Multi-Agent Systems for Information Retrieval on the WorldWideWeb

Michael Bleyer

Michael.Bleyer@student.uni-ulm.de

Diplomarbeit

Diplomarbeit an der Universität Ulm
Fakultät für Informatik

1. Gutachter: Prof. H. W. von Henke
2. Gutachter: Dr. A. Uhrmacher

Abstract

This Diploma thesis describes the implementation of a prototype multi-agent system. The system consists of four different types of agents and is based on the Java Agent Template, an agent framework freely available from Stanford University.

The purpose of the multi-agent system is to aid users in searching and retrieving information available on the WorldWideWeb. Information is categorized in concepts and the different agent share and exchange the knowledge about concepts and documents on the WWW that matches these concepts.

This thesis presents how the information is modeled and how it is communicated between the agents of the system. It also includes prototypes of the agents that demonstrate a working implementation of the approach.

Acknowledgments

I wish to thank everybody who has contributed in making this work possible. It started out with a semester at a foreign university and since it has taken a little longer than expected, I really appreciate everybody's patience and support.

First of all I wish to thank Martin Strecker for his help in organizing the foreign semester in France. Great thanks to the Laboratoire d'Intelligence Artificielle, my supervisor Jacqueline Ayel, the members of LIA for their helpful hints and companionship and for a very productive working environment (the office with the most awesome view).

I also wish to thank my supervising professor, H. W. von Henke for supporting and encouraging my uncommon idea of starting a thesis at a foreign university and for his helpful advice and cooperation.

I specially wish to thank my assistant Dr. Adelinde Uhrmacher for her extensive support and time to review the parts of my project and for her many helpful hints. I have learned a lot in our discussions.

Last but not least I would like to thank my family for supporting me through all these years.

Table of Contents

1 Introduction....8

2 Information Retrieval on the WorldWideWeb....10
- 2.1 The structure of information....11
- 2.2 The meaning of information....12
- 2.3 Locating information....13
- 2.4 The nature of a search query....15
- 2.5 Information gathering and query....16
- 2.6 Conclusion....17

3 The agent paradigm....18
- 3.1 An agent from the user's point of view....19
- 3.2 Agent properties....20
  - 3.2.1 Environment....20
  - 3.2.2 Intelligence....21
  - 3.2.3 Learning....22
  - 3.2.4 Autonomy....22
  - 3.2.5 Communication....22
  - 3.2.6 Multi-agent systems....23
  - 3.2.7 Mobility....24
- 3.3 Conclusion....24

4 The CEMAS information model....25
- 4.1 Definition of user and search....25
- 4.2 The concept architecture....26
  - 4.2.1 Definition of a link....26
  - 4.2.2 Definition of a concept....27
  - 4.2.3 The concept tree....30

5 The CEMAS agent architecture....33
- 5.1 The Knowledge Query and Manipulation Language....34
- 5.2 The Java Agent Template (JAT)....37
  - 5.2.1 JAT knowledge....37
  - 5.2.2 JAT communication....38
- 5.3 CEMAS agents....41
  - 5.3.1 CEMAS knowledge....42
  - 5.3.2 CEMAS communication....43
  - 5.3.3 The ConceptBroker agent....46
  - 5.3.4 The ConceptServer agent....48
  - 5.3.5 The ConceptClient agent....49
  - 5.3.6 The ConceptSearch agent....51

6 Example....53
- 6.1 Example concept tree....53
- 6.2 Example session....54
  - 6.2.1 Example ConceptClient session....55
  - 6.2.2 Example ConceptSearch Session....60

7 Implementation 64
7.1 JAT agent overview..........64
7.2 JAT messaging 67
7.3 Package agent..........69
7.4 Package context..........72
7.5 Package resource 73
7.5.1 Resource types 75
7.5.2 ConceptFile class 78
7.6 ConceptUtil package 79

8 Discussion..........80
8.1 The agent paradigm..........80
8.2 The concept model 81
8.3 Implementation..........83
8.3.1 Communication..........83
8.3.2 ConceptBroker 83
8.3.3 ConceptServer..........83
8.3.4 ConceptClient 84
8.3.5 ConceptSearch 85
8.4 Conclusion..........86

9 Bibliography 87
Appendix A 90
Appendix B 91
Appendix C 92
Appendix D 95
Appendix E 96

List of Figures

Figure 4.1 A Link....27
Figure 4.2 A Concept....29
Figure 4.3 Example of a Concept Tree....31
Figure 5.1 Two-layer Message....35
Figure 5.2 CEMAS Architecture....42
Figure 5.3 ConceptBroker Window....47
Figure 5.4 ConceptServer Window....49
Figure 5.5 ConceptClient Window....51
Figure 6.1 Example Concept Tree....54
Figure 6.2 Example ConceptClient Window at Startup....55
Figure 6.3 Example ConceptClient Communication: startup....56
Figure 6.4 Example ConceptClient Communication: address lookup....57
Figure 6.5 Example ConceptClient Communication: root concept....58
Figure 6.6 Example ConceptClient Communication: concept....59
Figure 6.7 Example ConceptClient Communication: links....59
Figure 6.8 Example ConceptClient Communication: insert....60
Figure 6.9 Example ConceptSearch Communication: request concept....61
Figure 6.10 Example ConceptSearch Communication: request links....62
Figure 6.11 Example ConceptSearch Communication: add new link....63
Figure 7.1 JAT Packages....64
Figure 7.2 AgentContext Startup Class Instantiation....65
Figure 7.3 Agent Startup Class Instantiation....66
Figure 7.4 Class Invocation when Receiving a Message....67
Figure 7.5 Class Invocation when Sending a Message....68
Figure 7.6 Agent Class Hierarchy....69
Figure 7.7 MessageHandler Class Hierarchy....70
Figure 7.8 ResourceManager Class Hierarchy....71
Figure 7.9 Context Class Hierarchy....72
Figure 7.10 Interface Classes Hierarchy....73
Figure 7.11 Resource Container Class Hierarchy....75
Figure 7.12 Resource Class Hierarchy....76
Figure 7.13 ConceptFile Class Hierarchy....78
Figure 7.14 Class Hierarchy of the ConceptUtil Package....79

List of Tables

Table 5.1 Reserved KQML Performatives 36
Table 5.2 Performative Parameters...36
Table 5.3 JAT Resources 37
Table 5.4 JAT Knowledge 39
Table 5.5 JAT Ontology 39
Table 5.6 JAT Performatives 41
Table 5.7 CEMAS Knowledge 42
Table 5.8 CEMAS Ontologies 44
Table 5.9 Performatives, Object: service 45
Table 5.10 Performatives, Object: concept..45
Table 5.11 Performatives, Object: link 46
Table 7.1 Resource classes 74

1 Introduction

The amount of information that is available on the WorldWideWeb these days can only be called enormous, and it keeps growing daily. It can be assumed that sooner or later information about any topic known to man will be available somewhere on the WWW (see "What is available on the Web", [Boute96]). As new information is constantly being added the chances are high and continuously increasing that some particular information on any given topic is already available.

The average user searching for information about a particular topic is therefore increasingly overwhelmed by the sheer volume of information on the WWW. The manual searching process of browsing or using one of the available search engines[1] is a time-consuming effort, and quite often the user ends up without finding what he was looking for. Therefore a number of different approaches have been suggested to improve the current situation and simplify finding the location of required information. These approaches range from some changing the standards and structure of the WWW to others that provide additional services which are more or less strongly bound into the existing WWW structure.

One type of approach circles around the general idea of an agent or system of multiple agents, a software tool that relieves the user of certain tasks or makes them easier to handle. These tools are called agents in analogy to a human agent that specializes in providing a certain set of services. The general paradigm of a software agent is not bound to a specific problem domain. Instead it can be seen as a different way to interpret a piece of software that has certain qualities.

Our approach was motivated by two main ideas:

1. To implement a multi-agent system prototype and to experiment with the question of how to apply the agent paradigm to the problem of information retrieval on the WWW. The aim was to find a way in which such a multi-agent system could actually be implemented and to find out how this agent-oriented approach would differ from conventional WWW search mechanisms and whether it provides any advantages over them.

2. To create an agent tool that helps to find information on the WWW. The users we had in mind were scientists typically specializing in a certain topic area of research, looking for comprehensive information about this topic area. This type of user is not only interested in finding all of the information related to his interest, but also wants to be kept up-to-date about newly available information. Another important requirement was to share each users knowledge or expertise about information with other users in the community. Therefore the distributed multi-agent approach, where each user is represented by his own agent in the

[1] Meant are "classic" WWW search engines, some of which are listed in Appendix A.

system. The agents are thought of as representatives of their users, requesting and offering information on their behalf.

This thesis will explain our multi-agent implementation, and will also give an overview of the agent-oriented approach to searching and finding information on the WWW. The next chapter discusses the information available on the WWW and some of the problems of searching and finding the wanted information. The third chapter gives an overview of the agent paradigm and Internet-based software agents. In the forth chapter we will describe how information on the WWW was modeled. The fifth presents our own implementation called CEMAS (Concept Exchanging Multi-Agent System), a multi-agent system of cooperating agents, followed by an example session. The seventh chapter explains the technical details of the implementation. We will conclude with a final discussion.

2 Information Retrieval on the WorldWideWeb

This chapter will give an introduction to the problems of searching and finding information on the WorldWideWeb. Even though the main object of this paper is the implementation of a multi-agent system, it is necessary to review the problem field the system was applied to, since that had an impact on several design decisions. The intention is to show how the agent paradigm can be applied to a practical problem.

In terms of a general Information Retrieval model, the WWW can be seen as a single large database, with the URLs[2] pointing to specific documents as the objects. To find information about a certain topic therefore means to find documents containing text about this topic.

There are several differences to keep in mind though. Since the number of WWW-servers and the documents they serve is very large and keeps growing steadily, the exact size of the database at any given point in time is not known and approaching infinity (see [Beigb97]). As opposed to a closed-world model like a local database where the contents are exactly known, the WWW is an open-world database model with no boundaries.

This results in the following differences for information contained in our virtual database:

- With regard to format: since there is no requirement to adhere to a standard or common format when publishing information on the WWW (thereby entering it into the database), the document formats vary widely, depending on their authors.
- With regard to content: as opposed to a fixed database where the topic of the contents is known (e.g. medical database), no a priori confining assumptions about the contained information can be made. Every topic should be expected to be available and a user may search for anything inside the database, so the system should not constrain the search by limiting it to some topics.
- With regard to existence: a piece of information may be available somewhere, but it still cannot be accessed because its existence is not known. Since the size of the database is virtually infinite, there can never be a complete index of its content (as in a closed-world model).

These differences and their consequences will be discussed in the next three sections, followed by some additional considerations. Due to the size of the WWW, the amount of available information cannot be managed manually. Consequently, there is a need for methods that support automated processing.

[2] A Uniform Resource Locator (URL) identifies a document on the WWW.

2.1 The structure of information

Throughout this thesis, the term information will be used as a synonym for HTML[3] text documents. Types of data other than text (pictures, sounds, films, etc.) are not taken into account, because they are much more difficult to analyze and process (using automated techniques) and consequently pose a set of altogether different problems. Formatted or typed data from a knowledge- or database (e.g. X500 directory service) is also not taken into account, even though such data can be processed more easily by automatic means. A strict requirement for the data to be formatted or typed limits the application domain. Information that does not meet such a format criterion cannot be properly processed. In a system where the aim is to be able to process most of the available documents, there should be no such requirements towards format. This does not exclude the possibility to use document format or type to gain additional information about the document content, but it should be optional.

Thus, HTML documents are the main source of information, because they are the most frequently used format to be found on the WWW. Even though most of the documents are in HTML format, an agent system should assume no more than plain text, simply because it is the only common denominator. The fact that HTML contains additional tags is of no concern, as they can be filtered out easily thus reducing a document to unstructured plain text. Only some HTML tags define the structure of a document, so they could be used to extract additional knowledge about the information contained, using certain tags as indicator to assign the marked text a higher weight or importance (e.g. the Meta, Title, Heading or Anchor tags). This may lead to errors though, since tags are commonly used to achieve a certain layout and not only to structure content. Layout tags (e.g. Bold, Font, etc.) do not express any semantic value of the contained text. In general it seems to be difficult to automatically extract a concept schema from the HTML tags (see [Catar97]). Consequently this means that a system should use document structure tags to extract additional information whenever possible, but that it should not completely rely on it.

For a system that needs to be open to include all the information that exists on the WWW, any prior requirement in terms of format or structure limits the flexibility. The only safe assumption is that each document found can be converted to plain text. As a severe limitation it is still challenging and complex to extract semantic knowledge from unstructured, natural-language text (see [Boone98]).

[3] HyperText Markup Language (HTML), a language used to encode pages or documents on the WWW (see [Ragge95]).

2.2 The meaning of information

By definition, not only the number of documents available on the WWW but also the number of topics contained therein is virtually infinite. It is not known what the information being dealt with will be about. At the beginning of every search, the information content that is being sought after must be semantically described. Such a semantic description can be given by a set of keywords or using a formal query language or ontology.

A trivial assumption that is usually made when dealing with automated text processing is that the meaning of a text can be deduced from the words contained therein in some abstract way and that a document representative can be calculated automatically. The follow-up of this leads to the idea that two texts dealing with the same topic at least partly share the same semantic description (see chapters 2,3 and 6 in [Rijsb79] for further details).

In a system that has to handle vast amounts of topics that change dynamically, it is practically impossible to implement static content-dependent formal descriptions (like description logics or ontologies) for all contained information (see [Oukse97]). Such descriptions can only be properly applied if the contents of a database are finite and exactly known (e.g. medical or technical databases) and more or less well-structured.

However it is possible to apply such formal descriptions to a certain defined subset of information, a distinguished topic area. Some systems follow this approach and also try to interconnect different sets of formal descriptions into one large system (for example a combination or fusion of several medical databases). While this makes sense and seems to work well for related information that is typed, some limitations and losses of flexibility always remain. As a prerequisite, such approaches usually require the information dealt with to be well-categorized and well-formalized. This is rather strict and offers no flexible solution for a general information retrieval system.

For a broader approach, keywords offer a high flexibility, which is necessary for a domain like the WWW where the contents are not exactly known. Keywords have the drawback of not being interpreted in their context, only as stand-alone objects by themselves. Furthermore, a system based on keywords is subject to synonymy and polysemy problems. Different words may be used to describe the same meaning, and the same words may be used in a different sense, having another meaning. However keywords can be used as a first step to limit the amount of potentially relevant documents to a smaller subset, upon which more sophisticated full text analysis can be performed. These sophisticated methods are usually very demanding in terms of computational complexity, so they require too much time and processing power to directly apply them to large amounts of data.

2.3 Locating information

As an identifier for a document, the WWW uses URLs (Uniform Resource Locators), therefore the notion of indicating the existence of some piece of information to another agent is a synonym for giving an URL to it. It is assumed that the URL is an absolute and exact identifier of a document, that the document can be retrieved and therefore the information contained within it can be accessed.

Due to the structure of the WWW no complete index or map of its contents exist. The following shall be no attempt to construct such a complete index, nor a discussion of approaches or methods dealing with that problem. The idea will be rather to use existing resources, tools and methods and combine them with others to provide a value-added service.

There are basically two possibilities to search for information: by browsing and by a query to a search engine. Browsing is the activity of following hyperlinks (anchors, URLs) in HTML documents that lead to other documents. A search engine is commonly queried with a set of keywords and yields a list of hyperlinks as a result, so the documents can be accessed directly.

The first possibility requires the existence of some kind of “home page” to start with. In the best case, such a page may have a collection of links grouped thematically into categories serving as an introduction to a certain topic. In addition to requiring an appropriate starting point though, a single page can never give a complete overview about a topic, but only reveal the information that is linked from that page. The chance that all available documents related to a topic are completely interlinked is rather slim. Therefore only a subset of the allover information available on the WWW can be found by following hyperlinks to related documents.

The nature of hyperlinks between HTML documents is chaotic, there is no common standard or structure and the criteria by which the links are organized depend entirely on the document’s author. In the worst case, a link leads to a new document that has nothing in common with the one just left. For these reasons browsing as a means to find relevant information is only suited to a limited degree.

The second possibility to find information is to send a query to a search engine to retrieve a list of hyperlinks as a result. A query consists of one or more keywords (some search engines offer boolean logic operators) which are matched with the search engines index. The search engines continually explore the WWW by following all the links they find and then index the found documents. Even though not all documents can be found and indexed, the search engines can be expected to cover a rather large percentage of the WWW, especially if the query results of several of them are combined.

A document entry can therefore be known to exist if it has either been indexed by a search engine or if another document links to it (and is in turn either indexed or linked to, recursively). An author is very likely to indicate a documents existence to a search engine or link to it from somewhere when it is published on the WWW, since he wants the document to be found by the public. It is also assumed that information about the topic the user is searching for is generally available somewhere on the WWW. This is important in terms of the general type of information, if it is known or can be expected that a certain kind of information simply is not provided anywhere on the WWW (because it is non-public or classified for example), it makes no sense to try to use the WorldWideWeb as a source thereof.

The search engine's keyword indexing algorithms are rather simple, since they ignore the keyword context[4]. Either a portion or all the words of a document are indexed, if they have enough significance to be useful in identifying it (stop words like "and", "the", etc. are disregarded of course). Thus, a document is scored as a match if the query keyword is contained anywhere in it, regardless of the word's actual context in the document.

Consequently, a search engine usually returns a very long list (in the range of hundreds or thousands) of matched documents. Such a list will contain mostly unwanted irrelevant information (noise), because the context-free keyword interpretation returns a lot of out-of-context matches. Even if these search engines would index the keywords in a manner that would take the context into consideration, the query language would need to offer the possibility to describe a keyword's context to actually make use of it. In other words, both parties (the user and the search engine) would need to have the same understanding of a keyword, a common ontology or formal language, some means of defining the context. As already noted in section 2.2 above, this is practically impossible to implement for the vast amounts of information available on the WWW.

An additional problem is to find significant keywords that have two qualities: identification and discrimination. If the keywords are too specific, the search engine often returns with no results at all. On the other hand if the keywords are too broad, then they match an extraordinary large number of documents. So the keywords should properly identify the searched information and at the same time discriminate it from similar but unwanted information. This keyword significance is both document-dependent and query-dependent. Depending on the document collection indexed in the database, a good keyword occurs only in some documents, to clearly identify and distinguish them from the rest. Good document-dependent keywords can be globally defined if the contents of the database are static. In case of the WWW that is impossible, good keywords would have to be generated dynamically, since for every new query a user defines a new collection of documents (the ones that

[4]This conclusion is derived from the author's personal experience with search engines (e.g. the "refine search" feature of AltaVista). The actual algorithms used by the search engines are a well-kept secret of the companys that own them. For obvious reasons, these companys want to protect their own investment and prevent abuse of their search engines.

match the users query). The keywords would have to include exactly the documents a user wants and exclude all others. Since the number of possible topics is very large and the topics are not necessarily distinct, good keywords for a topic cannot be statically calculated in advance (see [Rijsb79] for details).

Additionally, some of the search engine's knowledge is not passed on to the user, it remains transparent. This is mostly due to the owner's interests in protecting the technological know-how, so the exact algorithms are not revealed. For a system that makes use of these search engines, being able to access this knowledge would allow better exploitation of the knowledge provided. This includes for example the ranking/scoring criteria of the returned matches, or the reasons why certain documents where returned in response to a query.

As a result, it is clear that the current method of a search using keywords and a search engine lacks a desired power and expressiveness, which is quite a limitation. That is the main problem, apart from the fact that a document may not even be indexed by a search engine and its existence is therefore virtually unknown.

2.4 The nature of a search query

Using a search engine on the WWW has several implicit properties. First of all, it is client-server based. That means that the user must actively start a query and gets a single answer in response to it. Since new documents continuously appear on the WWW, the user must repeat the search query from time to time to find out about newly added documents. It is not possible for the search engines to initiate a message informing the user that something new (of possible interest) has arrived.

Furthermore, search engines are aimed at providing an answer to a single topic query. They are not well suited to search for a collection of documents that provide a more-or-less extensive overview over a certain topic. Keywords that fit such a slightly broader range usually result in too much unwanted noise. To satisfy such an information need with a search engine, many queries with a lot of manual filtering are required.

It is important to distinguish between a search query for a topic and a single specific document. If the user searches for a specific document and knows the discriminating keywords, a query to a search engine usually proves successful. If the document of interest is the FAQ of a newsgroup for example, it can be located with a search engine, because the user knows the keywords that exactly identify the document and discriminate it from most others at the same time (e.g. "FAQ" and "comp.infosystems.www.browsers"). However such knowledge of proper keywords on the user's part cannot be assumed when searching for a number of documents related to a topic.

2.5 Information gathering and query

An information retrieval method for the WWW can be roughly divided into two parts, indexing and query. The first part consists of creating an index of the documents contained in the WWW (our database), which is more or less the functionality that classical search engines provide. These indexers register known documents and generate a document representative, usually a set of describing keywords. For our model the WWW's known contents can therefore be reduced to the information indexed by the search engines being used. The second part consists of matching a search query against the document representatives in the index, to find the relevant documents. As mentioned before, this approach does not discuss the problem of indexing documents on the WWW. The indexing functionality provided by Internet search engines is just used as it is.

The focus will be on what can be done with the results of such a query to a search engine, divided into two steps, a collection step and an analyzing step. In the collection step the system will try to find all conceivably relevant documents by means of query (finding in this case means finding out about their existence and location on the WWW), while trying to keep the irrelevant noise to a minimum to reduce computation time for the second step. During the analyzing step, the document's contents returned as a result from the first step are being classified as either matching the search query topic or not. This means filtering out irrelevant documents as well as keeping (not accidentally filtering) relevant ones.

Two important measures are defined in classic information retrieval to describe a system's effectiveness, document recall and precision. Recall is the ratio of the number of relevant documents retrieved to the total number of relevant documents existing. Precision is the ratio of the number of relevant documents retrieved to the total number of documents retrieved.

These two measures cannot be exactly calculated of course, since the relevance of a document is subjective even for humans and (even if considered objective) the number of actual relevant documents on the WWW is unknown. To improve a search on the WWW, the measures can be used as follows:

- Towards high recall: Locating as many documents as possible that are likely to be relevant to the queried topic, which will generate a first intermediate set of documents. Then, eliminating as few relevant documents from this collection in the analyzing process (not incorrectly eliminating good documents).
- Towards high precision: This is affected by the analyzing step, depending on the accuracy of the classification process, weeding out the unwanted noise from the relevant parts.

To optimize the results of the whole process, both steps need to be improved, the query to the search engine (collection) and the classification of the resulting list (analyzing).

2.6 Conclusion

On the WWW, the core problems of IR remain the same: retrieving a number of relevant text documents from a collection in response to a query. However, the size of the WWW and its dynamic and unstructured nature make it difficult to apply classic IR solutions. Full-text classification or categorization algorithms require too much computation time to employ them in a search engine that dynamically calculates an answer to a single query on-the-fly.

This paper proposes a different approach to satisfy an information need, not as single response to a one-time query (like a search engine), but as a continuous process. It combines automatic and user search efforts and allows the implementation of a more sophisticated information comparison and matching using whole documents instead of simple keywords (e.g. IR methods). The approach is based on the agent paradigm presented in the next chapter.

3 The agent paradigm

In this chapter, the software agent paradigm and its possible application to the Internet and information retrieval on the WWW will be explained. Several papers have attempted to define what an agent is, but until now no commonly shared exact definition of agency exists and shall not be attempted here. In addition, many papers describe agents in a very anthropological way, with terms and attributes often used to describe humans (like intelligence, autonomy, learning, communication, etc.). One of the reasons for this may be to suggest some kind of similarity between agents and humans. In reality (and many agent implementations including the one discussed in this paper), this is often reduced to rather pragmatic methods and algorithms. Consequently, these terms should be read and understood with caution, they are used here for reasons of familiarity and their intention is not to suggest that currently existing agents exhibit anything near human-like qualities. Using these terms has become common to describe agent attributes, whether that is justified is a different issue omitted in this paper.

In the following overview, some of the properties often associated with agents are listed. These agent properties are not technologies, they rather describe the problem that is to be solved [Petri97]. Instead of arguing whether such a property is a “must have” for a software to justify calling it agent, it will be explained what the property can be good for. The important points are how a certain agent property is embedded into the agent paradigm and why it may be useful for a particular implementation. Not all possible properties are necessary for every agent, in a particular application some may be more important than others.

The properties of a software agent system, the tasks of “what it does” and “how it does that”, can be described from two points of view, a psychological and a technical [Singh97].

The first is an abstract way of describing an agent, from the point of view of the user to whom the underlying techniques are transparent. Even though such a definition is rather vague and cannot be measured precisely, it can be discussed, compared and evaluated to a certain degree. Such a discussion is still of interest and important, not in order to know whether it is justified to call a piece of software an agent or not, but in terms of usability improvement for the user. Does the agent-oriented approach give the user an advantage in dealing with a problem and what are the differences?

As a second point, even though the exact techniques, mechanisms and algorithms used do not constitute part of a definition of agenthood, they are the means to implement the described properties. Looking at existing projects to see how they actually approach the implementation of agent-like behavior is certainly of interest. This will not necessarily

result in a better understanding or definition of what an agent is or should be, but will hopefully bring in some new leads and ideas of how the ideal agent could be accomplished.

