Coordinating Mobile Databases

Fausto Giunchiglia and Ilya Zaihrayeu
{fausto, ilya}@dit.unitn.it

Dept. of Information and Communication Technology
University of Trento, Italy

Abstract. We are interested in the development of a database management layer which is completely portable and, therefore, "pluggable" on top of multiple host platforms. This layer, so called *Peer Database Management System* (PDBMS), must be able to remotely connect with its database and to connect it with other peer databases. We realize mobility by storing PDBMS on a flash drive. We realize network independence by developing a fully decentralized data coordination model. The two notions at the core of our model are Interest groups and Acquaintances. The first notion allows for a global aggregation of nodes carrying similar information, while the second allows for a local logical point-to-point data exchange between databases. The system has been developed on top of the Peer-to-Peer platform JXTA.

1 Introduction

We are interested in the study of new paradigms allowing conventional database technology to be effectively operational in mobile settings. We think of database mobility as a database network, where databases appear and disappear spontaneously and their network access points may change, and are not known a priori. There is a further request that databases must know, independently of their network access points, how to locate other databases, and how to interoperate with them on servicing user requests (i.e., queries and updates).

Examples of the application domain for mobile databases are medical care or the real estate domain. For instance, think about a person, called John, who goes for skiing and suffers an accident. He is taken to a local clinic for a treatment. Doctors need to know whether John has any contra-indication against some particular drugs. John does not know these details, but luckily his database management layer, that he keeps on his flash drive, has a link to the family doctor's database, where the history of his treatments is stored. Thus a simple query helps to solve the problem. Another example is an application where real estate agents coordinate their databases in exchanging real estate information with the goal of pushing sales. Since they travel to their customers (who may

This work is partially funded by the IST programme of the EU Commission under the KnowledgeWeb project and by the FIRB programme of MIUR under the RBNE0195K5 ASTRO Project.

want to sell or, instead, to buy), they always carry relevant data with them. When one is on the site of a customer, who wants to sell a house, the agent updates his database and makes this data available for other agents. Or, when an agent talks with a potential buyer, and nothing from the agent's database satisfies the client, the agent may want to query other agents' databases to look for additional sale options.

We implement these functionalities within a so called *Peer Database Management System* (PDBMS) which is supposed to run on top of a standard database management system. We see three major requirements for a PDBMS. First, it must be self-contained and relatively small in size. This allows it to be totally independent from other databases and middleware. It also allows it to fit on a small capacity storage device as a flash drive, which can be easily handled around. Second, it must be system, networking platform, and IP independent; this in order to make it "pluggable" on top of multiple host platforms. Third, PDBMS must know how to connect to its database and how to connect it with other databases in order to exchange queries and data.

PDBMS runs on top of JXTA [11]. JXTA provides an IP-independent naming space to address nodes. It implements a *Peer-to-Peer* (P2P) decentralized networking model where each party (called a *node* or a *peer*) has equivalent abilities in providing other parties with data and/or services. Peers are largely autonomous from other peers, and they interoperate in a local, point-to-point manner. All these notions are crucial from the point of view of mobility – databases may come and go, interact with different databases at different times or for answering different queries, the size of the network can dynamically shrink and expand depending on how many nodes are online, and databases can benefit from collaboration with one other by coordinating their data at runtime.

Existing database integration solutions are inapplicable for our application domain. In conventional database integration technology, interoperation is reached by means of introducing the notion of a global virtual schema [13]. Queries are posed against the global schema and then reformulated w.r.t. local schemas which describe real data in the system. Should a new schema be imported to the system (which is not supported by the global schema), then the global schema must be reconsidered as a design time activity. The dynamic factor of mobility, i.e. the fact that parties are regularly unavailable, and the open-ended autonomous nature of P2P make these solutions impractical. Therefore, we propose a new solution to P2P databases, that we call *database coordination*. We see coordination as the ability of peers to effectively manage, at runtime, *semantic data dependency links* among databases in a decentralized, distributed and collaborative manner.

The key notions of the database coordination model are *Interest Groups* and *Acquaintances*. Interest groups support the formation of peers according to data models they have in common; and acquaintances allow for peers inter-operation. The combination of database and P2P technologies has already received a lot of attention, see for instance [9, 5, 7, 10]. Among many other things (see [4] for a detailed discussion of the related work) our solution considers a new dimension

for P2P databases – mobility, where PDBMS, database, or both, can be mobile. [4] provides the vision of our approach, whereas this paper makes it concrete by presenting the key notions, algorithms and architecture at a reproducible level of details.

This paper is organized as follows. Section 2 introduces the four basic architectural notions of our data coordination model. Section 3 explains how these notions are implemented in JXTA. Section 4 discusses the logical architecture that we propose. Finally, Section 5 gives the conclusions.

