An Information Retrieving Framework for Multi-Agent Systems

Niklas Eklund
Department of Microelectronics and Information Technology
Royal Institute of Technology, Stockholm, Sweden

June 6, 2001

Abstract

As multi-agent systems become more commonly used in companies, there is a need to supply those systems with quality information. The information will come from different information sources, such as financial, intranet, and other core systems. In addition to the actual information, the multi-agent systems also need to understand the information, and how various pieces of information relates to other pieces of information. The framework presented in this thesis, tries to provide all the necessary information to multi-agent systems in order to make them as powerful as possible. All kinds of information sources are supported, and can be integrated seamlessly into the system.

The presented framework is a part of the Informats project at Spectral AB, and is based on Java, Jini/JavaSpaces, and XML.

Contents

Section 1

Introduction

1.1 Background

In today's world when there is more and more information available, there is an increasing need for users to receive only relevant information. Therefore the use of agent systems that filter, and prepare information, will become more and more widespread. In a typical medium to large company, there might be more than ten or fifteen different information sources that would be of interest for employees. The task of keeping up to date by querying all those sources actively, is daunting and time consuming. A lot of the information contained in the sources is also likely to be related to information in other sources. To complicate things, the terminology used in the different sources normally differs. This difference in terminology can manifest itself by for instance calling employees 'employees' in one system, and 'workers' in another. To overcome these differences, a common domain model, or ontology, is necessary.

New knowledge can be attained when information from different sources can be related to each other. Obvious relations like the employee-worker relationship can be exploited, but also new, and previously unknown, relations can be discovered. To find these relations, and to filter information for a specific user, agent technology is interesting. The agent paradigm is not only suited for per-user information filtering, but also to reason about meta information and to draw conclusions. A lot of effort and research is being put into exploring the agent paradigm.

People are also accessing systems using different types of clients with different capabilities for presenting information. Clients can range from normal web browsers to mobile phones using SMS. In order to support a great number of client types, flexible presentation systems are required. By using XML[1] and XSL[2] transformations, that presentation flexibility is successfully achieved in many systems.

At Spectral AB[3] a new product, with the working name Informats, is being developed to overcome the overload of information within companies. The aim of the product is to provide personalized, updated, and relevant information to employees. The information can be delivered at request, or by subscription. The product, hereafter referred to as IM, should allow employees to access it from basically anywhere, with whatever kind of client they have at their disposal. As a first step IM will only provide information metadata, and pointers to where the actual information can be accessed.

The IM system is composed of three major layers as seen in Figure 1.1:

- Adapter layer - retrieves metadata and provides ontology information from different information sources.
- Multi-agent layer - reasons about the metadata and determines which users would be interested. Also stores the metadata.

[1] Extensible Markup Language
[2] Extensible Stylesheet Language
[3] Spectral AB is the industrial commissioner of the work related to this thesis (http://www.spectral.se).

- Presentation layer - presents the information to the user in a suitable form, depending on the client used.

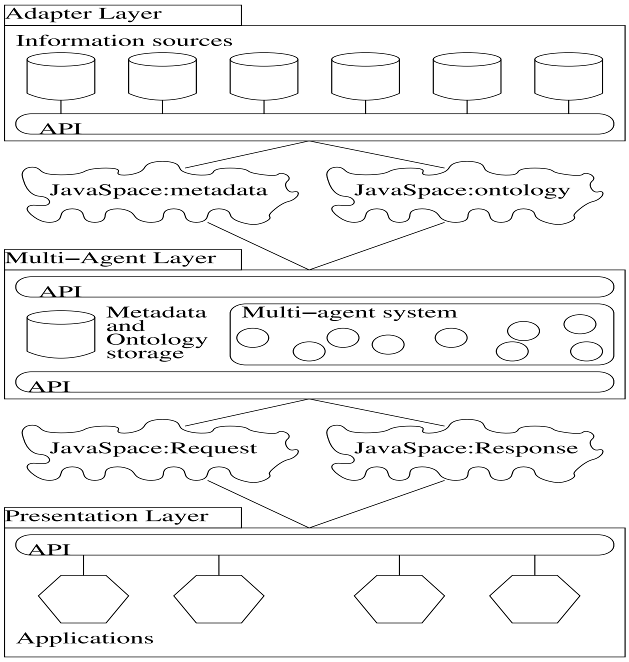

Figure 1.1: An overview of the different layers of the IM system and their internals.

This thesis covers the various aspects of the adapter layer. Specifically how information is retrieved, and how it is sent to the multi-agent layer. The current IM system is to be considered a prototype or as a "proof of concept". It is meant to prove that the concept is feasible, and worth to explore. As a consequence, some functionality has been left out or has not been fully implemented. For instance, there is only support for one type of client (web browsers), and the multi-agent system does not try to find new relations within the information.

Work made in the multi-agent layer and the presentation layer of the IM system, is presented in [Kalen, 00], and [Mattson and Noren, 01], respectively.

1.2 Objective

This thesis is intended to describe a framework for information retrieval from different information sources. The main objective is to investigate the information-related requirements of multi-agent systems in general, and the type used in the IM system in particular, and to design the framework accordingly. The proposed framework will be targeted at the IM system, since it was conceived as a part of it, but it can be used with other similar multi-agent systems with little adaptation.

The initial requirements are that the framework should be extensible, and that it should be easy to add new information sources. The new information sources should be seamlessly integrated into the system, without requiring any manual intervention. For the multi-agent system, the only impact should be that there is more information available. The framework should also be able to support different types of multi-agent systems using different notations for ontologies.

As a part of this thesis work, two information sources will be implemented in the IM system using the proposed framework. The two information sources are Spectral's two major products, Spectral Projektverktyg[4] and IntraWise. Both products are web-based tools that are used by

[4]This is a temporary name due to trademark issues with the former name.

several medium to large companies. They are described in Section 3.

1.3 Related Work

There are a lot of similar projects that try to unify a number of information sources. Most of them are targeted at seeing the different sources as one big information source. A lot of effort has gone into developing query languages and caching algorithms for these systems.

Some systems like InfoZone [InfoZone] take the approach of storing all data in an internal database, and just deliver the contents. InfoZone does not use agents at all, but is more of a content management system or EIP[5]. It lets applications, using the InfoZone tool set, treat all content in the system as one large XML document. InfoZone can access a variety of information sources through wrappers and is implemented in Java.

The SIMS [Arens et al., 93] system describes itself as an information mediator. The system does not store any information internally, and the location of the actual information is transparent to the user. All queries posed to the system are in a uniform language, independent of the location, and the type, of the information source. The query is analyzed and then split up over the different sources that need to be queried. The partial queries are then translated into the relevant sources' native query language, and sent to the source for processing. When the results from each source are returned, they are combined according to the original SIMS query, and returned to the user. In order to be able to query the SIMS system, all sources must be included in the system information model that describes the information world. The SIMS system can only handle Oracle databases and Loom knowledge bases.

A slightly different approach is InfoSleuth [Bayardo et al., 97]. InfoSleuth consists basically of collaborating agents that perform tasks initiated by a user. The agents are highly specialized and cooperate to complete the requests. A broker agent controls the system by keeping records of the other agents, and acts as a matchmaker service. The other agents ask the broker which agent to contact for a specific service. All information sources are wrapped using information resource agents that handle all communication with the information source. Like the SIMS system, InfoSleuth does not store any information internally, and queries are handled similarly.

InfoQuilt [Shah and Sheth, 98] is a project aimed at building logical, semantic, webs using metadata. It is an information mediator system that uses metadata to correlate information from different sources. The metadata is stored in a metabase as RDF[6] objects. By constructing directories of these metabases, user agents can present a visualization of the available information to the user. Broker agents handle queries in a similar manner to SIMS and InfoSleuth.

Another system is Mappa [Mappa], which seems to be more targeted at customer support in the retail industry. Users will have personal agents that are activated when the user accesses an information kiosk, or POS[7]. The system will contain metadata about products, such as price, availability, and descriptions. The personal agents are capable of information filtering, and subscriptions. An adaptive presentation system allows the customer to perform transactions using diverse clients. The system uses XML and Java technology.

The interesting part of all of the systems mentioned above are the handling of the information sources. It ranges from copying all information, to having agents query the source at run-time. A common denominator is that ontologies are involved in some way. The IM system does not store the actual data, but only metadata. This allows for faster searching, since the data sources themselves do not have to be queried, and there is no need for complicated query planning. Instead it has the problem of keeping the stored metadata in synchronization with the data in the information sources.

Unlike some of the systems mentioned above, no agents are used as wrappers around the information sources. The idea is to have a simple flow of data from the data sources into the system, without any filtering or processing. The system is not limited to a few types of data

[5]Enterprise Information Portal
[6]Resource Description Framework
[7]Point Of Sale

sources. It can handle all types of sources through adapter extension, via a simple API. When security is introduced later in the project, the information sources themselves must however be queried for permissions. Security is discussed in the final section, which deals with future work.

