Mobile Agent-Based Services for View Materialization *

Kyriakos Karenos[a] cs98kk2@cs.ucy.ac.cy
George Samaras[a] cssamara@cs.ucy.ac.cy
Panos K. Chrysanthis[b] panos@cs.pitt.edu
Evaggelia Pitoura[c] pitoura@cs.uoi.gr

[a]Department of Computer Science, University of Cyprus, Nicosia, Cyprus
[b]Department of Computer Science, University of Pittsburgh, Pittsburgh, PA, USA
[c]Department of Computer Science, University of Ioannina, Ioannina, Greece

Mobile agents are ideal for mobile computing environments because of their ability to support asynchronous communication and disconnected data processing. In this paper, we present a prototype set of extensible mobile-agent based services that allow the definition, materialization, maintenance and sharing of views created over remote web-accessible datasources, called ViSMA (Views Supported by Mobile Agents). ViSMA's primary goal is to support the customization and personalization of views by mobile users carrying lightweight devices of various connectivity and resources such as portable computers and PDAs. It achieves efficient view customization by localizing the materialization of view fragments of a complex view within different mobile agents which monitor each other's movement in order to minimize communication costs. ViSMA has been fully implemented over a mobile agent platform and tested using three alternative mobile client types.

I. Introduction

Data accessing in the recent years has been affected by three major trends: the vast amount of available sources, the increase of mobile and wireless clients and the need to support personalization and customization. Current information systems are built across several datasources and, in their majority, frequently request access to distributed data. Gathering data in a centralized manner usually introduces increased processing costs and makes poor use of network resources as mobile clients increase the distance and change the connectivity between the access location and the location of the data. An effective strategy for confronting these problems with respect to distributed datasource access has been proven to be the use of mobile agent technology, which considerably improved system performance [11].

Mobile and wireless computing itself, introduces additional restrictions to network resource availability and it is characterized by frequent disconnections and delays. Device variability in computational power and resources is a factor that hinders the process of building generic systems to work effectively within diverse contexts. In addition, specific user requirements need to be taken under consideration both with respect to the requested data (i.e., personalization of data) as well as the functionality provided over these data (i.e., customization of services).

Views provide a functional and flexible answer to multi-source data collection and are quite appropriate for defining personalized data sets. View materialization in mobile computing [13, 22] introduces a number of challenges although a primary concern that is frequently stressed is keeping the views as close as possible to the mobile user [19]. Additionally, support must be provided for thin clients and finally it should be possible to dynamically add new datasources to the view.

New tools that can meet the presenting challenges must become available to users. ViSMA is one such system that provides the functionalities of defining, materializing and maintaining views over multiple datasources[1] by taking advantage of mobile agent technology with additional features to support mobile and wireless clients. The role of mobile agents in ViSMA is twofold: Firstly, views are carried within mobile agents called View Agents [8] and may relocate themselves to reduce the distance from users that frequently request them. Secondly, they are used for migrating to a remote data source and locally execute update propagation and query materialization operations, relieving remote clients and local datasources

*This work was partially funded by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2001- 39045 SeLeNe project and the IST-2001- 32645 DBGlobe project.

[1]ViSMA integrates any *datasource* that can export a relational/object-relational interface, which is the assumption of JDBC/ODBC connectivity. In the examples used in this paper, we consider the datasource to be a relational database thus the terms 'datasource' and 'database' are used interchangeably.

from performing this task while saving precious network resources.

The ViSMA system is fully implemented in Java, which is suitable for the development of mobile-agent based applications mainly due to its platform independence. We have also used Voyager ORB [12] to be the mobile agent platform. The Extensible and Flexible Library (XXL) [2, 21] was used to instantiate the relational database functions executed by the agents. We have also utilized Tracker [15], an efficient location management system, to enable agent coordination and communication. We have implement three alternative client components that aim at supporting clients of variant resource availability, namely, the ViSMA Light Client Applet, the ViSMA Servlet Engine and the ViSMA Client Agent.

The rest of this paper is structured as follows: We first provide a general overview of Mobile-Agent technology (Section II). We then describe the architecture for the ViSMA system in detail (Section III) and clearly define the system's services and the functionality provided by the ViSMA middleware (Section IV). We also describe ViSMA's materialization scheme, which is based on the concept of the *Data Holders* and illustrate how view mobility is achieved. We conclude with a discussion of ViSMA's feature (Section V) and future work (Section VI).

II. Mobile-Agent Technologies

Mobile Agents emerge from incorporating active computations with objects and making them mobile in order to accomplish tasks in a highly dynamic and heterogeneous environment such as a mobile computing one. They are defined as processes dispatched from a source computer to accomplish a specified task [3, 6]. Each mobile agent is a computation along with its own data and execution state. After its submission, the mobile agent proceeds autonomously and independently of the sending client. When the agent reaches a server, it is delivered to an agent execution environment. Then, if the agent possesses necessary authentication credentials, its executable parts are started. To accomplish its task, the mobile agent can transport itself to another server, spawn new agents, or interact with other agents. Upon completion, the mobile agent delivers the results to the sending client or to another server.

By letting mobile hosts (clients) utilize mobile agents, the burden of a computation is shifted from the resource-poor mobile hosts to the fixed network. Mobility is inherent in the model; mobile agents migrate not only to find the required resources but also to follow mobile clients. Finally, mobile agents provide the flexibility to adaptively shift load to and from a mobile host depending on bandwidth and other available resources.

