Communities: Creating and Quering Ad-hoc Databases based on Concepts.[1]

Chara Skouteli*, George Samaras*, Christoforos Panayiotou*, Evaggelia Pitoura+
Department of Computer Science, University of Cyprus*
CY-1678 Nicosia, Cyprus
{chara, cssamara, cs95gp1}@ucy.ac.cy
Department of Computer Science, University of Ioannina+
Ioannina, Greece, Greece

Abstract. Today, the largest amount of data is not stored in traditional database systems. Instead, a significant amount of this data is stored in personal computers, in the various types of PDAs and lately in the so called smartphones. The ability to network these devices creates, what we called, a "global" or "universal" database. This database is characterized by an enormous volume of information, high distribution, and high degree of heterogeneity. The vision, and the theme of this paper, is for the user to have the ability to effectively query such a database, taking into consideration the mobility of the user and the device. In order to face heterogeneity, and at the same time maintain and respect the autonomy of the various nodes, we assume that the querying and accessing of this data is done via mobile web services. What we proposed in this work, is the semantic grouping of these services over this global computational net. The main idea is to create a dynamic overlay network –like the peer-to-peer systems– that connects and groups semantically related services, effectively creating a network of "communities" and have the user querying instead this network. The definition of what services belong to a particular community (i.e., what services are semantically similar) is build around the notion of a "concept" which is either given a priori, during or derived from the definition of the service. Concept is a semantic notion and describes a specific property, for example, "travelling", "weather", "taxi reservation" etc. Context based queries, containment and continuous queries are also of special interest within this new idea of communities and concepts and we attempt to study them within this paper.

1. Introduction

The plethora of today's data sources in combination with the increasing number of mobile devices along with the progress of wireless networking have enabled a new kind of data-offering services. This new kind of services is characterised by mobility, locality and variety. The first characteristic –mobility– comes from the fact that the users, as well as many data sources are themselves mobile. An example of such a mobile service would be the provision of local weather data by a sensor attached to a mobile device (e.g. PDA). On the other end, the data consumer (a.k.a the user), is interested only to the local to him data – making locality a characteristic of these services. To fully understand the need for locality we must examine the mobile user and his data needs. Doing so we see that when a user is on the go he's more concerned for data on his whereabouts (e.g. Where can I stop to eat? Where's the nearest ATM?) or his destination (e.g. Is it raining at where I'm going? Will I find a good hotel there?). The last characteristic of this new era of

[1] This work is partially funded by the Information Society Technologies programme of the European Commission under the IST-2001-32645 DBGlobe and IST-2001-35495 MEMO projects.

information offering services –variety– derives from the vast number of different types of available data and information. Furthermore the same data can be interpreted and presented in many different ways adding to the "variety factor". The problem, however, is that these characteristics make the retrieval of useful information much more complex and difficult for the user.

So, having in mind these three characteristics, the need to design a system that hides the imposed complexity from the mobile user is formulated. The ideal would be that the mobile user is able to identify the closest to him mobile service provider (while still moving) and then select the service he wants to use. This leads to a twofold problem: first identify the close by services (and consequently mobile service providers) and then select the wanted service. The first aspect of the problem could be seen as a distributed data management problem: we have several (distributed) services that a) need to be described and b) need to be located. Having in place a mechanism to manage the description, indexing and querying of data and services as well as the discovery of services that are offered by mobile devices is the first step in solving this problem. Furthermore, such a mechanism should be implemented at the infrastructure level. In other words the cells that provide the network link to the mobile devices should also handle the previously stated data. In effect that means that we need to have a Cell Manager – a server that manages the services discovery and all related data for each mobile device within its coverage area. In order to present our approach we have designed a set of specifications, as well as produced an early prototype implementation of the Cell Manager infrastructure (essentially a group of interconnected Cell Managers). A short description of the Cell Manager infrastructure will be given at section 2.1.

Following the discovery of services the mobile user needs to select the services to use. However this is easier said than done. At the time being the mobile computing environment is still poor as far as resources such as display capabilities and processing power are concerned. So we must limit the user's possible service selections to the ones that are (strictly) relevant to his interests. Thus we need to know his preferences and to have a good description of the available services. As already discussed the Cell Managers support the description of mobile services. In effect they manage all the required data (user's, device's, service's profiles), allowing us to exploit these data in order to contain the user's choices to the relevant ones. All primary objects, users, devices, services and data, are captured in the form of an ontology, which fully describes them. The use of ontologies gives the opportunity to further manipulate these data by querying not only their location but also on their semantic concept (the concept that is used to describe them). Having this possibility enables us to contain the user's choices to a specific concept as well as to localize them. Utilizing the notion of concept we would like to introduce the notion of *context containment queries*. Context containment queries are like regular containment queries in the sense that their purpose is to contain the results of a query within the boundaries of a specific element. In our case this element is the context of the data, namely a specific concept (e.g., in case of services, the service type and description). So using context containment queries (i.e., queries around a concept) would allow the user to make an easier and fastest selection of mobile services.

However, the introduction of new users and services, the mobility of users and services, creates a dynamic and constantly changing environment that we must also take into account. We need to continually re-evaluate each user's context containment queries in order to be able to provide him with up-to-date, available and relevant to his needs services. It is thus, imperative to add continuity to the context containment queries. This

can be achieved by merging the notion of context containment queries and *continuous queries*. In addition to the temporal dimension, mobility adds the spatial dimension as well. In the mobile environment it is desirable (and some times imperative) to use the closest to the requestor mobile services/resources. The need to include the location concept within the context containment queries is a must. Using the location (i.e. distance of mobile objects) we can prioritise/sort the query's results.

Viewing the location as a semantic concept allows us to treat it the same way as any other concept. Location can be considered, in some extend, as a semantic concept that has only spatial characteristics. And as such it can be used in conjunction with any concept in a containment query. So viewing location as a minor concept or a parameter concept of the service's main semantic concept enables us to use the same mechanism for answering both context containment and location queries.