The client must access the source directly to retrieve the actual data. This approach lets the data sources themselves present the actual information, which makes sense, since a lot of care is likely to have gone into presenting the information in a logical and user-friendly manner. There is no need to waste time building a second user interface, and information security can be enforced by the source itself. In addition, the user is likely to feel more comfortable using an interface that he or she has used before, and is accustomed to.

InfoZone and InfoSleuth are the systems most similar to the IM system. InfoZone uses the same fundamental technologies, namely XML and Java. The IM system depends natively on agent technology to serve personalized information pointers, in contrast to InfoZone. Although that could be implemented through an application using InfoZone's tool set, it is not included in its core technologies. InfoZone is on the other hand able to update information contained in the sources, but that feature is not needed in the IM system.

InfoSleuth also uses Java, and is capable of integrating basically any information source into its system, via wrapper agents. There is also support for subscriptions, but as mentioned above, there is no intermediate metadata storage, and the system performs advanced query planning.

1.4 Technology

Both the framework and the IM system are implemented using the Java programming language. The choice of Java was purely practical. It is portable, network oriented, and available for a large number of platforms. These are some of the reasons why it is also the favored programming language used at Spectral. The reader is supposed to be familiar with Java and object orientation concepts.

XML is the new popular standard for data exchange. The IM system uses XML for messages and information sent between the different layers. XML is also used internally in the presentation layer.

A number of JavaSpaces are used in the IM system to cleanly and neatly separate the layers. In the multi-agent layer, Jini technology is used to implement the agents.

Ontologies are an important part of the IM system. Ontologies are domain models that describe the terminology and the relationships between entities in a given domain.

These technologies are described in Section 2.

1.5 Methodology

The initial phase of the thesis work was partly dedicated to studying the IM system and the related technologies. In addition, the information sources to be used in the implementation were studied. After initial reading, the Internet was searched for related work. Some previously known similar systems were also used as a starting point. Publicly available information about the found related work was then read and analyzed.

To get familiar with the technologies to be used, especially JavaSpaces and ontologies, some practical training was also necessary. A few technology-related tools were also tested.

All members worked upon a few parts of the IM system in order to get familiar with the system design. The metadata repository, or metadata storage, was for example one of those parts.

After getting familiar with the technologies, a first attempt at a design was implemented. The initial design was looked at and analyzed for strong points and shortcomings. Then the real design phase started, and while carefully taking into account the lessons learned, the new, final, design was completed.

1.6 Structure of Thesis

In Section 2, the technical platform selected for the implementation of the prototype is presented. Section 3 presents the two information sources to be used in the implementation. It also examines what kind of metadata that can be retrieved from the sources. There is also a discussion about each source's information structure, and ontology. Section 4 discusses the design and implementation of the framework. A framework design is presented there. Two information sources implemented in the IM system, by using the framework from Section 4, is presented in Section 5. The final section contains a discussion on what has been achieved with this work, as well as possible future considerations and improvement suggestions.

Section 2

Technology Overview

In this section the core enabling technologies used in the framework, and in the IM system, are presented. The framework and IM implementation uses Jini, JavaSpaces, ontologies, agents and XML. A short subsection on data integration is also included to present some of the work that has been performed in this area.

2.1 JavaSpaces and Jini

Jini [Jini] is a technology from Sun Microsystems, which initially was conceived as a solution for enabling ad hoc networking between hardware devices, such as printers and other service oriented devices. It has since been proven to be useful for non-hardware development, by helping to provide front-ends for software services.

The beauty of a Jini network is that it is almost self-contained after it has been set up. No further planning, installation, or human intervention is necessary, in theory at least. Each device provides services to other services or clients, and each device also provides the interfaces to access those services. When joining a network, services register themselves with a lookup service by uploading a Java object that implements all of its services' interfaces. This object is commonly referred to as a server proxy object. In order to use a service, it has to be looked up in the lookup service. The lookup service provides the desired service's proxy object for downloading. When the client has downloaded the proxy object, the lookup service is not needed until a new service is requested. The client communicates with the service directly via the proxy object, by invoking its methods. The proxy object handles the communication with the back-end of the service. If the service does not have to perform any complicated computations, there might not even be a back-end to the service object. A simple hypothetical time service, that only provides the current time, could for instance have its implementation entirely in the proxy object.

If a service leaves the network, it simply cancels its registration with the lookup service. In order to provide robustness to the network, the services must periodically tell the lookup service that they are alive. This is solved by using "leases" that are granted for a short time period. If a service fails to renew its lease, it diverges from its expected behavior, and is quietly removed from the network.

2.1.1 JavaSpaces

JavaSpaces [JavaSpaces] is a Jini service that is based on the concept used by Linda systems [Linda Systems]. It can be described as an object store that can handle distributed transactions. A JavaSpace can be used for long or short time storage of Java objects that implements a certain marker interface, called Entry, from one of the core Jini packages. In order to allow the objects to be created without knowing specific details about the objects' class, they must also have a constructor that takes no arguments. Another restriction on the Java objects, is that fields that are to be used for matching, i.e. public variables, must be objects themselves, and not primitive types.

A JavaSpace only supports three basic operations. Objects can be read, taken, or written. These three simple operations are enough for efficient use of a JavaSpace. The space can be either persistent or transient. A persistent space remembers its contents when restarting, and is mainly used for longer-term storage, while transient spaces are mainly used for temporary objects. The interface and operations are the same regardless of the nature of the space.

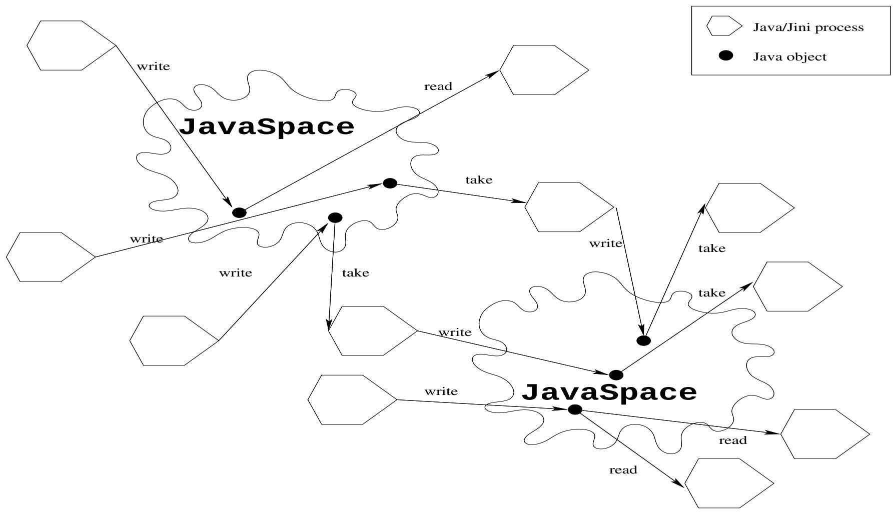

Figure 2.1: An example of object reading, writing, and taking with JavaSpaces and multiple processes.

When a program writes an object to the space, the object is serialized and uploaded to the space. The object is stored in the space for some amount of time using leases in the same way as for Jini services. If the write operation is part of a transaction, and the transaction fails, the object is not written to the space.

To get hold of an object in a space, the object has to be taken. To take a specific object, a template has to be filled out with matching values. A template is simply an instance of the object's class, with its fields set to values that match the desired object. Unset fields, that are `null`, are considered to be wildcards. Matching is done on both class and fields. If the operation was successful, and there was a matching object in the space, the object is sent to the client and then removed from the space.

The read operation is used to peek at the contents of a space. It is similar to the take operation, but instead of the actual object, a copy is sent to the client, while the object remains in the space.

JavaSpaces can be used as object stores, distributed work queues, or as simple message queues, for instance. In the IM system, JavaSpaces are used as abstraction layers between the system's different components in the form of distributed message queues. Each space can have multiple taker and writer programs running concurrently on different machines. This allows the system to scale well. If a service is running too slowly, it is only a matter of starting another copy of the service on another machine. If it is the actual space that is the bottleneck, a new space, with the same name, can be added to balance the load. If new spaces are added, some care must be taken by the writers to distribute the objects over all spaces, as well as for the clients to look in all spaces that have the same attributes. There are, however, vendors that are working on truly distributed JavaSpaces, i.e. JavaSpaces that are logically centralized, but physically distributed across several machines.

In [Freeman et al., 99], JavaSpaces are described more extensively, and their uses and benefits are thoroughly explored.

2.2 XML

XML [XML] has become the new popular standard for data exchange. XML is an abbreviation for Extensible Markup Language, and is a subset of SGML[1]. There exists a wealth of powerful tools, both free and commercial, for XML parsing and manipulation. The technology is not only in demand at the time of this writing, but it is also being integrated into all thinkable software products.

An XML document is composed of a series of entities in a tree structure. Each entity can contain one or more logical elements, which in turn can have one, or more, attributes. The attributes normally describe the way in which the element is to be processed. XML provides a formal syntax for describing the relationships between entities, elements, and attributes. The XML tags fully enclose all data in the document by surrounding the data fragments with a start-tag and an end-tag, unlike for instance HTML, where some tags are allowed open. While HTML has a set of predefined tags, in XML it is totally up to the creator what tags to use.