3.1 An agent from the user's point of view

So why should we call these information-searching software tools agents? Let's compare this to a classical example, the human travel agent. This agent is an expert that specializes in a knowledge domain (travels) and offers this as a service. He is called up once and then he works autonomously and most likely returns with a result a while later. Maybe the query for a travel connection has to be refined. Thus the agent communicates with the user and other information sources. The agent uses his intelligence to reason and reach the goal (finding the right travel connection), while having other goals with different priorities at the same time. If the user is a frequent customer, the agent will most likely learn some of the users preferences and sometimes pro-actively suggest or assume something without explicitly being asked for it.

The main emphasis here is on the fact that the agent alleviates the user of some work, because a task can be delegated to it. On a lower level (with a simple agent) the user saves time, because he does not have to do it himself even if he could. On a higher level (with a powerful and very capable agent), the user gains some extra possibilities, because he lacks the domain-specific knowledge to do the task himself. Maybe the agent does not employ much intelligence to satisfy a query (e.g. just passing it on to a travel database), but that remains transparent to the user. How the agent carries out its task internally does not matter, as long as it comes up with a qualified answer to the user's need and thus provides a solution to the problem. A related issue is the interface that is used to communicate with agents. Naturally it should be intuitive and easy-to-use (maybe natural language typed into a keyboard or even spoken), but such a definition is very ambiguous.

Humans tend to characterize complex systems like human beings, describing their behavior using attitudes like knowledge, belief, intention and obligation (see [Shoha93]). Thus the agent metaphor may improve the way a user interacts with an agent. Wooldridge [Woold94] takes the approach of defining agents through "mentalistic notions" one step further, by requiring that agents should not only be describable with such human attitudes, but that they actually need to be based on a formal logical framework that consists of such attitudes and allows to reason about them. Attitudes are divided into two groups. Information attitudes store the agent's knowledge and pro-attitudes store its actions. So the human attitudes are not "simulated", but the core of the agent is actually modeled using these attitudes and a formal logic to represent and manipulate them. However, this view of agents is already an implementation-related issue which rules out many other agent models and is therefore too narrow for this discussion.

Maes[Maes94] mentions two broad problems to be solved with respect to the user, competence and trust. Competence is the knowledge the agent “needs to decide when to help the user, what to help the user with and how to help the user” and trust exists when “the user feels comfortable delegating tasks to the agent”. The acceptance depends mostly on the solution of these two problems, on the other hand such a quality requirement is true for practically every kind of software.

The issues discussed here also apply to an information retrieval agent that finds information for the user on the WWW. The agent should be a helpful tool that the user can delegate this task to, working autonomously by itself. It should employ intelligence, its specific domain knowledge, reasoning and learning to search and find the information the user wants. The agent should fulfil the user’s information needs both reactively in response to a request, but also pro-actively suggesting new information that it deems interesting. A perfect IR agent would interface with the user in an intuitive way, maybe understanding the users natural language.

The conclusion is that from a user’s point of view, it makes a lot of sense to apply the agent paradigm to the problem of information retrieval on the WWW, because mentalistic intentional attitudes (belief, knowledge, free will, etc.) are a convenient abstraction for a complex system that humans are comfortable to interact with [Singh97], [Woold94]. So far a description using such terms remains a very colloquial one. The key problem remains as how to implement such agents.

3.2 Agent properties

Several papers give an introduction to what constitutes an agent. At the same time they admit that these properties do not constitute a definition of agenthood, merely an attempt to capture the idea of the agent paradigm. The following part will list some of the properties that are frequently mentioned, while not claiming to be complete or extensive.

As already noted by [Frank96] and [Petri96], agents act, they do something. The task they accomplish or the service they offer is their most basic property. This acting takes place in a given environment and continues over a longer period of time as opposed to a one-time function being called.

3.2.1 Environment

The environment the agents “live” (sense and act) in determines their scope, they can sense their surroundings (input) and affect them through their actions (output) [Frank96]. As opposed to a physical robot, a software agent interacts using a direct user interface, by communication using a certain protocol or by calling external commands or functions. On the Internet, agents interact with the WWW using the HTTP protocol and with other agents using an appropriate communication language.

A frequently mentioned part of the agent paradigm is that agents live in a “real” environment. In this case “real” does not have the meaning of “physically real” as used in connection with robots, because the agents are software agents. Just the same, “real” captures an important quality of the environment: uncertainty. This usually results from the fact that the environment is open (very large or virtually unlimited in size) and inhabited by other agents or entities that may act in an undeterminable way. The Internet and the WWW share these qualities, it is impossible for an agent to know in advance what it will encounter. Agents should therefore be fault-tolerant and have a certain robustness. Otherwise, users or agents they interact with in their environment may do “damage” to them (on purpose or unintentionally)[5]. Just the same, an agent should not be the cause of problems or damage to its environment.

3.2.2 Intelligence

Basically there are two approaches in AI to model intelligence: symbolic knowledge-based systems and reactive architectures. In a deliberative symbolic approach, a symbolic model of the environment and a set of action descriptions are created. The agent then uses symbolic reasoning to combine a sequence of such actions to a plan. The plan describes a path of action that leads to a predicted desired goal state, which the agent is deliberately trying to reach.

Reactive systems are built on the basic assumption that intelligent behavior can be generated without explicit representation of the world model, but rather emerging from a complex system of behaviors. Essentially there is a set of rules or behaviors from which one is chosen or activated according to the currently sensed environment.

Since both of these approaches are imperfect, hybrid systems have been developed in an attempt to combine the advantages of both. Some properties that are still problematic or difficult to implement are dynamic generation of goals (usually these are predefined and fixed) and dealing with multiple, possibly conflicting goals. A good overview about this topic can be found in [Woold94].

In case several possible ways exist to reach a goal and the result of actions in the given environment is uncertain, a decision-making process is required to find a solution. On the other hand, if there is only a small finite number of possible actions, whose results are exactly known (e.g. more like a function call), the intelligence can also be “hard-coded” into an algorithm. In other words, even though the Internet as environment poses some degree of uncertainty, the results of actions are known more or less exactly. For example if an URL is sent to a WWW server to retrieve a document, it will either return that document or not (because of a time-out or if it does not exist). Apparently an action like that is more like a

[5] Search engines like AltaVista may serve as an example for an Internet-based agent that is being lied to (for this example, we will disregard the aspect of whether a search engine should be considered an agent or not). Users deliberately put false keywords into their HTML pages to trick the search engine into showing their page as a result when being queried with those keywords, even though the page’s content has nothing in common with them.

function call and can very simply be used by an algorithm. Whether such an algorithm can still be called “intelligent” or not is a very debated issue that will not be discussed here.

3.2.3 Learning

Learning is related to or sometimes seen as part of intelligence, it can also be described as adaption with the goal of improving or optimizing the own performance [Imam96]. Single agents can learn to improve their knowledge or their problem-solving method. Within a multi-agent system the improvement can also aim towards the interaction or cooperation. The reward function reflects this in a way that agents try to maximize the combined reward of the whole system. In a competitive environment however, agents would usually behave selfish, trying to improve only themselves and maximize their own reward.

Another distinction can be made with respect to the system’s architecture. Either it is fixed and designed exactly to perform its task, in which case learning takes place through knowledge data being modified or the architecture itself is adapted and evolves as part of the learning process.

3.2.4 Autonomy

Autonomy is sometimes used to explain that the agent does something without constant user interaction or supervision, or even more detached, without direct human intervention at all. But this is a slightly inaccurate description, as it does include simple agents who only act in response to a users query. Therefore the emphasis is also on an agent having its own agenda or goals to pursue. It observes its environment to recognize changes and takes action when a certain state or change is observed. Such a behavior can be seen as autonomy, since the agent itself can decide when to do what, it is not following a direct external order. This implies intelligence to a certain degree, some kind of rule set that the agent uses to make these decisions.

In a multi-agent system or an environment with many agents, autonomy also means being separate from other agents. Each of the agents can be clearly distinguished from the others, it is an encapsulated entity, it has its own internal state and goals which may be different or even contradictory from those of other agents. Note that this does not rule out a cooperative multi-agent system where some agents depend on others (voluntarily or not), because they still have their own goals and are thus autonomous, yet they may not be independent. The agent maintains the distinction between itself and its environment.

3.2.5 Communication

Agents should be able to interact - with other agents and the user (either directly or via an interface). Genesereth sees the ability to communicate as the most important property, as it gives societies of agents the opportunity to “solve problems that cannot be solved alone” [Genes94].

To be able to communicate, agents need a standardized communication language whose syntax and semantics are clearly defined. Both procedural and declarative languages have advantages. Depending on the underlying agent model and implementation either one will be better suited to communicate the agent's expressions. Agent implementations based on a speech act, ontology or belief model tend to use a declarative language to express their speech. By virtue of being sent, a message is intended to result in some action being performed (see [DARPA93], [Labro96] and [Genes92]). Contrary to that, a procedural language seems to be useful for mobile agent approaches to send programs or blocks of code to another location (see [Gener96] for an example). In any case, communication is one of the important parts of software agents, since they interact with their environment mainly by sending messages or calling functions in one way or another.

The communication in a community of many agents can be direct, with each agent talking to each other. This requires some self-organizing architecture, each agent has to know about the other agent's presence and capabilities. It can also be indirect like in a blackboard system, via a message router or a mixed approach in which communication is assisted by specialized agents (e.g. by an agent name server or service broker). With respect to autonomy, each agent must be able to initiate messages [Petri96]. So the communication must be based on peer-to-peer connections, which rules out client-server based protocols and architectures.

3.2.6 Multi-agent systems

As already partly implied in the communication property, many approaches consist of multiple agents. Two main types may be distinguished, based on competition or cooperation. In a competitive architecture the different agents offer solutions to the same problem and compete with each other to offer a better solution than others (e.g. contract net). As opposed to that, in a cooperative architecture the different types of agents each specialize to solve a part of the problem, and work together to combine their capabilities (e.g. specification sharing, blackboard system).

[Sycar95] suggests an approach for a cooperative multi-agent architecture where agents are divided into two groups, task-specific and information-specific agents. Task-specific agents specialize in managing a certain task and usually interact with the user. Information-specific agents specialize in managing access to an information source and offer this functionality to other agents. Advantages like higher flexibility or reusability are well known from classic ideas of distribution, modularity or object-oriented approaches. Many users can share the functionality of agents, new functionality can be added easily and agent modules can be re-used which allows an easier development of similar agents.

3.2.7 Mobility

An additional strain of research focuses on mobility as a property of an agent, though it is mainly required for physical agents (robots), as being part of autonomy. For a software agent, mobility means to be able to relocate itself on its own intention, while preserving execution state and data. While mobility may offer some advantages in certain application fields and the Internet is seen as an ideal environment for mobile agents, there seems to be no problem that cannot also be solved by a non-mobile agent system.

Additionally there are a number of new problems that arise when dealing with mobile agents moving to other hosts, mainly related to security issues. For our approach mobility was not required, a further discussion of it will therefore be left out.

3.3 Conclusion

While it is not clearly defined what exactly an agent is, the agent paradigm may serve in two ways. It may be used to describe the behavior of a complex software system to make it easier for a user to understand it and it may serve as a guideline for the properties we need to implement to get something that could be called an intelligent autonomous agent.

An agent that searches for information on the WWW fits well into this paradigm. It lives in an open and uncertain environment, the Internet. The agent should be autonomous to a certain degree, searching on his own (unsupervised) for information the user wants and return with a result. It should also pursue this goal with a certain amount of intelligence, especially concerning the quality of the information returned, it should filter out the relevant parts from the large amount of unwanted noise. In this respect the agent specializes in solving a distinct problem, and may therefore serve as an entity to which such problems can be delegated, expecting that the agent can solve them by applying its expertise.

4 The CEMAS information model

The implementation of the CEMAS architecture has the main goal of providing a working prototype. Since the amount of time for the implementation was limited, the objective was to create a platform that provides the necessary minimum requirements for a multi-agent system that handles information. A basic requirement of such a system is a model of how to represent and exchange the information which is available on the WWW.

Once that is provided and defined, additional new agents can always be added and included into the system later on. Moreover, the functionality of existing agents can be enhanced without the necessity to change the whole multi-agent system design. Thus, the result of this work was an open experimental platform that can be used to try out and combine some of the ideas or approaches of IR with a multi-agent system.

This chapter explains what the system's typical user looks like and how information is modeled.

4.1 Definition of user and search

In this application, the overall problem field of searching information on the WWW will be applied to the scientific domain. The model user is a scientist who wishes to find an extensive overview of a scientific area or topic or has a general interest in the topic. Thus the user is not just searching for a single document or piece of information. Additionally, the search is not thought of as a single action, but as extended over a longer period of time. It should also reveal newly available information, to reflect the user's continuing interest. Since scientific institutions are generally connected to the Internet and use the WWW to publish scientific documents, it can furthermore be assumed that requested information about the topic will be available somewhere on the WWW. The user normally has an idea of what he is looking for. This does not necessarily mean that he knows the right keywords to start a query, or that he is an expert on the topic in question. Consequently the system should provide some guidance by suggesting several topic categories as reference to start with.

One of the main reasons for using a multi-agent approach was the idea to have each user represented by his own agent in the system. These user agents are the interface to the user and communicate with other agents to acquire the desired information.

Another goal of this design was to share the knowledge about the existence and location of information on the WWW between users with the same interest, thus to re-use their work effort of finding certain information sources, since scientific users are commonly specialists in a certain area and know the available information sources reasonably well. The informa-

tion model should also be able to support the query for all the information sources added by another user (in addition to a query for a certain concept). So if user A finds that user B has similar interests, user B's knowledge about information sources on the WWW can be requested.

Information on the WWW is typically viewed with a browser client (Netscape Navigator, Microsoft Internet Explorer). Therefore the user interface should be integrated into the browser. It is assumed that the user has a direct connection to the Internet and knows how to use the WWW and browser clients to search and find information.

4.2 The concept architecture

In CEMAS, agents exchange knowledge[6] about information (documents on the WWW). This requires that the agents can communicate several issues between each other. To be able to do this, they need a common form of representation for the knowledge. Two general types of knowledge are used:

- Knowledge about one specific document on the WWW. This identifies precisely one document and will be called a *Link*.
- Knowledge about a topic, a collection of documents that contain similar or strongly related information. Such a topic of interest will be called a *Concept*. Multiple concepts are organized in a hierarchical concept tree.

This knowledge is described in the following sections.

4.2.1 Definition of a link

A link represents the knowledge about a single specific document on the WWW. Since each link contains one URL which is by definition unique on the WWW, a link is an exact pointer to the information contained in the document. As long as the original document exists, each agent has access to the same content. A link consists of the following fields:

- **Title**: The name of the document to which the link points
- **URL**: The Uniform Resource Locator that identifies a document on the WWW
- **Description**: A short description of the documents contents
- **Origin**: Where this link came from (an identifier of the agent that added it to the system)
- **Password**: To verify access rights of the owner (change/delete)

[6] The term "knowledge" is used in its general abstract meaning, not referring to a specific formal representation

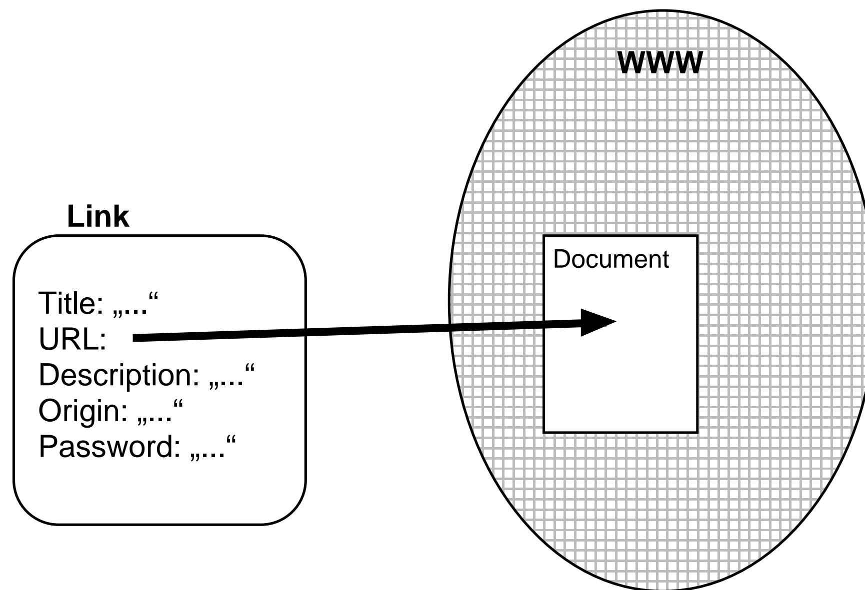

Figure 4.1 A Link

The link includes the document's location and file type (HTML, text, postscript, etc.), which is both captured in the URL. The title and description provide additional information in human-readable form to indicate the contents of the document. The description may be an abstract or a set of keywords. The origin field is used to keep track of the agent that created this link. In case of a user's personal agent, this field contains the user's email address. For other agents it may contain the agent name or the email address of the person running it (the agent administrator). The password ensures that a link can only be modified or deleted by its owner, the agent that created it.

Example:
Title: “Abteilung KI”
URL: “http://www.informatik.uni-ulm.de/abt/ki/”
Description: “Homepage of the AI lab, computer science department, University of Ulm”
Origin: “bleyer@ki.informatik.uni-ulm.de”
Password: “topsecret”

4.2.2 Definition of a concept

A concept represents the knowledge about a topic area. Here we must distinguish between the general concept model (the data structure) and what will be called the meaning of a specific concept (the topic it is about). The semantics of the model are precisely defined, since

agents need to have the same exact definition of the data structure when they exchange concepts. The meaning represented by a specific concept is defined in a flexible way and not by a strict formal language or structure. The differences are described below.

4.2.2.1 The meaning of a concept

As explained in chapter 2, it is difficult to capture the meaning of a concept, in such a way that two communicating agents have the same understanding of it. Keywords are too ambiguous and therefore too weak for an exact definition of meaning. An ontology[7] which defines an absolute view of all existing information on the WWW is practically impossible. Therefore a new approach of defining a concept's meaning was used.

Each concept is associated with a number of documents, that are chosen as being the reference or definition of said concept. The concept's meaning is ultimately defined by the full text of these reference documents. To test whether a document in question matches a certain concept, it can be compared to the concept's reference documents. Based on the experiences of Information Retrieval, it is assumed that text documents can be compared by algorithmic means and a similarity measure can be calculated. If the similarity of the document in question compared to the reference documents is higher than a certain threshold, it matches the definition of the concept and can be considered to contain information about the same topic.

Thus, defining a concept through the full text of reference documents is more expressive than simple keywords. At the same time it preserves the flexibility to define any existing topic, simply by providing a number of reference documents. The number of reference documents is flexible as well, so a concept's meaning can be refined by extending the list of reference documents.

Documents on the WWW are exactly identified by a link, so agents need only exchange a set of links. Each agent can thereby retrieve the full document text at any time. If agent A wants to communicate a concept's meaning to agent B, it can simply pass a list of links to the reference documents to that agent. Any other document sufficiently similar to these reference documents can be assumed to match the concept.

In addition, the method of comparing the document's content (full text) is left open. How an agent actually implements the comparison between documents depends on the agent, so the concept is independent of the algorithms or architecture used to implement the agents. Agents are autonomous to interpret the definition of a concept in their own manner, yet the definition remains clear and it is exactly the same for all participating agents[8].

To summarize the above, a concept is a topic or area of interest which is defined by a number of links that point to documents about that topic on the WWW.

[7] See [Grube93] for a discussion of formal ontologies.

[8] This is in fact similar to how humans would handle such an issue, even if a definition is clearly given by a text, the interpretation of each person may vary.

4.2.2.2 The concept model

All the knowledge represented by a concept is stored in a concept node. When two agents exchange this knowledge, the data structure is clearly defined and must be known to both agents.

Similar to the links, a concept has an originating agent, which serves the concept to the multi-agent system. This agent is the concept's maintainer and acts as sole authority of the concept's definition, being the only entity who can add or delete reference links and change the concept data.

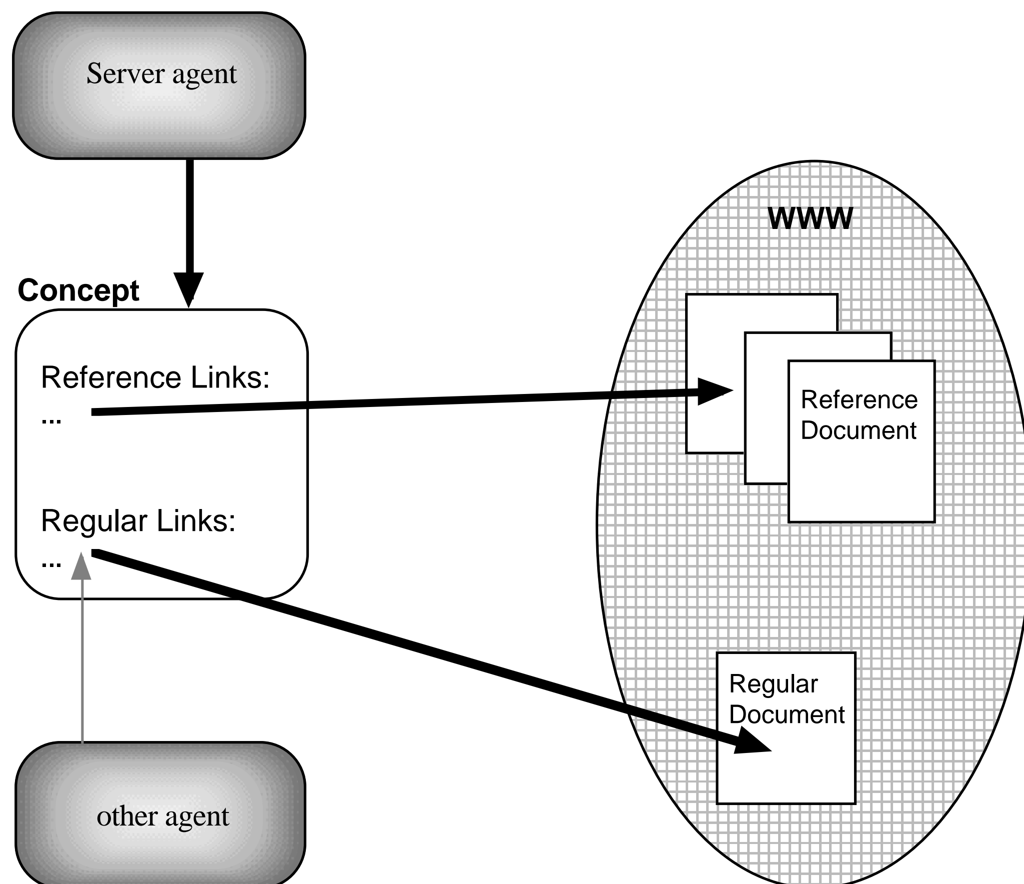

Figure 4.2 A Concept

The system's knowledge is dynamic by its nature. New links are continuously added (or old links deleted) by its agents. The concepts serve as predefined containers, into which new links can be categorized by a boolean yes/no classification. So in addition to providing the reference links that define it, the concept stores regular links to documents that cover the same topic. These regular links are not part of the collection defining the concept's meaning. To ensure that the definition of a concept does not change, the reference links are distinguished from links that serve only as information about the concept. Any agent in the sys-

tem can add such a regular link to a concept if it finds that the document matches the concept.

A concept node stores all the information of a concept. It has the following fields:

- **Name**: The concept's identifier, a single word (containing no whitespace)
- **Context path**: The concept's pathname - a chain of all the parent concepts (without the concept name)
- **Abstract**: An abstract text for the user, explaining what this concept is about
- **Keywords**: Keywords to identify the concept
- **Origin**: The name of the agent that this concept came from
- **Subtrees**: A list of all the subconcepts
- **Relations**: A list of all the related concepts (their full pathname in the tree)
- **Server Links**: A list containing the reference links that define the concept
- **Other Links**: A list with regular links (not used to define the concept)
- **My Links**: An extra list for agents to store their own links

A unique name identifies each concept when handled by the agents internally. It consists of the context path and the name (described in detail in the next section). If the concept does not contain any links, the keywords can be used as a simple definition. Ultimately however, the concept will be defined by the reference links it contains. The origin of a concept indicates which agent the concept came from.

To summarize, the model of a concept serves two main purposes: It is used to store the system's knowledge of the locations where information can be found and it is used to define a topic that agents can exchange information about.

4.2.3 The concept tree

Concepts are organized in a hierarchical tree. Each concept is represented by a node in the tree and has exactly one parent concept and up to several subconcepts. The path of parent nodes that lead to a concept from the tree root is called *context*, and clearly identifies a concept together with its name.

The concept tree may be seen as similar to a "weakly structured ontology" (as described by [Iwazu95]), which has no conceptual relations (like concept-value, class-instance, part-whole, etc.) except for the hierarchic connections.

The tree structure is based on a categorized topic hierarchy which serves as a guidance and navigation aid for a user who does not exactly know what he is looking for. The user can navigate through the concepts along the edges of the tree.

The further down a node is, the more specialized is the concept it represents. Likewise, the further up the node is (closer to the tree root), the more generalized is the concept. Thus, the hierarchic tree is a very simple semantic graph (similar to Usenet News hierarchy or

Yahoo’s topic categorization). The semantic of moving a branch down the tree is thus a “specialization” of the current concept. Correspondingly, moving a branch up the tree means getting to the context of a concept. It indicates the general topic area of how a concept is to be interpreted. Except for this context and the branch pathname, a concept does not inherit anything from its parent concepts. It is a separate entity, though in the context of the parents.

In addition to the hierarchical connections, a non-hierarchical connection can be established between two concepts to indicate the existence of a semantically related content. Such a connection is called *relation* and is always bidirectional (if A is related to B, then B is also related to A). It may exist between any two concepts in the tree. A relation simply indicates a similar topic in a different context and is not typed otherwise. It serves only as a navigation shortcut for the user and has no further significance or meaning in the tree structure.

