits of that service automatically.

3.2 Service-oriented architecture

Service-oriented architecture (SOA) is a software architecture methodology designed to help in the implementation of large, complex systems [3]. By the definition in [4], *"Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains."* From a pragmatic point of view, in SOA, software functionalities are encapsulated into services. These services can then be used as building blocks to compose new software functionalities.

Service-oriented architecture systems are considered to be one of the application fields of this work. Another reason for discussing SOA is that systems implemented with some other paradigm can also benefit from SOA solution models.

In a stereotypical SOA implementation, the services are usually web services implemented by open standards, such as Basic Profile 1.0 [6], which relies heavily on XML. The Web Services Interoperability Organization (WS-I) is the standard organization responsible for the most relevant standards regarding web services.

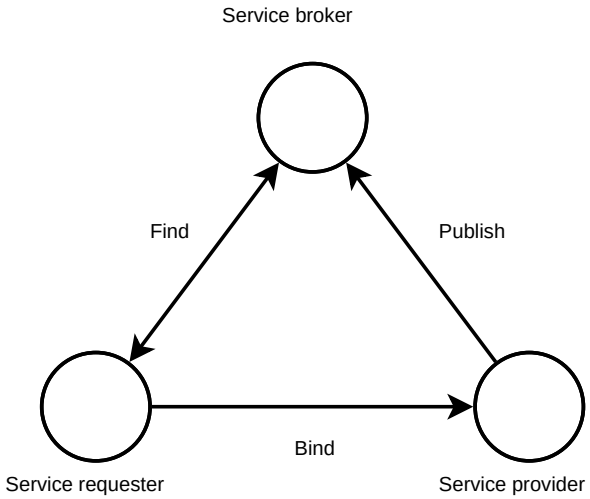

Figure 3.1: SOA architecture

The basic idea, as visualized in 3.1, is to have the functionalities encapsulated as services published in a repository. The service requesters then find desirable services from the registry, and bind to them. The service repository can be just a catalog of the services available. This kind of catalog is known as a registry. If the registry offers intelligent search capability and taxonomy or classification data, it is identified as a broker. If the broker can also describe business logic (interaction rules between components, services in this case), it can be referred to as an aggregator or a gateway [7].

3.3 Agent

An agent here is understood as a software component complying to a wider definition from [8]. It defines an agent to be *"anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors"*. As a simple example clarifying the meaning of sensors and effectors, a human being can be thought to be an agent, and its eyes of being sensors and hands being effectors. Many other sensors and effectors could also be described for a human, of course. A more practice- oriented definition from [10] defines agents as *"active, persistent (software) components that perceive, reason, act, and communicate"*, which is also consistent with this work's use of the term. Various other definitions of agents exist, but they are not relevant to describe for this work.

3.4 Multi-agent systems

Multi-agent system is an architectural model proposed as useful for system development of larger systems [9]. The basic component in it is an agent that acts independently in the system environment. The idea is to avoid centralized control logic.

The benefit of using agents as basic components is that the idea of an agent is inherently modular, and an agent can be constructed locally for each resource, provided it can be made capable of communicating with the rest of the system [10]. In [11], the positive and negative aspects of multi-agent systems are discussed, stating multi-agent

systems to be well-suited for reactive and ubiquitous systems. According to the same source, the negative aspects of multi-agent systems are the absence of overall system controller, poor global perspective of the agents, and the difficulty for humans to trust in autonomous software components. The problem of lack of trust can probably be seen with systems where an agent would be able to use a person's bank account for making purchases.

There are various different ways to implement multi-agent systems. For example, all the agents can be similar or different, and they can act democratically or hierarchically. A discussion on different kinds of agent systems and characteristics of agents can be found in [10].

The interoperability between agents is usually important, and it can be accomplished with a common communication protocol (ACL, for example), a language, and shared ontologies. If a common communication standard is used, even different MAS software platforms can possibly interoperate with each other. As with SOA, a standards organization, The Foundation for Intelligent Physical Agents (FIPA), has been set up to promote interoperability. An example of multi-agent systems is UBIWARE platform, used in the practical part of the work. UBIWARE is described later in 8.3.2.

3.5 Knowledge base

In general, a knowledge base is a collection of facts. A more informal definition, taken from [8], is as follows: a knowledge base is a set of representations of facts (sentences) about the world (the environment it is related to). The sentences are expressed in a formal representation language, called a knowledge representation language.

3.6 Ontology

In computer science, an ontology is an explicit specification of a conceptualization. Conceptualization in turn is an abstract, simplified view of the world that we wish to represent. In practice, for example, a common ontology can be used to define the

vocabulary of queries and assertions used within some system [12]. Ontologies are a key part of semantics-based technologies [13], and they have been developed to facilitate knowledge sharing in artificial intelligence [14].

Ontologies provide a common access to information, and it is an interesting benefit regarding the scope of this work. The common access to information is useful when multiple agents base their actions on the same information. The ontology is used as an agreed standard for mapping the information [13].

When applied explicitly in a technical context, ontologies are often encoded in formal languages known as ontology languages. An example of such language is OWL, which is a family of languages for authoring ontologies endorsed by W3C [15] based on RDF. RDF is a model of metadata [17] usually serialized as XML or Notation3 [16]. As a simple example of an ontology, the following definitions can be considered: a person can either be a man or a woman, and a mother is a woman who has a child.

3.7 Inference

Various AI applications use inference. In general, inference means drawing conclusions from existing information. Within this work, it can mean deriving new facts from a given set of facts (logical inference) or deriving new facts from quantitative data (statistical inference). There are various methods how inference can be accomplished, but they are irrelevant for this work and therefore, out of scope. Descriptions of them can be found for example in [8].

3.8 Sample AI applications

The solution is concerned mainly in how to integrate artificial intelligence into a larger system. Therefore, a few artificial intelligence methods are chosen for test cases and described here. The choices here are to be taken as demonstrative rather than exclusive. They have been used as a reference during the design and analysis of the solutions described later.

3.8.1 Mathematical models

A wide range of AI applications based purely on mathematical models exist with various benefits (and drawbacks), and they pose an interesting test case for this work. For example, statistical inference (such as Bayesian inference [8]) can be used in decision making, or artificial neural networks, which are basically mathematical models working like functions with input-output mapping [18, 19]. The inner workings of such methods vary, and are out of the scope of this work. What is within the scope is their usage, which is usually defined by the values they take as input and what they produce as output.

3.8.2 Expert systems

Expert systems are reasoners that work according to the knowledge that is hard-coded into them. They are usually narrow in their scope, focusing on some well-defined problem domain, possessing a knowledge base on the subject. By an exact definition, expert systems are systems where the program logic and rules used in reasoning are separated [20]. In a more technical view, it is stated that computer-based expert systems approximate expert reasoning through two major components, a knowledge base and an inference engine [21].

Time saving, revenue increases, cost cuts, knowledge preservation and propagation, consistency improvement, and reduced development times have been listed as benefits of expert systems [22]. Expert systems also have their drawbacks. Being complicated systems, they are not an optimal solution everywhere. It is also argued that building the knowledge base can be problematic, as the reliability of the information can vary depending on the quality of the building process [23].

As an example AI application, expert systems provide an interesting test scenario because of their possible conversational nature. An expert system might ask for different kinds of additional information, depending on the previous input values. Supporting this kind of activity is a notable requirement from a technical point of view.

3.8.3 Semantic reasoners

Semantic reasoners are logical reasoners that can also employ semantic information. Like all logical reasoners, they work via forward or backward chaining and infer logical consequences from a set of facts. The difference is that they might use ontologies in inference, therefore being aware of semantic relations, which can be considered to be their main benefit. This automated inference is, alongside ontologies, an important part of semantics-based technologies [13]. Using the ontology from 3.6, the following is an example of functionality of a semantic reasoner: given Mary is a woman, and Mary has a child, the reasoner can infer the fact that Mary is a mother. For this work, semantic reasoners provide a test case for integration of semantic technologies.

4 Use cases

To give the problem a background, possible practical applications for a solution are described. These are given as examples, and many other possible applications for the solution could be described.

4.1 Intelligence for the Internet

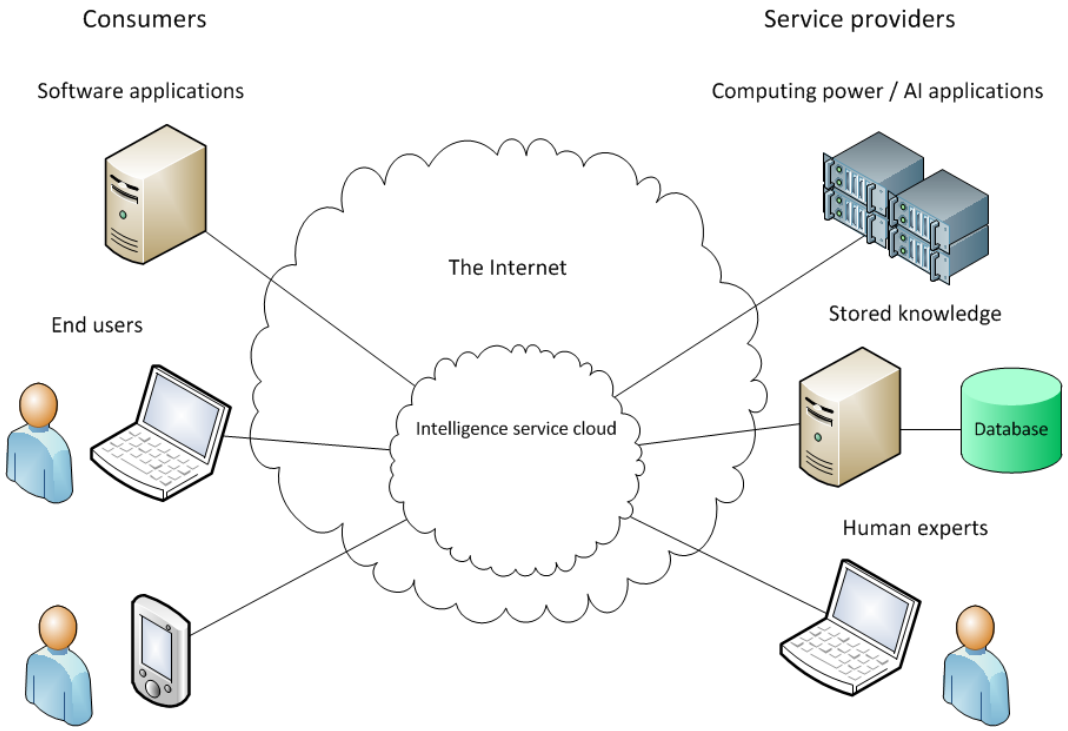

Figure 4.1: Intelligence for the Internet

In 4.1, an usage example for the Internet is visualized. In this scenario, intelligence services are exposed to the public Internet. An intelligence service could be used and published by anyone with proper technological resources. It is visualized as a cloud inside a cloud (the Internet). The clouds represent the idea of abstracting away the

actual technical infrastructure. The consumers are pure software applications or regular everyday Internet users working with the help of some software facilitating their web use. Example service providers here provide human expertise, computing power like resource-intensive AI algorithms ran on computer clusters, and stored knowledge that can be benefitted from.