XML can be described as a hierarchical database in a text file. By using a couple of standard API's[2] such as SAX[3], and DOM[4], it is possible to extract any of the information contained in the XML document. The extracted data is, however, only useful if its semantics are understood. To make sure a document is understandable to a receiver, it should conform to an agreed upon DTD[5] or XML-Schema. DTD's and XML-Schemas, describe the syntax, i.e. the tag names and possible attributes that are allowed in a conforming XML file. XML-Schema is the successor to, and more elaborate than, DTD's.

```
<?xml version="1.0"?>
<order orderid="13432" customerid="523423">
   <cd id="INXSKICK" artist="INXS" cost="130">Kick</cd>
   <cd id="U2UNF" artist="U2" cost="115">The Unforgettable Fire</cd>
   <book id="GRISHCLIENT" author="John Grisham" cost="235">The Client</book>
</order>
```

Figure 2.2: An example of how an order of two CD's and a book could look when expressed in XML.

There are also tools to transform one XML document into another XML document by using a stylesheet. A stylesheet is basically a set of rules that dictates what to output, for any given input tag. This is very common in web publishing since it can help separate the content from the layout and design.

2.3 Ontologies

An ontology is a vocabulary of different basic terms and their definition, and how they are related to each other. In order to serve the agents in a multi-agent system, some form of "explanation" must accompany the information supplied. The "explanation" should define the basic terms and describe their relationships. It should, to be useful in our case, also include other useful information, such as agent-specific instructions and rule sets, i.e. axioms. The inclusion of axioms would strictly speaking make the resulting "ontology" something more than the strict concept of an ontology. It is, however, commonly accepted to do this inclusion, and call the composition an ontology.

Good, large scale, ontologies should, as stated in [Hee Hwang, 99], have the following somewhat inter-dependent properties:

[1] Structured Generalized Markup Language
[2] Application Programming Interface
[3] Simple API for XML
[4] Document Object Model
[5] Document Type Definition

- Open and dynamic - construction and modification should be easy and, preferably, possible to automate
- Scalable and interoperable - it should be scalable and be able to merge with other ontologies
- Easily maintained - it should be easy to keep the ontology up-to-date since it will otherwise be short-lived
- Semantically consistent - the representations and axioms should be semantically sound to avoid ambiguity
- Context independent - makes it easier to merge ontologies, and otherwise requires context-discharging axioms

These properties must also in some parts apply to, or at least work in harmony with, the system using the ontology. It is for instance no real need for an ontology to be context independent if there is only one well-structured data source. On the other hand, if there are a lot of different data sources, the ontologies should be scalable and interoperable.

There are many languages available for writing and constructing ontologies. Some are used today in real-world systems, while others are mainly for education and research.

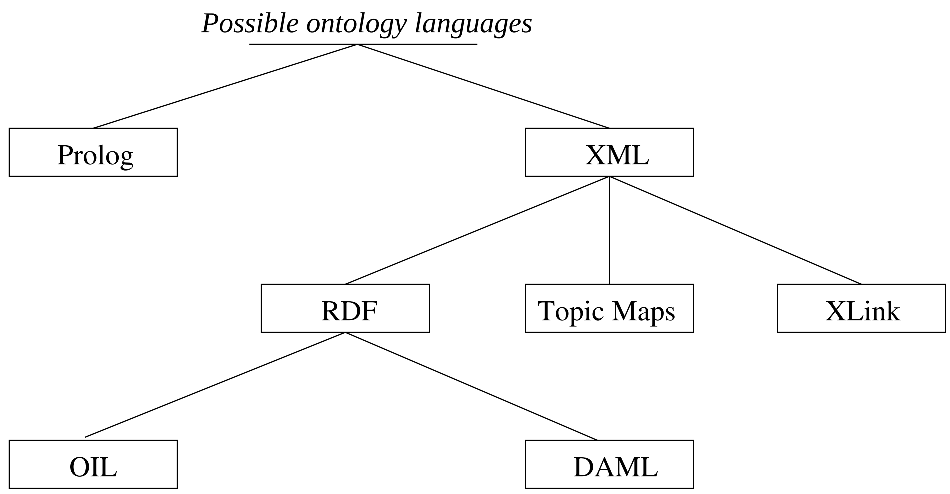

Figure 2.3: The relationship between different ontology language alternatives.

By using pure XML, one has total freedom to introduce whatever axioms and language constructs needed, to make the language simple and comprehensible. This freedom means, on the other hand, that the syntax and the semantics have to be designed especially, and that common standards can only be used for inspiration. To be able to reason using a home-grown XML ontology there are basically two options. Either a tailored reasoning engine must be implemented, or two stylesheets must be provided that translates the XML syntax into some other language, and back again. A translation could, for instance, be performed, from the XML syntax to Prolog clauses, and thus have a Prolog engine do the inference.

RDF is and extension to XML where certain tags have been designed to describe net resources, and provide links to them. The descriptions are domain neutral and do not impose any semantics. A few inference engines with tools for processing RDF exist, but they are not well suited for ontology processing.

By using a layered approach on top of RDF, OIL, the Ontology Inference Layer [OIL], adds more and more complex semantic functionality with each layer, while keeping compatibility with the underlying layers. OIL adds formal semantics and reasoning notions provided by description logic. Commercial inference engines exist that handle OIL, and conversion stylesheets/software from OIL to FaCT[6], that allows a FaCT inference engine to be used.

DAML, the Darpa Agent Markup Language [DAML], is another layer on top of RDF that adds semantic syntax for expressing ontologies. The technology is backed by DARPA, and there are a number of ontologies available in DAML that covers domains as diverse as job titles and beer. Work is on the way to integrate DAML with OIL. The tool support situation is, at the time of this writing, unclear.

Topic maps is a general technology for connecting concepts and describing them using XML. It can be used for many other purposes such as information navigation. It can also be used to express ontologies and relationships if augmented with some semantics. Some tools and API's exist for using Topic Maps.

XLink, the XML Linking Language [XLink], is a standard that, as the name suggests, allows the expression of links, or relationships, between XML elements. It also allows metadata to be linked to elements, or to the links themselves. Since it is constructed on top of XML and is very flexible, it can serve as a base for an ontology language. A couple of APIs exist to take advantage of the XLink technology.

Prolog clauses can also be used to describe ontologies, with the included advantage that the language is ready for inference by any Prolog inference engine. The disadvantages are similar to those of pure XML, that one has to define everything and that there is no standard.

A simple "pure" ontology in a pseudo, OIL-like, language in the domain of football is shown in Figure 2.4.

```
ontology-container
  title "Football"
  description "A simple football ontology"
  type "ontology"
ontology-definitions
  slot-def passes-to
    inverse is-passed-by
  slot-def has-part
    inverse is-part-of
    properties transitive
  class-def player
  class-def referee
    subclass-of NOT player, NOT manager
  class-def manager
    slot-constraint is-part-of
      has-value team
  class-def defined teammate
    subclass-of player
    slot-constraint passes-to
      value-type teammate
    slot-constraint is-part-of
      value-type team
  class-def defined opposing-player
    subclass-of player, NOT teammate
    slot-constraint passes-to
      value-type opposing-player
```

Figure 2.4: A small, and simple, example ontology describing football.

The example ontology contains a couple of entity classes and relations. It states that some players are team-mates, and that team-mates only pass to other team-mates. The ontology also states that a manager is part of a team, and that a referee is neither a player, nor a manager.

2.4 Agents

The software agent paradigm sprung out of the need to have many different programs interoperate. With programs written by different people, using different languages, and at different times, that task is not easy. This heterogeneity made standard ways of writing middleware programs unsuitable. The agent programming style uses components, called agents, which communicate with their peers using some expressive communication language. Agents can be as simple as subroutines or

[6]Fast Classification of Terminologies

if-clauses, but typically they are more complicated. The agent communication language should allow the agents to exchange data and logical information, simple commands, and possibly even program instructions.

Usually agents expose what kind of services and capabilities they have. Other agents or/and applications can then utilize these services. By having a dynamic network of agents working together, the whole system can come up with solutions to difficult problems more efficiently than a single program would. Agents normally keep an internal model of its surrounding environment and are autonomous, i.e. they do not need any human intervention.

An incomplete listing of some agent types that exist, is presented below. The list is by no means exhaustive, and is here for orientation only. For a proper overview see [H. S. Nwana, 96].

- Collaborative agents - tries to find solutions to problems by using other agents
- Interface agents - learn from user behavior or other agents, and assist a user
- Mobile agents - roams networks and performs task by collection information from the network and/or other agents
- Reactive agents - sense the environment and acts upon certain stimuli

Agent types that will be used in the fully implemented IM system are collaborative, interface, and reactive agents.

