MOBISPACE: A DISTRIBUTED TUPLESPACE FOR J2ME ENVIRONMENTS

Anders Fongen  
The Norwegian School of IT  
Oslo, Norway  
email: anders@fongen.no

Simon J E Taylor  
School of Information Systems, Computing and Mathematics  
Brunel University  
Uxbridge, UK  
email: simon.taylor@brunel.ac.uk

ABSTRACT
The tuplespace programming model is based on the concepts of shared storage and synchronized retrieval. There has been much work on the design of distributed applications over tuplespaces. However, work in the use of tuplespaces in mobile distributed environments is still in its early stages. Our work proposes a system that is targeted at these environments. Our system is called MobiSpace and is implemented in Java 2 Micro Edition. This paper introduces MobiSpaces and discusses our approach taken to consistency and replication, as well as our experiences in implementing our prototype system.

KEY WORDS
Distributed tuplespace, causal consistency, j2me

1 Introduction

During development of distributed application, it is necessary to establish a set of roles for how the tasks are to be divided into smaller parts and assigned to network nodes. There is also a need to define the rules for how the roles are going to collaborate in order to meet the objectives of the application. The general approach is to directly assign tasks and data to nodes, and to hide the interactions between the nodes as method invocations. This is the basis for the familiar client-server model.

A more sophisticated approach is to let the nodes take on tasks and data in a dynamic and decentralized manner, which is the basis for the tuplespace model [3]. A large body of knowledge has been established on how to design distributed applications over the tuplespace abstraction.

We are interested in the use of the tuplespace model in mobile environments. Currently, there are few implementations. This may be due to the fact that most tuplespace implementations require a degree of connectivity which is likely to be unavailable in an occasionally connected mobile application. Our contribution described in this paper is a description of MobiSpace, a tuplespace system designed for mobile applications. The system is designed for the scarce set of resources present in a mobile unit and for connection interruption of unknown length. For portability reasons, the Java 2 Micro Edition (j2me) platform has been chosen for the implementation. We have also identified the necessary semantics for a distributed tuplespace which is able to run on any platform and in any language.

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 using replication methods that exploits a combination of these communication facilities.

The usefulness of the presented work 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 tuplet 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

The rest of the paper is organized as follows: Chapter 2 introduces the tuplespace in a distributed and mobile environment. Chapter 3 presents and overview of the MobiSpace system and a discussion of its replication protocols. Chapter 5 provides a short proof on our consistency claims. Chapter 6 and 7 discusses our current implementation of MobiSpace. Section 8 concludes the paper with a summary and a short discussion of future work.
2 The context of the problem

The programming model known as “tuplespace” was proposed by Gelernter in 1985 [3] 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 [6, 2] and IBM TSpaces [4].

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 it is 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 \( T \) resulting in a set of tuples \( V \) can be expressed as follows:

\[
V = \text{match}(t_1, T)
\] (1)

The original Linda model [3] 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.

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.

No transactional protection is mentioned in the Linda proposal, but both JavaSpaces and TSpaces offer “composite atomic” operations, which means that e.g. the result of several write operations is indivisible.

2.1 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 [8] has presented a fault-tolerant distributed design which requires high availability of network resources. The LIME system (Linda in a Mobile Environment) [9] offers a platform for mobile agents which bring a small tuplespace with them as they migrate and make them accessible to other agents residing on the same host. The SwarmLinda system [1] offers a mechanism for distributed clustering of tuples in a p2p environment and claims to be highly scalable. No distributed tuplespace implementation for the j2me environment has been reported.

In order to conserve the transactional semantics of a tuplespace 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 tuplespace designed for an ocasionally-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 coordination between nodes is required. Ordering semantics combined with lazy replication appears to be useful concepts in such a scheme.

2.2 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.

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 are selecting 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 \(\langle b,a \rangle\).

Although much efforts have gone into semantic definitions of tuplespace-based coordination models, e.g. [7], there have been no reports on the semantics of tuple ordering.

2.3 Challenges

In the views of the authors, the current tuplespace implementations are not applicable for mobile systems for the following reasons:

- They do not allow for efficient operations in an occasionally-connected environment.
- They do not offer a clear semantics on tuple ordering, necessary for a replicated tuplespace.
- Java versions of tuplespace implementations (JavaSpaces and TSpaces) are not portable to j2me environment since they rely on object serialization.

In the remaining of this paper the design of our system MobiSpace will be presented which will allow mobile devices to coordinate their activities through a distributed tuplespace. The basis for the system is a number of j2me units with communication capabilities and one central server running Java 2 Standard Edition (j2se). The system is designed with the limited resources in a j2me unit and with the occasionally-connected nature of mobile system in mind.

3 System overview

The system diagram is shown on Figure 1. 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.

3.1 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 [10, 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.

3.2 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.

