WP5: "Knowledge Networks"

Deliverable D5.2: Extended Beta Release Software for Knowledge Networks

Status and Version:	Version 4, Final	
Date of issue:	27th of June 2007	
Distribution:	Project Deliverable R+P	
Author(s):	Name	Partner
	Matthias Baumgarten	UU
	Kieran Greer	UU
	Franco Zambonelli	UNIMORE
	Rico Kusber	UniK
	Nicola Bicocchi	UNIMORE
Partners	Unimore, UU, UNIK	
Checked by:	Kevin Curran	UU
	Franco Zambonelli	UNIMORE

Table of Contents

1.1 Purpose and Scope 4
1.2 Reference Documents 4
1.3 Acronyms & Definitions 5
1.4 Document History 6
1.5 Document overview 6
2 Knowledge Networks: Status of Things 7
3 Beta Prototype Overview 8
3.1 Execution Space 8
3.2 Distributed Communication Software 10
4 Structural Components 11
4.1 Service Architecture 11
4.1.1 Example Services 15
4.2 Knowledge Atom 18
4.2.1 ACE Atoms 18
4.2.2 Sensor Atoms - micaZ 18
4.3 Knowledge Container 22
4.3.1 Network Based Organisation 22
4.3.2 Concept Based Organisation 24
5 Knowledge Execution 27
5.1 Knowledge Querying 27
5.1.1 Query Process 27
5.1.2 Autonomic Querying 29
5.1.3 The Query Language 30
5.1.4 Ontology-Based Search 30
5.1.5 KN Query Interface 32
5.1.6 Summary 33
5.2 Knowledge Verification 33
6 Integration with the ACE Framework 35
6.1 ACE Integration Process 35

6.2 Knowledge Network Interface 36
6.3 Realising Knowledge Network via ACEs. 37
6.4 Implementing Knowledge Atoms 38
6.5 Implementing Knowledge Containers 39
7 Research Directions 40
Appendix A – User Guide 42
A.1 Knowledge Atom 43
A.2 Embedded Knowledge Atom 43
A.3 Remote Knowledge Atom 44
A.4 Atom Registration 47
A.5 Atom Add-On Services 47
Appendix B – Examples 49
B.1 Embedded Atom 49
B.2 RPC Atom and Add-On Service 50
B.3 Skeleton Class for Push Mechanism 51
Appendix C – Class Structures 52
C.1 Atom 52
C.2 KN Service 53
C.3 Overall Framework 54

Introduction

1.1 Purpose and Scope

This document represents the M16 deliverable for the CASCADAS WP5 "Knowledge Networks". Firstly, it summarizes on some aspects of knowledge networks and refines individual aspects of the initial specification. Secondly, it provides a description of relevant components as developed for the beta release of the knowledge network toolkit. Finally, it elaborates on aspects related to knowledge execution before outlining relevant research directions for the remainder of the project.

This deliverable is also accompanied by a software package containing individual algorithms and packages that reflect the first beta release of the KN toolkit.

1.2 Reference Documents

[D5.1] Deliverable D5.1: Knowledge Networks Specifications, Mechanisms, and Alpha Software Release

[XMLRPC] http://ws.apache.org/xmlrpc/

[HTTPCore] http://jakarta.apache.org/httpcomponents/

[OTK-RQL] G. Karvounarakis, V. Christophides, D. Plexousakis, S. Alexaki, "Querying community web portals", Technical Report, Institute of Computer Science, FORTH, Heraklion, Greece, 2000, http://www.ics.forth.gr/proj/isst/RDF/RQL/rql.pdf.

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

[DAMLOIL] I. Horrocks, "DAML+OIL: A Reason-able Web Ontology Language, Advances in Database Technology", 8th International Conference on Extending Database Technology, Prague, Czech Republic, pp. 103-116, March 25-27, 2002. Proceedings, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, http://www.daml.org/

[SPARQL] http://www.w3.org/TR/rdf-sparql-query/

[Xcerpt] http://www.xcerpt.org/

[BicMZ07] N. Bicocchi, M. Mamei, F. Zambonelli, "Self-organized Spatial Regions for Sensor Network Infrastructures", 2007 IEEE Symposium on Pervasive and Ad-Hoc Communications, IEEE Press, May 2007

[CasMZ07] G. Castelli, M. Mamei, F. Zambonelli, "Engineering Contextual Knowledge for Pervasive Autonomic Services", submitted to the International Journal of Information and Software Technology, 2007.

[Bau07] M. Baumgarten, N. Bicocchi, R. Kusber, M. Mulvenna, F. Zambonelli, "Self-organizing Knowledge Networks for Pervasive Situation-aware Services", IEEE International Conference on Systems, Man, and Cybernetics, Toronto (CA), October 2007, to appear.

1.3 Acronyms & Definitions

Context:	In general terms, the context defines the "surrounding and interrelated conditions in which something exists" (Mirriam-Webster Dictionary). In CASCADAS, the context identifies the operational environment in which a service situates, which could include network, application, social, and physical context (Cfr. Knowledge).
Contextual Information:	Information related to some actual characteristics of the operational environment, i.e., to some facts occurring in it.
Context-awareness:	The capability of software (i.e., as far as CASCADAS is concerned, of services) of being aware of the context in which they are invoked and/or executed, and of adapting their behaviour accordingly.
Concept of Interest:	A computational model of any real world object or event (including services and processes).
Ontology:	A formal specification detailing how to express concepts of interest in a specific area. In CASCADAS, a shared ontology is expected to be defined for ACE's so as to enable them to properly represented in a semantic and inter-operable way all needing contextual information.
Knowledge:	Contextual information as it can be made available to some actors (i.e., ACE's) to make them aware of some facts and reason about them. In CASCADAS, we account for: network knowledge, representing facts about the current configuration of the physical network and of the related devices; application (or ACE-level) knowledge, representing facts about the current status of (some) ACE's; social knowledge, representing facts about the human actors currently exploiting the network and its ACE-based services, and the social context in which they are doing so; physical knowledge, representing facts about the physical world. We emphasize that the difference between Contextual Information and Knowledge is really subtle, and mostly related to the observation viewpoint: the context generates contextual information which then becomes something that the agent knows, i.e., knowledge.
Knowledge Atom KA:	A knowledge atom is a generic access concept that describes, references and provides access to a single data entity of any type, complexity, size, location etc. For example, one can imagine that the information related to the current physical location of a person can be a knowledge atom reporting the name of that person, its location in terms of latitude and longitude, and possibly some information related to the activities currently undertaken by that person.
Situation:	In general terms, a situation defines a "relative position or combination of circumstances at a certain moment" (Mirriam-Webster Dictionary). Accordingly, in CASCADAS, a situation is considered as "something

	that is happening in the context" and, for generalization, also something that "is likely to occur at a certain moment in the future".
Situation-awareness:	In general terms, situation-awareness relates to the capability of being aware and of adapting behaviour to situations other than to context (Cfr. Context-awareness). While components and services (i.e., ACE's) are situated in a context and can perceive contextual information in the form of knowledge atoms to become context-aware, perceiving situations (present and future) and becoming situation-aware implies a higher degree of understanding. In particular, it requires properly acquiring all the needed knowledge about "combinations of circumstances".
Knowledge Network KN:	A network of knowledge is an ontology-based structured collection of knowledge atoms, describing specific situations, and built in order to facilitate ACE's in acquiring high degrees of situation-awareness in an efficient way. This is not to be confused with "network knowledge", intended as the information available about the status of a network.
Knowledge Container KC:	As it will appear clearer in the remainder of this document, the structuring of knowledge atoms in networks may also imply the need to create higher-level structures aggregating existing knowledge atoms into a component aggregating a set of related knowledge atoms or other knowledge containers into a composite.

1.4 Document History

Version	Date	Authors	Comment
1.00	01/03/2007	MB	Document Created
2.00	15/05/2007	WP5 Partners	ToC finalized
3.00	25/06/2007	MB	First Complete Draft integrating all contributions by partners
4.00	26/06/2007	FZ	Corrections and minor restructurings

1.5 Document overview

The document is structured as follows. Section 2 summarizes the organization of the development process and its current status. Section 3 overviews the architecture of the knowledge network software. Section 4 provides a detailed description of individual components and their usage within the developed prototype. Section 5 elaborates on aspects that are related to knowledge execution and in particular to knowledge querying and knowledge verification. Section 6 discussed the plan of action for the integration with the ACE framework. Section 7 provides the research scope for the remainder of the project. Finally the Appendix sections supplements this document with the user guide on how to build specific atom realisations as well as other relevant material.

2 Knowledge Networks: Status of Things

As specified in [D5.1], the high level goal of knowledge networks can be summarized as the provision of a vehicle capable of creating, storing, propagating and discovering information in a light-weight, scale free and multi-view environment. In particular the organisation and the provisioning of knowledge at different levels of granularity is of particular importance as it allows pre-organisation of available knowledge based on *(a)* dynamic models that are derived via self-contextualisation of the knowledge providers (bottom-up organisation) and *(b)* model-based organisation where distinct granular levels are introduced by the services and applications that use the knowledge network (top-down organisation). In any case, the knowledge network must be able to self-organise itself in the sense that it autonomically monitors available context within the virtual space it is operating in and provides the required context and any other necessary knowledge and operational support to the requested services, and self-adapts when context changes.

From an evolutionary perspective, three distinct stages have been identified to be relevant for the construction and effective usage of knowledge networks (see Figure 1). Firstly, structural requirements that provide necessary components capable of holding knowledge at different levels of granularity including the implementation of a highly flexible framework capable of linking individual knowledge components or any group thereof into distinct purpose-build sub-networks. Secondly, behavioural requirements which deal with more dynamic aspects of knowledge networks such as self-organization, self-optimisation, self-adaptation and self-configuration activities. Thirdly, predictive requirements enabling detailed analytics of individual knowledge components in order to derive new, useful and understandable knowledge.

The overall research development process is accordingly organized along three stages, which also reflect the three tiered duration of the project.

During the first year of the project relevant conceptual components have been defined and the overall structural framework has been drawn up, which has lead to the alpha release of knowledge network software. Self-organized knowledge aggregation algorithms have been studied and experienced too during the first year, but not integrated in the alpha version of the software.

The research within the second year has and will largely concentrate on aspects that support autonomic principles thus promoting concepts that support self-organisations, self-adaptation, context verification as well as the utilisation of stigmergic principles in support for flexible query optimisation as well as initial self reasoning mechanism. From the software viewpoint, this has lead so far (i.e., by the half of second year) to the beta release of knowledge network software, which extend the alpha one with a refined and more flexible structural management, flexible querying mechanisms, and with the integration of some of the studied algorithms for both network-based and concept-based self-organized knowledge aggregation. WP5 partners will utilise the second half of this year to *(a)* refine and further develop the concepts proposed so far, *(b)* initiate additional research directions depending on upcoming results and requirements, *(c)* integrate the results of WP5 into the overall framework of the project and in particular into the ACE framework and *(c)* evaluate the runtime capabilities of the resulting KN framework.

The final year will then concentrate on activities that include self-reasoning over the content of the network and / or its usage, more sophisticated and situation specific context orchestration as well as propagation. In addition flexible context verification mechanisms will be introduced to guarantee a certain quality of context to individual knowledge consumers.

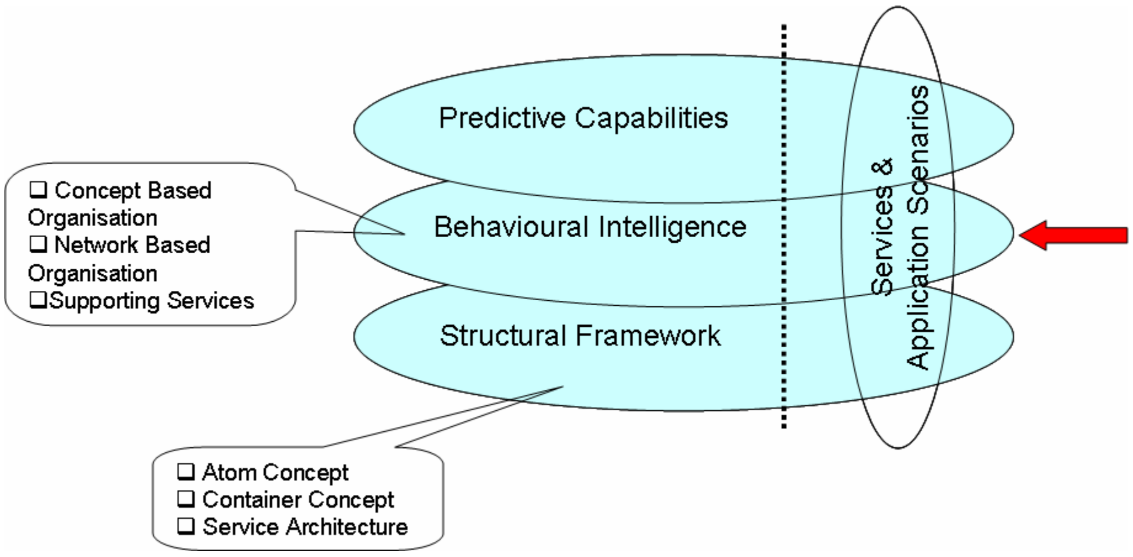


Figure 1: Conceptual Components of Knowledge Networks.

3 Beta Prototype Overview

The beta prototype consists of a number of packages that are partially integrated into a single RPC based execution environment. Note, that one of the objectives for the remainder of this year will be to replace this temporary execution environment with an ACE based solution. Later in this document we describe how we plan to realize such porting.

3.1 Execution Space

The basic architecture is to have a number of “knowledge servers” running on any number of computational resources, which is depicted in Figure 2. Simplified each server may host any number of sub-networks which internally as well as remotely communicate via RPC. Client access as well as data provisioning, via knowledge atoms, is also facilitated via RPC.

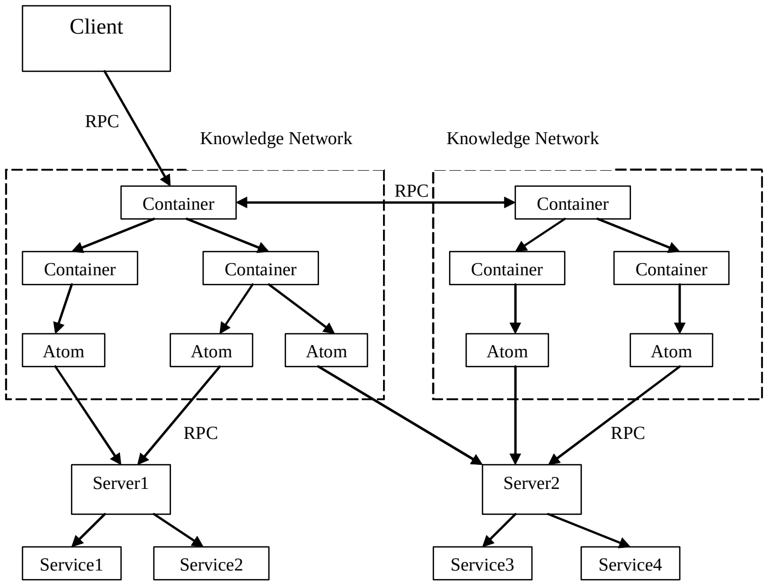


Figure 2: Basic test environment architecture

Although each computer may comprise a number of networks it should only host one server, which provides a single access point to all nested components. The server then stores a base component to which the root nodes of the network are added. For our tests there was just one root node of the Knowledge Network type. The root node then stores the network components of the network (nodes, sources or services). All components are stored in the root network component as services, as well as being hierarchically organised through references. To call a remote component, you specify the URL of the remote server, access the network and then access the individual component. Services can be nested however and so can be called recursively through the root service. For example, to call a sub-service service in an atom one would need to specify the atom engine first to access the specific service it wants to address. Each remote server needs to know the addresses of the other servers in the network. A client then only needs access to one server to access the whole KN. For a user it is indifferent which server is initially addressed as each server will effectively have access to all other servers as well. Nonetheless, the concept of private networks may be facilitated by preventing the registration of individual servers into the scope of a global knowledge network.

This concept is visualised in Figure 3 where a number of knowledge sources, simulated or not, register into the scope of individual knowledge servers (hosts) which are linked together via RPC. Individual applications may now access either one of the hosts available in order to query for knowledge. Although hidden from the user the underlying KN may query the whole available knowledge space or any part thereof to answer a user's query.