The challenge now is how to manipulate more efficiently the available metadata information in order to support these context containment and continuous queries. Towards this end we introduce the notion of data communities. A community can have any combination of spatial, temporal, or thematic characteristics that define the context (namely the concept) that the community revolves around. Vice versa the collection of data on mobile objects around a specific context (e.g., location, user/service/device characteristics, etc.) forms a data sharing community. For example, all primary objects (a PO can be a user, service or device which may or may not be mobile) that provide weather services for Athens belong to the "Athens weather" community. If a PO wants to know if it is raining in Athens, it will use the "Athens weather" community in order to find another PO that can answer its question. In essence forming and linking this communities, creates a dynamic overlay network that can be used in order to achieve our goals. In this paper we describe how to develop these communities and discuss why they are a more efficient way to support the above mentioned context containment and continuous queries.

2. Overview

The Cell Managers can be viewed as a distributed data management system infrastructure. The main objective of this infrastructure is to connect a number of devices and provide support for (i) describing, (ii) indexing and (iii) querying their data and services. Cell Managers adopt a service-oriented approach to capture data in order to be exchanged between the participant devices. As already stated one of our goals is to present a mechanism of creating and processing ad-hoc databases over the Cell Manager infrastructure. The objective of this mechanism is to support context continues and containment queries as more efficient as possible, considering problems like disconnection and device limitations. Towards this end, each Cell Manager provides an interface that enables the creation of a distributed context aware ad-hoc database system comprising of data provided by the registered to the Cell Managers mobile devices.

2.1 Cell Manager Infrastructure

POs are the main source of the data and resources to be handled. Their number and location may change over time; new POs may leave or enter in the perimeter of the Cell Managers (CM) infrastructure. The POs may be connected to the system and possibly to each other in order to exchange data through services. The interconnected CMs form the

necessary infrastructure to support the connection to the system and cooperation between the POs. The following paragraphs (briefly) present the basic components of the Cell Manager infrastructure, as well as the structure and functionality of POs and CMs.

2.1.1 Primary Objects. The primary providers of resources are either mobile (e.g. smart phone, PDA) or stationary (e.g. pc, workstation) devices called POs (Primary Objects). A PO has the following basic functionality (i) request and retrieve data, (ii) produce and share data, (iii) create and publish a service, (iv) communicate with an access point (a.k.a. a CM) and function as a source.

2.1.2 Cell Manager Specification. The Cell Manager is consisted by the following components (also see figure 2.1).

1. *Cell Observer,* which is responsible to listen to the cell for user messages. When a new device wants to connect to the system then it must discover the cell manager. In order to do that the device broadcasts initiating messages. The monitor retrieves these messages and replies to the device with the name/address of the cell manager.
2. *Device Location Manage.* In order to enable the creation of a location management mechanism each PO has a home location (which is the address of the CM that the device registered for the first time) and a current location which is the address of the CM that it currently belongs. The Device Location Mechanism is responsible i) to inform the appropriate home CM of a device when the PO enters its domain, and ii) to be aware of the location of all POs that have it as home CM. Because of object mobility the location management mechanism is a very important factor for the system efficiency. What happens when a PO moves to an other cell? Do we have to move its services to the next cell too or not? How often a device moves to a new cell? This kind of questions must be answered beforehand in order to adapt the location management algorithm. In our case we assume that the cell covers a small geographical domain and the POs moves fast from one cell to another and thus, it would be inefficient to re-register PO's services to the current CM. Our approach is to update only the location of a PO and when we need to find a local service we query the home CMs of all POs that are currently in the local cell.
3. *CM Directory,* a list of all the system's CMs. The CMs must be aware about the existence of all other CMs in order to i) allow the creation of a location management mechanism, and ii) pass on queries that can not be answered locally (by them). The CM directory is a set of CM ontologies which describes each CM. The CM ontology includes contact information for each CM.
4. *Data Store Manager,* which is responsible to maintain the database used to store the services', users' and devices' profiles. It is also able to query these data. Finally the Data Store Manager classifies all the captured data according to their concept.
5. *Community Directory,* a list of all the system's communities. The usage of communities will be discussed in next paragraphs.

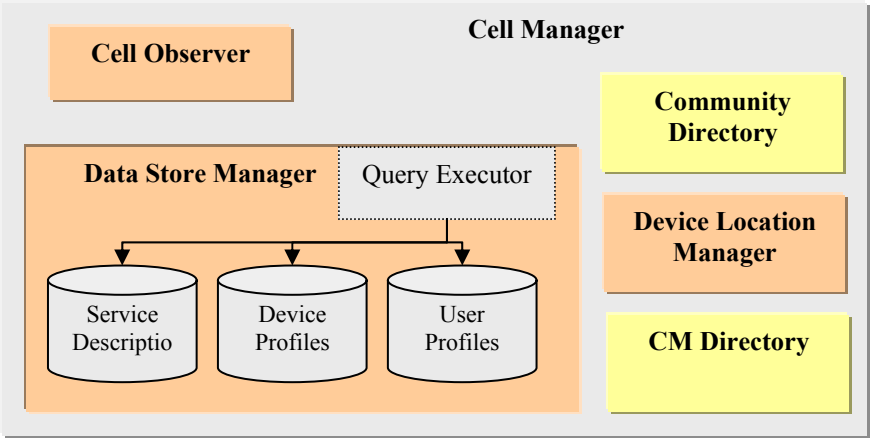

Figure 2.1: Cell Manager Architecture

2.1.3 Metadata management. The metadata management mechanism that is adopted is an important design issue for systems like this one. Firstly because it is responsible for the efficient data retrieval, and secondly because it affects the type of queries that the system can answer. In our case data stored in CMs can be divided to three major categories (i) content data which are the actual data registered by the user on the PO, (ii) profile data that describes the user and his/her devices as well and (iii) movement data to record the movement of a user or a device in a continuous basis. Essential metadata provide an abstract view of data and services, kept on a PO. All those types of metadata aim to give a classification and at the same time add semantic context to PO devices, users and services. CMs use XML to exchange this information, RDF Schemas to assign semantic context to structural elements and finally ontologies to denote hierarchies and relationships.