II.A. Mobile-Agent Platforms

Several mobile agent platforms have been proposed. Java-based platforms have dominated both in research and in commercial applications due to the inherent advantages of Java, namely, platform independence support, highly secure program execution, and small size of compiled code. These features of Java combined with its simple database connectivity interface (JDBC API) that facilitates application access to relational databases over the Web at different URLs make the Java approaches very attractive for our implementation of web database connectivity.

The Java-based mobile agents platforms include IBM Aglets Workbench [5], Recursion Software Voyager [12], Mitsubishi's Concordia [20], IKV++ Grasshopper [1] and General Magic's Odyssey [6]. While all these systems provide the basic functionality expected from such mobile platforms, they differ significantly in their system architecture, the communication mechanism employed, the additional functionality they provide and their performance. Any of these mobile-agent platforms can support our proposed system. However, we have chosen to use Voyager due to the facts that (i) it is a commercial product used currently in real applications and (ii) it provides relative good performance, as shown in [4, 10].

II.B. Location Management: Tracker

Tracker [15] is a distributed location management middleware which has the ability to manage the location of mobile agents that travel autonomously across the Internet in search of useful information. Being MASIF compliant [9] and based on Java, Tracker is independent of the semantics of any particular Java based mobile agent platform and able to integrate with each one of them seamlessly. Major advantages of this middleware include: (a) the ability to manage the location of any mobile agent and (b) allowing agents to locate other agents, independently of their native execution environment.

Tracker provides two classes: the *TrackerAgent* and the *TrackerRegistry*. By extending the abstract class `TrackerAgent`, a Tracker enabled agent can be created. Upon its creation, this agent is registered to the `TrackerRegistry` of the local node. The

local `TrackerRegistry` informs the proper nodes (i.e., their `TrackerRegistry`) about the presence of the new agent. During a move of a Tracker-enabled agent from one node to another, the agent itself (or its proxy) informs the destination-node's `TrackerRegistry` about its arrival. This is the responsibility of the agent and the appropriate method is part of the TrackerAgent's definition. In turn, the destination node's `TrackerRegistry`, informs the appropriate nodes about the arrival of the agent based on the current location mechanism. As soon as this procedure finishes and again in accordance with the current location mechanism, all appropriate nodes are informed about the departure of the agent in order to delete it. Now in any node, a user can ask the local `TrackerRegistry` for the location of any agent by simply using the agent's name.

III. Architecture

In order to allow the creation and maintenance of distributed views we have designed a multi-tier agent-based architecture. The architectural components are distinguished based on their placement and functionality (Figure 1) to: *Server Side Components*, *Datasource Side Components* and *Client Side Components*.

Server side components are the agents that form the core of the system. Typically, they reside at a centralized location called collectively, the *ViSMA Server*. In our current implementation, we assume an enterprise-wise ViSMA server, that is, there is a single ViSMA server that supports all users and applications within an enterprise or computing environment. As we will discuss in Section V, we are currently extending ViSMA to support cross-enterprise interactions by configuring ViSMA Servers in a P2P-like system.

Datasource side components are agents generally reside at a datasource location (e.g., at a remote database server location) or are created at the ViSMA server but migrate and execute the bulk of their operations at one or more datasource location.

Client side components represent the interface between a user and the ViSMA middleware thus, need to be accessible by the user. Therefore, they may be located at the users machine or to an intermediate location known to the user transparently with respect to the ViSMA server. Client side components generally need to interact only with the Server side agents but not directly with the database side agents. Note, though that it is possible and feasible in specific application contexts, for a client side agent (e.g., the ViSMA agent client) to directly contact another agent without requesting mediation from the ViSMA Server since all agents can be located via Tracker.

In ViSMA, agents are distinguished into either *mobile* or *stationary*. A mobile agents may move from one network node to another in order to complete a predefined task while a stationary agent migrates once to a specific node, remains ("parks") at that location and periodically execute its task according to the application requirements.

Wherever mobile agents need to execute, a *Mobile Agent Platform* needs to be installed (i.e., the Voyager server). For the data source side, this is practically the sole system requirement since all remaining functionality is dynamically deployed. At the server side, the Voyager server is integrated with a web server used for two basic purposes: (i) remote class loading by agents (via HTTP) and (ii) for providing the web interface to web browser users.

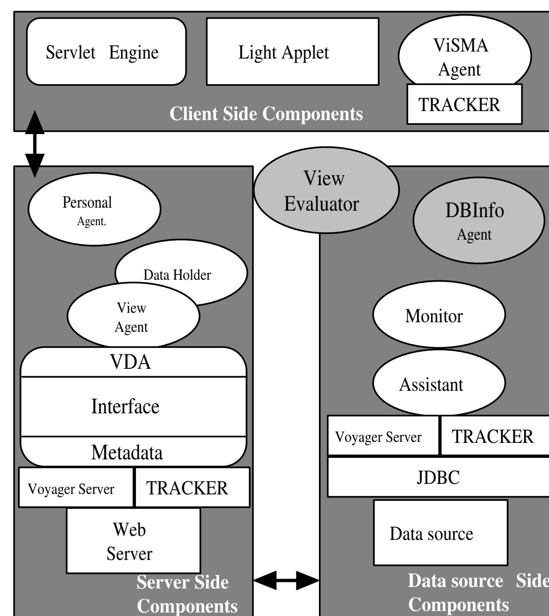

Figure 1: ViSMA Architecture

III.A. Server Side Components

The ViSMA Server can be characterized as a "dictionary" in terms of meta-data on participating data sources and created views. With respect to views, metadata include the view definition as well as the current location of the mobile agent carrying the view. The ViSMA server consists of four types of agents: *View Dictionary Agent, View Agent, Data Holder* and *Personal Agent*.

