

Location based mobile computing - A tuplespace perspective

Anders Fongen (Corresponding author)*, Christian Larsen

The Norwegian School of Information Technology, Oslo, Norway

Gheorghita Ghinea, Simon J E Taylor, Tacha Serif

Brunel University, Uxbridge, UK

April 11, 2006

Abstract

Location based or “context aware” computing is becoming increasingly recognized as a vital part of a mobile computing environment. As a consequence, the need for location-management middleware is widely recognized and actively researched.

Location-management is frequently offered to the application through a “location API” (e.g. JSR 179) where the mobile unit can find out its own location as coordinates or as “building, floor, room” values. It is then up

*E-mail address: anders@fongen.no, Tel:+4792018988

to the application to map the coordinates into a set of localized variables, e.g. direction to the nearest bookshop or the local timezone. It is the opinion of the authors that a localization API should be more transparent and more integrated: The localized values should be handed to the application directly, and the API for doing so should be the same as the general storage mechanisms.

Our proposed middleware for location and context management is built on top of *Mobispace*. *Mobispace* is a distributed tuplespace made for mobile units (J2me) where replication between local replicas takes place with a central server (over GPRS) or with other mobile units (using Bluetooth). Since a Bluetooth connection indicates physical proximity to another node, a set of stationary nodes may distribute locality information over Bluetooth connections, and this information may be retrieved through the ordinary tuplespace API.

Besides the integration with the general framework for communication and coordination the middleware offers straightforward answers to questions like: *Where is node X located? Which nodes are near me? What is the trace of node Y?*

Keywords: Distributed tuplespace, location based, context aware, J2me

1 Introduction

The term *Location based computing* refers to mobile programs that allow the current location to influence on its execution. A typical example is that the pro-

gram need localized information like distance and direction to the nearest hospital, the name of local currency and a map of the neighbourhood.

Middleware for location based computing is typically found behind location APIs like JSR-179¹, through which the client program can inquire about its own position. It is then the responsibility of the application to retrieve the necessary localized information. This operation may possibly involve transformation of co-ordinates to retrieval parameters which potentially is a complicated process.

A more straightforward approach to the retrieval of localized information is needed. In this paper, a location middleware is offered as an integral part of a distributed tuplespace system. Localized tuples (i.e. tuples containing localized information) are retrieved from the tuplespace as any other tuple, using special data types in the template.

The proposed implementation of the location service is based on a distributed tuplespace for J2me (Java 2 Micro Edition) called *Mobispace*[6], in which mobile nodes update each others local store over a Bluetooth connection. It is thus possible to configure “stable” nodes with a Bluetooth adapter working as a “beacon” so that any other node within radio range of the beacon will know the name of the “area” it is in, and on the basis of this information fetch localized tuples from the local store. The focus of this paper is to provide detailed information on the principles of this mechanism.

The rest of the paper is organized as follows: In Section 2 and 3 we present the underlying *Mobispace* system and discuss a few principles of tuplespace pro-

¹<http://www.jcp.org/en/jsr/detail?id=179>

gramming. In Section 4 the the principles of a distributed tuplespace are presented and discussed. Section 5 gives an overview of the Mobispaced-based location management system and constitutes the core of the paper. Sections 6-9 provide a detailed presentation of the underlying replication protocols and a proof on the associated ordering semantics. Section 10 concludes the work and suggests future research on this topic. Readers who are only interested in a brief overview of the location management mechanisms can skip to Section 5 and read the other sections as needed.

2 An overview of Mobispace

Mobispace is an implementation of the tuplespace model for coordination, communication and storage, also known as ‘Linda’ [8]. A large body of knowledge has been established on how to design distributed applications over the tuplespace abstraction (e.g. [2]).

For a tuplespace to be working in a mobile and distributed application, it should be memory-efficient and able to work in an occasionally-connected environment. It should also be reasonably network-efficient since a mobile unit (using GPRS and Bluetooth connection) have scarce communication resources.

The Mobispace system is designed for mobile applications. It utilizes the scarce set of resources present in a mobile unit and is designed for connection interruption of unknown length. For portability reasons, the Java 2 Micro Edition (J2me) platform has been chosen for the implementation for portability reasons.

The typical communication facilities for a J2me device is a GPRS/GSM service which offers HTTP connections through the Internet, and/or a Bluetooth device offering short-distance communication with other mobile units (or possibly a larger computer). MobiSpace uses a distributed and replicated tuplespace employing replication methods that exploits a combination of these communication facilities.

The attractiveness of the Mobispace is that it offers a familiar and flexible programming model with a high abstraction level to developers of mobile systems. The loosely coupled coordination and indirect interactions offered by the tuplespace model fits well with the dynamic environment of mobile systems.

