Support for Mobile Location-aware Applications in MAGNET

Dr. Patty Kostkova[1] and Dr. Julie McCann[2]

[1] Department of Information Science, Institute of Health Sciences, City University, London, UK
patty@soi.city.ac.uk
[2] Department of Computing, Imperial College, London, UK
jamm@doc.ic.ac.uk

Abstract. A key characteristic of mobile applications is the need for up-to-date location-dependent information, while the physical location changes frequently. Recent improvements in wireless communication and hardware technology and Internet-based data exchanged created a new type of mobile applications whose requirements are not met by traditional relational and object database systems. In this paper we describe MAGNET, a tuplespace-based framework for dynamic information storage and retrieval addressing the needs of application in frequently changing mobile environment and discuss how this approach could enable flexible data exchange. In addition to type-free data storage and user-customized requests for data records, MAGNET enables adaptation to a changed environment by supporting constant monitoring of selected information.

1 Introduction

In the business climate, an increasing number of people are expected to perform complex work-related tasks while on the move. Change in physical location results in the volatility of location-dependent information (e.g., the local time, the nearest library). Therefore, a key requirement of mobile users[1] is the retrieval of dynamically updated location-dependent information. The need for database support for *mobile applications*[2] has become important since frequent travelling has become commonplace. Only recently have improvements in hardware support for wireless computing enabled mobile application requirements to be fulfilled. Key advances in hardware technologies allowing the current boom in mobile computing include: improvements in reliability, speed and coverage of wireless communication, decreasing hardware size, the invention of the colour LCD display, the track-ball and the touch-pad, and, the rapidly decreasing size and weight of mobile phones (frequently used for dialling-up). The timely combination of these achievements has enabled the widespread of PDAs – small lightweight transportable (portable) computers designed for specific mobile applications running specialized software. Also, the Internet phenomenon has become ubiquitous as it enables format-free data storage, retrieval, search and dynamic exchange without restrictive data modelling and with the freedom to query its content by looser means than relational algebra could offer.

That is, as a result of the combination of the availability of the Internet and the affordability of wireless communications, new type of applications have emerged requiring a new type of database support. Traditional database applications are still commonplace at enterprises and organizations where there is a need for relational data modelling, SQL-type of processing, traditional strong consistency and two-phase transactions, however, there is also a need for support of the new type of applications which find traditional relational and object databases too restrictive.

In this paper, we focus on the problem of retrieving dynamically updated location-aware information, rather than "classical" mobile problems dealing with the fluctuation in quality of a wireless communication network, or change in the degree of connectivity. This paper describes an information exchange model, MAGNET, which supports dynamically updated information among mobile users who frequently change location. MAGNET is investigating a different approach to information storage and queries, and is not

[1] By 'mobile users' we mean weakly-connected (dialing -in) users frequently changing their location (e.g. ,taxi-drivers, tourists) typically using portable computers or PDAs.
[2] A 'mobile application' we define as a distributed application run by mobile users (e.g., users with portables working while in transit, tourists while sightseeing, taxi-drivers etc.) dealing with location-dependent information (e.g., a local resource, local 'data', a location-based request) in a changing environment.

based on a traditional database framework. It is based on a shared information pool permitting operations for data insertion and their user-customized location-aware withdrawal – a query. In addition, it supports the monitoring of information placed in the pool, and user-defined adaptations to changes in the environment.

In the next section, we discuss our motivations, and define the requirements for a dynamic information-sharing infrastructure. Section 3 presents an overview of the MAGNET model. Section 4 describes the support for information monitoring in greater depth and section 5 demonstrates the use of MAGNET using an example of a taxi navigation system. Section 6 summarizes work relevant to MAGNET; section 7 discusses the project's current status and directions for future research. Finally, section 8 contains concluding remarks.

2 Motivations

In this section, we start with summarizing current problems of traditional database systems, then give a brief outline of the mobile environment typical for the applications requesting up-to-date location-dependent data. Then, we discuss characteristics of this class of applications, and elaborate on the requirements for database support.

2.1 Shortcomings of traditional DBMS

DBMS have matured to become large, expensive and lumbering pieces of systems software.. Consequently, as closed systems, a DBMS not only has sole ownership of the data, but data access is restricted to the DBMS through either query-languages or programming interfaces. Below we list a set of DBMS shortcomings, which were mostly highlighted by participants of a recent ICDE conference [1]:

1. multi-media and its content searching cannot be carried out by DBMS
2. information retrieval techniques are not incorporated into current DBMS systems
3. once a query is issued the user cannot make changes during processing
4. self administration and self tuning are currently unfeasible
5. data structures for mining need to be evaluated
6. improved data structures for model migration are needed
7. deeper internet integration e.g. compile once run everywhere
8. extensibility and reconfigurability could then provide e.g. 24 x 7 runtime support.

This list did spur on some research into DBMS flexibility, however to enhance a DBMS, historically new features were simply added to the kernel. This combination of *shoehorning* and *wrapping-up* of add-on services reduces not only performance [2, 3] but impairs core DBMS flexibility.

2.2 Characteristics of Location-aware Applications

Typical location-aware applications include: mobile portable users requiring local resources in different offices, tourists running guide-like information software on PDAs [4], or taxi-drivers using PDAs to navigate to the next destination.