2.2 Context Awareness Queries

The increasing number of mobile devices creates the vision of a more sophisticated behaviour and interaction among them. Mobile devices may present various limitations, however they also present an opportunity for developing applications that are able to serve user needs while he is on the move. Such applications are mainly based on context information, e.g. user location, user preferences coupled with current activities, device characteristics etc. These applications need support by a distributed data management system infrastructure -in our case interconnected CMs- that will enable the execution of context awareness queries. By context awareness queries we denote the queries that (a) give context information in a contained fashion and (b) give results in a continues fashion. Thus we have the merger of two types of queries: context containment queries and continues queries.

2.2.1 Context containment queries. Basically context containment queries are an extended notion of regular containment queries. The goal here is contain the results of the query within the boundaries of a specific contextual concept. This of course means that the data that are queried are related to a specific concept, either directly or indirectly. To understand this lets view the case were we have a service for hotel bookings. The service

would be directly related to the hotel concept. To find the entire hotel services a regular containment query would suffice. However since the hotel concept is related to the tourism concept so would be the service, this time indirectly. Even further the same service could be related to the Asian concept if, for example, it covers only hotels in Asia. Thus another (regular) containment query that tries to retrieve information about Asia culture would also include this service to its results. As we can see, there is the need to give more accurate descriptions of the used concepts both as a query criterion and as a describing attribute of a service (or other data). In other words we need to define the context under which the containment query will run, and of course how this context will affect the produced result.

The first step towards defining the context of a concept is to give a more detailed description of it. Towards this end we can enlist the aid of ontologies. An ontology is used to fully describe an entity [6, 7, 8, 10]. However this is not enough in order to avoid the previously mentioned situation. To do so, we need a way to classify and interrelate the used ontologies. In order to achieve this classification and interrelation we can use taxonomies in which the elements are the aforementioned ontologies.

Such taxonomy would have the form of a tree in which the inner nodes contain an ontology that describes their children nodes. This means that each child of a node (with the exception of leaves) would contain another ontology, which describes its own children, but one that would be under the contextual umbrella of the parent node. Recursively this leads to a hierarchy of ontologies that with each (deeper) level of the hierarchy we have a more refined and accurate description of the enquired data (in our case these data are services). The leaves of the taxonomy tree contain the actual data (e.g. service details, location, requirements etc.).

Given a taxonomy we can run an inquiry for a specific topic by doing a top down search for matching ontologies within the taxonomy tree. During the search, the parent ontologies are used to narrow down the contextual domain of the children nodes resolving in this way the aforementioned problem where we are asking for information on Asian culture and we get hotel reservation services. This search is in effect what we call context containment query: a containment query that runs over a taxonomy that gives us contextual constrains.

2.2.2 Continues queries. A continual query is a standing query that monitors updates of interest using distributed triggers and notifies the user of changes whenever an update of interest reaches specified thresholds or some time limit is reached.

It is expressed in terms of a normal, SQL-like query, a trigger condition, and a stop condition. Some continual queries may also include a start condition and a notification condition (i.e., when the condition for notification of results is different from the trigger condition).

In our case, continuous queries come into play in our effort to keep the result of a concept based query up-to-date. This is indeed essential since services are quite mobile and new POs might join or leave the system dynamically.

3. Querying Cell Managers via Communities

CMs handle a plethora of data stored either on the POs or on themselves. Our objective is to exploit these data in order to (better) support continuous context

containment queries (context awareness queries). CMs store data that describe user's profiles, services and devices. These data are the core of the available context information. Because of the fact that CMs are geographically organized they are able to support location aware queries. However, context information is not limited just to location data but also includes user preferences, device characteristics and provided services, which is very important information as well adding significantly to the complexity and dynamicity of the system.

The problem now is how to handle or manage all these data. What happens when the user who provides a service gets disconnected or moves to another cell? What if the user who requested the service changes location? How a new user and the new services are dealt? This dynamic environment is more exigent than the classic mobile environments where only the requester can move around. Thus, we are in need for a fast and efficient way to find and query services that are available to POs. Using continuous queries that constantly update the list of the available services would solve this. However the high cost of the queries and scalability requirements make it extremely difficult to meet the needed level of efficiency or even achieve reasonable scalability. The problem becomes more intense considering that the queries asked are context containment queries because we need to minimize as much as possible the produced results. Thusly, the inquired services must be organized/indexed in the appropriate manner.

A good way to organize these services is by grouping them based on some common characteristics and then rerouting the incoming context queries to the appropriate group (which has a greatly reduced volume in comparison with the whole list of available services). In effect these groups form a "*community*": the citizens of the community are POs' ontologies of services or devices or people. All the citizens of the community share a common concept that they revolve around. E.g. the citizens of the weather community are weather forecast services.

Having the communities allows the incoming context awareness queries to run faster. As stated, our goal is to efficiently answer queries like the following "Find me all the services that provide weather forecasts and are near Athens." If we take the concept of the query to be "weather" we can easily utilize the weather community to efficiently answer it. The efficiency comes from the fact that communities are, after all, a collection of data pointing to the resources/data around a concept, just like a database index. *That bodes well with the overall spirit of creating and managing a very large, ad-hoc, metadata database*. The argument here would be that if we use communities as an index, why not use one regular, centralized and unified index? The answer to this question firstly lies to the scalability issue, as having one centralized index is not the optimal way to resolve it, in contrast with a distributed approach. The heterogeneity of the environment is another factor that makes the use of such an index cumbersome. A third factor is the semantic nature of the required index. We need an index that can give as the location of a service based on its semantic description, thus complicating the structure of the index. Finally, as the complexity of the index increases the cost of its updating becomes prohibiting. Communities on the other hand tackle all these problems with grace:

- The distributed setup of communities provides the much needed scalability.
- Heterogeneity in the environment doesn't affect communities as long as we use a standardised way of describing the available resources/data.
- Communities can, by design, provide semantic based indexing of resources/data.
- Updating a community (which has much fewer members than a unified index) is more cost effective. The updates are distributed to a number of communities (hosts), a fact that limits the load on each individual community.

