Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

A Java Based Simulator for the PeerSpaces Coordination Language

Fernando Magno Quintão Pereira
Wendell Figueiredo Taveira

Technical Report
Programming Language Laboratory

LLP002/2002

Av. Antônio Carlos, 6627
31270-010 - Belo Horizonte - MG

August 26, 2003

Abstract. This technical report presents a Java-based simulator for the PeerSpaces coordination model. PeerSpaces is a shared space coordination model designed for ad hoc mobile networks. The model is based on the concept of tuple spaces firstly proposed by Linda parallel programming language. In order to suit well the dynamic environment that characterizes ad hoc networks, PeerSpaces does not assume the presence of any centralized structure for communication and coordination. This report also describes an algorithm for termination detection in ad hoc networks built on PeerSpaces using the proposed simulator.

1 Introduction

Ad hoc networks are mobile networks that do not rely on any based station infrastructure for communication [6, 12]. Instead, hosts depend on each other to send and receive messages. In ad hoc networks, two hosts can exchange messages whenever they happen to be at communication range. Bluetooth is an example of network technology that makes this form of wireless communication possible.

Designing applications for such networks presents many interesting problems [15, 16]. Particularly, coordination is a challenging task. Since a user can find himself in a different network at any moment, the services available to him change along the time. Thus, computation should not rely on any predefined and well known context. Also, coordination should not assume the existence of any centralized node, because permanent availability of this node cannot be granted. Communication should also be uncoupled in time and space, meaning that two communicating entities do not need to establish a direct connection to exchange data nor must know the identity of each other.

This report describes a Java based simulator for the PeerSpaces model and a distributed algorithm built on this simulator. PeerSpaces is a coordination model for ad hoc networks based on Linda [5]. In Linda, several process communicate through a central data repository called *tuple space*. Processes communicate by inserting (**out**), reading (**rd**) and removing (**in**) ordered sequences of data from this space. Tuple retrieving is associative because it is based on a pattern against which a matching tuple is non-deterministically chosen from the space. If a matching tuple is not found, the caller process is suspended until such tuple is posted in its tuple space.

Communication in Linda presents many characteristics that are desirable in mobile environment. Particularly, communication is asynchronous and uncoupled in time and space. Communicating processes do not need to create a socket-like connection to exchange data. The associative mechanism allows communication based on the contents of the messages rather than on their addresses or other identifiers. Moreover, the blocking semantics used to retrieve tuples automatically provides synchronization among processes. All these features are important in mobile systems, since they are characterized by dynamic and short-lived patterns of communication

In traditional Java-based Linda systems, like TSpaces [19] and JavaSpaces [4], the tuple space is a centralized and global data structure that runs in a

pre-defined service provider. In the base station scenario this server can easily be located in the fixed network. However, if operation in ad hoc mode is a requirement, this choice is not available, since in this case the fixed infrastructure simply does not exist. This suggests that standard client/server implementations of Linda are not suitable to ad hoc scenarios, since they assume a tight coupling between client and servers and the permanent availability of the latter.

Aiming to answer the new requirements posed by ad hoc mobile computing systems, PeerSpaces departs from the traditional client/server architecture used in Linda and push towards a completely decentralized one. In the model, each node (or peer) has the same capabilities, acting as client, shared space provider and as router of messages. In order to provide support to operation in ad hoc mode, service lookup is distributed along the network and does not require any previous knowledge about its topology.

The report is organized as follows. In Section 2 we informally present the PeerSpaces model, including its main design goals, concepts and primitives. Section 3 gives an overview of a simulator for the PeerSpaces model and describes how the more complex primitives of the model are implemented. Section 4 describes the implementation and simulation in PeerSpaces of a termination detection algorithm. Detecting the termination of diffusing computation is a important problem in the distributed processing field and is used in this documment as an example of distributed algorithm that can be build on PeerSpaces. Finally, Section 6 concludes the report.

2 PeerSpaces: A Coordination Model for Ad Hoc Mobile Systems

PeerSpaces assumes an ad hoc network of mobile devices. Thus, there is no infrastructured network and hosts may connect or disconnect at any moment. As usual in ad hoc settings, two hosts can communicate when their wireless interfaces are in the same vicinity. The model does not assume any centralized structure and does not promise to provide any kind of shared memory abstraction encompassing connected hosts. Instead, it fosters a peer to peer model of computation, where any connected node has the same capabilities. Furthermore, hosts can discover each other using a decentralized lookup service and then communicate using remote primitives.