By using ontologies for reasoning, agents can come up with associations and even unknown facts that can be beneficial to a user. An agent could for instance contribute when starting a new project, by using known ontology relations. It could provide the project manager with reports from other similar projects, suggest suitable project member candidates, and prepare project documentation. While the project agent does this, other agents could analyze all its data, and other known data from other sources, on a higher level. This analysis could come up with implicit rules and associations such as "projects using supplier A that run during the summer need an extra week's running time since that supplier's deliveries are always late during the summer"; or "if a manager has a scheduled meeting in the company's offices in city X at 10:00, a meeting scheduler agent should not book any meetings for her in city Y that same day".

Of course these implicit rules and associations differ greatly depending on the nature and quality of the data available. It also differs on what kind of reasoning capabilities the agent system have, and what reasoning algorithms are used. A description of how the agents in the SIMS system use ontologies is presented in [Ambite and Knoblock, 95].

2.4.1 Multi-Agent Systems

In a multi-agent system, MAS, the agents work together to achieve a set goal. The best results have been achieved with systems that are heterogeneous, and using learning agents. The main characteristics for a MAS are, as stated in [Jennings et al., 98]:

- Each agent has limited information and capabilities
- There is no central system control
- Data is fragmented and spread out
- Work is done asynchronously

To achieve the goal given to the MAS, the agents need to communicate with each other. There are a number of agent communication languages, ACL's, available. Among the most well known are KQML, the Knowledge Query and Manipulation Language [KQML], and FIPA[7] ACL [FIPA ACL].

KQML is a high-level message-based communication language that is independent of the message content language, and the ontology language. It is also independent of the transport protocol,

[7]Foundation for Intelligent Physical Agents

and can work with any transport such as TCP/IP, SMTP or IIOP. The language was created with the design goal of being able to support a variety of different agent architectures. There is an extensible set of primitive functions, and the syntax is based on balanced parenthesis lists, as used in Lisp.

FIPA ACL is similar to KQML except for different names for the primitives. The language is basically independent of the message content language. There are, however, some primitives that imply that the agent understands a language called Semantic Language. There is also a difference in semantics. For detailed information, and a comparison of these two agent communication languages, see [Labrou et al., 99].

The agents in the MAS must cooperate using some strategy in order to be successful. A few strategies exist that have proven more or less effective for different problem domains. A survey of these strategies, and multi-agent systems in general, is conducted in [Stone and Veloso, 96]. The strategies include game theory and resource management.

There are a lot of possible applications for multi-agent systems. One example application is the one used by the SIMS system and InfoSleuth. They use the MAS to make a user or, client program, see a set of distributed information sources as one information space. To query the data, they do advanced query planning to pose fragmented queries to the information sources, and then combine the result.

Information about the agent structure, application development, and example applications in InfoSleuth, is presented in [Nodine et al, 98].

2.5 Data integration

Many multi-agent systems strive to view multiple distributed heterogeneous information sources as one big information source. A lot of research has gone into designing techniques to query distributed information sources using a uniform language. [Levy, 98] gives an overview of how artificial intelligence techniques can be used for data integration, and in [Kashyap and Sheth, 96], an approach to combining information sources by using domain specific metadata, is presented.

A framework for extracting metadata, and using a metadata repository, similar to the IM system, to query multiple information sources, is presented in [Martin et al., 98]. That framework has its focus mostly on web sources and pure information systems, rather than agent systems.

Section 3

Information Sources

The information sources that are to be used in this thesis work are Spectral's two major products, IntraWise and Spectral Projektverktyg. They are both web applications, IntraWise is implemented in Java, and Spectral Projektverktyg in ASP[1] and Visual Basic.

3.1 IntraWise

IntraWise is an Internet/intranet information tool that allows users with the proper authorization to publish information. Information can be published in the form of objects on pages, such as paragraphs, news items, and tables. A standard web browser is used for editing of the different pages, i.e. there is no need to use an external editor, such as Microsoft FrontPage or Macromedia's DreamWeaver. Users do not have to be familiar with HTML, since the tool performs the formatting. See Figure 3.1.

Since IntraWise is written in Java, and is implemented using servlets and special Java utility servers, the system is platform independent. The data storage is also platform independent, as the data can be stored in any relational database.

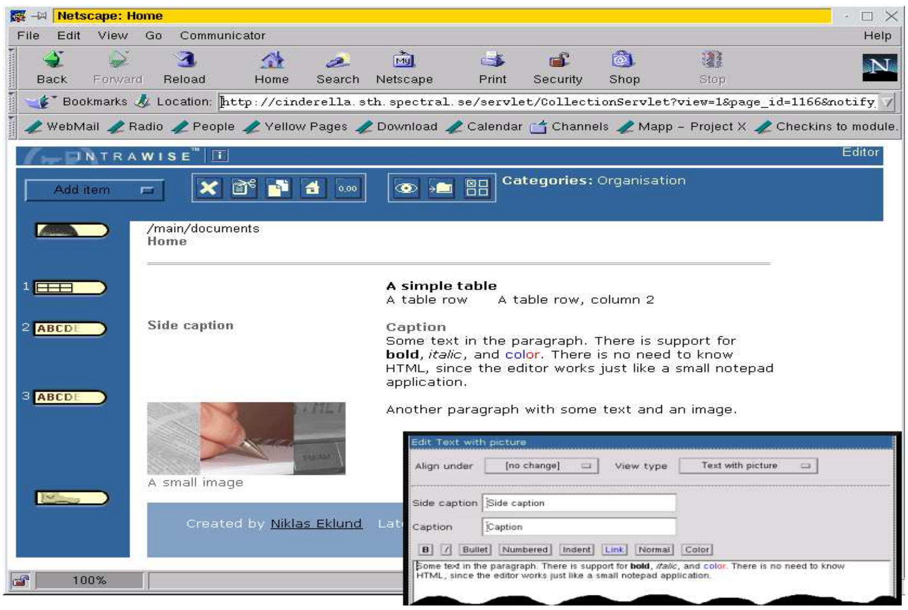

Figure 3.1: The page editing view in IntraWise. Inlined is an image of the paragraph editing view.

[1]Active Server Pages

The pages are held in a folder structure similar to a conventional file system. Images and documents can be uploaded and put on pages, together with hyperlinks and user information, such as name and photo. The layout is locked to the company standard layout using templates, and users can concentrate on the actual information they want to present. See Figure 3.2. One of the greatest advantages, besides the in-browser editing, is the possibility of distributed information publishing. The user from whom a piece of information originates, can directly publish that piece of information if the permissions are set up properly. She does not have to mail it to the Webmaster or some editor-in-chief.

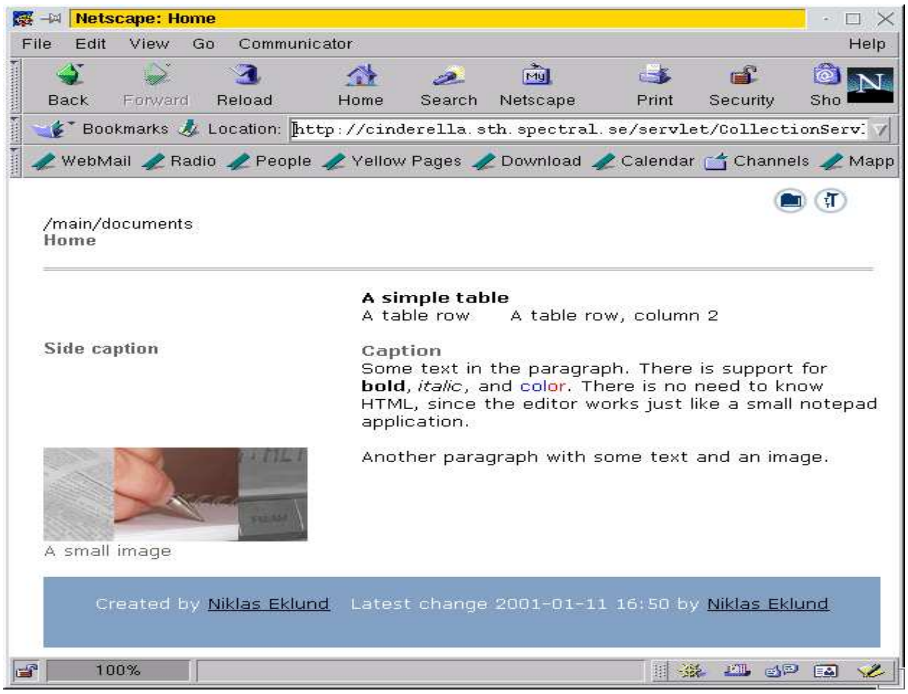

Figure 3.2: This is how the page looks to the user. The layout is locked to the company layout.

In addition to the textual information, each page can also be categorized according to a set of categories ordered in a tree structure, by using the category module. The system's administrator builds the category tree. The categories can be used to navigate the pages in the system in the same manner as Yahoo[2], or dmoz[3].

There is also a booking module in which authorized users can book custom-created objects. Reservations can be per half hour, hour, or day. The user can see when a bookable object is available in a graphical schedule. The schedule also shows who has booked the object for all times, and optionally why. There is also the possibility to send an e-mail to any user in the schedule.