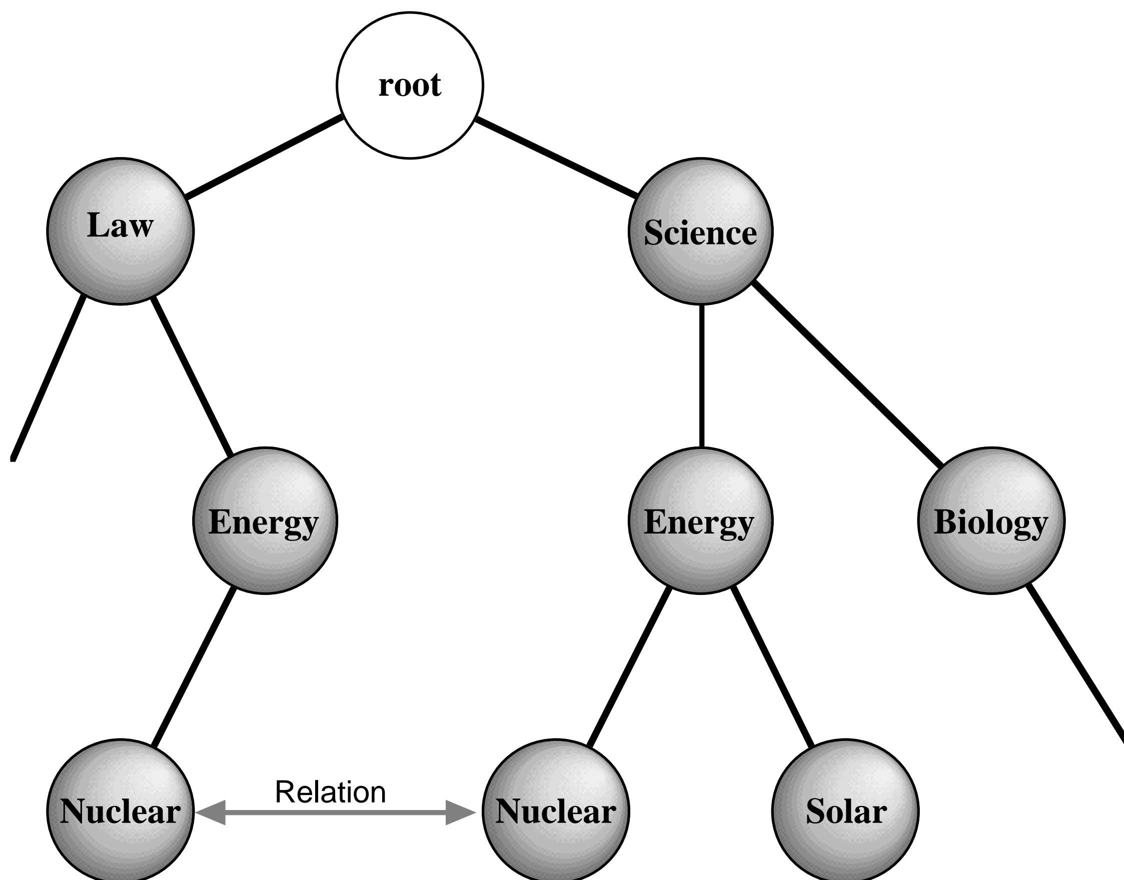

Figure 4.3 Example of a Concept Tree

For example “Energy” could be a subconcept of “Science” and in turn have the subconcepts “Solar” and “Nuclear”. The concept “Science” - “Energy” - “Nuclear” would be about

the scientific and technological aspects of nuclear energy. At the same time, another concept with the same name could reside in a different part of our concept tree, for example “Law” - “Energy” - “Nuclear” would be about the legislative issues of nuclear energy. Note that even though both have the same name, they are considered two different concepts, because their context path is different. As a result, they would be interpreted as having a different meaning. At the same time they are semantically related (though in a different context), so they are connected by a relation.

A concept tree captures one view of a certain information domain and is manually created by one or more (human) experts who have a good understanding of that domain. Different trees may exist and offer different schemes (topic hierarchies) of categorizing their contained concepts. In the multi-agent system, different trees may be served from different agents. An agent will always serve a complete tree (starting from the root). It is not possible for agents to just serve partial subtrees or single concepts. This ensures there is a single origin for each concept tree and thereby a single authority for the tree’s structure.

Agents other than the originating agent are not allowed to modify a tree’s structure nor to redefine concepts, they can only add links to already existing concepts. This is necessary because otherwise the tree would change uncontrollably, thus destroying the original categorization of concepts.

Users and agents need a definite handle to concepts to be able to exchange data with each other, they need to have the same understanding of a concept and therefore the same organization of concepts. This is granted through the usage of the same tree which has a single agent as authority to assure its consistency.

5 The CEMAS agent architecture

This chapter describes the architecture of the Concept Exchanging Multi-Agent System. A basic idea was to have a specialized agent for each main task. These main tasks can be identified as follows:

[1] The agents exchange concepts and links (which point to information on the WWW). This exchange and the concept tree needs to be managed by an agent.

[2] Each user should be represented by his own agent in the system, so that the agent serves as interface.

[3] Since new information is continuously being added to the WWW, a specialized agent should continuously search for new relevant documents matching existing concepts.

[4] To coordinate all these agents and enable their cooperation in the multi-agent system, a facilitator or broker agent is required.

Thus, the system consists of four different types of agents that cooperate with each other. It is based on the Java Agent Template (JAT) version 0.3, a framework for Internet-based software agents written in Java from Stanford University. At the time of implementation, the JAT was the only agent framework that met the requirements to build our multi-agent system upon: because of Java it is platform-independent, it is freely available (including the source code), provides the necessary basic agent functionality and well implemented (bug-free). The choice was made to avoid having to reinvent the wheel concerning agent functionality.

JAT agents communicate with each other in the Knowledge Query and Manipulation Language (KQML), an Agent Communication Language (ACL) defined in a draft by the DARPA Knowledge Sharing Effort. This ACL is based on a speech-act view of agents; what an agent “says and hears” (messages) and what it “does” (actions) is interconnected (see [Finin97], the idea of speech-act was initially described by [Austi62]). Therefore the agents will mainly be described from the point of view of communication, since this also describes what they do.

The agent paradigm assumed by KQML sees agents as “communicating attitudes about information” (querying, stating, subscribing, offering, etc.) and managing a knowledge base. This is meant as an external view of an agent, describing how it behaves towards the surrounding environment. Since agents are encapsulated autonomous entities, their internal operation may be completely different.

As a matter of fact, this is seen as one of the main applications for JAT agents: Being a wrapper around an internal system and enabling a common agent-based interaction with

that system, independent of what is inside (e.g. database, expert system or any other program). JAT provides a basic framework upon which such wrapper agents can be built.

This chapter will describe the Agent Communication Language (KQML) that was used, followed by an overview of the Java Agent Template and finally the description of the four CEMAS agents. It will focus on a view of the interaction between the agents, as this is considered the more important and therefore larger part of this work. As already mentioned in the previous chapter, the main goal was to provide a working framework, rather than an internal agent implementation.

5.1 The Knowledge Query and Manipulation Language

Agents communicate with each other by sending messages in KQML. This communication language was chosen for several reasons. First, it is already supported by the JAT. Even though it is only defined in a draft ([DARPA93]), there seems to be a large support in the scientific community[9], so it is regarded as a standard and already being used by a variety of agent applications. It was specifically developed as communication language for software agents and therefore seemed appropriate for the CEMAS application.

Note that communication is asynchronous and due to the agents autonomy, an agent may send a response to a message immediately or later. This peer-to-peer connectivity is an essential assumption of underlying agent architectures where they differ from classic client-server based architectures. Even if an agent is assumed to cooperate and give an immediate response to a message (and tries to answer as quickly as possible), agents should be able to handle a response that arrives quite a while later (possibly hours or days).

Agents are described as autonomous and deliberative, they are seen as having an intention and attitudes about information (believing, achieving, offering, stating, querying, etc.). The language was developed to reflect the need to communicate this. The agent paradigm adopted by KQML also treats an agent as if it were managing a knowledge base. As an encapsulated entity the agent hides its internal implementation from the environment, even though it interacts as if it were based on a KB. Thus, agents are also said to be managing a virtual knowledge base (VKB). So in the scope of agent communication (and in this paper), the terms knowledge or knowledge base should be understood in an abstract way. It does not mean that the agent's internal representation actually consists of a KB as defined by common AI.

Messages are sent with regard to such a virtual KB, commonly statements or questions about the content or requests to add or delete knowledge. A KQML message is also called *performative*, because it is usually a request to perform an action (speech-act). Most KQML messages also have a *content* statement, which is the direct object of the performative (in

[9] Additional papers discussing KQML were published, see [Labro96].

the linguistic sense). If the performative is *tell* for example, then the content is the statement being told.

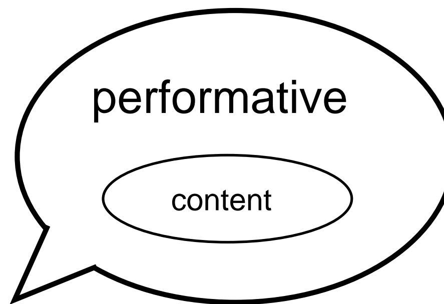

Figure 5.1 Two-layer Message

Agents talk about statements contained in their virtual KBs using KQML performatives, but the content statements can be encoded in a variety of other languages (KIF, Lisp, Prolog, etc.). This makes KQML independent of the actual content of the agent KBs and allows for a very flexible two-layer communication.

For example the performative *tell* may be used to inform another agent that a certain statement is in one's own KB, but the actual contained statement string can be in a representation language other than KQML.

Only the outer KQML layer and the performatives need to be defined a priori for the agent communication. The message content as well as additional information is set forth by the message parameters. In addition to the sender and receiver of the message, the parameters also indicate the language and ontology used to encode the content. Thus, agents can exchange knowledge in a flexible way, using different languages and ontologies.

A formal example of a typical KQML message is given below (the complete KQML syntax definition in BNF can be found in appendix B).

```
(performative :sender <word> :receiver <word>
:language <word> :ontology <word> :content (<expression>))
```

Most messages use the *content* parameter containing the statement about which the performative expresses an attitude. The language and ontology parameters are only used in combination with a content. They assign the representation language and the ontology[10] of the contained statement. The language parameter contains a keyword, which indicates the language syntax used to encode the content statement. The ontology parameter also contains a keyword, which indicates the ontology used to encode the content. The words,

[10]The agent communication language (KQML) uses a strict ontology. This is not to be confused with the discussion about ontologies to structure information on the WWW, which is a different issue in this paper.

expressions and constants used in the content statement must be a subset of the set defined by the ontology. The ontology defines the meaning of objects.

KQML contains a list of reserved performatives. Their meanings are defined in the draft specification. Agents are not required to support all listed performatives, but may use only a subset thereof. In addition, agents may extend this list with their own performatives as well.

Performative	Meaning
ask-one	Sender wants one of Receiver's answers to a question
ask-all	Sender wants all of Receiver's answers to a question
eos	end of a stream of responses to an earlier query
tell	the statement is in Senders virtual KB
untell	the statement is not in Senders virtual KB
register	Sender can deliver performatives
unregister	Sender does not (anymore) deliver performatives
insert	Sender asks Receiver to add content to its virtual KB
delete	Sender asks Receiver to delete content from its virtual KB
subscribe	Sender want's updates to Receivers response to a performative
unsubscribe	Sender does not want updates from Receiver anymore
sorry	Sender cannot provide a more informative reply
error	Sender considers Receivers earlier message to be mal-formed

Table 5.1 Reserved KQML Performatives

KQML messages are sent in plain ASCII text and are expected to be sent and received as a whole (not split up into parts) and in the same order. The table above shows a subset of the reserved performatives, those that are used by CEMAS.

The message parameters are field-value pairs, where a keyword identifies the parameter (starting with a colon “:”), followed by its value. The following table shows the reserved KQML parameters used by CEMAS.

Parameter	Value
:sender	actual sender (agent) of the message
:receiver	actual receiver (agent) of the message
:content	information (statement) about which the performative expresses an attitude
:language	name of the representation language of the :content parameter
:ontology	name of the ontology used in the :content parameter

Table 5.2 Performative Parameters

Throughout the rest of the paper, the term *performative* will be used to denote the command (a single word), whereas *message* will denote the whole message including parameters and content.

5.2 The Java Agent Template (JAT)

The JAT is a package of Java classes that provide some basic functionality necessary to write Internet-based software agents. It neither endows agents with specific capabilities beyond those needed for communication and interaction, nor does it impose any particular internal knowledge representation.

A primary application of the JAT is to “wrap” existing programs by providing them with a front-end that allows them automatically to communicate with other agents, sending and receiving messages, files and program code. Thus, such a wrapper serves as a layer that implements agent characteristics, the underlying program can interact according to the agent paradigm through the agent wrapper. Of course there need not be a separate underlying program. The functionality can be fully incorporated into the JAT architecture as well.

This section explains the basic features of the JAT and how its agents manage knowledge and communicate.

5.2.1 JAT knowledge

Since agents are seen as managing a virtual Knowledge Base, the data or information they handle are defined as their knowledge. Basic JAT agent knowledge consists of knowledge about resources. This resource knowledge is vital for every JAT agent as it contains information about the state of the agent itself and the surrounding environment. It also enables the agent to exchange messages and other data. Agents store this resource knowledge in their own virtual KB and exchange it with other agents, either directly or via a broker agent. If an agent does not find a resource inside its own virtual KB, it may ask other agents about it. This is explained in more detail in the next section (communication).

Every resource has a name, type and value. The name is used to identify a single resource. The JAT framework supports the following resource types: *address*, *class*, *interpreter*, *language* and *file*. A resource’s value is interpreted differently depending on the type.

Type	Value
address	host:port
interpreter, language, class	code_base class_name
file	base_url file_name

Table 5.3 JAT Resources

JAT agents reside statically on a host (they are not mobile) and exchange messages over a dedicated port. Thus, an address resource identifies an agent by its name. The value of this resource contains the host’s IP address and communication port number. Address resources have somewhat special status in the multi-agent system, because they are required even before communication can happen. An agent needs to know which other agents exist, to be

able to communicate with them. Therefore address resources are exchanged via an agent broker. At startup, each agent knows its own address and that of the broker.

The class, interpreter and language resources describe the name and location of a Java class file. An agent can retrieve such code and execute it. Due to the Java class loader the code location is transparent, an agent can load the Java class from a remote host as well as locally. Thus, the resource value consists of the code base (its location) and the class name. So even though JAT agents are not mobile, they are able to exchange (mobile) code. This feature is supported by the platform-independency of Java[11]. The interpreter and language classes are handled separately because they are necessary to interpret messages, explained in the next section.

File resources indicate the location of plain data files that agents can exchange (using protocols like FTP or HTTP[12]).

5.2.2 JAT communication

Communication between agents is asynchronous and peer-to-peer, meaning every agent can initiate messages to any other agent at any time. In theory an agent is considered autonomous, it can answer a message as it deems appropriate, possibly hours or days later, or not at all. In a cooperative multi-agent system it can be expected that an agent will try to give an answer as fast as possible, but even then the underlying transport layer (the Internet) can cause messages to be delayed or switched in order. Especially in a larger and further distributed system, this can add to the problem of several messages arriving at the same time.

The JAT framework provides a message buffer for incoming messages, they are stored in a queue and can be handled one at a time. Likewise there is a queue for outgoing messages, which are then sent one after another. Any extra functionality on top of that has to be provided by the additional agent implementation. The agent needs to be able to handle messages that are malformed (bad syntax), do not make sense (right syntax, but invalid or not understood), arrive in a mixed up order or late (they are not of interest any more).

JAT agents communicate in KQML. When a message is received, it is handled according to its main performative. Each of the messages used by JAT agents contain a performative and at least the sender and receiver parameters. The latter identify the sending and receiving agent by name.

Most of the messages also have a content parameter. The value of the content field holds the knowledge that is being exchanged. At the same time the performative expresses what should be done with that knowledge (e.g. added to the agents KB, deleted or changed).

[11]See the Java class documentation for ClassLoader ([Sun97]).

[12]The File Transfer Protocol (FTP) and HyperText Transfer Protocol (HTTP) are communication protocols used to transmit data on the Internet.

A content parameter is always supplied with a language and ontology parameter. The content statement may be encoded in any language, the default is KQML. The content statement is then interpreted according to its language and ontology.

The knowledge of JAT agents is implemented through resources. This knowledge is the object of their communication. JAT agents use “KQML” as encoding language for the content statement. The resource knowledge is defined by the ontology “agent”.

Knowledge about	Ontology to communicate it
resource: address, interpreter, language, class, file	agent

Table 5.4 JAT Knowledge

There is no formal exchangeable definition of the ontologies used (other than the messages), they are simply identified by their key name. An agent either “knows” an ontology or not. More specifically, an agent has an interpreter that can handle the symbols of that ontology (or it does not have the necessary interpreter). If it does, this knowledge is implicit (part of the code that interprets the message). In the KQML definition, the term ontology is used without a clear description of what an ontology is or can be. It may represent anything from a large system of entities or objects with different relations between them to a simple set of keywords with some parameters.

In the case of JAT and CEMAS, content statements are encoded with the KQML syntax. Likewise, the statements share the same form as a KQML message, a keyword followed by several parameters (field-value pairs). However the semantics are not that of a performative, but that of an ontological object. Such an object can have several properties with different values. The keyword names the type of object and the field-value pairs name the properties of the object.

Thus, the content expression of a message is interpreted as follows:

```
(object :property <value> :property <value> :property <value> ...)
```

The ontology “agent” contains only one object, a resource. Listed in the table below are all the possible properties of an object.

Ontology	Object	Properties
agent	resource	name, type, value, non-unique-name, unique-name

Table 5.5 JAT Ontology

An example of a JAT KQML message is given below.

```
(ask-one :sender MoneyPenny :receiver ANSagent :language KQML
:ontology agent :content (resource :type address :name JamesBond))
```

In this example an agent called “MoneyPenny” lacks the knowledge of an address resource. So it asks an agent called “ANSagent” for the address resource of the agent “JamesBond”. The content statement is encoded in KQML syntax, as indicated by the language parameter. The semantics of the content statement is defined by the “agent” ontology (which describes resource knowledge).

A special feature of the JAT agents is the ability to request and load the Java code necessary to handle the content statement, if either the language or the ontology of the statement are unknown. The necessary Java classes can be requested from the agent that sent the message. Using the dynamically loaded Java code, the message’s content can be interpreted. This feature is supported by the language and interpreter resources mentioned in the previous section, which are used to locate these class files. The possibility to load these classes from remote agents is not used by CEMAS agents, so it will not be explained further. Every CEMAS agent has all the classes it needs to handle its messages, so dynamic loading later on is not required (see also chapter 7 for implementation details).

A prerequisite for communication is the knowledge of the partners address (or even existence). The JAT agents require a broker agent also called Agent Name Server (ANS), to exchange their addresses. This broker serves as a central point of exchange for all the addresses. After initialization, an agent reads the broker’s address from an init file. Thereby it is possible to query the broker for a specific address or a list of existing agents. At the same time an agent registers its own address with the broker, announcing its presence to possible communication partners. After an agent knows the address of a partner, it will send the messages directly to that partner. The broker serves only as central address repository, it does not route or forward messages[13]. The ANS agent as provided by the JAT offers simple address exchange functionality. This was extended by additional broker features (also see section 5.3.3).

[13]A new version of the JAT supports such a routing/forwarding functionality, see the JATLite homepage at http://java.stanford.edu/

The following table lists all the messages that JAT agents use. The content statement is encoded in the KQML language (same as the outer wrapper message) and uses keywords defined by the ontology “agent”.

Performative	Content
ask-one	(resource :type <resource type> :name <resource name>)
tell	(resource :type <resource type> :name <resource name> :value <resource value>)
register	(resource :type <resource type> :name <resource name> :value <resource value>)
unregister	(resource :type <resource type> :name <resource name>)
non-unique-name	(resource :non-unique-name <agent name> :unique-name <agent name>)

Table 5.6 JAT Performatives

The performative “ask-one” is used to query an agent about a resource (commonly the broker), which should be answered by a corresponding “tell” message. An agent uses the “register” and “unregister” messages to indicate its own resources (e.g. address) to a broker. The difference between *tell* and *register* is that an agent registers only its own resources whereas knowledge about other agent’s resources is told (provided the agent has such knowledge inside its KB, as in case of the broker). This reflects the difference in an agent’s attitude towards knowledge. It “knows” about its own resources, but it only “believes” that resources of other agents are available (since that other agent may have gone offline without further notice in the meantime).

The last performative is only used by a broker, when an agent registers itself with a name that already exists. Since JAT uses the names to identify agents, they must be unique. Thus the broker will pick a new unique name for the agent and send a “non-unique-name” message back, which contains the new unique name.

All the messages need to have the sender, receiver, language and ontology parameters set accordingly.

5.3 CEMAS agents

To implement CEMAS, the basic JAT agent was extended with further functionality. The approach was to have a specialized agent for each of the main tasks in the system. An agent that implements the functionality to satisfy a task is said to offer a *service*. Theoretically it is possible for each agent to offer multiple services of different types, but for simplicity and easier implementation an agent in CEMAS currently provides only one type of service. Consequently there are four types of agents that provide the four different services: ConceptBroker, ConceptClient, ConceptServer and ConceptSearch. Depending on the service an agent offers, it uses a set of messages necessary to communicate the service-related knowledge. In addition to KQML communication, the agents (except the ConceptBroker) also retrieve information off the WWW using the HTTP protocol.

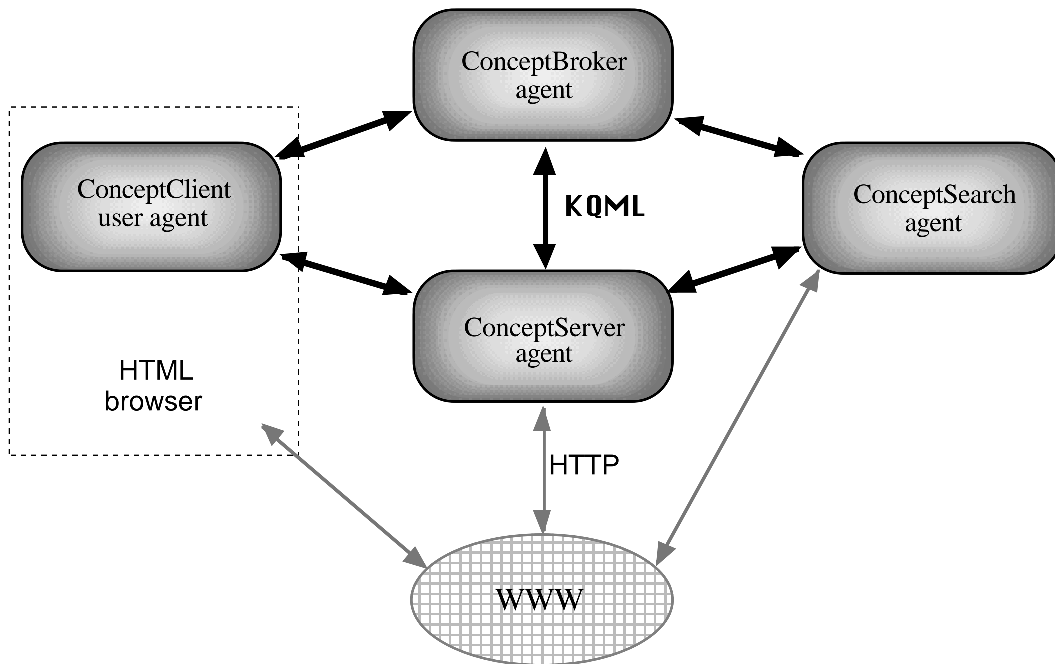

Figure 5.2 CEMAS Architecture

The above figure shows the four types of agents of CEMAS. They implement the four main tasks in the system. The ConceptServer manages the concept and link knowledge. Each user has his own ConceptClient agent as an interface to the system. The Concept-Search agent searches for new information and the ConceptBroker coordinates agent cooperation in the system. The agents are each explained in more detail in the following sections.

5.3.1 CEMAS knowledge

The CEMAS agents use three categories of knowledge: resource, service and concept. The functionality to handle knowledge about resources is inherited from the JAT agent framework. CEMAS extends this by adding the category of services and the category which handles concepts and links.

Knowledge about	**Ontology to communicate it**
service: ConceptBroker, ConceptServer, ConceptClient, ConceptSearch	broker
concept, link	concept

Table 5.7 CEMAS Knowledge

The second category is knowledge about the *service* an agent offers. A service can be one of four types: ConceptBroker, ConceptServer, ConceptClient or ConceptSearch and indicates that an agent provides a certain functionality associated with a service (the four services will be described in more detail in the following sections).

A service object has the following four fields: *name*, *type*, *agent* and *contact*. The name field identifies a service by its name. The service name is unique for each agent, an agent may not provide two services with the same name. Type is one of the four above-mentioned service types. The agent field contains the name of the agent that provides the service. Thus, a service is uniquely identified by the name of the providing agent (which is unique in the system) and the service name. Finally, the contact field contains the email of a person who is responsible for that service. This information may have several uses, but it is mainly motivated by the idea that someone (a human) must be responsible for the agent's actions. This would usually be the user who executes the agent program. Since an agent's actions depend on the services it provides, this information was included in the service knowledge. There is currently no way of preventing a malicious user or agent to provide false information in this field, but the idea was to have a way to contact someone for unintended problems and not to prevent intentional abuse.