View Dictionary Agent (VDA): The basic agent at the Server side is the View Dictionary Agent. The VDA is the central communication and coordinating agent,

which is contacted by users, or agents receiving requests via a well-defined ViSMA user-access (service) interface.

Figure 2 shows a high level definition of the VDA interface. This is divided into three sections comprising a WSDL-like definition of basic system-offered services. The first section represents a set of system-defined object types, used as service parameters including view definition, view actual data and data-source access information. The second section provides a sample set of exchanged messages definitions such as a view creation message. The last section is a set of definitions involving the system operations themselves that make use these messages.

It should be noted that any message passing protocol can mediate the exchange of messages between the client applications and the VDA. Remote Process Calls (RPC, e.g., Java RMI [17]) as well as Web Services technologies [18] provide excellent implementation solutions.

View Agent (VA): The View Agent is another key agent of the ViSMA architecture. A VA is a mobile agent that is created when a user defines a view to be materialized and it is initialized by the VDA. A VA is responsible for creating, materializing and maintaining a view. A VA allows for its materialized view to be queried by external entities. Therefore, sub-views can be derived from a VA view. For sharable views, its materialized data are available and can be accessed by all ViSMA clients. Since a VA is a mobile agent this data can be carried with it as it moves. Therefore, view migration is achieved.

Data Holder Agent (DH): Data Holders are mobile agents. When a VA is created, it initializes and dispatches a number of DHs to support it in the materialization and maintenance of its view. The DHs created by a VA are configured in a hierarchy with the VA as the root. Each DH in the hierarchy is responsible for handling an individual view *fragment*. At its creation time, a DH receives the definition of a view fragment for which it is responsible. The VA is responsible to combine the fragments and produce the final view. Effectively, DHs form the structure of the materialized view.

Personal Agent (PA): The Personal Agent is a mobile agent that is used for creating personalized, non-shareable sub-views. It derives its sub-view from the view of a VA and maintains its sub-view by only issuing user queries to the VA. The PA may communicate directly with its client and may move as its client moves.

```
<types>
 <schema>
   <complexType name="ViewDefinition ">
   </complexType >
   <complexType name="ViewResultSet ">
   </complexType >
   <complexType name="DatasourceInfo ">
   </complexType >
 </schema>
</types>
```

```
<message name=" GetAvailableDatabases ">
</message>
<message name=" GetAvailableViews ">
</message>
<message name=" AvailableDatabasesResponse ">
</message>
<message name=" GetAvailableViews ">
</message>
<message name=" ShareableViewResponse ">
</message>
<message name=" DeleteView ">
<part name="identification" type="long"/>
</message>
<message name=" DeletionResponse ">
</message>
<message name=" GetViewData ">
<part name=" viewname " type="string"/>
</message>
<message name=" DataRetrievalResponse ">
</message>
<message name=" RegisterDatasource ">
<part name=" param " type=" visma :DatasourceInfo "/>
</message>
<message name=" CreatePersonalizedView ">
<part name=" def " type=" visma :ViewDefinition "/>
</message>
<message name=" CreateShareableView ">
<part name=" def " type=" visma :ViewDefinition "/>
</message>
```

```
<portType name="vismaPortType ">
  <operation name=" DatasourceRegistration ">
    <input message=" tns :RegisterDatasource "/>
    <output message=" tns :RegistrationResponse "/>
  </operation>
  <operation name=" ClientInitialization ">
    <input message=" tns :GetAvailableDatabases "
               name=" ClientInitInput"/>
    <output message=" tns :AvailableDatabases "
               name=" ClientInitOutput "/>
  </operation>
  <operation name=" ShareableViewCreation ">
    <input message=" tns :CreateShareableView "/>
    <output message=
        " tns :ShareableViewCreationResponse "/>
   </operation>
</portType >
```

Figure 2: Basic WSDL-like Service Description

III.B. Datasource Side Components

The datasource side components are two mobile agents, namely, *DB Info Agent* and *View Evaluator Agent*, and two stationary agents, namely, *Assistant Agent* and *Monitor Agent*.

Assistant Agent: The Assistant Agent is a stationary agent that maintains a pool of connections to the data source to serve visiting agents [11]. These agents provide transparent connectivity between datasources and agents that require access to the datasources. Changes to a datasource connectivity settings need only made known to the Assistant Agent.

DB Info Agent: The DB Info Agent migrates to the remote datasource site, collects its metadata (schema and data types), sends them to the VDA and finally self-destructs.

Monitor Agent: The Monitor Agent is another stationary agent created and dispatched to a datasource site by some DH to enable view maintenance. A Monitor Agent "parks" at the remote site and periodically re-queries the datasource and sends changes to the DH. Thus, DH may receive simultaneous updates from multiple Monitor Agents.

View Evaluator Agent (VEA): Another fundamental agent is the View Evaluator Agent. A VEA is a mobile agent that is sent by a DH to travel from one datasource to another to collect the data required for the materialization of the view fragment that the DH is responsible for. A VEA combines the collected data as soon as possible at their retrieval sites in order to minimize the size of its transported data while at the same time incrementally materializes the view fragment. Access to the datasources is provided by each local Assistant Agent. A VEA needs not be destroyed upon completion of a materialization. On the contrary, it can be reused to re-execute the materialization plan whenever necessary.

III.C. Client Side Components

A client component can be any entity that can contact the VDA directly or indirectly (e.g., via a gateway which translates user requests to VDA calls). We have implemented three alternative types of base clients, each of which can support mobile users with different capabilities. These alternatives are: *ViSMA Client Agent, Light Applet* and *ViSMA Servlet Engine*.