The main concepts used in PeerSpaces are the following:

Hosts The model assumes that hosts are mobile devices. Each host has its own local tuple space and a set of running threads. The host-level tuple space has three main purposes. First, it is used for local coordination among threads running in the host. Second, it is used for remote communication, since there are primitives in the model to retrieve and output messages in the space of remote hosts. Third, it is used to publish *services* and to retrieve the results of *lookup queries*. A service is any entity available in the host that can be useful to other hosts. Services in PeerSpaces are defined by tuples, whose fields describe the

attributes of the service. Finally, a lookup query is a query performed along the network to discover services.

Network Mobile hosts in the model are connected by a wireless and ad hoc network. As usual in such networks, connectivity is transient and determined by the distance among hosts. Consequently, the topology of the network is continuously changing. Moreover, any host in the network can act as router, propagating messages between nodes that are not directly connected.

2.1 PeerSpaces Primitives

PeerSpaces defines a set of primitives to assemble applications using the previous defined concepts. The set of primitives of PeerSpaces is a superset of the primitives originally proposed by Linda.

As in Linda, the **out** t primitive inserts tuple t into the tuple space of the local node. The remote **out** h, v primitive inserts tuple v in the tuple space of host h. The remote **out** primitive is asynchronous meaning that if host h is not reachable the operation returns after leaving the tuple v temporarily stored in space of the issuing host h'. When a communication path is established between h' and h the tuple is automatically transmitted to its final destination.

As in Linda, the **in** t, x primitive removes a tuple matching pattern t from the local space of a node and binds it to x. If there are several matching tuples, one of then is chosen non-deterministically. If there is no matching tuple, the calling thread remais blocked until the operation can be completed. The remote operation specifies a remote host name as a parameter as in **in** h, v, x.

The non-blocking version of **in** is called **inp** (*probe* **in**). If it is not possible for **inp** t, x to match a tuple, a null reference is binded to the name x. There is also a remote version for the **inp** operation. The last primitive of the **in** family is called **ing**. The operation **ing** $t, a[]$ retrieves from the local tuple space all tuples matching pattern t and stores them into the array a. The operation does not block the calling thread if no tuple is found.

In order to retrieve information from tuple spaces without removing the data PeerSpaces provides a reading operation. The **rd** t, x operation is similar to a local **in** but it does not change the state of the tuple space. The remote version of the operation is **rd** h, v, x and the non blocking version is named **rdp**. Like **rd**, the non blocking operation also has a remote version. Finally, **rdg** $t, a[]$ fills the array a with all tuples in the local space matching pattern t.

PeerSpaces extends Linda with a lookup primitive used to discover services in the network. Since the model is designed for use in ad hoc networks, this primitive does not make use of any central authority, as a directory service. Instead, the execution of the lookup primitive, called **find** , is distributed along the federation of connected devices. The **find** p operation queries hosts in the network for tuples matching pattern p. All matching tuples found are copied asynchronously to the local space of the host that has called the operation.

For example, a PDA-based auction system that wants to find other PDA's users selling 21 inches TVs despite its branch, price and seller can issue the follow-

ing operation: **find** $\langle \texttt{mall}, \texttt{sellers} \rangle, \langle \texttt{tv}, 21, ?, ?, ? \rangle$. The operation will trigger a query for tuples matching the pattern $\langle \texttt{tv}, 21, ?, ?, ? \rangle$ in the hosts of subgroup `sellers` of the root group `mall`. Matching tuples, like $\langle \texttt{tv}, 21, \texttt{foo}, 325, \texttt{h} \rangle$, where `foo`, 325 and `h` are respectively the TV's branch, the TV's price and the name of the mobile host offering the TV, will be outputted in the local space of the PDA sometime after the operation was issued. The PDA system can then retrieve these answers using the local **in** primitive and place an offer v using the remote operation **out** h, v.

The **find** operation originates a query that is propagated to all nodes of the network. Basically, the host that originated the query transmit it to its neighbors, that retransmit it to their neighbors and so on, until the network graph is covered. This protocol is similar to the one used by distributed file sharing systems in the Internet, like Gnutella [7]. Not surprisingly, the logical network created by Gnutella over the fixed Internet presents many characteristics that are typical of wireless and ad hoc networks.