2 A model for data coordination

We consider the notion of a *DB peer* (or just *peer*) as primitive, taking it as any device supporting one or more networking protocols. There is a further request that each peer provides a *source database* described by a (*source*) *schema*, or supplies only the schema. In this latter case a node acts as a kind of *mediator* in transitive propagation of data. Peers define *semantic data dependency links* between their schemas and use these links to coordinate data, i.e., answer input queries, propagate query results and updates. An input query can come from a user (*user query*) or from another node on the network (*network query*). Both user queries and network queries are formulated w.r.t. the source schema of a particular node. A *P2P database network* (or just *network*) is a collection of DB peers and semantic data dependency links relating schemas of pairs of nodes. Peers are largely autonomous, in particular in what data they store, in which nodes they establish semantic data dependency links with and coordinate their data, etc. We define data coordination in terms of four basic notions. They are: *Interest Groups*, *Acquaintances*, *Correspondence Rules* and *Coordination Rules*.

2.1 Interest Groups

Usually, nodes know very little about the *topics* other nodes can answer queries about. Intuitively, "Ford cars", "Trentino libraries", "Movies" are all possible topics. An *Interest Group* (or a *Peer Group*) is a set of nodes able to answer queries about a particular topic. Interest groups form a hierarchical parent-child relationship, where each child has only one parent. Each child peer group has a more specialized topic w.r.t. the topic of its parent group. The interest groups hierarchy is a tree, where the root stands for "All Topics" interest group (ATG). ATG is used by nodes, whose schemas do not correspond to any of the other groups in the hierarchy. The lower a group in the hierarchy, the more specialized its topic is and the more "specific" queries its nodes are able to answer. We describe the topic of an interest group by the path from ATG to that interest group in the hierarchy and by a set of keywords which additionally describe the content of the group. This information is useful for peers looking for appropriate groups to join. Consider the following example.

Example 1. Consider Figure 1, where an example of interest group hierarchy is depicted. The "Arts" group includes nodes with databases storing rather general

data about arts, for instance, categories of arts (music, photography, etc), addresses of relevant museums and names of relevant people, etc. Its child group, titled "Movies", contains information about particular movies, directors and actors, producers, and so on. The topic for group "Movies" in this case could be $GT(\text{“}Movies'') =< ATG-Arts-Movies, \text{“}movies, cartoons, video, celebrities'' >$

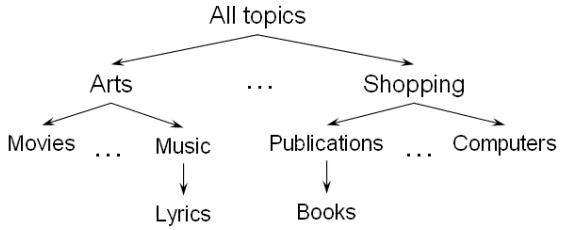

Fig. 1. An interest group hierarchy

Each interest group has a Group Manager (GM) whose task is to provide peers with information about the group. In particular, GM is responsible for collecting statistical information, such as the number of nodes which have joined the group, the number of nodes which are online in average, average number of queries per node in a unit of time, volume of data in the nodes' databases, average volume of a query answer, and so on. GM keeps up-to-date this information and makes it available for nodes willing to join. GM stores the topic information of its interest group as well the data about its child interest groups. A child interest group is formed by a node of some (parent) interest group. This node becomes the group manager of this new group. Once a new group is created, new nodes may join, and, eventually form new child interest groups. Nodes self-organize into interest groups and build the interest group hierarchy in a decentralized fashion. As a thumb rule, any peer can participate in more than one interest group independently on their position in the hierarchy (providing that this peer has relevant relations for these groups), and can join and leave groups spontaneously.

2.2 Acquaintances

Acquaintances are nodes a node knows about. There is a further request that a node must know how to translate an input query (or its part) w.r.t. the schema of an acquaintance and how to translate query results or an update coming from an acquaintance w.r.t. its source schema. The acquaintance is not a symmetric notion, i.e. the fact that a node is acquainted with another node does not necessarily mean that the vice versa also holds. A node is an *acquainted node* for some other node if the latter is an acquaintance of the former.