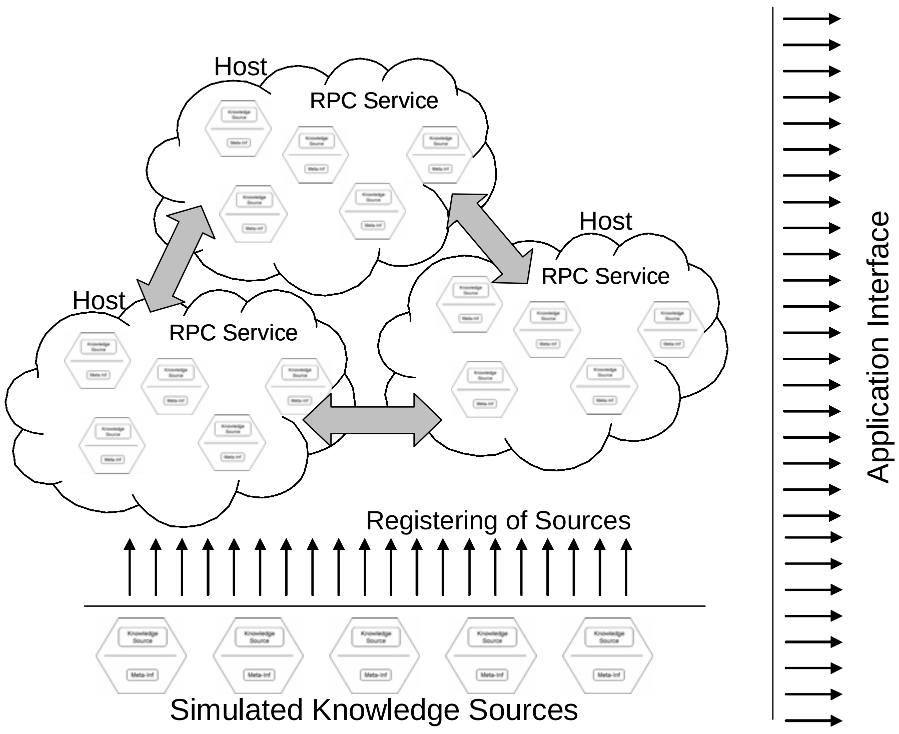


Figure 3: Knowledge Network Realisation

3.2 Distributed Communication Software

In order to facilitate flexible, lightweight and extendable distributed communication among KN objects two distinct packages have been explored. Firstly the XML-RPC based communication package as provided by Apache (http://ws.apache.org/xmlrpc/) and secondly an internal communication package based on HttpCore (http://jakarta.apache.org/httpcomponents/).

In a nutshell, both approaches provide a flexible mechanism to access individual components and their methods directly via an RPC interface. Thus, each knowledge network component can be seen as a distinct resource that is accessible via a unique URI. This promotes one of the key objectives of the Cascadas project. That is to realise knowledge networks with independent and light weight components that can be linked together in a distributed environment. Furthermore, it allows for the dynamic extension of individual components in a way that specific services are added / removed at runtime.

XML-RPC is a popular protocol that uses XML over HTTP to implement remote procedure calls. Currently in version 3 it has full XML support and also allows for vendor specific extensions. In addition, it has some limitations which limit the flexibility during integration. Probably the most awkward thereof is that registered RPC handlers are not object safe which means that the handling of local or session based data is difficult. Finally, although being a lightweight mechanism, it still comprises functionality that is not required by KN's thus a specialized sub version should be generated to allow for very small and lightweight implementations. As a consequence, an internal communication mechanism has been explored that has been specifically tailored towards the needs of knowledge networks Also based on a Remote Procedure Calling mechanism it enables components to call each other's methods in a distributed as well as localised fashion. Named LICAS (Lightweight Internet-based Communication for Autonomic Services) it has been designed to be as small as possible yet as powerful as necessary. All communication uses Java Reflection to dynamically call another component's methods while internal processing is based on XML. LICAS already has support for a number of different data types including XML and serialisation. Vendor specific extensions are also possible to support more specific data types via individual parsers. They must be able to serialize the object into an XML format or parse from XML back to the object. Alternatively, the user can implement the Serializable interface, when the object will be serialized and then passed as part of the XML message. The final transport mechanism is as a String, when the XML is automatically converted into and back from a String. A standard class is used to specify the method to call, which can be on a local or remote object. A full method description is required, when Reflection will then match this to the called object's methods. The parameters are entered simply as objects and the package will automatically serialize and parse them based on the parsers stored. Other parameters that may be included are passwords and communication IDs. Basic security services such as password protection or communication ID are also provided for. This package is based on the Apache Jakarta HttpCore package and the JDOM parser, though it should also be possible to parse XML objects of the Java DOM type as well. The system uses JDOM internally, so it would be recommended to use this for parsing other xml objects as well. While this package is currently under internal evaluation, it may be used at a later stage to facilitate remote communication among KN components

4 Structural Components

4.1 Service Architecture

The knowledge network envisioned can be seen as an advanced, fully distributed, and dynamic knowledge provisioning system that serves knowledge or, to be more precise, access to knowledge based on specific user or application requests. The main principle employed to keep the internal structures as lightweight as possible is that neither knowledge nor its sources are duplicated and as such the underlying data are always available wherever they are produced. Utilising this concept not only results in simplified organisational structures inside the KN but also ensures that knowledge, when accessed, is fully up to date. However, from a knowledge utilisation point of view some additional functionality are necessary in order to optimise some important factors such as processing power, network traffic generated when using the KN or finally, but also very important, usability of the KN framework. Some examples of such required functionality include the possibility of notifying other applications if knowledge that is of interest to them has recently changed (push mechanism) or a dedicated history component that provides a history buffer to other services.

Obviously, one possibility is to outsource this mechanism to individual applications themselves. As such, each application needs to constantly query the knowledge source it is interested in, comparing its current state with the previous one. Although beneficial for some applications, this mechanism has several drawbacks. For example:

- Multiple applications need to observe the same value, which inevitably requires higher processing power to serve multiple requests.
- A single application needs to observe multiple values (often to detect that values have not changed yet), which wastes processing power at the application side.
- Knowledge sources are remote and as this increases network traffic.
- Knowledge to be compared may be large, thus increasing network traffic, wasted processing power and delaying response time.

Considering the above limitations renders an outsourcing of some services such as the push functionality as inefficient and actually, in most cases, counter-productive with respect to the optimum use network and computational resources.

Another more effective mechanism is to connect a dedicated service with the knowledge source under observation and as such embed the functionality required directly into the service structure of the KN. For instance, if an application needs to be notified when a knowledge source changes its value it could simply register itself with the push service of the related knowledge source component. Thus, the knowledge source component periodically monitors its own values and notifies all registered listeners if a change has occurred. This concept is more efficient due to the following reasons:

- Applications may (de-)register dynamically via a single standardised interface.
- Applications will only be notified if new / updated knowledge is available and then can decide how to deal with it.
- Monitoring is performed directly at the knowledge source, thus no unnecessary network traffic occurs.
- Minimum processing overhead at the knowledge source component because individual compare procedures are only evaluated once per polling cycle and not once for every request.

Finally, realising this mechanism as a KN service promotes dynamic orchestration of components and services depending on individual circumstances which is another goal of the overall KN framework.

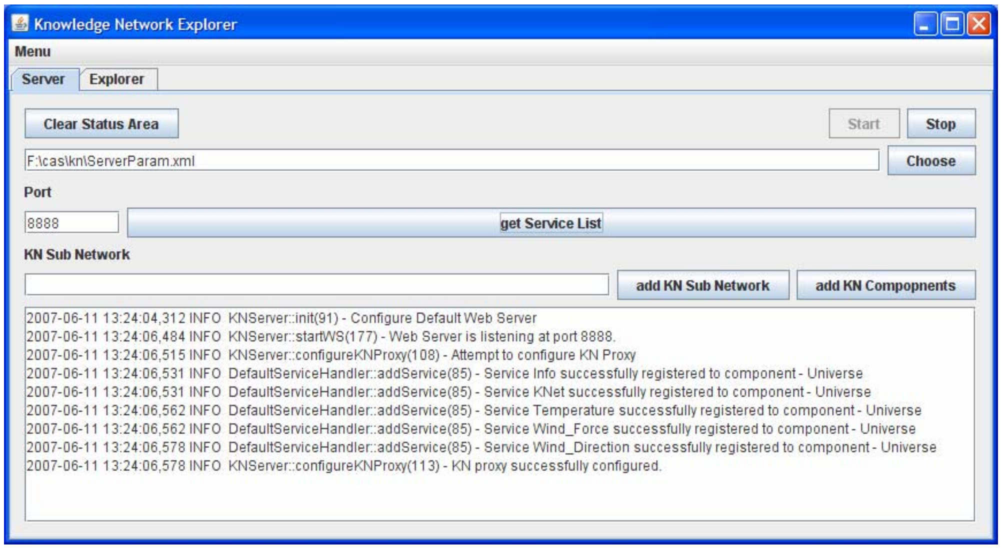
The current XML-RPC based architecture allows hosting a knowledge server within a JVM. Having a specific servicehandler incorporated it allows for other services to be added or removed at initialisation or at runtime. Furthermore, as long as services are visible by the standardised Java URILoader class they may be loaded locally or remotely. Individual methods of added services may be explored or executed at runtime via reflection utilising the build in execution handler. This allows for maximum flexibility as new functionality becomes instantly available without the need for registering any components.

```
- <Services>
- <!--
Extra services the component can provided as added components
  -->
- <Service>
  <Name>The name of a service this component provides as an add-on</Name>
  <Description>Sementic description of the service</Description>
  <URI>The address of the service, can be null for a local service</URI>
  <ClassName>The Java class name of the service object</ClassName>
- <Login>
  <User>The username to access if required</User>
  <Password>The password to access if required</Password>
  </Login>
- <Parameters>
- <!--
Intitialisation parameters of the service
  -->
- <Parameter>
  A single parameter for the method
  <Name>The parameter name</Name>
  <Type>The parameter type</Type>
  <Value>The parameter value</Value>
  </Parameter>
  </Parameters>
- <Methods>
- <!--
A list of methods for the service
  -->
- <Method>
- <!--
A single method specification
  -->
  <Name>The methods name</Name>
  <Description>Sementic description of the method</Description>
  <Return>The return type of the method</Return>
- <Parameters>
- <!--
A list of parameters for the method
  -->
- <Parameter>
  A single parameter for the method
  <Name>The parameter name</Name>
  <Type>The parameter type</Type>
  <Value>The parameter value</Value>
  </Parameter>
  </Parameters>
  </Method>
  </Methods>
  </Service>
  </Services>
```

Figure 4: Service Description

This proposed framework allows the fully generic construction of hierarchical based service structures where each component is by default a service itself and as such can incorporate other services. Thus services may be orchestrated via other services which in turn contain services themselves and so on. Obviously this could result in very complex service structures so that it should used with care rather than as a commodity.

Available and executable services may be described using the xml template shown above and may be added to any component that implements the serviceHandler interface using the respective addComponent Method. For more information on this the authors refer to the Java Documentation.



F:\cas\kn\ServerParam.xml

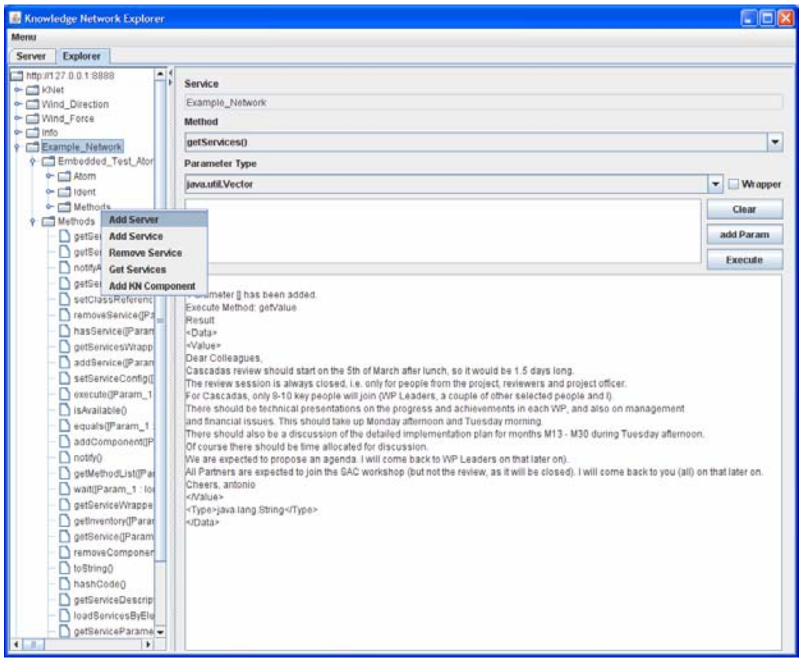
8888

```
2007-06-11 13:24:04,312 INFO  KNServer::init(91) - Configure Default Web Server
2007-06-11 13:24:06,484 INFO  KNServer::startWS(177) - Web Server is listening at port 8888.
2007-06-11 13:24:06,515 INFO  KNServer::configureKNProxy(108) - Attempt to configure KN Proxy
2007-06-11 13:24:06,531 INFO  DefaultServiceHandler::addService(85) - Service Info successfully registered to component - Universe
2007-06-11 13:24:06,531 INFO  DefaultServiceHandler::addService(85) - Service KNet successfully registered to component - Universe
2007-06-11 13:24:06,562 INFO  DefaultServiceHandler::addService(85) - Service Temperature successfully registered to component - Universe
2007-06-11 13:24:06,562 INFO  DefaultServiceHandler::addService(85) - Service Wind_Force successfully registered to component - Universe
2007-06-11 13:24:06,578 INFO  DefaultServiceHandler::addService(85) - Service Wind_Direction successfully registered to component - Universe
2007-06-11 13:24:06,578 INFO  KNServer::configureKNProxy(113) - KN proxy successfully configured.
```

Figure 5: Host Admin GUI

Figure 5 shows a simple admin GUI that allows configuring a host thus launching a dedicated web server providing the RPC interface to other servers and client applications. In addition, it allows adding sub networks as well as other services into the scope of the knowledge server. Individual actions are logged and if required presented to the user.

Once the URI of a server is known the knowledge network explorer may be used to explore, execute or modify the server's services. As visualised in Figure 6, any number of Servers can be added to the explorer (Add Server) and a list available services (get Services) as well as methods (Method node) can be retrieved. Once a service has been loaded into the explorer its methods can be executed utilising the right side of the explorer where a user can select the method to be called as well as relevant parameters. Once a method is executed the corresponding service is identified and remotely executed. Currently, the response is then displayed within the explorer. In the example below the get value method has been called on the atom service of "Embedded_Test_Atom" which in turn is a specific services of "Example_Network" which is itself a service running on host "http://127.0.0.1:888.



Service: Example_Network
Method: getServices()
Parameter Type: java.util.Vector

```
Parameter [] has been added.
Execute Method: getValue
Result
<Data>
<Value>
Dear Colleagues,
Cascadas review should start on the 5th of March after lunch, so it would be 1.5 days long.
The review session is always closed, i.e. only for people from the project, reviewers and project officer.
For Cascadas, only 9-10 key people will join (WP Leaders, a couple of other selected people and I).
There should be technical presentations on the progress and achievements in each WP, and also on management
and financial issues. This should take up Monday afternoon and Tuesday morning.
There should also be a discussion of the detailed implementation plan for months M13 - M30 during Tuesday afternoon.
Of course there should be time allocated for discussion.
We are expected to propose an agenda. I will come back to WP Leaders on that later on).
All Partners are expected to join the SAC workshop (but not the review, as it will be closed). I will come back to you (all) on that later on.
Cheers, antonio
</Value>
<Type>java.lang.String</Type>
</Data>
```

Figure 6: Knowledge Network Explorer

As can be seen the hierarchical as well as the vertical organization has practically no limit and can be controlled by the application itself or any other supervision mechanism. Furthermore, the methods available by individual services are also dynamic in a way that it cannot be guaranteed that a certain type of functionality exists always. Thus KN structures may be explored and tested for the existence of specific functionality instead of using them in a brute force approach where static structures are assumed. While this is actually a feature with respect to dynamic service orchestration it is a drawback with respect to documenting the utilisation thereof. Because of the dynamicity of the overall system no clear and finite functional repository can be defined, instead a service discovery mechanism should be implemented into the dynamic exploration of the system so that the utilization of dynamic functionality can be automated. This is envisioned to be facilitated via the GN-GA protocol of WP1. For standardized functionality that is concerned with e.g. the (de-)registration of services, please refer to the java documentation provided.

4.1.1 Example Services

Standard KN components may be extended via the plug-in of desired sub-services that perform a specific goal that is either specific to the application that has registered the service or it may provide additional, not standardised, functionality to be used by other applications. For knowledge atoms two good examples for such services are the provision of a set of history values or a dedicated push service that eases the detection of change within data. Both are exemplified within this section and will be provided as part of the final tool-set.