Often it is useful to query connected hosts for a service and keep the query effective until such service is available. In this way, a client does not need to periodically send lookup queries to detect new services that may become available since the last query was issued. In PeerSpaces, lookup queries that remain active after their first execution are called continuous queries. Continuous lookup queries are issued adding the lifetime t to the **find** primitive: **find** p, t. This primitive will search the hosts of the network for available services matching pattern p and for services that may become available in t time units.

3 A Java-based Simulator for PeerSpaces

In order to design and test distributed ad hoc systems in PeerSpaces, a simulador was implemented for the model. The simulator was written in Java and is available as a Java package. For tuple space manipulation, the simulator uses a Java-based tuple space system named LighTS [8].

The simulator supports a stochastic model of simulation. The behavior of network elements is based on probabilistic parameters. The simulation is driven by a discrete time and random events are generated following the parameters defined by the user. The user specify the number of nodes in the network and the size of the grid where the nodes are located. For each node, the user defines its pattern of movement and the range of its wireless interface.

In order to implement and test a distributed algorithm, the user has to extend some pre-defined classes and interfaces. Each algorithm is a set of classes that implements the `Command` interface. This interface has two methods: `eval` (to evaluate the command guard) and `exec` to execute the command action. Several algorithms can be composed together in order to create a more complex one.

The simulator was implemented in two layers. In the first layer the user has the illusion that communication between two entities is carried on directly. In this layer, the user makes use of the primitives of the model. The second layer

implements the routing protocol used by the PeerSpaces primitives. In this layer, messages are transmitted only between adjacent nodes.

The routing algorithm implemented in the second layer of the simulator is based on flooding [17]. According to this technique, after receiving a message whose final address is not its address, a node send the message to all its neighbors. Although the implementation of this algorithm is straightforward, the exchange of information between two nodes that are not neighbors can generate a large number of messages in the network. On the other hand, flooding is fast and reliable in the sense that if there is a path connecting sender and receiver, then a message will be delivered in the shortest possible time. The problem of loops in the transmission of messages is solved by assigning identifiers to each message.

3.1 Implementation of the PeerSpaces primitives

Remote out In order to handle the asynchronism inherent to the remote **out** operation, it is executed in two steps. Suppose that a node h' executes an **out** h, v to post a message v in the space of host h. In the first step, a tuple v_h, called a *misplaced tuple*, is inserted in the tuple space of h'. At the same time, h' sends to h the tuple v_h. After receiving this tuple, h extracts v from v_h and attempts to retrieve the misplaced tuple from the tuple space of h' with an **inp** operation.

If there is not a path connecting h and h', tuple v_h will be kept in the local space of h' until its propagation to h is possible. Whenever a new node connects to the network, a notifying message is sent to each connected node. After receiving this message, a node retransmits all misplaced tuples it currently holds. Although expensive in terms of number of messages exchanged, this alternative makes sure that, if there is a possibility, the misplaced tuples will be delivered. In order to guarantee a exactly once semantic for the delivery of tuples in the remote **out** operation, each misplaced tuple owns an identifier that is stored by the receiving node.

Continuous queries Each node should keep a registration of all valid continuous queries which exists in a certain time. In order to address this necessity, every node was given a data structure called *temporary space* for keeping tuples wanted by persistent queries. The temporary tuples are augmented with three extra fields: $\langle c_1, c_2, \ldots c_n \rangle \Rightarrow \langle c_1, c_2, \ldots c_n, r, i, ttl \rangle$. r is a remote reference to the waiting node, ttl determines the instant of time when the querie will expire and i is a identifier to avoid inserting repeated queries in the same temporary space. Whenever a new tuple t is inserted in the regular tupla space of a node, the temporary space is scanned for tuples t' matching t. If the result of the search is positive, t is sent to the remote nodes that are looking for it.

The remotion of continuous queries can not be a task of the node that issued the search, because as the topology of a ad hoc network is dynamic, it is possible that the searching node do not is present when comes the time to remove the query. The solution was to bind a time to live to each persistent search. A kind of garbage collector service available on each node is responsible for checking the

validate of the temporary tuples and removing the ones whose period of life has passed.