MobiSpace supports:

- Primary-based replication based on a central server connected to secondary (J2me) nodes through a GPRS/GSM service (or any service that can offer an Internet connection)
- p2p-based replication between secondary nodes based on Bluetooth communication
- Secondary nodes express their tuple selection criteria during replication through a set of templates called an *interest profile*
- Open protocols (XML, HTTP, RFCOMM) for interoperability with non-J2me agents. Secondary nodes can run on any platform and in any language
- Unknown and dynamic number of secondary nodes

- Straightforward ordering and synchronization semantics

3 The principles of tuplespace programming

The programming model known as “tuplespace” was proposed by Gelernter in 1985 [8] as a combination of an associative shared storage mechanism and synchronized retrieval operations in a model called *Linda*. Today there are two major implementations of tuplespace in a Java environment: JavaSpaces from Sun Microsystems [12, 7] and IBM TSpaces [10].

The basic data structure used in the tuplespace is the *tuple*, which is an ordered set of *fields*. Tuples may be written to the tuplespace, after which they are available for retrieval by any client of the tuplespace. The original tuplespace model makes a clear distinction between consuming and non-consuming retrieval operations: A consuming retrieval operation is an atomic read-delete operation, so that it guarantees that only one client retrieves the tuple. A non-consuming retrieval operation returns a tuple without affecting its existence. Tuples are *immutable*, which means that they never change once they are added to a tuplespace. “Updating a tuple” is done by replacing it with a new tuple in tuplespace. A tuple does not need to have any unique fields in the sense of a primary key.

Retrieval of tuples is done through the use of a *template* parameter. The retrieval operation selects a set of tuples *matching* the template, and one or all of the matching tuples are returned to the caller. A template resembles a tuple by its ordered set of fields, but some of the fields may be “wildcards” i.e. they have no

defined value. A tuple matches a template if all these conditions are met:

- they have the same arity (number of fields),
- the fields of the template and the tuple have pair-wise the same value and type. Wildcard fields in the template matches any field value (and type) in the tuple.

Formally, a match operation where a template t_1 is applied to a tuplespace \mathbf{T} resulting in a set of tuples \mathbf{V} can be expressed as follows:

$$\mathbf{V} = \text{match}(t_1, \mathbf{T}) \quad (1)$$

The original Linda model [8] uses typeless wildcards, and the JavaSpaces implementation follows this principle. IBM's TSpaces, on the other hand, uses *typed wildcards*, in which the type of the wildcard is checked against the type of the tuple field in an object-oriented fashion.

The result of a retrieval operation is any tuple that matches the template, and neither the Linda model nor JavaSpaces offer any defined order of retrieved tuples. TSpaces, on the other hand, offers 'FIFO' ordering as a configuration option. In JavaSpaces, any ordering requirements is left to the application which must implement a sequence number scheme in the tuple design.

4 Distributed tuplespaces

Both JavaSpaces and TSpaces implement their services based on a central server. A central server facilitates consistency and transactional semantics while at the same time creating a scalability bottleneck and a single point of failure. Also, a central server most often requires permanent connectivity between the client and the server. Therefore, several distributed tuplespaces have been proposed: Patterson [15] has presented a fault-tolerant distributed design which requires high availability of network resources. The LIME system (Linda in a Mobile Environment) [16] offers a platform for mobile agents which bring a small tuple space with them as they migrate and make them accessible to other agents residing on the same host. The SwarmLinda system [5] offers a mechanism for distributed clustering of tuples in a p2p environment and claims to be highly scalable. No distributed tuple space implementation for the J2me environment has been reported.

In order to conserve the transactional semantics of a tuple space system the clients need (in practice) to be permanently connected to the server, so the state-oriented operations between the nodes can be effectively conducted. A consuming read, for instance, will require a lock on the same tuple in all replica in order to provide a guarantee that the tuple is taken by only one client, and such a stateful distributed operation requires high availability of network resources.

A distributed tuple space designed for an occasionally-connected environment, where it can be weeks and months between network connections requires a reformulation of the transactional semantics. A scheme that allows for relaxed co-

ordination between nodes is required. Ordering semantics combined with lazy replication appear to be useful concepts in such a scheme.

4.1 Ordering and consistency semantics

The correctness of a replicated storage system relies on the ordering of write operations being passed across the network. If two replica receive write operations in different order, they may end up in different (inconsistent) states.

A system where all replica receive the results of write operations in the same order is called *sequentially consistent*. A more relaxed requirement is that all nodes should receive *causally related* write operations in the same order, in which case the system is *causally consistent*. The corresponding ordering requirement is called *causal ordering*. Mobispace offers causal ordering semantics.