- Push Service

This section proposes the skeleton of an embedded push mechanism based on the concepts outlined earlier. Of course each application needs to subclass a dedicated listener interface to accommodate for specific notification procedures. Although, this can also be standardised depending on the communication interface chosen for the overall KN framework.

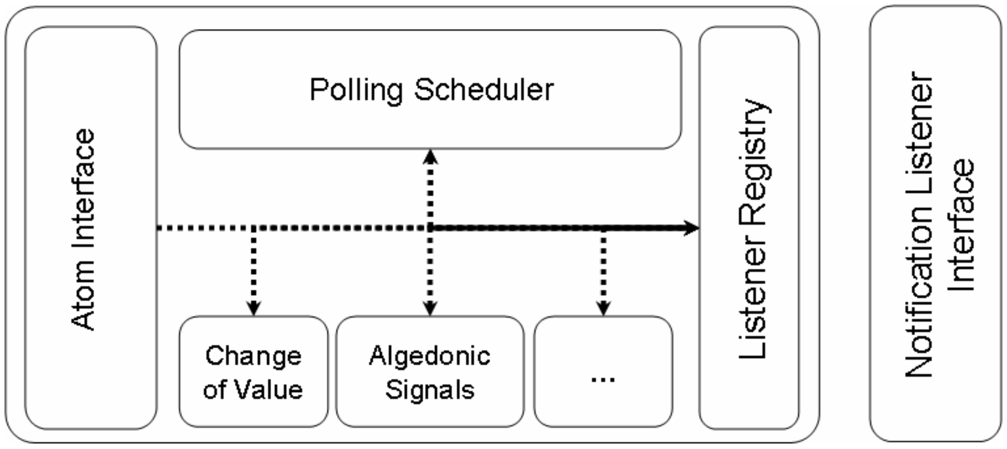


Figure 7: Push Service (Schematic Architecture)

Figure 7 shows a schematic architecture of an embedded push mechanism where relevant data is accessed form the push service via the atom interface (left side) and where applications can register specific notification listeners (right side). The service component itself consists of three parts that are compulsory. Namely, (a) a generic data access mechanism (atom interface), (b) a configuration part that controls the overall service based on individual configuration parameters (polling scheduler) and (c) a listener registry where registered notification handlers are stored (Listener Registry). Other parts may be added to this component to perform specific evaluation task that are relevant to this service. For instance, a change of value sub-component could test for a change of value whereas another component could serve specific algedonic signals such as a heartbeat to interested (registered) applications. In any case, it has to be stressed that none of these components should serve the underlying data to applications but should act as a notification service. If an application needs to access the data after being notified then it may requests these data via the Atom interface directly.

Skeleton Interface for Notification Listener

```
Interface NotificationListener {
        public notify(Object msg);
}
```

Simplified, the only one relevant configuration parameter for a generic push service is the minimum polling interval. It may be passed directly as a long value representing an interval in e.g. milliseconds. In addition, we have to assume that a dedicated scheduler is present which is able to execute the needed methods at the proper time. The polling configuration module, which utilises the scheduler, is responsible for executing itself and relevant sub-components at the given frequency in order to serve listeners at the requested intervals. Finally, every application which intends to utilise the push service has to implement the NotificationListener interface, which includes just one method in order to pass a generic notification message. For now it is assumed that the method itself handles the overall notification mechanism independent of the fact that the receiver is local or remote. For convenience, the skeleton class for a default PushService can be found within the appendix.

- History Component

As the name suggests a history component may provide a history of the atoms values which it is loaded into. To be generic the atoms history should comprise objects rather than specific data types. While this limits the analytical functionality of the component it ensures generic concept of the overall framework. In practice all standardised services should be realised in a generic fashion simply because of the fact that the perception of data, the usage thereof or the calculation / derivation of other data may differ between applications. Thus services should be as generic as possible to be specialised via the dynamic utilisation of specific evaluation functions which is again in line with the dynamic service architecture provided.

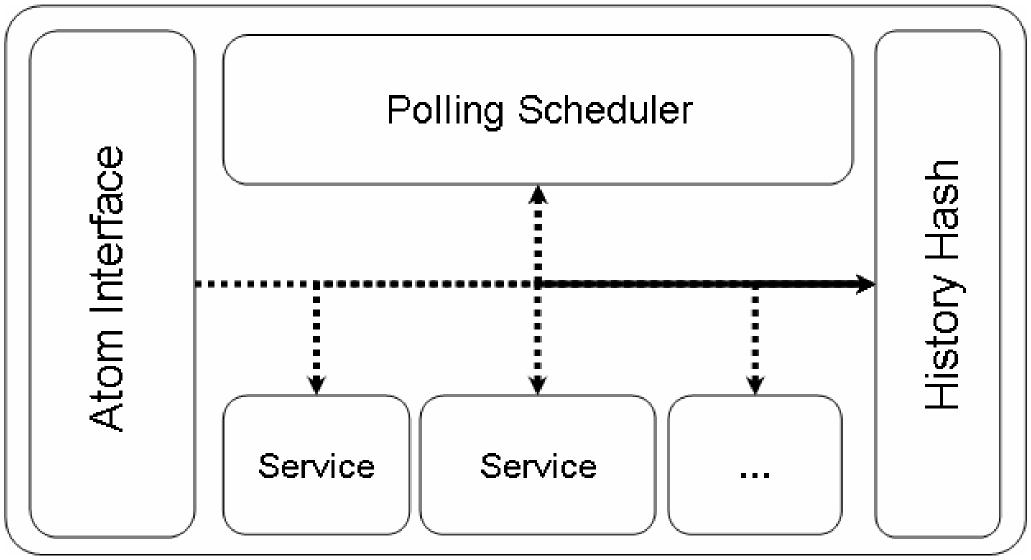


Figure 8: History Service Architecture

Figure 8 depicts the generic architecture of a history component comprising access to the atom the history should be maintained for, here visualised via the atom interface indicating that the history service being a sub-service of atom could be hosted locally or remotely depending on how it is loaded. This is beneficial in that the atom host does not have enough resources to also host the history service. Other sub-components of the history service are the history hash itself providing the data set, a polling scheduler supervising when a new value should be taken and obviously a bus architecture that links all sub-components together and also provides the functionality to further orchestrate the components itself with other more application specific services. For the sake of completeness a property component may be added that provides individual configuration parameters.

Interestingly at this stage, the polling mechanisms described previously actually comprises the atom interface as well as the polling scheduler. Thus the history component could actually be implemented as a more specific solution of the push service proving the orchestratability of generic services.

4.2 Knowledge Atom

4.2.1 ACE Atoms

Being the most basic concept of a knowledge network, a knowledge atom represents the generic data access layer for Knowledge Networks. In a nutshell it contains or provides access to the knowledge source object and relevant descriptions that provide the context of the data or knowledge represented by it. As defined in Del.5.1, the sole purpose of a knowledge atom is to provide generic access to a specific data source and to allow the registration thereof into the scope of a knowledge network. It is not concerned with any organisational aspects within or outside the knowledge network nor is it responsible for the configuration, maintenance or (de-) registration thereof. To be more specific, a knowledge atom provides access to data and knowledge, it is not concerned with the sensing, construction, maintenance or organisation thereof.

Currently, data may be loaded into a local atom implementation called an Embedded Atom or specific proxies may be provided that (a) realise access to the data wherever it is hosted or generated and (b) expose the atom interface so that they can be used from within the knowledge network. For more information on how such atoms are realize and registered into the scope of a KN please refer to the appropriate user guide as provided in the appendix.

At this stage it has to be stressed that all specific realizations are only temporary. Considering that within Cascadas, basically everything will be realized via the concept of ACE's, atoms may also be realised as ACE's. Alternatively, an ACE may choose to publish data via the atom interface directly. In this case an ACE is not actually an atom but provides certain data by this interface. This reflects a nice solution in a way that an ACE does not have to "turn" into an atom to provide data and more importantly it also allows publishing more than one atom from within the same ACE.

4.2.2 Sensor Atoms - micaZ

This section describes the software package needed to integrate real world sensors nodes into knowledge networks. The chosen devices are CrossBow micaZ motes. They are equipped with an 8bit CPU, 4Kb of memory, 512kb of EEprom storage, and a radio implementing the Zigbee protocol. Each node can be extended with different sensor boards. The most common boards provide readings for temperature, light, sound, magnetic field and acceleration (seismic). Moreover, these nodes can be connected with any external device with an analogical output through a proper I/O board. For the scope of our demonstration we have used a simple sensor board which provides sound, light and temperature readings. To integrate motes with the KN tool kit, three separated pieces of code were produced.

The first one has to be executed over the actual sensors. It has been written using nesC language and compiled over an event driven operating system called Tiny OS. The goal of this part is to push towards a base station data packets containing environmental readings. Each sensor periodically reads its own values, pack them into a Zigbee radio packet and send it towards the base station. Moreover each node, in order to build a fully fledged multi hop routing tree rooted at the base station, can act as a router forwarding packets received from neighbours nodes. Using this kind of architecture we can also show an example of a fully distributed knowledge network running directly over the sensors. In particular we have implemented an algorithm to in-network aggregate data values. In this way we can build over a flat WSN different regions characterized by different patterns of sensed data. This kind of mechanism can be considered a sort of low level knowledge network. A mobile user, or a service interacting directly with the network, can in fact perceive the environment not only as a dispersed collection of single nodes but as a simpler set of 'virtual' macro sensors represented by the regions. This is, in our opinion, a clear example of knowledge organization, inspired to the principles of knowledge networks, implemented at a lower abstraction level.

Once data packets, containing either simple sensor readings or more complex region based aggregated data, arrive at the base station, the major design issue concerns publishing them to the external world. We choose to use the HTTP interface. Each time a packet coming from a sensor reaches the base station, a hash map data structure is updated. The key values of this structure are sensors' ID and the values are arrays of properties related to that particular sensor. The software running on the base station is also used for the sake of managing properly the ageing of the data coming from the sensor network. All the readings are time stamped and, if some data become too old, they are discarded. All the sensors' data can be accessed from external services using an HTTP request such as:

```
http://baseStationHost:port/sensors/sensorID/
```

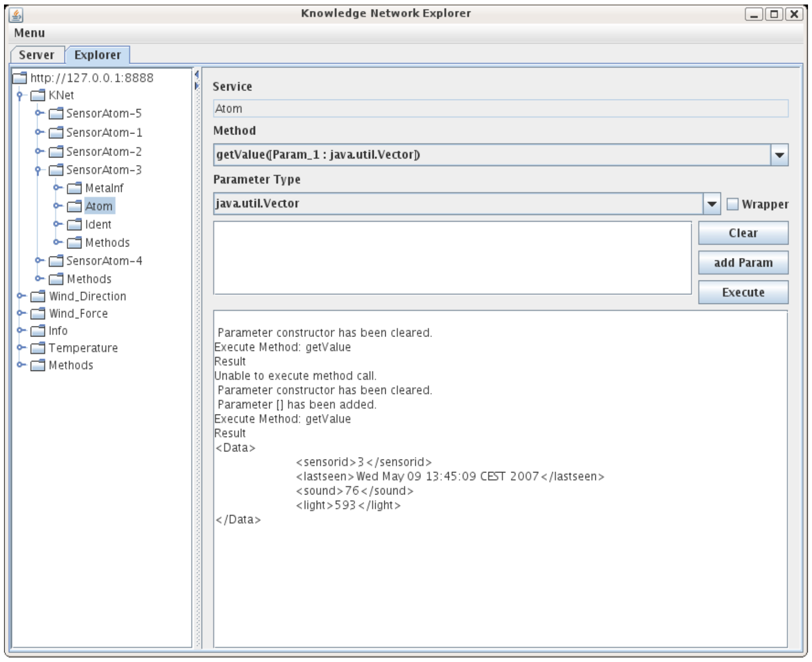
Text 1: HTTP Request

The software, after checking that data is available and fresh, replies with a HTML page organized as follows:

```
sensorid = 3

lastseen = Wed May 09 13:41:21 CEST 2007

sound = 76

light = 593
```

Text 2: HTTP response

The example shown in Text 2 describes only sound and light readings. Nevertheless, any additional measure can be inserted with almost no additional effort.



Service: Atom

Method: getValue(Param_1 : java.util.Vector)

Parameter Type: java.util.Vector

```
 Parameter constructor has been cleared.
Execute Method: getValue
Result
Unable to execute method call.
 Parameter constructor has been cleared.
 Parameter [] has been added.
Execute Method: getValue
Result
<Data>
            <sensorid>3</sensorid>
            <lastseen>Wed May 09 13:45:09 CEST 2007</lastseen>
            <sound>76</sound>
            <light>593</light>
</Data>
```

Figure 9: Knowledge Network Explorer showing data coming from real sensors.

The final step to close the loop and make sensors' readings accessible from the Knowledge Networks toolkit is to write a proper interface to the HTTP server running on the base station. To reach the goal a generic HTTP KA was specialized into an HTTPSensorAtom KA. This new java class is obviously able to build a connection to a generic HTTP server but also to understand the syntax used from the based station to publish the data. By this way, using our generic Knowledge Networks Explorer, we can browse through a network fulfilled with data coming from a spread range of sources including real sensors (as shown in picture below).

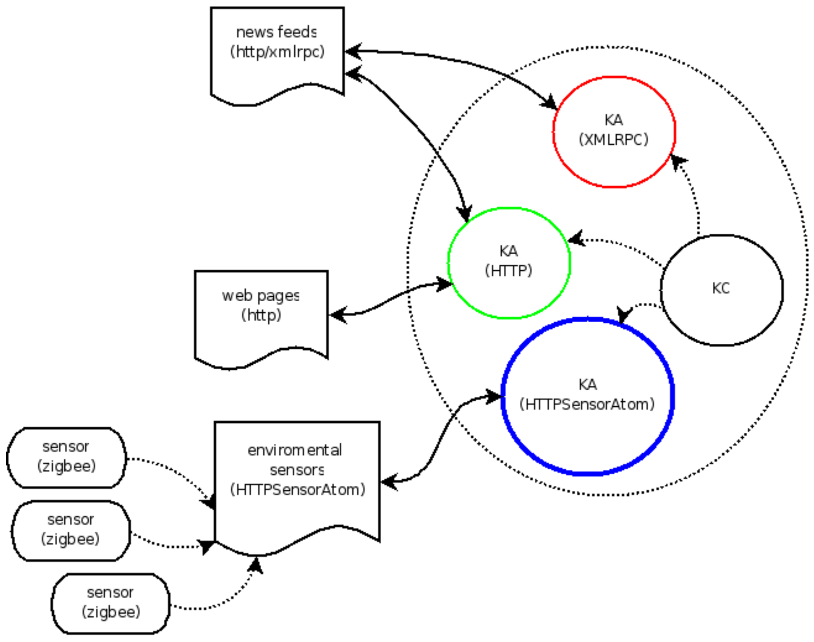


Figure 10: ***Knowledge Networks interaction with different sources. In this case, zigbee sensors are accessed through a specialized HTPP interface. The internal KC organise heterogeneous data.***

It is worth emphasizing that whatever nesC code can be used over sensors. In our current version (Figure 10), for example, data are periodically pushed from the network to the base station which exposes them through the described HTTP interface. Actually, this is only one possible way to interact with sensor nodes. We can for example program the base station to wait for queries from the outside, translate them using some WSN framework like TAG or Agilla (Figure 11), wait for an answer from the network and then reply to the application which queried the data. Our knowledge network framework can handle whatever data source with the only burden of writing the proper KA needed to fetch data.

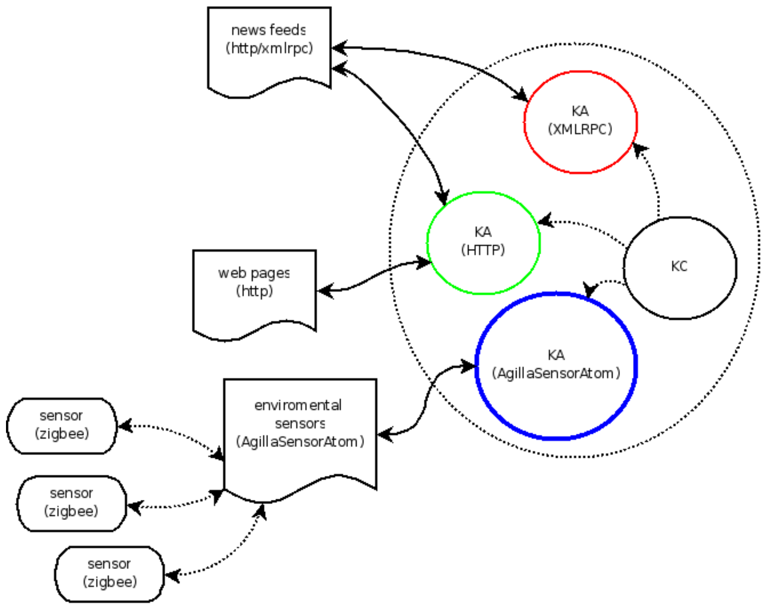


Figure 11: Knowledge Networks interaction with different sources. In this case, zigbee sensors are accessed through the Agilla Framework. The internal KC organises heterogeneous data.