The benefits of a working such system could be numerous. In fact, the World Wide Web resembles such a system in a passive form. The vision of intelligent services presented here would activate the web, making effective use of its content easier. For example, instead of browsing Wikipedia, you could ask it questions and it would answer to best of its knowledge. This is, actually quite convergent with the idea of Semantic Web [24].

In practice, to prevent abuse or to guarantee quality, access control needs to be implemented, either for the users or the service providers, or both. The scenario seems intriguing in the sense that it would increase the functionality of the Internet, but somewhat unrealistic because of the possible security and interoperability issues.

4.2 Intelligence for industrial use

Another usage example for the solution is visualized in 4.2. This is a scenario possibly more interesting for corporate parties, as it represents integration of an internal information system. The need for this kind of systems is expressed for example in [25].

Here, the intelligence service cloud is as the same as in 4.1, but the larger cloud depicts a closed internetwork instead of the Internet. The consumers and service providers are internal systems or workers, and the intelligence service cloud works like an internal service bus for intelligence requiring activities. The main benefit would be reducing the effort needed for integrating different intelligence applications. The problems with interoperability and security are less of a threat in this scenario, as it deals with a more closed environment.

The actual use for the prototype described in the practical part falls into this category. The prototype is implemented for a middleware platform (UBIWARE) designed

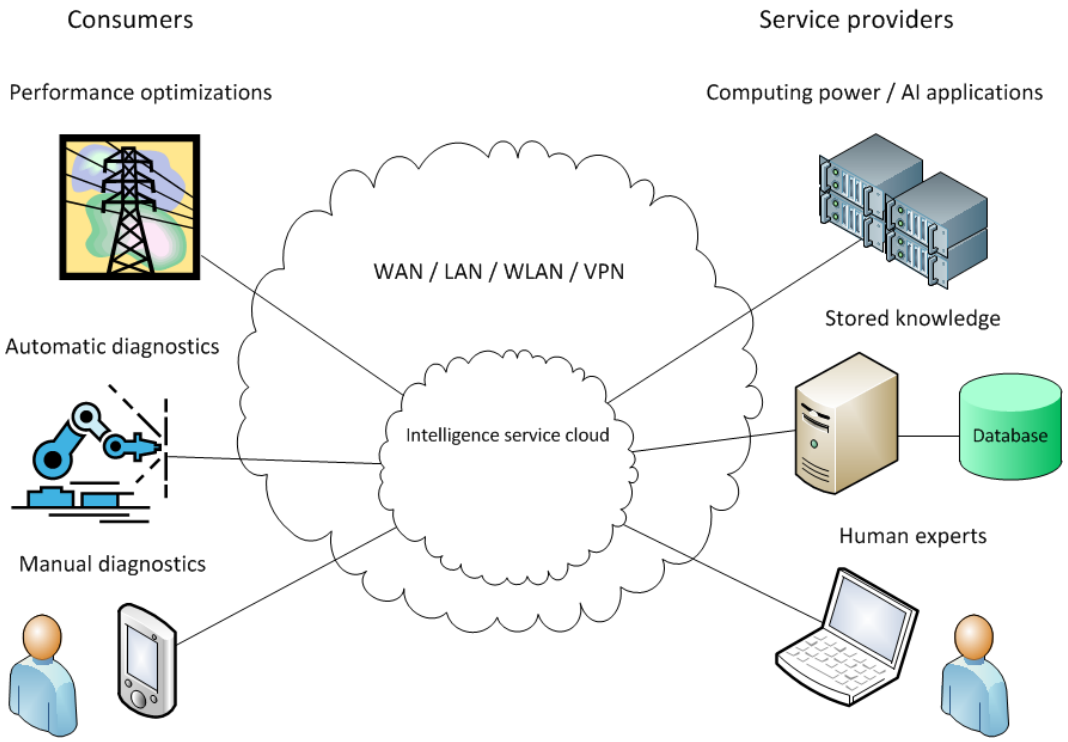

Figure 4.2: Intelligence for industrial information system

to support, amongst other scenarios, industrial scenario software integration.

5 Problem decomposition and proposed solutions

In this chapter, the problem is decomposed to different parts with the aim of identifying the necessary requirements for a system providing intelligence functionalities as services. Abstract solutions are also proposed.

The problem is analyzed from the service consumer's and provider's perspective. This is considered to be a logical approach for systems consisting of services. The presumption of consumers possessing local information is made, as it seems to be the norm when considering complex systems with separate processes, such as multi-agent systems or SOA systems.

An important point is the communication between the consumer and the service providers, and it is handled first. Human user as the intelligence source and learning possibilities for the intelligence services are also considered. The done analysis does not directly rely on any source.

5.1 Communication

The idea of providing intelligence functionalities as services sets the stage up for at least two roles: the service consumer and the service provider. A presumption that some communication, either direct or indirect, between these two is needed is then quite rational. Otherwise, the system would not really resemble a system providing services. The need of communication obviously arises from the consumer using the intelligence services.

When using external intelligence, the consumer needs to identify how it wishes to use it. This practically translates here, in an environment with different intelligence services, to what intelligence service it should use. This needs to be addressed somehow.

Another need in the communication is the possible transfer of knowledge required

from consumer to intelligence service, and the other way around. The consumer might need to transfer knowledge to the intelligence service for it to work, and the intelligence service might need to transfer knowledge to the consumer about what knowledge it requires to work. The answer to the question, obviously, also needs to be communicated to the consumer somehow by the service provider.

The following issues are identified as problems in communication:

- Mutual comprehension: The consumer and service provider must understand each other. Simply put, they must always be talking of the same things. Otherwise, any question or answer is useless. This does not directly imply that the consumer and the provider must use the same terminology. Translation can be an option.

- Functional representation of knowledge: Even if the communicating parties understand each other verbally, they might have problems with the differences in the functional representation of their knowledge [26]. A simple example of such scenario could be the following: service provider asks the consumer for its coordinates in a scale with accuracy of 10 centimeters, but the consumer only has its coordinates on a scale with accuracy of 3 centimeters (let us assume the origin of both the coordinate systems is the same). The problem is simple for human mind, but special program logic is needed for automating it.

- Data synchronization: If the data used in the process is replicated in any stage of the process, its synchronization must be taken into account. For example, the consumer sends all its relevant local knowledge to the service provider for it to perform reasoning based on the relevant knowledge. Now, the possibility that the consumer's relevant local knowledge changes while the service provider is still reasoning must not preferably cause any indeterministic behaviour in the system.

- Coupling of the service consumers and providers: More concretely, the question if consumers use intelligence services directly, or via some mediator mechanism. Mediator here is understood as it is understood in the well-known design pattern

from [27]. Note, however, that the pointed out design pattern is understood here as a model of design rather than a design pattern implemented at program code level.

5.2 Solutions for communication

A distributed software environment presumably already has a standard way of communication for transferring content between the components, such as XML messages or ACL. It is reasonable to use it, but this does not solve the issue in mutual comprehension of how content is understood. A format like XML does not provide much information about the content.

Schema definitions, such as XML schema could be used to increase the semantic information of the content. The idea applied lightly leads to a message passing architecture with custom message formats, such as SOA with WS-I web service standards. Applied vigorously, it leads to a knowledge representation language like RDF/XML with more semantic information.

So, a proposed solution consists of a standard way of knowledge representation. For this to be useful, shared ontologies are also considered as a requirement. However, the shared ontologies should be consistent enough to avoid problems in functional representation of the knowledge. The knowledge represented with the ontologies should comply with interoperable standards. For coupling the consumers and services, a mediator approach is proposed. All these proposed solutions to the identified problems in communication are described in more detail in the following subsections.

5.2.1 Knowledge representation

As discussed, some sort of solution for representing semantic information of content of messages between service consumers and service providers is required. Two kinds of options are identified: a knowledge representation language, or some other less expressive language. If a language is more expressive than a designated knowledge

representation language, it can be very well considered as a knowledge representation language. The difference between these two is visualized below.

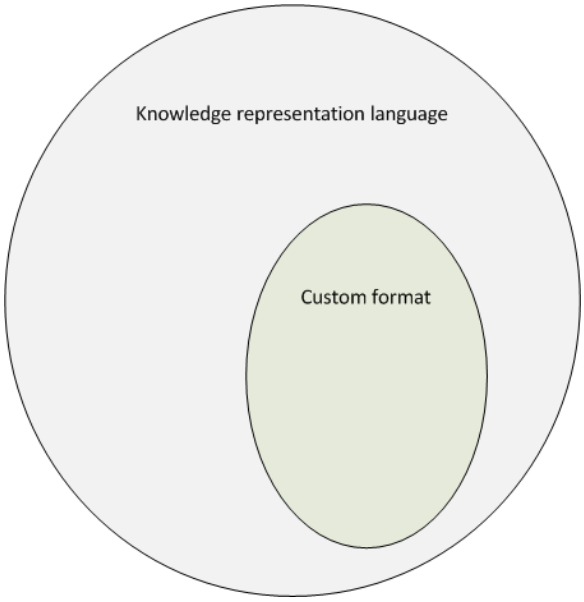

Figure 5.1: Expressiveness of languages, comparison

In figure 5.1, the difference of expressiveness is as expected. A possible problem that can arise in practice is visualized in figure 5.2. In practice, something might be harder to express with a language that is more expressive than other, up to the point of being impractical. As an example, bitmap data might be quite impractical to express in RDF/XML.

Special cases aside, a knowledge representation language should be the most expressive formal language available when representing data according to explicitly defined ontologies, as this is what they are designed for. This is why they are proposed as a useful tool in the design of an intelligence service interface solution.

5.2.2 Shared ontologies

Using the same ontologies in data representation can be a way to unify the information which consumers and service providers use. Without such, the consumers and service providers can understand different words (representations of concepts) differ-

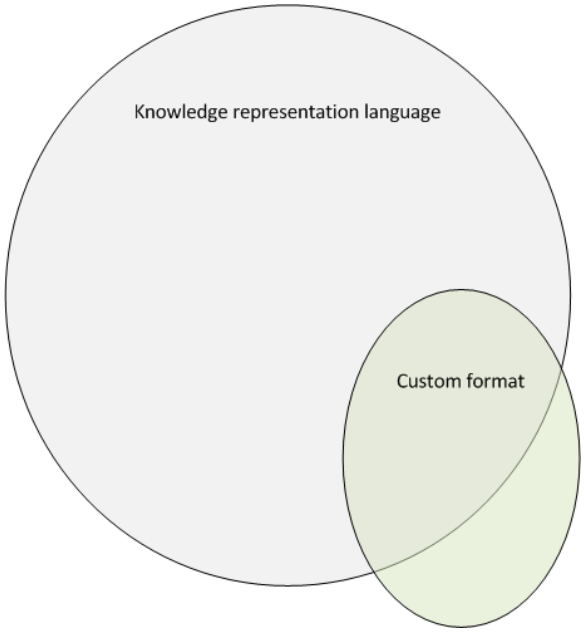

Figure 5.2: Expressiveness of languages in practice, comparison

ently, what should be avoided. This does not mean that a system that does not use formally defined ontologies would be automatically faulty; such system can be seen using ontologies which are not explicitly defined. If the service and consumer software are to be developed in isolation, which is an aspect that should be supported with a low level of coupling, an explicitly defined formal ontology is proposed to be useful.