The support for mobile computing can be investigated from various angles and at different levels, regarding the particular class of applications that are targeted. For mobile applications requiring dynamically updated location and time-dependent information, the crucial problem is the absence of service support for information sharing and run-time update. These applications, cannot be satisfied by traditional database engines, as currently these offer a too restrictive format, that is, relational algebra-based queries are too rigid, typically not providing any means for location and time based awareness and adaptation.

Owing to recent significant improvements in wireless communication, weakly connected applications no longer suffer from unreliability of the communication infrastructure (in terms of higher error-rate, frequent disconnections, or limited coverage). In addition, for some mobile users it is essential to be provided with *local information* (`local' in the term of the user's current location, e.g., `the nearest library', `the closest taxi'). The high volatility of the local information encountered by mobile users' moving location necessitates specialised database support which is able to be tailored to their new requirements and needs as

they change. The primary role of the database support should be enabling type-free information to be stored and exchanged, and different type of queries to be supported which allow wider user-customized location and time aware searching.

2.3 Tuplespace-based Database

In order to design the information exchange framework, we need to define the high-level requirements of mobile applications. The primary role of the database in this type of application is to enable the sharing of dynamic information (both local and location-independent). We term such an information-sharing infrastructure an *information pool*. That is, in order to avoid the confusion with the term database which is commonly used for a relational or object databases. The information pool, described above, holds data items termed *tuples* (i.e., structured records) which express information to be requested by mobile users. To achieve full generality, the information pool should not constrain the format or semantics of tuples it contains. This permits virtually unrestricted *extensibility* of existing services, data formats, in order to adapt application behaviour to changes in environment, or dynamically extend system functionality. In addition, dynamic information update (based on manual altering of information or automated monitoring) is required in order to adaptation to frequent changes in the mobile environment.

To query the pool, we use a mechanism we term tuple *matching*. Again, as the matching process semantically differs from traditional database queries, we would stick to the term *matching* to avoid confusion. In order to achieve generality, user-customisation of the requests for data stored in the pool is required. An addition, to distinguish our framework from other systems, we use the term binding refers to the result of the tuple match (1:1 relation, by default). In other words, when the match function has found a tuple satisfying the request, a *binding* between the two components which inserted the matching tuples is established.
To satisfy the crucial requirement of mobile applications – dynamic information exchange and update, a mechanism for automated monitoring is required in order to announce updates of monitored information provided by components themselves. Finally, updated information is fully utilized if the system can dynamically adapt to a new environment. This process is called *rebinding*.

3 Overview of the MAGNET Architecture

MAGNET is a high-level framework enabling applications in mobile frequently changing environments to store, update and query location-depended information. The full description of the architecture and its usage in dynamic resource management and other application areas could be found in [5]. Ideally MAGNET would be placed within a component-based systems architecture. This architecture would also reconfigure on demand and this too can be controlled by MAGNET (i.e., components can be activated and bound to other components at runtime). However in this paper we focus on information mapping mainly.
The key component of the framework is a *Trader* that collects information on services, records and all application data and dynamically matches requests against demands, in other words, performs user-customized search. One of the key features of the tuplespace is not to constrain the format or the semantics of stored information to allow type-free dynamically defined data to be stored and searched by a user-customized matching process. This provides *extensibility* in terms of enabling new records, service requests and actual services to be dynamically generated but also in terms of customizing the matching process itself. Further, to support runtime adaptation and system reconfiguration *dynamic rebinding* is required. That is, the old binding is dropped and a new binding is established in order to better meet application requirements. This may be as a result of client, server or a third party initiation. For example, a mobile client currently using its local disk may wish to join a new, more stable environment in an office to upload data. Therefore it will unbind from its current disk and rebind to the office disk. Information on client demands and service capabilities is maintained either manually (i.e., carried out by the components themselves) or automatically (by a monitoring process). The stateless nature of tuples saves the pool from having to provide a state-maintaining scheme, for example, check-pointing or recovery procedures. In addition, it improves the generality and reliability of the system. If state is required, it can be incorporated as a parameter of tuples. Finally, decoupling the server from the client (servers produce tuples of interest to any client) permits communication to proceed anonymously.

3.1 The Trader

The Trader is the key component in the MAGNET architecture. The Trader accesses a shared data repository available to all applications and objects represented by components. We call this data structure an *information pool*, its structure is similar to the tuplespace[3]. The Trader consists of three distinctive elements:

1. The information pool (a tuplespace-like data structure),
2. The Trader operations on tuples for their manipulation, and
3. The tuple matching function (an operation providing the actual querying or communication).

Figure 1 illustrates the structure of the Trader, and its three components. Darwin, an architecture-description language [6], provides a convenient formalism for defining bindings in distributed systems.

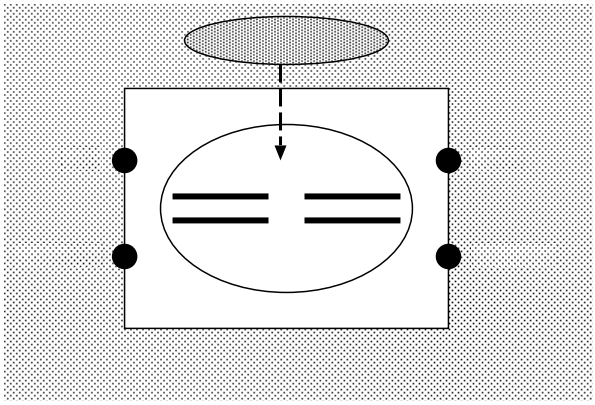

Figure 1: The Trader Structure

3.2 The Information Pool and the Matching Function

The information pool is a distributed data structure accessible by all components using MAGNET. Tuples can be inserted in, or withdrawn from, the tuplespace by a set of clearly defined operations. Tuples describing data of mobile components often contain additional information, such as interface references for accessing the component. These are all expressed as tuple elements. Therefore, the tuple distinguishes between the number of all tuple elements *n* and the number of matching elements *m*. This extension, which we have incorporated into traditional tuple matching, enables the restriction of the matching process to matching the first *m* elements.
We define a tuple as follows:

A **tuple** T is an ordered set of (n+2) elements $T=(n, m, p_1, p_2, \ldots p_n)$, $n>m$ where *n* represents the number of tuple elements and *m* is the number of "matchable" tuple elements p_i are the values of tuple elements i.e., the actual parameters.