The ViSMA Client Agent provides each user with a private mobile agent that can interact in a P2P manner with other agents. A ViSMA client Agent can submit a query to a VA without any VDA intercession while it interacts with VDA in order to create views and retrieve metadata on existing views. The ViSMA Client Agent assumes that the user device has adequate computational resources to host the mobile agent platform such as the case of laptop and notebook machines.

The second client type is Applet-based and enables any user with any Java supporting browser to download a light applet GUI to interact with the VDA. In this case, VDA also accepts queries to views. It forwards any query to the appropriate VA and returns to the client the response of the VA. The applet does not require a mobile agent platform to be installed on the device and does not require administrative privileges.

Finally, for users with no Java support or with resource restrictions, the ViSMA Servlet Engine is provided. The Servlet engine is a middleware component that accepts standard HTTP requests and replies in standard HTML. It only assumes a simple web browser which nowadays is available on almost any relatively low resource mobile devices such as PDAs and Smart Phones.

IV. System Services

In this section we will illustrate the functionality of the system using a concrete example. Consider the following distinct databases in a medical information system. For simplicity, we assume that each of the following databases is represented as a single table (Figure 3). In this schema, "PatientID" is the primary key for "Pa_Hospital", "Ni_Hospital" and "DoctorDB" and is represented in the same way (i.e., same type) in all tables. "Medication" is a foreign key in "DoctorDB" that references the primary key of "PharmacyDB."

Consider a doctor who visits two separate hospitals in Cyprus for patient treatments, one in Paphos ("Pa_Hospital") and one in Nicosia ("Ni_Hospital"). The hospital databases are updated several times a day regarding the condition of patients. The doctor also maintains her own database ("DoctorDB") to keep track of the treatments related to her personal patients. "DoctorDB" database is also located in Nicosia but managed by different database management system than "Ni_Hospital." The "PharmacyDB" is in Pittsburgh, USA and keeps data on types of medication.

IV.A. Datasource Registration

In order for a datasource such as "Pa_Hospital", "Ni_Hospital", "DoctorDB" and "PharmacyDB" to be available for access, the datasource administrator must register its location to the ViSMA system. The

Pa_Hospital

PatientID	Pressure	Temperature	Pulse

Ni_Hospital

PatientID	Condition	Respiration	Pressure

DoctorDB

PatientID	LastVisit	Diagnosis	Medication

PharmacyDB

Medication	Price	Available

Figure 3: Example Databases Schemas

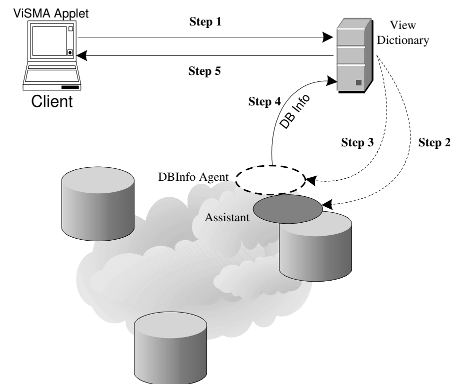

Figure 4: Datasource Registration

datasource administrator is simply required to download the datasource registration applet (or access the corresponding Servlet page if Java is not supported) and provide the datasource's location in the form of a URL or IP and the port where the data source server is listening for connections.

A datasource administrator must also provide valid data access codes (Figure 4, Step 1). By providing for valid access codes, security and access control is preserved by ViSMA. Valid codes will correspond to users, groups or roles that encode what is visible to the user/application and hence can be exported by a datasource and imported by the VDA. That is, a datasource administrator does not actually register an entire datasource, but rather data views or portions of the datasource identified by the access codes. Thus, before registering a datasource, the administrator may create the appropriate datasource view, define roles and set the necessary protections.

The registration process is followed by an *import* process (Steps 2 to 4): Provided the aforementioned information, the VDA creates and dispatches an Assistant Agent, which migrates and connects to the datasource using the access codes. Afterwards, the VDA creates and dispatches a DB Info Agent to collect the metadata of the datasource with the mediation of the Assistant Agent. When a DB Info Agent returns with the metadata, VDA creates an entry in its dictionary for the datasource. In the final step (Step 5), the VDA acknowledges the inclusion of the datasource to the dictionary.

This two-phase strategy of registration and importing, on one hand, avoids user-related errors in providing the datasource metadata needed for the dictionary and on the other hand, allows for asynchronous operation and lets the VDA to handle other incoming requests.

IV.B. View Definition

The client side components provide an easy-to-use GUI in the form of a "wizard" which downloads all available datasources and existing views from the VDA and guides the user through the definition of a new view. During this process the user select the databases, tables and attributes to retrieve as well as sets the restrictions on each attribute. The user can also define which relation attributes are most important to her and select to specifically monitor those attributes for changes. View definition is assisted by retrieving the metadata of the selected databases from the VDA locally to each client side component.

The user can define a view to be either *shareable* or *private*. Shareable views are visible to every client that connect to the system while private views are only available to the clients which have created them. Additionally, *personalized* view can be defined. Personalized views are materialized subviews of existing views created with the help of a Private Agent (PA).

ViSMA handles view definitions consisting of project, select and join operators and combinations of *Project-Select-Join* (PSJ) queries. PSJ views are most common and, to a large extent, cover the requirements of the problem investigated [23]. We categorize views into either *simple* or *complex*. A simple view can be broken down to a sequence of PSJ operations that actually represent a linearized query evaluation tree. The materialization of the view represented by this tree is achieved by a single VEA (View Evaluator Agent). Complex views consist of at least two simple views combined with some set operator such as "Union" or "Minus".