3.1 Organizing Communities using Taxonomies

Just having a collection of communities doesn't entirely solve the problem. We also need to organize the communities. An efficient way to do this is to classify them with a taxonomy of describing ontologies (as discussed when presenting the notion of context containment queries), as shown in figures 3.1 and 3.2. Having this taxonomy (tree) enforces that each community will be interrelated with the other available communities. The interrelation is achieved by building the communities using concepts which are taken by the taxonomy tree. Note that the concepts are described by an ontology. Using an ontology to denote the concept of a community, we can have a more analytical description of its purpose, and in extend, what services should belong (be enlisted) to it. To this end, community concept denoting ontologies are comprised of keywords classified in tree form to describe the objective of the community. [9] used a similar approach to represent entities.

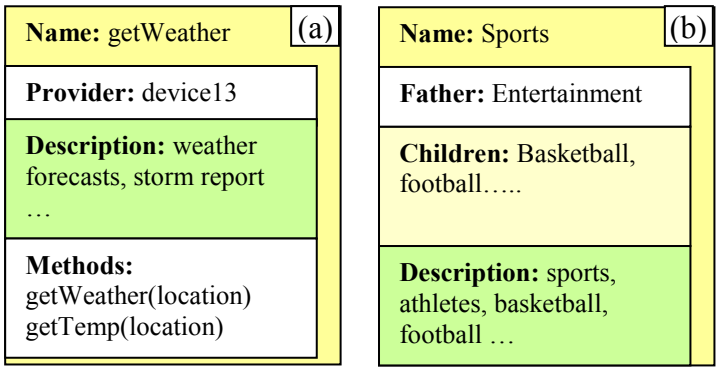

Figure 3.1: Service (a) and community (b) describing ontologies

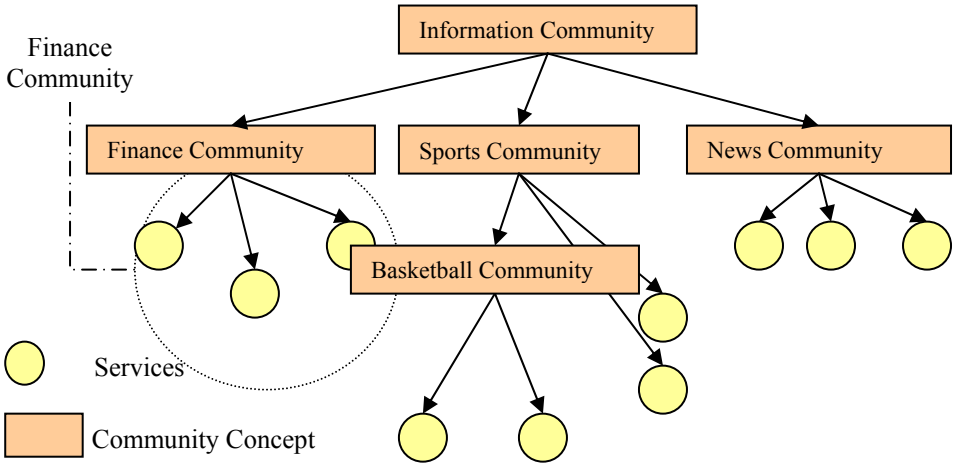

Figure 3.2: Taxonomy of communities

To implement these communities we introduce the notion of the Community Administrator Server (CoAS). CoASs are responsible for the creation and management of

the communities. Each Community Server covers a partial or the entire set of CMs, and each CM may be covered by many CoASs, in a dynamic manner. Who is covering who is decided in real-time according to the current location of the online resources/data (that move along with POs). In order to support this real-time operation, the CMs propagate to the relevant CoASs the service's describing ontology, when for example there is a change in the service's ontolology (e.g. a service changed its scope to a more specific one: from weather forecast to storm warning). Figure 3.3 shows a possible geographical distribution of the various CoASs. The complete set of CoASs constitutes a global taxonomy tree of communities. (Each CoAS serves one and only one community and vice versa). Also note that having in place these CoASs in essence, we form a dynamic overlay network that is above the low level network infrastructure but still able to answer queries referring to data sources that are connected to that low level infrastructure.

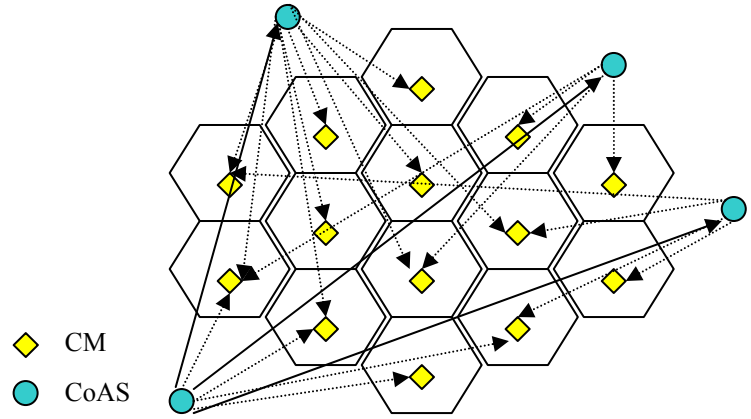

Figure 3.3: Geographical distribution of communities: a dynamic overlay network

3.2 Community Administrator Server (CoAS)

Community Administrator Servers (CoAS) are interconnected through a network, their major objective is to create communities of services, according to a concept (e.g. the type of the service, the type of the device that provides the service or the provider's profile). In other words, each CoAS has a contextual concept (described by an ontology) taken from a global taxonomy of concepts, which is used to build the relevant community. Essentially this means that a CoAS groups together data objects according to its specific concept, e.g. a community of all services which provide weather information.