For example, to describe a component book Romeo and Juliet by William Shakespeare we may use the following server tuple:

A = (6, 5, 12345, William, Shakespeare, Romeo and Juliet, Penguin, ISBN 654321)

That is 6 tuple elements, 5 of which can be matched*: 12345 (Author ID), William, Shakespeare, Romeo and Juliet, Penguin (publisher). ISBN* is a reference to the book (service) described by this tuple. Naming for interface references is derived from the naming scheme used in the computing or application environment, e.g., ISBN, library identification.
An equivalent client tuple looking up *Romeo and Juliet* would be:

[3]The information pool is actually a tuplespace. However, the term "tuplespace" is often associated with the Linda distributed programming language [17]; therefore, we decided to call our data structure `information pool' to avoid confusion.

*B = (6, 5, *, William, Shakespeare, Romeo and Juliet, *, reader ID)*

requesting this book published by any publisher (* sign) and ignoring the *Author ID*. This tuple definition incorporates advanced operators, such as *. These are defined in details in [5].

3.3 The Matching Function

By *matching*, querying the data structure, as was explained above, we mean an equality of tuple elements, or a user-defined "match" enabling quality of service to be taken into account. (However, this is beyond the scope of this paper, further details can be found in [5]).

A client tuple $T_1 = (n_1, m_1, p_1, p_2, \ldots p_n)$, $n_1 > m_1$ and a server tuple $T_2 = (n_2, m_2, q_1, q_2, \ldots q_n)$, $n_2 > m_2$ **match** iff $m_1 = m_2$ and $(p_i = q_i)$ for all $i \in \{1, m_1\}$.

As incorporating non-matching values into tuples is optional, and may differ between a client and a server-tuple, the equality of tuple size ($n_1 = n_2$) is not a required matching condition. Similar to SQL forms, in MAGNET the condition (WHERE) would be given as a value in particular column and the sign * would indicate that values of this column are irrelevant. As the information pool is shared by all components in the system, potentially of different applications, the notion of table (FROM statement) would have to be expressed as a column in a tuple. Finally, SELECT statement and the condition to return all, not just one matching tuple (as is the default) would have to be implemented as user-customized matching function. For example,

SELECT BookName
FROM Books
WHERE AuthorSurname = Shakespeare AND AuthorName = William

(following the example above) would be defined as:

*T = (6,5, *, William, Shakespeare, *, *, ReaderID)*

And the customized matching function would return all *BookName* of relevant tuples to *ReaderID* component.
As SQL query is not the goal of the architecture, it is obvious that this implementation is far less efficient, however, the example demonstrates that relational algebra could be built into the framework, if necessary. Nevertheless, the key focus of the architecture is to provide flexible dynamic matching of format-free records for mobile applications. For example, here, the request (client tuple) could be "waiting" in the pool for a server record to arrive, which is impossible in traditional databases. Above all, the user-customized matching function enabling an extra flexibility and framework extensibility is a powerful mechanism needed in dynamic frequently changing environments.

3.4 The Trader Operation

A set of predefined operations exist to manipulate tuple data, e.g., insert and delete. MAGNET's Trader includes the operations: *Bind*, Advert, *WithdrawC*, and *WithdrawS*. These are described below in more detail.

Operation **Bind (T)**, T is a client-tuple. The Trader searches the information pool for a complementary matching tuple. If such a tuple is found, T is returned to the server component (which inserted the matching tuple) without being withdrawn from the pool. If no such tuple exists, the operation results in inserting tuple T into the information pool until a match becomes available and the request is fulfilled.
Operation **Advert (T)**, T is a server-tuple, which is inserted into the information pool. The trader also searches the pool for all complementary matching tuples. If such tuples are found, they are removed from the pool, and returned to the calling server component.
Operation **WithdrawC (T)**, where T is a client-tuple, results in removing tuple T from the information pool; while operation **WithdrawS (T)**, where T is a server-tuple, results in removing tuple T from the information pool.

3.5 Components for the MAGNET Architecture

Figure 2 illustrates the structure of the MAGNET architecture. The system consists of four classes of component: *the Trader, Client, Server* and *Tree* (components performing the matching process). There is only a single instance of the Trader component per physical computing node, in contrast to multiple instances of Client, Server and Tree. In addition there are two types of subcomponent performing dedicated functions: these are a pair of Binders (the *Client-Binder* and the *Server-Binder*) present in all Clients and Servers; and the *GlueFactory* included in all Trees. The GlueFactory hands over a client tuple to the Server to initialise the establishment of the binding carried out by the Binders. Therefore, Binders in cooperation with the GlueFactory establish the resultant client-server binding. In order to provide scalability of the framework, it could be distributed into federations. However, in this paper, we focus on a single federation as a discussion on scalability is beyond the scope of this work. More details could be found in [5, 7, 8].