When applied to a tuplespace system, the consistency requirements need to be slightly reformulated, since clients do not necessarily retrieve the same tuples. Retrieval operations select tuples on the basis of a template parameter, so two clients will possibly retrieve different sequences of tuples. The reformulated requirement reads:

If one tuplespace client retrieves two causally related tuples matching the same template in the order (a,b), then no other client should retrieve them in the order (b,a).

Although considerable effort have gone into semantic definitions of tuplespace-based coordination models, e.g. [14], there have been no reports on the semantics

of tuple ordering.

The details of the Mobispace architecture and the replication protocols are given in Section 6 onwards. The text will now proceed with a presentation of how the Bluetooth technology can be used for location management purposes.

5 Bluetooth as a basis for location management

On top of the current Mobispace configuration, location management comes almost for free. A secondary node must for this purpose be equipped with a Bluetooth adapter. Two Bluetooth units can “discover” each other and inquire about the other node’s name and available services, and then connect for transport of data.

A stable (non-moving) secondary node can be configured to act as a *beacon*, and the area within radio range of its Bluetooth adapter is called a *zone*. Other nodes within radio range will pick up its “friendly name” as a designation of its location. The “zone designator” is used as a field to construct the template being used for retrieval of *localized tuples* i.e. tuples which are valid only in this zone.

Several research projects attempt to utilize Bluetooth hardware for purposes of location management [9, 3, 13, 1, 4]. Although not designed with instant device discovery in mind, Bluetooth is widely deployed in mobile units (mostly with other applications in mind) and can be used through well-established APIs.

Figure 1 shows an example on how a Mobispace network can be configured for location management purposes. Three secondary nodes are deployed as bea-

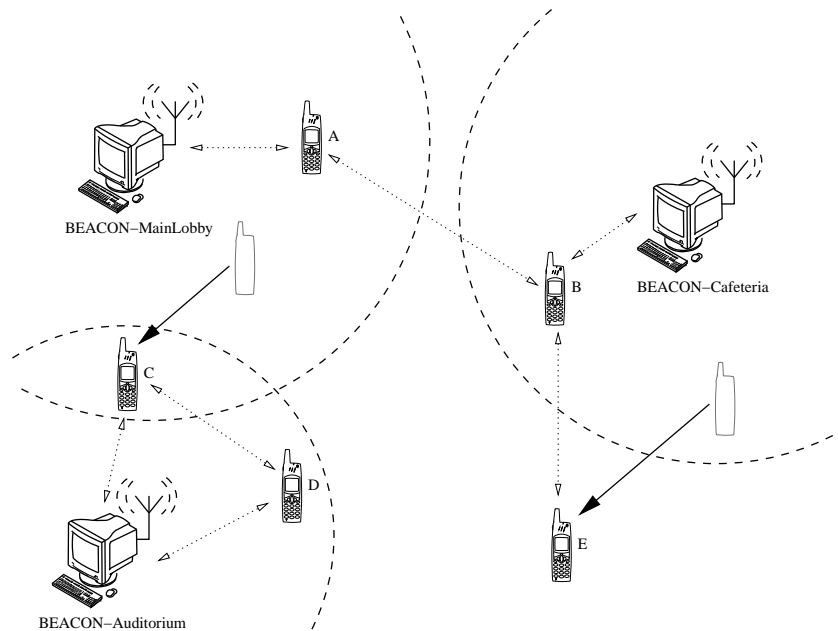


Figure 1: Configuration and position of localization hardware

cons on ordinary PCs representing the three zones “MainLobby”, “Cafeteria” and “Auditorium”. They are identified as beacons by other nodes by naming convention.

The mobile nodes that are within radio range of one beacon (nodes A, B and D) will have discovered the beacon and set up their tuplespace retrieval templates accordingly. The mobile nodes C and E have recently moved as shown with solid-line arrows on the figure. Node E is now outside the range of all beacons, but retain its association with the zone “Cafeteria” until it eventually moves within radio range of another beacon. Node C has moved within radio range of the beacon representing the zone “Auditorium”, but since it still hears the old beacon (“MainLobby”) it will still be associated with this zone.

The dashed-line arrows show a selection of secondary replication links. They are included to show that the secondary replication takes place fully independent of the associations of nodes to zones: B and A replicate while in different zones, and C replicates with a beacon which it is not associated with.

Although not shown on the figure, secondary nodes (mobile and beacons) are optionally conducting primary replication with the primary server; beacons are likely to use wired connections for this purpose, whilst mobile nodes would e.g. use GPRS.

The selection of localized tuples represents a process independent from the replication strategy. In other words, the localized tuples are not actively fetched from a server when a new zone designator is discovered. The localized tuples are replicated between the nodes in the same fashion as any other tuple. Which means that the *interest profile* (See Section 6.2) must be set accordingly for the mobile node to receive localized tuples.