3.2.1 CoAS architecture. Figure 3.4 illustrates the general architecture of a CoAS. In detail, the components that comprise a CoAS are the following:

i. *Service Ontology Directory* lists all service ontologies currently handled by the specific CoAS.
ii. *CoAS directory* lists children CoAS. This directory is necessary in the case where the specific CoAS is unable to satisfy an incoming query; the query will be forwarded to another CoAS. Listing the children-communities enables finding a finer grained community that can answer the current query in more detail. However if none of the children can answer it the query needs to be forwarded to the

parent/root community which may pass it down to its other children-communities or up to its parent/root community. The parent-children notion comes from the organization of communities in a tree (global taxonomy of CoASs and concepts).

iii. *Query executor* is the most important component of the CoAS, as it is responsible for matching an incoming query's criteria with service describing ontologies, including any location constraints. In effect, it is the context awareness query processor.

iv. *Concept Alerts Directory* is used to better support continuity for incoming queries by providing triggers for them. It keeps a set of context information that may change during time and the names of the user agents that wish to be informed in case of updates. Upon such an update all interested user agents are informed. E.g. a user agent wish to be informed whenever a new game service is available. A user agent can be any application that uses the functionalities of the system.

v. CM Directory lists all the CMs of the system. This directory maintains CM ontologies. These ontologies have information about the location of the respective CM as well as how to contact all other CMs. This directory is used when a new community is created. The community will inform all CMs for its existence in order to enable acceptance of service registrations from them.

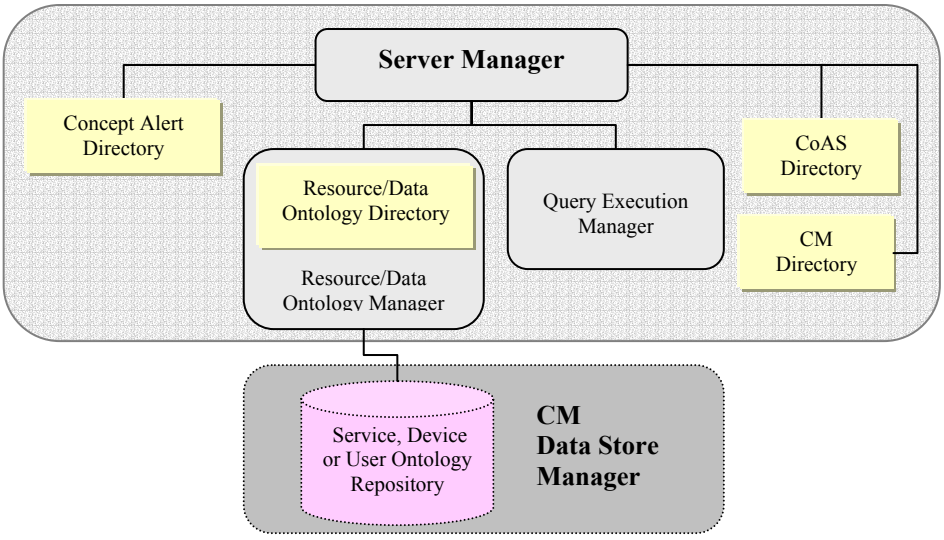

Figure 3.4: CoAS Architecture

3.2.2 Creating the Global Taxonomy Tree and the Communities. As mentioned in section 2.1.2 CMs classifies internally (and locally) the services registered to their cell according to their concept. We can use the classification of the CMs to start the creation of the global taxonomy tree. After creating the generic communities the root CoAS informs the CMs about the available communities. CMs must know the available communities in order to be able to forward incoming queries and service changes to the appropriate CoAS. CMs are the first point of contact with the actual POs, which are the information source, thusly putting them in a premium position for detecting updates in either, their location or provided services and/or profiles. It is easier (more efficient) to keep CoASs updated with service changes, changes to members of their community, if

they are notified by the relevant CASs for example. When a service does not belong to an existent community then the CAS will send its registration to the root CoAS, which is responsible to create a community with the service's concept. Upon the creation of the community a message will be broadcasted to all CASs informing them about the new community.

In case where a community becomes too large, reducing its efficiency, it can be split into two sub-communities. In order to decide what concept should be used to build the new community we cluster the existing community and then select the concept of the largest cluster. This update means a minimum amount of changes for the CoASs tree: only the children of the splitting CoAS and its parent need to be updated. On the other hand, as far as the CASs are concerned, the number of changes to be made is high: every CAS needs to update its copy of the CoAS tree. The update can be broadcasted on the DBGlobe network. To ensure though that during the updating user queries can be answered, a CAS can send the query to the parent CoAS if it fails to contact the wanted one. Keeping the rate of the CoASs splitting low ensures that a CM will always find a CoAS to redirect his incoming queries. CoASs, knowing their children, can further redirect incoming queries to them and so the query eventually will reach a CoAS where it can be satisfied. More details on how context awareness queries are satisfied are presented in paragraph 4.2.

3.2.3 Keeping Communities Updated. As already mentioned the system's environment is constantly changing with new POs coming online and others going offline or moving around. As these changes greatly affect the communities it is imperative to keep them updated. In order to keep them updated and keep their efficiency level intact, the responsibility of detecting the changes and notifying the relevant communities is pushed to the CMs. CMs are ideal for that job since they are the actual point of contact with the POs, and undoubtedly, are the first to detect any changes.

Before continuing our discussion on how the communities are kept updated we must first identify the type of the possible updates. There are four distinct types of updates each caused by a different kind of events:

1. The first kind of events covers the connection of new POs resulting in newly available resources/data. If a PO gets connected then it subscribes its services to the local CM. The CM is responsible to register those services with the appropriate CoAS. The CM can use, for example, its service taxonomy to somehow identify the appropriate CoAS (see section 3.2.4).
2. When a PO wishes to unsubscribe a given service, then the relevant CoAS must unsubscribe the service as well.
3. The third type of events occurs when POs get disconnected, rendering their services unavailable. The noticing CM must inform the relevant CoAS(s) of the temporarily unavailability of the PO's services.
4. Finally the last kind of update producing events is the detection of changes in the services' describing ontologies (changed characteristics). E.g. if a generic service previously provided by the given PO changes into a more specific one then it is possible that it should be member of another community. The CM has the responsibility of the service's migration from the, now irrelevant, community to the relevant one.

The next issue that needs to be resolved now is how CMs know which one is the appropriate CoAS for every update? Utilizing bloom filters [4, 5] in combination with the services' descriptions and the community concepts does the trick.