Views can also be defined to extract data from a single datasource (*single views*) or from multiple datasources (*multi-views*). In the latter case and for simple view, the linearized query evaluation tree can serve as the itinerary for the VEA responsible for materializing the simple view.

Returning to our example, suppose that the doctor has to leave for a conference in Athens but needs to

```
DEFINE SHAREABLE ‘Doctor-View’ AS

       SELECT  Ni_Hospital.PatientID,
               Ni_Hospital.Pressure,
               DoctorDB.Diagnosis,
               PharamacyDB.Available
       FROM    Ni_Hospital,
               DoctorDB,
               PharamacyDB
       WHERE
               Ni_Hospital.PatientID =
                 DoctorDB.PatientID
               AND
               DoctorDB.Medication =
                 PharamacyDB. Medication
               AND
               DoctorDB.LastVisit
                   > 1/1/2004

   UNION

       SELECT  Pa_Hospital.PatientID,
               Pa_Hospital.Pressure,
               DoctorDB.Diagnosis,
               PharamacyDB.Available
       FROM    Pa_Hospital,
               DoctorDB,
               PharamacyDB
       WHERE   Pa_Hospital.PatientID =
                 DoctorDB.PatientID
               AND
               DoctorDB.Medication =
                 PharamacyDB. Medication
               AND
               DoctorDB.LastVisit
                   > 1/1/2004

   MONITOR Ni_Hospital.Temperature,
           Pa_Hospital.Pressure
   EVERY   5 minutes
```

Figure 5: Sharable View Definition

keep track of her patients while being away. She selects to define a view (e.g., using the Applet-Based Client from her laptop) called `Doctor-View` before leaving for her trip. This complex, multi-view expressed in an SQL-like language is defined as shown in Figure 5. This SQL-like language is similar to the language proposed in [8].

The `MONITOR-EVERY` clause defines the set of attributes to be monitored as well as the time interval between re-querying the sources for changes on these attributes.

IV.C. Materialization Scheme

In ViSMA, each view is associated with a *view definition object* stored by the VDA. This object encapsulates the structure of the view consisting of each individual PSJ sub-query along with the operators that connect them. Additionally, the query definition object contains the list of attributes to be monitored. In our previous example (Figure 5), the two ‘select-from-where’ clauses before and after ‘UNION’ represent two simple view definitions (fragments) connected with the ‘UNION’ operator whereas `Temperature` of `Ni_Hospital` and `Pressure` of `PA_Hospital` are the attributes to be monitored every 5 minutes.

When a view is defined, it is pre-processed into a number of simple view fragments to be stored in the view definition object. The pre-processing functionality of constructing the view definition object is rather simple and is provided by the client side component. This is feasible since all required metadata is downloaded locally. This also relieves the VDA from an additional task of having to create each view definition object.

When the VDA receives and saves the view definition object of a new view, it creates a VA (View Agent) and passes it the view definition object. For each simple view in the view definition object, the VA creates a Data Holder (DH) to administer this simple view. Each DH, in turn, creates a VEA and provides a materialization plan. This is achieved using the *Query Builder* component that can be programmed to optimize a query plan given datasource-related information which can be obtained from the VDA. The ViSMA Query Builder currently creates a plan that attempts to minimize the size of data moved from one datasource to another.

Once a VEA is created and receives its materialization plan, it travels to each datasource specified in its plan, retrieving the requested data which is passed to the DH at the completion of the plan. Figure 6 shows how a simple query referencing three distinct datasources is materialized. In our example view definition, this plan reflects the materialization of any one of the two ‘select-from-where’ clauses. Eventually each DH will receive a materialized view fragment.

Each DH is also responsible for dispatching the necessary Monitor Agents to any database that includes table attributes selected for monitoring by the user. For the example above, Monitor Agents need to be sent to `Ni_Hospital` and `Pa_Hospital`, querying the `Temperature` and `Pressure` attributes respectively every 5 minutes.

As mentioned in Section III, the final DH structure

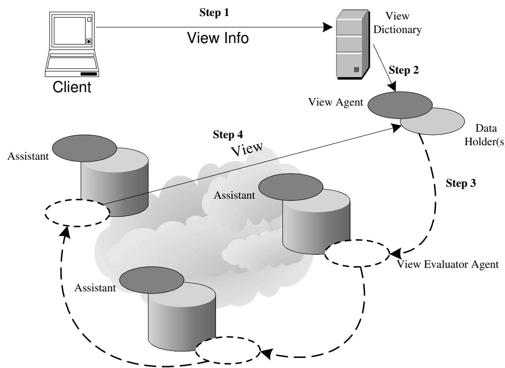

Figure 6: Simple View Materialization

is tree-like (hierarchical), combining at each level a number of DH data. As a more generic example, consider the following complex query in which three simple views Q1, Q2 and Q3 are combined by UNION and MINUS.

```
Q = (Q1 UNION Q2) MINUS Q3
```

Each of the query definitions is passed to a DH. Figure 7 shows how the DHs are structured within a VA. Discontinued lines imply flow of data whereas continued lines imply an operation between this DH data and the data of the related DH. We should note that this view data decomposition allows for an efficient update of the view since individual fragments are updated independently. Although view querying requires view fragments managed by different DH to be recombined to produce a reply, the recombination overhead is primarily reduced by the ability of a VA to cache the view and answer multiple queries using its cache. A VA executes a recombination of the DH data periodically and only if changes have occurred. (This last piece of information is easily extracted from the DHs.)