For consumers, this would mean at minimum that the service call or subscription is made according to the shared ontologies. If consumer's local knowledge needs to be made available for service providers, it should also be represented according to the shared ontologies.

For service providers, this would mean at minimum that at least part of it understands the ontologies (preferably the adapter, introduced later in 6.2), translating the queries to the underlying intelligence software application if necessary. Translation is also an option to be considered if sharing ontologies between the consumer and the service provider is impossible or otherwise impractical.

5.2.3 Functional representation of knowledge in communication

As noted in [26], shared ontologies often are not enough for functional mutual comprehension in communication. If an intelligence service's answers are according to the shared ontologies, the consumer might still not be able to understand it. The consumer might lack the business logic required to benefit from the answer. It seems unrealistic to expect the consumer to be able to react to everything possible to express with the shared ontologies, if they are not overly simple. The answers should be therefore somehow defined. It can be as simple as answering either true or false, or the information included in the answer could be somehow defined in the service signature.

Agreed standards could be used to avoid problems with different scales in knowledge representation. Different scales could still be used, if translation for them is provided. The standards would then help in avoiding interoperability problems with different functional representations of knowledge. In practice, with explicitly defined ontologies in use, this would mean that it is impossible with them to express knowledge that is not translatable to other possible representations of the same knowledge.

5.2.4 Gateway as the mediator

The mediator pattern [27] seems to be a simple solution for the problem of how the communication between consumers and intelligence services could be done. Otherwise, the consumer should communicate directly with all of the intelligence services, at least to find out which ones are providing services the consumer needs. This seems difficult, given it should also be able to locate all of them. Also, given a dynamic service environment, this would mean additional logic and resource requirements for the consumer. The consumer would need to have the functionalities to discover and keep track of the services.

Additionally, as an alternative or an augmentation to the mediator pattern, a registry pattern [28] could be used. It addresses the issue of the need to communicate with all of the intelligence services by helping the consumer (or the mediator) choose which ones to communicate with. A solution with a combination of a mediator and a

registry sounds therefore very promising. This approach is dubbed as a gateway, and a visual representation of it follows in 5.3. The term selection of gateway is based on the usual use of it within software engineering design patterns, as in [28], or in [7], which slightly differs from the afore-mentioned.

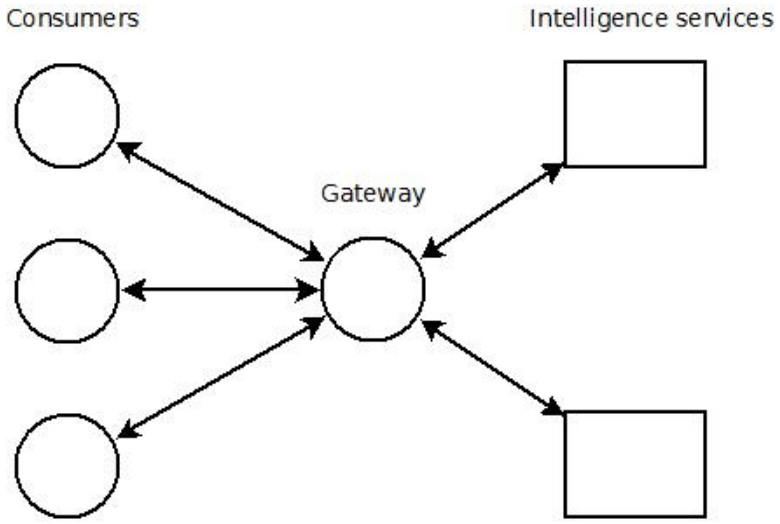

Figure 5.3: The gateway architecture

The additional component between the service consumer and service providers also reduces the work needed to be done by a single consumer. When the consumer software processes run in a resource-limited environment, this could be an important factor. Some functionalities are also facilitated by the mediator pattern, such as caching queries, auctioning, composing answers from various services, keeping track of quality of service and applying security policies.

5.2.5 Data synchronization

Three approaches for data synchronization are identified. These are full, partial, and no synchronization. The full synchronization is simple: it is just made sure that no replicated data is handled by separate processes at the same time. This would be like the common mutual exclusion (*"mutex"*) algorithms.

No synchronization is also simple, as it is effectively no synchronization at all. The challenge with it is to have a system which does not suffer from lack of any

synchronization. As an example, a system with totally stateless components could use this approach.

Partial synchronization lies obviously in between these two afore-described approaches. In it, only necessary parts are synchronized.

Data synchronization becomes an important issue when deciding whether intelligence services should be push type or pull type. Push type services seem to be considerably more difficult to synchronize, compared to pull type services. This is because of the consumer knowing better when to expect the service of pull type services. Because of knowing the exact delivery time of the service, it can easily synchronize the necessary resources effectively without losing time unnecessarily. However, if the system does not need any synchronization related to external intelligence services, using push type services becomes easier.

If any synchronization is done, the relevant synchronization for the problem is synchronizing the data the intelligence functionalities use as their input, and the data their output might have effect upon. Various possibilities on how the synchronization could be done exist. Possibly the simplest is full synchronization done in a way that the caller of an pull type intelligence service denies any changes to its data as it waits for the intelligence service to answer. The main negative aspect of this approach is that it increases the idle time of the caller process, as the intelligence service's answer might not be delivered in an instant.

5.3 Problems from consumer's perspective

There are mainly two problems for the consumer of the intelligence services. These are quite fundamental when discussing interfaces.

- How to use the service: More precisely, how to inform the service provider about the problem the consumer needs to resolve. For pull type service, the problem is how to ask the question. The problem includes the definition of how to invoke the service and how to formulate the question. For push type services, the problem

is how to formulate the subscription. Both types of services have the problem of how to transfer required knowledge from consumer to the service provider.

- What to expect in response: This is the problem of how to benefit from the provided services. Consumers need to be able to benefit from providers' answers.

5.4 Problems from intelligence service provider's perspective

The problems from the service provider's perspective are related to the interface and the integration of the intelligence software into the system. The following problems are identified:

- How to handle the questions: This is mostly the issue of translating the questions to the intelligence software represented by the service.
- How to provide responses for the consumer: The actual response is dependent on the protocols used in the technical implementation, but the theoretic problem is what to respond.
- How to define the required input data: Different intelligence applications require different input data, which might even be defined at runtime. This should be addressed somehow in the solution.
- How to gather the required input data: While communication with the consumer is important, facilitating also communication with other sources than the consumer can increase intelligence service's capabilities. The problem is how the required input data is gathered from the caller and elsewhere for the service provider.
- Conflicts: A risk exists that some intelligence services might understand different questions in a different way. This can be caused by the question format. As an example, if the question would be defined as a subject- object pair, the question *"I, sick"* might cause different kinds of interpretations for expert systems and

semantic reasoners. An expert system might try to determine if the service consumer is sick, while a semantic reasoner might try to determine if the service consumer and the concept of sick are equal.

5.5 Solutions for involved parties

Though the problems were identified separately for service consumers and providers in 5.3 and 5.4, the proposed solutions apply mostly for both. The solutions are therefore presented here for both together. The section ends the abstract analysis of the aspects deemed as most relevant regarding the whole research problem. Having human users as intelligence providers and learning are shortly discussed on the following chapters, however.

5.5.1 Adaptation

If the consumers or the service providers provide technically heterogeneous interfaces, they need to be somehow adapted to provide a common interface. Otherwise, they cannot communicate in a standard way. Another possible problem that can require adaptation is differences in knowledge representation.

A straight-forward approach to adaptation is to create adapter components. They are later discussed in detail in 6.2. The adapter would be responsible of and handling communication flow and translating the communication if needed.

5.5.2 Question and reply format for calling the service

When a component in the service environment wishes to use some intelligence service's ability to answer questions, it provides them somehow the question, and receives an answer. The basic idea of a useful interface is to provide a mean of accessing some functionality. Here, it translates to a definition of the way how the question is presented to the service provider by the consumer and how the service provider communicates its response to the consumer. As the scope of the intelligence provided by the services was

answering questions somehow, an approach of defining a question and a reply format for all intelligence services is proposed as a solution.

The question is formulated obviously according to the defined question format, and the service replies with an answer according to the reply format. For facilitating adding varied types of intelligence services to the system, the formats should be flexible. Importantly, the question format should be as unambiguous as possible, to avoid conflicts in interpreting or delivering the questions. This was described earlier when discussing problems in communication in 5.4.

This kind of solution would address mostly the consumer's view's problems described earlier. It could be a good idea to have the question and reply formats something as close as possible a language as already is used in the system where the intelligence service interface is implemented. For example subject-predicate-object clauses in an environment that already uses subject-predicate-object clauses as its language. In the absence of such a language a new one has to be obviously employed. The basic requirement for the format is that it should be able to express questions. A knowledge representation language is therefore desirable.

5.5.3 Defining questions an intelligence service can answer

It seems reasonable to have the intelligence services define the questions they have a chance of answering successfully. Otherwise, locating potential answerers from the services can become resource consuming, assuming there is a large number of them. Without the services identifying what kind of questions they can answer, a question needs to be made available for all of them to find the best answer. Having the intelligence services define somehow what kind of questions they can answer can presumably narrow the amount of resources needed to get the best result. Therefore it is suggested.

An interesting aspect on defining the questions an intelligence service can answer is how it is actually implemented. If a knowledge representation language is used for formulating questions and answers, a logical approach would be to also use it for defining the questions an intelligence service can answer. The most direct solution

seems to be to describe the questions a service can answer in the same format as they are asked. This would resemble the idea of a function call signature, and it is dubbed here as a service call signature.

An intelligence service capable of answering numerous amounts of different questions needs special treatment, however. It would be impractical, for example, to separately define all the questions a service can answer, if it takes a real number as an input value. The solution could be devised at the syntactic level by using wildcards, which could represent groups of possible input values. Other possible way is to employ ontologies and match questions to service call signatures with semantic networks. The intelligence service could then simply define, for example, that it can answer all speeds regarding the top speeds of cars. It would then be identified by its service call signature as able to receive and answer questions about top speed of any car model.

The approach of using ontologies makes adding semantic information to the service call signature easy. The idea then is basically the same as in semantic annotation of a service; the functionality (problem solving) is hard-coded, but the parameters it takes are semantically described (the question). This is not very far from other ideas, such as semantic web services (as for example described in [29]) or OntoNuts described in [30]. Both have been used as an inspiration source.

5.5.4 Transferring consumer's local knowledge to the service provider

The consumer's relevant local knowledge should be available for service providers it is using. This way the intelligence services have more knowledge to base their functionality on. The question is whether it is the responsibility of the consumer, or the service provider, or should there be other kind of support for it. This is discussed next.

If the responsibility of providing all the necessary is left for the consumer, one solution for pull type services would be to define all the necessary input information in the service call signature. This sounds sub-optimal from the coupling point of view, for because it would change the call signature each time when the required input information changes. This would mean that different intelligence services answering the

exact same question would end up with different service call signatures. It would raise the level of coupling significantly. This could be overcome, however, with dynamically forming the service calls at run time. Additionally, if the intelligence software works in a conversational way, like an expert system might, it would then be very likely to often require more information than it is actually needed from the caller. This could, however, because of the lower layer communication protocols' overhead, lead to less traffic generated by the communication in some cases.