Another question that is addressed in the simulator is the maintenance of consistent states in the network. Every node in a cluster of connected nodes has to keep the same set of continuous queries recorded in its temporary space. When the topology of the network chances, it is necessary to update the state of the nodes in order to keep the consistent property. This synchronization follows a best effort strategy similar to the one used to propagate queries in the network. For example, suppose the engagement of host h in group g. Any query owned by h that is not already in g is propagated to one of the hosts in the group, that in turn propagate it to its neighbors and so on, until the query is propagated to all hosts in the group. The same occurs with queries owned by a host in group g and that do not exist in h. Finally, when a lookup query is propagated from a host to another its remaining lifetime is honored by the target host.

The use of continuous queries can be expensive in terms of memory and traffic of messages but just its existence do not implies in any cost. This service should be avoided on large cluster of devices with very limited capacity because these systems can saturate rapidly. In order to minimize the costs of continuous queries it is worth considering the importance of the concept of group of nodes for reducing the size of the search space.

4 Example: A Termination Detection Algorithm

In order to present the main techniques for the design of distributed algorithms in the PeerSpaces model, an algorithm for termination detection of diffusing computation is described next. Termination detection is an important problem in distributed systems. Basically the problem consists in a node getting the information that a computation previously spread through the network has been done by every element in the system. For instance, in some public key cryptographic systems each node must change its public key after a time for the sake of security. If such a system is used in a distributed environment, when a node changes its key it should be assured that each other element of the network knows the new key before sending secure messages.

The solution described in this report was first presented in [14] and is based on a well-know solution proposed by Dijikstra and Scholten [3]. The approach adopted can be split in three phases. In the first phase, a partial ordering is build on the network, starting with a root node, in such a way that, in the end of the process, any node a, b and c will have different identifiers. Those identifiers must follow the transitive property on the *less than* operation, meaning that if $a < b$ and $b < c$ than $a < c$. The second phase of the algorithm involves the activation of idle nodes through the propagation of the job. Upon activation, a node becomes the child of the activating node, causing a transition from idle to active status. In the third and last phase, the termination of the distributed computation is actually detected. Each node after terminating the requested job,

passes this information to a node of higher rank and eventually all termination reports will reach the root node, which owns the highest rank.

Figure 1 describes the main variables and auxiliary functions used by the guarded commands of the algorithm.

n	Host identifier
root	Root node identifier
π	Name of the local tuple space of the node
count (t)	Number of occurrences of tuples matching pattern t
bestRank(a[])	Returns the highest rank in the array of tuples a
newRank (r)	Returns a rank identifier not yet used in the network
isBetter(r_1, r_2)	True if r_1 is a better rank than r_2

Fig. 1. Global variables for node n and auxiliary functions

The PeerSpaces simulator supports the modular construction of distributed programs. Thus, the termination detection algorithm was developed in three separated modules, which are explained in the rest of this section.

4.1 The Creation of a Partial Ordering in The Network

In this example, just one node is interested in detecting the termination of a distributed computation. It is called the root node. If the execution environment was a fixed network, every node achieved by the work request could report the event of finishing the job directly to the root node. In a ad hoc environment, however, a direct path connecting the root and a activated node may not exist. It is interesting also that a node becomes idle as soon as it finishes its task and reports this event just once aiming to reduce the traffic of messages. In order to solve these two problems, the termination detection algorithm involves the creation of a partial ordering in the network. The root node has the highest rank. A host, upon finishing its computation, sends this information to a node with a better rank. The received reports are recorded by the accepting node and propagated when it is possible. Following this pattern of transmission of events, the root node will receive the reports of all the nodes that compound the mobile network and will be able t o verify the termination.