4.3 Knowledge Container

A Knowledge Container is the component of the knowledge network where the actual organization of knowledge is facilitated. It will aggregate atoms as well as other containers depending on generic or application specific concepts. Ideally it does not access the underlying data which however may be bypassed for data related to e.g. the geographical location of a sensor where the underlying organisational concept may be the same as the actual data object provided by the atom. Two organisational concepts have been explored so far and are described in this Section. While the first one offers a more network oriented organisation that could be applied from within the atoms themselves the other one is based on semantic meanings.

4.3.1 Network Based Organisation

A dedicated way of organizing knowledge that has been exploited in the context of knowledge networks is network-based organization. Network-based organization concerns giving KC the capability of identifying and enforcing direct relations between KA's that are somewhat "related" with each other. Such relations between KA might generally derive from observed "matches" between the feature sets of different KA's, and from the logic exploited to evaluate such matches.

In general, we can assume the existence of a matching function MF that, applied on two (or more) knowledge atoms KA1, KA2, determines if there is a relation between such KAs. Depending on its specific characteristics, such function can return either a Boolean or a sort of "distance" measure between KA1 and KA2. (e.g., Dist = MF(KA1, KA2)). By relying on this pattern matching function, and by having the KC internal logic and processes apply such function to the various KC in a knowledge network (even in a fully distributed environment in which KAs can reside on different nodes), it is possible to self-organize a network of relations between KAs. Such networks of relations can be used to provide to application-level services a higher-level yet simplified view of the current situation of a context.

We have already applied such form of knowledge network organization for performing spatial data aggregation in sensor networks (as already described in Deliverable D5.1). Data aggregation is a very general operation in sensor networks. The idea is to aggregate sensors with similar readings, so as to provide services with compact information about the environment, instead of the individual values of each sensor in such environment. Over an existing environment fulfilled of sensing ACE's (i.e., sensor nodes acting as ACEs) exposing the KA interface we can inject a pattern-matching algorithm to build links between logically correlated neighbour sensors (i.e., the KAs representing these sensors). The pattern matching function "Distance" currently exploited simply measure the distance between the valued sensed by sensors, and a relation "rel" between two KA is reinforced is the distance is below a threshold "Th", weakened otherwise:

Pattern Matching Function - Distance

```
if Distance(v($s_i$),  v($s_j$)) < Th {
        rel($s_i$,$s_j$) = min(l($s_i$,$s_j$) + delta, 1)
} else {
        rel($s_i$,$s_j$) = max(l($s_i$,$s_j$) - delta, 0)
}
```

Eventually, having such process repeated over and over, lead to the self-partitioning of KAs into distinct regions: KAs related by a "rel" of value 1 belong to the same region, "rel" of values 0 identify frontiers between different regions.

Let us refer to Figure 12 to clarify the above concepts. In Figure 12 (a), one can see the KA associated to sensor, where the arrows link the sensor couples between which to apply a pattern-matching function (which, in the case of sensor networks, corresponds to sensors in wireless range with each other, to which to apply the "Distance" function). From Figure 12 (b), one can see that, eventually, the application of the pattern-matching function lead to identifying relations (represented by red arrows) between KAs. In the case of sensor networks, this implies identifying non-overlapping regions of sensors related to each other (regions A, B, C, in the figure). From Figure 12 (c), one can see how one specific KC can be associated to each of the identified region, to act as the access point to the data in this region, and as an aggregation point. Interestingly, the same approach can also be applied at higher-levels, i.e., at the level of KCs associated to group of related KAs (i.e., in the case of sensor networks, to neighbour KCs representing neighbour regions). That is, it is possible to identify relations between KCs (as from Figure 12 (d).

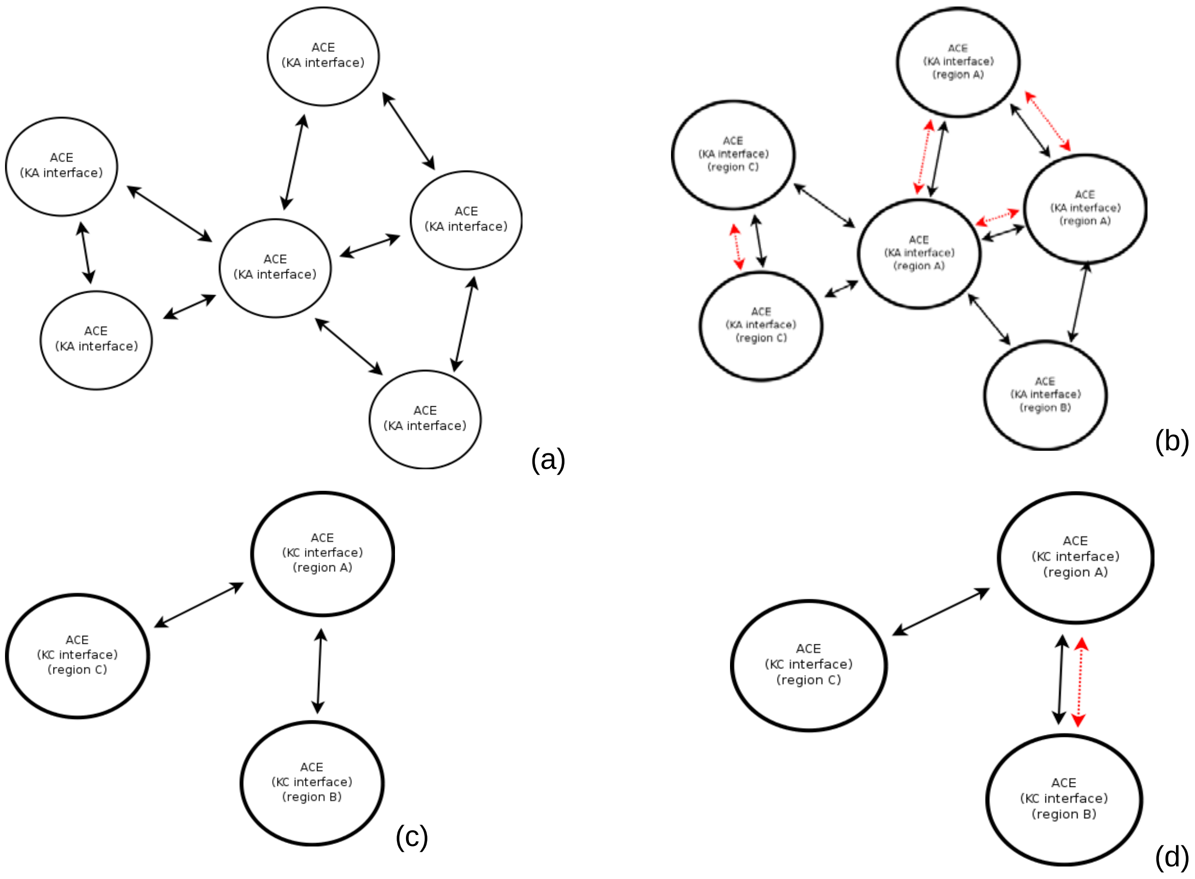

ACE (KA interface)
(a)
ACE (KA interface) (region A)
ACE (KA interface) (region C)
ACE (KA interface) (region B)
(b)
ACE (KC interface) (region A)
ACE (KC interface) (region C)
ACE (KC interface) (region B)
(c)
(d)

Figure 12: Network-based Self-Organisation

The details of the self-organized region formation algorithms applied to sensor networks are described in more detail in [D5.1] and in [BicMZ07]. Additional experiments on general pattern-matching functions and aggregation algorithms are being performed within WP5.

4.3.2 Concept Based Organisation

The objective of the proposed knowledge network is to organise data in a way that makes it easier for a user to retrieve context and situation aware collections of knowledge. The two base components that are utilised to construct knowledge networks are knowledge atoms and knowledge containers. While the former facilitates generic access to data the latter is solely concerned with the organisation thereof. In theory, the knowledge network will be hierarchical, grouping atoms based on their semantic meaning. For that the knowledge in the atom will be represented by a set of semantic keywords. The knowledge container may then contain links to the set of atoms it groups together. Alternatively, the container may also reference other containers, thus creating a hierarchical or network like architecture. Such a structure is depicted in Figure 13 showing a simple knowledge network that represents a weather sensor system.

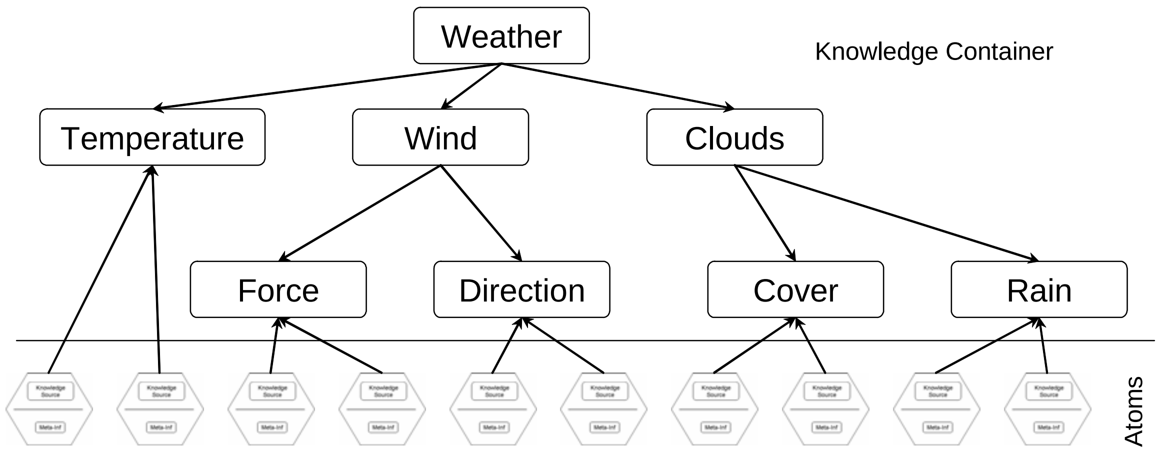


Figure 13: Example Knowledge Network (Weather)

This example shows the following elements. There are three types of sensors: one to measure temperature, one to measure wind force and one to measure wind direction. Note that there may be any number of actual sensors that are collectively grouped in the respective categories. Wind force and wind direction can be grouped into wind, and wind and temperature can then be grouped into weather. The weather element is a container that contains other containers called temperature, wind and clouds. While the Temperature container only contains atoms, wind and clouds are sub-grouped further as depicted in the example shown.

Example Atom - Semantics

```
<Keywords>
        <Keyword>Weather.Wind.Force</Keyword>
        <Keyword>Europe.UK.Belfast</Keyword>
        <Keyword>GPS[Standardised GPS Information]</Keyword>
        <Keyword>metoffice.gov.uk</Keyword>
</Keywords>
```

The semantics describing a resource should follow a hierarchical dot-separated namespace or alternatively may be represented in RDF thus allowing for more complex representations as well as a better understanding about the state of the resource. Constructing such relations from the bottom-up, that is from the data (atom) level, requires that that each atom is self-descriptive in a way that it provides the semantics it is based on. For instance, a KA representing the wind force @ location Belfast may provide a set of keywords as shown above. Thus providing all necessary information to either link or generate respective ontology's as shown below.

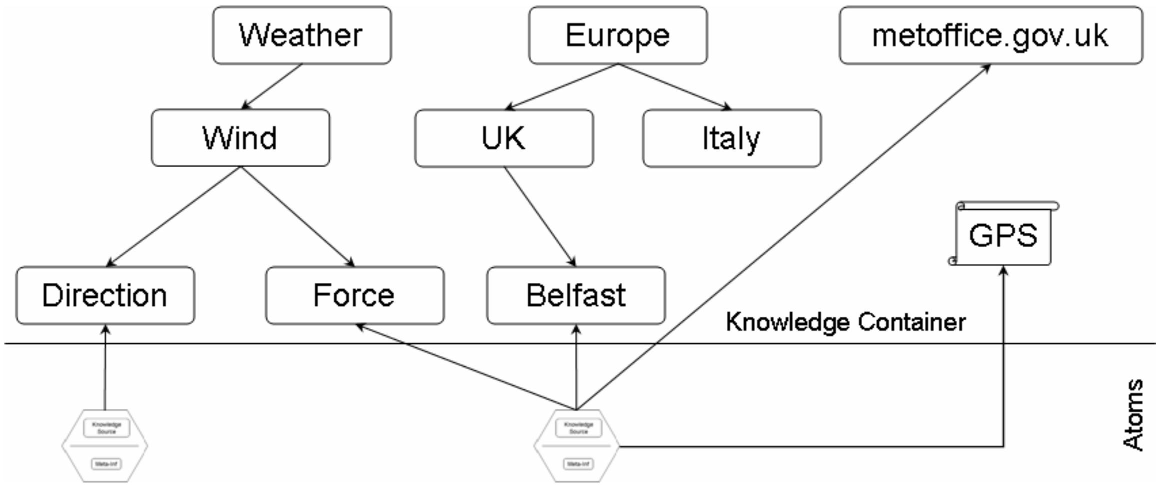


Figure 14: Example Ontology

Note that such an ontology is always shared, this means that no part of any given construct is created or used only for a single object but always available for all objects that are within the same organisational space. Thus, as shown, the respective ontology of the example atom depicted may contain other concepts, e.g. the Weather.Wind.Direction created through e.g. the atom depicted on the left. Alternatively, containers may be added directly as visualised through the Europe.Italy container which is not yet linked to any knowledge source. Thus knowledge network constructs may be created n the following fashion

- **Bottom-Up through self descriptive Knowledge Atoms:** The obvious advantage of this method is that the resulting hierarchical or network like knowledge structures can be generated autonomously without any interference from the user. However, if the semantics provided by the data layer is incorrect or purposely falsified then the resulting knowledge structures are also incorrect.
- **Top-Bottom through existing ontology's:** For instance, if an ontology is known beforehand for a particular domain, then this ontology may be modelled via knowledge containers and declared static in a way that it may not be altered by internal self-organisation. While this renders the organisational space to be static it also ensures that knowledge can only be mapped into the existing structures thus avoiding the generation false relations. In addition, a user or application can directly specify the type and structure of the knowledge of interest. This allows for the generation of purpose build knowledge structures serving specific needs for specific contexts.
- **Mixed Construction and validation:** In practice, individual relations or full scale ontology's cannot always be provided for any scenario. Furthermore, different ontology's may be required for different applications. Thus, generating KN's dynamically via self-descriptive knowledge atoms is always preferable. Nevertheless, the resulting knowledge structures should be validated wherever possible in order to provide a continuing quality of service. Such a service can be performed at three different levels; (a) when creating a distinct relation, (b) as a

background service that constantly compares existing and validated knowledge with the structures created and (c) at application level, where the knowledge consumer provides the ontology's the knowledge has to match.

The above has, so far, only dealt with static semantic concepts such as the location (provided semantically), the purpose and the domain of the knowledge to be mapped. However, other more dynamic concepts also have to be dealt with. Probably the best example of this is GPS data that provide the geographical location of a resource as also depicted in Figure 14. When static they may be translated into a more meaningful and human understandable semantic representations such as addresses, town names etc. On the other hand when used for movable resources (e.g. mobile phones and RFID tags) such mapping is not always possible or desired. On the contrarily, GPS data can be used directly for geographical mapping purposes, distance calculations, clustering etc. For this to be used efficiently within knowledge networks such data should be linked to active knowledge containers that utilise distinct algorithms that provide purpose based organisation.

Finally, considering that knowledge atoms as well as knowledge containers are fully independent entities (ACE's after they have been integrated into the Cascades framework), they may be hosted locally or fully distributed thus guaranteeing a lightweight knowledge structure as well as scalability as each knowledge object operates independently.

In order to explore and validate concept based organisation, a prototype has been implemented that is part of the M18 software package [KNOrg]. Although not yet integrated into the service architecture it shows that independent knowledge sources can be successfully organised resulting in dynamic ontology's constructs that can be created queried efficiently. Subsequent steps will include a full integration into the service architecture to explore the distributed organisation at a larger scale in more detail.

5 Knowledge Execution

5.1 Knowledge Querying

Following the example depicted in Section 4.3.2, a KN may represent any number of knowledge atoms that are organised based on various semantic concepts. Assuming that this structure exists and is readily available, the question is how can it be most efficiently queried? This section discusses some important aspects related to the problem of querying a KN like structure to allow for efficient knowledge retrieval. While a prototypical querying engine has already being developed it is not yet fully integrated into the KN architecture and as such not part of the beta prototype.