3.2.4 Using Bloom filters to Match Resources/Data to Communities (CoASs). Bloom filter summaries are a mechanism to efficiently identify the CoAS that may contain data relevant to an incoming query. Bloom filters are hash based indexing structures designed to support membership queries [5]. As CoASs are represented by ontology schemas, we need a method to query these schemas. Bloom filters are most likely to use a query language based on XPath [3] that allows us to exploit the structure of the schema as well as their content. Bloom filters as described in [5] are multi-level structures that support the efficient evaluation of path expressions including partial match and containment queries.

In detail, Bloom filters are compact data structures for probabilistic representation of a set that supports membership queries. Consider a set $A = \{a_1, a_2, \ldots, a_n\}$ of n elements. The idea is to allocate a vector v of m bits, initially all set to 0, and then choose k independent hash functions, $h_1, h_2, \ldots, h_k$, each with range 1 to m. For each element $a \in A$, the bits at positions $h_1(a), h_2(a), ..., h_k(a)$ in v are set to 1. A particular bit may be set to 1 many times. Given a query for b, we check the bits at positions $h_1(b), h_2(b), ..., h_k(b)$. If any of them is 0, then certainly b is not in the set A. Otherwise we conjecture that b is in the set although there is a certain probability that we are wrong. This is called a "false positive" and it is the payoff for Bloom filters' compactness. The parameter k and m should be chosen such that the probability of a false positive is acceptable.

Bloom filters are used to determine which CoAS should be updated when a new service is added or deleted from the system. They are also used to find which CoASs are relevant to a given query. In particular, given a service description, using the Bloom filters, we can efficiently locate the appropriate CoASs.

There is one multi-level Bloom filter for each CoAS that we call community Bloom filter. Let CBF(A) be the community Bloom filter that corresponds to CoAS A. To produce the CBF(A), the k hash functions are applied to all the concepts (keywords) that describe the community A. Given a service s, to find the CoAs that are associated with the service s, we apply the hash functions to each of the concepts (keywords) that describe the service. For each such concept of the service, we check which community Bloom filters match it. There is a match, if all bits at the corresponding positions of the community Bloom filter are set to 1. The communities that match the service s are the communities whose community Bloom Filters match all concepts describing the service.

3.3 Scenarios and Queries Utilizing the Communities

In this section we present use case scenarios of (i) how a new service is getting known by the CoASs and (ii) how a query that arrives to a CM is executed. Finally a list with the supported query types and their needs follows.

3.3.1 Registering a new service. When a PO logins to the system for the first time it needs to register its services with it. In order to make the idea of communities to work, the newly available services must also register to a CoAS. The process of the registration is as follows:

1. The PO logins for the first time to the system. The CM that the PO connected is considered its "home location". As such it has the responsibility to point to the real location of the PO while it is moving around (figure 3.5: 1).
2. Just after the first login the PO registers its services to the connected CM. The CM gets the services', user's and device's ontology schemas and stores them to his local database (figure 3.5: 2).

3. Upon the completion of the registration the CM utilizes Bloom filters (figure 3.5: 3) in order to find the community (or communities) that the given service belongs to. Having found the appropriate community, it forwards to it the resource's describing ontology. In case that there is more than one community that can accept the resource as its member, the ontology is forwarded to all of them. Step 3 is repeated for each and every one of the new PO's resources.
4. When a CoAS receives a new resource/data describing ontology it checks if it should be accepted as a member of its community. This is necessary as Bloom filters are not entirely accurate and can point to irrelevant CoASs along with the relevant ones. If the resource/data is not accepted as member it is simply ignored and the process for that specific CoAS is terminated.
5. The accepted resources/data are registered in the CoASs by means of storing their describing ontology in their directory (figure 3.5: 5). Along with the describing ontology, the "home location" CM is also stored. Doing so enables CoASs to satisfy location criteria of incoming queries by asking the "home location" CM for the service's current whereabouts. It also has the advantage of eliminating the need to keep all the CoASs updated with the movement of their member resources (and in effect their containing POs). Thus a scalability and efficiency pitfall due to the expensive and continues updating is avoided.
6. The final step of the process is to notify the concept alert directory (figure 3.5: 6) about the new member of the community. The concept alert directory will check for any pending queries that might be interested in the new member and it will trigger them.

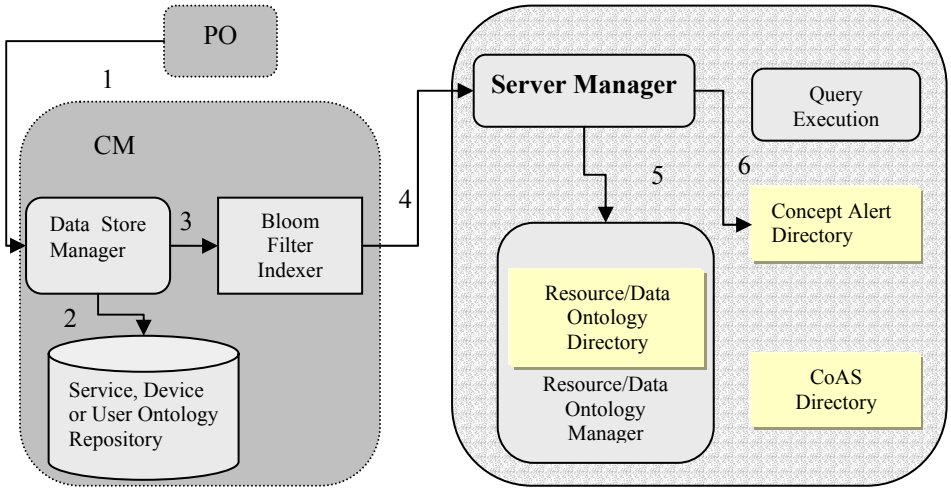

Figure 3.5: Login Use Case Scenario

3.3.2 Servicing a Query. From time to time the various POs, services and users need to ask queries to locate/discover the available services. The general procedure for servicing such a query is as follows:

1. A PO, service or user sends a request/query to its currently connected CM (figure 3.6: 1). The general form of a supported query is a request to find a particular resource/data. To make the scenario more understandable for the rest of the scenario we will use the following request: "Find a Nearby Restaurant".