The guarded commands responsible for creating the partial ordering in the network are exposed in the Figure 2. The root node is assumed to have a tuple like $\langle$"*root*"$\rangle$ in its local tuple space. The algorithm begins with the root node retrieving that tuple from there. The algorithm is powerful enough to allow the existence of more than one root node, because the tuples that represent the rank position of a node contain the identifier of the node responsible by that rank creation. In the command *RankPropagation*, each node, upon receiving the rank descriptor, sends an incremented rank value to each of its neighbors. A node can receive more than a rank descriptor because of this pattern of propagation. The command *RemoveExtraRanks* assures that in a given time a node will have only one rank tuple. The command *RemoveMisplacedTuple* guarantees that the

StartRankDistribution
Guard:
$\langle$"*rank*", String, String$\rangle \notin \pi \wedge n = root$
Action:
out $\langle$"*rank*", n, $highestrank\rangle$

RemoveExtraRanks
Guard:
count ($\langle$"*rank*", String, String$\rangle$) > 1
Action:
Tuple [] ranks;
ing $\langle$"*rank*", String, String$\rangle$, ranks
tuple $\tau \leftarrow$ bestRank(ranks)
out τ

RankPropagation
Guard:
$\langle$"*rank*", String, String$\rangle \in \pi \wedge \langle$"*activated*"$\rangle \notin \pi$
Action:
rdp τ, $\langle$"*rank*", id *as* String, r *as* String$\rangle$
$\forall$ neighbor v of n do
out v, $\langle$"*rank*", id, **newRank** $(r)\rangle$
out $\langle$"*activated*"$\rangle$

RemoveMisplacedTuples
Guard:
$\exists$ MisplacedTuples $\tau \mid \tau \in \pi$
Action:
do nothing

Fig. 2. Guarded commands for creating a partial ordering in the network

algorithm will not be interrupted while there are misplaced tuples generated by remote **out** operations in the local tuple space of any node.

In this algorithm, the rank is described by a sequence of k integer numbers like $n_1|n_2|\cdots n_k$. The rank descriptors are compared according two main criteria. Initially the number of fields is compared. The sequence with less fields is considered higher. If two sequences have the same number of fields, than the field values are compared from left to right. For instance, 0|1 is a better rank than "0|2" and "0|2" is higher than "0|0|0". The ranks transmitted by a node are always one field bigger than the rank that node first received. These new ranks are formed by appending an extra field in the received rank descriptors. Following this pattern, if a node with two neighbors is given the descriptor "0|1", it will send for one of the adjacent nodes the sequence "0|1|0" and the sequence "0|1|1" for the other.

4.2 Job Diffusion

Job diffusion begins with the root node and follows a flooding pattern of transmission. Each node, after receiving a job, *activates* all of its neighbors by retransmiting the job. The job executed in each node is represented by a tuple that matches the pattern $\langle$"*job*"$, n_o, ttl\rangle$, where n_o is the node that sent the tuple and *ttl* is an integer used to simulate the time needed to finish the execution of the job.

StartJobDiffusion
Guard:
$\langle$"*job*", n, $Integer\rangle \notin \pi \wedge n = root$
Action:
out $\langle$"*job*", n, *job duration*$\rangle$

RemoveRepeatedJobs
Guard:
count ($\langle$"*job*", String, Integer$\rangle$) $>$ 1)
Action:
Tuple [] jobs;
ing $\langle$"*job*", String, Integer$\rangle$, jobs
Tuple $\tau \leftarrow$ jobs[0]
out τ

JobPropagation
Guard:
$\langle$"*activeJob*", $String$, $Integer\rangle \notin \pi \wedge count(\langle$"*job*", $String$, $Integer\rangle) = 1$
Action:
rdp $\langle$"*job*", s_1 **as** String, ttl **as** Integer$\rangle, \tau$
if$(s_1 \neq n)$
out $s_1, \langle$"*child*", $n\rangle$
$\forall$ neighbor v of n do
out $v, \langle$"*job*", n, $ttl\rangle$
out $\langle$"*activeJob*", s_1, $ttl\rangle$

Table 1. Job diffusion

The activation of a node by another node is recorded by both participants, but in different ways. The node n_a that has received the job puts the tuple $\langle$"*child*", $n_a\rangle$ in the tuple space of the node that sent the job. Node n_a then changes its states from idle to active, which is indicated by the presence of the tuple $\langle$"*activeJob*", ?, ?$\rangle$ in the tuple space of the just activated node. Figure 1 describes the commands responsible for the job diffusion. Since flooding is used to propagate jobs, a node can receive the same job more than once. The command *RemoveRepeatedJobs* assures that just one job request will be addressed on a given task diffusion.