For each acquaintance, a node keeps one or more *acquaintance queries*. An acquaintance query is a query over the relations of the database of an acquaintance, whose answer satisfies one of the relations of the schema of an appropriate

acquainted node. In data integration terminology this kind of mappings is called *Global-as-View*, or GAV [6]. An acquaintance query is the minimal block for building semantic data dependency links between peer databases. We represent acquaintance queries as *conjunctive queries*, which can express select-project-join queries [13]. All input queries in a network are also represented in the conjunctive form. An acquaintance query (or an input query) has the following form:

$r(\overline{X}) :- r_1(\overline{X}_1), \ldots, r_i(\overline{X}_i), c_{i+1}(\overline{X}_{i+1}), \ldots, c_n(\overline{X}_n)$

where r, $r_1, \ldots, r_n$ and $c_{i+1}, \ldots, c_n$ are predicate names. $r(\overline{X})$ is called the *head* of the query and refers to a particular relation. $r_1(\overline{X}_1), \ldots, r_i(\overline{X}_i)$ are the *relation subgoals*, and $c_{i+1}(\overline{X}_{i+1}), \ldots, c_n(\overline{X}_n)$ are the *comparison subgoals* of the *body* of the query. Comparison subgoals stand for arithmetic comparisons, such as $<, \leq, =, \neq$. Tuples $\overline{X}, \overline{X}_1, \ldots, \overline{X}_i$ contain variables, whereas tuples $\overline{X}_{i+1}, \ldots, \overline{X}_n$ contain either constants or variables from the relation subgoals. Variables from $\overline{X}_1, \ldots, \overline{X}_n$ are called *body variables*, and variables from $\overline{X}$ are called *head variables*. We allow only meaningful queries, i.e. we require that all head variables must also be body variables, names of the relation subgoals must be mutually different, etc. If a query respects these rules then we say that that this query is *valid*.

The head of an acquaintance query is equal to some relation of the source schema, and its relation subgoals and body variables refer respectively to the relations and attributes of the schema of an acquaintance. Comparison subgoals of an acquaintance query represent restrictions over the domain of the acquaintance database which are caused by the source database domain constraints; they are used for update propagation. For an input query, the head is equal to the answer relation, and the relation subgoals refer to the relations of the source schema.

Acquaintance queries are used for the translation of input queries and their propagation to acquaintances as well as for update propagation from acquaintances. Each acquaintance query is stored in two copies – one at an acquainted node and another at its respective acquaintance. We will say that an acquaintance query is *relevant* for some relation, if either it is stored at an acquainted node and the head of the query is equal to that relation, or if it is stored at an acquaintance and that relation appears in the body of the query. Consider the following example:

Example 2. Consider two nodes A and B, whose relations store information about banks (as names, departments, cities, countries) and bank employees (as names, departments they are attached to, and salaries). The relations of the two nodes are:

node A	node B
$Banks(bankName, city, country)$	$Banca(nome, stipendio, banca, cita')$
$Departments(depName, bankName)$	
$Staff(name, depName, salary)$	

A particularity of database B is that it stores information about employees working in Italian banks with salaries more than 2000 Euros. Node B is acquainted with node A w.r.t. the following acquaintance query:

$$Banca(n, s, b, c) :- Banks(b, c, ctr), Departments(dn, b), Staff(n, dn, s), \\ ctr = \text{“}Italy'', s > 2000$$

As it can be seen from the acquaintance query, two comparison subgoals apply restrictions on the salary and country attributes of relations of A in accordance with the domain of the database of node B.

2.3 Correspondence Rules

In most cases, participating databases are semantically heterogeneous, namely, they represent the same concepts differently [8]. Correspondence rules define how constants from the local domain are translated into constants in the domain of an acquaintance (forward translation) and vice versa (backward translation). Correspondence rules are defined on the domains of all head variables of acquaintance queries (since they appear in input queries) and of all those body variables which appear in the comparison subgoals. Note, that correspondences between relations and attributes are already defined by the acquaintance queries. Namely, one relation in a peer source schema may correspond to several relations in an acquaintance's one. Also an attribute in a relation (i.e. a head variable) may correspond to several attributes in acquaintance's relations (i.e. body variables).

We denote by $FT(x) = y$ a function for forward translation and by $BT(x) = y$ a function for backward translation. Forward translation is used for translating constants in queries, whereas backward translation is used for translating constants in query results and updates. Note, that these two translations are not necessarily symmetric, i.e. it is not necessarily true that the following holds $FT(BT(x)) = x$. An example of this situation could be a translation between two currencies. In the simplest case, when two nodes share the same domain, constants are translated by correspondence rules into themselves, i.e. $FT(x) = BT(x) = x$. In this case we will say that such correspondence rules are *plain translations*. Consider the following example.

Example 3. Recall the nodes, relations and acquaintance query from Example 2. The variables, which are associated with correspondence rules, are n, s, b, c and ctr. Now imagine that node A is in the United States, and therefore all salaries are in US dollars. In order to allow queries with the salary attribute in the body, the following correspondence rules might be set up at B for this acquaintance query:

$FT(s) = s * 1.27$; $BT(s) = s * 0.78$;

Correspondence rules for variables n, b, c and ctr are plain translations since we assume that both databases use the same natural language (e.g. English) to refer to personnel, bank, city, and country names.