Another possible solution, which would leave the responsibility of making the necessary consumer's local information available for service providers, is to have the consumer transfer all of its relevant knowledge for the service time every time the service is invoked. This kind of approach seems to be vulnerable to performance issues in environments with relatively high amounts of local consumer knowledge. The transfer of all the knowledge could be therefore done in a piece-meal fashion. For example, the consumer would first transfer all of its knowledge, and later on, only changes that have occurred in its local knowledge.

To remove the need for a two-way connection without requiring any special reactivity from consumer or the service provider, an external system storing consumers' local knowledge could be used. This seems like a solution introducing additional complexity, however. It would either reduce the system to be basically a centralized system, or it would ask for replication of high amount of data. This would introduce difficulties in synchronizing. It seems to be a viable option for both push type and pull type services, though.

If the responsibility of finding relevant consumer's local knowledge is left for the service providers, they should be able to query the consumer. To make it possible, the technical system environment should provide a support for service providers to query additional input information from service consumers. For pull type services, a callback mechanism could be implemented.

If no external storage for consumers' local knowledge is used, a push type service would require an ability to connect to consumer for querying additional information. Another option, which is practically the same, would be the consumer keeping an active

connection available for the service provider to use. A push delivery already implicitly requires an ability to connect to consumers for delivering its updates for the subscribed consumers, but it does not imply it has the ability to read consumer's local knowledge. The issue needs also therefore be addressed with push type services.

5.6 Machine-to-Human interoperability

While the problem is primarily scrutinized as a software integration problem, nothing in it really rules out providing user interfaces for human users. Human users could be in intelligence and consumer roles. The possibility to employ human intelligence as the intelligence of a service is shortly discussed, alongside with the possibility of humans being consumers of the intelligence services.

Using an adapter to connect the intelligence functionality to the system does not dictate the encapsulated software application to not be controlled by a human. To add human intelligence providers to the system, only a control interface (preferably a graphical user interface) to the adapter is needed. The control interface should note the human user of incoming queries and it should also be capable of letting the human user answer them. Some sort of translation of the incoming messages and outgoing answers can be required, as the messages are encoded in machine understandable format (not necessarily human understandable). This kind of approach would be a simple way to create a heterogeneous service provider population serving both human and artificial intelligence.

The same approach should work for letting human act as a consumer in the system. Message translation and a graphical user interface for sending queries and answering queries should be enough. This would then provide an interesting user interface for multiple intelligence applications, backed up by artificial or human intelligence.

The challenges of machine-to-human interoperability seem to mostly lie in knowledge representation method's usability for human users. The human users should be aware of the ontologies and possible translations should be as unambiguous as possible. Another issue of note is that introducing humans as controllers of parts of the query

process can cause unpredictable latencies. This can be however true for pure software processes also, more so if they interact with physical environment.

5.7 Learning

The issue of learning is shortly covered, just to make sure the proposed solutions do somehow support it. Within the scope of artificial intelligence applications of this work, two types of learning seem to be reasonable to take account in the design of the solution. The relevant types are reinforcement learning and supervised learning.

5.7.1 Reinforcement learning

Reinforcement learning would be as it is generally understood in the context of machine learning, meaning receiving feedback on actions, and learning from it. Within the context of this work, this would mean the intelligence service receiving input, choosing answer based on the input, and receiving information of the effectiveness of its answer. The intelligence service would then adjust its future answers based on the perceived effectiveness of previous answers [31].

With reinforcement learning, the problem seems to be the workings of a feedback mechanism. It obviously needs the involvement of both the consumer and the provider, unless some sort of external feedback mechanism is devised. The external mechanism sounds likely to be a more complicated option.

5.7.2 Supervised learning

By definition from [8], supervised learning is giving the learning element correct value of the function for particular inputs. The learning element then changes its representation of the function to try to match the given information.

Here, supervised learning means the possibility of external components having an ability to alter the intelligence service's answers by affecting the underlying intelligence mechanism. In practice it would be giving the service provider certain inputs and the

expected outputs.

5.7.3 Solutions for learning

If the adapter is the only component communicating with the intelligence software, the adapter obviously needs to participate in any learning that is dependent on feedback from the service environment. This would mean an interface in the adapter for learning activities.

As a system that would externally monitor actions in the service environment sounds difficult to implement, it is proposed that the consumer takes part in the learning. When giving feedback, as in reinforcement learning, the consumer, being best aware of query result's correctness and its effect on the environment, initiates the action. For supervised learning, the input-output mappings can come practically from anywhere. This implies that the consumer should preferably be able to control it.

6 Possible concrete solutions

A few possible approaches for a concrete solution to the problem are presented. They are not really original in their abstract sense, but the applications within this problem domain are claimed to be at least of independent origin. They are based on the previous analysis of requirements and sub-problems. The solutions are analyzed, and a concrete implementation of one combination of them is later described in the practical part.

6.1 Pull type service

As identified earlier in 5.2.5, push type services seem to be harder to synchronize in certain contexts. As the problem is present with the technologies this work considers as possible application fields, push type services are dismissed as being unpractical. From this point on within this work, services are considered to be of pull type only.

6.2 Adapters

The need for adaptation was identified in 5.5.1. A proposed approach here is to implement adapters for consumers and every intelligence software application that is to be integrated in to the system. The idea of an adapter is quite universal, but as a more formal definition of it, the adapter design pattern [27] can be used.

When using adapters, instead of directly embedding the software into the system, a separate component for each of them is created to integrate them into the system. This is called the adapter component. Here, with the service providers, it provides the interface for potential users of the intelligence software. The consumer of the intelligence service does not communicate directly with the intelligence service's adapter, but with the interface the adapter adapts the intelligence software. For example, this

might be an agent in a MAS or a web service in SOA.

The intelligence software is required only to expose enough of functionality to be used by an adapter, practically meaning some sort of application programming interface. The idea of the intelligence service's adapter is visualized in the image below.

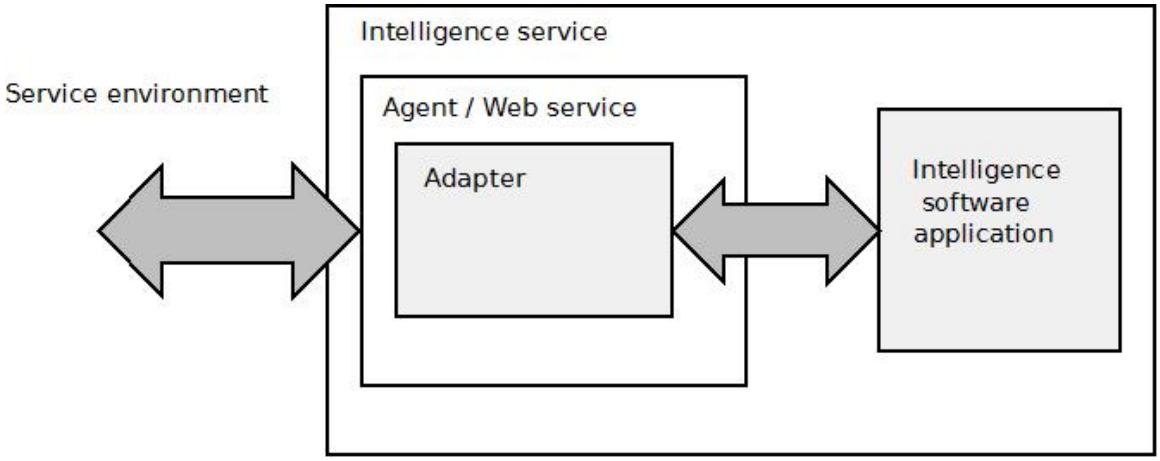

Figure 6.1: The adapter architecture

In the case the intelligence software needs additional information, either from the consumer or from elsewhere from the environment, its adapter should take care of the task it or delegate it further. This seems to logically be the responsibility of its adapter, if it is desired to keep the intelligence software independent of the service environment. Then, its adapter is the only component interacting directly with the intelligence software.

If the consumers are technically homogeneous and their knowledge is already encoded according to the shared ontologies, only one adapter implementation can be enough. This is the case in the practical part's implementation with UBIWARE. Otherwise, same kind of approach as with service providers is recommended for implementing consumer adapters.

6.3 Intelligence services as web services

An obvious approach applying SOA principles would be to implement intelligence services as web services. This has the benefit of making them relatively easy to integrate to SOA architectures. That would also retain ease of integration to multi-agent systems, because various solutions have been devised for it [32, 33, 34]. These solutions address the problems that systems implemented with the two different paradigms (MAS and SOA) might have in the communication by having a mediating system in between the different systems. They do, however, add complexity to the solution if used.

If an AI software has a programmable interface, it should be possible in most of the cases to create an adapter working as a web service for it, as web services are in practice programmable interfaces. The web service standards define methods for data transfer for caller and callee; however, they are not enough according to the previous analysis in 5.4 if a two-way connection is desired.

If the agents are made to directly use the web services, some problems arise from the nature of web services being by their definition one-way in their call structure. This would mean increased complexity if the callee wants to call the caller. One possible solution for implementing two-way communication of the web service caller and the callee is to create HTTP sessions.

With direct agent-to-web service communication, problems might also arise if the service environment is a multi-agent system software platform and high level integration of the intelligence services to the environment is desired. For example, data gathering for decisions from data sources that are represented by agents in the MAS platform would require the web service to also understand the communication standards of the MAS platform. Such would add to the complexity of the solution.

A question and a reply format should be defined, for constructing queries and replies, and a format for defining what questions an intelligence service can answer. All these could be possibly defined by an ontology language like OWL. This would practically lead to web services with their meaning semantically described, also known as semantic web services.

For the web services to be found and for them to be useful without a high level of coupling, a registry or a broker is needed. This complies with the usual SOA design. If a registry is used, the services in this particular registry would be identified by the questions they can answer. In the case of a broker, the broker would, according to some logic, find the potential answerers for queries. The broker is actually mostly the same approach as the gateway solution described next.

6.4 The gateway

To address the need of a mediator, a component functioning as a gateway to the intelligence services can be considered, as was rationalized in 5.2.4. The SOA broker mentioned in 6.3 could function as a gateway to the intelligence services, as it has been already stated. Another way to implement a gateway precisely for a multi-agent system could be to implement an agent or a group of agents functioning as a gateway.

When using gateway design pattern, the elementary idea is for the gateway to function as an access point to some other system or resource, as described in [28]. This would mean the consumers send their queries to the gateway, and the gateway communicates with the intelligence services, and returns an answer by them to the consumer awaiting it. This definition does not however directly solve the need for two-way communication. Some sort of callback mechanism for it is needed.

To address the need for such, the gateway can receive calls containing questions from consumers and invoke suitable intelligence services, acting as a proxy (like the proxy-pattern in [27]) in the process if the intelligence services need to communicate with the caller via callbacks.