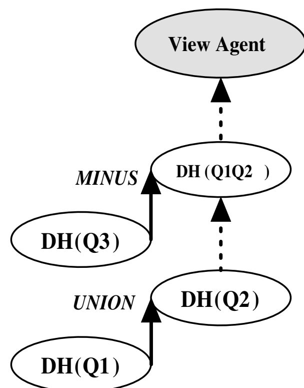

Figure 7: View Structure Based on Data Holders

Two interrelated assumptions were made regarding the efficiency of the VA caching: (1) In general, materialized views required by mobile clients are expected to be highly specialized and hence relatively small and can be stored and efficiently transported by a VA as part of its state; and (2) given that the views maintain limited size of data, the frequency of updates is expected to be low and localized to few datasources at any given point in time.

As discussed just above, in the current ViSMA implementation, the data of the materialized view is part of the agent's state. This means that the used mobile agent infrastructure imposes a restriction on the maximum size of a view. It is possible, however, to decouple the view payload during movement and integrate it to the agent after it has completed the migration, hence alleviating any restrictions on the view size. Currently we are working on view storing alternatives (such as using XML files and communicate them via standard HTTP) to enable larger size views to be stored and relocated dynamically.

The materialization scheme described above provides two basic advantages deriving from mobile agent technology. Firstly, VEAs move and execute directly at the datasource. Therefore, they minimize/eliminate the need of expensive communication such as persistent connections or repeated creation of ports and remote exchanged of messages. Secondly, a view can be created and maintained while the user is disconnected and on the move. Upon reconnection the view is available and updated in accordance to the monitoring parameters.

IV.D. Maintaining Views

As mentioned previously, Monitor Agents are used to provide updates for changes occurring at the data sources. Monitor Agents reside at the datasource to which they connect and periodically re-query to retrieve the values of the attributes they monitor. Monitor Agents always keep the latest values of the attributes they monitor. When new values are retrieved, a comparison is made between the last and the current values and the view difference (Δ View) is created. The Monitor Agents sends only the Delta View (changes) to their dispatching DHs. Based on these changes a DH can select to delete removed rows or resend a VEA to re-materialize the view. Note that a DH may receive updates from Monitor Agents located at different datasources (Figure 8).

Our materialization scheme (that is, the scheme used by DHs) extends and adapts the simple Strobe Algorithm [23] to suit the needs of a mobile envi-

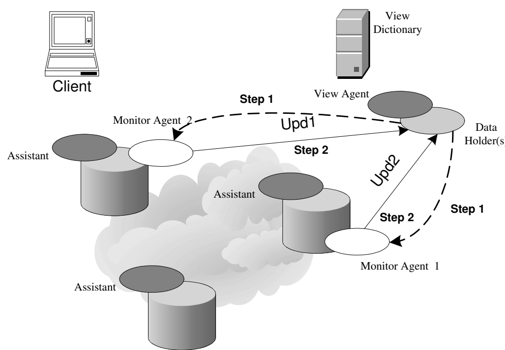

Figure 8: Datasource Monitoring

ronment and mobile agent technology per se: Simple Strobe algorithm has three high level operations. These are (i) a *source_evaluate(Q)* operation which evaluates the portion of query *Q* that is still not known, (ii) operations performed at the data source which include update propagation and actual evaluation of a query and (iii) operations performed at the warehouse which include receiving the updates and performing the necessary changes to the view.

Regarding each operation above we have made the following adjustment: The main changes concern the *source_evaluate* operation. This is basically performed by the VEA. A VEA is provided the complete materialization plan for a query beforehand by the DH, so it needs not calculate which part of the query is to perform next.

Operations at a datasource become a responsibility of different agents. As noted above, the evaluation is performed by the VEA thus, no operation is required by the datasource itself (i.e., the code is moved to the source). Updates are encapsulated in update operation messages (as with Strobe) by the Monitor Agents and sent to the appropriate DH.

Operations originally performed by the data warehouse are executed by each individual DH. Two basic elements which are part of Strobe need to be repeated here: (i) As noted above, updates are sent in the form of update operation messages, i.e., delete and insert operations[2]. Therefore, a DH maintains an *Action List* (AL). (ii) For consistency purposes, the DH maintains a *pending(Q)* set of operations while a VEA is evaluating a query *Q*.

As a concrete example, consider again our `Doctor-View` and the case where the temperature of a patient in Nicosia Hospital ("Ni_Hospital") has changed. After the lapse of the monitoring period, the Monitor Agent will detect the change (as a deletion/insertion) and send the update message to the DH responsible for the simple view shown in the first part of Figure 5. Based on our materialization scheme, the DH will resend a VEA to the three datasources to reconstruct the updated view fragment.

It should be noted that due to the underlying location management system, Tracker, the message exchange among the agents is transparent with respect to movements of VEAs, DHs and VAs.

IV.E. View Data Retrieval

IV.E.1. Shareable and Private Views

Sharable views created can be shared among all users of the system while private ones cannot. We have mainly experimented on the more interesting case of sharable views since they allow for the creation of personalized views.

A user may select to get any specific view's latest data by requesting it from the corresponding VA. In our example, the head-doctor may want to check on the status of the patients of the traveling doctor. Retrieving the data from a VA is done by requesting it either directly (ViSMA Agent client) or indirectly through the VDA (e.g., when using the Applet Client or the Servlet Engine). The retrieved data is presented to the user in a format that again depends on the client type that was used for the request. For example, the reply to a Servlet Engine request is in pure HTML.

IV.E.2. Personalized View Creation

Users may be interested in a specific subset of some view. In this case, the system allows the creation of a personalized subview from an existing view in ViSMA, which cannot be shared and which is transparent with respect to the original datasources.