Apart from the translation of constants, correspondence rules do another operation, namely transformation of a relation subgoal of an input query into a query w.r.t. the schema of a particular acquaintance. Depending on the number of acquaintance query definitions, bound to some relation subgoal, each subgoal may have zero, one, or more transformations. The transformation consists of two main phases: (1) unfolding of the relation subgoal in accordance with the definition of a respective acquaintance query; (2) "pushing" the comparison subgoals to the reformulated query. In the first phase we omit all comparison subgoals from the acquaintance query, and, in the second phase, we add only those from the input query, which refer to the head variables from the relation being translated. There are two reasons for doing this. First reason is to allow queries which ask for data beyond local domain constraints. If comparison subgoals of acquaintance queries were added, resulting query might already become unsatisfiable. The second reason is that, by "pushing" the input query comparison subgoals to a query w.r.t. acquaintance database, we reduce the amount of data, returned by the acquaintance. On the completion of the second phase, correspondence rules translate constants, if any. Consider the following example:

Example 4. Assume that the user of node B from Example 2 submits a query asking for the names of employees in Rome with salaries less then 1800 Euros. The corresponding conjunctive query is:

$$Q_B(n, b) : -Banca(n, s, b, c), s < 1800, c = \text{“}Rome''$$

According to the definition of the acquaintance query from Example 2, formulated for relation "$Banca''$", Q_B is unfolded to the following query:

$$\begin{aligned} Q'_B(n, b) : - \; & Banks(b, c, ctr), Departments(dn, b), \\ & Staff(n, dn, s), s < 1800, c = \text{“}Rome'' \end{aligned}$$

Note, that comparison subgoals from the acquaintance query are omitted, whereas the ones from the user query are added. If the comparison subgoals of the acquaintance query were left, then we would have two comparison subgoals which are mutually inconsistent (i.e. $s > 2000$ and $s < 1800$). Finally, we apply correspondence rules for translating constants 1800 and "Rome", and we get the following query, ready for being executed at node A:

$$\begin{aligned} Q_A(n, b) : - \; & Banks(b, c, ctr), Departments(dn, b), \\ & Staff(n, dn, s), s < 2286, c = \text{“}Rome'' \end{aligned}$$

2.4 Coordination Rules

Each node has a set of *coordination rules*. Their primary goal is managing *data coordination* with acquaintances and acquainted nodes. They are run by special kind of events, called *data coordination events* and, depending on the event, perform a particular action. The data coordination events are: (1) a database manipulation operation, such as *select*, *insert*, *delete* or *update* formulated w.r.t. to the source schema and submitted by the user; (2) a network query coming from an acquainted node; (3) query results coming from an acquaintance node;

or (4) an update request coming from an acquaintance. An action, performed by coordination rules, can be transformation and propagation of an input query to an acquaintance, reconciliation of results and their propagation to an acquainted node, etc. Let us consider how coordination rules process the four data coordination event types.

Database manipulation operations. SELECT is used when a user submits queries. User queries are checked for validity, and then they are evaluated against the source schema. The evaluation involves a check of whether referenced relations exist, whether they are given certain number of variables, etc. If a node is equipped with a source database and it is accessible, then the variables in the conjunctive query are assigned corresponding attributes, and then the query is submitted to the database.

Then, for *each* relation subgoal in the query coordination rules check whether there are any relevant acquaintance queries. For each found acquaintance query, correspondence rules are applied to get a transformed query w.r.t. the schema of the appropriate acquaintance. Finally, all reformulated queries for all subgoals, which passed the consistency check, are propagated to appropriate acquaintances. Since different acquaintance queries may refer to nodes from different interest groups, a user query may be propagated within several interest groups at the same time.

INSERT leads to an update of data at some source database. When executed at some node, it may evoke update propagation to some other, acquainted node(s). Nodes may agree on automatic propagation of updates related to some acquaintance queries. An update propagation works as follows: when an insert operation is performed, coordination rules look for acquaintance queries of acquainted nodes, which contain the relation being modified amongst their relation subgoals. For each acquaintance query found, the node computes the acquaintance query keeping only the newly inserted tuples for the corresponding relation. The reason for this is that in this case the query computes only new tuples, w.r.t. to ones which might have been already computed during the previous insert operations or queries. Note, that now the comparison subgoals of acquaintance queries are also used in the computation of result tuples (which is not a case in query answering). Then, the node sends the computed tuples to respective acquainted nodes, with the IDs of corresponding acquaintance queries.

DELETE removes tuples from a relation. Nodes may agree also on the automatic propagation of delete updates. For each relevant acquaintance query, we treat tuples, which are in the difference of the query computation results before and after the delete operation, as candidates for being deleted from the corresponding relation at the respective acquainted node. In order to compute them, before executing the delete operation on some relation, we query that relation asking for all attributes where the condition part is equal to the one of the delete operation. Then we compute the acquaintance query involving the result of the last query. Finally, the result of the computation is sent to the acquainted node

in an update message with the ID of corresponding acquaintance query with the mark for deletion.