In another possible solution, the gateway can function like a registry. The consumer finds suitable intelligence services from the registry, and binds to them on its own. This was already proposed for a web service based system, but it would nevertheless also be a possible approach for a multi-agent system. It seems to be a rather simple approach, but however eliminates some interesting possibilities, such as automatic answer composition from multiple answerers. Such could of course also be implemented

in the consumer side logic, though. The downside of that would be increased resource requirements on the consumer side.

For supporting dynamic addition and removal of intelligence services, the intelligence services should be responsible for informing the gateway of their availability. In this sense, its functionality would resemble that of a registry. If there are multiple agents functioning as a gateway, it is necessary for them to share their knowledge about intelligence services. Otherwise they will not be able to serve the gateway's users with equal quality.

6.5 Multi-agent approach

The use of multiple agents to solve the problem can also be applied to the problem. If the target application field is multi-agent systems, this would be a natural approach. If not, using a multi-agent architecture for solving this particular problem might introduce unnecessary technical complexity, and therefore should be carefully considered.

The main approach would be to implement intelligence services as agents in the multi-agent system environment. With an adapter approach, it would mean wrapping each intelligence software application in an agent interface. This brings the benefit of using the messaging standards of the multi-agent system platform. Implementing intelligence services as agents in a multi-agent system can therefore make the communication with other components (agents) in the environment simpler.

Compared to web services, agents are seen as more potential problem solvers. Agents are considered to be better in composing functionalities [35] and integrating results provided by several services [36]. These points can be used to argue that multi-agent approach is a better way to implement a system providing intelligence functionalities as a service.

If the multi-agent approach is employed, the need for a gateway still remains for the most of it. Applying design patterns originating from object-oriented design to multi-agent systems is not uncommon [37], and it is also done here. The analysis done earlier in 5.2.4 on the subject is still valid, though possible alternative solutions can

easily be found. The intelligence service agents could, for example, form a peer-to-peer network handling queries. The gateway approach is, however, selected as the most potential solution.

7 Summary of analysis

As a summary of the analysis, the following points are proposed as the fundamental requirements for a system providing intelligence services via a unified interface:

- Method of communication
- Unambiguous method of knowledge representation
- Standard ways of representing functional knowledge
- Adaptation of the components to the system

The most promising solutions for communication issues seem to lie in what are known as gateway, registry, and mediator patterns. Knowledge representation languages and shared ontologies are proposed to be the best solution for unambiguous knowledge representation. To improve the functionality of represented knowledge, interoperable standards are proposed. Adaptation can be done with separate adapter components, if necessary. In the next chapters, a demonstrative concrete solution is described and analyzed.

8 Practical part introduction

An implementation of the system solving the research problem was implemented to test the done analysis. It was tested with three AI software applications of different types. The results are described here, and they can be used as a demonstration of the feasibility of the design, or as a reference architecture for future designs, or whatever else they may suit for. The resulting implementation is analyzed here from the relevant points of views regarding this work.

8.1 Chosen approach

From the solutions described before, suitable parts were selected and the composition of them is described here. A multi-agent system, UBIWARE platform, is used as an application field, affecting design choices. The technical solution itself is implemented on the JADE platform, using JADE libraries. The integration of the creation with UBIWARE is quite loose, and therefore the solution could be easily modified to work just on JADE, and possibly with ease on other systems implemented atop of JADE.

The abstract solution is also potentially possible to implement on other similar platforms. The desired functionalities from such a platform are support for communication between components and support for automatically discovering other components. A flexible way of representing knowledge in the platform can also be very useful.

8.2 Rationale of design

Most of the design choices are affected by UBIWARE's and JADE's design. The solution is designed to integrate with ease to existing UBIWARE architecture, leveraging on the work already done. JADE supports development of multi-agent systems nicely [38],

and UBIWARE's system of knowledge representation seemed to fulfill the requirement for a knowledge representation system. The author's familiarity of UBIWARE and JADE, and their relevance for the research projects on which this work is related were also factors affecting the technology choices.

8.3 Technologies

The used technologies are given a short introduction here. For details, see the references given. The whole solution is implemented atop Java, making it easily portable. The technologies selected are not to be taken as exclusive in any way. The solution should also be possible to implement with many other technologies.

8.3.1 JADE

JADE is a FIPA standards compliant multi-agent system software framework implemented in Java programming language. It is designed to function as middleware in multi-agent systems [39]. It is free and open source software, and is currently used in various projects by corporations and universities alike [40].

8.3.2 UBIWARE

UBIWARE is a software middleware platform developed at University of Jyväskylä built using the JADE framework. It is a multi-agent platform, and according to [41] it *"will allow creation of self- managed complex industrial systems consisting of distributed, heterogeneous, shared and reusable components of different nature, e.g. smart machines and devices, sensors, actuators, RFIDs, web-services, software components and applications, humans, etc."* As such, it provides an interesting environment for intelligence services.

Aside S-APL, which is described next, UBIWARE platform agents can be programmed with RABs (reusable atomic behaviour). RABs are Java classes, which interface with S-APL. A RAB is employed in the implementation of the solution for

UBIWARE agents to connect with the intelligence functionalities.

8.3.3 S-APL

The UBIWARE platform's agents are programmed with S-APL, which is a subset of N3. It uses subject- predicate-object triples and linked sets of them as its main construct. The S-APL scripts describe the main logic of an agent, and the S-APL subject-predicate-object triples are considered as the agent's beliefs. The beliefs usually change dynamically during runtime, reflecting the agent's behaviour. Yet another useful point is that the vocabulary of S-APL can be defined by ontologies. S-APL is proposed also as a capable content language in communications [42, 43]. This aspect is employed in the example solution.

9 Technical description

In this chapter, a more detailed desciption of the main technical points of the example solution is given. The inner details of the implementation are quite irrevelant for this work, and therefore they are mostly left out.

9.1 High level view of the architecture

A multi-agent approach (as discussed in 6.5) with a gateway (as discussed in 6.4) is used. The gateway and intelligence services are implemented as agents in a multi-agent system, and possible consumers also are agents in the same system. The consumers are supposed to be UBIWARE agents, but other agents could also function as consumers. The service environment therefore is the multi-agent system, and all the identified components exist in the same environment.

As the gateway approach is used, it separates the agents into three different roles: consumers, gateway, and service providers. The consumers and service providers only communicate with the gateway. The communication flows are visualized in 9.1. While the gateway is not spoken of in plural, it could be a group of several agents acting together. However, support for such was not implemented in the prototype. Doing so would result into some additional complexity.

The UBIWARE platform's S-APL already functions as a way to represent knowledge in programming UBIWARE agents, and its use as a knowledge representation language is also applied elsewhere in the solution. It is used as a content language for communicating queries and responses between the consumers, gateway and service providers. The service signatures of intelligence services, which are an important part of the communication, are also defined with it. The gateway functions like a registry for the services, performing the binding of the queries to the services capable of answering

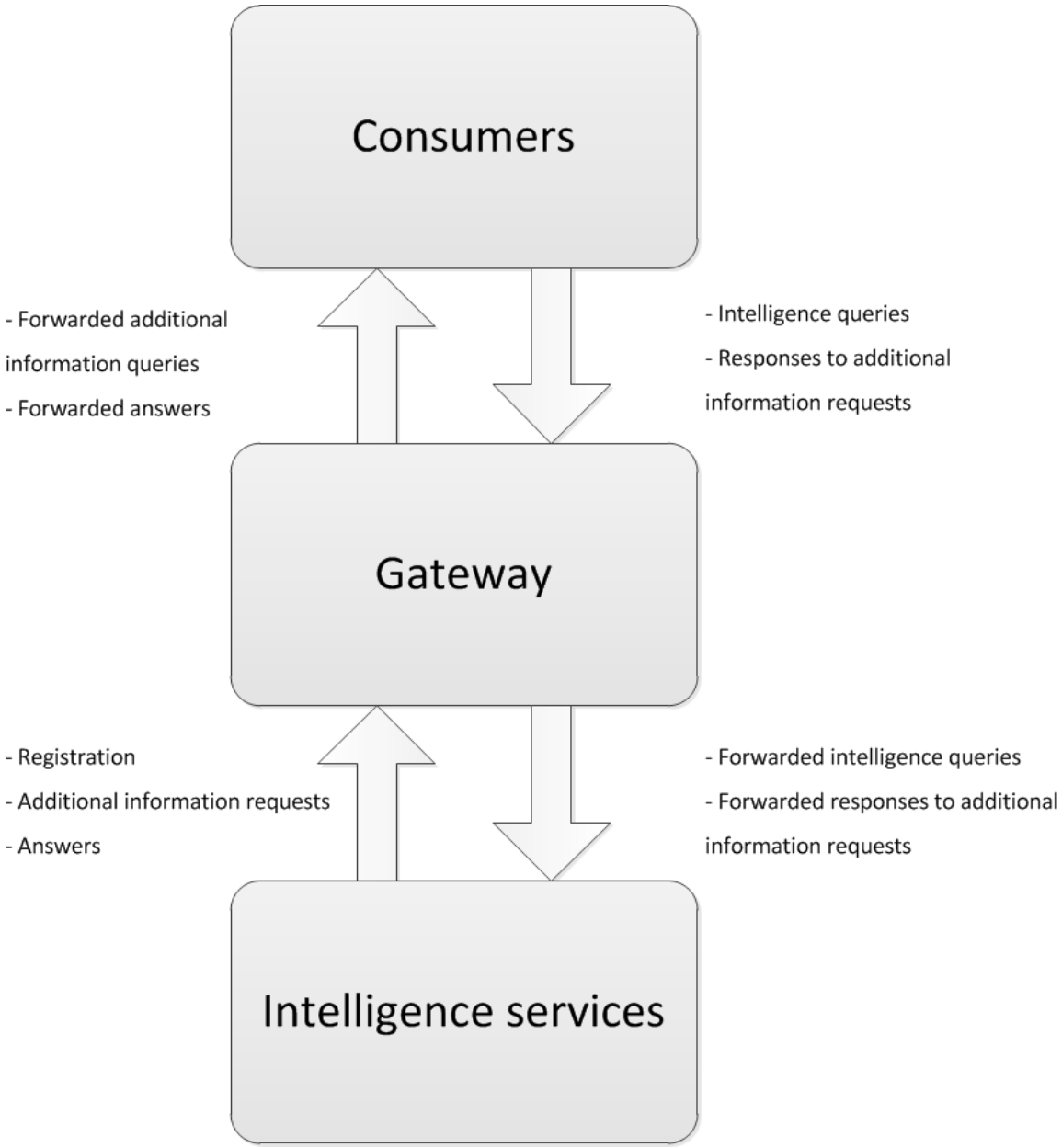

Figure 9.1: Communication

them.

9.2 Agents

The consumers, gateway and intelligence services are all implemented as agents on the JADE platform. The consumers are the only part of the implementation that is connected to the UBIWARE. The consumers are UBIWARE agents, which are JADE agents having additional functionalities making them UBIWARE agents. The gateway and the intelligence services are pure JADE agents. The distinctive technological difference between JADE and UBIWARE agents is that UBIWARE agents are programmed with S-APL and possibly with Java in the RABs, while pure JADE agents are programmed only with Java.