In our previous example, suppose that the doctor's assistant wants to define a view on the patients, whose pressure rises over 120/80, viewing the results from her PDA. Assuming that her PDA provides a simple web browser and has limited memory, she can use the Servlet Engine client to define the sub-view shown in Figure 9.

```
DEFINE PERSONALIZED 'Assistant-View' AS

SELECT PatientID, Pressure
FROM View 'Doctor-View'
WHERE Pressure > 120/80
```

Figure 9: Personalized View Definition

The query is sent to the VDA (Figure 10), which

[2] An update is a sequence of a deletion and an insertion [23].

creates a Personal Agent that may move close to the VA to initially query it. In general, Personal Agents monitor the movement of VAs and their clients using Tracker and adjust their own location to minimize communication costs. However, a Personal Agent typically follows its user in order to reduce delays of the accesses to the personalized view which are expected to be frequent and remotely re-query the VA for updates which are less frequent.

Note that unlike the updating scheme of the VAs (push-based), the Personal Agent is kept updated by re-querying the VA (pull-based). Essentially, a Personal Agent captures a snapshot of the original view. This function is done periodically by the Personal Agent but it can be initiated by the user as well when necessary.

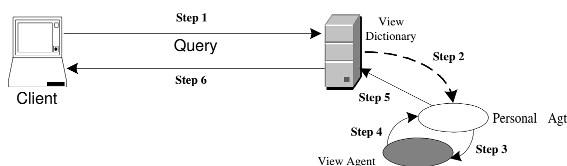

Figure 10: Personalized View Creation

IV.F. View Mobility

Since a VA is a mobile agent, it can select to move closer to the group of users who use it most frequently. Since a materialized view is captured by a VA and its supportive DHs, whenever a VA needs to move, it directs its attached DHs to move along to the same location. The relocation message is passed through in a hierarchical manner matching the actual hierarchy of the DHs.

To better illustrate view mobility, let us revisit our example one last time. Suppose that the doctor connects again to ViSMA upon arrival to the conference and queries the VA responsible for the `Doctor-View`. By detecting the new location and/or by noticing increased delays, the VA moves from Nicosia where was originally created to Athens from where the doctor is currently connected, at the local network (or the doctor's local machine), making request noticeably faster and reducing network traffic. Subsequently, if the network conditions are reversed or when the doctor returns to Nicosia, the VA moves back to Nicosia. In assessing its movement decisions as well as to locate other agents, the VA uses Tracker.

IV.G. View Deletion

When a user decides to delete a view she created, she sends a message to the VDA. The VDA will notify the corresponding VA, which will hierarchically notify its DHs (Figure 11).

Users may select to keep the last version within the Personal Agent, however, generally views are considered invalid (outdated) if not deleted or updated for a specified period of time which is customizable and usually dependent on the type of the application for which the views are used.

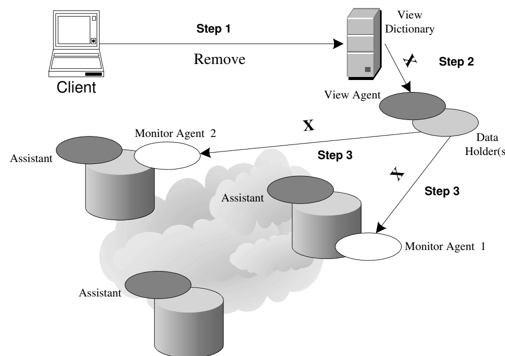

Figure 11: View Deletion

V. Discussion

ViSMA is a full-fledged prototype system that provides complete functionality by covering extensively all stages of the view manipulation processes. It evolved from two previous systems, namely, DVS [16] and VG [14], which have illustrated the basic ideas of the use of mobile agents to support web views but had limited functionality.

A primary objective of ViSMA is the customization and personalization of views by mobile users in an efficient manner. In ViSMA, this is provided by both private views as well as personalized views which incorporate ideas from [7, 8] for customizing consistency and currency: During view definition, the user may explicitly select which columns to monitor for changes and at what re-querying frequency. It should be noted that an understanding of how view consistency is affected by view currency customization in a mobile agent environment as those supported by ViSMA is provided in [8]. View customization efficiency is enhanced by the mobility of all types of materialized views, namely, sharable, private and personalized. The responsible mobile agents can monitor each other position and movement of clients and decide their own location in order to minimize commu-

nication costs.

A unique charecteristic of ViSMA is that follows a new materialization and maintenance philosophy based on the concept of Data Holders that capture the structure of a materialized view. By localizing the materialization of simple view fragments composing a complex view within Data Holders, efficient materialization at the level of individual view fragements can be developed as described in the system services section (Section IV). Further, Data Holders, being mobile themselves allow for the View Agent to relocate any one of the view fragments independently.

Another feature of ViSMA is system extendibility, which can be visualized in two directions: *client extendibility* and *functional extendibility*. Client extendibility can be achieved since ViSMA architecture allows additional types of users to have access to the VDA via the request interface. The user side components need only decide how the data received will be handled (e.g., filtering the result to WML for a WAP-enabled device). We would also like to note that non Java-enabled clients can also be supported in ViSMA by providing a user-friendly wizard-like GUI to assist view and sub-view definition which also allows for individual attribute monitoring settings.

Functional extendibility is provided for developers who may extend current agents to execute specific, system-defined functions viewing ViSMA as an Application Programmer Interface (API). For example developers may supply the Query Builder with additional query plan optimizations or recode the Monitor Agent's update propagation method. We are currently working on extending the enterprise-wise ViSMA to a Multi-ViSMA system which supports cross-enterprise interactions. A Multi-ViSMA system will consist of a collection of distributed View Dictionaries, each servicing a set of clients and datasources. View Dictionaries may create ad-hoc networks and communicate in a P2P manner thus, interconnecting different organizations.