5.1 The Design of a localized tuple

A new field data type has been introduced for the purpose of location management, the *Location*. Due to the type matching of templates and tuples, no localized tuple will be returned to a client unless the template has a field of this data type. A distinct data type for this purpose thus strengthens the separation between localized and ordinary tuples.

For the current state of this project, localized tuples are in the form of (key,value)

pairs. A localized tuple has the following design:

$$localizedTuple = \{Location(zone), String(key), String(value)\} \quad (2)$$

which means that it contains of three fields, the first one being of type *Location*, the two following of type *String*. The value of the first field indicates the zone designation that the tuple belongs to.

The retrieval of localized tuples which belong to a particular zone will use the zone designation and value key as fields in the template parameter:

$$localizedTemplate = \{Location(zone), String(key), String(null)\} \quad (3)$$

The management of localized tuples (creation and deletions) may be given to any node in the system, but the best solution would be to leave this task to the beacons itself or a central coordinator.

5.2 User-centric localized tuples

In addition to these “zone-centric” localized tuples there exist also localized tuples that do not describe properties of locations, but of *users*. User-centric localized tuples are used to describe the whereabouts of user/nodes² so that questions like: “In which zone is Christian?” or “Who is in the *Cafeteria* zone?” may be answered.

The design of a user-centric localized tuple involves the same structure as

²We assume that a node represents a user, and thus the location of nodes reveals the location of a person.

before but involves a “wildcard” zone designation which indicates its special role in locating and listing users of the location-aware application:

$$localizedUserTuple = \{Location(wildcard), String(user), String(zone)\} \quad (4)$$

The management of user-centric localized tuples is done automatically by the Mobispace middleware. As soon as a node comes within radio range of a beacon and establishes a link with it, the Mobispace software of the mobile node will remove the tuple containing its former location and replace it with an updated value (with the designation of the new zone). This information (both the tuple deletion and the new tuple) will eventually propagate to all nodes through replication sessions.

The retrieval of localized tuples which describe the location of a particular user will use a template like:

$$localizedUserTemplate = \{Location(wildcard), String(user), String(null)\} \quad (5)$$

The question “who is in my zone” may be answered by a retrieval operation based on this tuple: The retrieval of localized tuples which belongs to a particular zone will use the zone designation and value key as fields in the template parameter:

$$localizedZoneTemplate = \{Location(wildcard), String(null), String(zone)\} \quad (6)$$

Likewise, questions like “who is in zone Y” or “who is in all zones” is answered

by applying the appropriate template to a retrieval operation.

5.3 Zones larger than the radio range

A mobile node picks up the zone designation as it discovers a beacon, and keeps that designation as ‘its’ until another beacon is heard. Consequently, a node belongs to a zone from the moment it discovers one beacon until it discovers the next (as indicated on Figure 1). This condition of the system can be exploited in order to have zones which are larger than the radio range of a small Bluetooth beacon: A beacon may be placed e.g. in the entrance of a building in order to have one zone for the entire building, since every mobile node present in the building has had to pass the beacon in the entrance. Efficient physical placement of beacons should therefore not only consider the propagation of radio waves, but also the movement patterns of the users.

5.4 Scaleability and responsiveness

The described form of location management depends on the responsiveness of the underlying communication services. The example just mentioned with a beacon in the entrance of the building requires that a node quickly detects a that a beacon has come inside radio range and quickly establishes the identity of the new zone. Also for application where it is necessary to keep a trace of movements in the form of a sequence of zone designations, it is important that this process completes before the user moves out of radio range again. In other words, the size of the Bluetooth

“cell” should be large enough so that even a user in constant movement should be able to establish the zone identity before it moves on. It also becomes necessary to consider scalability issues: There is an upper limit on how many mobile nodes that can enter the building at the same time so that everyone discovers the beacon.

Bluetooth technology is not particularly designed for quick link establishment. Bruno and Delmastro [3] show how the discovery time (equivalent to “link setup”) forms a two-lobed probability distribution with peaks at approx. 0.5 sec and 3.0 sec. The two-lobed distribution is due to the random selection of frequency sequences in the bluetooth nodes. Their report also shows that in piconets with 7 nodes or less, half of the nodes will be found within 0.8 sec. After 3.3 sec all nodes are found by the inquiring master, even in configurations with as many as 15 nodes.

After a device discovery phase, the inquiring node will normally initiate a *Service Discovery* phase in order to find out if the detected nodes belong to the same application. The outcome of a Service Discovery is a URL which can be used to connect to the announced service in another node.

Whereas the Device Discovery phase is mandatory in order to establish a Bluetooth link between two nodes, the Service Discovery phase can be bypassed if one happens to know the associated URL through other means (caching, lookup etc.)