A third useful module is the discussion board. It is actually more than a discussion board since it can be used to administer FAQ's, present quizzes, as well as the normal discussion board features.

Every time a user creates or updates a page, a timestamp is stored as well as the user's username. This makes it simple to see when a page or news item has been changed, and by whom. It is also possible to find out additional information about users, such as their email address and full name, since IntraWise normally is connected to an LDAP[4] server. Because IntraWise can be, and is, used to publish any kind of information, it is difficult to find domain specific meta-

[2]www.yahoo.com

[3]www.dmoz.org

[4]Lightweight Directory Access Protocol

information. There is however a lot of domain-neutral meta information in database fields, such as titles, subtitles, and dates/timestamps.

3.2 Spectral Projektverktyg

Spectral Projektverktyg is a web-based tool to manage virtually all aspects of multiple projects. Projects can be set up, members can be added, and the progress of the project can be monitored. The projects are assumed to follow a derivative of the PROPS project model, developed by Ericsson. Each project has its own web section containing all information about the project. To get an overview of all projects, there is a project list where the projects are categorized by their status. The project list can be sorted, for example, using project type and project manager.

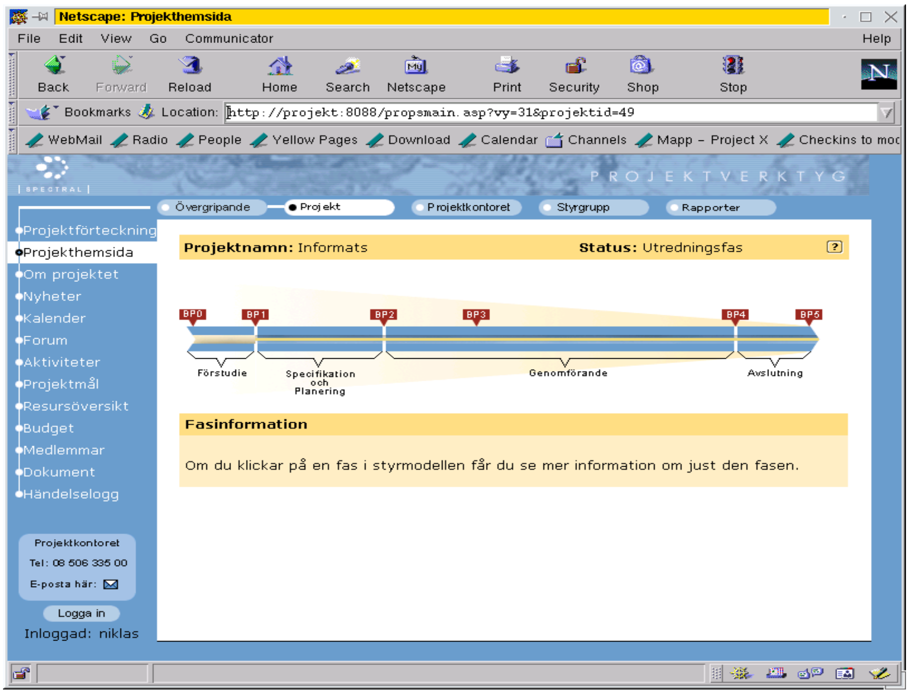

Figure 3.3: The project page for the IM project. The project model shown is called "raketen", the rocket, which is Spectral's project model.

A review board that has the authority to advance the project's phase, when the proper milestones are reached, controls each project. The operational control is, however, in the hands of the project manager. The project manager can insert estimations of time and funds, along with resources, project goals, and deadlines. When a project is underway, estimated costs can be reviewed in respect to the actual costs, as well as resource use, and the fulfillment of project deadlines.

All documents concerning the project can be uploaded and accessed by the project's members. When a new project is started, and created in the tool, a new set of project documents is automatically created from templates. In the project's home pages a user can get an overview of the achieved goals, and see an activity list complemented by a calendar. It is also possible to produce reports with the integrated report generator that supports many different selection criteria. Graphical reports can be shown for a quick overview, as well as comparisons to other projects.

Included in the tool is also information about the project model being used. Users that are unfamiliar with the project model can quickly learn from studying the contained information. Subjects include when to do what, in which order, and which documents are involved.

The information contained in Spectral Projektverktyg is a lot more structured than in IntraWise. The problem the product solves is more easily identified, which simplifies the question of what to extract from the system. Although the projects themselves can be from all domains, the meta-information available about the projects is domain-neutral.

Section 4

Framework

In this section a framework for information retrieval from multiple information sources is presented. The information delivered by an implementation of this framework, is targeted at multi-agent systems. That implies providing the necessary data, in addition to the actual information, needed by a MAS, such as ontologies. The framework is based on Java and Jini/JavaSpaces technology, although JavaSpaces is only considered a transport medium that can be changed. The design criteria for the proposed framework are as follows:

- It should be usable with basically all multi-agent systems, i.e. flexible
- Extensible so that new information sources are easy to incorporate
- Independent of ontology language
- Independent of data/content language
- Independent of transport medium
- It should have a simple API
- It should have simple semantics
- The MAS should primarily handle metadata

In order to keep the framework simple, some restrictions have been imposed. There is no callback support of any kind, which means that the MAS cannot query the sources, and must be happy with receiving notifications, or events, only. The main reason for this is to avoid introducing, choosing, and possibly designing, a query language that would suit all thinkable information sources. A reasonable work-around is to store the metadata in a database, in the MAS, or between the adapters and the MAS, and to query that database instead. Although this could be seen as a limitation it should result in a more responsive, and faster system. That is the way the IM system is designed.

Caching is performed in the SIMS system, but not to the same extent, since that system uses a distributed query language. How caching can be designed in such a system, and how it is implemented in the SIMS system, is described in [Arens and Knoblock, 94].

Another consequence of not providing any kind of callback interface is that there is no way to check access permissions using the framework. Enforcing security and designing security systems is difficult and complex, since security crosses module boundaries in systems. Introducing security can make even the simplest technical solution become extremely complex, depending on the granularity of the security that is needed. If retrieving publicly available information from the sources is not enough, it is possible to circumvent the lack of security callbacks by using additional security-related metadata. The MAS could remove information that a user is not allowed to view by filtering, using the extra metadata.

4.1 MAS requirements and needs

The framework is intended for using passive information sources, that only send out events when information has been changed, added, or removed. As a consequence the MAS must be able to handle a flow of events, where each event can carry a payload. This makes the framework unsuitable for agent systems that need to query the information sources directly. It is instead suited for systems that use an intermediate database to hold the information sent by the adapters.

The framework is mainly targeted at handling metadata, and not actual data. There is no restriction that inhibits the use of actual data, but the basic idea of the framework is to keep the actual data in the information sources. Only descriptive metadata, and some kind of pointer to the actual data, is meant to be sent to the MAS for processing. In order to access the actual data, the client has to access the source itself, by using the pointer, which could be an URL, for instance. By using metadata only, the amount of data to be held in the intermediate database can decrease significantly, if the actual data is large. And as mentioned before, well thought-out user interfaces for accessing the data probably already exist, and thus it is a waste not to use them.

4.2 Information source types

The framework should allow any type of system to be used as an information source. There are a lot of different types of information sources, such as mail systems, financial systems and the web hosts on the Internet. These different types of sources require different methods for extracting data. Most of these systems do not provide an API to retrieve data, which only leaves the option of accessing the source's data storage directly. Basically all systems store their information in some kind of database, typically a relational database. Some systems might store their data in plain files, and yet others do not store any information at all. All relational databases can be queried by using SQL, and the JDBC[1] API.

If the system provides an API of some sort, the framework should be able to support it and use it. Some systems have support for events that can be sent to 3rd party applications when information in the system changes. Others have support for general queries of the data contained in the system.

Polling is a technique that can be used when there is no event-based API to the information source. At regular intervals, the state of the information is checked, and everything that has changed is sent as events, or messages. When polling, it is important to save the current state of the source's information, in order to detect changes between polls. For a table in a database, this could mean keeping all of the keys in a table cached. Only storing the keys will help find new and removed rows, but it will not detect modified ones. Therefore additional means of detection must be provided. Depending on the data storage design, there might be timestamps available. If there are timestamps available, a cached timestamp can be compared to timestamps in the database table, to be able to tell when the table was last modified. The only problem with the timestamp solution is that removed rows, do not have timestamps, and thus will not be detected. A combination of storing the table's keys as well as checking timestamps, is probably the best approach in most cases when polling is used.

There are, however, systems that do not remove deleted rows, but instead flags them as removed. A batch job then performs the actual removal at regular intervals. In these types of systems it is easier to track changes, especially if there are timestamps involved as well, as the deleted rows can be probed for.

Some systems provide logs, which can be parsed by the adapter, of events in the system. In most cases the logs are not really sufficient to be the only means of detection, but as a complement they could be very useful.