2. If the receiving CM can satisfy the query (figure3.6: 2.1) then it returns the results to the issuing PO, service or user and the process terminates (figure 3.6: 2.2). For our example, the CM could satisfy the query if there is any "restaurant service" currently connected to it.
3. In case that the incoming query cannot be satisfied by the receiving CM, then it must be forwarded to the appropriate CoAS. To find out who is the appropriate CoAS, CMs utilize Bloom filters. In our example the CM will check the Bloom Filter Indexer (figure 3.6: 3.1) for communities on the concept "food services". Upon finding the appropriate CoAS, the CAS will forward the query to it (figure 3.6: 3.2).
4. The Server Manager of a CoAS upon receiving a query sends it to the Query Execution Manager (figure 3.6: 4.1). The query manager in cooperation with the Service Ontology Manager (figure 3.6: 4.2) will find all the matching services. The matching is done on a semantics level. All the matching services are marked in a list. For the example query, the resulted list will contain all the available restaurant services currently registered in the CoAS unless other constrains are also imposed.
5. If there are any location constrains then the resulting list of step 4 will be queried to satisfy them. In order to do so, first the service ontology manager will retrieve the services current locations from the "home location" CMs (figure 3.6: 5). After this retrieval, the query execution manager will apply all the location constrains. To relate with our example, a range query will be run on the service list to limit it to the nearby services, of the CM from which the query originated.
6. The resulted list is the answer to the incoming query and it will be relayed back to the issuing PO, service or user through the CoAS's service manager (figure 3.6: 6).

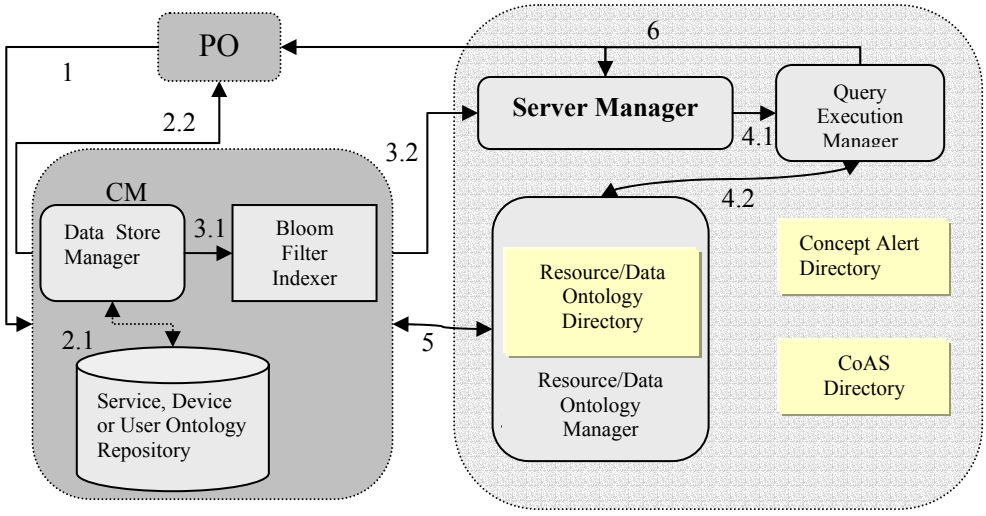

Figure 3.6: Querying use case scenario

3.3.3 Examples of Supported Queries. The queries that are supported by CoASs are divided into two major categories. In the first category we have queries that are run once and in the second category we have standing queries that need to be revaluated from time

to time. These two categories can be further broken down according to if the issued queries include location criteria or not. The location criteria, however, supported is the same for both categories.

Before continuing, note that all queries support by CoASs are forwarded to them by a CM. Thus, the term "forwarding CM" means, the CM that received an incoming query from a PO, service or user and forwarded to a specific CoAS. To make our presentation simpler note that the term issuer will be used to denote the PO, service or user that posed a query.

The first category includes the following types of queries:

- *Find all services that provide weather forecasts.* The query can be easily answered by the weather community; we just need to forward it to the weather community. In a SQL like language the above query would look like this:

```
SELECT * FROM SERVICES WHERE type LIKE 'weather forecasts'
```

- *Find all the restaurant services that do not offer a vegetarian menu*. To answer this query we can retrieve all the members of the restaurants community and then exclude the members that also belong to the vegetarian community. Again in a SQL dialect the query would look like this:

```
SELECT * FROM SERVICES WHERE type LIKE 'restaurants' AND NOT IN
(SELECT * FROM SERVICES WHERE type LIKE 'vegetarian restaurants')
```

The second category includes the following types of queries:

- *Notify me for each newly available weather service.* As shown in the service registration use case scenarios, this kind of queries can be answered with the use of the Concept Alert Directory. In a SQL like manner the above query would be:

```
SELECT * FROM SERVICES WHERE type LIKE 'weather forecasts' AND
available_from = (time – dt)
```

- *Give me weather services updates every five minutes.* Again this query can be answered with the aid of the Concept Alert Directory. To answer it, firstly the query *"Give me the weather services updates"* (which is a query of the first category) will be stored in the Concept Alert Directory. Afterwards, every five minutes the Concept Alert Directory will feed the stored query to the query execution manager and the results will be pushed to the issuing PO, service or user. Again, in SQL would look like this (note the "repeat" and "every" extending keywords that are necessary to express the continuity of the query):

```
SELECT * FROM SERVICES WHERE type LIKE 'weather forecasts' AND
available_from = (time – dt) REPEAT EVERY 5min
```

Adding location criteria requires that the current location of the inquired services is known. Towards this goal, communities, as already mentioned, know the "home location" CM for each of their members. When there is the need to know the current location of a member service, the CoAS asks it from the "home CM". The "home CAS" is able to keep track of the whereabouts of its registered services. So, in order to answer queries with location we use the following procedure:

1. The query is answered by the Query Execution Manager without the location criteria.

2. For each member service in the intermediate results the Service Ontology Manager ask its "home CM" for its current location.
3. The Query Execution Manager applies the location criteria to the updated intermediate results producing the final query results.

In the case that the original query's type was of the second category, then location criteria are included when it is modified and stored in the Concept Alert Directory.