Despite the fact that other technologies would be better suited for fast device discovery (e.g. RFID), the availability and deployment scale of Bluetooth makes it the chosen technology for this research effort, which also includes studies of different optimization techniques in order to bypass the Service Discovery phase

where possible:

- Bypass the Service Discovery Protocol (SDP). One purpose of the SDP is to determine the URL necessary to connect to a particular service of a Bluetooth node. This URL will change each time the node restarts its service. Our choice has been to put the URL as a tuple in tuplespace when the service is started, so that a client may look for the URL in tuplespace rather than doing a SDP inquiry. If there is no URL in tuplespace, or the given URL does not work, the node initiates a SDP inquiry. Experimental evaluation estimates the effect of this technique to be approximately 1.1 second.
- Don't let beacons do Device Discovery (DD). During Device Discovery a node cannot be discovered or receive connections from others, so a beacon increases its availability to others if it refrains from DD. Mobile nodes will know that this is a beacon (by convention in its Friendly name) and connect to it. Since two beacons are never expected to connect to each other, this scheme works without problems.

To improve the reliability of zone detection, a “diffusion” technique has been considered, but not tested in practice. A node that for some reason has not detected the present zone may be informed about the zone from another mobile node. This technique may increase the number of nodes that gets informed about the present zone, possibly on the expense of accuracy, since mobile node may diffuse inaccurate information under some circumstances. The diffusion technique

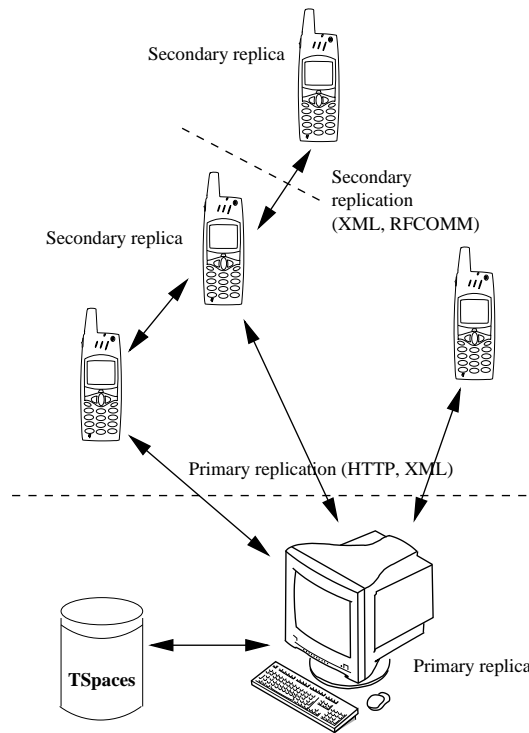


Figure 2: Architecture overview of the distributed store

is thoroughly presented by Spratt in [17] and is adopted by the Bluetooth Local Positioning Working Group as its positioning algorithm.

6 Mobispace system overview

The system diagram is shown on Figure 2. As seen on this figure, node types are either *primary* or *secondary*, which imposes different roles on them, and the replication sessions between them are different.

6.1 Primary-based replication

The presented approach to a distributed tuplespace uses a ‘primary server’, i.e. a computer with enough resources to keep all the data in the tuplespace. Mobile J2me units serve as ‘secondary servers’ (or simply ‘secondaries’) which keep a selection of the tuplespace on behalf of local clients, but they are also able to exchange tuples without primary server involvement.

The approach where secondaries can exchange information directly makes this system different from ordinary primary-based replication system [18, pp.337–341], since the primary does not necessarily have the most current state of the system; new tuples may be created in a secondary and passed on to other secondaries before they eventually become known to the primary server.

6.2 The interest profile

Secondaries are (for resource reasons) not expected to keep the entire tuplespace, but a selection of tuples from it. A secondary A expresses its selection criteria as a set of templates called an *interest profile*, $IP[A]$. Tuples not matching any of the templates in the interest profile will never be delivered to the secondary and thus remain unknown to the clients of this secondary.

6.3 UUID, Local Timestamps and Deatch Certificates

During operation, a node A maintains a logical clock, $LC[A]$ which is a counter that is incremented by every event in the node. Every time a message is sent

or a tuple is created, the clock is incremented. All created tuples are assigned a globally unique id, $t.uuid$, formed by appending a large random number at the end of the logical clock value³. During replication, the two nodes exchange their logical clock value, and the clocks are adjusted to the highest of the two values. This arrangement, known as *Lamport timestamps* [11], ensures that for every pair a, b of tuples in the entire system, where $a \rightarrow b$ (meaning that a causally precedes b):

$$(a \rightarrow b) \Rightarrow a.uuid < b.uuid \quad (7)$$

I.e. any tuple that causally precedes another will have a lower UUID value in the entire system.