When combining these techniques by using "layers" of adapters, similar to the layers of agents presented in [Unruh et al., 98] for InfoSleuth, more elaborate events could be triggered. The adapters could use statistical analysis, summaries of data, and combined events to decide if, and

[1] Java Database Connectivity

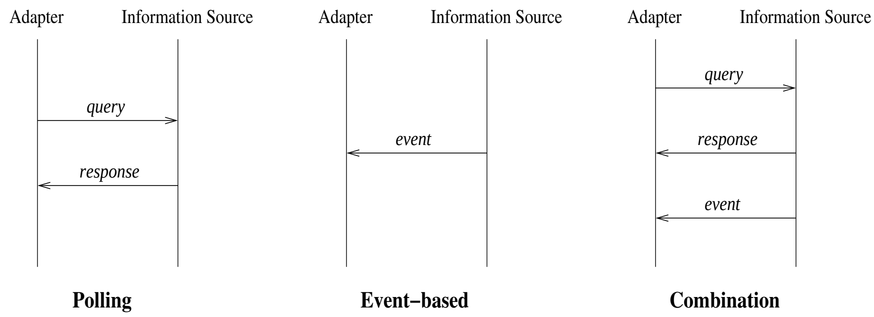

Figure 4.1: Sequence diagram showing different types of communication between an adapter and an information source

when, to send an event. The framework presented here can supply the same higher-level pattern functionality in a similar manner, by having information adapters that work with other information adapters. These extra adapters would receive their input from a number of other adapters, and depending on their combined outputs, decide whether to send an event.

4.3 Method

To be able to integrate an information source into the framework, the information source must be studied and understood. For this a domain expert, i.e. someone that has expert knowledge about the domain the information source handles, could be employed. The first thing to consider is what it is desirable to extract from the available information. This information can probably be divided into different information fragments in many ways. The fragments must be appropriate for the MAS implementation, and must be reflected in the ontology that accompanies the information. What information to extract is also dependent on what the entire system should be able to accomplish. If, for instance, an information source is a financial system that could deliver real-time data from every transaction made in the company, and the entire system only is designed to show day-to-day graphs of the financial situation, the extra per-transaction information is unnecessary. The information should be summed up and delivered once a day instead.

The next step is to decide how to extract the information. If there exists an event-based API, it should probably be taken advantage of. If that is not enough, then a combination of the polling technique and analyzing log files could be suitable. In the absence of an API, then there is no choice but to rely on querying the data storage itself, and/or parsing log files.

In many cases it may be an advantage to use different "views" of the same information source. If, for instance, the source contains a wide variety of information, a couple of adapters could divide the work between them. The smaller adapters help avoiding the complexity, and can still share resources. That approach could even have experts that only know part of the information source, to handle their special domain.

Finally the ontology for the extracted information has to be constructed. When constructing the ontology a domain expert is invaluable. Care must be taken to ensure that the ontology complies with the rest of the system, i.e. that no inconsistencies are introduced.

To sum it up - the information source integrator must understand the information source, what type of information can be extracted, and how that information should be explained via the ontology.

4.4 Information source adapters

To be able to use virtually any type of information source, it is practical to put a "wrapper" around it. The wrapper should be familiar to the framework, and be able to present the data contained in the information source in a general way. It does not have to contain any special intelligence, other than being able to extract data that is interesting to the MAS. In the framework, an information source is viewed as a source for a flow of events that each can have a payload of data. The different simple events supported are events for new, modified, and removed data. The payload for the first two events is the data itself, while the third event does not have a payload. The term wrapper and adapter will be used interchangeably to describe the same thing. See Figure 4.2 for an example of information source wrapping.

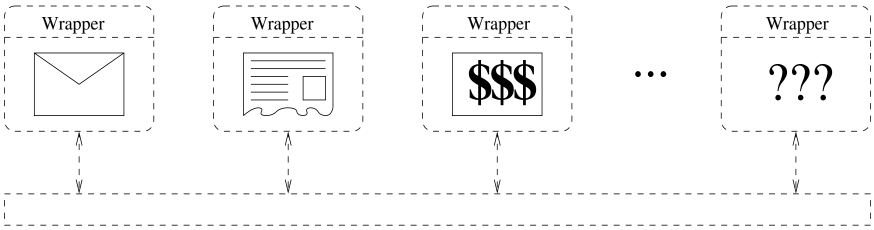

Figure 4.2: By wrapping a number of different systems, including a mail system, an information system, and a financial system, they all interact with the outside world using the same interface

4.4.1 Adapter containers

In order to make it possible to conserve resources, such as database connections, it is practical to be able to group adapters together. When using some databases, for example Oracle, the number of connections simultaneously open, can determine the license cost. Information source containers, or adapter containers, do the conservation of resources in the framework. The adapter container class passes on the information received from the contained adapters to the MAS. Each container must contain at least one information adapter, and acts as a gateway for the contained adapters to the MAS. The container implements an interface that allows the contained adapters to access common objects, such as connection pools or log facilities. Since the container is the gateway to the MAS, it is responsible for sending the events to the MAS. In order to be independent of the transport medium, the class that actually handles the transport is pluggable. It is loaded at run-time using the class name from a property value, and must implement a special interface. This pluggability, which is also the way an adapter container knows which adapters to instantiate, makes it possible to write code to handle any transport medium and protocol. The MAS must of course use the same transport medium and protocol.

At run-time, the container will load a property file and try to instantiate the configured adapters. It will also instantiate the configured transport class, and instruct it to connect. The instantiated adapters will then be started and given a reference to the container in order to allow its services to be used. The adapters will try to load any previously saved internal state to be able to pick up where they left off. The adapters will then try to access the resources they need from the container's context object. If an adapter does not find a resource it is looking for, it tries to create that service, and store it in the context object for other adapters to use also. For instance, if a container contains three adapters that have three different views of the information source, as discussed above, the first one that tries to use a connection pool to the source, will not find it, and will create it. When the second adapter comes looking for a connection pool it is already created, and ready for use. All of this has to be thread-safe to avoid both deadlock and race condition issues, as well as to avoid unnecessary extra work.

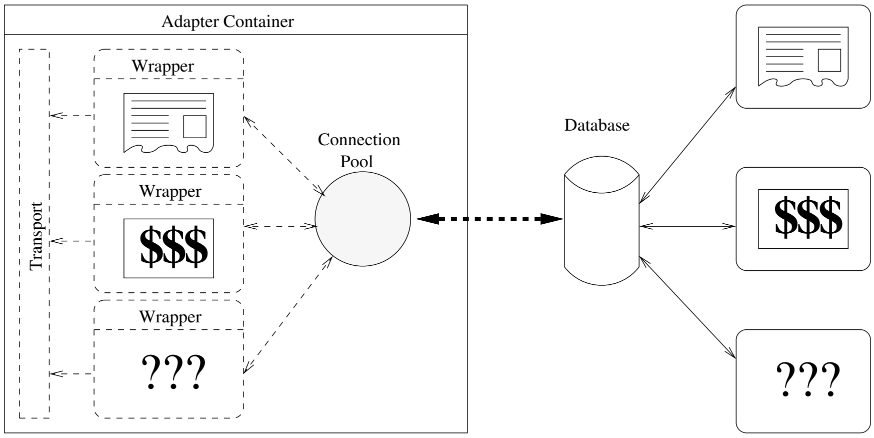

Figure 4.3: An example of resource sharing, by using a database connection pool, between three systems whose data are all stored in the same database

The container also sends the ontology to the MAS using the plug-in transport handler. To be able to support as many different ontology languages as possible, the framework assumes that the ontology is in a text file. The text file can be in any format, since the framework treats its contents as data. It only reads the file, and reads a property to identify the ontology language in use, and then put the language identifier in the message header. The ontology language is handled specially to allow different agents in the MAS to handle different ontology languages without having to parse the entire message.

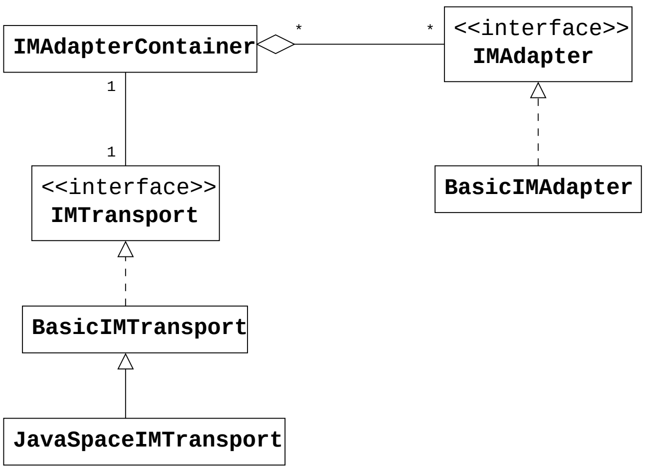

Figure 4.4: UML diagram of the core classes and interfaces in the framework

Section 5

Adapter implementation

In this section a description of how the two information sources, IntraWise, and Spectral Projektverktyg, were integrated in the IM system using the proposed framework. First the structure of the metadata is presented, then the ontologies used are discussed. The concept of subscriptions is thereafter introduced, and finally, implementation notes for the two implemented information sources are presented.