More specifically, if an agent offers a service it means that this agent will respond to a set of messages in a certain fashion. The messages and their responses define the functionality (language-action perspective), again from a point of view outside of the agent. The functionality represented by each of these services is implicitly known by all the agents of the system. Consequently an agent only has to announce that it provides a service of a given type. It does not need to make a separate announcement for each of the messages associated with a service (e.g. that it intends to give a response to such a message).

The last category holds the knowledge about concepts and links, as described in chapter 4. CEMAS agents implicitly know how the knowledge is modeled, there is no formal specification being exchanged. Except for the ConceptBroker, all agents use and exchange this type of knowledge.

5.3.2 CEMAS communication

CEMAS agents use the same performatives to exchange all categories of knowledge (resource, service and concept). Only one language (KQML) is used, to encode both the outer message and its content. The choice to encode the content statement in KQML was made to keep the implementation simple, since the JAT already provides a parser for the KQML syntax.

Three different ontologies are used to communicate the knowledge. Resource knowledge is communicated with the ontology *agent,* the service knowledge with the ontology *broker* and the ontology *concept* is used to communicate knowledge about concepts and links. Since all the content statements are in the same language as the wrapper message, the messages differ only in the performative, the content statement itself and the ontology used for the statement.

When a message is received, an agent checks which ontology was used for the content and chooses the matching interpreter. The interpreter handles the knowledge coded in the content statement according to the performative (adds it to the KB, tries to answer a request, etc.). More specifically, it knows the objects and properties that are part of the ontology (the semantics of the language elements). Thus, it is able to interpret what the knowledge in the content statement represents (e.g. a service or concept). At the same time the interpreter knows the KQML performatives, so it can decide what to do with the knowledge (for implementation details, see section 7.2).

The following table shows a list of ontologies and the objects they define, as well as a list of all the possible properties of an object.

Ontology	**Object**	**Properties**
broker	service	name, type, agent, contact, available, remove
concept	link	name, concept, URL, description, owner
concept	concept	name, abstract, keywords, relation, linknumber

Table 5.8 CEMAS Ontologies

The broker ontology contains only a single object: the service. Only the concept ontology contains two objects, the link and the concept object[14]. Apparently these ontologies are very simple. Nevertheless they were not grouped into a single larger ontology, because they are better structured in three separate ontologies. The ConceptBroker does not use concept knowledge, so it does not need to know the ontology (the interpreters to handle it). Even though both resource and service knowledge is used by all agents, the service ontology offers more possibilities that are not required for resource knowledge (e.g. to subscribe for a service), thus they are separate.

The exchange of resource knowledge is inherited from the JAT framework. Since JAT agents require some kind of broker agent for the coordination of the system, this functionality was combined with administration of services in the ConceptBroker agent. As with the resources, agents can register a service they offer with the ConceptBroker. An agent that requires a list of services that are currently available in the system can subscribe for that type of service. Consequently, all CEMAS agents use and know the “broker” ontology. The following table shows the list of messages used to communicate with the ConceptBroker.

[14]Please distinguish that in the KQML ontology parameter, “concept” is only a keyword which indicates that the corresponding ontology was used. In the content statement however, the word “concept” describes a concept object.

Performative	**Content**
tell	(service :type <service type> :available ("<service name>" <service agent name> "<contact info (email)>" ["<service name>" <service agent name> "<contact info (email)>"]*))
untell	(service :type <service type> :remove ("<service name>" ["<service name>"]*))
register	(service :name "<service name>" :type <service type> :contact "<Contact responsible person (email)>")
unregister	(service :name "<service name>")
subscribe	(service :type <service type>)
unsubscribe	(service :type <service type>)

Table 5.9 Performatives, Object: service

Register and unregister are used by an agent to signal the availability of a service provided by itself to the ConceptBroker. To stay continuously up-to-date about available services of a given type, an agent uses the subscribe message. The counterpart unsubscribe serves to signal the end of interest for a service availability. Tell and untell are used by the ConceptBroker to inform agents that have subscribed for a service about the changes.

The third ontology, *concept*, defines the vocabulary necessary to exchange the concept knowledge between the agents. This ontology defines two object types, link and concept.

Within the system, the ConceptServer agent stores and manages the exchange of the concept and link knowledge. Agents can request concepts and links, and insert and delete links from existing concepts.

Performative	**Content**
tell	(concept :name <full concept name> :abstract "<concept abstract>" :keywords "<concept keywords>" :linknumber <no. of links> :subtree (<subconcept name> [<subconcept name>]*) :relation (<related concept name> [<related concept name>]*))
ask-one	(concept :name <full concept name>)
ask-all	(concept :keywords "<concept keywords>")
eos	(concept :keywords "<concept keywords>")

Table 5.10 Performatives, Object: concept

Agents can ask for a single specific concept (identified by its name) or for all concepts matching a list of keywords. The ConceptServer responds by telling the appropriate concept, or several concepts followed by an end-of-stream message. To get the root of a concept tree, the concept with the name “root” is requested. The concept tree is expected not to change (or rarely), so there is no untell message for concepts. In addition the ConceptServer would need to keep track of every message sent out, to be able to invalidate it.

Agents can ask for all the links of a concept, which the ConceptServer will reply with a tell reply message for each link. They can also insert new links into existing concepts and delete links that they own. Thus the ConceptServer can be a depository of all the links of an agent.

Performative	Content
tell	(link :concept <full concept name> :type <links agents service type> :name "<link name>" :url <link url> :description "<link description>" :origin "<user id (email)>")
ask-all	(link :concept <full concept name>)
insert	(link :concept <full concept name> :name "<link name>" :url <link url> :description "<link description>" :origin "<user id (email)>" :passwd "<user authorization>" :service <agent service type>)
delete	(link :concept <full concept name> :url <link url> :passwd "<user authorization>")

Table 5.11 Performatives, Object: link

A complete list of all the supported messages can be found in appendix C.

Note that the names of the ConceptServer and ConceptClient agents may be misleading, because they suggest a client-server communication architecture, which is not the case. The communication is still asynchronous and peer-to-peer, all agents are essentially equal communication partners (and autonomous to decide if and when they want to send a message). The ConceptServer's name only reflects the fact that it serves the concept knowledge to the system. The ConceptClient's name derives from its role of being a client software for the user to access the system.

5.3.3 The ConceptBroker agent

The ConceptBroker agent is a central depository for resource and service knowledge in the multi-agent system. Agents that want to make this knowledge available to the system will register it with the ConceptBroker. Likewise, agents that lack such knowledge and want to find out about it will ask the ConceptBroker by default. The ConceptBroker serves as the backbone of the multi-agent system's architecture. Therefore at least one ConceptBroker is required to run continuously as part of the cooperative system. Since CEMAS is a small system with few agents, currently only one ConceptBroker is used.

In the CEMAS system, the ConceptBroker is the first agent to be started. Other agents can then join and leave the system dynamically. During initialization, an agent reads the ConceptBroker's address from an initialization file. Then it registers itself (its own name resource) with the ConceptBroker. The ConceptBroker ensures the name of an agent is unique when registered and assigns a new unique name in case of a conflict.

As a next step, the agent registers the service it offers. It can then subscribe for services of a given type (ConceptServer, ConceptClient or ConceptSearch). In response, the Con-

ceptBroker will send the inquiring agent a list of all the agents that offer the subscribed service. Furthermore the ConceptBroker will send an update whenever another agent that offers such type of service becomes newly available or unavailable.

Before going offline or shutting down, an agent should properly unregister all of its resources and services, since they thus become unavailable to the system. If for some reason an agent does not properly unregister them, the ConceptBroker will automatically remove knowledge connected to an agent from its KB if the agent can no longer be reached. This ensures that the ConceptBroker's KB does not become obstructed with out-of-date information.

To summarize, the knowledge of a ConceptBroker agent is the information about the agent name resources and services in the multi-agent system. Like all agents in CEMAS, it supports messages using the "agent" and "broker" ontology to communicate this knowledge.

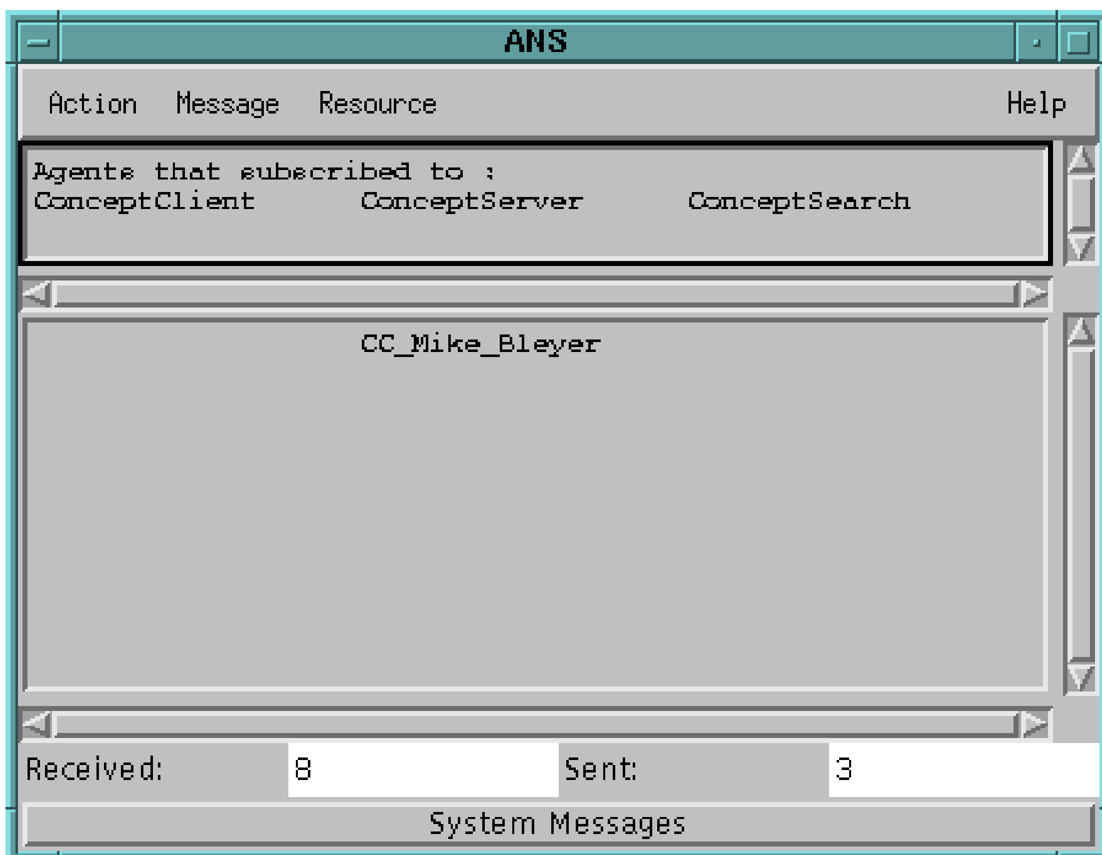

Figure 5.3 ConceptBroker Window

The ConceptBroker agent is implemented as a Java application. Since it is not designed as an interactive agent, but rather one that runs in the background, it only displays some sta-

tus information. In its main window it displays a list of all the agents that have subscribed for one of the three services. The above example shows that an agent named "CC_Mike_Bleyer" has subscribed for updates of services of the type ConceptServer.

5.3.4 The ConceptServer agent

The ConceptServer manages all the knowledge about concepts and links. More specifically it stores all this knowledge for the other agents in the system and thereby facilitates the exchange thereof.

Knowledge managed by the ConceptServer is considered public (except for the link passwords), there are no private concepts or links in the CEMAS system. The system can be used to manage and organize a collection of URL bookmarks, but the intention is to share them and not keep them private only for a single user.

A ConceptServer agent always manages a complete concept tree. The data is stored on the local filesystem and served to other agents. The ConceptServer agent is also the single responsible maintainer of the tree structure and the only one that can change the tree, adding or deleting concept nodes. In addition to the concept tree, the ConceptServer provides the reference links that define the concepts. These reference links cannot be changed by other agents and have the ConceptServer as their origin. Other agents can still add their own links to a concept which the server will save and redistribute, however they are distinguished from the reference links through their origin. These links are merely saved as users' preferences (for the user agents), and as additional suggestions to the reference links. Agents can delete links if they own them (if they are the originating agent).

Having a server agent as the single source for the concept knowledge also has two technical reasons. The ConceptClient agents cannot store data locally, since they are implemented as Java applets, which cannot access the local file system due to applet security restrictions. Therefore the ConceptClient agents need to store their knowledge somewhere else (at the ConceptServer). A central source also has the advantage of all the systems knowledge being available to every agent all the time, the users are not required to keep their agents running so their knowledge can be shared with others.

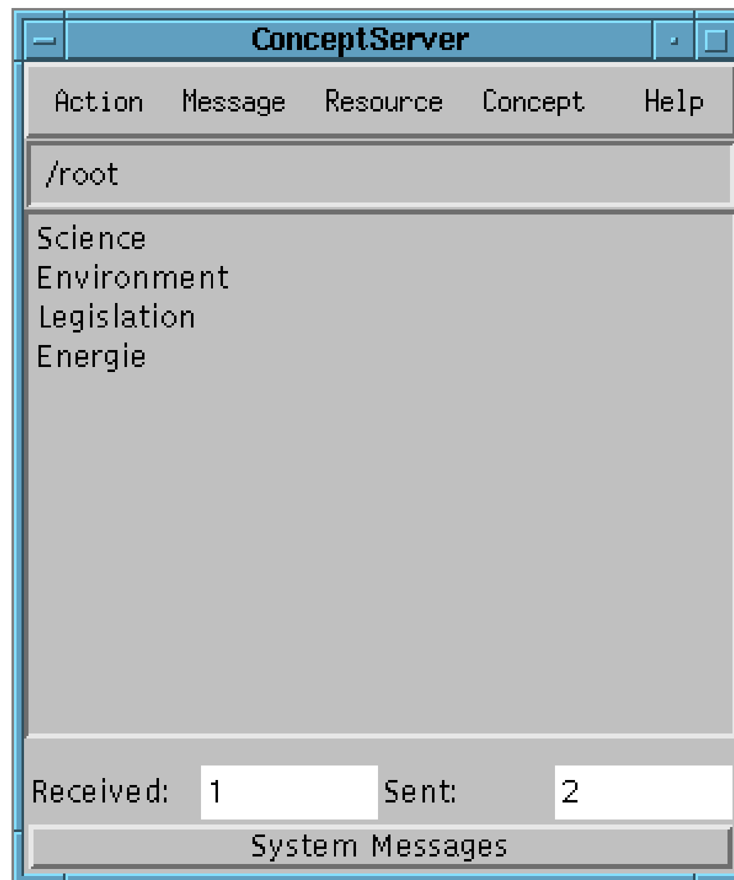

Figure 5.4 ConceptServer Window

The ConceptServer agent allows its owner to edit the concept tree through its interface. The main window displays a node in the tree with the according subtrees, much like a file dialog window. Two menus allow the user to edit the tree data (add or delete subtrees) and the concepts content (the links and description).

5.3.5 The ConceptClient agent

Each user runs his own copy of the ConceptClient agent, which acts as a bidirectional front-end interface. In one direction, the agent displays the information received from the system to the user. In the other direction, the agent represents the user within the system and carries out the user's information requests by communicating with other agents.

The agent is implemented as a Java applet, so it can be incorporated into a Web-browser[15] window. This has the advantage that the user does not need to install a Java runtime environment, but only a Web-browser (which is required anyway, to view the HTML documents). At the same time an applet enables a closer interaction between the agent and

[15]The Web-browsers that support Java applets and that have been tested are Netscape Navigator and MS Internet Explorer.

the Web-browser by means of the applet-specific methods. The ConceptClient agent applet is loaded into the user's Web-browser from a WWW-server.

Due to Java's security restrictions, applets can only open socket connections to the host that they were loaded from. As a consequence, the ConceptServer and ConceptBroker agents need to run on the same host as the WWW-server that the ConceptClient agent was loaded from, since it needs to communicate with both.

Furthermore, applets cannot write data to the local filesystem, so all the data is passed to and stored by the ConceptServer agent. The central storage has the added advantage that the knowledge about information gathered by all agents is available all the time, the user's ConceptClient agent does not need to run continuously to provide its knowledge.

At initialization, the agent asks the user for his email address and a password. An email address has the advantage of being a unique name for the agent and at the same time identifies the links in the concept tree as belonging to a certain user. The password is used to grant write access (deleting or changing) to the link owner.

The agent then connects to the system, registers itself with the ConceptBroker and subscribes for available ConceptServers. The concept tree root is requested from the ConceptServer agent and displayed in the agent window.

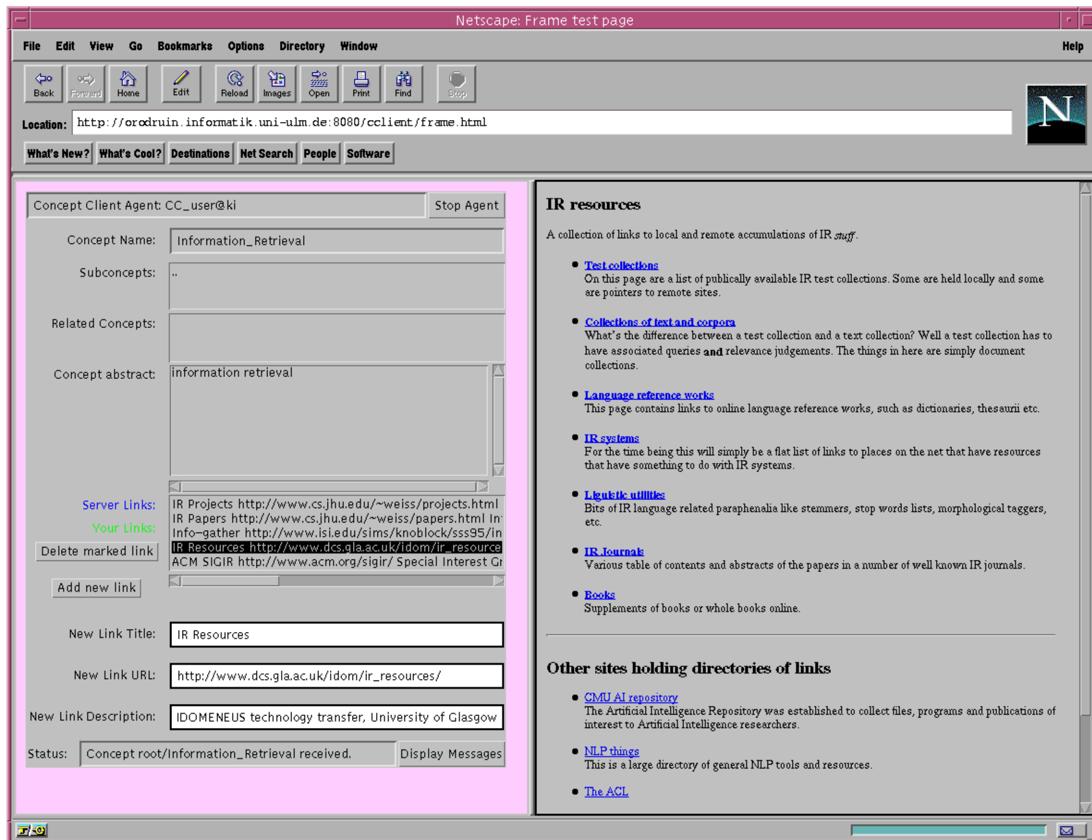

Figure 5.5 ConceptClient Window

The browser window is split into two frames, the left frame containing the agent and the right frame containing the document pointed to by a link. If the user clicks on a link in the concept list, a request function of the browser is called to fetch the URL and display that document in the right frame.

The agent window on the left presents all the concept information. Starting at the top with the current concept name, the subconcepts, the related concepts and the concept abstract. The user can navigate through the concept tree by following the hierarchical branches up or down. In addition, it is possible to move directly to related concepts, by clicking on an item in the corresponding list (thus bypassing the tree hierarchy). All the links of the concept are displayed in a list. The user can add his own links to this list and delete them if he feels they no longer belong there. Links of other agents can not be modified or deleted.

5.3.6 The ConceptSearch agent

This agent continuously searches the WWW for new available information. It reflects the dynamic nature of the WWW, so the system's knowledge about information can grow and new links are added, even if the users do not add links by themselves.

The search is intended as a continuous background task, not a one-time response to a query. The ConceptSearch agent searches for new links for each concept in the tree. If new links are found to be matching a concept, they are sent to the ConceptServer to be inserted into the tree. The ConceptSearch agent may use the data captured in a concept as an aid for the search or to compare possible new links to decide if they match the concept topic.

It can use the concept name, the context, the keywords and the documents pointed to by the reference links as a definition of the topic. Furthermore it can use the links (only reference links or all available) as a starting point for a search of new links.

In a first implementation, the approach is to query one of the WWW search engines (in this case AltaVista) with a set of keywords (concept name and keywords) to retrieve a list of documents that are likely to match the concept. The documents from this list are then retrieved and classified as to whether they fit into the concept or not.Thus, a new link to a document can be added to a concept after a simple boolean classification (matching/not matching).

A concrete implementation of a ConceptSearch agent that uses such a classification algorithm was realized in another project (see [Ferna97] for details). It uses the list obtained from the search engine to retrieve a collection of possible new documents from the WWW. It compares these new candidates by employing a classification engine from classic Information Retrieval, the SMART system. The classification engine is used to calculate a vector space representation of each of the new documents. They are then compared to the reference documents of the concept. If the similarity is higher than a certain threshold, the comparison results in a match and the agent adds the link of the new document to the concept.

A number of different search agents may work in parallel on the same tree. Since there are different methods to search the WWW (using search engines or following links) and algorithms to classify documents, each may use a different approach (see also chapter 8 for a discussion).

6 Example

In this chapter an example setting will be introduced, to illustrate the functionality of the system. Please note that the current agent implementation is a working prototype, but not a complete application. It serves to test the concept model and communication architecture. Several improvements may be necessary to increase the system's usefulness, (see chapter 8). One particular extension that was added during the time of this project by P. Fernandez Presa will be introduced here, to show that and how it is possible to extend the CEMAS architecture.

The example setting consists of one of each of the four agents and a small concept tree. The next section will introduce the sample concept tree and the second section will show a sample session with the messages that are exchanged between the different agents.

6.1 Example concept tree

The following example concept tree is an excerpt from a larger tree used to test the multi-agent system. It was used during this project to handle all the information on the WWW that was relevant or of interest. Note that the structure of a concept tree depends mostly on the content and the user constructing it. There is no absolute view of how to optimally categorize a certain set of topics. In addition, this paper does not deal with the content of concept trees and how to organize it best. The emphasis is on having a possibility to fit in all of the diverse information, thereby sacrificing an exact categorization (if such a categorization is possible at all). This example will show one example of how it can be done. Please also note that the naming of the concepts serves only as first indicator to what they are about, it is purely intuitive. The definition of a concept is given by its keyword list and reference links.

The example tree presents some concepts from the field of multi-agent systems. The tree is rooted at the concept of "Computer Science", since this is the context of the topic. Note that concepts closer to the tree root do not necessarily contain many links, since they are not specific enough for that to make sense. They are mainly used to group other concepts together and to define the context of their subconcepts.

In this case, the "Computer Science" context sets the bounds for the concept "Language", since we are only interested in formal languages and issues that have something to do with a computer implementation and not the complete field of languages. The subconcept "Communication" captures the languages that are used for communication, as opposed to programming languages for example. Finally, "Agent" captures the set of communication languages that are suited or intended for agent communication. This concept contains links

to information about KQML for example. The concept “Ontology” is another specialization of languages, in this case it is not specialized further.

In the second branch of the example, the concept “Artificial Intelligence” captures other issues of agent systems, the “Distributed” nature of “Multi-Agent Systems” and the “Machine Learning” aspect. The short grey line between the concepts “Agent” and “Multi-Agent System” indicates a relation. This means that the two concepts are similar or may have a lot in common, even though the emphasis of “Agent” is on communication issues and the emphasis of “Multi-Agent System” is on DAI issues.

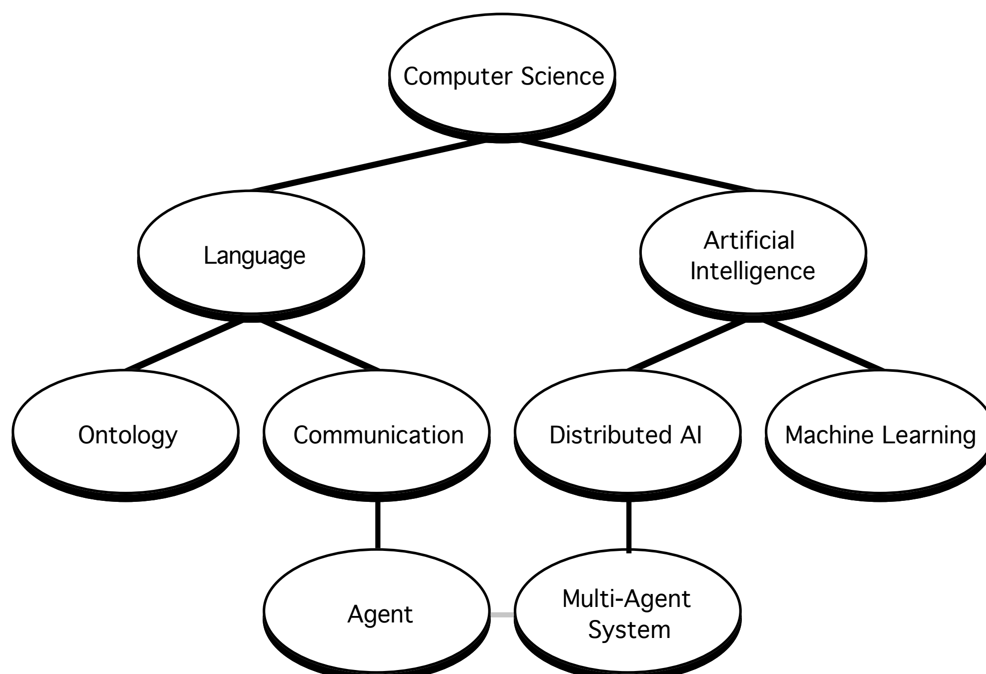

Figure 6.1 Example Concept Tree

Of course this tree shows only a fraction of concepts, there are more issues relevant for the topic area of agents.