4 Support for Information Monitoring

In order to enable adaptation to changes in system characteristics, service definitions, which are placed in the Trader, must be kept up-to-date. Therefore, MAGNET must *monitor* resource characteristics. For this reason, the framework presented in this paper is equipped with two additional components providing monitoring: the *Monitor* (monitoring server provisions), and the *Updater* (monitoring changing client requirements). In this section, we will describe the semantics of these two components and the actual monitoring process.

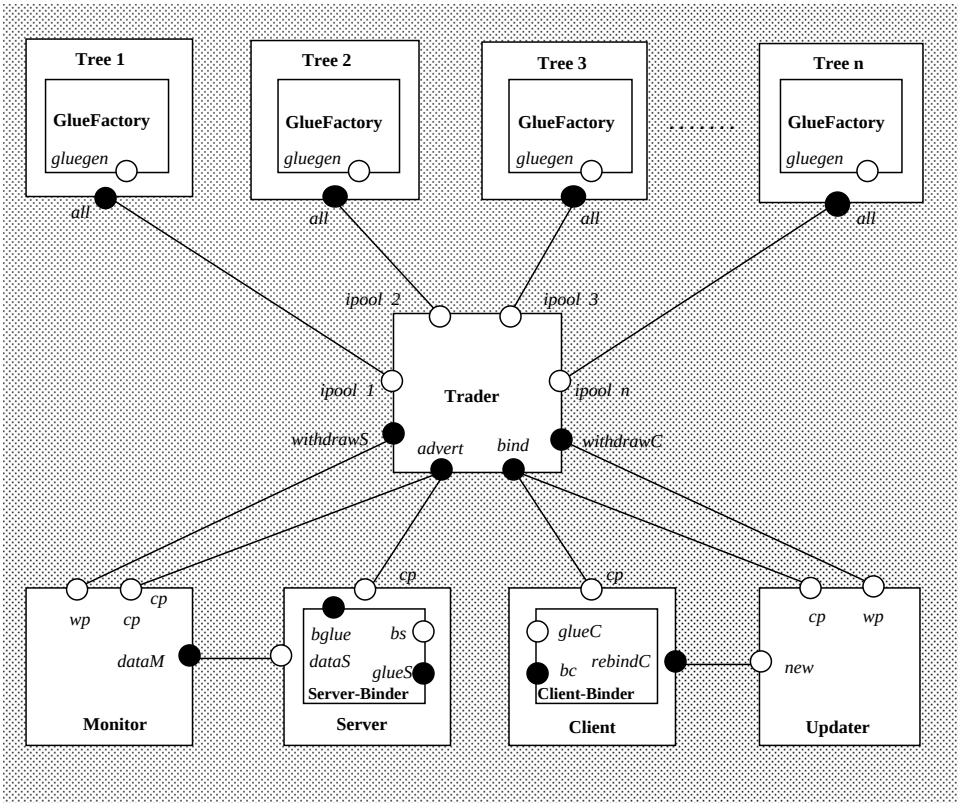

Figure 2: The Architecture with the Monitor and the Updater

4.1 Components for Monitoring

As the MAGNET framework distinguishes between the roles of the client and server, it is necessary to approach their monitoring differently. Therefore, MAGNET has two monitoring components providing this functionality – the *Monitor* and the *Updater* – both these application-level components are attached to server or client respectively. They are created together with the components they serve, and are instructed

by them to provide component-tailored functionality. Here we discuss their interface to MAGNET and expected functionality. The components are illustrated in Figure 2.

4.1.1 The Monitor

The task of the Monitor component is to observe changing characteristics of the server it is attached to, and keep the server tuple up-to-date. Tight cooperation with the server enables the Monitor to be informed about current service characteristics, so that it can periodically update relevant tuples in the pool (by removing them and replacing with updated ones). The granularity of this operation depends on the server strategy, in particular on the actual feature being updated, and on the overall character of an application (for example, real-time applications rely on finer-grained updates). However, in accordance with our assumptions, we expect the monitoring to be performed with frequency of minutes, rather than seconds and milliseconds.

4.1.2 The Updater

As there are not many clients require rebinding after having found a requested service, the monitoring of client requirements is less crucial. Also, client-tuples do not reside in the pool (once a match was found), and therefore there is no need to keep them up-to-date. However, clients in systems with frequently changing characteristics may rely on a guaranteed level of service (e.g., network throughput). For those, adaptation to change in conditions are unavoidable (e.g., switching to lower-quality audio and video, etc.) For these reasons, the framework must also provide equivalent support for monitoring clients.
The Updater is a dedicated component instructed by the client it is attached to. It searches the pool for a tuple meeting the client's current requirements more precisely, or looks for a different tuple if the client's requirements have changed (e.g., mobile users on the move need to update a requirement for the nearest server, etc.) The monitoring of the information pool is not the only function of the Updater. As changes might result in rebinding the client to a new server, the primary functionality of the Updater is to assist in third party rebinding.

4.2 Monitoring

In this section we describe the monitoring process, as provided by the dedicated components: the server-attached Monitor, and the client-attached Updater. Server-Monitor and Client-Updater interactions are established statically in advance by a system administrator, not using MAGNET.