VI. Conclusion and Future Work

Recently, data access and retrieval have presented a number of challenges mainly due to the vastness of available data, the particularities of mobile and wireless computing and the need for personalization and customization.

In this work we present ViSMA, a functional system that aims at assisting the dynamic definition, materialization and maintenance of database and web views. The primary design goal is to support mobile and wireless clients and be adaptable to the various types of user needs depending on the device capabilities and connectivity conditions.

We propose a mobile-agent based architecture on which the system is build and describe how mobile agents can be dynamically deployed in providing a number of view manipulation services which are enhanced by view mobility as well as structured storage of views based on the Data Holder concept. We also highlight the fact that the system can be extended to accept new types of clients and may adjust its agent's functions to implement designer specific strategies by using PDAs and laptops.

Our first milestone in the development of ViSMA was to produce a functional prototype. As future work we intend to conduct an evaluation of the performance tradeoffs concerning update efficiency versus query and retrieval of view fragments held by multiple DHs. Further we plan to work on enhancing the materialization plan at the Query Builders as well as experiment with other materialization strategies and algorithms. Finally, we shall thoroughly evaluate our system scalability with respect to the type of client and the type of query, which are a function of the view storage scheme and the mobile agent platform capabilities.

References

[1] C. Bäumer and T. Magedanz. Grasshopper: A Mobile Agent Platform for Active Telecommunication. In *Proc. of the Third Int'l Workshop on Intelligent Agents for Telecommunication Applications*, pages 19–32, August 1999.

[2] M. Cammert, C. Heinz, J. Krdmer, M. Schneider, and B. Seeger. A Status Report on XXL - a Software Infrastructure for Efficient Query Processing. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 26(2):12–18, 2003.

[3] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant Agents for Mobile Computing. *IEEE Personal Communications*, 2(5):34–39, 1995.

[4] M. Dikaiakos and G. Samaras. A Performance Analysis Framework for Mobile-Agent Systems and in infrastructure for agentsand multi-agentSystems and Scaleable Multi-Agent Systems. In *Porceedings of the FourthInt'l Conference on Autonomous Agents*, pages 180–187, June 2000.

[5] IBM Japan Research Group. Aglets Workbench. Available at http://aglets.trl.ibm.co.jp.

[6] White J.E. Mobile agents, general magic white paper, 1996. Available at http://www.genmagic.com/agents.

[7] S. Weissman Lauzac and P. K. Chrysanthis. Programming views for mobile database clients. In *Proc. of the 9th DEXA Int'l Workshop on Mobility in Databases and Distributed Systems*, pages 408–413, August 1998.

[8] S. Weissman Lauzac and P. K. Chrysanthis. Personalized Information Gathering for Mobile Database Clients. In *Proc. of The ACM Symposium on Applied Computing*, pages 49–56, February 2002.

[9] D. S. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. B. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF: The OMG Mobile Agent System Interoperability Facility. *Mobile Agents*, 1477:50–67, 1999.

[10] S. Papastavrou, P. K. Chrysanthis, G. Samaras, and E. Pitoura. An Evaluation of the Java-Based Approaches to Web Database Access. In *Proc. of the Int'l Conference on Cooperative Information Systems*, pages 102–113, September 2000.

[11] S. Papastavrou, G. Samaras, and E. Pitoura. Mobile Agents for World Wide Web Distributed Database Access. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):802–820, 2000.

[12] Recursion Software Inc. Voyager ORB. Available at www.recursionsw.com/products/voyager/.

[13] N. Roussopoulos. Materialized Views and Data Warehouses. *SIGMOD Record*, 27(1):21–26, 1998.

[14] G. Samaras, C. Spyrou, and E. Pitoura. View Generator (VG): A Mobile Agent Based System for the Creation and Maintenance of Web Views. In *Proc. of the 7th IEEE Symposium on Computers and Communications*, pages 761–767, July 2002.

[15] G. Samaras, C. Spyrou, E. Pitoura, and M. Dikaiakos. A Universal Location Management System for Mobile Agents. In *Proc. of European Wireless Conference, Next Generation Wireless Networks: Technologies, Protocols, Services and Applications*, pages 25–28, February 2002.

[16] C. Spyrou, G. Samaras, E. Pitoura, S. Papastavrou, and P. K. Chrysanthis. The Dynamic View System (DVS): Mobile Agents toSupport Web Views. In *Proc. of the 17th Int'l Conference on Data Engineering*, pages 30–32, March 2001.

[17] SUN Microsystems. Java Remote Method Invocation Technology (Java RMI). Available at http://java.sun.com/products/jdk/rmi/.

[18] World Wide Web Consortium. Web Services. Available at http://www.w3.org/2002/ws/.

[19] O. Wolfson, A.P. Sistla, S. Dao, K. Narayanan, and R. Raj. View Maintenance in Mobile Computing. *SIGMOD Record*, 24(4):22–27, 1995.

[20] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet. An Infrastructure for Collaborating MobileAgents. In *The First Int'l Workshop on Mobile Agents*, pages 86–97, April 1997.

[21] XXL Project. XXL: eXtensible and fleXible Library. Database Research Group, Univ. of Marburg, Germany, Available at http://www.mathematik.uni-marburg.de/DBS/xxl.

[22] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proc. of ACM SIGMOD Conference*, pages 316–327, May 1995.

[23] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. Algorithms for Multi-Source Warehouse Consistency. In *Proc. of the 4th Int'l Conference on Parallel and Distributed Information Systems*, pages 146–157, December 1996.