6.2 Example session

For this setting, we will assume that one ConceptBroker and one ConceptServer agent are already running. The ConceptServer has registered itself (its name) and its service with the ConceptBroker. The first section shows the interaction of a ConceptClient agent with the system and the second the interaction of a ConceptSearch agent. In the following figures, the sender and receiver parameters have been left out of the messages for a better overview. The respective sending and receiving agents are indicated with icons.

6.2.1 Example ConceptClient session

The ConceptClient agent is loaded as a Java applet from a WWW-server which must be running on the same host as the ConceptBroker and ConceptServer agents. The user is presented with two fields where he has to enter his identity (email address) and a password. These two are used to mark the links that the user inserts into the concept tree. The user's mail address is also used as name for the agent (preceded by "CC_"), since that automatically leads to a unique agent name in the multi-agent system.

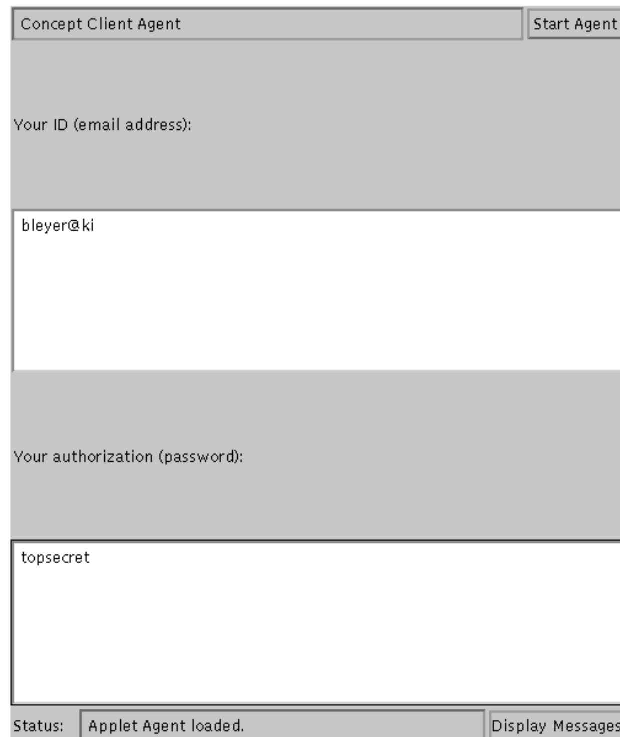

Figure 6.2 Example ConceptClient Window at Startup

After a click on the "Start Agent"-button, the ConceptClient agent connects to the ConceptBroker to register its name and service, thereby announcing its presence to the system. The address of the ConceptBroker is received from an init file, it is the only address that the ConceptClient initially needs to know to connect to the system. In the example below, the name of the ConceptClient agent is "CC_bleyer@ki" and the name of the ConceptBroker agent is "ANS". Note that the ConceptClient agent uses different ontologies for the two pieces of knowledge (its address resource and its service type) that it communicates.

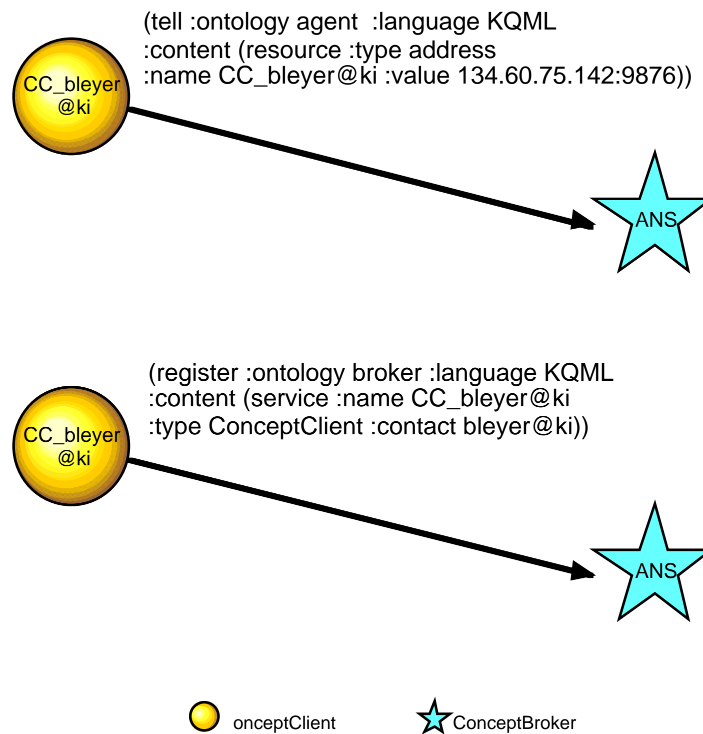

Figure 6.3 Example ConceptClient Communication: startup

In the next step, the ConceptClient asks for an available ConceptServer, from which the concept tree can be retrieved. More specifically it subscribes to be notified about all the available services of the type “ConceptServer” (which are known to offer a concept tree). The ConceptClient receives the name of a service (“AgentInfo”, in our example there is only this one) that is available. Along with the name of the service it receives the name of the agent providing it (“KIServer”) and the contact information of the administrator (his email address). Note the difference between the name of the service (“AgentInfo”) and the name of the agent that provides it (“KIServer”).

Now the ConceptClient knows where to get the concept tree from, but it does not yet know the ConceptServer agent’s address resource. Consequently, it asks the ConceptBroker again for the appropriate answer.

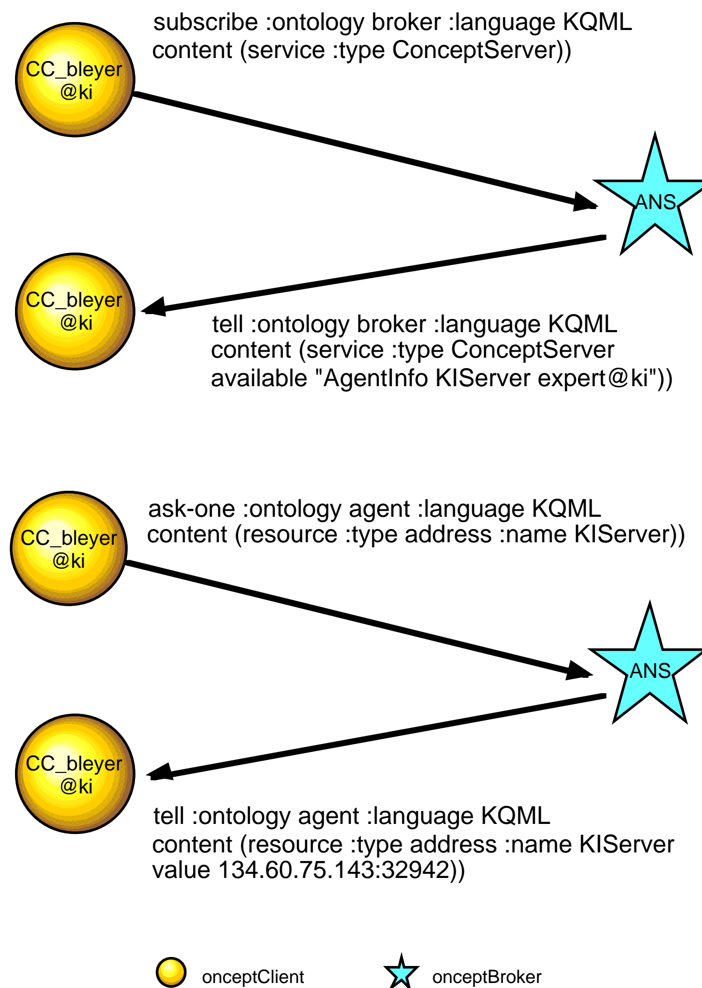

Figure 6.4 Example ConceptClient Communication: address lookup

Once the ConceptClient has received the address of the available ConceptServer agent, it can start communicating with it at the corresponding host IP address and port number. At first, it requests the root of the concept tree from that agent.

At the time of receiving the request, the ConceptServer does not yet know the ConceptClient agent's address resource[16]. Since the two agents have never communicated with each other before, they never needed to know each others address. The ConceptServer only knows the sending agent's name from the message (the sender parameter is omitted in the figures and indicated by icons instead). So before it can answer the request for the root concept, the ConceptServer needs to find out about the address resource of the ConceptClient.

[16]Please note that at the transport level (TCP/IP), the receiving agent knows from which host the connection came from. Thus, it could answer it directly without having to ask a broker or name server agent for an address, but there are two reasons not to. A message may be forwarded by an agent, in which case the sending host is not equivalent to the original sender (indicated in the message itself by the "sender" parameter). KQML makes no assumptions for the actual transport layer, so in an environment other than the Internet (TCP/IP) it may not be possible for an agent to know the originating host of a message directly.

Thus, it asks the ConceptBroker for the address resource and then answers the ConceptClient's request.

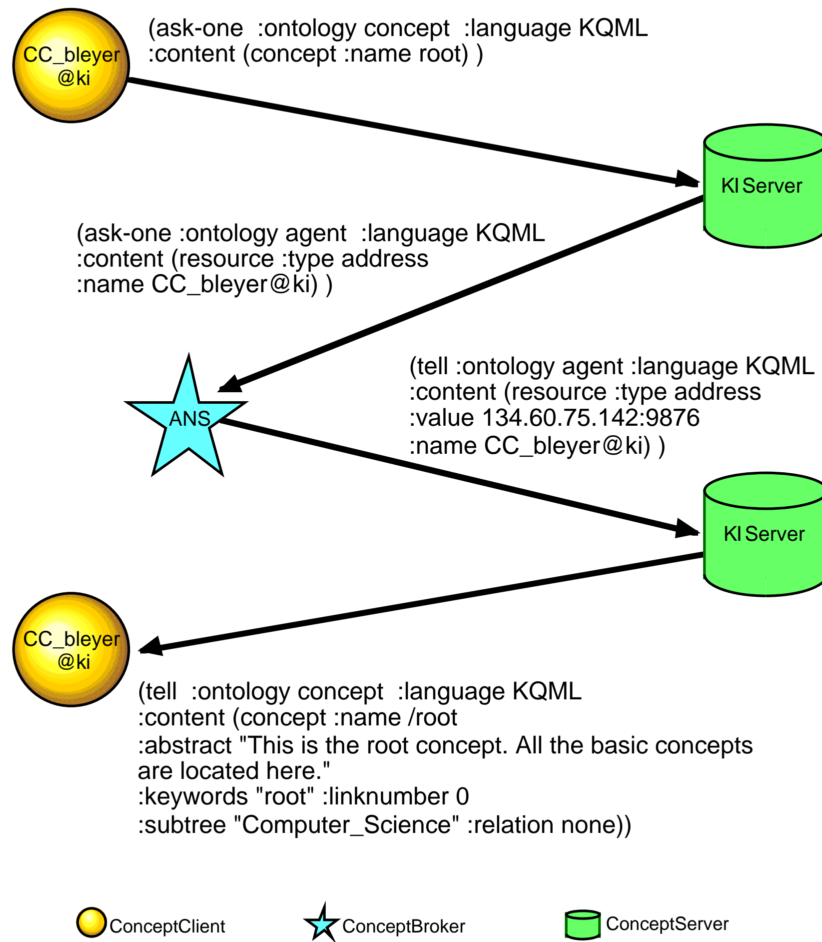

Figure 6.5 Example ConceptClient Communication: root concept

The first concept requested by the ConceptClient agent is always the root of the concept tree. This root concept is simply a container for all the other basic concepts. Once it has received a concept, the agent displays it in the interface window (see Fig. 5.5).

Depending on the user's actions (mouse clicks, etc.), the agent requests further concepts and their links as the user browses the concept tree. The figure below shows the request of a concept named “Language”.

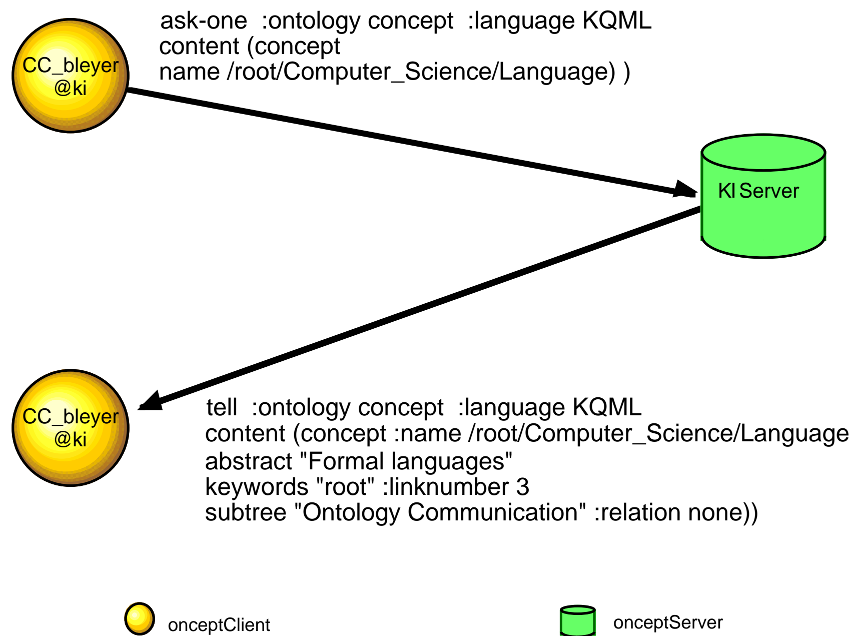

Figure 6.6 Example ConceptClient Communication: concept

After having received the concept node, the ConceptClient checks if it contains any links, indicated by the linknumber parameter. In case it does, it requests all the links of the current concept and receives each link in a separate message (the example below shows only one link).

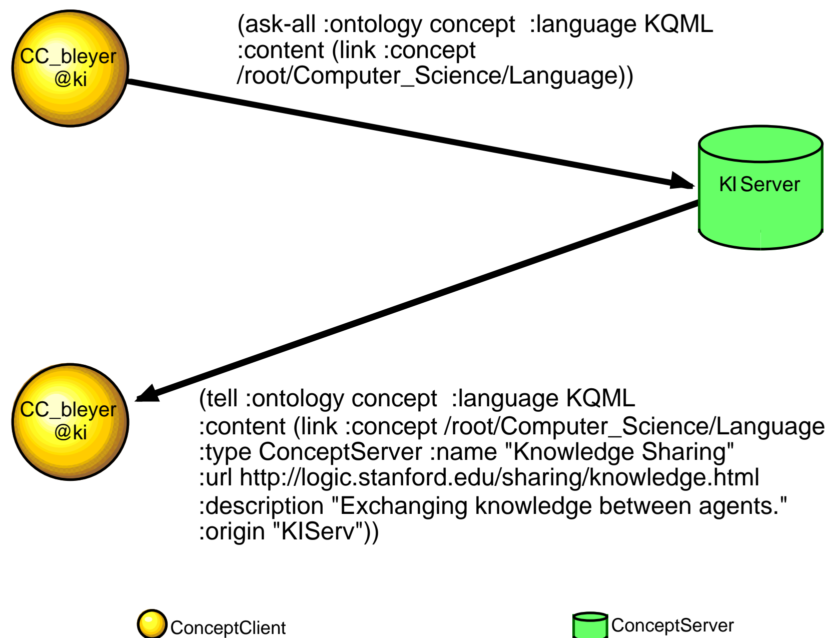

Figure 6.7 Example ConceptClient Communication: links

Finally, our example user has found a new document while browsing the WWW, which he thinks matches the existing concept in the tree. He adds a link to that concept and the ConceptClient agent sends the appropriate message to the ConceptServer. The user's identity (email) is supplied as parameter, to mark which user has inserted the link. The password is a simple protection to make sure only the link's owner can delete it. In addition, the identity can be used to establish a user profile, which can be derived from the set of links added by a single user. The service type serves as an orientation for the ConceptServer, it can group together links added by users (ConceptClient agents) or ConceptServer agents.

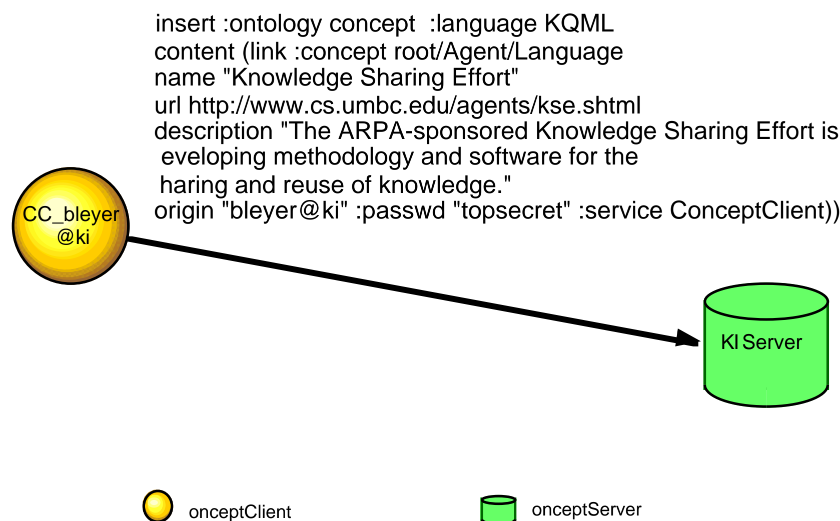

Figure 6.8 Example ConceptClient Communication: insert

6.2.2 Example ConceptSearch Session

The ConceptSearch agent runs like a background process, similar to the ConceptBroker and ConceptServer. Its continuous task is to search for new links to insert into the concepts of the tree.

When started by its administrator, that person's identity (email address) and a password are passed as command-line arguments to the agent. Again, these two are used to mark the newly found links that the agent inserts into the concept tree.

The ConceptSearch agent's name is passed as another command-line parameter (in our example "SmartSearch"), as opposed to the name of the ConceptClient, which is derived from the users email address. In the following figures, the names of other agents are the same as in the above example.

Since the initial communication is equivalent to that of the ConceptClient, the Figures are not listed again. After startup, the ConceptSearch agent connects to the ConceptBroker to register its name and service. It subscribes with the ConceptBroker for available "Con-

ceptServer” services and then connects with our example ConceptServer “KIServer” to retrieve the root of the concept tree.

For all the concepts contained in the tree, the ConceptSearch agent cycles through the same procedure. First, it requests the concept (in our example “Language”).

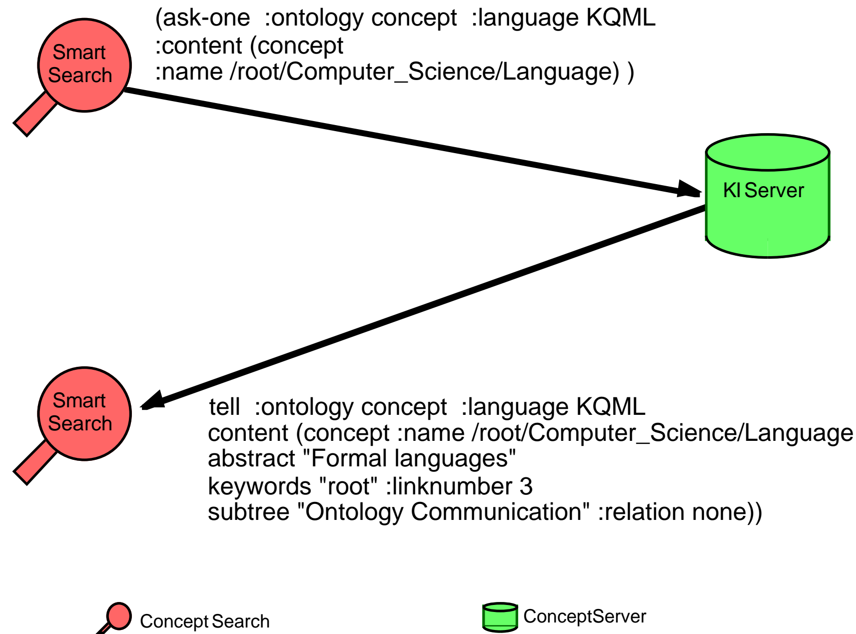

Figure 6.9 Example ConceptSearch Communication: request concept

Then it requests the links of that concept. Since reference links define the concepts meaning, the corresponding documents are retrieved from the WWW and cached locally for processing. Their content provides the base for a comparison and classification of newly found documents. Whether a link is a reference link or not can be derived from the service type parameter. If it is set to “ConceptServer”, then it is a reference link. If it is set to “ConceptClient” or “ConceptSearch”, then the link was added by another agent and is thus not a reference link.

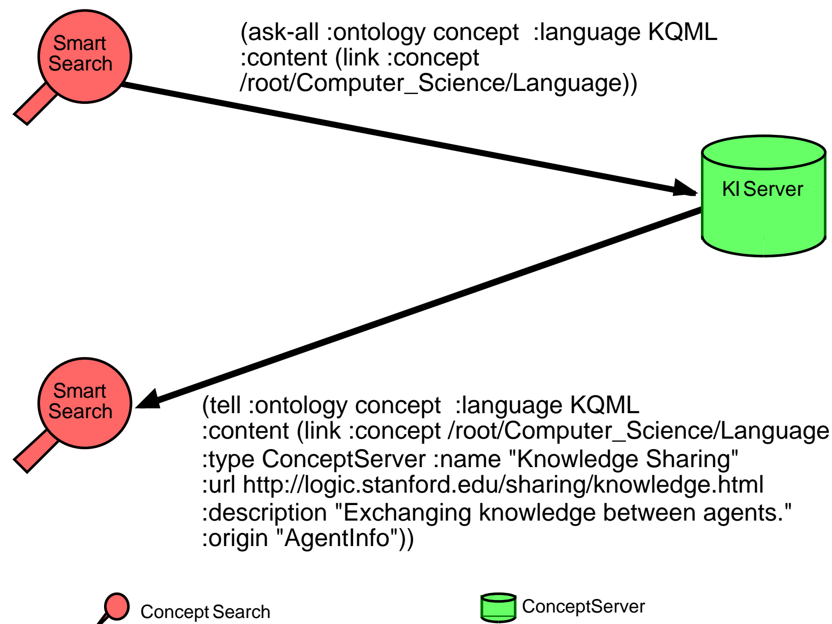

Figure 6.10 Example ConceptSearch Communication: request links

The extension of the CEMAS ConceptSearch agent developed by P. Fernandez Presa (see [Ferna97] for details) uses the SMART Information Retrieval engine to calculate a Vector Space representation of the retrieved reference documents.

A WWW search engine (in this case AltaVista) is queried with the concept's keywords. As a result, it returns a list of links to possible new documents of interest. The documents of this result list are retrieved from the WWW and processed by the SMART IR engine (in this example the first 10 items from the list are retrieved). If any of them match the set of reference documents (their similarity is above a certain threshold), their link is added to the concept.

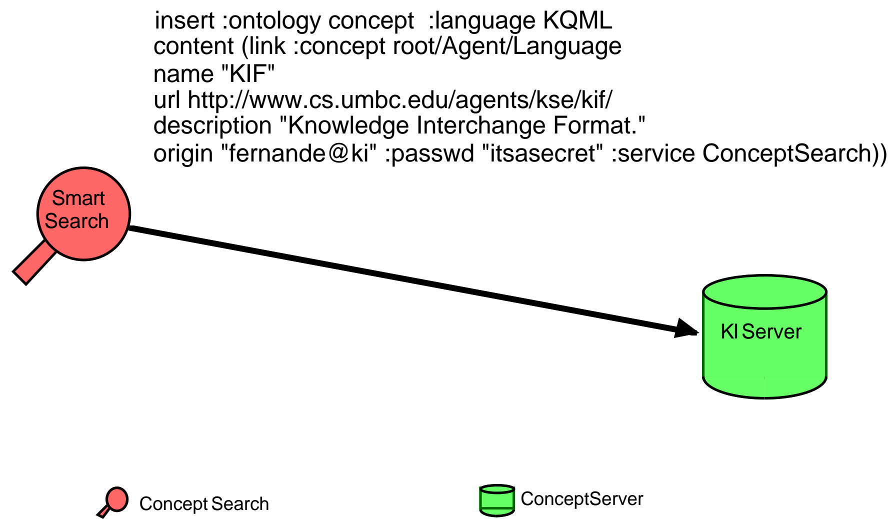

Figure 6.11 Example ConceptSearch Communication: add new link

The ConceptSearch agent used in the current example currently executes does only a single run through the concept tree. A repetition of this task has to be initiated manually after a reasonable time, when it can be expected that the WWW search engine will return new results (e.g. one week).

7 Implementation

In this chapter an overview of the Java class hierarchy will be given, to illustrate the agent structure. The overview will explain the important classes of the JAT package and the CEMAS extensions of it. The Java class packages are listed in appendix F.