UPDATE may cause three different actions to be performed on the database of an acquainted node. Namely, an update may produce new tuples for an acquainted peer due to the fact that some comparison subgoals are now satisfied; it may lead to a deletion of some tuples because some comparison subgoals are no longer satisfied; or, it could lead just to changes in some existing tuples. In order to handle the update operation, we proceed as follows. First, we query for all attributes of the relation R being updated with the condition clause in the query equal to the one of the corresponding update operation, and get the result set A. Then, we compute each relevant acquaintance query by substituting R with A, and get the result set B. B contains tuples which might have been already computed and propagated to a particular acquainted node. The update operation is performed over R and A. Then, we re-compute relevant acquaintances over updated A and get the result set B'. Finally, for all tuples which are in B, but not in B' the node sends a delete request message, and for all tuples which are in B' and not in B the node sends an insert request message. Note, that at the moment tuples which just need to be changed are updated by a delete-insert sequence.

Network query. Coordination rules process network queries in the same way as user queries with only two differences. First, the results of querying the database are sent back to the acquainted node, which sent the query. Second, queries are allowed to be propagated only to the nodes of the same interest group. In order to do this, all network queries are sent with the interest group ID which identifies the scope of further propagation.

Query results handling. Query results coming from an acquaintance can be seen as *additional tuples* for some relation appearing in an input query. It is crucial to compute *new* tuples for this input query (new tuples w.r.t. previously computed results). In order to reach this goal we proceed in two steps. Analogously to the updates case, we *substitute* existing tuples of the given relation by the new results and re-compute the input query. Then, from the newly computed tuples we delete those, which are duplicates of the previously produced results. In order to do this we store input query computation results until the query answering is complete. Finally, after deleting the duplicates, the remaining tuples are either reported to the user, or propagated backward to the appropriate acquainted node.

Update requests handling. An update message is a request for an insert or for a delete. When a node receives such a message, it translates it using correspondence rules and updates the corresponding relation. With an insert, the node updates its relation avoiding inserting duplicates. Analogously to query answering, coordination rules look for relevant acquaintance queries, and, depending on the kind of the update (insert or delete), initialize corresponding

update propagation procedures. Update propagation is different from query answering in that an intermediate node may decide not to accept an update and thus stops further update propagation.

2.5 Data coordination

A fundamental question in data coordination is how nodes cooperate *globally* (in the scope of an interest group) in the overall processing of an initial data coordination event, such as a user query or an update. A query or an update at a node may lead to its propagation to some other nodes, they in turn may propagate it further, and so on. At the network level, we see data coordination as a *transitive* propagation of data via chains of nodes as the result of local point-to-point interactions of nodes with their acquaintances and acquainted nodes. Since nodes are free to make acquaintances with any other nodes they like, the "acquaintance" topology of the network may be absolutely arbitrary. In such settings it becomes crucial to process correctly *loops* in order to avoid indefinitely long propagation of data, and to determine when query answering (or update propagation) is complete.

We handle loops in query and update propagation differently. We avoid them in query propagation by propagating with a query a path, consisting of acquaintance query IDs, which have been used for propagation of the query. Once a node receives a query, it does not propagate it further using whose acquaintance queries which IDs are already in the path. However, this allows for a query to pass over the same node more than once. This is possible when a query comes to a node via different paths of nodes, or, when there is a loop in the path of nodes, and different acquaintance queries are used at the same node but at different times for further propagation of the query.

In order to clarify when query answering is complete, we introduce some additional notions. Namely, we call acquaintance queries, *incoming links*, if these acquaintance queries are used by some acquainted nodes for querying the source databases of the given node. We call acquaintance queries, *outgoing links*, if that node uses these acquaintance queries to translate and propagate queries to its acquaintances. We say that an incoming link *depends* on an outgoing link, or that an outgoing link is *relevant* for some incoming link, if amongst the relational subgoals of the incoming link there is a relation appearing in the head of the outgoing link. Importing data tuples from an outgoing link may produce new tuples for all incoming links which depend on that outgoing link.

Query answering for some query at a given node is complete, if all outgoing links used for the propagation of the query from this node are in the state "closed". When an outgoing link is used for propagation of a query, its state initially is "open", which means that new tuples may be imported using this link. An acquaintance "closes" an incoming link if: (a) there is no further propagation of the query from this node, or (b) all its outgoing links which are relevant for this incoming link are in the state "closed". When all outgoing links of a node are in the state "closed", the node becomes also "closed". The query answering for some query is complete when all nodes participated in answering of this query

are "closed". Note, that the node where a user query was originally submitted gets to the "closed" state last. It worths saying that this algorithm guarantees termination with the presence of loops in the network topology.