4.3 Verifing Termination

The tuple $\langle$"*activeJob*", ?, *ttl as Integer*$\rangle$ simulates the job assumed by a node. *ttl* is a integer representing the job complexity. At each iteration of the command *nodeExecuteTask* the *ttl* value is decremented by one until it reaches zero, meaning the job is done. When a node finishes the task it was given, it generates an *idle report* by putting a tuple matching $\langle$"*idleReport*", n, *children*$\rangle$ int its local tuple space. The next step concerns to propagate the idle report to a node with a better rank. A node performs this action by sending to every neighbor a tuple containing the information about the idle reports it currently holds. This tuple, called *node info* contains the rank, activated children and the identifier of the node. Upon receiving a such tuple, the receiving node compares its rank with the rank of the remote node. If the local rank wins, the receiving node attempts to retrieve the idle report tuple from the remote node. A non-blocking

NodeExecuteTask
Guard:
$\langle$"*activeJob*", s_1 *as String*, *ttl as Integer*$\rangle \in \pi \wedge ttl > 0$

Action:
rdp $\langle$"*activeJob*", s_1 **as** String, *ttl* **as** Integer$\rangle, \tau$
out $\langle$"*activeJob*", s_1, $ttl - 1\rangle$
if $(ttl - 1 = 0)$
Tuple [] children;
ing $\langle$"*child*", String$\rangle$, *children*
$\forall$ $\langle$"*child*", s_i **as** String$\rangle \in$ *children* do
out $\langle$"*idleReport*", n, $s_0|s_1|\cdots s_k\rangle$

PropagateIdleReports
Guard:
$\langle$"*idleReport*", String, String$\rangle \in \pi \wedge n \neq root$
Action:
rdp $\langle$"*rank*", *root* **as** String, *rank* **as** *String*$\rangle, \tau$
rdp $\langle$"*idleReport*", *father as* String, *children* **as** String$\rangle, \sigma$
$\forall$ neighbor v of n do
out $v, \langle$"*nodeInfo*", *father*, *children*, *rank*, $n\rangle$

AccepetIdleReports
Guard:
$\langle$"*nodeInfo*", String, String, String, String$\rangle \in \pi$
Action:
inp $\langle$"*nodeInfo*", f **as** String, ch **as** String, r **as** String, s **as** String$\rangle, \sigma$
rdp $\langle$"*rank*", *root* **as** String, *localRank* **as** String$\rangle, \tau$
if (isBetter(localRank, r))
out $\langle$"*idleReport*", $f, ch\rangle$
inp $s, \langle$"*idleReport*", $f, ch\rangle, x$

RemoveIdleLeaves
Guard:
$count(\langle$"*idleReport*", n, *String*$\rangle) > 1 \wedge \langle$"*done*"$\rangle \notin \pi$
Action:
inp $\langle$"*idleReport*", *father as String*, *children as String*$\rangle, \sigma$
inp $\langle$"*idleReport*", *child as String*, $\emptyset\rangle, \tau$
if ($\tau = \langle$"*idleReport*", *root*, $\emptyset\rangle$)
out $\langle$"*done*"$\rangle$
else
out $\langle$"*idleReport*", *father*, *children* $-$ *child*$\rangle$

Table 2. Termination detection

inp operation is used in this case because as the node info may be sent to several nodes more than one **in** operation could be performed over the same tuple.

The process of propagation of idle reports do not continues forever because there is not a node with a better rank than the root node. When receiving the idle reports the root attempts to build and prune a tree of activated nodes. If it is possible for the root to remove all the leaves from the tree it determines the termination of the distributed computation.

5 Related Work

Many characteristics of PeerSpaces have been inspired in file sharing applications popular in the Internet, like Napster [10], Freenet [2] and Gnutella [7]. Particularly, the peer to peer network created by Gnutella over the fixed Internet presents many properties that are interesting in mobile settings, like absence of centralized control, self-organization and adaptation to failures. PeerSpaces is an effort to transport and adapt such characteristics to mobile computing systems.

Jini [1] is a distributed object infrastructure that adds support to dynamic service registration and lookup to Java RMI [18]. However, the system assumes the existence of a central server to run the lookup service, which restricts its use to networks with base station support. The Jini framework also includes a Linda-like shared data space implementation, called JavaSpaces [4]. Once more, the system assumes that the data space resides in a central server, which precludes its utilization when operating in ad hoc mode. The same problem is shared by other client/server implementations of Linda, like TSpaces [19].