The uuid value remains constant during the lifetime of the tuple, also during replication. The tuple will have an additional *local timestamp* which is simply the value of the logical clock when the tuple was created or received. The role of the local timestamp is to assist in the tuple selection during replication sessions. The local timestamp $t.ts$, is causally ordered within the scope of a tuplespace node. For all tuples stored in the same node, the following is true:

$$(a \rightarrow b) \Rightarrow a.ts < b.ts \quad (8)$$

When tuples are deleted, they are replaced by *Death Certificates* (DC) which will inhibit the tuple to “ressurrect” during replication. In principle, the DCs must be kept forever since the population of secondaries are unknown and that they repli-

³Other source of unique numbers, like MAC- or Bluetooth addresses, may be used as well

cate with unknown time intervals, but a design choice has been made to delete the DCs in node A that are older (have a UUID value less than) the variable $TS_{odc}[A]$ ⁴. The consequence is that a secondary A cannot accept tuples with $t.uuid < TS_{odc}[A]$, i.e. older than the oldest Death Certificate (but can accept DCs of any age). When a tuple is deleted, it is actually converted to a DC by marking it as “dead”. It retains all its original field values, but is given a new $t.ts$ value.

7 Replication sessions

There are two distinct replication sessions in this system: Replication of data between a primary server and a secondary (called *primary replication*) and between two secondaries (called *secondary replication*). Primary replication will typically use a connection over a GSM/GPRS service, a wired network (cradle or Ethernet) or even a Bluetooth connection to a combined GPRS/Bluetooth unit acting as a connection proxy.

Secondary replication happens between secondaries over a Bluetooth connection. The Bluetooth technology offers excellent services for ad-hoc connections, since discovery of devices and services is an integral part of the protocol stack. The units will therefore spontaneously connect to other devices within radio range.

⁴More correctly, TS_{odc} is compared to the LC that can be extracted from the UUID (the leftmost digits).

7.1 Primary replication

The secondary A keeps a record of the logical clock value at the end of the last primary replication, $TS_{pr}[A]$. It connects to the primary server P (through a HTTP connection), and sends to P a message with the following content:

1. All tuples (including DCs) with $t.ts > TS_{pr}[A]$
2. Its interest profile, $IP[A]$
3. Its $TS_{pr}[A]$
4. The value of its logical clock, $LC[A]$.

The primary server will enter the received tuples into its tuplespace (existing tuples already received from elsewhere are ignored). Received DCs will replace tuples with the same $t.uuid$. The primary server will now select every tuple t from the entire tuple collection \mathbf{T} that matches the secondary's interest profile $match(IP[A], \mathbf{T})$ and has a $t.ts > TS_{pr}[A]$. The response message back to the primary will contain these elements:

1. All selected tuples
2. Value of the logical clock, $LC[P]$

The secondary A accepts a received tuple t if $t.uuid > TS_{odc}[A]$ and the tuple does not already exist in the node (but assigns $t.ts = LC[A]$), and stores the received value of $LC[P]$ into $TS_{pr}[A]$. After a complete replication, the secondary can choose to delete some of the oldest Death Certificates and advance the $TS_{ods}[A]$ accordingly.

7.2 Secondary replication

The secondary replication is more “symmetric” than the primary replication, although the parts must take on different roles, which we will call $S1$ and $S2$. Each node S will keep a vector $TS_{sr}[N, S]$ containing the timestamp of the previous secondary replication with the secondary named N . In case no data about N is available, the value of $TS_{odc}[S]$ is used.

7.2.1 Template requirements

In order to maintain causal consistency during secondary replication, the sender and the receiver must have “equal” templates in their interest profiles. If a sender S will send a tuple t from its collection $\mathbf{T}[S]$ to receiver N because the receiver has presented a template r in its interest profile $IP[N]$ and $t \in match(r, \mathbf{T})$, it must be sure that all causally preceding tuples ever to be received by N is present in $\mathbf{T}[S]$. Otherwise, tuples preceding t may later be received by N from other nodes, which would violate causal consistency.

Therefore, S will only send tuples to N that match those templates in $IP[N]$ for which S has an *equal template* in $IP[S]$. Two equal templates have the same number of fields, and each pair of field has the same type and value (regarding “wildcard” as a value).

When S receives $IP[N]$ during secondary replication, it will “prune” the templates in it and remove those templates for which S does not have an equal template in $IP[S]$. The resulting interest profile is denoted as $IP_p[N]$. We will re-visit

the causal consistency issues in the next section.