5.1.1 Query Process

There are a number of query languages that can be used to query XML, which is used to represent the knowledge that is embraced by a KN. One possibility is to have two different phases to the querying. The first phase is the search through the network to find the most suitable sources. The second phase is to actually query the sources. One suggested approach is to firstly find all relevant sources and retrieve their addresses and then, at a second stage, to query the sources directly. This is appealing for the following reasons: We can separate the search process from the query process. This means that we can use a different language to search rather than to query. If the knowledge is stored as keywords, then a simple structure like RDF can be used to represent this. We can then use one of several RDF query languages to navigate through it. Because this is a simpler structure than XML, it means we can make the nodes in the network more lightweight if they only have to process this information. The sources however may be large XML documents with a lot of complexity. To properly query such documents a more sophisticated query language is needed. However, because all of the relevant sources are known beforehand, a heavyweight query engine as part of the client or root node that does this part of the querying would be possible which also would not be a direct part of the network as a whole. The main drawback to this approach is that it is more centralised than distributed and so goes against an autonomic approach.

An alternative approach would be to perform the query search and actual query execution through the network. The nodes in the network used for navigation will store and process RDF only. These are the containers. We then have a set of atom leaf nodes that access the sources directly. These do not perform any extra navigation but rather query a source to retrieve relevant values. These nodes do not have to process RDF but can be heterogeneous with respect to the query language and store a query engine suitable to the source. If we have a complex XML document, then the query engine might be something like XQuery or Xcerpt. If we have a simple sensor, then an RDF query language could be utilised directly. This leaf node will receive a query request from a parent node, convert it into a format suitable for its query engine, execute it and then return the result. The result will be an XML reply leaving the only problem of combining the replies into a consistent document. In a hierarchical network it is preferred to allow references only in a downward manner, so as to prevent cycling in a search process. However, links to parent components can exist but not be used as part of the search process. Then, when a path is clearly defined, the backward reference can be used to perform the navigation. If using a select-from-where statement, the centralised and distributed approaches can be partially combined. The 'where' comparisons can be evaluated locally at nodes that compare a source to an exact value, for example 'value1 greater than 0'. Thus fewer nodes need to be returned.

The following scenario may describe what the heterogeneous approach would allow: Say we have a number of temperature sensors distributed over e.g. Belfast and we have a number of XML documents with knowledge on the buildings in Belfast and the people that work in them. A possible query could look like "retrieve the temperature from buildings where people work who wear ties and eat cornflakes" or in SQL like format "Select weather.temperature, From weather Where weather.building in (Select buildings.name From buildings Where buildings.people In (Select people.name From people Where people.wear Equals ties and people.eat Equals cornflakes))"

This is a typical select-from-where statement. While XML processing would be desirable for a complete system, it is not absolutely necessary for a test prototype. The select-from-where statement that queries for simple values should suffice for most of the required tests. If we enforce a nesting for processing then this could be done in individual stages. We first query the people, then retrieve the buildings and query that and then finally query the temperature sensors. When the query parts are sent through the network, all of the keywords relating to the whole query should be sent with it, not just the part currently being processed.

The query process will describe routes through the network that satisfy the query. If the user is satisfied with the query result, then the knowledge network itself may be updated to reflect this sort of successful knowledge retrieval. In fact, the network could be modified in an experience based way based on its use. A relatively simple way to do this is the following: We note the nodes visited to answer a query and construct links between nodes commonly associated together. So if a user submits a query, the first part is constructed and a path through the network is navigated. The sources process the query and send the reply back through the network. The nodes that receive the reply note that they have been used in processing this query. If the user then confirms that the query result is acceptable, these nodes then permanently update a structure that stores references to the other nodes used to answer the query (or query part).

5.1.2 Autonomic Querying

The knowledge network can use the querying process to autonomically organise itself with respect to knowledge. This is done by dynamically creating links between nodes that answer the same type of query. When a query is answered, the query engine can inform the nodes that were used to answer it. Related nodes can then form links between each other, which can be used as an organisational and optimising mechanism. For example, if there exists 100 temperature sensors and only 1 is used as part of a query, then only that single sensor could be linked to another part of the network while the other sensors are not included. Consider the following example:

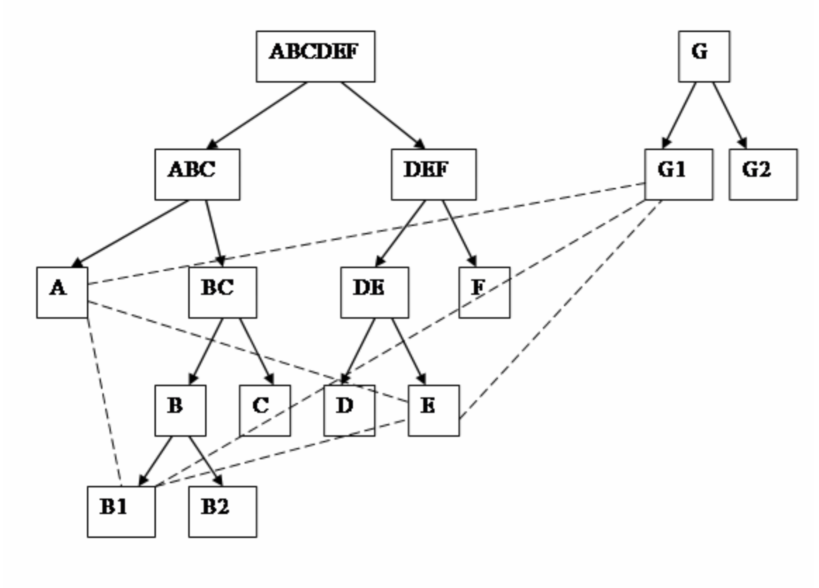


Figure 15: Example of Node Linkage through Querying

In this example we have a number of keywords semantically grouped together as shown by the solid arrows. The query that is consistently executed is ABE(G). The nodes that consistently answer this query advertise the keywords that they do not have. The other nodes reply that they also answer the query and temporary links are set up. There may be many nodes all of type A or type B or type E. We do not want node G to store references to all of these nodes, so only the ones that answer the query are referenced. After a communication, node ABCDEF sends the result to its sub-nodes that answered the query and they can update their weights. If full paths are stored then only those specific sources need to be informed. When it is discovered that this behaviour is consistent, these atom nodes then store more permanent references to one another. These are references to just single nodes not types of nodes and provide shortcuts through the network. These references are shown by the dashed lines. If node A is then used in some query that includes the keywords B, E and G, it also returns the references to the other nodes B1, E and G and if their tree needs to be queried they can be looked at first. This could help to speed up the query process and has the advantage of linking parts of the network that would otherwise maybe not be obviously linked together. When recording the references to sources, we can also store the path information, making it easy to directly access the sources if they are later specified. At the same time, if there exists a link that is then not subsequently used, the weight for it can be decreased until the link is permanently removed. This can be done through the network structure again as it was for the link creation. It could also be done by the source nodes communicating with each other directly after a query answer to check that they both answered it.

Obviously, this type of reorganisation relies on the same queries being consistently executed. For disperse data the queries may always be varied. But then there is no reason to modify the network structure based on the queries if it is always queried in a different manner.

5.1.3 The Query Language

XML is being used as the standard information representation mechanism for a knowledge network. It should be noted that for testing, complex XML-based sources may not be available or used. Thus a select-from-where statement that queries much simpler source values will probably be suitable. However, in the wider context of knowledge querying, XML-based query languages are important and so some will be described in these sections. This report will focus on three potential query languages that can be used to query XML data. These are OTK-RQL [OTK-RQL], SPARQL [SPARQL] and Xcerpt [Xcerpt]. All three languages can query RDF [RDF], while Xcerpt can also query XML in general. These languages will be assessed with regard to their suitability for processing RDF and XML in the knowledge network in the project.

5.1.4 Ontology-Based Search

Although the construction of the knowledge network is a separate issue, some mention of it should be made as it affects how the network is searched. We have a knowledge network with nodes defined by sets of keywords. One possibility is for the parent nodes to contain the keywords for the nodes in their subtrees. If this is not the case then we may need to search the entire network to retrieve the relevant sources. The other option is to have an ontology to provide the semantic relationships between the keywords. The subtrees in the network essentially define the different ontology domains, or there may be separate trees at different locations. Maybe the key construct for our ontology would just be the 'subclass' construct, which would link the keywords of parent to child nodes.

The hierarchical structure seems to store the relationships between concepts quite well. Maybe an advantage of an ontology is if there are unrelated concepts that can be linked to each other. For example pasta and bolognaise sauce. Then if the user asks for information on pasta, the bolognaise sauce node is also searched. But this is not a subclass relationship and requires prior knowledge of the contents of the network. Our method to dynamically link sources through stigmergy could be used to provide this sort of information. Another advantage of using the ontology could be to define all of the concepts a user can query. This gives the user an idea of what the network contains. If two separate trees both use the same ontology but store different components of it, then users querying the different trees can query over the whole ontology even if all information is not available at that specific tree. Next is a summary of the main query languages to be considered for a prototype implementation. The list of query languages discussed is by no means complete but represents the languages that are seen as mostly relevant for querying the proposed KN like structure.

OTK-RQL is an ontology-based language used for querying RDF. It was created as part of the On-To-Knowledge project. It is an extension of the RQL RDF query language. The project aims to achieve the goals of intelligent search instead of keyword matching, query answering instead on information retrieval and support for document maintenance and exchange. OTK-RQL queries ontologies written in DAML + OIL [DAML + OIL], where the OIL Core is the minimum specification. OIL stands for Ontology Inference Layer. RDF provides a simple data model for representing formal semantics of information, i.e. meta-information. RDF Schema provides a simple ontology modelling language on top of RDF that can be used to define vocabulary and structure of RDF. OIL adds a simple description logic to RDF Schema. It allows you to define axioms that logically describe classes, properties and their hierarchies. Sesame is a repository and querying facility for RDF schema. It has a query engine for the RDF Query Language (RQL). OTK-RQL provides a select-from-where filter and its syntax is like SQL. It supports almost the complete RQL vocabulary with some differences. The answer is returned as an RDF document. The main difference between OTK-RQL and RQL is the support for multiple domain and range queries.

SPARQL is another query language used to query RDF. The current trend is for pattern-based matching and SPARQL is based on pattern matching of graphs. SPARQL is used more for information retrieval and does not use ontologies. It does however has a describe operator that can return an RDF graph that describes the structure of one or more sources combined together. This could be used to retrieve a kind of ontology of the tree(s) being searched. Xcerpt is also a pattern-based query language that can query both RDF and XML. As such it can be used for querying complex XML documents. Xcerpt is also deductive and can deduce knowledge about its sources. Xcerpt requires that the source locations be specified, so it could only be fully constructed after the navigation phase.

As a knowledge network may contain sources of complex types, a sophisticated language is required. Something like Xcerpt may be capable of performing such a method of querying. This language will always only be used at the source nodes. If we have ontologies then we should generate these in the form of OIL and use OTK-RQL for navigation as it is also deductive. If we choose not to use ontologies, then SPARQL has the useful describe command that might be a help to the user to construct a query. For example, keywords are flat, but the describe operator would return a structure. Also, without ontologies the deductive element is missing and one should go for two pattern-based approaches.

5.1.5 KN Query Interface

An interface can be provided to allow querying of the knowledge network. Depending on the type of query to be executed we can allow a different type of interface. One type of query will search simply nodes that contain certain concepts. Another type of query will ask for specific value types for these nodes. Another type of query may ask a question or require lightweight reasoning. There may be different ways to accommodate these different query types, but they can all be specified within a select-from-where structure. If asking just for navigation to specific concepts, then only the select and from parts need to be used. For example, if the user asks 'retrieve all nodes that relate to clothes'. This could be specified as:

Select * from *clothes*.

If the user asks 'retrieve all nodes that relate to clothes that describe red shirts', then the query might look like:

Select shirts from *clothes* where *shirts.colour* equals *red*.

The stigmergic linking that has been studied will also allow some basic reasoning. This might allow the use to ask questions, such as 'what is the best colour of shirt to wear in the summer?' Links that relate shirts to the concept of summer, for example, can be retrieved and the shirt colours aggregated to give a best value. For example, if there a 5 links from shirts to the summer concept and 3 are red while 2 are blue, then the red colour could be described as the best.

A relatively simple GUI will allow the user to specify all of these query types. The query process will search for sources that answer each of the comparison parts in the 'where' clause in turn. Only sources that answer one comparison can be used for the other query parts. The search process is guided by the hierarchical structure of the network and also by any links that exist. The links can specify that only certain sources should be looked at, thus reducing the search space. Linked features will also allow for memory management, where sources can be restricted in the number of links they are allowed to store, or even learning algorithms to learn certain parameters. Queries can be run with or without the linking features.

While the current querying considers only simple value types, if more complex and heterogeneous sources were to be used, then the atom would require its own query engine to retrieve the required concepts from its XML-based source. This could be one of the XML-based query languages already described. The process would then require a search to find the sources that contain the concepts and then ask those sources to retrieve the data relating to the query specification only. Some pattern-based mechanism could then be used to combine the different replies.

5.1.6 Summary

The above has suggested a querying system for knowledge networks that takes advantage of the hierarchical structure of the network. It suggests two different query stages – one to traverse the network and one to query the sources. As it is applied to a hierarchical structure, this will largely guide the search. One can move to the node that closest matches the query specification based on e.g. keyword similarity. Because of this, we have tried to improve the query search in an experience based way with respect to directly linking sources for certain query types, which has been done in a stigmergic and autonomic manner. There may be a number of sources that all offer the same service, so it would be helpful to note that if a certain source is used with a certain query, then other sources are also often used. This could speed up the query process or possibly improve the quality of service. For the query language, the order of execution is critical. In addition, keeping individual components of knowledge networks as lightweight as possible is another important factor. Thus a query a language like Xcerpt may not be ideal for knowledge containers. The knowledge atoms however may need to be more heavyweight with regard to querying as they may have to query complex structures or even databases. It should be noted that a select-from-where statement can be used to query RDF as well, which may be the preferred language for storing container concepts.

The experience-based updating can be considered as an extra phase of querying. It is only performed after a query is executed and offers self-optimisation via introspective learning. The updating in effect also updates the ontologies created for the source nodes. This is turn changes the knowledge stored in the network through its constructed relations. But because it is done on a personal level for each source, it does not affect the overall general organisational structure.

5.2 Knowledge Verification

Equipping applications with the ability to adapt to different situations requires information about the applications' environment. This information is called "context". Any contextual information has to be sensed, measured, or derived from a data source before it can be used as an input for a context aware application. In the flow of this process errors can occur at several points. Defective hardware, misinterpreted data, or unforeseen circumstances may cause context failures. Faulty context data may lead to a malfunction of a context-aware application. For this reason a method for context data verification is required. Contextual data can be distinguished into low-level context and high-level context, where low-level context comprises data directly derived from sensors and high-level context is information inferred from low-level context. For example, the longitude and latitude of a GPS data set is low-level, whereas the interpretation of this data, "at home", is high-level context.