Only one view per adapter has been implemented.

5.1 XML

XML is the choice of syntax for the application protocol in the IM system. By choosing XML, the system will be relatively easy to extend with new functionality without altering much code, since no structural changes will be necessary. When new functionality is needed, a new handler class can be implemented and added to the system. Another advantage is that all messages can be processed in the same way and use the same software packages and components. The choice of XML also makes it easier to import and export information to and from the system, since the information is in a format that can easily be parsed by widely available tools.

5.1.1 Metadata elements

The information sources send out a flow of XML messages to the agent system via the adapters, and the metadata database. These XML messages from the information sources are all formatted alike, no matter from which information source they originate. They consist of two levels of XML tags, i.e. a root node and a number of child nodes. The root node, or tag, indicates what kind of metadata element this message contains, and the child nodes contain specific data for the metadata element. A typical metadata element is a news item, and an example of one is shown in Figure 5.1.

The root node has an attribute called *id* that is the context specific identifier of this metadata element, i.e. the identifier that, combined with the root node name, identifies this piece of data in the information source. The child nodes contain different meta-information depending on the type of the metadata element. For a news item this would typically be the title, any subtitle, and the author, plus other metadata.

Implicit metadata elements

Since, for instance, the author of a news item of a web page can be identified by something other than their name, namely their username, these types of attributes are called implicit metadata elements. Although one can find a news item by its title, two news items could have the same title. However, two users cannot have the same username. To signal that the child node is, or points to, an implicit metadata element, the child tag has an *id* attribute just like the root node. Child

```
<?xml version="1.0"?>
<newsitem id="33">
   <uri>http://intranet.acme.se/servlet/CollectionServlet?page_id=43</uri>
   <pageitem id="43">Acme News</pageitem>
   <title>New CIO appointed</title>
   <subtitle>John Doe has joined Acme</subtitle>
   <author id="hr">Human Resources Dept</author>
   <editedby id="hr">Human Resources Dept</editedby>
   <created>2001-01-22 9:21:23</created>
   <changed>2001-01-22 9:21:23</changed>
   <fromdate>2001-01-22 9:21:23</fromdate>
   <todate>2001-01-29 18:00:00</todate>
</newsitem>
```

Figure 5.1: An example of the structure of a news item.

nodes without the *id* attribute are considered normal metadata. Implicit metadata elements are useful for reasoning, and to provide choices to a user interface. When reasoning, two metadata elements with the same *id*, can be identified as the same. A user interface could retrieve all implicit metadata elements and present them in a checkbox, as described below when discussing subscriptions.

5.2 Ontologies in the IM system

In the IM system ontologies are expressed in XML to minimize the number of different API's in use in the system. Other alternatives were considered, but when also considering the availability of free powerful tools, as well as its extensibility, XML was chosen.

The MAS currently implemented in the IM system is relatively limited. It is currently made up of reactive agents that do not do any reasoning, and thus do not really use pure ontologies at all. The resulting ontology handling in the IM system is targeted at reactive agents, and not at reasoning agents. Therefore, no work has been put into handling what would be considered a pure ontology, i.e. concept terms and relationships between terms. The ontology language instead consists of axioms to be used by reactive agents. The mechanism for transporting, and handling, of ontologies is implemented, but it only supports ontology axioms. This unfortunate situation will be addressed later on in the IM project.

5.2.1 Subscriptions

An extension to the reactive agent paradigm that was introduced, is the concept of subscriptions. Users of the IM system should be able to subscribe to certain types of information. What kind of information they should be able to subscribe to is regulated in the ontology as system specific axioms. In order to have fine grain control over the subscriptions, metadata filters are used. The subscribable metadata "objects", or entities, can be retrieved and shown in a user interface as well as the possible filters the user can choose. By dictating what kind of metadata is subscribable, the complexity of the subscription handling system is reduced.

An example of the added subscription axioms is shown in Figure 5.2. Here the subscribable elements are news items and pages. The *byId*-attribute indicates whether it is possible to subscribe to changes to an entity with a particular *id*, which is not useful for news items that are short-lived, and rarely get changed.

The implicit metadata elements described above, can be used by the user interface by providing a set of choices. When specifying a subscription, if the user chooses to subscribe to news items, she can select to subscribe to news items written by a specific author. The author candidates can be fetched by the user interface from the IM system, just like any other metadata.

The user could, for instance, subscribe to all news items coming into the IM system that are written by a named author. Another possibility is to subscribe to all information regarding a

```
<ontology>
   <subscribable name="newsitem" desc="IntraWise news items" byId="false">
      <filter name="editedby" desc="The author" />
   </subscribable>
   <subscribable name="pageitem" desc="IntraWise pages" byId="true">
      <filter name="editedby" desc="The author" />
   </subscribable>
</ontology>
```

Figure 5.2: An example of the subscribable axiom from the IntraWise adapter.

specific project, including web pages about the project, and posts on the company message boards. Depending on the sophistication of the subscription handler, various subscription features, such as free-text matching in metadata, can be specified. The current subscription handler in the IM system only handles simple AND-type filters, without free-text matching. An example of a subscription is: "give me all news items where the author is Niklas Eklund AND the news item concerns the IM project". The user interface could help the user by showing a listbox with all known authors, and another with all known projects.

5.3 Adapter implementation

The adapters for IntraWise and Spectral Projektverktyg both inherit from the *BasicIMAdapter* class. They differ somewhat in the implementation, since IntraWise has API support for events, and Spectral Projektverktyg has not. Both systems contain news items, and the tags for them are the same.

The means of transport is JavaSpaces, logically one for the ontology, and one for the metadata, but they could physically be the same. The metadata messages being sent has the form seen in Figure 5.3. There are two fields that help identify the sender of this message, the sender's type, and the sender's identifier string. The combination of the two should uniquely identify a sender. The message also contains a pointer to the actual data, that this message "represents", and one field that can be used to display this message, or rather the modified data. The *dataType*, and *dataTypeId* fields are used for identifying the contents of the message, without having to parse the XML payload. These fields contain the root node name, and *id* attribute value, of the metadata payload XML.

```
import net.jini.core.entry.Entry;

public class DataMessage implements Entry {

    public String senderType;     // The type of the sender of this message
    public String senderIdent;    // The identifier of the sender of this message
    public String dataURI;        // URI to the payload in this message
    public String dataType;       // The type of the root element in the payload xml
    public String dataTypeId;     // The id attribute to the root element
    public String dataTypeValue;  // The display name of the type-id item
    public String dataPayload;    // The metadata payload
}
```

Figure 5.3: The DataMessage class that is used to send metadata to the MAS.

The ontology messages are only sent on adapter start-up in the current implementation. It would, however, be easy to add a check on the file containing the ontology, and re-send the ontology message if the file changes. The ontology message is similar to the metadata message, and its form is shown in Figure 5.4. Like the *DataMessage*, the *OntologyMessage* contains two fields for identifying the sender. It also contains one field to indicate which ontology language the ontology payload is in.

```
import net.jini.core.entry.Entry;

public class OntologyMessage implements Entry {

    public String senderType;       // The type of the sender of this message
    public String senderIdent;      // The identifier of the sender of this message
    public String ontologyLanguage; // The ontology language used in the payload
    public String ontologyPayload;  // The ontology payload

}
```

Figure 5.4: The *OntologyMessage* class that is used to send ontologies to the MAS.

A lot of interface code has to be implemented in order to start using Jini, since the current Jini development kit provided by Sun is not yet mature, and is still quite low-level. To avoid having to write additional code, and to get started more quickly, the Service Writer's Toolkit (SWT), created by K. Edwards, was used. The SWT provides, among other services, the necessary utility classes to handle Jini leases, and exporting of Java code to clients. More information about the Service Writer's Toolkit can be found in Appendix A

The adapters were implemented by using SWT, and are fully-fledged Jini services. That is not a necessity, but to later be able to introduce a callback interface, it is probably simpler to have a proper Jini service that can be looked up.

5.4 The IntraWise adapter

IntraWise stores its information in a normal relational database. There is support in IntraWise for event notification when pages are created, altered, or removed. The support also extends to categories. This notification scheme is used to inform the included 3rd party search engine so it can update its indexes. The notification API was used in the implementation of the IntraWise information source adapter, together with the polling technique.

Only a small set of the meta-information contained in IntraWise is currently used in this implementation. The selected metadata elements are pages and news items. Implicit metadata elements consist of users and categories.

The subscribable entities are pages, and news items, with author as the single implemented subscription filter.

5.5 The Spectral Projektverktyg adapter

There is no support for notifications in Spectral Projektverktyg, as there is in IntraWise. That means that this information source must be polled to discover when information had been changed. This is a straightforward implementation of a database-polling adapter that could be used as a model for other similar adapters. The keys to the interesting tables are cached, as well as timestamps. Differences in the table rows are detected by checking timestamps, and new and deleted rows are detected by checking the keys.