9.2.1 Consumer

The UBIWARE agent consumer's ability to query the intelligence services is implemented as a UBIWARE RAB. It can be seen as the consumer's adapter for the intelligence service system. It takes several parameters, and the most relevant of them is the S-APL statement, which is the question itself. At the current state, the queries can only be answered with true or false, and the question statement is added to the agent's beliefs accordingly. It would be, however, technically trivial to modify the system so that the queries are answered with any S-APL statements, which could easily be added to the consumer's beliefs. After the answer is in agent's beliefs, it is up to the programmed logic of the agent to react to it.

9.2.2 Gateway

The gateway agent does not need any modifications from the potential user of the example solution. It is a JADE agent binding queries to services, holding a registry of available intelligence services, keeping track of the conversations, and forwarding queries and answers to and forth between the consumers and the service providers.

The agent's implemented logic keeps track only of states of the conversations; the communication between consumers and services is otherwise asynchronous.

9.2.3 Intelligence service

The intelligence services are integrated to the rest of the system with the help of an adapter created separately for each intelligence service. The adapter, as discussed in 6.2, is implemented as inheritable (or, extendable in Java terminology) Java classes in the example solution. As much as possible of the functionality is included in those classes. The classes to be extended with inheritance are themselves inherited from a JADE agent and behaviour base classes.

The essential included functionalities in the extendable classes are support for registering and unregistering the service, error recovery in the query process, and a way to query the consumer for additional information (the callback mechanism). The consumer is queried for more information in practice essentially in the same way the belief matching is done in UBIWARE agent programming with S-APL. The matching pattern is forwarded to the consumer via the gateway, and the consumer replies to it based on its beliefs.

An important issue in implementing intelligence services is to use the same ontologies as the consumers in their logic providing the intelligence. Or, if it is not possible, the intelligence service should be able to translate the essential information from the consumer's represented knowledge to its inner data representation. The example solution does not provide any help for such translation.

9.2.4 Example intelligence query

An example of one intelligence query is depicted in a UML sequence diagram in 9.2. Note, however, that the intelligence service could query for additional information more than once (the query for additional information part could be repeated any number of times).

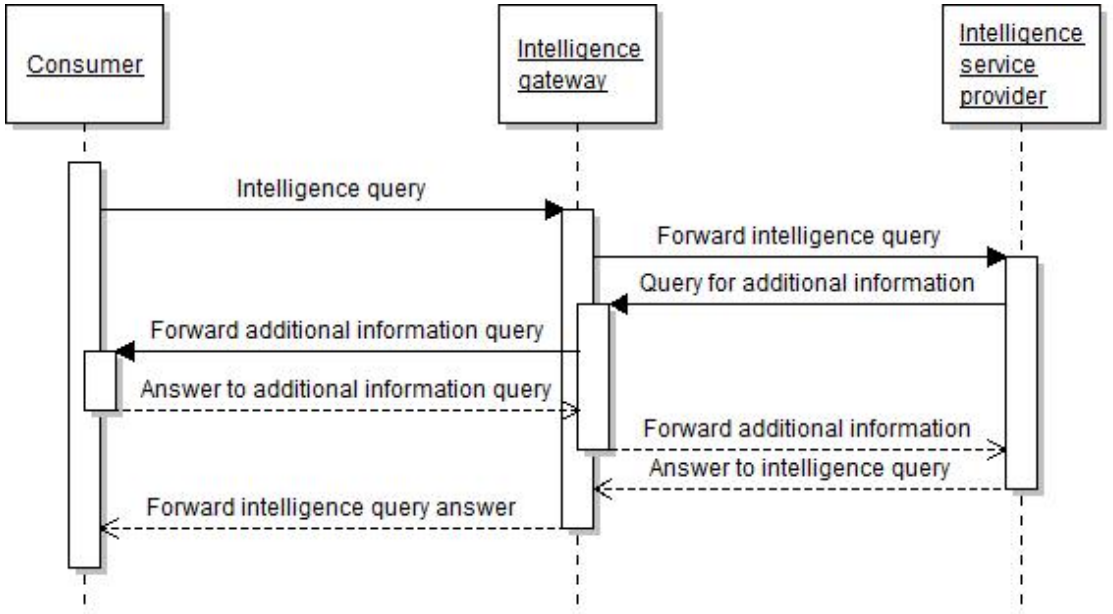

Figure 9.2: Sequence diagram of an intelligence query

9.3 Fault tolerance

To cope with errors, the implemented system keeps track of the duration of the conversations. Timeouts can be configured to discard conversations taking too much time, possibly because of failures in consumer's or service provider's program logic. No specific reason has been found why the gateway agent could not be made redundant with some amount of technical work. The gateway agent could be clustered, or they could form a peer-to-peer network. This was not, however, implemented in the example solution.

9.4 Security

The problem and its offered solutions do not really concentrate on security issues, but the topic is important enough to give it a note. The example solution does not offer any security solutions on its own, but it relies on security solutions implemented on the lower layers of the architecture. The JADE project has various solutions which can be employed, and if a closed internetwork (such as VPN) is used as the whole system's

environment, security problems become easier to solve.

10 Using the solution

A description about the usage of the example solution is given with examples. For running the system, a UBIWARE installation was augmented with the implemented RAB for UBIWARE agents to use, a JADE gateway agent, and a small class library used in implementing the intelligence service adapters.

10.1 Example cases of AI services

Three different kinds of intelligence services were implemented to test the plausibility of the solution from the aspect of a service provider. They were chosen to represent AI software applications which could be possibly used in real systems, with some diversity. Descriptions of the findings follow. It should be noted that the evaluations are not done in a very systematic way, and they are therefore very subjective. The created implementations are available in appendices for further review.

10.1.1 Mathematical model

A simple model basing its reasoning in a function was used for testing the implemented system. The model asks for pulse and age information as input to the question whether the consumer is sick. From those values, a decision is made, and it is offered as an answer. This simple model functions as a test case for applications that have a static set of input values. The inner workings of the model are not important, and an overly simplified model with no scientific base was used.

The task of integrating such a model to the system was relatively easy using the implemented framework classes. The required information is queried by the adapter from the consumer, and if something is missing, the process can be aborted or default values used. A sample code can be seen at A. The required two classes were imple-

mented, and the created framework supported the implementation quite well, as can be seen from the program code samples which are not overly complicated or lengthy.

10.1.2 Expert system

To test integrating expert systems, Drools Expert rule engine was used. Drools Expert is an open source rule engine part of a larger Drools business logic integration platform [44]. The purpose of this test case was to provide information on how difficult it is to integrate expert systems to the implemented system.

Drools Expert offers many possibilities how it could be employed with its wide API. The approach used in the integration was to make the expert system logic to be able to work directly on the concepts defined by the ontologies used by the entire system. It was also made able to query the consumer from the expert system logic if required.

The taken approach was not overly difficult. Various issues however arose during the work, and they are possibly enough to question the whole approach. As expert systems are designed to function on a narrow domain, their technical workings also try to support it. This conflicts with the taken approach that aimed for generalization. The expert system used in testing (Drools Expert) is focused on domain-specific solutions, and designed to technically support it. So, in practice this can be seen on its use of objects, which are supposed to model domain-specific concepts. During the testing it became clear that this does not really work well instantly when working with information expressed in a knowledge representation language like N3, as it differs quite a lot in its idea from data modeled with static class definitions.

Different kinds of possible approaches were identified for integrating expert systems. First, using objects can be ignored totally. This is however a solution for which the used expert system software was not designed for. Second, the information used could be mapped to belief objects, or something similar. This is a bit more like what the Drools Expert software is designed for, but it is somewhat unwieldy still in practice. The third approach is to create and use objects that represent the domain concepts with the expert system software. This makes using the example expert system eas-

ier, but requires more classes and translation of information from the used knowledge representation language (S-APL) to objects.

The first approach was tested, as the second one did not seem very useful, and the third one would have mostly resembled ordinary expert system implementation. Some problems were identified with the first approach. In addition to the problem of the approach not being supported well by the used expert system software, it was noted that using S-APL notation and querying for more information during the reasoning progress will probably require additional solutions in integrating Drools Expert.

For the querying of additional information, there are basically two kinds of ways: to control the queries from the rule logic, or from the programming code logic. The first is more difficult to implement, but it adheres better to the principle used in expert systems of separating programming logic from the rule logic.

Nevertheless of the problems found, the main intelligence interface solution does seem capable of also supporting expert systems. The integration of the object-oriented Drools Expert to the entire system just is not a trivial task. It is suggested also that something else than an object-oriented expert system could be possibly easier to integrate with the entire system, based on the problems which object-oriented programming support caused in the test.

10.1.3 Semantic reasoner

For testing the solution for semantic reasoners, Jena was used. Jena is a Java framework for building Semantic web applications, originally developed by Hewlett Packard Labs [45]. An intelligence service was set up which responds to queries about whether some statement expressed with the ontologies it uses for reasoning is true.

During the test, no major problems were found regarding the solution. After the intelligence service receives the query, the consumer is queried for the relevant information, and semantic reasoning engine checks whether the statement in the query is true, and the service responds accordingly.

The querying of the additional information could be optimized to not query all the

information expressed with the service's ontologies, but this would require integration of the adapter to the reasoning engine to be tighter. It would also create more messaging, possibly eliminating the benefits of the optimization in the communication. But if optimizing the transfer of consumer's local knowledge is not taken into account, integrating Jena seemed to be an easy task.

11 Analysis of the implementation

The abstract solutions given are claimed to work as purposed, solving the initial problem described. However, while the example solution seems to work plausibly, more testing would be needed for better understanding. A system large and dynamic enough should be used in the testing. The simple test cases serve only as a starting point. Before larger testing, additional improvements should be considered. Semantic service signature matching and clustering of the gateway are identified as possible technical improvements.

One distinct problem with using ontologies as a tool in integration, as pointed out in [46], is that a change in the ontologies can easily cause problems in the system. As an example, if there is a change in an ontology that an apple does not mean an apple anymore, but an orange instead, this needs to be communicated to all the parties relying on the meaning of an apple. In practical terms, a change in an ontology can possibly break all components using it. Finding solutions for this could be beneficial.

If technically heterogeneous components are to be used, the implemented prototype might become cumbersome. Currently, it is practical with any technology that interfaces easily with Java, as the adapters are Java classes. If something other is desired, a need for another adapter might arise. This starts to sound sub- optimal and overly complicated. Therefore, if it is desired to avoid such scenarios, the adapter should be technologically more agnostic, such as a web service interface. Another possibility could be to define a standard message format and a protocol of communication with the intelligence services. This could be something encapsulated in ACL, for example.

12 Summary

An analysis of the problem of providing intelligence functionalities as services has been made, and concrete approaches for a solution proposed. One of such was selected to be tested, and a test implementation was made using the UBIWARE platform.

The problem was decomposed to sub-problems of communication and the interfaces of the involved parties (the service consumer and the service provider).

The problem of communication was further divided into sub-problems of data synchronization, mutual comprehension, functional knowledge representation and coupling. For data synchronization, no other point was made than that pull type services are significantly easier to synchronize than push type. For mutual comprehension, knowledge representation languages and shared ontologies are proposed as solutions. For functional knowledge representation, interoperable standards are proposed as a solution. A mediator design pattern and a registry pattern matching services by their service call signature are proposed as solutions for coupling.