The agents are written in Java and based on the Java Agent Template. Some of the JAT classes have been modified and extended to support necessary functionality for CEMAS. To compile the CEMAS agents, these modified JAT classes are needed. However the basic architecture of the JAT remained the same.

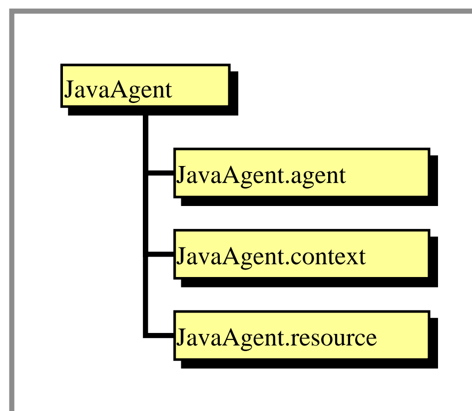

Figure 7.1 JAT Packages

The Java class files of the JAT are grouped into the three packages `agent`, `context` and `resource`. Each of the CEMAS agents uses classes from the same three packages. The necessary JAT classes are subclassed by CEMAS. Thus, they inherit the functionality of the JAT agents while extending this by some of their own.

7.1 JAT agent overview

This section explains the startup process of a JAT agent, to give an overview of the role of main classes in the order they are instantiated.

An agent is started by executing its corresponding `AgentContext` class file, thereby instantiating it. The AgentContext class sets the parameters which define the agent's environment (agent name, pathnames, init files, communication port, etc.). Some of these parameters can be overridden on the command-line when starting the agent. In case of a stand-alone agent, the context class also contains the *main()* routine which reads the command-line arguments. In case of an applet, the agent starts with the *init()* method and extracts the parameters from the applet tag.

The figure below illustrates how the further parts of the agent are started. The context instantiates the `Agent`, the `AgentFrame` and the `CommInterface` classes. The CommInterface handles the agent's communication on the transport level (e.g. TCP/IP sockets) while the AgentFrame instantiates and controls the `InterfacePanels` that make up the GUI for the user.

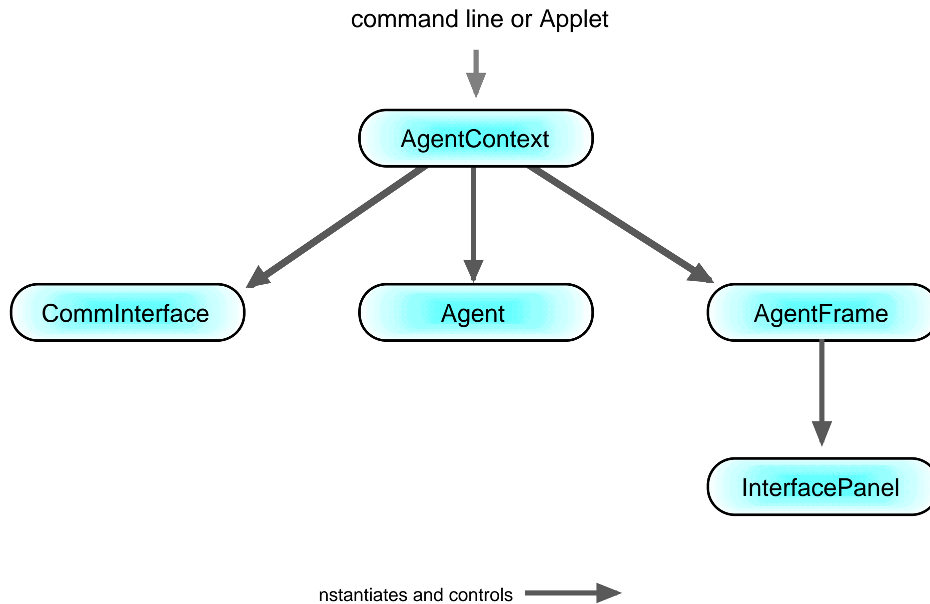

Figure 7.2 AgentContext Startup Class Instantiation

The agent itself is characterized by a set of resource objects. It instantiates a `ResourceManager` to handle and store these resources. Resources represent the internal knowledge of the agent and can be either data or Java class definitions (program code). Each resource type has a container class that provides the storage for it. These containers are subclasses of the `RetrievalResource` class, which provides synchronized access to the resources stored within. This is necessary because the agent uses multiple threads which may try to access the same resources. Resources fall into one of six categories: `Addresses`, `Services`, `Languages`, `Interpreters`, `Classes` and `Files`.

At startup, the ResourceManager controls several resources. The agent needs to know the KQML language from the start, so the Languages class contains the `KQMLmessage` class to store and parse the messages. A received message is interpreted by either the `KQMLInterpreter` (a generic interpreter) or `AgentInterpreter` (if it contains a statement encoded with the "agent" ontology). An agent also knows its own address resource. Further resources are added later when the agent learns about them by communicating with other agents.

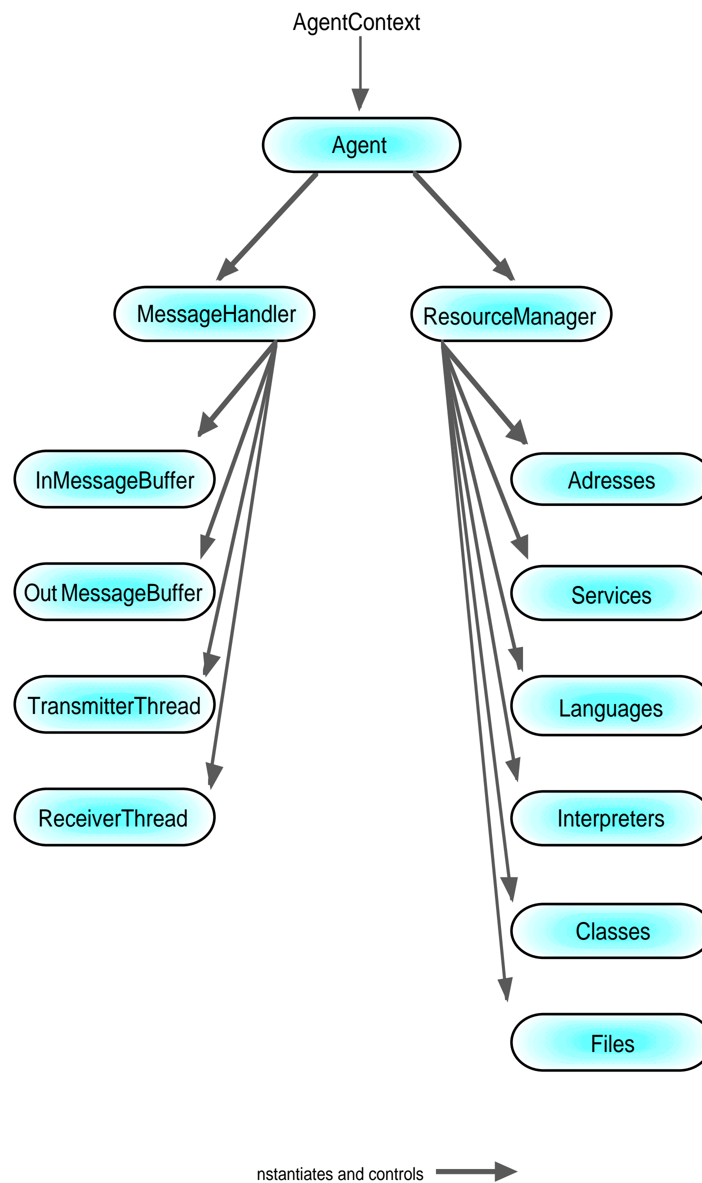

Figure 7.3 Agent Startup Class Instantiation

Agents communicate with each other by sending and receiving KQML messages. For this purpose, the agent instantiates a `MessageHandler`. The MessageHandler controls the complete communication process and invokes the necessary methods of the involved classes. Two buffer queues are used to serialize the incoming and outgoing messages (`InMessageBuffer` and `OutMessageBuffer`). A `ReceiverThread` pulls the incoming messages from the buffer so they are processed and the `TransmitterThread` takes care of messages waiting to be sent in the outgoing buffer.

7.2 JAT messaging

Communication is the main task of an agent and most actions are either triggered by an arriving message or result in a message being sent. This section explains how the agent works from the point of view of class interaction. It presents an overview of the main classes that are involved in the process of communication.

When the agent receives a message, it is accepted by the `CommInterface` and stored in a string. The CommInterface calls the agent's *receiveMessage* method and hands over the message string, which is passed on to the agent's `MessageHandler`. The MessageHandler stores it in the `InMessageBuffer` for processing. This completes the low-level part of message reception.

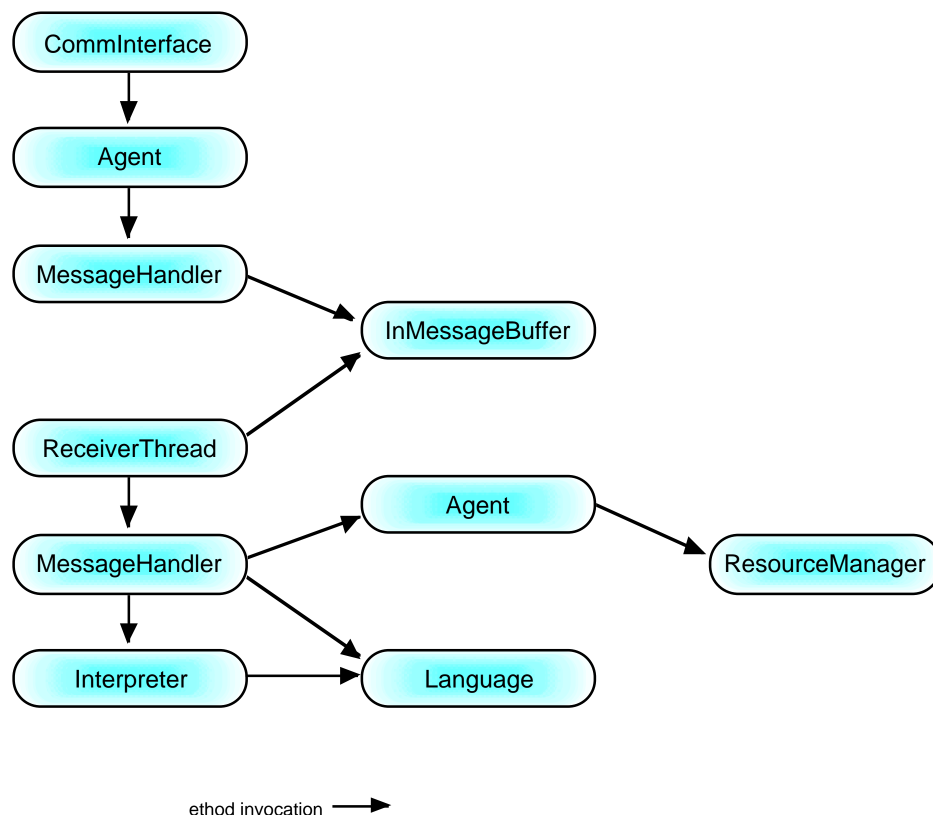

Figure 7.4 Class Invocation when Receiving a Message

The message string is pulled from the InMessageBuffer by the `ReceiverThread`, which passes it on by calling the MessageHandler's *interpretMessage* method. At this point the actual message interpretation begins. A new instance of a `Language` subclass is instantiated, which stores the message and parses the syntax. Since JAT agents communicate in KQML, the appropriate subclass of Language is `KQMLmessage`. The message reception differs from the originial JAT framework. Originally the message was stored in a KQMLmessage object directly after reception by the CommInterface. This step of the message pro-

cessing was moved to the MessageHandler to add the functionality of a syntax check and consequent error message.

If the syntax check returns an error, the MessageHandler sends an error message back to the message's origin. If the syntax of the message is correct, the MessageHandler checks whether the message has a content field and which language and ontology the content statement is encoded in. The `Interpreter` matching the specified ontology is requested from the `ResourceManager` (via the Agent).

The KQMLmessage is passed to the *interpretMessage* method of an instance of the Interpreter, which contains procedural instructions for handling a specific message. The message is interpreted according to the language specified for the content statement. If the content string is encoded in a language other than KQML, a corresponding subclass of Language is instantiated to contain the statement. Like the Interpreter classes, the different Language classes can be requested from the ResourceManager.

For an agent, the process of message interpretation means that some kind of action is triggered. Usually this will result in a change of the agent's internal state, a modification of its knowledge or a reply message being sent.

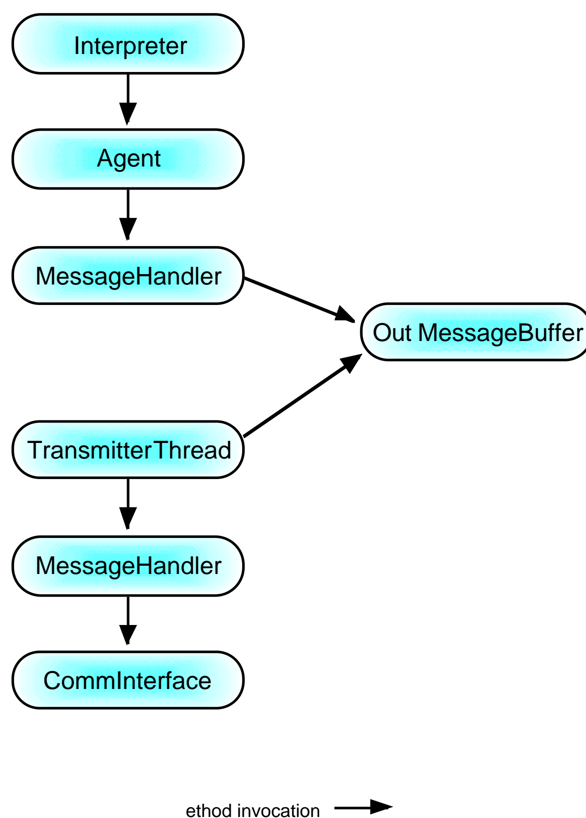

Figure 7.5 Class Invocation when Sending a Message

The reverse process of sending a message works in a similar fashion. The Interpreter or the Agent composes a new message by instantiating a KQMLmessage object. The message is handed over to the MessageHandler, that stores it in the `OutMessageBuffer`, where it waits to be sent.

The `TransmitterThread` pulls the waiting message from the buffer and passes it to the MessageHandler. Calling the CommInterface's *sendMessage* method, the message is sent to the receiving agent.

7.3 Package agent

The `Agent` class implements the core of a software agent. The ConceptServer, Concept-Client and ConceptSearch agents are all based on this class directly. The JAT framework also provides a basic ANS agent, which serves as central address exchange for a group of agents. The ConceptBroker is an extension of this ANS agent. The table below shows the class hierarchy of the agent classes.

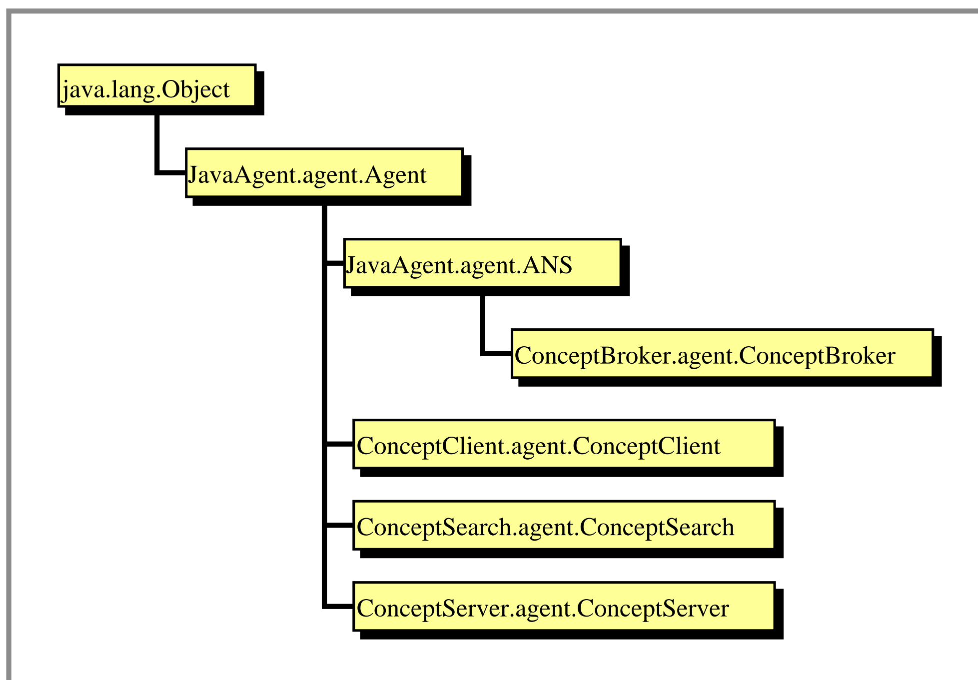

Figure 7.6 Agent Class Hierarchy

The `Agent` class provides the methods necessary to handle the exchange and management of resources for an agent itself. Every agent in the system requires and uses this basic functionality.

In addition, the `ANS` class manages the address resources for the system. It answers requests for an address, if the requested address is known (has been registered). If an agent unregisters its address again, the ANS sends a notification to all other agents, to inform

them that the address is no longer valid. The `ConceptBroker` extends this by managing the services offered by the agents in the system. Agents can subscribe for a service type, which will cause the ConceptBroker to add them to a list of subscribers. It will also send all the currently known services of the indicated type to the subscriber. The subscriber will get continuous updates whenever a new service becomes available or unavailable. If an agent unregisters its address resource, all the services it offers are removed as well.

The additional information managed by the `ConceptClient` consists of its user's identity and password, the connection status with the next ConceptBroker and ConceptServer (registered or unregistered), the current concept being displayed and whether a new concept has been requested.

The `ConceptSearch` agent needs to know its connection status with the next ConceptBroker and ConceptServer, to be able to search new links for the concept tree. It stores the concepts that are yet to be processed in a list and the concept it is currently working on. Furthermore it needs to be able to send a HTTP query to the Internet search engine AltaVista and extract the links from the answer. These documents are again retrieved with the HTTP protocol and evaluated versus the reference links in the current concept.

The `ConceptServer's` additional capabilities consist of managing the concept tree. For this purpose, it needs to store the concept data on the local filesystem and send out concept and link data on request. If another agent attempts to insert or delete a link, the validity of the query must be tested, to prevent unauthorized manipulation of the concept and link data.

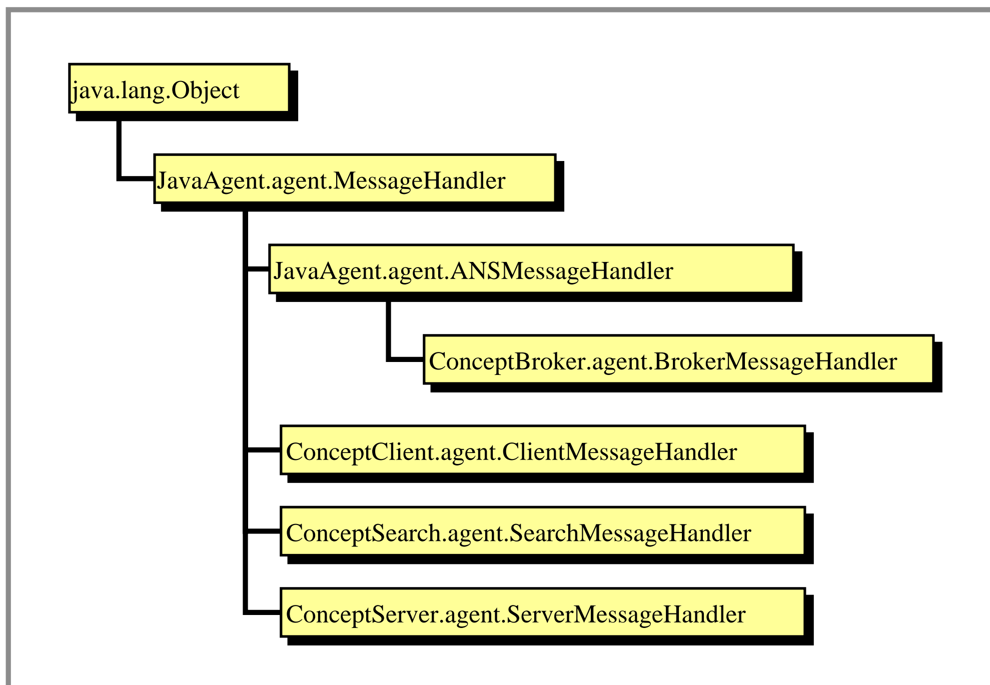

Figure 7.7 MessageHandler Class Hierarchy

Controlling the communication process is the task of the `MessageHandler` class. The subclasses extend this by providing additional methods to compose the specific messages they need to send. In particular, the `BrokerMessageHandler` provides methods to notify other agents about a change of available services. The `ClientMessageHandler` and `SearchMessageHandler` provide methods to exchange concepts and links from the ConceptServer.

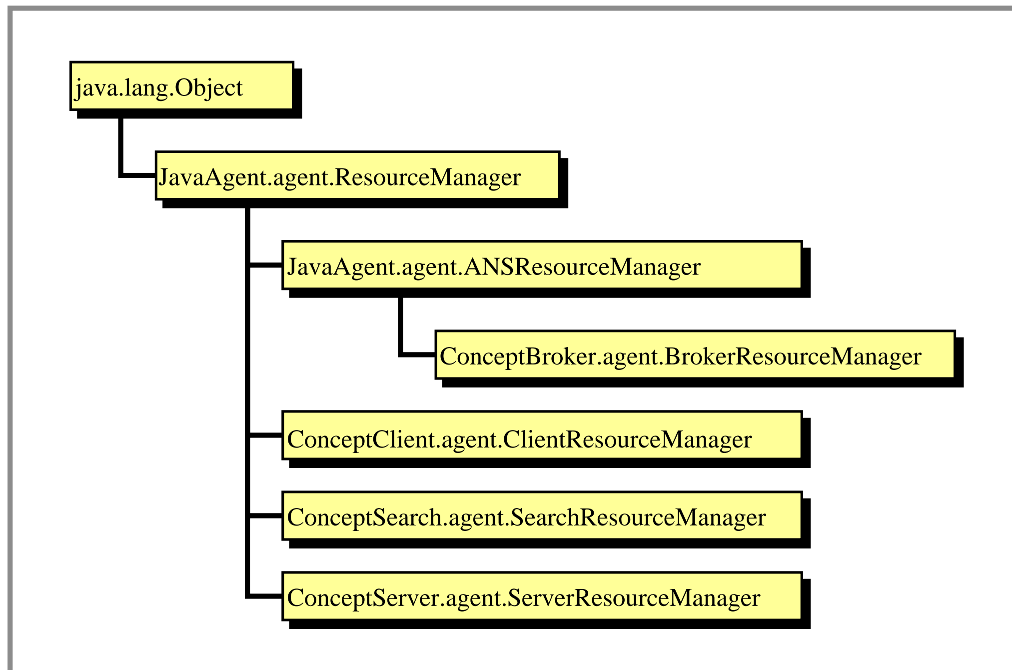

Figure 7.8 ResourceManager Class Hierarchy

An agent's resources are controlled by the `ResourceManager`. The specific subclasses differ only in the constructor method, where the agent's own resources are added to the empty resource lists. Each agent stores its own address and service resource, as well as the interpreters it requires. The generic KQMLInterpreter and AgentInterpreter (which interprets content statements of the "agent" ontology) are initialized by the superclass ResourceManager. All the agents add their own BrokerInterpreter (to interpret content statements of the "broker" ontology). In addition, agents dealing with concept knowledge add the ConceptInterpreter in their ResourceManager's constructor method (to interpret content statements of the "concept" ontology).

7.4 Package context

An agent is created by instantiating its corresponding context class. The context defines the environment for the agent and provides methods to interact with it. It reads command-line parameters, creates a `SocketInterface` for communication and an `AgentFrame` GUI if necessary. Finally it instantiates the Agent class itself.

Each agent requires its own AgentContext subclass, since each agent needs its own context to instantiate its corresponding Agent and AgentFrame classes. Especially the Concept-Client requires its own context, since its environment is encapsulated by the applet class, as opposed to the stand-alone application agents. However, the communication via sockets is the same for all the agents.