4.2.1 Monitoring Server Provisions

The Monitor component is attached to the server by a binding established between service interfaces *dataS* and *dataM*. The server keeps the Monitor informed about relevant changes. Then, according to the granularity of update (how often it is performed), and the `out-of-dateness' accepted (how much can a tuple in the pool differ from current characteristics), the Monitor decides when to perform the operations *WithdrawS* and *Advert*. That is, the actual update in the pool (through service interfaces *cp* and *wp*). From the Trader's point of view, monitoring is performed transparently, indistinguishable from a sequence of operations *WithdrawS* and *Advert* performed by the server itself.

4.2.2 Monitoring Client Requirements

The Updater component is instructed by a client about service requirements it should search for. These two components communicate through a statically established binding between service interfaces *new* and *rebindC*. In this case, the initiative is on the Updater component, in contrast to the Monitor that acts only when invoked by the server. The Updater calls the operation *Bind* on a tuple with higher requirements (through service interface *cp*), or performs *WithdrawC* and *Bind* operations when the requirements of the client have changed. The bind-tuple, inserted by the Updater, waits in the pool until it finds a match. According to the Updater protocol and the `stage' of client interaction, the Updater decides if rebinding is beneficial (rebinding of a client close to finishing might not be beneficial, taking the overhead of the rebinding process into account). Therefore, the new server tuple can be ignored, or client rebinding can be performed. Details of the rebinding issues are beyond the scope of this paper.

4.3 Efficiency: Discussion

Data monitoring efficiency is an important issue. For applications requiring only course-grained monitoring strategies (with frequency of minutes), tuple updates performed by a withdrawal and reinsert (as discussed

in this section) are sufficient. However, for applications requiring finer-grained updates of their data in the pool (with frequency of seconds and milliseconds), the complexity of the Trader operations must be added to the complexity of the update operation. In order to improve efficiency, specific trusted Monitors and Updaters might be authorized to have direct access to the Tree holding their tuples. However, this solution fundamentally violates protection of the information pool (encapsulating Trees behind the public Trader's operations). For the reason of protection of other data in the pool, and protection of Trees that might be misused by untrustworthy Monitors, this approach is not a part of the framework design.

5 Example: The Taxi Navigation System

In this section, we illustrate the use of MAGNET on a typical mobile application – a taxi navigation system. This simplistic example illustrates MAGNET's functionality in terms of the applications' dependency on dynamically updated location-dependent information, requiring monitoring and support for adaptations to the changed situations.

5.1 The Application Description

The taxi navigation system consists of a number of taxis that are characterized by their location [X,Y], and passengers, characterized by their location – place where they are waiting for a taxi. Passengers can hail a taxi on the street, call for it by phone, or wait for it at a city taxi-rank. After a taxi drops off a passenger at the destination, it returns to the nearest taxi-rank, if not directed to another pick-up location. If there are no waiting passengers, taxis remain at the taxi-ranks. We can imagine that information about the length of the queue and passengers waiting at the taxi-rank is provided by a camera placed above the taxi-rank recording the queue and updating the information pool accordingly. All taxis are equipped with small PDAs and GPSs and are directed to their destination by a special component called the *Navigator* which transmits directions to the destination. The *Navigator* to select the best route at runtime, according to the city plan, and in response to dynamic changes, such as traffic jams, road-works, road closures due to accidents, etc. Figure 3 illustrates the situation.

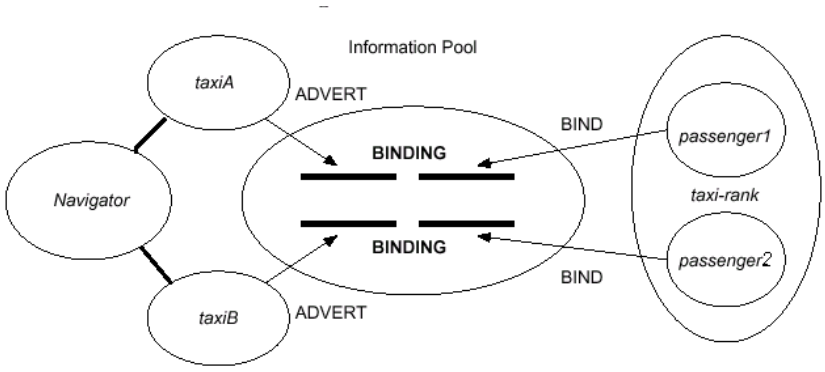

Figure 3: Components in the Dynamic Taxi Navigation System

5.2 MAGNET Support for Dynamic Taxi Allocation

We will describe MAGNET functionality on a simple scenario. MAGNET is used to find the best matches between available taxis (expressed by placing their offer into the pool) and waiting passengers (also expressed by placing their request for a taxi into the pool). Also available taxis have tuples in the pool, one a customer is picked up, the tuple is withdrawn as the taxi is no longer available.

1. **Taxi A and Taxi B available**

We start with a situation when there are two taxis (taxiA and taxiB) in the example, both currently available, being navigated by the Navigator to the taxi-rank. In MAGNET, the situation is described by two server tuples representing the taxis inserted into the pool by operation *Advert*:

TAXIA = (3,2,X1,Y1,TA)
TAXIB = (3,2,X2,Y2,TB)

Where the [X1, Y1] and [X2, Y2] are the coordinates of current location of the taxis. The Monitor components attached to both taxis ensure the tuples are updated in requested intervals, e.g., 5 minutes. TA and TB are references to the taxis, e.g., a direct contact to their Navigator components, or phone numbers of their drivers etc.