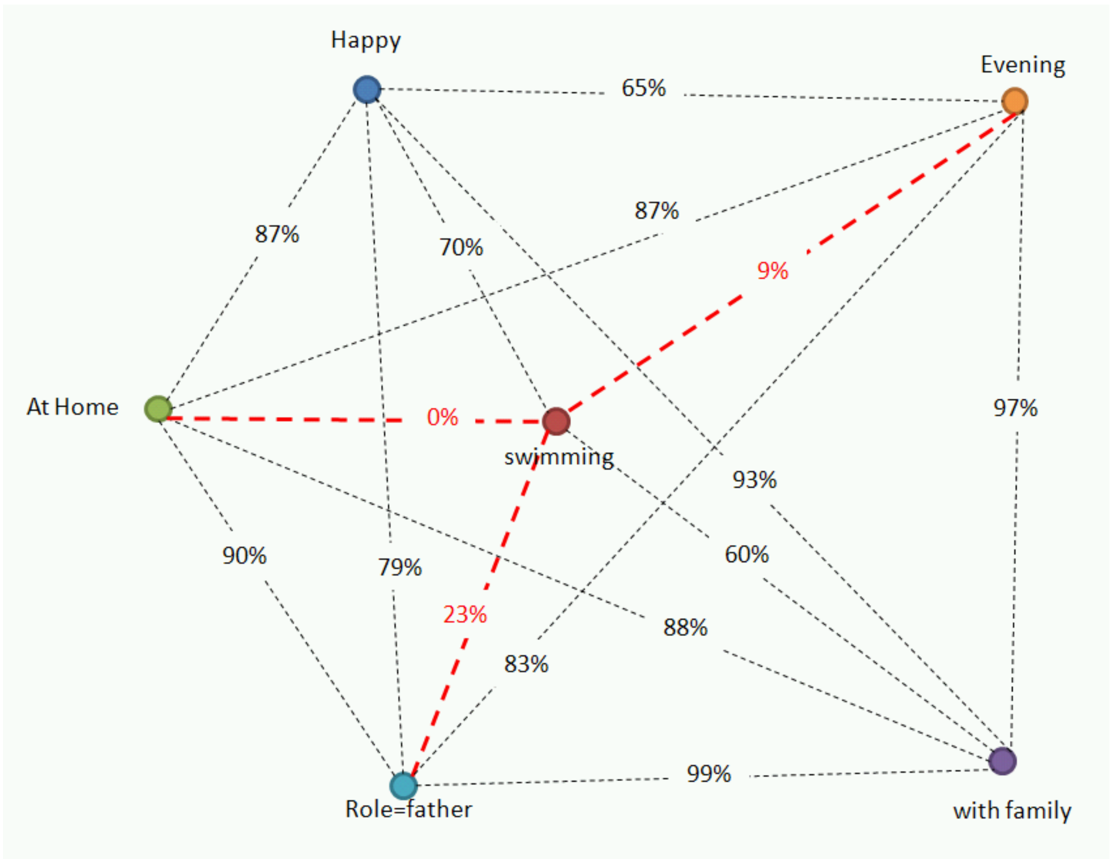


Figure 16: Groups of interrelated high-level contexts; because the "father's" activity "swimming" rarely happened in the "evening" and never "at home" the context data "swimming" is suspicious

In our context verification approach we introduce context patterns, which are defined as states a context group can reach. Each context group consists of interrelated high-level contexts which can have different values and can therefore create different patterns. Observing the context values and calculating their occurrence frequencies then enables us to detect potentially faulty high-level contexts (cf. Figure 16). Thereafter, it is required to locate the source of the error. In our approach, we assume that a high-level context usually is inferred from several low-level contexts and a low-level context usually is involved in the creation of several high-level contexts (cf. Figure 17). Following this and given that we have located a faulty high-level context, we need to analyse all low-level contexts from which the detected faulty high-level context is inferred. Each of them has to be individually verified by analysing all inferred high-level contexts and the context patterns of all groups they belong to. Detecting further anomalies then indicates that the underlying low-level context is an error source.

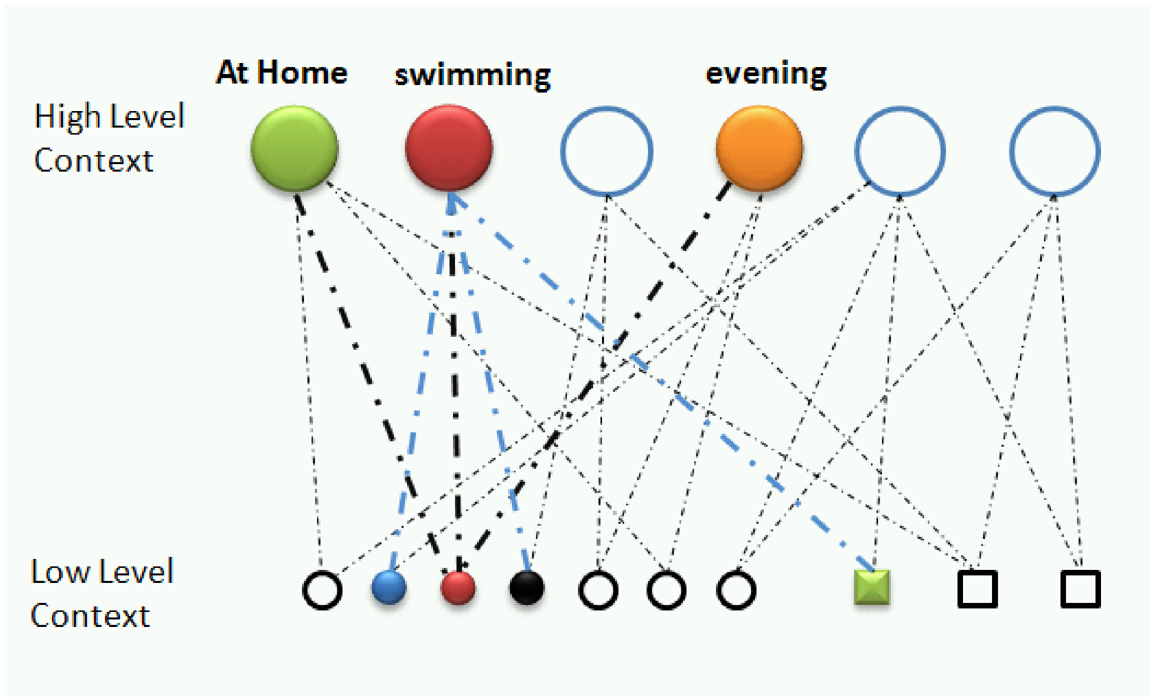


Figure 17: Inferring high-level context from low-level context

6 Integration with the ACE Framework

The KN software developed so far has to face, in the next few month, a notable restructuring in order to make it inter-operable with the ACE framework and in order to realize it in terms of ACEs, to exploit the additional autonomic features of ACE components. Although this activity has not practically started yet, we have already clear ideas on how to proceed, which we report in this section.

6.1 ACE Integration Process

The integration of the ACE model (WP1) and knowledge networks (WP5) will proceed in two main directions:

- Firstly, an interface will be developed to retrieve, access and produce information in the knowledge network that is compatible with the standard way adopted by ACEs to interact with each other (i.e., GN-GA protocol and contracting). Basically, this interface makes knowledge networks available and accessible like any other ACE-based resource.
- Secondly, the core mechanisms of knowledge networks will be integrated on top of the ACE architecture. This completes the integration in that the knowledge network components (knowledge atoms and knowledge containers) are actually ACEs themselves.

6.2 Knowledge Network Interface

To describe the interface to the knowledge network it is fundamental to briefly recap how ACEs interact with each other and with external resources. Of course, it would be valuable if ACEs could access the knowledge network with the same mechanisms they already have. The gateway is the ACE internal component in charge of communicating with the external world. The gateway provides two communication mechanisms:

- An event-based, session-less mechanism supporting the GN-GA protocol and other simple interactions. This mechanism is realized upon the services offered by the REDS1 event-dispatching middleware.
- A point-to-point session-capable mechanism supporting complex interactions among ACEs. This mechanism is realized via the MirrorAgent* functionalities provided by the DIET agent framework (this is actually similar to Java RMI functionalities).

The standard interaction procedure exploits both of the two services:

- The event-based mechanism is used to locate suitable interaction partners. This typically involves (i) an ACE sending a GN (Goal Needed) message to advertise its need for a specific goal, and (ii) other ACEs, that are able to execute such goal, answer with a GA (Goal Achievable) message describing in detail the goal/task they are willing to provide.
- The requesting ACE then selects the other ACE with which to interact, and uses the point-to-point mechanism to actually invoke the service (the point-to-point mechanism also support contracting to let the ACEs agree on the conditions under which the service invocation will be realized).

Such a general interaction mechanism can be readily exploited to access the knowledge network:

- The GN-GA protocol will be used by ACEs to locate the most appropriate Knowledge Container (KC) to satisfy their knowledge need. Specifically we subclass the GN and GA message classes to have knowledge network specific message format: KNM (Knowledge Needed Message), KAM (Knowledge Available Message).
    a. KNM provides a template to express interest to a class of context information (e.g., I want all the information related to a specific region, or to a specific user)
    b. KAM provides another template specifying which kind of information is produced by a given entity.

  The communication middleware (i.e., REDS) takes care of routing KNM messages to matching KAM messages so that relevant information sources are discovered.

[1] See WP1 documentation for details.

- Once suitable KCs have been discovered, they can be directly accessed via the point-to-point mechanism to actually retrieve the requested information. Such point to point interaction also allows negotiating on specific information access modalities (e.g., quality of information, sampling rate, access rights, policies etc.).

Given this interaction mechanism, the knowledge network, and in particular KCs are perceived by other components as if they were ACEs.

6.3 Realising Knowledge Network via ACEs.

The following step is to realise the core mechanisms and components of the knowledge network on top of the ACE architecture. The ACE architecture as developed by WP1 is depicted in Figure 18. In order to modify knowledge network components, namely KA and KC according to this architecture (as well as any other ACE-based component) the Self Model and the required Functionalities thereof has to be specified. All other components are mainly static and constitute the backbone of any kind of ACE's.

For a complete description of these ACE internal components, the reader should refer to WP1 documentation. Here only the basic mechanism of ACE's that are relevant to knowledge networks are recapitulated:

- The ACE Self Model describes at an abstract level what the ACE is capable of doing and how it realizes its functionalities.
- The Facilitator creates a Plan on the basis of the current system circumstances (i.e., context) by loading specific behaviour from the Self Model. The Plan is represented by means of a finite state automaton. (from another perspective, we can say that the Self Model contains all the Plans the ACE is capable of performing. The Facilitator selects one of these plans to be executed at a given time for a specific purpose).
- The Reasoner is the component in charge of running the finite state automaton triggering its transitions.
- Upon the triggering of a transition, the Plan indicates which Functionality has to be called to achieve the transition.

All these components apart from the Self Model and the Functionalities are either common to all the ACEs (and already implemented by WP1) or dynamically generated (e.g. Plan from the Self Model).

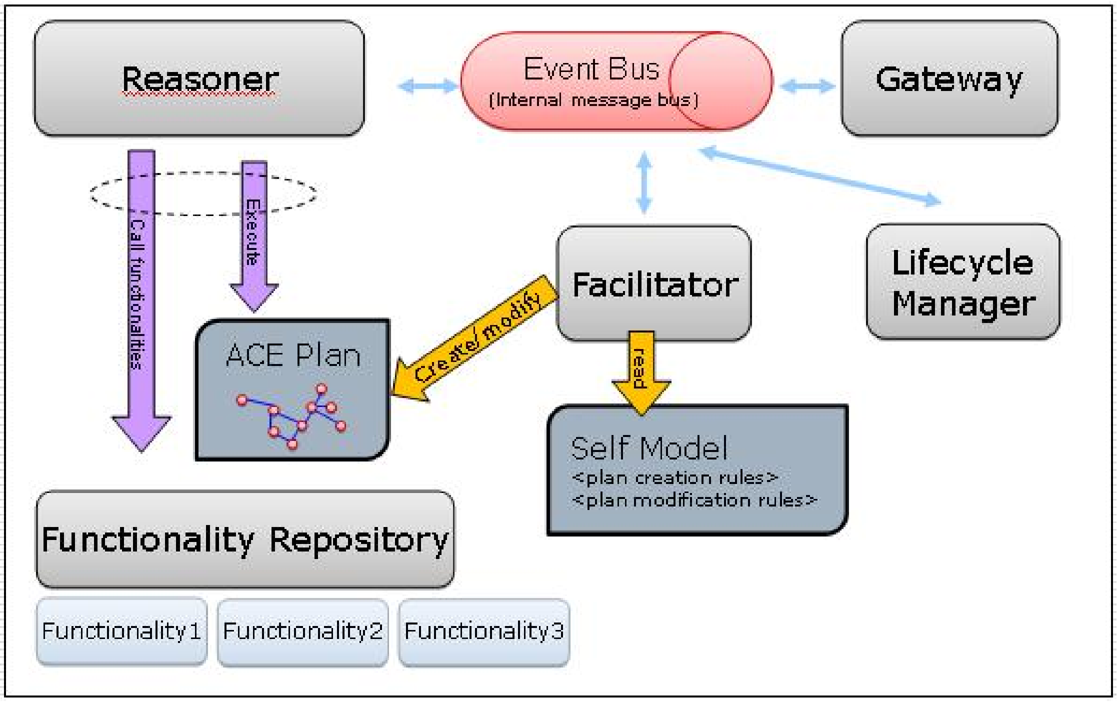


Figure 18: ACE Architecture

6.4 Implementing Knowledge Atoms

At the most basic level KAs will be implemented according to the trivial Self Model (that actually constitutes a single constant Plan) as depicted in Figure 19.

This Self Model mainly states the following actions:

- If receiving a KNM message that matches the kind of information the atom is able to produce, then reply with a KAM message describing such kind of information
- If a query point-point message arrives, then execute the getValue functionality.
- The getValue functionality actually accesses the low level sensor machinery, retrieves the result and sends it back to the requesting ACE. It is worth remarking that since point-to-point messages are really sort of RMI invocation[2], the sending of the result back to the client is actually a simple return statement.

[2] See WP1 documentation for details

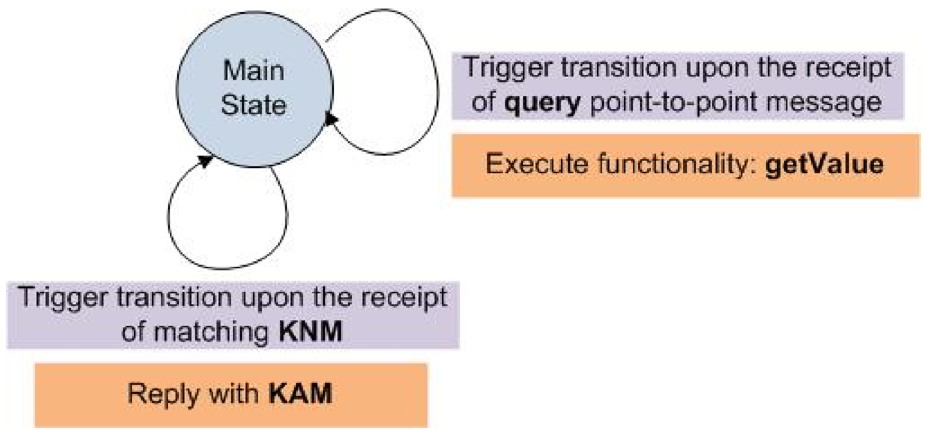

Main State
Trigger transition upon the receipt of **query** point-to-point message
Execute functionality: **getValue**
Trigger transition upon the receipt of matching **KNM**
Reply with **KAM**

Figure 19: Self-Model of a Knowledge Atom

6.5 Implementing Knowledge Containers

KC will be implemented according to the Self Model in Figure 20. The Self Model basically states:

- If receiving a KNM message that matches the kind of information the atom is able to produce, then reply with a KAM message describing such kind of information
- If a query point-point message arrives, then execute the queryAtoms functionality.
- The queryAtoms functionality actually sends a point-to-point query message to the atoms registered within this knowledge container (see point 4). The replies will be combined according to a KC-specific policy – to be implemented within the queryAtoms functionality – and the final result is returned to the enquired ACE.
- Every T seconds, the knowledge container looks for knowledge atoms (or other knowledge containers) able to provide information that are suitable to its need. It performs this by sending a KNM message specifying the kind of information it is willing to aggregate (i.e., contain).
- Upon the receipt of suitable KAM messages the container calls the storeAtomAddress functionality.
- The storeAtomAddress actually registers the atom in the container by storing its address in a suitable repository. Registered atoms will be queried upon the receipt of a query message (see point 2).

It is finally worth noticing that the above self models and functionalities are only a first attempt of mapping knowledge network functionalities into the ACE model. In future versions more advanced mapping taking into account more context-related information would be developed.

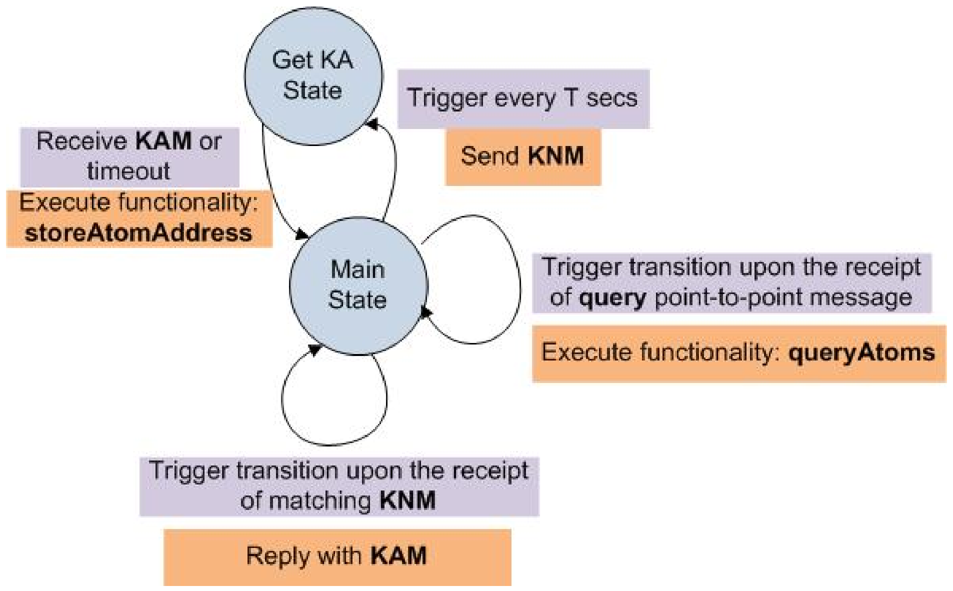

Get KA State
Trigger every T secs
Send KNM
Receive KAM or timeout
Execute functionality: storeAtomAddress
Main State
Trigger transition upon the receipt of query point-to-point message
Execute functionality: queryAtoms
Trigger transition upon the receipt of matching KNM
Reply with KAM

Figure 20: Self-Model of a Knowledge Container

7 Research Directions

Obviously, one of the main objectives for the reminder of the project is the integration of the developed concepts, prototypes and algorithms into the overall ACE framework (as described in Section 6) as well as synchronising individual aspects between work packages towards a common demonstrator.