7.2.2 Secondary replication protocol

The role of $S1$ (the “client”):

1. Send the interest profile $IP[S1]$, the value of $TS_{sr}[S2, S1]$, and the current value of $LC[S1]$
2. Receive the tuples selected according to $TS_{sr}[S2, S1]$ and the pruned interest profile $IP_p[S1]$. Accept those “new” tuples that have $t.uid > TS_{odc}[S1]$. In the same message, receive the interest profile $IP[S2]$ and $TS_{sr}[S1, S2]$ and $LC[S2]$ from $S2$.
3. Select the tuples that matches $IP_p[S2]$ and $t.uid > TS_{sr}[S1, S2]$, and send these together with the current value of $LC[S1]$.
4. The value of $LC[S2]$ received in message 2 is stored as the new value of $TS_{sr}[S2, S1]$

The role of $S2$ (the “server”) is simply the opposite: It receives a message containing $IP[S1]$ and $TS_{sr}[S2, S1]$, selects relevant tuples and returns them to $S1$ together with its $IP[S2]$ and $TS_{sr}[S1, S2]$. It then receives the tuples that $S1$ has selected and $LC[S1]$.

8 Why is this causally consistent?

Tuples are causally ordered from the following reasons:

- During retrieval operations, tuples are ordered by their local timestamp $t.ts$, i.e. when more than one tuple match a template, the tuple with the lowest timestamp value is returned.
- A created tuple t will be assigned a local timestamp value higher than any other tuples in this node (since the local clock is ever-increasing). *No other* tuples than the locally stored tuples can causally precede t . Therefore, a collection of tuples all created locally will have a causal ordering on local timestamp values.
- During replication sessions, tuples are exported by S to another node R in increasing timestamp order. Therefore, if $t.ts < u.ts$, then t will be sent before u and have the lowest timestamp value assigned by R . Thus, the same relation between the two tuple timestamps holds after replication. If t and u are causally related, they are (by the definition in Section 4.1) matching the same template $tmpl$:

$$(t \rightarrow u) \Rightarrow (t, u) \in match(tmpl, \mathbf{T}[S]) \quad (9)$$

And, since S and R have equal templates in their interest profiles, all causally related tuples in S will be sent to R during a secondary replication session.

- Causal consistency during primary replication is maintained since tuples accepted from a secondary S are either tuples created in S or tuples received from another secondary with an equal template, and in both cases

are causally related tuples received by the primary in causal order. Causally related tuples will therefore always have a local timestamp value according to Equation 8

9 Mobispace implementation discussion

The design which has been presented in this paper has been implemented in Java. The code for maintaining the secondary replica has been programmed on the Java 2 Micro Edition API, and the primary server has been programmed as a Java servlet. The primary server uses the TSpaces system as its “storage engine”, as indicated on Figure 2. This section will provide a few remarks to the implementation efforts.

9.1 Take/delete semantics

The system offers a consuming retrieve operations, *take*, which in ordinary tuplespace implementations acts like an atomic read-delete operation which guarantees that the tuple is retrieved by one and only one client. This form for coordination is infeasible in a distributed and occasionally-connected environment, so the *take* operation has been interpreted to affect only *other clients on the same node*. The tuple is replicated to other nodes (even after it has been taken) and may be retrieved elsewhere. The *take*-operation does not *delete* the tuple, it only makes it *invisible* from clients on the same node.

The *delete* operation has a different semantics, since it convert the given

tuplet to a Death Certificate, and the DC will be subject to replication and cause the tuplet to be deleted in other replica as well. Since causal consistency is ensured, the DC will never be replicated before the tuplet in question, so a tuplet replace operation (add new tuplet - delete the old) is safe and will be executed in the correct order everywhere.

9.2 Synchronization - multituples

There are blocking variants of the retrieval operations, called `waitToTake` and `waitToRead`. There are also operations that retrieve all matching tuples (in the correct order) and returns them in a vector.

9.3 Persistence management

The implementation offers persistent storage of tuples and all state information so that a unit can be switched off and on again and continue the expected service. The storage system resident in J2me systems has been used for this purpose. The persistence service can be set up so that it saves “checkpoints” of the system state at regular intervals, and the system will automatically perform as if it had been “rolled back” if it restarts after a crash.

9.4 Resource management

The semantics of the `take` operation raise a concern that need to be solved by the programmer: Two threads on different nodes may enter a producer/consumer

relationship where one thread (the producer) is doing a series of `write` operations and another (the consumer) a series of `take` operations. In our system, this would normally fill up the producer's node with a growing collection of written tuples. They are taken on another node, which does not affect the producer's storage. On the expense of transparency a `send` operation has therefore been introduced: It works like `write`, but leaves the tuple in an *invisible* state after the next primary replication. The `send` may be used under the condition mentioned above: The tuple is supposed to be consumed by clients on other nodes and is not of interest to clients on this node.