3.3 UUID, Local Timestamps and Death 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 value \(^1\). 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 [5], ensures that for every pair \(a, b\) of tuples in the entire system, where \(a \rightarrow b\) (meaning that \(a\) causally precedes \(b\)):

\[
(a \rightarrow b) \Rightarrow a.uuid < b.uuid
\]

\(^1\)Other source of unique numbers, like MAC- or Bluetooth addresses, may be used as well.
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 \( t.t.s \), is causally ordered within the scope of a tuplespace node. For all tuples stored in the same node, the following is true:

\[
(a \rightarrow b) \Rightarrow a.t.s < b.t.s
\]  

When tuples are deleted, they are replaced by Death Certificates (DC) which will inhibit the tuple to “resurrect” during replication. In principle, the DCs must be kept forever since the population of secondaries are unknown and that they replicate 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 \( TS_{odec}[A] \). The consequence is that a secondary \( A \) cannot accept tuples with \( t.uuid < TS_{odec}[A] \), 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 \( t.t.s \) value.

4 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.

4.1 Primary replication

The secondary \( A \) keeps a record of the loccical 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 \( t.t.s > TS_{pr}[A] \)
2. Its interest profile, \( IP[A] \)
3. Its \( TS_{pr}[A] \)

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 \( T \) that matches the secondary’s interest profile \( match(IP[A], T) \) and has a \( t.t.s > TS_{pr}[A] \). 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 \( t.uuid > TS_{odec}[A] \) and the tuple does not already exist in the node (but assigns \( t.t.s = LC[A] \), 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 \( TS_{odec}[A] \) accordingly.

4.2 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 \( TS_{odec}[S] \) is used.

4.2.1 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 \( T[S] \) to receiver \( N \) because the receiver has presented a template \( r \) in its interest profile \( IP[N] \) and \( t \in match(r, T) \), it must be sure that all causally preceding tuples ever to be received by \( N \) is present in \( T[S] \). 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 have 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 \( IP_{pr}[N] \).

We will re-visit the causal consistency issues in the next section.
4.2.2 Secondary replication protocol

The role of \( S_1 \) (the “client”):

1. Send the interest profile \( IP[S_1] \), the value of \( TS_{sr}[S_2, S_1] \) and the current value of \( LC[S_1] \).

2. Receive the tuples selected according to \( TS_{sr}[S_2, S_1] \) and the pruned interest profile \( IP_p[S_1] \). Accept those “new” tuples that have \( t.uuid > TS_{ode}[S_1] \). In the same message, receive the interest profile \( IP[S_2] \) and \( TS_{sr}[S_1, S_2] \) and \( LC[S_2] \) from \( S_2 \).

3. Select the tuples that match \( IP_p[S_2] \) and \( t.uuid > TS_{sr}[S_1, S_2] \), and send these together with the current value of \( LC[S_1] \).

4. The value of \( LC[S_2] \) received in message 2 is stored as the new value of \( TS_{sr}[S_2, S_1] \).

The role of \( S_2 \) (the “server”) is simply the opposite: It receives a message containing \( IP[S_1] \) and \( TS_{sr}[S_2, S_1] \), selects relevant tuples and returns them to \( S_1 \) together with its \( IP[S_2] \) and \( TS_{sr}[S_1, S_2] \). It then receives the tuples that \( S_1 \) has selected and \( LC[S_1] \).

5 Why is this causally consistent?

Tuples are causally ordered from the following reasons:

- During retrieval operations, tuples are ordered by their local timestamp \( t.t_s \), 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 \( t.t_s < u.t_s \), 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 2.2) matching the same template \( tmpl \):

\[
(t \rightarrow u) \Rightarrow (t, u) \in \text{match}(tmpl, T[S])
\]

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 3.

6 Implementation discussion

The design which have 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 1. This section will provide a few remarks to the implementation efforts.

6.1 Take/delete semantics

The system offers a consuming retrieve operations, \texttt{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 \texttt{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 \texttt{taken}) and may be retrieved elsewhere. The take-operation does not \texttt{delete} the tuple, it only makes it invisible from clients on the same node.

The \texttt{delete} operation has a different semantics, since it convert the given tuple to a Death Certificate, and the DC will be subject to replication and cause the tuple to be deleted in other replica as well. Since causal consistency is ensured, the DC will never be replicated before the tuple in question, so a tuple replace operation (add new tuple - delete the old) is safe and will be executed in the correct order everywhere.

6.2 Synchronization - multitudes

There are blocking variants of the retrieval operations, \texttt{waitToTake} and \texttt{waitToRead}. There are also operations that retrieves all matching tuples (in the correct order) and returns them in a vector.

6.3 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.
6.4 Resource management

The semantics of the `take` operation raises 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 leave 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.

6.5 Bluetooth operation

There exists a standard API, JSR-82, 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.

7 Conclusion and future work

The tuplespace is well suited for applications that need both a shared medium for storage of application data and a message oriented service for a publish/subscribe type of thread communication.

We have presented a straightforward tuplespace implementation of the j2me environment, where the occasionally-connected nature of mobile applications has been an important consideration. The presented system is flexible and resilient to different networking conditions and provides an ordering schemes which is straightforward in practical use.

MobiSpace has been subject to prototype evaluation and verification. Future work on the system will include a proof-of-concept application within a mobile learning experiment, where a personalized “student handbook” will be represented as a tuplespace and replicated to mobile units based on the needs (expressed in the interest profile) of its owner.

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

References


---

1 http://www.jcp.org/en/jsr/detail?id=82