Internally, WP5 will concentrate on the following aspects for the next 6 month and beyond (which to most extent correspond to what already stated in the roadmap of [D5.1] and in the document of worl).

- **Evaluation and refinement** of existing components and algorithms;
- **Add-on Services:** As described earlier on KN’s are service oriented in a way that specific solutions may be integrated into the KN toolset itself thus realising specific functionality. From the discussion so far, it is clear that the amount of information relevant for contextual knowledge and reasoning thereof are characterized to be rather large, unstructured, unrelated and possible redundant. This calls for advanced knowledge lifecycle mechanisms and for mechanisms to ensure the consistency of sensed and newly derived knowledge. With respect to the former, the issue is to evaluate how long the information should be held and how much of its history should be stored for future use and for predictive features. Simplified, it is the possibility of a system to “forget” things which is proven to be often more difficult than making a system learn. With respect to the latter, the issue is to evaluate discrepancies for inconsistent and incomplete knowledge, and how to measure reliability and accuracy of information.

- **Context Verification:** Current context verification method as addressed by UniK will be further researched and implemented. We will develop a simulation environment capable of investigating the context verification system. A scenario within the scope of the overall CASCADAS scenario Pervasive Exhibitions will be created and will form the basis for experiments. User behaviour and sensor data will be simulated in order to map real world situations. All processed information will be handled via knowledge networks. Multiple experiments are intended to be performed to investigate the behaviour and to prove the correctness of the context verification system. Our work can be further extended by investigating additional aspects and properties with the help of experiments. One more question that could be worked out is how to correct errors once they are detected.
- **Situation awareness:** Applications and services need to take advantage from knowledge organization along different dimensions e.g. the semantic and temporal dimensions, or along additional application-specific dimensions. This calls for more advanced algorithms that, ideally, can operate on multiple dimensions thus providing situational knowledge derived from multiple contextual levels. From the semantic viewpoint, it is necessary to integrate self-organization algorithms that enable to discover and enact relations among initially uncorrelated knowledge atoms. From the emerging network of such relations, it may then be possible to acquire new knowledge about facts and situations, which could be made available via knowledge containers. From the temporal viewpoint, the basic idea is that the analysis (both spatial and semantic) of contextual information about the past can be used to infer information about the future. For instance, the analysis of the fact that a visitor at the exhibition has already visited specific sections of an exhibition can be used not only to increase the accuracy of its profile but also to reasonably predict what sections/events in the exhibition he is most likely to visit next. Accordingly, one can tune the information displayed on the screens close to her/him.
- **Advanced Knowledge Execution and Optimisation:** Bio-inspired mechanisms, which incorporate concepts such as stigmergy, swarm intelligence and heartbeat signals offer new perspectives that are relevant for the optimization and adaptation of structures that are complex in nature, highly distributed and often unrelated with each other or with the domain /situation they are used for. In theory, they enable distinct optimization without the requirement for an intrinsic stimulus, which allows for truly distributed self-mechanisms. Furthermore, introducing autonomous features into systems where selective adaptation and control mechanisms are difficult to implement makes such systems not only more robust and reliable but often they make them usable in the first place. The main objective for this type of work is to research such mechanisms in support of autonomic principles. In particular the utilization of stigmergy to enable self-organization, self-optimization and self-contextualization will be analysed. The use of stigmergic overlay networks will be explored for optimizing knowledge querying, enabling reasoning over knowledge structures and for constructing and utilizing usage pattern thereof

Appendix A – User Guide

Knowledge Networks

How to Create Your Own Atom Realisation

Version 0.1

A.1 Knowledge Atom

Knowledge Atoms (KA) represent the generic data access layer for Knowledge Networks (KN). Representing the most basic component of a knowledge network, the rough structure of a knowledge atom is depicted in Figure 0. It contains a knowledge source object and relevant descriptions that provide the context of the data or knowledge represented by it. The sole purpose of a knowledge atom is to provide generic access to a specific data source and to allow the registration thereof into the scope of a knowledge network. It is not concerned with any organisational aspects within or outside the knowledge network nor is it responsible for the configuration, maintenance or (de-) registration thereof. To be more specific, a Knowledge Atom provides access to data and knowledge, it is not concerned with the sensing, construction, maintenance or organisation thereof.

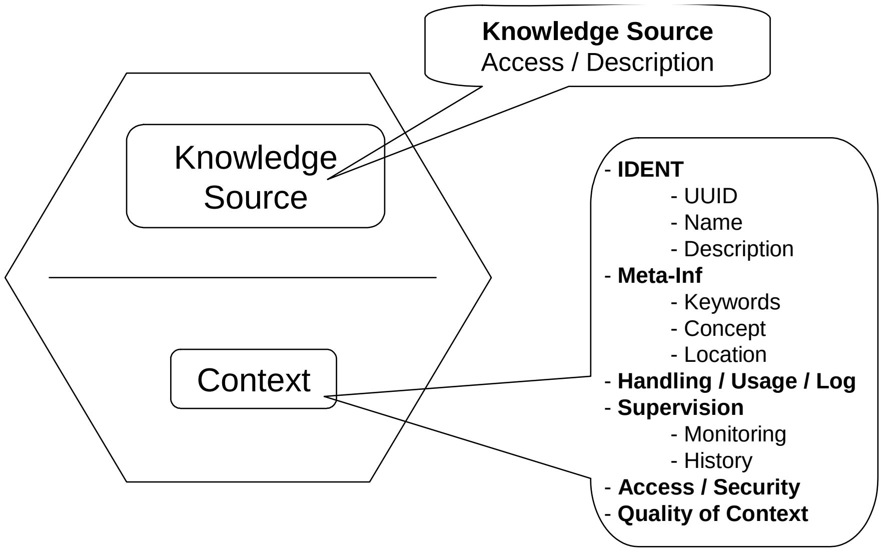


Figure 21: Knowledge Atom

Simplified, KA's may be grouped into two distinct types, namely embedded and remote atoms. While these may appear to be the same to the outside, the way data is provided differs. From a knowledge network point of view, an embedded KA stores the data it serves locally (inside the network) while a remote KA provides its data based on the concept that is behind its final realisation. In any case, for the latter, data only exists wherever they are produced. If not for other reasons they do not appear in any way inside the KN.

A.2 Embedded Knowledge Atom

The rational for this type of atom implementation is to serve data that is of a (semi-)static nature. That is, it should be non-volatile and relatively small in size. As the name suggests, relevant data is directly included within the atom's runtime object (inside the KN) and as such readily available on request. Some good examples of such data include policies, rules, locations, etc. While these atoms may be altered, referenced or cloned as desired, a sophisticated master data manager may be desired to preserve the original instance thereof. To allow for the dynamic handling and provision of embedded atom instances, a distinct java class has been complied which conforms to the generic Atom specification. An XML construct, as shown below may be used to configure individual atom instances with specific data. The construct provides distinct parts that relate to the value and type of the data to be served, optional configuration parameters as well as a section that is specific to knowledge networks. Although the size and complexity of the “Value” section is not limited, it should be kept within reasonable limits and instantiations of relevant objects may be refused if the size is outside the capabilities of the hosting resource. The overall atom itself may be further configured and registered via services as described later on.

```
<Data>
    <Value>Atom value</Value> // Compulsory
    <Type>Data type of the atom value</Type> // Compulsory
    <Config>Configuration of the atom source</Config> // Optional
    <KNModel>Data specific to the concept of Knowledge Networks</KNModel> // Optional
</Data>
```

XML Block 1: XML Specification - Embedded Atom

A.3 Remote Knowledge Atom

As the name suggests, remote atoms provide data that is not contained within their own logic or memory space. The rational for this type of atom implementation is to enable access to data stemming form various concepts, e.g. sensors, databases, software modules, web page, etc. In this case the data source itself exposes the atom interface or any other interface deemed necessary. Access to the recourse is then facilitated using a specific atom realisation to be provided by the vendor or user which may be registered into the scope of a KN. Due to the fact that the underlying data will be, at runtime, requested from the underlying source, this type of atom realisation may be used to serve volatile data independent of its size, complexity, location or context. The Atom instance may be loaded and linked directly (specific realisation) to the data source or may be hosted remotely (data source exposes the atom interface directly). A knowledge requester will only communicate with the interface and is indifferent to the location of the actual data or specific access mechanisms. Because of the fact that the underlying data represents volatile concepts, it may not be altered or cloned from the outside. However, it may be referenced as desired to allow for maximum access optimisation during internal network knowledge discovery.

```
<Service>
    <Name>Name of the Atom Realisation </Name>
    <Description>Detailed Description</Description> // Optional
    <URI>URI where relevant class files may be loaded from</URI> // may be null for internal classes
    <ClassName>Class name to be instantiated</ClassName>
    <Parameters> // Optional
        <!--Constructer Paramenters-->
        <Parameter>
            <Name>Name of the Parameter</Name>
            <Type>Type of the Parameter</Type>
            <Value>Value of the Parameter</Value>
        </Parameter>
    </Parameters>
<Services>
```

XML Block 2: XML Specification - Remote Atom

To provide for distinct remote atom realisations, dedicated proxy classes have been compiled which can be used as a base class for specific implementations. Alternatively, individual realisations may implement the atom interface, as shown in Code Block 1, or simply conform to the specification of the atom interface without actually implementing it. In this case access to the methods is realised via reflection. Specific atom realisations may finally be used to instantiate the generic atom proxy within the knowledge network. An XML construct, as shown below may be used to configure the proxy object advising on what atom realisations may be loaded and how they should be configured.

```
public interface Atom {
   /**
    * Convenience method to test if the Atom is available.
    * @return true if the Atom is available, otherwise false;
    */
   public boolean isAlive();

   /**
    * This method provides an XML based description about the value of the Atom and its type.
    * The structure of the XML object has yet to be specified.
    * params can be used to pass relevant parameters.
    * @return The Atoms value following the following xml format.
    *
    * <PRE>{@code
    * <Data>
    *      <Value>
    *          The Value of the Atom
    *      </Value>
    *      <Type>
    *          The Type of the Atom
    *      </Type>
    * </Data>
    *}</PRE>
    */
   public Element getValue(Vector params);
   /**
    * This method provides an XML based description about the type of the value the Atom provides.
    * The structure of the XML object has yet to be specified.
    * @return The Atoms type  following the following xml format.
    * <PRE>{@code
    * <Type>
    *      The Type of the Atom
    * </Type>
    *}</PRE>
    */
   public Element getType();
   /**
    * This method provides an XML based description about the Atom source (not its value).
    * The structure of the XML object has yet to be specified.
    * @return A description about the Atom source  following the following xml format.
    *
    * <PRE>{@code
    * <Config>
    *      This Section is open and may differ for each Atom.
    *      Standardised content may be defined at a later stage
    * </Config>
    *}</PRE>
    */
   public Element getConfig();
}
```

Code Block 1: Atom Interface

Currently, four specific realisations have been compiled and may be used as base classes. These are (a) an implementation to serve embedded data as described previously (EmbeddedAtomImpl), (b) an RPC solution that connects to RPC based web servers (RPCAtomImpl), (c) a service that accesses specific and publicly available web services (WSAtomImpl) and (d) a sensor implementation which allows access to specific types of MicaMots. The overall atom itself may be further configured and registered as described later on. Within the context of Cascadas, it is envisioned that all but (a) will be replaced by a specific ACE solution which will serve as a generic component where data is gathered from. Nevertheless, other solutions may also be catered for in order to provide access to legacy systems.

A.4 Atom Registration

Within the current RPC based solution as well as within the overall ACE model, atoms represent services (or jobs in DIET terms) which may be configured using the XML construct shown below. The atom service itself may be added or registered to a running RPC based knowledge network by calling "addComponent(Element knComponent)" on the respective network instance.

```
<KNComponent>
    <Ident>
        <UUID>Unique Identifier</UUID>
        <Type>Atom</Type>
        <AccessInfo>
            <URI>URI specifying the location of the Atom Instance</URI>
        </AccessInfo>
    </Ident>
    <Atom>
        <Service>
            <!--Atom Realisation as specified earlier-->
        </Service>
    </Atom>
    <Service>
        <!--Additional Services that may be loaded and linked to the Atom-->
        <!–currently, this should only be used within WP5-->
    </Service>
</KNComponent>
```

XML Block 3: Atom Service Specification

While embedded atom realisations may be loaded as well as registered to the scope of a knowledge network, remote atoms should only be registered. The atom interface itself should be linked to the data source itself or if possible, embedded therein. This would allow the fully distributed and lightweight construction of network like structures at a later stage. However, to allow for simulated data sources, distinct atom realisations (remote atoms) may also be dynamically loaded. This functionality however will be depreciated at a later stage.

A.5 Atom Add-On Services

One of the objectives was to keep individual atom implementations as lightweight as possible. Because of this, the basic atom realisation will only have a basic functionality which is only relevant to the provision of data itself. To make the concept of atoms as a generic access layer for knowledge networks completely generic however, there will be times when extra functionality will need to be added to satisfy the user requirements. Thus individual atoms may be orchestrated by (un-)loading and linking other services to the overall atom instance. This can be done at initialisation as well as at runtime. In both cases individual services have to be provided as executable code and also have to be described via XML conforming to the structure shown below. Using this construct a service can be added by simply calling the addComponent method on the respective atom object, passing the XML description as a parameter. Note that this can be done directly or via RPC. Respective methods may then be explored and executed utilising the service explorer and execution manager provided by the RPC framework.

```
<Services>
    <!--Extra services the component can provided as added components-->
    <Service>
        <Name>The name of a service this component provides as an add-on</Name>
        <Description>Sementic description of the service</Description>
        <URI>The address of the service, can be null for a local service</URI>
        <ClassName>The Java class name of the service object</ClassName>
        <Parameters>
            <!--Intitialisation parameters of the service-->
            <Parameter>A single parameter for the method<Name>The parameter name</Name>
                <Type>The parameter type</Type>
                <Value>The parameter value</Value>
            </Parameter>
        </Parameters>
        <Methods>
            <!--A list of methods for the service-->
            <Method>
                <!--A single method specification-->
                <Name>The methods name</Name>
                <Description>Sementic description of the method</Description>
                <Return>The return type of the method </Return>
                <Parameters>
                    <!--A list of parameters for the method-->
                    <Parameter>A single parameter for the method<Name>The parameter name</Name>
                        <Type>The parameter type</Type>
                        <Value>The parameter value</Value>
                    </Parameter>
                </Parameters>
            </Method>
        </Methods>
    </Service>
</Servies>
```

XML Block 4: Atom Add-On Services Specification

Appendix B – Examples

B.1 Embedded Atom

```
<KNComponent>
    <Ident>
        <UUID>Embedded_Test_Atom</UUID>
        <Type>Atom</Type>
        <AccessInfo>
            <URI>http://Localhost:8888</URI>
        </AccessInfo>
    </Ident>
    <Atom>
        <Data>
            <Value>
                Our central objective with CASCADAS is to identify, develop, and evaluate a general-purpose
                abstraction for autonomic communication services, in which components autonomously achieve
                self-organisation and self-adaptation towards the provision of adaptive and situated
                communication- intensive services.  We will achieve this objective by developing a common
                abstraction, called an ACE (Autonomic Communication Element), which represents the
                cornerstone of our component model. We will also use four key, underpinning scientific principles
                in CASCADAS, which are situation awareness, semantic self-organisation, self-similarity, and
                autonomic componentware to help guide the project.
            </Value>
            <Type>java.lang.String</Type>
            <Config>Confidential</Config>
            <KNModel>
                <Keywords>
                    <Type>Sensor</Type>
                    <Key>Weather</Key>
                    <Key>Wind</Key>
                    <Key>Wind_Direction</Key>
                    <City>Berlin-Schoenefeld</City>
                    <Country>Germany</Country>
                </Keywords>
            </KNModel>
        </Data>
    </Atom>
</KNComponent>
```