Handling of loops in update propagation is more complex than in query answering. Since nodes update their databases, re-computing acquaintance queries, after an update reached a node following a loop of nodes, may produce new tuples to be further propagated into the loop. Thus, an update sequence may go through a loop of nodes several times until no more new tuples are produced for any of the nodes in the loop. A node stops update propagation if this update brings no new tuples for this node, and this node is in the sequence of nodes which propagated this update. For doing this, when each node propagates an update, it adds itself to the nodes sequence and sends it with the update. In the rest, the termination of update propagation is determined analogously with the query propagation case. For a thorough discussion on how updates are handled in a P2P database system see [2].

The second important question in data coordination is how acquaintances, correspondence rules and coordination rules are actually formed at runtime. Nodes of an interest group may search for other nodes of the group for the purpose of making acquaintances with them. The process of making an acquaintance, involving the creation of acquaintance queries, correspondence rules and coordination rules, is called the *getting acquainted protocol*. The protocol works as follows: a node (say, node A) locates a potential acquaintance (node B), it retrieves the source schema of B and matches it with its own source schema (see [3, 12] for a discussion on schema matching techniques). The matching results show how the elements (i.e. relations and attributes) of one schema correspond to the elements of another. This information is used by the system to build (likely with the help of the user) acquaintance queries and correspondence rules. In order to "activate" coordination rules for an acquaintance (or acquainted node) it is sufficient to provide relevant acquaintance queries.

3 Implementing data coordination in JXTA

JXTA provides an open set of protocols which allow to build P2P applications. *JXTA peers* are devices which implement one or more JXTA protocols. JXTA-powered applications, amongst other things, can: create groups of peers, locate peers on the network, create messages, where a message can carry arbitrary type of data (e.g. images, code, query results, etc), create communication links (called *pipes*) with other nodes and send messages onto pipes. The pipe *endpoints* are referred as the *input pipe* (the receiving end) and as the *output pipe* (the sending end). Pipe endpoints correspond to available peer network interfaces (e.g., TCP port and IP address). JXTA allows for the definition of a set of *services* that a peer makes available for other peers. Services fall into two categories: *peer services* and *peer group services*. Peer services are provided by single peers, and, should a peer fail, the service also fails. Peer group services are provided by a collective set of peers, and, should a peer fail, the service does not fail assuming

that there are other peers providing this service. JXTA defines the core set of services necessary for a full-functional operation of a peer. Some of them are: *Discovery Service*, *Membership Service* and *Pipe Service*. The Discovery Service allows peers to locate and publish information on the network. The Membership Service is used by current members of a peer group to reject or accept a new group membership application. The Pipe Service allows peers to create pipes with nodes from the same group. Apart from this, JXTA uses sophisticated algorithms to generate unique (for an interest group) IDs to identify various resources. We use this machinery to generate IDs for queries, acquaintance queries, update propagation, and so on.

New peer groups may include (a subset of) the core services as well as *custom services*. Custom services allow for the creation of peer groups which will provide their peers with desirable functionality. A JXTA peer group is a set of peers which agree on the common set of services. All *network resources* in JXTA (i.e. peers, peer groups, pipes, etc) are described by *advertisements*, which are language neutral XML documents. Peers can publish, discover and use advertisements (e.g. create a pipe from an advertisement). JXTA provides an IP-independent naming space to identify network resources, and supports various network protocols, such as TCP/IP or Bluetooth. Moreover, since JXTA is implemented in Java, it is platform independent and can run both on Windows and on Linux. As discussed in Section 1, all this is very important from the point of view of mobility. Namely, a peer can enter the network from different places, use available platforms and networking protocols and it will be easily located and identified by other peers.

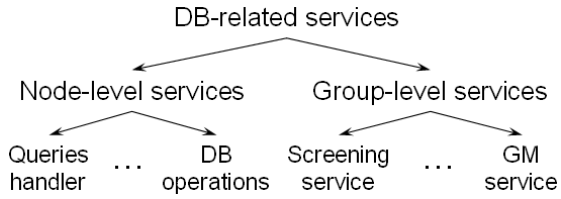

Fig. 2. Classification of DB-related services

We implement database coordination by implementing *locally* at each node the four basic notions of our data coordination model. In order to do this we use JXTA to implement peers, interest groups and acquaintances. Coordination rules and correspondence rules are implemented in the application software. We implement peers as JXTA peers. We extend the standard JXTA peer advertisement to encapsulate the source schema information of a peer. Once an advertisement of a peer is located by another node, the latter extracts the schema and executes the getting acquainted protocol. Finally, if the user finds the results relevant and wants to make an acquaintance, then, the two nodes exchange pipe advertise-

ments, and create input and output pipes. At this moment, one node is said to be acquainted with another.