The solutions for the consumer's and service provider's interface apply generally for both. A knowledge representation language is proposed as a suitable tool for defining service signatures and as suitable content language for intelligence service's answers. Shared ontologies are proposed as useful in defining general service signatures. A callback mechanism in the consumer is proposed as a solution for the intelligence services to be able to communicate their input information requirements dynamically. Separate adapter components are proposed as a solution to widen the range of applications that can be integrated to the system.

Three example artificial intelligence software applications were integrated to the test implementation to test the usefulness of the solution. The integration did not provide much difficulty, and therefore the solution seems even more plausible. It was however found out that if the technical ideas of the to-be-integrated software application

differ much from the technical idea of intelligence interface, the integration might be somewhat challenging. 10.1.2 provides an example case.

After all, the problem of providing intelligence functionalities (if they are defined as capabilities of answering questions) as a service, in general, seems to be a problem of how to provide some specific software functionalities as a service. Based on the research done, it is concluded that at the current state of related software solutions, raising the abstraction level from that would result in less useful solutions, and lowering the abstraction level would increase the work needed in integration.

Depending on the amount of intelligence functionalities needed to integrate, a lower level of abstraction can be acceptable or not. With less intelligence functionalities, there is less work in integration. A higher amount of intelligence functionalities can be more practical to integrate with the *"intelligence as a service"* - approach. High, of course, being a relative term dependent on the context.

13 Acknowledgements

The author wishes to thank the UBIWARE project for providing funding and an interesting research topic for this thesis work.

14 References

[1] Johann Eder, Gerti Kappel, and Michael Schrefl, *Coupling and Cohesion in Object-Oriented Systems*, Technical report, University of Klagenfurt, 1994.

[2] Marvin Minsky, *The Society of Mind*, Simon & Schuster Inc., New York (NY), 1988.

[3] Michael P. Papazoglou et al., *Service-Oriented Computing: State of the Art and Research Challenges.* Computer, Vol. 40 Issue 11 (2007), p. 38–45.

[4] OASIS, *Reference Model for Service Oriented Architecture 1.0*, available in WWW <URL: `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf`>, referenced 6.8.2010.

[5] M. P. Papazoglou and D. Georgakopoulos, *Service-Oriented Computing*, Communications of the ACM, Vol. 46 Issue 10 (2003), p. 25–28.

[6] The Web Services-Interoperability Organization, *Basic Profile, Version 1.0 (Final)*, available in WWW <URL: `http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html`>, 16.4.2004.

[7] Joseph Bih, *Service Oriented Architecture (SOA) A New Paradigm to Implement Dynamic E-business Solutions*, ACM Ubiquity, August (2006), p. 4:1–4:1.

[8] Stuart J. Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, Prentice-Hall International, Inc., New Jersey, 1995.

[9] Nicholas R. Jennings, *An Agent-based Approach for Building Complex Software Systems*, Communications of the ACM, Vol. 44 Issue 4 (2001), p. 35–41.

[10] Michael N. Huhns and Munindar P. Singh, *Readings in Agents*, Morgan Kaufmann Publishers, Inc., San Francisco, 1998.

[11] Nicholas B. Jennings and Michael J. Wooldridge, *Applications of Intelligent Agents*, in Agent Technology - Foundations, Applications and Markets, (ed. Nicholas R. Jennings and Michael J. Wooldridge), Springer, Berlin Heidelberg, 1998, p. 3–28.

[12] Thomas R. Gruber, *Toward Principles for the Design of Ontologies Used for Knowledge Sharing*, International Journal of Human-Computer Studies, (Vol. 43, Issues 5-6) 1995, p. 907–928.

[13] Michael Uschold and Michael Gruninger, *Ontologies and Semantics for Seamless Connectivity*, ACM SIGMOD Record, Vol. 33, Issue 4 (2004), p. 58–64

[14] Michel Klein et al., *Ontologies and Schema Languages on the Web*, in Spinning the Semantic Web, (ed. Dieter Fensel et al.), The MIT Press, Cambridge (MA), 2005, p. 95–139.

[15] Michael K. Smith, Chris Welty, Deborah L. McGuinness, *OWL Web Ontology Language Guide*, available in WWW <URL: `http://www.w3.org/TR/owl-guide/`>, referenced 19.8.2010.

[16] Tim Berners-Lee, *Notation3 (N3) A readable RDF syntax*, available in WWW <URL: `http://www.w3.org/DesignIssues/Notation3`>, referenced 19.8.2010.

[17] Ora Lassila and Ralph R. Swick, *Resource Description Framework (RDF) Model and Syntax Specification*, available in WWW <URL: `http://www.w3.org/TR/PR-rdf-syntax/`>, referenced 19.8.2010.

[18] Bernard Widrow and Michael A. Lehr, *30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation* in Proceedings of the IEEE (Vol 78, Issue 9) 1990, p. 1415–1442.

[19] Simon Haykin, *Neural Networks - Comprehensive Foundation*, IEEE Computer Society Press, Macmillan College Publishing Company, Inc., New York (NY), 1994.

[20] Seppo Linnainmaa, *Asiantuntijajärjestelmät*, in Tekoälyn ensyklopedia, (ed. Eero Hyvönen, Ilkka Karanta, Markku Syrjänen), Gaudeamus, Hämeenlinna, 1993, p. 264–276.

[21] Chuck Williams, *Expert Systems, Knowledge Engineering, and AI Tools - An Overview*, in Expert Systems - A Software Methodology for Modern Applications, (ed. Peter G. Raeth), IEEE Computer Society Press, Los Alamitos (CA), 1990, p. 2–6.

[22] Larry Bielawski and Robert Lewand, *Intelligent Systems Design - Integrating Expert Systems, Hypermedia, and Database Technologies*, John Wiley & Sons, Inc., New York (NY), 1991.

[23] Peter J. Denning, *Towards a Science of Expert Systems*, in Expert Systems - A Software Methodology for Modern Applications, (ed. Peter G. Raeth), IEEE Computer Society Press, Los Alamitos (CA), 1990, p. 420–423.

[24] Grigoris Antoniou and Frank van Harmelen, *A Semantic Web Primer*, The MIT Press, Cambridge (MA), 2004.

[25] Jeff Y. C. Pan and Jay M. Tenenbaum, *An Intelligent Agent Framework for Enterprise Integration*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 21, No. 6 (1991), p. 1391–1408.

[26] Yolanda Gil, *Knowledge Mobility*, in Spinning the Semantic Web, (ed. Dieter Fensel et al.), The MIT Press, Cambridge (MA), 2005, p. 253–278.

[27] Erich Gamma et al., *Design Patterns - Olio-ohjelmointi - Suunnittelumallit*, (trans. Anita Toivonen), IT Press, Helsinki, 2001.

[28] Martin Fowler et al., *Patterns of Enterprise Application Architecture*, Addison-Wesley, Boston (MA), 2003.

[29] Sheila A. McIlraith, Tran Cao Son and Honglei Zeng, *Semantic Web Services*, Intelligent Systems (IEEE), 16 (March 2001), p. 46–53.

[30] Sergiy Nikitin, Artem Katasonov and Vagan Terziyan, *Ontonuts: Reusable semantic components for multi-agent systems*, in The Fifth International Conference on Autonomic and Autonomous Systems (ICAS 2009), 21–25 April, 2009, IEEE CS Press, Valencia, Spain, p. 200–207.

[31] Leslie Pack Kaelbling, Michael L. Littman and Andrew W. Moore, *Reinforcement Learning: A Survey*, Journal of Artificial Intelligence Research, 4 (1996), p. 237–285

[32] Dominic Greenwood and Monique Calisti, *Engineering Web Service - Agent Integration*, in 2004 IEEE International Conference on Systems, Man and Cybernetics, 10–13 October, 2004, p. 1918–1925.

[33] M. Omair Shafiq, Ying Ding and Dieter Fensel, *Bridging Multi Agent Systems and Web Services: towards interoperability between Software Agents and Semantic Web Services*, in Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06), IEEE Computer Society, Washington (DC), 2006, p. 85–96.

[34] Katia Sycara, *Multi-agent Infrastructure, Agent Discovery, Middle Agents for Web Services and Interoperation*, in Multi-agents systems and applications (ed. Jaime G. Carbonnell and J. Siekmann), Springer-Verlag New York, Inc., New York (NY), 2001, p. 17–49.

[35] Michael N. Huhns, *Agents as Web Services*, IEEE Internet Computing, July-August (2002), p. 93–95.

[36] Katia Sycara et al., *Automated discovery, interaction and composition of Semantic Web services*, Journal of Web Semantics, 1 (2003), p. 27–46.

[37] Elizabeth A. Kendall and Margaret T. Malkoun, *Design Patterns for the Development of Multiagent Systems* in Multi-Agent Systems Methodologies and Applications, (ed. Chengqi Zhang and Dickson Lukose), Springer-Verlag, Berlin Heidelberg, 1997.

[38] Fabio Bellifemine et al., *JADE: A software framework for developing multi-agent applications. Lessons learned*, Information and Software Technology, 50 (2008), p. 10–21.

[39] Fabio Bellifemine, Agostino Poggi and Giovanni Rimassa, *Developing Multi-agent Systems with JADE*, in Itelligent Agents VII, (ed. C Castelfranchi and Y. Lespérance), Springer-Verlag, Berlin Heidelberg, 2001, p. 89–103.

[40] Telecom Italia SpA, *JADE project web site*, available in WWW <URL: `http://jade.tilab.com/`>, referenced 19.8.2010.

[41] Industrial Ontologies Group, *UBIWARE project description*, available in WWW <URL: `http://www.cs.jyu.fi/ai/OntoGroup/UBIWARE_details.htm`>, referenced 31.8.2010.

[42] Artem Katasonov and Vagan Terziyan, *Semantic Agent Programming Language (S-APL): A Middleware Platform for the Semantic Web*, in Proceedings of the 2nd IEEE International Conference on Semantic Computing, August 4–7, 2008, Santa Clara, USA, Copyright IEEE, p. 504–511.

[43] Artem Katasonov, *UBIWARE Platform and Semantic Agent Programming Language - Developer's guide*, available in WWW <URL: `http://users.jyu.fi/ akataso/SAPLguide.pdf`>, referenced 31.8.2010.

[44] JBoss Community team, *Drools*, available in WWW <URL: `http://jboss.org/drools`>, referenced 22.8.2010.

[45] *Jena Semantic Web Framework*, available in WWW <URL: `http://jena.sourceforge.net/`>, referenced 29.8.2010.

[46] Sergiy Nikitin, Vagan Terziyan and Michal Nagy, *Mastering Intelligent Clouds - Engineering Intelligent Data Processing Services in the Cloud*, in Proceedings of the 7th International Conference on Informatics in Control, Automation and Robotics (ICINCO-2010), Vol.1, (ed. J. Gilipe, J. Andrade, and J.-L. Ferrier), 15–18 June, 2010, Funchal, Madeira, Portugal, p. 174–181.