B.2 RPC Atom and Add-On Service

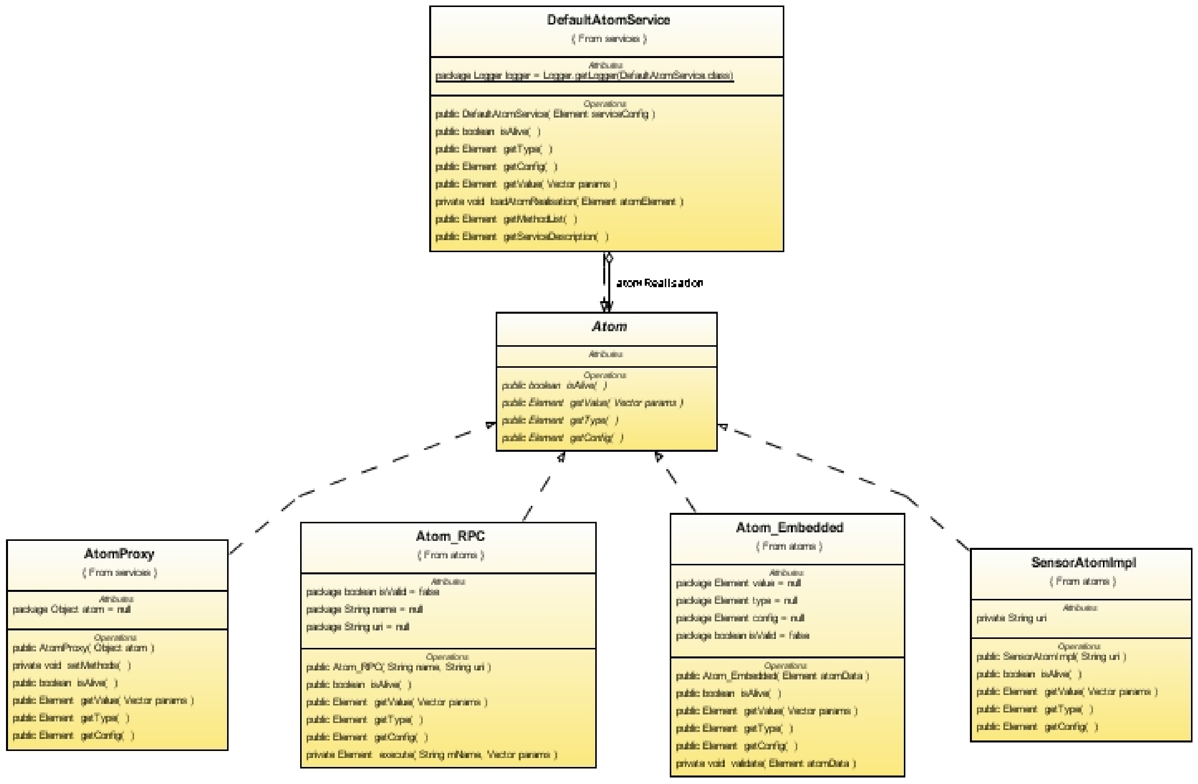
```
<KNComponent>
    <Ident>
        <UUID>Unique Identifier</UUID>
        <Type>Atom</Type>
        <AccessInfo>
            <URI>URI specifying the location of the Atom Instance</URI>
        </AccessInfo>
    </Ident>
    <Atom>
        <!--Atom Realisation as specified earlier-->
        <Service>
            <Name>RPC_Atom</Name>
            <Description>Atom service</Description>
            <URI>null</URI>
            <ClassName>org.cas.kn.impl.services.atoms.Atom_RPC</ClassName>
            <Parameters>
                <Parameter>
                    <Name>name</Name>
                    <Type>java.lang.String</Type>
                    <Value>Wind_Force</Value>
                </Parameter>
                <Parameter>
                    <Name>uri</Name>
                    <Type>java.lang.String</Type>
                    <Value>http://127.0.0.1:8888</Value>
                </Parameter>
            </Parameters>
        </Service>
    </Atom>
    <Service>
        <!--Additional Services that may be loaded and linked to the Atom-->
        <Name>MetaInf</Name>
        <Description>null</Description>
        <URI>null</URI>
        <ClassName>org.cas.kn.impl.services.DefaultMetaInfService</ClassName>
        <Parameters>
            <Parameter>
                <Name>ServiceConfig</Name>
                <Type>org.jdom.Element</Type>
                <Value>
                    <MetaInf>
                        <Description>
                            <Name>Wind_Force.Karlovy Vary.Czech Republic</Name>
                        </Description>
                        <Keywords>
                            <Type>Sensor</Type>
                            <Key>Weather</Key>
                            <Key>Wind</Key>
                            <Key>Wind_Force</Key>
                            <City>Karlovy Vary</City>
                            <Country>Czech Republic</Country>
                        </Keywords>
                    </MetaInf>
                </Value>
            </Parameter>
        </Parameters>
    </Service>
</ Service
```

B.3 Skeleton Class for Push Mechanism

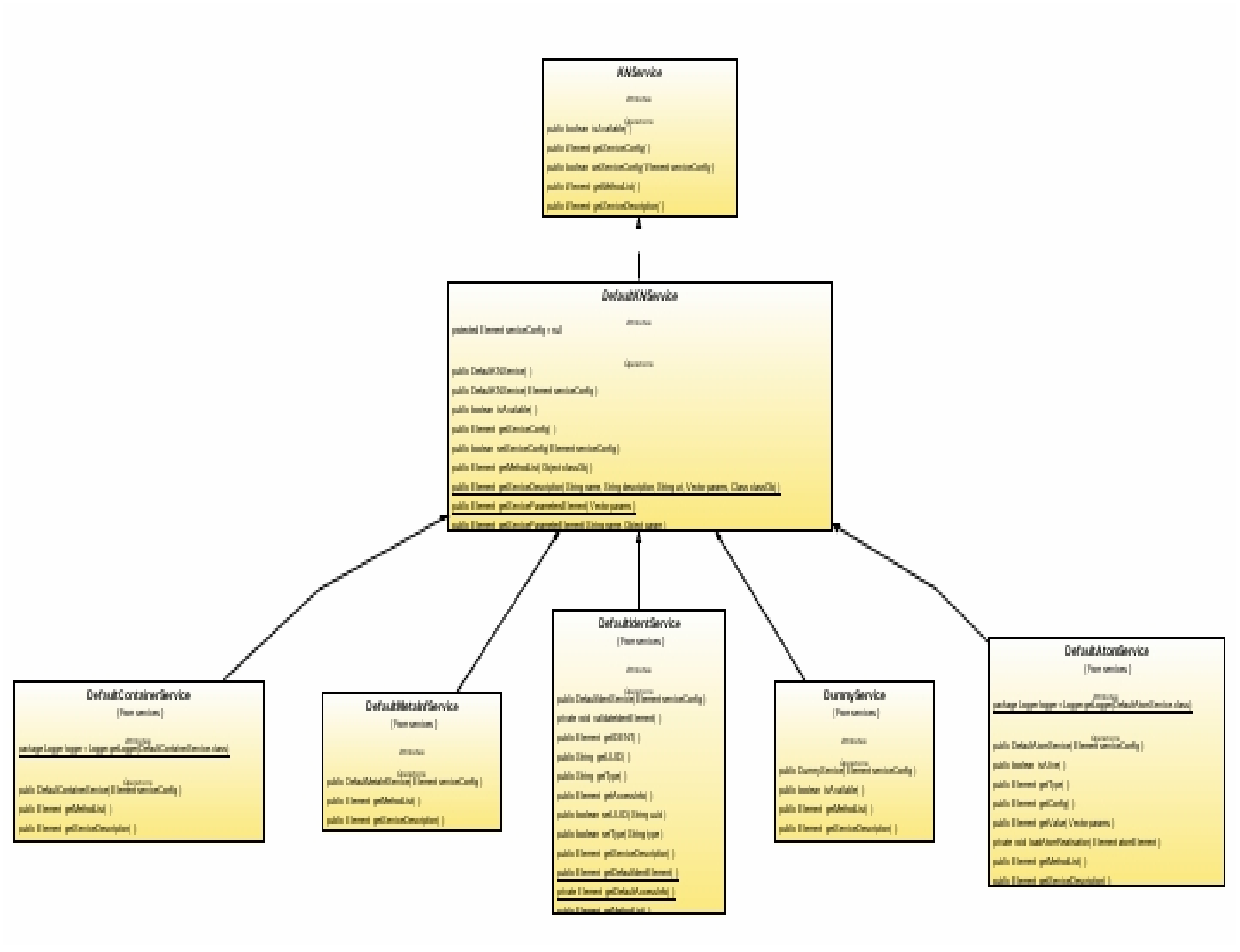
```
Class PushService {

   private Object lastValue;
   private long pollPeriod;
   private ArrayList receivers; //array of NotificationListeners

            public PushService(long pollPeriod, NotificationListener receiver) {
                        this.lastValue = null;
                        this.pollPeriod = pollPeriod;
                        receivers = new ArrayList();
                        start();
            }

            public registerListener(NotificationListener nl) {
                        receivers.add(nl);
            }

            public unregisterListener(NotificationListener nl) {
                        receivers.remove(nl);
            }

            public setPollPeriod(long pollPeriod) {
                        this.pollPeriod = pollPeriod;
            }

            public getPollPeriod() {
                        return pollPeriod;
            }

   private sendAll(Object currentValue) {
            int index;
      NotificationListener nl
            if (currentValue != lastValue) {
                        for (index = 0; index < receivers.size(); index++) {
                                    nl =  (NotificationListener) receivers.get(index);
                                    nl.notify(currentValue);
                        }
            }

            lastValue = currentValue;

   }

   private start() {
            sendAll(getValue());
            schedule(start(), pollPeriod);
   }
}
```

Appendix C – Class Structures

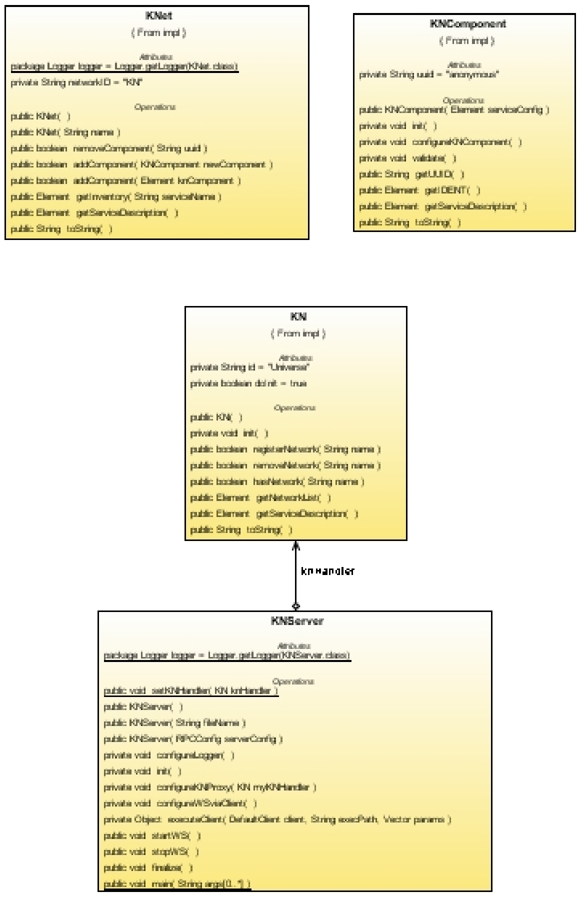
C.1 Atom


DefaultAtomService
(From services)
Attributes
package Logger logger = Logger.getLogger(DefaultAtomService.class)
Operations
public DefaultAtomService(Element serviceConfig)
public boolean isAlive( )
public Element getType( )
public Element getConfig( )
public Element getValue(Vector params)
private void loadAtomRealisation(Element atomElement)
public Element getMethodList( )
public Element getServiceDescription( )
atomRealisation
Atom
Attributes
Operations
public boolean isAlive( )
public Element getValue(Vector params)
public Element getType( )
public Element getConfig( )
AtomProxy
(From services)
Attributes
package Object atom = null
Operations
public AtomProxy(Object atom)
private void setMethods( )
public boolean isAlive( )
public Element getValue(Vector params)
public Element getType( )
public Element getConfig( )
Atom_RPC
(From atoms)
Attributes
package boolean isValid = false
package String name = null
package String uri = null
Operations
public Atom_RPC(String name, String uri)
public boolean isAlive( )
public Element getValue(Vector params)
public Element getType( )
public Element getConfig( )
private Element execute(String mName, Vector params)
Atom_Embedded
(From atoms)
Attributes
package Element value = null
package Element type = null
package Element config = null
package boolean isValid = false
Operations
public Atom_Embedded(Element atomData)
public boolean isAlive( )
public Element getValue(Vector params)
public Element getType( )
public Element getConfig( )
private void validate(Element atomData)
SensorAtomImpl
(From atoms)
Attributes
private String uri
Operations
public SensorAtomImpl(String uri)
public boolean isAlive( )
public Element getValue(Vector params)
public Element getType( )
public Element getConfig( )

C.2 KN Service


KNService
Attributes
Operations
public boolean isAvailable( )
public Element getServiceConfig( )
public boolean setServiceConfig(Element serviceConfig)
public Element getMethodList( )
public Element getServiceDescription( )
DefaultKNService
Attributes
protected Element serviceConfig = null
Operations
public DefaultKNService( )
public DefaultKNService(Element serviceConfig)
public boolean isAvailable( )
public Element getServiceConfig( )
public boolean setServiceConfig(Element serviceConfig)
public Element getMethodList(Object classObj)
public Element getServiceDescription(String name, String description, String url, Vector params, Class classObj)
public Element getServiceParameters(Vector params)
public Element getServiceParameter(String name, Object usage)
DefaultContainerService
{ From services }
Attributes
package Logger logger = Logger.getLogger(DefaultContainerService.class)
Operations
public DefaultContainerService(Element serviceConfig)
public Element getMethodList( )
public Element getServiceDescription( )
DefaultMetainfService
{ From services }
Attributes
Operations
public DefaultMetainfService(Element serviceConfig)
public Element getMethodList( )
public Element getServiceDescription( )
DefaultIdentService
{ From services }
Attributes
Operations
public DefaultIdentService(Element serviceConfig)
private void validateIdent(Element)
public Element getIDENT( )
public String getUUID( )
public String getType( )
public Element getAccessInfo( )
public boolean setUUID(String uuid)
public boolean setType(String type)
public Element getServiceDescription( )
public Element getDefaultIdentElement( )
private Element getDefaultAccessInfo( )
public Element getMethodList( )
DummyService
{ From services }
Attributes
Operations
public DummyService(Element serviceConfig)
public boolean isAvailable( )
public Element getMethodList( )
public Element getServiceDescription( )
DefaultAtomService
{ From services }
Attributes
package Logger logger = Logger.getLogger(DefaultAtomService.class)
Operations
public DefaultAtomService(Element serviceConfig)
public boolean isAlive( )
public Element getType( )
public Element getConfig( )
public Element getValue(Vector params)
private void loadAtomRealisation(Element atomElement)
public Element getMethodList( )
public Element getServiceDescription( )

C.3 Overall Framework


KNet
(From impl)
Attributes
package Logger logger = Logger.getLogger(KNet.class)
private String networkID = "KN"
Operations
public KNet( )
public KNet(String name)
public boolean removeComponent(String uuid)
public boolean addComponent(KNComponent newComponent)
public boolean addComponent(Element knComponent)
public Element getInventory(String serviceName)
public Element getServiceDescription( )
public String toString( )
KNComponent
(From impl)
Attributes
private String uuid = "anonymous"
Operations
public KNComponent(Element serviceConfig)
private void init( )
private void configureKNComponent( )
private void validate( )
public String getUUID( )
public Element getIDENT( )
public Element getServiceDescription( )
public String toString( )
KN
(From impl)
Attributes
private String id = "Universe"
private boolean doInit = true
Operations
public KN( )
private void init( )
public boolean registerNetwork(String name)
public boolean removeNetwork(String name)
public boolean hasNetwork(String name)
public Element getNetworkList( )
public Element getServiceDescription( )
public String toString( )
knHandler
KNServer
Attributes
package Logger logger = Logger.getLogger(KNServer.class)
Operations
public void setKNHandler(KN knHandler)
public KNServer( )
public KNServer(String fileName)
public KNServer(RPCConfig serverConfig)
private void configureLogger( )
private void init( )
private void configureKNProxy(KN myKNHandler)
private void configureWSviaClient( )
private Object executeClient(DefaultClient client, String execPath, Vector params)
public void startWS( )
public void stopWS( )
public void finalize( )
public void main(String args[0..*])