The implementation only covers a limited set of the information extraction possibilities available in Spectral Projektverktyg. The information entities extracted are projects, news items, project milestones, project membership and document uploads. The only implicit metadata element is users.

The subscribable entities are projects, news items and project milestones. The current subscription filters are author and project for news items, project leader for projects, and project for project milestones.

Section 6

Discussion

6.1 Conclusions

Although the needs of individual multi-agent systems are different, the proposed framework can be used in a variety of applications. The only major limitation preventing the framework to be used in basically all information handling multi-agent systems, is the lack of a query callback interface. By having a caching database in the MAS, or between the MAS and the adapter layer, that requirement is not necessary. The caching database could then be queried in place of the information sources.

By using pluggable adapters, it is easy to integrate a new information source into a framework-based system. It requires only the implementation of an additional adapter to extract any information desired from the information source, and to provide an ontology explaining the structure of the extracted information.

The means of transport to the MAS is very flexible, since the transport handler class is pluggable, similar to the way adapters are pluggable. If the MAS is using a different transport medium, for which there is no previous support, only a new transport handler class needs to be implemented. No modification is necessary to the adapters, or the adapter containers.

Information security is conspicuously left out of all the described related systems, or it at least takes a very small part. That is remarkable, since some of the systems target commercial environments, which inevitably will demand some kind of security. Support for security in the IM system was planned from the beginning, but was left out of the framework initially. Security crosses module boundaries, and would have introduced too much complexity if included in the thesis work.

The framework has successfully been used when implementing the IM system, including its initial adapters for IntraWise and Spectral Projektverktyg. It has proven itself well suited for the current type of multi-agent system used in the IM system, and is designed to be equally well suited for the originally planned, more feature rich, multi-agent system.

6.2 Future work

As indicated above, the most important addition would be a callback interface that could support permission checking, queries, and possibly information updating. The current implementation of the base framework could be made more robust, to handle communication failure more gracefully.

By adding permission checking the adapter could verify that a user is authorized to view a piece of information coming from the adapter. The adapter would typically use any security manager/handler for the handled system that might be available, or in some other way determine the permissions.

If the MAS is allowed to query the information sources via the adapters, a query language that is suitable for a variety of information, and information sources, must be incorporated. In order to simplify implementing new adapters, the design should use the same pluggable paradigm used for the transport protocol.

Updating of the information would probably be the hardest thing to support in a general manner. Type checking must be enforced to validate incoming data, the error handling must be uniform, and new interfaces for updating must be created. In addition to type checking, information security must also be enforced. To be able to support updating, the entire framework might have to be slightly redesigned.

The adapter ontology axioms could extend to include user interface commands, and presentation hints. The user interface could use the extra information to better present the data. The ontology handling would become more complicated, but since subscription and user interface axioms are not strictly ontology related, they could be sent and handled separately.

Bibliography

[Ambite and Knoblock, 95] J. Ambite and C. A. Knoblock. Reconciling distributed information sources. In Working Notes of the AAAI Spring Symposium on Information Gathering in Distributed Heterogeneous Environments, Palo Alto, CA, 1995.

[Arens and Knoblock, 94] Y. Arens and C. A. Knoblock. Intelligent Caching: Selecting, Representing, and Reusing Data in an Information Server. Proc. ACM CIKM , 1994.

[Arens et al., 93] Y. Arens, C. Chee, C. Hsu and C. Knoblock. Retrieving and Integrating Data from Multiple Information Sources. In Journal of Intelligent and Cooperative Information Systems, Vol. 2, June 1993.

[Bayardo et al., 97] R. J. Bayardo Jr., W. Bohrer, and et. al., InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments, ACM SIGMOD, 1997.

[DAML] DAML, http://www.daml.org

[FIPA ACL] FIPA ACL, http://www.fipa.org/repository/aclspecs.html

[Freeman et al., 99] E. Freeman, S. Hupfer, and K. Arnold. JavaSpaces(TM) Principles, Patterns and Practice (The Jini(TM) Technology Series). Addison-Wesley Pub Co; ISBN: 0201309556, 1999.

[Hee Hwang, 99] C. Hee Hwang. Incompletely and Imprecisely Speaking : Using Dynamic Ontologies for Representing and Retrieving Information. Microelectronics & Computer Technology Corporation, 1999.

[H. S. Nwana, 96] H. S. Nwana. Software Agents: An Overview. Knowledge Engineering Review, Vol. 11, No 3, pp.1-40, Sept 1996. © Cambridge University Press, 1996.

[InfoZone] InfoZone, http://www.infozone-group.org

[JavaSpaces] JavaSpaces, http://java.sun.com/products/javaspaces

[Jennings et al., 98] N. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. Journal of Autonomous Agents and Multi-Agent Systems, 1:275–306, 1998.

[Jini] Jini, http://www.java.sun.com/products/jini

[Kalen, 00] M. Kalén. A Jini-based Multi-Agent System, Master's Thesis at KTH, Stockholm, Sweden. Spectral AB, 2000.

[Kashyap and Sheth, 96] V. Kashayp and A. Sheth. Semantic Heterogeneity in Global Information System: The Role of Metadata, Context and Ontologies. Cooperative Information Systems: Current Trends and Directions, M. P. Papazoglou, Schlageter, G., Eds. London: Academic Press, 1996.

[KQML] KQML, http://www.cs.umbc.edu/kqml

[Labrou et al., 99] Y. Labrou, T. Finin, and Y. Peng. The current landscape of Agent Communication Languages. IEEE Computer Society, Intelligent Systems, volume 14, number 2, March/April 1999.

[Levy, 98] A. Y. Levy. Combining Artificial Intelligence and Databases for Data Integration. University of Washington, 1998.

[Linda Systems] Linda Systems, http://www.cs.yale.edu/Linda/linda.html

[Mappa] Mappa, http://www.sics.se/mappa

[Martin et al., 98] P. Martin, W. Powley, and A. Weston. Using Metadata to Query Passive Data Sources. Proceeedings of the 31st Hawaii International Conference on System Sciences, 1998.

[Mattson and Noren, 01] S. Mattson and M. Norén. An extensible communication protocol for a mult-agent system (tentative title), Master's Thesis in preparation. Spectral AB, 2001.

[Nodine et al, 98] M. Nodine, J. Fowler, and B. Perry. An overview of active information gathering in InfoSleuth. MCC Technical Report, 1998. Submitted to Autonomous Agents '99.

[OIL] OIL, http://www.ontoknowledge.org/oil

[RDF] RDF, http://www.w3.org/RDF

[Shah and Sheth, 98] K. Shah and A. Sheth. Logical Information Modeling of Web-accessible Heterogeneous Digital Assets. Proc. of the Forum on Research and Technology Advances in Digital Libraries (ADL'98), Santa Barbara, CA, 1998.

[Stone and Veloso, 96] P. Stone and M. M. Veloso. Multiagent systems: A survey from a machine learning persective. Technical Report CMU-CS-97-193, Carnegie Mellon University, Pittsburgh, PA, 1996.

[Topic Maps] Topic Maps, http://www.topicmaps.org

[Unruh et al., 98] A. Unruh, G. Martin, and B. Perry. Getting only what you want: Data mining and event detection using InfoSleuth agents. Technical Report MCC-INSL-113-98, MCC InfoSleuth Project, 1998. Submitted to Autonomous Agents '99. MCC-INSL-114-98.

[XLink] XLink, http://www.w3.org/TR/xlink

[XML] XML, http://www.xml.org

All web resources listed above were found working as of mid May 2001.

Appendix A

The Service Writer's Toolkit

The Service Writer's Toolkit is a set of utility Java classes, written by K. Edwards, which helps the development of Jini services. Some of the necessary tasks involved when creating Jini services are repetitive, and are not really supported yet by the utility classes from Sun. Therefore the SWT is a great help to get started with Jini services. Some of the features are as follows (taken from the author's web page):

- A "recoverable" service wrapper that can checkpoint itself to persistent storage, and recover after a crash or after being shut down.
- A pluggable framework for persistence. You can use this framework to save service state to a file system, database, or other stable storage medium.
- Optional facilities for class file exporting. You can have the service wrapper instantiate a "micro" web server that will export class files to clients from a number of sources. The export framework will take care of setting codebase appropriately.
- A full implementation of the common Jini service administration interfaces *JoinAdmin*, *DestroyAdmin*, and *StorageLocationAdmin*.
- A Landlord implementation designed to work with Sun's LandlordLease implementation. This code uses RMI to communicate with client-side leases and can manage leases on event registrations or other allocated resources.

A short, and somewhat simplified, description of the steps involved when developing a service is:

1. Create a service class that inherits from the *BasicUnicastService* class, and adjust initial parameters
2. Create an interface, together with an implementing class to act as the service's proxy
3. Start the service

Depending on how complicated the service is, this might be all the steps needed. The constructed service will export any needed code, handle leases and registrations, and be fully administrable via Sun's service browser. When restarted, the service will also remember the groups it was registered with, and then re-register with those groups.

For more information about the Service Writer's Toolkit, see the author's web page, which is located at http://www.kedwards.com/jini.