We implement interest groups as JXTA groups by encoding database related functionalities into the custom set of services, called *DB-related services*. We extend the standard JXTA peer group advertisement to include the group topic information. We classify DB-related services into two categories: *node-level services* and *group-level services* (see Figure 2). There are several node-level services. For instance, there is a node-level service which creates pipes with acquainted nodes and listens to their endpoints. And, as soon as a network query arrives, the service processes it as it is described above in the corresponding section. Another node-level service implements handling of query results, coming from acquaintances. If the source databases is absent, this service uses another node-level service, that computes input queries from the results received from acquaintances. Note, that this service needs to perform only the *join* and *project* operations. Since comparison subgoals are "pushed" to acquaintances, and first data tuples are computed in real databases, then all tuples coming from acquaintances *already satisfy* the comparison subgoals of input queries, and therefore the *select* operation is not required.

One example of a group-level service is the *Screening Service*, obtained by modifying the JXTA Membership Service. The service helps supporting a proper constitution of an interest group w.r.t. its topic. A peer willing to join an interest group, first locates a current member, and then applies for membership, providing its schema as credentials. An application is accepted or rejected by a collective set of current members. In order to do this, each peer matches the schema of the newcomer with its source schema (without involvement of the user) and returns the number of discovered mappings. If the aggregated result for all available nodes is above a certain threshold then the membership application of a new peer is accepted and rejected otherwise. Another example of a group-level service is the GM service. This service is responsible for sending to group manager all information necessary for GM to run the group. As long as there are running peers in the group, these two services are available.

We build DB-related services on top of the core services provided by JXTA. As a consequence, the implementation of the basic P2P functionality (i.e., discovery, pipes, etc) is given. Each DB peer has a copy of both the core JXTA services and DB-related services. This helps us to implement the self-containment requirement as specified in the introduction. JXTA peer groups form a hierarchical parent-child relationship. This fact allows us to support the formation of interest group hierarchies, as shown on Figure 1.

4 The logical architecture and implementation

We describe the logical architecture at two levels of detail: the architecture of a node (first level); and the second level, which shows how the four basic notions are implemented in JXTA. The first level architecture is a variation of the high level architecture first reported in [1].

Consider Figure 3. A node consists of *PDBMS*, a *Source Database* (SDB) and a *Source Schema* (SS). SS describes a shared part of SDB. PDBMS consists of *User Interface* (UI), *Database Manager* (DBM), *JXTA Layer* and *Wrapper*. DBM implements the four basic notions described in Section 2. JXTA Layer is responsible for all node's activities on the network, such as discovering of new nodes and interest groups, joining and leaving groups, communication with group managers, sending and receiving queries and query results, and so on. Wrapper manages connections to SDB, it is responsible for extraction and maintenance of SS. Since different nodes may use different databases, this module is adjustable depending on the underlying database.

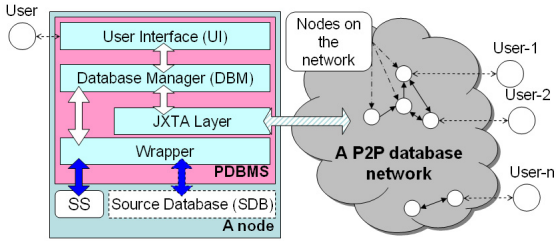

Fig. 3. First level architecture: a node

Arrows between UI and DBM as well as arrows between Wrapper, JXTA Layer and DBM have the same graphical notation bacause they represent procedure calls between different modules. Bidirectional arrow from JXTA Layer to a P2P database network has a different notation because it represents JXTA-supported messaging, mostly represented in the form of XML documents. The arrows between SDB, SS, and Wrapper have yet another notation because the communication is SDB dependent.

In Figure 4 we "open" DBM and JXTA Layer. Rectangles with rounded corners stand for data repositories which store various information. Normal rectangles represent executive modules. The meaning of arrows between UI, DBM, JXTA Layer and Wrapper is the same as in Figure 3, namely, they represent procedure calls. Continuous thin arrowed lines show information flows between modules and data repositories, as well as procedure calls between modules. Dashed arrowed lines show the functional dependencies between components. For example, they show that coordination rules, correspondence rules, acquaintance queries, peer advertisements and pipes all functionally depend on acquaintances.

Consider JXTA Layer. The advertisements repository stores all discovered and locally created JXTA advertisements. Inside the rectangle, three advertisement types are represented, although in practice there are also others. The peer group advertisement includes also the group topic information, and the peer advertisement includes the source schema information. The Services module implements the core JXTA services and DB-related services. We encode the input