2. Passenger 1 arrives

Then, Passenger 1 arrives to the taxi-rank and waits for a taxi. The camera at the taxi-rank records the client and inserts a client tuple using operation *Bind*:

P1 = (3,2,X3,Y3,P1)

Where [X3, Y3] is the location of Passenger 1 – the coordinates of the taxi-rank and P1 is the identification of the passenger. As soon as the operation *Bind(P1)* is called the matching function finds the best available match (the closest taxi) and allocates the passenger to that taxi (lets Taxi A is the closer one). Then, the P1 tuple is automatically withdrawn (as this is the definition of the operation), and tuple TAXIA is manually withdrawn by calling *WithdrawS* operation as a taxi usually cannot drive more than one passenger. The details of the matching function are discussed below.

3. Taxi 2 hailed on the Street by Passenger 2

TAXIB, still on its way to the taxi-rank, is hailed on the street by Passenger 2 – this 'binding' takes place without MAGNET to illustrate that in open systems components can establish binging also without the assistance of traders. Technically, this client did not insert any tuple into the pool, simply hailed a passing taxi. Consequently, the *WithdrawS (TAXIB)* operation is called by the TAXIB Monitor to reflect the change. Therefore, there are no tuples in the pool at the moment.

4. Passenger 3 calls for a taxi by phone

A passenger 3 calls for a taxi by phone, a client tuple P3 = (3,2,X4,Y4,P4) is inserted into the pool by operation *Bind* (we may imagine the automated phone operator calls the function). The tuple remains waiting in the pool as there is no taxi available. The, TAXI A drops off Passenger 1, the relevant Monitor reinserts the server tuple into the pool by operation *Advert (TAXIA*). Then, a match is achieved between TAXIA and Passenger 3 resulting in directing TAXIA to Passenger 3 location.

5.3 User-customized Matching Function

The key feature enabling this dynamic taxi-passenger matching is the customized matching function. In this example, we have assumed their [X, Y] coordinates indicate locations of taxis and passengers. As for taxis, these are recorded by GPSs attached to taxi Monitors, however, for clients they need to be calculated from street names. The user-customized matching function selects the closest client for each taxi (or vice versa). For example, there is a taxi tuple TAXI = (3,2,X1,Y1,T) and N waiting clients P_n = (3, 2, X_n2 ,Y_n2, P_n) in the pool. Then, the matching function finds the minimum distance (the closest client to the taxi):

TAXI matches P_i iff Min i $_{\forall i \in N}$ ($|X1 - X_i2| + |Y1 - Y_i2|$)

This calculation assumes grid street topology. We have adopted this approach to illustrate the notion of user defined matching, however, in many US cities this distance definition would be perfectly appropriate. However, the function could be further extended to allow more complex distance definitions, based on real street maps, and include dynamic changes, such as traffic jams and road closures. However, further investigation of these issues is beyond the scope of this paper.

5.4 Dynamic Adaptation

A passenger waiting for a taxi at a taxi-rank, is a classical situation, not requiring adaptation, assuming there is only one taxi-rank in our example spares the system solving the problem of redirecting taxis from one taxi-rank to another in response to varying lengths of queues. Typical adaptation-requiring situations include a taxi being hailed in a street when this was to pick up a customer at the taxi-rank, or a passenger phoning for a taxi (according to the taxi system priority policy, closest taxi could be allocated to calling

client despite the fact the taxi was going to pick up a different customer living further, etc.). First-party and third-party rebinding is a key feature of the MAGNET architecture and could be illustrated on the dynamic scenarios discussed in this section, however, full explanation of the support for rebinding is beyond the scope of this paper.

5.5 Discussion

For additional flexibility, the tuplespace structure as defined in MAGNET can allow the application to model time-constrained operations (e.g., a passenger urgently needs a taxi to get to the airport; but, if the taxi does not arrive within ten minutes, the plane will be missed, therefore, the passenger is not interested after ten minutes). Clients can choose how long they are willing to wait until their request is accepted. The time scale extends from zero (if the requested tuple is not available at that moment, an error status is returned to the client), through arbitrary time-out intervals (the tuple is waiting in the pool until the requested tuple is inserted or until the timeout expires), to unlimited waiting (the tuple persists in the pool forever if the required complementary tuple has never been inserted). In order to incorporate a "timeout" feature, the client can withdraw the tuple when he is no longer interested, or this could be provided automatically by an *Updater* component where the client sets a predefined threshold. This functionality can also be incorporated into the user-defined matching functions, if appropriate (e.g., in the previous example of a passenger travelling to the airport; the matching function can incorporate more flexible adjustment of the time interval according to the current traffic situation). Like clients, servers can also time-constrain their tuples in the pool; this variation is supported in a similar way, however, in our example it would only be meaningful to identify the time when a taxi drives finishes work.
Finally, the system could record the destination of clients in terms of extra tuple elements and their willingness to share a taxi with other clients. Then, the matching function could optimise taxi allocation so as clients travelling to destinations near each other could share a taxi which would result in more efficient transport. The optimisation matching function can be stored in the Trader with the tuples.