A Test case mathematical model

The test case with a mathematical model has two files, *TestAIAdapterAgent1.java* and *TestAIBehaviour1.java*. They represent what a mathematic model's integrator would need to implement and are both relatively simple.

```
// TestAIAdapterAgent1.java

package ai;

import java.util.ArrayList;
import java.util.List;

public class TestAIAdapterAgent1 extends IntelligenceAdapterAgent {

    @Override
    protected void setup() {
        List<String> matches = new ArrayList<String>();
        matches.add("<http://www.ubiware.jyu.fi/sapl#I> " +
                "<http://www.ubiware.jyu.fi/sapl#is> " +
                "<http://www.ubiware.jyu.fi/test#sick>");
        addMatchStrings(matches);
        register(1000);
        this.addBehaviour(new TestAIBehaviour1(this));
    }

}


// TestAIBehaviour1.java

package ai;
```

```
import jade.lang.acl.ACLMessage;

public class TestAIBehaviour1 extends IntelligenceBehaviour {

    public TestAIBehaviour1(IntelligenceAdapterAgent aiAgent) {
        super(aiAgent);
    }

    @Override
    public void action() {
        ACLMessage received = receiveQuery();
        if (received != null) {
            // Query is received. Querying additional information.
            String reply = askForOne(
                    "<http://www.ubiware.jyu.fi/test#pulse> " +
                    "<http://www.ubiware.jyu.fi/sapl#is> ?x", received);
            if (reply == null) return;
            int pulse = 0;
            try {
                pulse = Integer.parseInt(reply);
            } catch (NumberFormatException e) {
                fail("Could not understand pulse value.", received);
            }

            reply = askForOne("<http://www.ubiware.jyu.fi/test#age> " +
                    "<http://www.ubiware.jyu.fi/sapl#is> ?x", received);
            if (reply == null) return;
            int age = 0;
            try {
                age = Integer.parseInt(reply);
            } catch (NumberFormatException e) {
                fail("Could not understand age value.", received);
            }
            String isSick = "false";
            // The actual intelligence functionality
```

```
            if (pulse > 90) isSick = "true";
            if (pulse > 80 && age > 40) isSick = "true";
            answer(isSick, received);
            return;
        } else {
            block();
        }
    }

}
```

B Test case expert system

The test case with an expert system has five files, *TestAIAdapterAgent2.java*, *TestAIBehaviour2.java* , *ConsumerConnection.java*, *Answer.java* and *testrules.drl.* They represent what an expert system's integrator would need to implement and are both relatively simple. The file *testrules.drl* contains the rules the expert system uses in its reasoning, the knowledge base. The files *ConsumerConnection.java* and *Answer.java* are utility classes.

```
// TestAIAdapterAgent2.java

package ai;

import java.util.ArrayList;
import java.util.List;

public class TestAIAdapterAgent2 extends IntelligenceAdapterAgent {

    @Override
    protected void setup() {
        List<String> matches = new ArrayList<String>();
        matches.add("<http://www.ubiware.jyu.fi/sapl#I> " +
                "<http://www.ubiware.jyu.fi/test#should> " +
                "<http://www.ubiware.jyu.fi/test#getRepaired>");
        addMatchStrings(matches);
        register(1000); // The parameter is the timeout.
        this.addBehaviour(new TestAIBehaviour2(this));
    }

}
```

```
// TestAIBehaviour2.java

package ai;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBaseFactory;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.io.ResourceFactory;
import org.drools.runtime.StatefulKnowledgeSession;

import jade.lang.acl.ACLMessage;

public class TestAIBehaviour2 extends IntelligenceBehaviour {

    public TestAIBehaviour2(IntelligenceAdapterAgent aiAgent) {
        super(aiAgent);
    }

    @Override
    public void action() {
        ACLMessage received = receiveQuery();
        if (received != null) {
            // Query is received. The expert system is responsible
            // for querying additional information.
            KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
            KnowledgeBuilder kbuilder = KnowledgeBuilderFactory
                    .newKnowledgeBuilder();
            kbuilder.add(ResourceFactory.newClassPathResource(
                    "testrules.drl", getClass()),
            ResourceType.DRL);

            if (kbuilder.hasErrors()) {
                System.err.println(kbuilder.getErrors().toString());
            }
```

```
            kbase.addKnowledgePackages(kbuilder.getKnowledgePackages());
            StatefulKnowledgeSession ksession =
                kbase.newStatefulKnowledgeSession();
            ksession.setGlobal("connection",
                new ConsumerConnection(this, received));
            final Answer expertAnswer = new Answer(false);
            ksession.setGlobal("answer", expertAnswer);
            ksession.fireAllRules();
            answer(Boolean.toString(expertAnswer.getValue()), received);
            return;
        } else {
            block();
        }
    }

}


// ConsumerConnection.java

package ai;

import jade.lang.acl.ACLMessage;

public class ConsumerConnection {

    private final ACLMessage received;
    private final IntelligenceBehaviour behaviour;

    public ConsumerConnection(IntelligenceBehaviour behaviour,
                              ACLMessage received) {
        this.behaviour = behaviour;
        this.received = received;
    }

    public String askForOne(String infoQuery) {
```

```
        return behaviour.askForOne(infoQuery, received);
    }

    public int askForOneInt(String infoQuery, int onFail) {
        try {
            return Integer.parseInt(
                behaviour.askForOne(infoQuery, received));
        } catch (NumberFormatException e) {
            return onFail;
        }
    }

}


// Answer.java

package ai;

public class Answer {

    private boolean value;
    private String reply;

    public Answer(boolean defaultValue) {
        this.value = defaultValue;
        reply = "";
    }

    public boolean getValue() {
        return value;
    }

    public void setValue(boolean value) {
        this.value = value;
    }
```

```
    public void addToReply(String addition) {
        setReply(getReply() + addition);
    }

    public String getReply() {
        return reply;
    }

    public void setReply(String reply) {
        this.reply = reply;
    }

}


// testrules.drl

global ai.ConsumerConnection connection;
global ai.Answer answer;

rule "Suspicious motor value - bad oil pressure"
when
    eval( connection.askForOneInt
            ("<http://www.ubiware.jyu.fi/test#failRate> " +
                <http://www.ubiware.jyu.fi/sapl#is> ?x", 0) > 20 )
    eval( connection.askForOneInt
            ("<http://www.ubiware.jyu.fi/test#oilPressure> " +
                "<http://www.ubiware.jyu.fi/sapl#is> ?x", 100) < 100 )
then
    answer.setValue(true);
    answer.addToReply("Bad oil pressure");
end

rule "Suspicious motor value - battery level low"
when
```

```
    eval( connection.askForOneInt
            ("<http://www.ubiware.jyu.fi/test#failRate> " +
                "<http://www.ubiware.jyu.fi/sapl#is> ?x", 0) > 10 )
    eval( connection.askForOneInt
        ("<http://www.ubiware.jyu.fi/test#batteryLevel> " +
            "<http://www.ubiware.jyu.fi/sapl#is> ?x", 100) < 50 )
then
    answer.setValue(true);
    answer.addToReply("Low battery level");
end

rule "Suspicious motor value - bad fuel consumption"
when
    eval( connection.askForOneInt
            ("<http://www.ubiware.jyu.fi/test#failRate> " +
                "<http://www.ubiware.jyu.fi/sapl#is> ?x", 0) > 50 )
    eval( connection.askForOneInt
        ("<http://www.ubiware.jyu.fi/test#fuelConsumption> " +
            "<http://www.ubiware.jyu.fi/sapl#is> ?x", 100) > 100 )
then
    answer.setValue(true);
    answer.addToReply("Bad fuel consumption");
end
```

C Test case semantic reasoner

The test case with a semantic reasoner has two files, *TestAIAdapterAgent3.java* and *TestAIBehaviour3.java*. They represent what a semantic reasoner's integrator would need to implement. The sample code has some unwieldy manual string parsing, which could be replaced by a cleaner implementation. The issue is, however, irrelevant for the work's topic.

```
// TesAIAdapterAgent3.java

package ai;

import java.util.ArrayList;
import java.util.List;

public class TestAIAdapterAgent3 extends IntelligenceAdapterAgent {

    @Override
    protected void setup() {
        List<String> matches = new ArrayList<String>();
        matches.add("<http://www.owl-ontologies.com/generations.owl#.*> " +
                "<http://www.owl-ontologies.com/generations.owl#.*> " +
                "<http://www.owl-ontologies.com/generations.owl#.*>");
        addMatchStrings(matches);
        register(1000); // The parameter is the timeout.
        this.addBehaviour(new TestAIBehaviour3(this));
    }

}


// TestAIBehaviour3.java
```

```
package ai;

import java.util.List;

import com.hp.hpl.jena.ontology.OntModel;
import com.hp.hpl.jena.rdf.model.InfModel;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.rdf.model.ModelFactory;
import com.hp.hpl.jena.rdf.model.Property;
import com.hp.hpl.jena.rdf.model.Resource;
import com.hp.hpl.jena.rdf.model.ResourceFactory;
import com.hp.hpl.jena.rdf.model.Selector;
import com.hp.hpl.jena.rdf.model.SimpleSelector;
import com.hp.hpl.jena.reasoner.Reasoner;
import com.hp.hpl.jena.reasoner.ReasonerRegistry;

import jade.lang.acl.ACLMessage;

public class TestAIBehaviour3 extends IntelligenceBehaviour {

    private OntModel ontModel;

    public TestAIBehaviour3(IntelligenceAdapterAgent aiAgent) {
        super(aiAgent);
        ontModel = ModelFactory.createOntologyModel();
        ontModel.read("file:///example/generations2.n3", "N3");
    }

    @Override
    public void action() {
        ACLMessage received = receiveQuery();
        if (received != null) {
            // Query is received. Querying additional information.
            List<String> reply = askForAll("?x ?y ?z", received);
```

```
// Parsing the consumer's beliefs to the inference model
// for reasoning (nested beliefs are not supported).
for (String str : reply) {
    str = str.replace("<", "");
    str = str.replace(">", "␣");
    str = str.trim();
    String[] splitted = str.split("\\s+");
    try {
        Resource subject =
            ResourceFactory.createResource(splitted[0]);
        Property predicate =
            ResourceFactory.createProperty(splitted[1]);
        Resource object =
            ResourceFactory.createResource(splitted[2]);
        ontModel.add(
            ResourceFactory.createStatement(
                subject, predicate, object));
    } catch (Exception e) {
        System.out.println(
            "Error␣with␣adding␣belief␣" + str +
            "␣to␣the␣model.␣Stack␣trace:");
        e.printStackTrace();
    }
}

Reasoner owl = ReasonerRegistry.getOWLMiniReasoner();
Reasoner wreasoner = owl.bindSchema(ontModel);
InfModel infb = ModelFactory.createInfModel(wreasoner, ontModel);

// The answer's content is parsed to suitable format.
String[] splitted =
    received.getContent().split("::")[2].split("\\s+");
for (int i = 0; i < splitted.length; i++) {
    splitted[i] = splitted[i].replace(">", "");
    splitted[i] = splitted[i].replace("<", "");
}
```

```
            Selector select = new SimpleSelector(
                    infb.getResource(splitted[0]),
                    infb.getProperty(splitted[1]),
                    infb.getResource(splitted[2]));

            Model queried = infb.query(select);
            String replyStr = "false";
            if (queried.listStatements().hasNext()) replyStr = "true";
            answer(replyStr, received);
            return;
        } else {
            block();
        }
    }

}
```