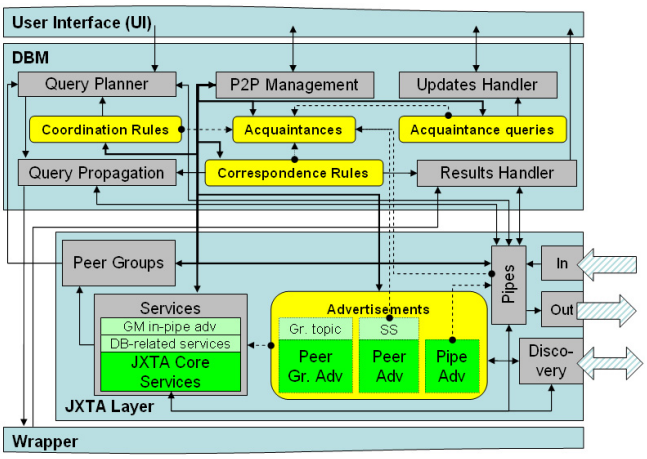

Fig. 4. Second level architecture: DBM and JXTA Layer

pipe advertisement of the group manager in the Services module. Nodes use this information to contact their appropriate GM. The Discovery module implements the Discovery Service, and the Pipes module implements the Pipe Service.

Consider now DBM. The P2P Management module allows users to control other modules and repositories from both DBM and JXTA Layer. For instance, it makes it possible to create a new pipe, to make a new acquaintance or to modify a coordination rule. The control lines are shown as thick arrows from P2P Management to other components. Query Planner processes all input queries. It uses acquaintance queries, acquaintances and interest groups information in order to detect groups and nodes for propagation. The Query Propagation (QP) module takes this information as input and uses correspondence rules for query rewriting. Finally, it uses pipes to send translated queries to acquaintances. When necessary, QP submits queries to the source database. Results Handler receives results coming from acquaintances and translates them using Correspondence Rules. If these results are for a user query, then it reports them to UI. Otherwise, it sends them backward to the node which sent respective network query. Apart from this, Results Handler gets results coming from Wrapper, and sends them either to UI or to the network. Finally, Update Handler provides all functionality necessary for updates processing.

The current version of the prototype implements a major part of the ideas described in this paper. In particular, acquaintances and coordination rules, as well as query and update propagation algorithms are fully implemented except some minor details. Interest groups and correspondence rules are not fully implemented at the moment. The prototype is implemented in Java and is about 6 Mbytes in size including required JXTA libraries and excluding all meta-data files (e.g. source schemas, JXTA advertisements, etc). The Java Virtual Machine runtime environment (about 40 Mbytes) is required to run the application. Thus

a self-contained application package can fit in space of about 46 Mbytes, which can be easily placed on a flash drive. The results of the first experiments show reasonable query answering and update propagation times in small size networks (up to 20 nodes). For the experiments we created various source databases with several thousand of tuples at each node, with different degrees of overlapping of the data at different nodes. Future work includes the study of the scalability property of our solution, as well as the implementation of interest groups and correspondence rules.

5 Conclusions

In this paper we have proposed a new solution for P2P databases which is applicable in mobile settings. The solution allows for the coordination of peer databases, where PDBMSs, databases or both can be mobile. We have demonstrated how data coordination can be implemented exploiting the four basic notions, namely Interest Groups, Acquaintances, Correspondence Rules and Coordination Rules. Finally, we have shown how our solution can be implemented in JXTA and proposed a logical architecture at two levels of detail.

References

1. P. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos andL. Serafini, and I. Zaihrayeu. Data management for peer-to-peercomputing: A vision. *WebDB*, 2002.
2. E. Franconi, G. Kuper, A. Lopatenko, and I. Zaihrayeu. A distributed algorithm for robust data sharing and updates in p2p database networks. *Proceedings of the P2P&DB international workshop, Heraklion - Crete, Greece*, March 2004.
3. F. Giunchiglia and P. Shvaiko. Semantic matching. *"Ontologies and Distributed Systems" workshop, IJCAI*, 2003.
4. F. Giunchiglia and I. Zaihrayeu. Making peer databases interact - avision for an architecture supporting data coordination. *6th International Workshop on Cooperative Information Agents(CIA-2002), Madrid, Spain, September18 -20*, 2002.
5. S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What can databases do for peer-to-peer? *WebDB, Workshop on Databases and the Web*, June 2001.
6. A. Halevy. Answering queries using views: a survey. *VLDB Journal*, 2001.
7. A. Halevy, Z. Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semantic web applications, 2003.
8. R. Hull. Managing semantic heterogeneity in databases: A theoretical perspective. *Bell Laboratories*, 1997.
9. A. Kementsietsidis, M. Arenas, and R. Miller. Data mapping in peer-to-peer systems. *ICDE*, 2003.
10. W. Ng, B. Ooi, K. Tan, and A. Zhou. Peerdb: A p2p-based system for distributed data sharing. *ICDE*, 2003.
11. JXTA project. see http://www.jxta.org.
12. E. Rahm and P. A. Bernstein. On matching schemas automatically. *VLDB Journal 10, 4*, Dec 2001.
13. J. Ullman. Information integration using logical views. *Theoretical Computer Science*, 1997.