6 Related Work

There has been some work on adaptive query processing. Examples of this work are pipelined hash join [9], hash ripple join [10] and the Xjoin [11]. Most of this work is with relational data and concerns aggregation queries as examples [12, 10, 13], however some have looked at XML [14]. Nevertheless this work has been very focused and they do not provide the level of customisation supported by MAGNET.
Contemporary research in mobile computing has explored problems with mobility and the unreliability of wireless communication networks [15]. Fluctuations in quality of service (QoS) and changing degrees of connectivity have also been studied [16]. However, systems which provide the functionality required by location-aware mobile applications to allow dynamic information update and adaptability have not been widely investigated.
As for trading architectures, Linda was the first system to support a generative communication model [17], providing several important features, but its fixed tuple format and semantics do not provide the flexibility required by mobile applications. A question-based system Osprey [18], which was motivated by Linda, implemented application-server coupling using tuple-based interaction. It added a level of flexibility by utilizing a result-based tuple naming scheme and replicating tuples over many nodes, but it did not address issues concerning user-defined matching. Also, JavaSpaces provide a Tuplespace-like distributed environment manipulating objects rather than data tuples. This enables global scalability and forms a base for the Jini technology [19].

Blair *et al* [20] investigated the tuplespace approach to QoS support in a mobile environment. It extends the traditional tuplespace with QoS management providing support for monitoring and adaptation for applications using heterogeneous networking environments. Its emphasis is on QoS monitoring and adaptation to changes in network connectivity in order to ensure the same level of service behaviour, unlike MAGNET that supports adaptation of system behaviour according to changes in the environment.
The problem of dynamic adaptation to a change in environment has been successfully addressed by the Personal Computer Memory Card International Association. PCMCIA Ethernet cards can be added and removed from the system without powering-off or rebooting the computer. The Linux *kernel daemon* is another successful attempt, enabling operating system kernel adaptation by adding or removing modules transparently on demand [21]. Incidentally, both these dynamic adaptation approaches can be implemented using MAGNET they are examples of dynamic resource reconfiguration as discussed in [5,7,8].

7 Current Status and Further Work

In this section, we summarize our assumptions, outline our implementation experience and discuss further work.

7.1 Assumptions

Here we summarize the assumptions we used when designing the MAGNET system, and discuss their implications and possible solutions. We assume all system components maintain their own *consistency*. That is, we assume that rebinding can be performed only when the system is in a safe state and that when a component has finished its operation it must leave MAGNET in a consistent state. Consequently, a more powerful framework consisting of transaction processing would be required. A related situation is where old tuples remain in the information pool. A periodic garbage collection routine can purge tuples that are marked as out-of-date by as defined by the originating component. Similarly, components are responsible for the validity of their tuples. In order to prevent components from leaving tuples in the pool when they finish the operation, a special subcomponent (present in every component) could automatically withdraw all inserted tuples. However, this solution requires co-operation with the component, in terms of initialisation of the operations, so it is not fully automated.

Further, user defined functions are assumed to be *secure* in that they return control back to the Trader. To overcome this we would have to extend the trader's functionality to finish any matching function by force after a timeout period. Also, we assume that unambiguous *naming schemes* are used. If the computing environment does not provide naming which meets these requirements, there must be an additional Trader naming scheme defined (or a more intelligent fuzzy mapping mechanism implemented). However, the former can be derived from common naming schemes, such as IP addresses.

As for *performance*, the estimated numbers of components in are in the region of tens and they have the potential to generate tens to hundreds tuples placed in the Trader. Likewise, the number of concurrent components accessing the Trader at one time are estimated to be in the region of tens. A higher number of components can result in the Trader becoming a bottleneck. A possible solution would be to implement the information pool in distributed shared memory.

Regarding *change frequency*, the framework is designed for components that will change their features with a frequency of minutes and hours, rather than seconds and milliseconds. Therefore the proposed support for monitoring and rebinding as a result of a change is adequate. The support for applications requiring finer grained updates (with a frequency of seconds and milliseconds) would not be viable. This can be improved by enabling direct access to the Tree components for trusted Monitors and Updaters.

7.2 Implementation

MAGNET has been implemented in Regis [22], an environment for constructing distributed systems. The tuple is implemented in C++ as a high-level base-class (Tuple) comprising the tuple size, the tuple matching size, and encapsulating the tuple-elements. All standard and user-defined tuple-element classes are inherited from a base tuple-element class TElm. Trees contain tree data structures supporting the search (and matching) for non-parameterised requests. The complexity of the Trader operations was calculated and was found to be linear to the number of tuple matching elements. The Trader is responsible for the efficient distribution of tree data structures over Tree components.

The matching function is implemented as an overloaded member function of tuple-element classes inherited from the base class TElm. A tuple-element type matches only the same type, and the “equality” of values can be re-defined according to the type.

As the focus of the architecture is to provide dynamic features, such as runtime adaptability, user-customisation and flexibility, the implementation results cannot be described in terms of performance. However, critical analysis of various features of the framework can be found in [5].

In addition, the MAGNET architecture also supports advanced QoS support. The extensibility of the framework allows applications to define and negotiate services using QoS characteristics. However,

support for QoS is beyond the scope of this paper. Further details of our QoS model, its design and implementation can be found in [5].

7.3 Further Work

We are currently looking at using the MAGNET infrastructure for the National electronic Library for Health which provides a single gateway to evidence-based medical information on the Internet. It supports QoS-based search for data and in the future, we will be looking at supporting adaptability for different degrees of connectivity, from wireless to fully connected.