Invisible tuples will be treated like Death Certificates. They are deleted after an aging period.

9.5 Bluetooth operation

There exists a standard API, JSR-82⁵, for operating a Bluetooth device in a J2me unit. This API is implemented on a growing number of J2me-enabled mobile phones, and is offered as additional software on handheld units using Palm OS and Windows Mobile. The JSR-82 provides access to the Device and Service Discovery functions, and communication over the L2CAP and RFCOMM protocols.

Secondary replication over a Bluetooth connection is free, fast, and is designed to offer the secondary replicas the "latest news" and more up-to-date information without the cost of GSM/GPRS based communication.

⁵<http://www.jcp.org/en/jsr/detail?id=82>

10 Conclusion and future work

This paper has discussed the Mobispace middleware in the context of location-aware distributed applications. The research effort has exploited a middleware for distributed tuplespace and the short range of Bluetooth tranceivers for location management purposes.

Although Bluetooth discovery mechanisms are not ideally suited for applications that require fast discovery of a large number of mobile units, the deployment scale of Bluetooth-equipped units make them interesting alternatives for location-aware mobile applications.

The research presented in this paper offers an integrated approach to tuplespace-based distributed systems and location management, in the sense that the the tuplespace offers the client location-sensitive tuples which are transparently representing properties associated with the current location of the mobile client. It also offers eay access to other location management information like who is in a given location (zone) and where a particular node is located.

The project is in its early stage, and only simple experimenting has been done as a proof of concept. Future experimenting will address reliability and scalability issues. Besides, the JSR-82 implementation are often to be buggy and incompletely implemented.

The website www.mobispace.org offers updated information about the project and downloadable source code.

References

- [1] L. Aalto, N. Göthlin, J. Korhonen, and T. Ojala. Bluetooth and wap push based location-aware mobile advertising system. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 49–58, New York, NY, USA, 2004. ACM Press.
- [2] P. Bishop and N. Warren. *JavaSpaces in practice*. Addison Wesley Longman Inc., 2003.
- [3] R. Bruno and F. Delmastro. Design and analysis of a bluetooth-based indoor localization system. In *Personal Wireless Communications, IFIP-TC6 8th International Conference, PWC 2003, Venice, Italy, September 23-25, 2003, Proceedings*, pages 711–725, 2003.
- [4] A. T. S. Chan, H. V. Leong, J. Chan, A. Hon, L. Lau, and L. Li. Bluepoint: a bluetooth-based architecture for location-positioning services. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 990–995, New York, NY, USA, 2003. ACM Press.
- [5] A. Charles, R. Menezes, and R. Tolksdorf. On the implementation of swarm-linda. In *ACM Southeastern Conference (ACM-SE)*, Huntsville, AL, 2004.
- [6] A. Fongen and S. Taylor. A distributed tuplespace for j2me environments. In *16th IASTED International Conference on Parallel and Distributed Computing and Systems*, Phoenix, AZ, 2005.

- [7] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns and Practice*. Addison Wesley Longman Inc., Essex, UK, UK, 1999.
- [8] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
- [9] A. Göker, S. Watt, H. I. Myrhaug, N. Whitehead, M. Yakici, R. Bierig, S. K. Nuti, and H. Cumming. An ambient, personalised, and context-sensitive information system for mobile users. In *EUSAI '04: Proceedings of the 2nd European Union symposium on Ambient intelligence*, pages 19–24, New York, NY, USA, 2004. ACM Press.
- [10] IBM. Tspaces. Available from: <http://www.almaden.ibm.com/cs/TSpaces/> [Jun 20, 2005].
- [11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
- [12] S. Microsystems. Javaspaces. Available from: <http://www.sun.com/software/jini/specs/jini1.2html/js-title.html> [Jun 21, 2005].
- [13] M. Nilsson, J. Hallberg, and K. Synnes. Bluetooth positioning. In *CSEE 2002*, 2002.
- [14] A. Omicini. On the semantics of tuple-based coordination models. In *SAC '99: Proceedings of the 1999 ACM symposium on Applied computing*, pages 175–182, New York, NY, USA, 1999. ACM Press.

- [15] L. I. Patterson, R. S. Turner, and R. M. Hyatt. Construction of a fault-tolerant distributed tuple-space. In *SAC '93: Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing*, pages 279–285, New York, NY, USA, 1993. ACM Press.
- [16] G. P. Picco, A. L. Murphy, and G.-C. Roman. Lime: Linda meets mobility. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 368–377, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
- [17] M. Spratt. An overview of positioning by diffusion. *Wirel. Netw.*, 9(6):565–574, 2003.
- [18] A. S. Tanenbaum and M. v. Steen. *Distributed Systems. Principles and Paradigms*. Prentice Hall, 2002.