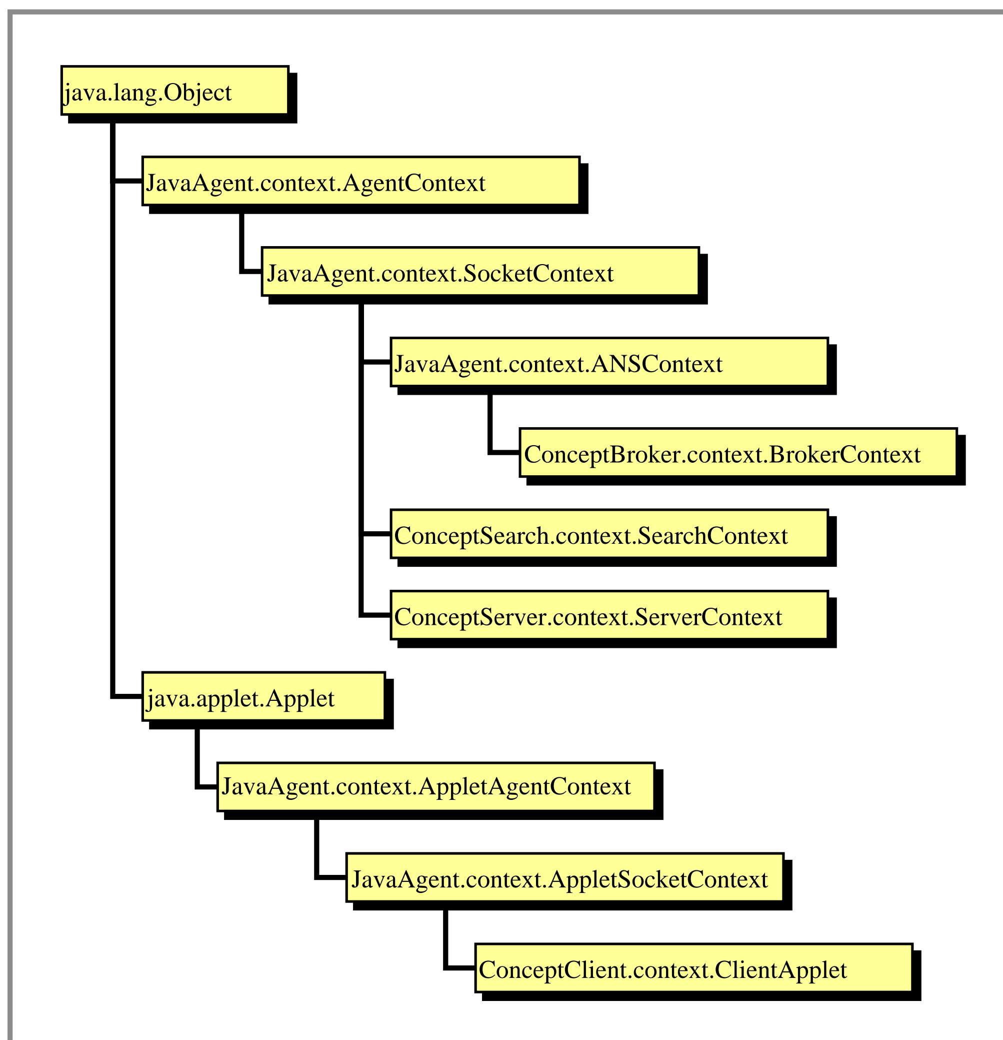

Figure 7.9 Context Class Hierarchy

The agent GUI is based on the Java `Frame` class. The basic JAT `AgentFrame` class provides the generic interface to control the agent and to show its knowledge, messages and

actions. Each of the agents shows different information in its main window (see screenshots in chapter 5). Therefore a specific subclass exists for each. An exception is the applet-based ConceptClient, its main interface is a subclass of `Panel`, since the Java Frame cannot be embedded into a browser window. For this reason the context classes provided by the JAT had to be completely rewritten, to adapt them to the applet setting.

Additional windows to display information are based on the `InterfacePanel` class. The ConceptServer uses two extra dialog windows to edit its concept tree and concept data.

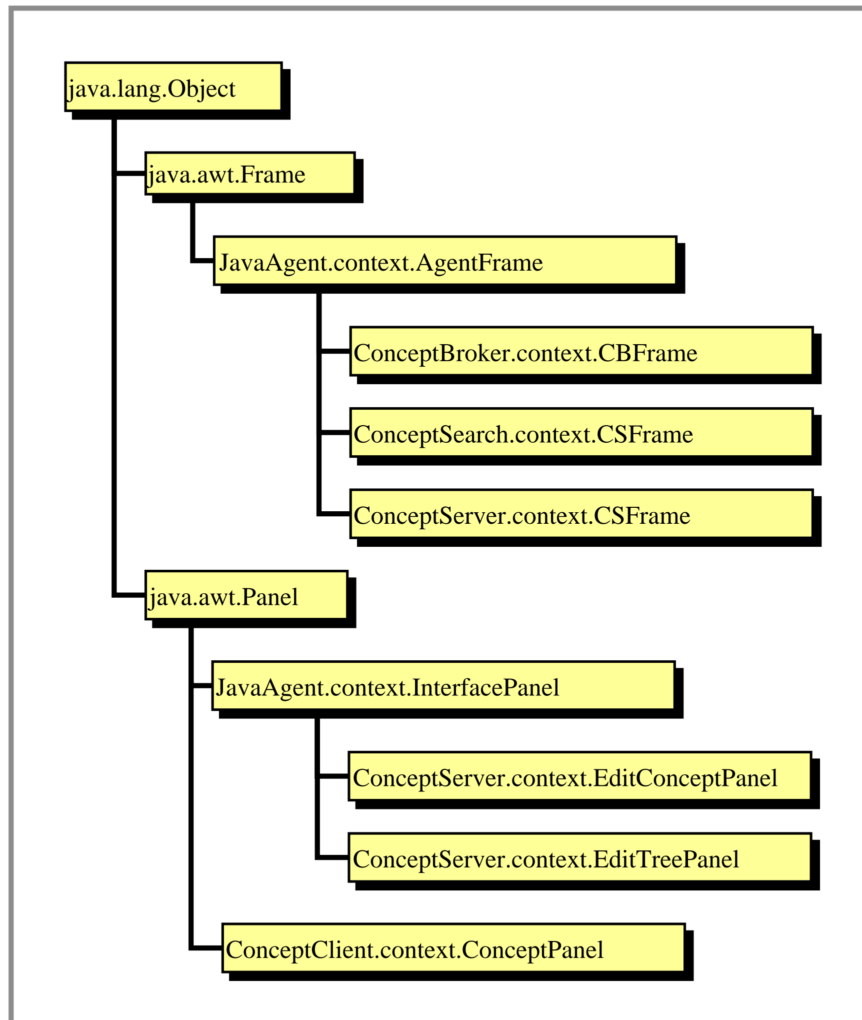

Figure 7.10 Interface Classes Hierarchy

7.5 Package resource

The resource package contains classes to manage agent knowledge, message parsing and interpretation. The resource objects managed by an agent may represent data and instructions (program code), stored and encoded in a Java class. For each resource type, the agent

has exactly one instance of a container class that manages all the resources of that type. The resource object itself is represented by a separate class that encapsulates the actual data. An agent may have multiple resources (instances) of the same type, all managed by one container for that type. The following table shows the six resource types and containers that manage them.

Type	**Container class**	**Resource class**
address	Addresses	AgentAddress
service	AvailableServices	AvailableService
interpreter	Interpreters	Interpreter
language	Languages	Language
class	Classes	Class
file	Files	FileLocation

Table 7.1 Resource classes

Resources in a container class can be added, deleted or queried for existence. Access to the resources needs to be synchronized by the container class, due to the multiple threads running within the agent. If the agent tries to use a resource which is not available, the container class can try to retrieve it by sending a message to another agent (e.g. if a requested `AgentAddress` resource is unknown, a message is sent to the ConceptBroker). This functionality is provided by the `RetrievalResource` class, which is the superclass for all the container classes. Technically, the automatic retrieval of resources which are unknown can be applied to all resource types, but in CEMAS it is only used for address and service resources. The following figure shows the class hierarchy of the container classes that manage resource storage and access.

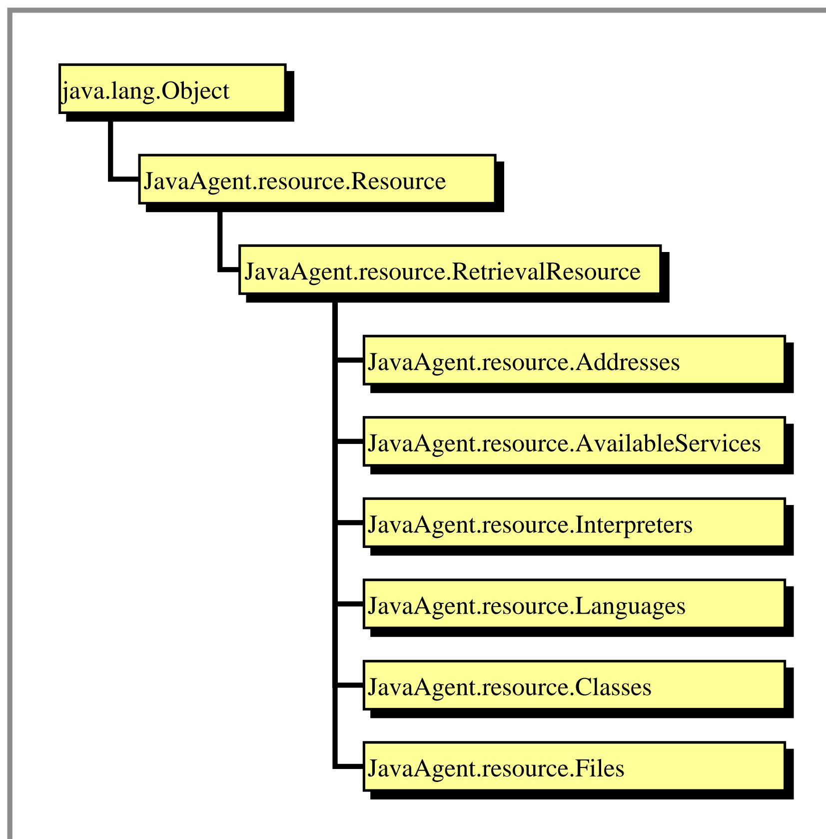

Figure 7.11 Resource Container Class Hierarchy

7.5.1 Resource types

Six basic types of resources are used equally by all agents of the system. Except for the type "service", which is an extension specific to CEMAS, they are provided by the JAT framework. In addition some Interpreters are agent-specific, described in more detail below. The following table shows the resource class hierarchy.

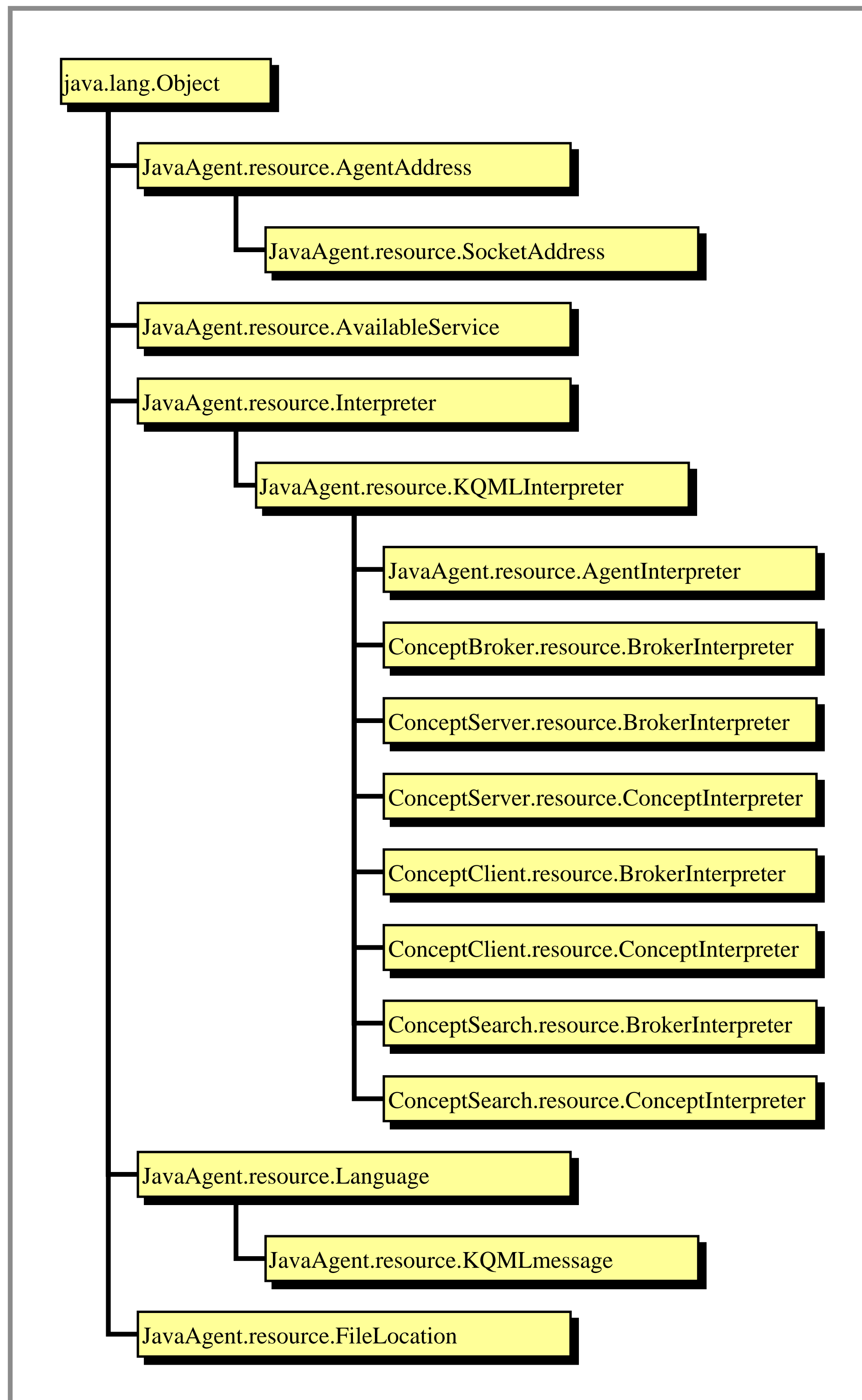

Figure 7.12 Resource Class Hierarchy

An agent's address is represented and stored in the `AgentAddress` and `SocketAddress` classes. The first is an abstract class to group different kinds of addresses for differ-

ent types of communication transport layers. JAT agents only communicate using TCP/IP sockets, so there is only one subclass SocketAddress, which stores the host and port number.

Knowledge about a service offered by an agent in the system is captured with the `AvailableService` class. It stores the name of the agent that offers the service, the service name, the service type and the contact information (administrator's email).

The abstract superclass `Language` handles a message in string format. Since CEMAS agents use only the KQML language, only the `KQMLmessage` subclass is required. Each message is stored in a separate instance of KQMLmessage. A KQMLmessage may be created by passing a string which contains the message as an argument for the constructor method (used for receiving messages). In that case the class parses the message syntax and provides methods to access the performative and parameters. Alternatively an empty message can be created and the performative and parameters can be added using the methods of the class. The composed message is then available in string format again (used for sending messages).

Essentially the handling of messages is implemented as provided by the JAT framework. Usually a KQML message has two parts: the outer message itself as well as an additional statement in the content field. The content statement may be encoded in languages other than KQML and may contain words from different ontologies. Some changes were made to the original JAT code to adapt the process of message interpretation to the KQML draft specification. The example agent included with the JAT framework used only one performative, evaluate, for all its messages. The actual message was encoded in the content statement (which was the object to be evaluated). The JAT agents did not make use of any other KQML performatives and shifted the real communication to the content layer, thus making the outer KQML layer obsolete. The JAT implementation of the message handling process reflected this by assuming that all messages were encoded this way (the performative always being "evaluate" and the real message being in the content statement). As a consequence, it was necessary to change the classes involved to enable the agents to handle other performatives and content statements as well.

To process a message, the MessageHandler passes it on to the appropriate `Interpreter` class. In other words, the Interpreter will initiate an action within the agent as a result of the message interpretation. Since CEMAS only uses KQML, all messages are per default passed to the `KQMLInterpreter`, which knows how to handle the performatives and parameters. If the message has a content statement, the Interpreter requires additional functionality, present in the appropriate subclasses.

To be able to deal with the content, an interpreter needs to handle both the language (syntax) and the ontology (semantics) used to encode the content statement. Thus it is language-specific. It can interpret a message statement in exactly one specific language and one specific ontology. Since CEMAS agents use three different ontologies for the content statements, an appropriate subclass exists for each. The `AgentInterpreter` handles content statements using the "agent" ontology. All agents use the same interpreter, since they all handle the knowledge expressed in the statement in the same way. The `BrokerInterpreter` is required by every agent as well, to deal with content statements using the ontology "broker". However each agent uses the knowledge about services in a different way (e.g. the ConceptBroker manages it, the ConceptClient needs to find the next ConceptServer, ...). The actions being initiated by the interpreters are different and thus each agent has its own BrokerInterpreter. The same is true for the `ConceptInterpreter` class, which deals with content statements using the "concept" ontology. Since each agent performs a different action upon receiving such a message, a separate interpreter class is required.

7.5.2 ConceptFile class

System-wide the concept tree data is stored by the ConceptServer agent in plain text files on its local file system. The ConceptServer stores references to the concept object and the corresponding file pathname in the `ConceptFile` class. This class also contains the methods for reading and writing the concept data.

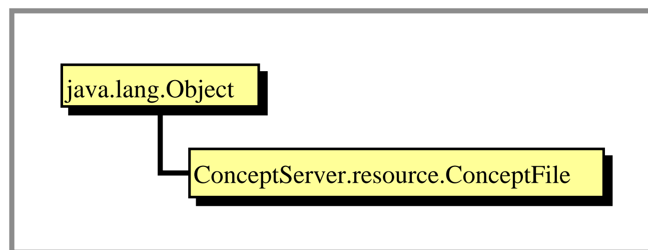

Figure 7.13 ConceptFile Class Hierarchy

Files are stored in the "working" directory, one of the agent parameters that is set at startup. The concept tree is rooted in a directory named "root", an abstract concept that has no content except for all the first-level concepts. The full name of a concept is the pathname of the directory, relative to the working directory where all concept information is stored. Consequently the names of subdirectories in a directory represent the subconcepts. The concept node data is stored in a plain text file with the name "concept.data". It contains all the information related to that concept, including all the links. The human-readable text file format

allows for easy debugging and changing of data. The structure of the "concept.data" file is listed in appendix D.

A concept in the tree is thus identified by its full name, taking one of the examples from a previous chapter that would be "root/Science/Energy/Renewable/Solar". This name is used as a pathname for the filesystem and appended to the "working" directory. The data for the above concept would be stored in a file named "$HOME/java/working/root/Science/Energy/Renewable/Solar/concept.data". Each subdirectory in this directory would contain a subconcept.

Concept data is loaded into memory from file as needed, to minimize memory requirements. The ConceptServer agent tags the concept when it is changed (a link is added or deleted), and writes the changes back to the file before freeing the memory space.

The fact that the concept tree is mapped to the underlying filesystem limits the possible namelength of a concept. The data storage is indeed an inadequate solution for a heavy workload, see the discussion chapter for possible improvements.

7.6 ConceptUtil package

This package contains utility classes used by several agents. The `Concept` class stores all the data that represents a concept node. Similarly, the `Link` class stores the data that is contained in a link. Both provide the necessary methods to access and modify this data.

The DocQuery utility class is used for retrieval of HTML documents from the WWW. It is a subclass of thread, since the retrieval task runs as a parallel action which times out after a certain time. This prevents the agent from blocking unnecessarily if the network connection is slow.

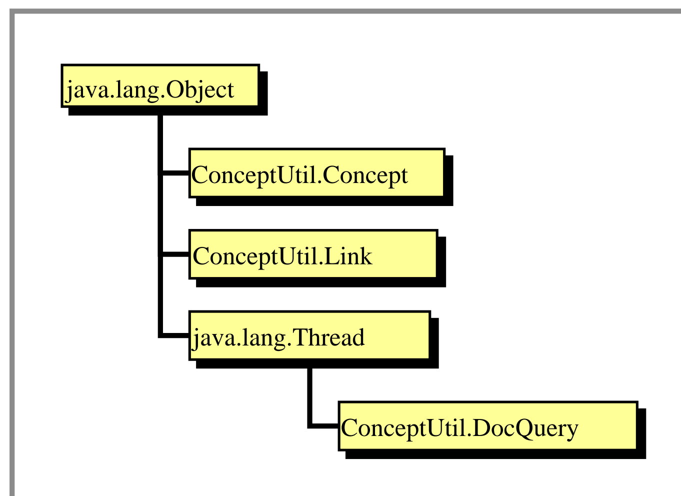

Figure 7.14 Class Hierarchy of the ConceptUtil Package

8 Discussion

When developing a system like CEMAS, it is impossible to yield a “perfect” solution right away. For once there is a lack of time to implement all the ideas and details that come up during the development process. In addition, some issues only become clear in retrospective, after the implementation is finished. As noted by Wooldridge and Jennings [Woold98], the field of agent-oriented development has some very own problems. This chapter will list some of the known limitations of the approach discussed in this paper. At the same time, some alternatives and ideas are introduced, which were a result of the ongoing interaction with the subject matter. Possible future improvements are mentioned as well, to show where the multi-agent system functionality could be extended. Some of these are already realized or prepared for in the current implementation, others are only ideas for future improvements.

8.1 The agent paradigm

Why is this software called a multi-agent system and not a collection of client server programs? Several papers give a good overview about what an agent is or should be (see [Finin97], [Frank96], [Genes94], [Petrie96], [Woold94]). While the conclusion seems to be that the agent paradigm is far from being a definition of what an agent is, it definitely serves as means to better understand what we want from such a software program. Even if a blueprint of how to build some of that functionality does not yet exist, the property description does show the direction of possible or necessary development, in order to make it more “agent-like”.

Some of these descriptions are on a rather abstract level. Even though CEMAS (and other JAT-based agent implementations[17]) currently is a pragmatic realization, the system possesses several of the agent properties mentioned in chapter three. The agents are placed in an environment that is uncertain (the Internet), they feature some kind of intelligence (methods to handle domain knowledge), are autonomous and communicate. Of course they are neither truly autonomous nor intelligent, because they contain much implicit knowledge about their functionality and their application domain. Therefore the term “agent” remains mostly a matter of the beholder’s definition.

As already noted, the speech-act view of an agent states that both speech and act are strongly interconnected. A software agent can interact with its environment through some kind of communication and internally with its knowledge base (or any other form of infor-

[17]See <http://www.informatik.uni-ulm.de/abt/ki/Students/mb/jat/> for a list of agent systems based on the JAT.

mation representation). Therefore, in a multi-agent world an agent's external communication capabilities should match its internal action capabilities.

In other words, it does not make much sense to have a very intelligent agent (e.g. that has a large knowledge base and is capable of performing high-level reasoning upon it), if it has only limited communication capabilities, because it will be unable to communicate the results of its internal processes, it will be isolated. For the same reason it does not make sense to have an agent that is capable of communicating anything with a very flexible communication language, if it is unable to make any use of messages or statements it receives.

In our case, the language used (KQML) is very powerful and flexible. There is no limitation for the enclosed statements that could possibly be exchanged, except that the agent does not necessarily know how to handle such a statement and make any use of it. The domain knowledge (interpreter) necessary to make use of a message may not be available. A flexible communication language like KQML offers more possibilities than an agent is able to use. For this reason the current CEMAS implementation uses only a subset of the KQML performatives and very small ontologies. But for most of the improvements suggested in this chapter, the current messages used are completely sufficient, they already provide the means to exchange the necessary knowledge between the agents.

8.2 The concept model

The organization of concepts in a predefined tree may be limiting for the user, because he cannot organize the concepts the way they are stored in his mind and he has to adapt to a given structure. For a user who uses the system mainly to store and organize his own links, this may seem as a drawback. On the other hand, for a group of users that want to exchange links, it is necessary to agree upon a common concept structure. In addition the tree can be a navigation aid for a user who does not exactly know what he is looking for or where to find it.

A predefined concept tree administered by a single expert is also necessary because of the system's distributed architecture. If concepts could be dynamically created by each agent, the concept graphs would grow uncontrollably and overlap partially (depending on what the user had in mind when creating them). This would cause problems when concepts of different graphs have to be matched or exchanged, as their definitions would be vague or too stretched.

Some kind of documents available on the WWW are difficult to classify into a single exact category or concept. For such cases, the approach must be able to handle conceptually promiscuous documents while circumventing the problem of having to define a complete well-defined concept world model for all the available information (see [Nishi95]). The approach of a concept as container for a set of example documents is a powerful but very

flexible way to describe a topic. The definition of such a concept can be derived from the set of example documents which are available on the WWW. This approach has already been successfully applied in other projects (see [Cohen96a] for a machine learning algorithm applied to WWW-based documents). A usable definition can be derived even when only few example documents are given (see [Emde94]).

The multi-agent approach supports a full-text document classification, since the search process runs as a continuous background task. The searching agent is autonomous and may take as much time as it needs to classify documents into an existing concept tree and does not need to reply as direct response to a query. This is necessary, because current algorithms used to compare full-text documents are computationally complex. In most cases such a classification will take longer than the amount of time a user is willing to wait for the response to a query. This makes it difficult to implement such a comparison in a classic client-server setting, like a single query to a WWW search engine, where an immediate reply is expected. Other single-agent implementations that combine or merge the query results from several search engines face the same problem, since they have to deliver an answer in real-time (while the user waits). Thus the time they may take to classify search results is rather limited (see [Fu96] for an agent implementation that uses unguided machine learning techniques to classify query results from search engines).

Currently it is not possible to express a desired level of similarity between a document and a concept. The classification is boolean, either a document matches or it does not. At the same time, neither the classification method nor the threshold level for a positive match is exactly defined. Thus, a match of a document with a concept is interpreted as “sufficiently close”, which always means “as close as possible”. As advantage of the current approach, concepts and links can be exchanged independently of the classification algorithm, thus allowing the use of multiple different algorithms in the system. To be able to express the level of similarity between a document and a concept, it would be necessary to extend the current ontologies used for messaging. In addition, each agent would be required to possess the domain knowledge of how the classification algorithm works. As a result it would be more difficult to add new agents or algorithms to the system.

From some of the examples it became evident that users find it difficult to construct a hierarchical tree to organize the information domain they want to capture, even if they know it well. It seems that humans deal with categories less in a hierarchical manner and more in a loosely chaotic structure (see [Lakof85]). The hierarchical tree structure is still useful, since it serves as a navigation aid for the user. Setting up such a concept tree is evidently simpler for distinct and specialized smaller areas of information as opposed to a larger but broader topic area.