Also, another our project Go!, a component-based Operating System which has shown that fine-grained componentisation does not only provide lightweightness and extensibility but also improvements in performance [23]. Effectively the aim of our current research is to prove that to provide an infrastructure that would support mobile database computing, one needs component-based technologies down all the layers to the hardware. We believe that Go! is a good starting point for this research as it has already proven that it can improve performance and be lightweight, but combined with MAGNET we should be able to show its true power. To do this we are currently expanding the operating system, and extending our work on models to describe hardware abstractions, components and their interaction and resource management. This is in terms of resource to request matching (R2R) and extends our work on resource mapping in MAGNET.

8 Conclusion

This paper has targeted a fundamental problem of mobile users requiring dynamically updated location-aware information. We have argued that the problem has become crucial, owing to a combination of recent improvements in wireless communication, and advances in hardware technology. As a result of these fundamental changes, there is a new class of applications requiring type-free data storage, frequent updates and modifications and user-defined flexible way to query these data. These applications need both flexibility and generality, and often no longer require the traditional database features, relational data modelling, transactions and security constrains.

As traditional database systems do not provide support for these types of mobile applications, we have investigated a tuplespace-based framework, MAGNET, allowing the searching and trading of information and data records in frequently changing mobile environments. This extends the notion of the tuplespace paradigm to provide a universal solution, which interestingly is not tied to mobile environments only. We illustrated how MAGNET meets the specified requirements by a taxi navigation system case study.

References

1. ICDE, 13th IEEE International Conference on Data Engineering, Birmingham, UK, April 7-11, 1997

2. M. Stonebraker, P. Brown. ‘Objects in Middleware: How bad can it be?’, Informix white paper, www.informix.com/informix/whitepapers/howbad/howbad.htm

3. J. Thomas, D. Batory. P2: An Extensible Lightweight DBMS, Technical Report, The University of Texas at Austin, Department of Computer Sciences, Number UTEXAS.CS//CS-TR-95-04, February 1995.

4. Distributed Multimedia Research Group. ABTA: The Active Badge Tourist Application. Computing Department, Lancaster University, Lancaster, UK. Electronic document available at http://www.comp.lancs.ac.uk/computing/research/mpg/most/abta_project.html

5. P. Kostkova. MAGNET: Dynamic Resource Management Architecture. PhD Thesis. Dept. of Computing, The City University, London, March 1999.

6. J. Magee, N. Dulay, S. Eisenbach, J. Kramer. Specifying Distributed Software Architectures. Fifth European Software Engineering Conference, Barcelona, September 1995.

7. P. Kostkova, McCann J.A. MAGNET: An Architecture for Dynamic Resource Allocation. Proc. of International Workshop on Data Engineering for Wireless and Mobile Access, ACM, August 1999.

8. P. Kostkova, J.A. McCann. Inter-federation Communication in the MAGNET Architecture. Published in the Third Grace Hopper Celebration of Women in Computing, Mass. USA., September 2000.

9. Wilschut A. N., Apers P.M. G.: 'Dataflow Query Execution in a Parallel Main-Memory Environment. In Proc. First International Conference on Parallel and Distributed Information Systems (PDIS), pp 68-77, 1991

10. Haas P. and Hellerstein J. 'Ripple joins for online aggregation'. In Proc. of the ACM SIGMOD Conference, 287-298, 1999.

11. Urhan T., Franklin M.J., Amsaleg L., 'Cost-based query scrambling for initial delays'. In Proc. ACM SIGMOD International Conference on Management of Data, 1998.

12. Avnur R. Hellerstein J. M. 'Eddies: Continuously Adaptive Query Processing', Proc. ACM SIGMOD Int. Conf. Management of Data, Dallas, TX, May 2000.

13. Hellerstein J. M., Haas P.J., Wang H.J. 'Online Aggregation', Technical Paper, IBM (1997)

14. Ives Z G., Levy A Y., Weld D. S., Florescu D., Friedman M. 'Adaptive Query Processing for Internet Applications'. IEEE Data Engineering Bulletin vol 23 no 2, pp 19-26, 2000

15. G.H. Forman, J. Zahorjan. The Challenges of Mobile Computing. IEEE Computer, 27(4), pp. 38-47, April 1994.

16. N. Davies, G. S. Blair, K. Cheverst and A. Friday. Supporting Adaptive Services in a Heterogeneous Mobile Environment. The 1st Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA, December 1994.

17. D. Gelernter. Generative Communication in Linda. ACM Transactions on Programming Languages and Systems, 7(1), pp. 80-112, January 1985.

18. D. Bolton, D. Gilbert, K. Murray, P. Osmon, A. Whitcroft, T. Wilkinson, N. Williams. A Question based approach to Open systems: OSPREY Internal TR, SARC, City University, London. March 1993.

19. J. Waldo.Jini Architecture Overview
http://www/javasoft.com/products/jini/whitepapers/architectureovervoew.pdf, Sun Microsystems, 1998.

20. G. S. Blair, N. Davies, A. P. Wade. Quality of service support in mobile environment: an approach based on tuple spaces. The 5th IFIP International Workshop on Quality of Service, New York, USA, May 1997.

21. H. Storner. Linux kerneld mini-HOWTO. http://www.image.dk/~storner/kerneld-mini-HOWTO.html, version 1.7, July 19, 1997.

22. J. S. Crane. Dynamic Binding for Distributed Systems. PhD thesis, Dept. of Computing, Imperial College, London, UK, 1997.

23. G. Law, J. A. McCann. Decomposition of Pre-emptive Scheduling in the Go! Component-Based Operating System, ACM SIGOPS European Workshop, 2000.