Lime [13, 9] introduces the notion of transiently shared data space to Linda. In the model, each mobile host has its own tuple space. The contents of the local spaces of connected hosts are transparently merged by the middleware creating the illusion of a global and virtual data space. Applications in Lime perceive the effects of mobility by atomic changes in the contents of this virtual space. However, even when used in a small federation of hosts, the main problems of transiently shared spaces are efficiency and scalability. The reason is the amount of global synchronization required to assure the consistency of the virtual space. Particularly, query operations must run as a distributed transaction to retrieve matching tuples. Moreover, the model allows users to define the destination tuple space of an outputted tuple. This leads to the notion of misplaced tuples, i.e., tuples that are temporally in a wrong tuple space waiting for the connection of its target host. Thus, the host engagement protocol also requires a distributed transaction to deliver misplaced tuples. Finally, disengagements in Lime should be announced, in order to remove event handlers placed at remote hosts.

There are several simulators for mobile distributed systems. Probably, the most well known are `ns` [11] and `GloMoSim` [20]. They do not support, however, the implementation of distributed algorithms using high level abstractions, like tuple spaces.

6 Conclusions

In this report we have presented a simulator for PeerSpaces, a coordination model for mobile computing systems. PeerSpaces was designed to overcome the main shortcoming of shared space coordination models when used in ad hoc wireless networks – the strict reliance on the traditional client/server architecture – while preserving the main strengths of such models – the asynchronous and uncoupled style of communication. The design of the model has privileged observance to

ad hoc networks principles. As usual in such models, transparency is sacrificed in name of scalability and soundness. In order to ilustrate the programming proposed by PeerSpaces, a termination detection algorithm was presented.

The PeerSpaces simulator, including its source code, can be downloaded from the URL: `http:\\www.dcc.ufmg.br\fernandm\~peerspaces`.

References

1. K. Arnold. *The Jini Specifications.* Addison-Wesley, 2nd edition, 2000.
2. I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *ICSI Workshop on Design Issues in Anonymity and Unobservability*, International Computer Science Institute, 2000.
3. E. W. Dijikstra and C. S. Scholten. Termination detection for diffusing computation. *Information Processing Letter*, i 4(11), Aug 1980.
4. E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice.* Addison-Wesley, Reading, MA, USA, 1999.
5. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan. 1985.
6. S. Giordano. *Mobile Ad-Hoc Networks*, chapter of Handbook of Wireless Networks and Mobile Computing. John Wiley & Sons, 2002.
7. Gnutella Home Page. `http://gnutella.wego.com`.
8. LighTS Home Page. `http://lights.sourceforge.net`.
9. A. L. Murphy, G. P. Picco, and G.-C. Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, May 2001.
10. Naspter Home Page. `http://www.napster.com`.
11. NS Home Page. `http://www.isi.edu/nsnam/ns/`.
12. C. Perkins. *Ad Hoc Networking.* Addison-Wesley, 2000.
13. G. P. Picco, A. L. Murphy, and G.-C. Roman. Lime: Linda meets mobility. In D. Garlan, editor, *Proceedings of the 21st International Conference on Software Engineering*, pages 368–377. ACM Press, May 1999.
14. C. Roman and J. Payton. A termination detection protocol for use in mobile ad hoc enviroment. Technical Report WUCS-01-29, Washington University, Department of Computer Science, St. Louis, Missouri, 2001.
15. G.-C. Roman, G. P. Picco, and A. L. Murphy. Software Engineering for Mobility: A Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 241–258. ACM Press, 2000.
16. M. Satyanarayanan. Fundamental challenges in mobile computing. In *ACM Symposium on Principles of Distributed Computing*, May 1996.
17. A. S. Tanenbaum. *Computer Networks.* Prentice Hall, 3rd edition, 1996.
18. A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems*, pages 219–232. USENIX Association, 1996.
19. P. Wycko, S. W. McLaughry, T. J. Lehman, and D. A. Ford. TSpaces. *IBM Systems Journal*, 37(3):454–474, Aug. 1998.
20. X. Zeng, R. Bagrodia, and M. Gerla. Glomosim: A library for parallel simulation of large-scale wireless networks. In *Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998.