The support location criteria include the following:

- Point queries. E.g. *what are the available weather services here?* An a SQL like example (notice the keyword 'near'):

  ```
  SELECT * FROM SERVICES WHERE type LIKE 'weather forecasts' AND location NEAR (CoAS_location)
  ```

- Area queries. E.g. *what are the available weather services for Athens?* Again, an a SQL like example would be (notice the keyword 'area'):

  ```
  SELECT * FROM SERVICES WHERE type LIKE 'weather forecasts' AND location IN AREA ('Athens')
  ```

- Range queries. E.g. *find me all the vegetarian services within one kilometre from my location.* Note that this kind of query also requires that the location of the issuer is also known. To make matter worse, if we add the criteria *"every five minutes"*, we also need to know how the issuer's location is updated. In order to enable this type of queries the CoAS must also know the "home CM" of the issuer. Fortunately this is a piece of information that can be retrieved by the issuer itself through the forwarding CM. Having the "home CM", the CoAS can retrieve the issuer's location from it, if the forwarding CM fails to deliver it (e.g. in case where the issuer moved to another CM). In an SQL like manner it would look like this (notice the keyword 'within'):

  ```
  SELECT * FROM SERVICES, POs WHERE SERVICES.type LIKE 'weather forecasts' AND PO.id = askingPOid AND SERVICES.location WITHIN (POs.location, 1 km)
  ```

- Nearest neighbour queries. E.g. *find me the closest vegetarian service.* This is the same as the range queries with the modification that the results are sorted based on the range in ascending order and only the first row is returned. The previous example would be modified as follows:

  ```
  SELECT TOP 1 * FROM SERVICES, POs WHERE SERVICES.type LIKE 'weather forecasts' AND PO.id = askingPO.id AND SERVICES.location WITHIN (POs.location, 1 km) ORDER BY RANGE (SERVICES.location, POs.location)
  ```

4. Implementation Issues

In order to implement the CoAS we use Java technology (JSDK 1.4.1). Additionally as mentioned CoAS are distributed across the network and communicate with other system's components like CMs and POs. This leads to the need of having a message passing model in order to exchange data between them. The communication between communities and CMs is achieved:

i. Through the remote method invocation model and more specifically the Java RMI package. Java RMI is also available for the J2ME platform, enabling light devices (such as mobile phones, PDA, etc.) to use it for communication purposes or to act as service providers.
ii. Web services model. In this case each major component (CM, CoAS) can been seen as a web service. The clients (CoAs, PO, applications) use the wsdl of the appropriate CM in order to contact it. This models allows implementation independency.

In order to allow devices to discover the CM covering their location they broadcast UDP packages in the local cell's access point in a predefined port. These (initiating) packages are retrieved by cell monitors who reply with the remote bind name and address of the local CM. When the device retrieves the name and address of the CM, it uses one of the above communication methods to interact with it. In case of communities, there is no need to discover the network (as in the devices' case). Communities know apriori the names and addresses of CMs and they can communicate remotely with them. Finally RMI or Web Services are also used in order to facilitate communication between the system's communities.

Another important issue that needs to be handled by the implementation is the format of the data stores. Currently we are experimenting with two different approaches in order to decide which one produces the optimal results in our case. The first approach is to store all the available context information in XML documents [2] and use X-Query [11] to retrieve information from them. The advantages of XML documents include (a) ease of transferal from device to device and (b) can be stored in light devices and (c) flexibility due to their semi structured form. The second approach is to use classic database management systems. The upside is the use of ready and optimal mechanism to store, retrieve and update the available data. The downside however comes from the very nature of the kept data which are ontology descriptions (mainly large text blocks) and that limits the speed advantage. The database approach also has the disadvantage of having to transcode the data to a light format in order to send them to a light device.

5. Conclusions

In this paper we developed a middleware that aims to provide high level semantic query processing. This middleware collaborates with the low level Cell Manager infrastructure, to answer not only location based queries but semantic level ones in an organized and efficient manner. The notion of ad hoc databases is formalized around the idea of communities. In fact, in this paper we utilized this idea and defined/created ad hoc databases around "concepts" and not just around location. We called these ad hoc databases, "communities" and instead of issuing queries just around the notion of location we do issue them over these communities and around the concept they represent. Communities provide somehow a semantic type of distributed index over heterogeneous resources. Context based queries, containment and continuous queries are also efficiently handled via the proposed middleware and its functionality. This middleware is being designed and its implementation is quite advanced.

References

[1] George Samaras, Constantinos Spyrou, Evaggelia Pitoura, Marios Dikaiakos, *Tracker: A Universal Location Management System for Mobile Agents.* Proc. The European Wireless 2002 Conference, Next Generation Wireless Networks: Technologies, Protocols, Services and Applications, pp. 572-580, Florence, Italy, February 25-28, 2002.

[2] T.Bray, J. Paoli and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 Specifications. *World Wide Web Consortium,* http://ww.w3.org/TR/Rec-xml

[3] XML Path Language (XPath). *World Wide Web Consortium,* http://www.w3.org/TR/xpath

[4] K. Koloniari and E. Pitoura. Bloom-Based Filters for Hierarchical Data. Technical Report 29-2002, Department of Computer Science, University of Ioannina, Nov 2002.

[5] G. Koloniari and E. Pitoura, "Bloom-Filters for Hierarchical Data", Proceeding of the 5th Workshop on Distributed Data and Structures (WDAS), 2003.

[6] Services Definition Language (WSDL), *Web page*, http://www.w3.org/TR/WSDL.

[7] Ouzzani, M., Benatallah, B., and Bouguettaya, A.: Ontological Approach for Information Discovery in Internet Databases. *Distributed and Parallel Databases Journal*, 8:367-392, 2000.

[8] A. Levy, D. Srivastava, and T. Kirk. Data model and query evaluation in global information systems. Intelligent Information Systems, 5(2), September 1996.

[9] Jason I. Hong. The Context Fabric: An Infrastructure for Context-Aware Computing

[10] D. Pfoser, N. Tryfona and V. Verykios. *Services-Based Data Management in a Global Computing Environment*. Computer Technology Institute Athens, Hellas

[11] XML Query (XQuery). *World Wide Web Consortium,* http://www.w3.org/XML/Query