8.3 Implementation

Currently, CEMAS is just a first basic implementation to show that the system works, to test and possibly develop the approach further. The individual agents lack some functionality that would be useful in a larger system or would make them behave more intelligently and autonomously. Some possible enhancements are introduced in the following section.

8.3.1 Communication

The communication protocol does not contain a performative to ask for a search of the whole concept tree (including all links) for a keyword. This would be a very useful extension to manual navigation of the concept tree.

An agent that is willing to give a proper response to a collection of messages currently indicates this by registering a service with the ConceptBroker. All agents in the system implicitly know which messages are associated with each service type. In contrast to this, the KQML specification suggests a more universal approach, where each message should be registered separately. More specifically an agent should announce that it is willing to give a valid answer to a single specific message, for each message it understands. While this is more flexible than registering a set of messages (grouped in a service), it certainly creates more communication overhead to register each single message. In CEMAS, it does not make sense to support only a part of the message functionality grouped together in a service (e.g. only the insert performative, but not the delete performative). The same is true for the implicit knowledge of how to handle a performative. For Example an agent needs to know how to perform both the “insert” and “delete” act upon its internal virtual KB. So in addition to being able to understand the message the agent must be able to carry out the corresponding action.

8.3.2 ConceptBroker

The current distributed architecture requires exactly one ConceptBroker agent as a mediator. All the agents in the system need this agent to acquire critical information. This results in the ConceptBroker being a single point of failure. Without it, the system can not work. A possible improvement would be to have multiple ConceptBroker agents, that exchange their knowledge with each other, so that if one of them crashes or hangs, the system continues to work.

8.3.3 ConceptServer

In a larger network with several ConceptServer agents, these could exchange their concept trees with each other. Messages and concept data would need to be forwarded or cached. The current message set already provides the necessary means to have multiple ConceptServers.

The internal storage of the concept data in simple text files should be modified to improve performance. A solution would be to use a database engine to store and retrieve the concept data[18]. Such a database engine also provides methods to implement a tree-wide search for a keyword in a simple fashion.

The ConceptServer agent could provide additional help for the administrating user to manage the concept tree. If the number of links in a concept node reaches a certain limit, the agent could cluster them into several distinguished subconcepts. The agent could generate relations between existing concepts that are close in content. Keywords for existing concepts could be optimized for IR purposes (in terms of discrimination and representation) from the existing links in the tree. All this could be done with different levels of autonomy, either as a suggestion to the agent's administrator (the domain expert that created the concept tree) or completely automatic, if the techniques are reliable . A continuous periodic verification of all the links in the concept tree would avoid dead links (whose documents no longer exist); they could be removed automatically.

8.3.4 ConceptClient

The reason that the user agent is rather limited is mostly due to the fact that it runs as an applet inside the user's browser. An applet can not save data to the local filesystem nor open connections to any host on the Internet, other than the host it was loaded from. Due to these restrictions both the ConceptServer and the ConceptBroker agent need to be executed on the same host that offers the ConceptClient applet. The first restriction is already solved by saving necessary data at the ConceptServer (as it is currently implemented with the concept data). The restricted communication ability can be solved by turning the ConceptBroker agent into a router[19] to pass on messages from the ConceptClient to other hosts on the Internet.

Since the applet runs only as long as the surrounding browser permits, it is not really suited to carry out long-term continuous actions. If the links were only stored locally by an agent, it would have to run at all times to be able to share them with other agents. While running, the agent would be using up the computer's memory and resources even if it is not actively being used (most users are reluctant to allow this). Consequently the user's ConceptClient agent is executed only as a temporary task and the concept data has to be stored in a central location (the ConceptServer) to be permanently available for all users.

A possible alternative would be to have a stand-alone Java program instead of an applet, which has some drawbacks. Communication between a stand-alone program and the browser is platform-dependent and therefore difficult to implement. The Java applet class has the advantage of providing some methods to interact with the browser (e.g. instructing

[18]This can be easily implemented by using the JDBC package (Java DataBase Connectivity).

[19]The routing functionality is a built-in feature of the next version of the JAT, still being developed. See the JATLite homepage at http://java.stanford.edu/.

the browser to fetch and display a certain document from the WWW). Another downside from the user's point of view would be that he has to install the stand-alone program first, as opposed to simply pointing his browser to a WWW page that includes the applet. A Java stand-alone program would require at least the installation of a runtime environment to be executed in, the common browsers already include such a functionality for applets. From the point of user-friendliness, the applet agent is clearly the better solution.

If seen as a representative of the user in the multi-agent system, the fact that the agent is only available part-time is a drawback. For example it is currently impossible for a Concept-Search agent to pro-actively suggest a link to a user's ConceptClient agent, to specifically alert the user's attention to that link. Even though the communication protocol provides the necessary message (tell), the ConceptClient currently ignores unrequested information, since it does not expect any and the agent's interface lacks the means to display it appropriately. In addition, such a message can not be sent to the ConceptClient while it is offline, since the message is not routed or cached by another agent (this could be done with an extended broker, as mentioned above).

The agent could track which links are selected most by users and allow the user to rate the links of a concept. This rating information could be passed back to the other agents as feedback about the quality of a link. The ConceptServer could use that to provide a ranking of the links in each concept (to rank the better links higher on the list) and the Concept-Search agent could use it as feedback for the accuracy of the classification, which can be employed in machine learning algorithms (like the ones used in [Balab95] and [Cohen96b]).

The interface currently lacks a possibility to search the complete concept tree with keywords.

8.3.5 ConceptSearch

The current CEMAS architecture assumes that the ConceptSearch agent works like a background process. It continuously searches for new links, periodically cycling through all the concepts in the ConceptServer's tree. New links can be added to the existing concepts in the tree, thus the existing predefined tree supports this background approach. That way the agent has the autonomy to take as much time as needed for a qualified document classification (based on full-text analysis). Since the idea was to offer information for a user with a long-term interest, the ConceptSearch agent does not offer the functionality to reply to a single one-time search query.

Based on this setting, it is possible to implement other types of searching agents. As shown by the work of P. Fernandez Presa [Ferna97], this agent can be extended by equipping it with additional functionality. Currently, it starts off with a list of possible links

acquired from a search engine, and then uses intelligent text analysis and classification techniques to filter out the documents that fit into a certain concept.

Different methods and approaches could be used to search the WWW and classify newly found documents. Instead of querying a search engine for potentially matching documents, the agent could start with a given HTML document and then follow the links contained therein (browse from document to document). Other methods that have been used in agent systems to compare and classify documents are natural language processing (see [Kesel97] and [Wonde96] for agent systems that use NLP), machine learning techniques using vector space representation and n-grams (see [Fu96], [Boone98] and [Cohen96b]) and neural networks (see [Chen96]). Note that NLP techniques are usually limited to a single language (english in most cases), which excludes documents in other languages (or even a mixed content of multiple languages).

A more pro-active ConceptSearch agent could extract a user's personal preferences from the concept tree and try to suggest specific links that it considers of high interest. This is essentially possible because the user's identity is stored with the links he has added, thus the agent could calculate a user interest profile from all the links added by a single user. In addition, the ConceptBroker already provides the possibility to subscribe for the "ConceptClient" service. In other words, it is possible for the ConceptSearch agent to be notified about ConceptClient agents going on- or off-line. Thus, the information about new links of high interest could automatically be sent to the user's ConceptClient as soon as it connects to the system the next time. Even if there is currently no way to notify the user via his agent (because the ConceptClient does not support it), an email message could be sent instead, since the user's email address is already known.

8.4 Conclusion

This paper describes CEMAS, an implementation of a multi-agent system applied to the problem of information retrieval on the WorldWideWeb. It illustrates the various questions that arise for such an implementation and presents a working solution.

The distributed multi-agent design provides flexibility and scalability, since each user and main task is represented by its own agent. Thus, single agents can be changed or added without the necessity to change the whole system. Knowledge about information on the WWW is described in a concept model which is flexible but more powerful than simple keywords. The multi-agent system allows to share and exchange this knowledge between users and agents, which is thus combined and re-used. Techniques that take a lot of computation time can be employed to classify documents, because the searching agents run continuously in the background.

9 Bibliography

[Austi62] J. Austin, “How to do Things with Words”. Oxford University Press, Oxford, 1962.

[Balab95] M. Balabanovic and Y. Shoham, “Learning Information Retrieval Agents: Experiments with Automated Web Browsing”. Proceedings of the 1995 AAAI Spring Symposium on Information Gathering from Heterogeneous, Distributed Environments, Stanford, Mar. 1995. <http://robotics.stanford.edu/people/marko/papers/lira.ps>

[Beigb97] M. Beigbeder, B. Doan, J. Girardot, P. Jaillon and C. Sayettat, “Information Retrieval in the WWW”. 1997. <http://www.info.unicaen.fr/~serge/3wia/workshop/.papers/paper17.html>

[Boute96] T. Boutell, “World Wide Web FAQ”. 1996. <http://www.boutell.com/faq/>

[Boone98] G. Boone, “Concept Features in Re:Agent, an Intelligent Email Agent”. To be presented at the second international Conference on Autonomous Agents, Minneapolis, 1998. <http://www.cc.gatech.edu/grads/b/Gary.N.Boone/papers/reagent.ps>

[Catar97] T. Catarci, L. Iocchi, D. Nardi and G. Santucci, “Conceptual Views over the Web”. In Proceedings of the 4th KRDB Workshop, 1997. <http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-8/paper-3.ps>

[Chen96] H. Chen, C. Schuffels, R. Orwig, “Internet Categorization and Search: A Self-Organizing Approach”. Journal of Visual Communication and Image Representation, Special Issue on Digital Libraries, Vol 7, No. 1, pp. 88-102, 1996. <http://ai.bpa.arizona.edu/papers/som95/som95.html>

[Cohen96a] W. Cohen and Y. Singer, “Learning to Query the Web”. To appear in The 1996 AAA-Workshop on Internet-Based Information Systems. <http://www.research.att.com/~wcohen/postscript/aaai-ws-96.ps>

[Cohen96b] W. Cohen and Y. Singer, “Context-sensitive learning methods for text categorization”. Proceedings SIGIR, 1996. <http://www.research.att.com/~wcohen/postscript/sigir-96.ps>

[DARPA93] DARPA Knowledge Sharing Initiative External Interfaces Working Group, “Specification of the KQML Agent-Communication Language (draft)”. 1993. <http://www-ksl.stanford.edu/knowledge-sharing/papers/kqml-spec.ps>

[Edwar96] P. Edwards, D. Bayer, C. Green, and T. Payne, “Experience with Learning Agents which Manage Internet-Based Information”. AAAI Spring Symposium on Machine Learning in Information Access, 1996. <http://www.parc.xerox.com/istl/projects/mlia/papers/edwards.ps>

[Emde94] W. Emde, “Inductive Learning of Characteristic Concept Descriptions from Small Sets of Classified Examples”. In Proceedings of the 7th European Conference on Machine Learning, Springer-Verlag, 1994.

[Ferna97] P. Fernandez Presa, “Support for Information Retrieval on the Internet”, Internship report, Abt. KI, University of Ulm, 1997. <http://www.informatik.uni-ulm.de/abt/ki/Students/mb/cemas/javasmart.ps.gz>

[Finin97] T. Finin, C. Nicholas and J. Mayfield, “Agent-based Information Retrieval”. A tutorial given at SIGIR'97, Jul. 1997. <http://www.cs.umbc.edu/abir/sigir97/>

[Frank96] S. Franklin and A. Graesser, “Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents”. In Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996. <http://www.msci.memphis.edu/~franklin/AgentProg.html>

[Frost96] R. Frost, “Java Agent Template v0.3”. Program documentation, 1996. <http://cdr.stanford.edu/ABE/documentation/index.html>

[Fu96] X. Fu, J. Hammond, R. Burke, “ECHO: An Information Gathering Agent”. University of Chicago, Technical Report TR-96-17, Sep. 1996. <http://www.cs.uchicago.edu/publications/tech-reports/TR-96-17.ps>

[Gener96] General Magic, Inc. “Telescript Technology:Mobile Agents”. White paper, 1996.

[Genes92] M. Genesereth and R. Fikes, “Knowledge Interchange Format, Version 3.0 Reference Manual”. Technical Report Logic-92-1, Computer Science Department, Stanford University, 1992.

[Genes94] M. Genesereth and S. Ketchpel, “Software Agents”. Communications of the ACM, July 1994, Vol. 34, pages 48-53.

[Grube93] T. Gruber, “Toward Principles for the Design of Ontologies Used for Knowledge Sharing”. Technical Report KSL93-04, Knowledge Systems Laboratory, Stanford Univarity, 1993.

[Imam96] I. Imam and Y. Kodratoff, “Intelligent Adaptive Agents: A Highlight on the Field and a Report on the AAAI-96 Workshop”.

[Iwazu95] M. Iwazume, H. Takeda and T. Nishida, “Ontology-based Approach to Information Gathering and Text Categorization”. In Proceedings of International Symposium on Digital Libraries, pages 186-193, 1995. <http://ai-www.aist-nara.ac.jp/doc/papers/mitiak-i/ps/isdl95.ps>

[JAT] The Java Agent Template homepage. <http://java.stanford.edu/>

[Kesel97] V. Keselj, “Multi-Agent Systems for Internet Information Retrieval using Natural Language Processing”. Thesis, University of Waterloo, Ontario, Canada, 1997.

[Knobl94] C. Knoblock, Y. Arens and C. Hsu, “Cooperating Agents for Information Retrieval”. Proceedings of the Second International Conference on Cooperative information Systems, University of Toronto Press, Toronto, Ontario, Canada, 1994.

[Labro96] Y. Labrou, “Proposed KQML Specification Document”. In Semantics for an Agent Communication Language (dissertation), 1996. <http://www.cs.umbc.edu/jklabrou/publications.html>

[Lakof85] G. Lakoff, “Women, Fire and Dangerous Things. What Categories Reveal about the Mind.”. University of Chicago Press, 1985.

[Maes94] P. Maes, “Agents that reduce work and information overload”. Communications of the ACM, 37, 7, 1994. <http://pattie.www.media.mit.edu/people/pattie/CACM-94/CACM-94.p1.html>

[Nwana96] H. Nwana, “Software Agents: An Overview”. Knowledge Engineering Review, Vol 11, No 3, pp.1-40, 1996.

[Oukse97] A. Ouksel, “Ontologies are not the Panacea in Data Integration: a Flexible Coordinator to Mediate Context Construction”. 1997.

[Petri96] C. Petrie, “Agent-Based Engineering, the Web, and Intelligence”. IEEE Expert, Vol.11 No.6, 1996, pp 24-29. <http://cdr.stanford.edu/NextLink/Expert.html>

[Petri97] C. Petrie, “What’s an Agent...and what’s so intelligent about it?”. IEEE Internet Computing. 1997. <http://computer.org/internet/>

[Ragge95] D. Ragget, “HyperText Markup Language Specification Version 3.0”. WorldWideWeb Consortium, 1995. <http://www.w3.org/pub/WWW/MarkUp/html3/CoverPage.html>

[Rijsb79] C. J. van Rijsbergen, “Information Retrieval”, 2nd. Edition, Butterworth, London 1979. <http://www.dcs.glasgow.ac.uk/Keith/Preface.html>

[Shoha93] Y. Shoham, “Agent-oriented programming”. Artificial Intelligence, 60(1), pp. 51-92, 1993.

[Singh97] M. Singh and M. Huhns, “Internet-Based Agents: Applications and Infrastructure”. IEEE Internet Computing, 1997. <http://computer.org/internet/>

[Sun97] Sun Microsystems, Inc. Java documentation and tutorial. 1997. <http://java.sun.com/>

[Sycar95] K. Sycara and D. Zeng, “Task-based Multi-agent Coordination for Information Gathering”. 1995. <http://www.isi.edu/sims/knoblock/sss95/zeng.ps>

[Taked95] H. Takeda, K. Iino and T. Nishida, "Agent Organization and Communication with Multiple Ontologies". International Journal of Cooperative Information Systems, pages 321-337, Dec. 1995. <http://ai-www.aist-nara.ac.jp/doc/papers/takeda/ps/ijicis.ps>

[Woelk95] D. Woelk and C. Tomlinson, "InfoSleuth: Networked Exploitation of Information Using Semantic Agents". COMPCON Conference, Mar. 1995. <http://www.mcc.com/projects/infosleuth/papers/compcon-95.ps>

[Wonde96] B. Wondergem, P. van Bommel, T. Huibers, T. van der Weide, "Towards an Agent Based Retrieval Engine (Profile - Information Filtering Project)". Technical Report. Accepted at the 19th Annual Colloquium on IR Research of the BCS, Nov. 1996. <http//:www.cs.kun.nl/~bernd/abstract_ppp.html>

[Woold94] M. Wooldridge and N. Jennings, "Intelligent Agents: Theory and Practice". Knowledge Engineering Review, Oct. 1994. <http://www.doc.mmu.ac.uk/STAFF/mike/ker95/ker95-html.html>

[Woold98] M. Wooldridge and N. Jennings, "Pitfalls of Agent-Oriented Development", To be presented at the second international Conference on Autonomous Agents, Minneapolis, 1998.

[Nishi95] T. Nishida, K. Koujitani and H. Takeda, "A Plain Indexing Method for Organizing Conceptually Promiscuous Data". AAAI Fall Symposium Series -- AI Applications in Knowledge Navigation and Retrieval, 1995. <http://ai-www.aist-nara.ac.jp/doc/papers/nishida/ps/fss95.ps>

Appendix A

A.1 Classic search engines on the WWW

There are a number of search engines available on the WWW. They are sometimes also referred as indexers, since they index URLs of available HTML pages by automatically extracting keywords from them. The main problem in this process is the mechanism to automatically extract the relevant keywords from a document.

AltaVista: http://altavista.digital.com
Lycos: http://www.lycos.com
Excite:http://www.excite.com
HotBot: http://www.hotbot.com

Yahoo differs as it is not really a search engine. It is rather an extensive list of URLs grouped into several topic areas, which are organized in a hierarchic tree. The categorization is only very broad and not very deeply specialized.

Yahoo: http://www.yahoo.com

Appendix B

KQML string syntax in BNF.

```
<performative> ::= (<word> {<whitespace> :<word> <whitespace>
    <expression>}*)

<expression> ::= <word> | <quotation> | <string> |
    (<word> {<whitespace> <expression>}*)

<word> ::= <character><character>*

<character> ::= <alphabetic> | <numeric> | <special>

<special> ::= < | > | = | + | - | * | / | & | ^ | ~ | _ |
    @ | $ | % | : | . | ! | ?

<quotation> ::= '<expression> | `<comma-expression>

<comma-expression> ::= <word> | <quotation> | <string> |
    ,<comma-expression>(<word> {<whitespace> <comma-expression>}*)

<string> ::= "<stringchar>*" | #<digit><digit>*"<ascii>*

<stringchar> ::= \<ascii> | <ascii>-\-<double-quote>
```

Appendix C

C.1 CEMAS KMQL messages

For a general description of KQML, please consult the KQML specification.

CEMAS supports the following message types, which are listed in terms of which ontology they belong to. This also shows which Interpreter will understand them. They are listed as the main performative followed by the value of the content field. The value of the language parameter is always “KQML” and the value of ontology is either “agent”, “broker” or “concept”, depending on the message. Each message must have all the fields specified for it. If a certain field happens to contain no value (e.g. description), the message must still contain that field with an empty string (two parentheses ""), indicating that there is no value. If some expression is in double quotes, this means it is a string that may contain several words separated by whitespace, otherwise it is considered to be a single word.

The agents expect all fields to have a valid value, if any fields or values are missing, the agents will refuse to process the message and send an error message back.

C.2 Messages without Ontology or Content:

- **sorry** :comment "<comment string>"
  This message is used to inform an agent that his message was understood but no further response can be given.
- **error** :comment "<comment string>"
  This message is used to inform an agent that his message was not understood because it was malformed or invalid.

C.3 Messages with Ontology and Content field

The following messages should be in the form:

```
(performative :sender <agent name> :receiver <agent name> :language
KQML :ontology <content ontology> :content (content field value))
```

Below only the main KQML performative and the value of the content field will be listed.

C.3.1 Ontology: agent

- **ask-one** (resource :type <resource type> :name <resource name>)
  This message type is used to ask about the value of a resource.
- **tell** (resource :type <resource type> :name <resource name> :value <resource value>)
  This message type is used to inform an Agent about a resource of a specific type, name and value. Where type may be one of “address”, “class”, “interpreter”, “language”, “file”, “local-file” or “service”. Value is interpreted uniquely for each type. (For address value is “host:port”; for class, Interpreter and language value is the “code_base class_name”; for file the value is “base_url file_name”; for

service it is one of “ConceptServer”, “ConceptClient”, “ConceptSearch”) If a local version of the file exists, it is not replaced. If <resource value> is “?”, then the sender does not know the value or location of the resource.

- **unregister** (resource :type <resource type> :name <resource name>)
  Sent by one Agent to invalidate an offered resource. The receiving agent should remove the resource information from its own KB.
- **non-unique-name** (resource :non-unique-name <agent name> :unique-name <agent name>)
  Sent by an Agent Name Server or Agent Broker to an agent who submitted a non-unique agent name. The ANS provides a new unique name for the agent.

C.3.2 Ontology: broker

- **subscribe** (service :type <service type>)
  Sent from an agent to the ConceptBroker asking for a list of currently registered services of the specified type (one of ConceptServer, ConceptClient or ConceptSearch). The agent wants to be kept up-to-date when the list changes.
- **tell** (service :type <service type> :available ("<service name>" <service agent name> "<contact info (e-mail)>" ["<service name>" <service agent name> "<contact info (e-mail)>"]*))
  List of available services, sent from the ConceptBroker to an agent. The list can contain several items, each with all of the 3 fields.
- **untell** (service :type <service type> :remove ("<service name>" ["<service name>"]*))
  List of no longer available services, sent from the ConceptBroker to an agent when a service went offline.
- **register** (service :name "<service name>" :type <service type> :contact "<Contact responsible person (e-mail)>")
  Sent from an agent to the ConceptBroker to register one of its services of the given type.
- **unregister** (service :name "<service name>")
  Sent from an agent to the ConceptBroker to signal that a service that has been registered is no longer available.

C.3.3 Ontology: concept

- **ask-all** (concept)
  Sent from an agent to a ConceptServer agent to ask for all the concepts of the concept tree.
- **ask-one** (concept :name <full concept name>)
  Sent from an agent to a ConceptServer agent, asking for a concept. If the name is “root”, then the root concept of the concept tree is served.
- **tell** (concept :name <full concept name> :abstract "<concept abstract>" :number <number of links> :subtree (<subconcept name> [<subconcept name>]*) :relation (<related concept name> [<related concept name>]*))
  Sent from a ConceptServer agent to another agent, contains the basic concept data. The subtree and relation fields can contain several items. The number of links that this concept node contains is also given.
- **ask-all** (link :concept <full concept name>)
  Sent from an agent to a ConceptServer agent asking for all the links of a concept.

- **tell** (link :concept <full concept name> :type <link type> :name "<link name>" :url <link url> :description "<link description>" :origin "<link origin>")
  Sent from a ConceptServer Agent to an agent, telling it a single link. Several of these messages are sent in a stream if the link list contains more than one element.
- **insert** (link :concept <full concept name> :name "<link name>" :url <link url> :description "<link description>" :service <service type>)
  Sent from an agent to a ConceptServer agent to insert a link into a concept node. The server will note which agent the link came from.
- **delete** (link :concept <full concept name> :url <link url>)
  Sent from an agent to a ConceptServer Agent to delete a link from the database. An agent can only delete its own links, requests to delete other agents links will be ignored by the server.

Appendix D

The concept.data file format is a plain text file. Field labels are used to identify the following information. They begin with the “#” character and must be written exactly as shown below (they are not comments!). *Name* and *Conceptpath* must be one line each, the *Definition* and *Keywords* can be several lines of text. A concept can have any number of *Relations*, one per line and each with their full pathname. The server links are listed after the *Links* field, one link per line and the 4 different fields of a link must be separated by tabs. Links from other agents are listed after the *Others* field.

```
#Name
Multi-Agent_System
#Conceptpath
root/Computer_Science/Artificial_Intelligence/Distributed_AI
#Definition
This concept contains information about multi-agent systems.
#Keywords
software agent, multi-agent system, intelligence, autonomy, machine
learning
#Relations
root/Computer_Science/Language/Communication/Agent
#Links
UMBC Agent Web  http://www.cs.umbc.edu/agents/
Collection of information about agents.
KIServ passwd
#Others
Agent Taxonomy http://www.msci.memphis.edu/~franklin/AgentProg.html
Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents
bleyer@ki topsecret
Intelligent Agents      http://www.doc.mmu.ac.uk/STAFF/mike/
ker95.ps    Intelligent Agents: Theory and Practice.
bleyer@ki topsecret
```

Appendix E

Listed below are the JAT and CEMAS Java packages. The complete source code documentation would be beyond the scope of this paper. It is available online in HTML format at:

<http://www.informatik.uni-ulm.de/abt/ki/Students/mb/cemas/docs/packages.html>

JavaAgent.agent
JavaAgent.context
JavaAgent.resource

ConceptBroker.agent
ConceptBroker.context
ConceptBroker.resource

ConceptClient.agent
ConceptClient.context
ConceptClient.resource

ConceptSearch.agent
ConceptSearch.context
ConceptSearch.resource

ConceptServer.agent
ConceptServer.context
ConceptServer.resource

ConceptUtil

Name: Michael Sebastian Bleyer
Matrikel-Nr. 0258996

Erklärung

Ich erkläre, daß ich die Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den

PostScript-Fehler (--nostringval--, findresource)