A Coordination Model for Ad Hoc Mobile Systems and its Formal Semantics

Marco Túlio de Oliveira Valente[1], Roberto da Silva Bigonha[2], Mariza Andrade da Silva Bigonha[2], Fernando Magno Quintão Pereira[2]

[1]Department of Computer Science, Pontifical Catholic University of Minas Gerais
[2]Department of Computer Science, Federal University of Minas Gerais
Belo Horizonte, MG - Brazil
E-mail: mtov@pucminas.br, {bigonha,mariza,fernandm}@dcc.ufmg.br

Abstract

The growing success of wireless ad hoc networks and portable hardware devices presents many interesting problems to software engineers. Particular, coordination is a challenging task, since ad hoc networks are characterized by very opportunistic connections and rapidly changing topologies. This paper presents the formal semantics of a coordination model, called PeerSpaces, designed to overcome the shortcomings of traditional coordination models when used in ad hoc networks. The PeerSpaces model does not assume any centralized structure. Instead, it fosters a peer to peer style of computation, where any connected node has the same capabilities. Mobile devices can discover each other using a decentralized lookup service and then communicate using remote primitives. The paper presents the PeerSpaces model and gives its operational semantics in terms of a process calculus. Besides a formal specification of the model, the semantics presented in the paper supports formal reasoning about applications built on PeerSpaces.

1 Introduction

Recent advances in wireless networks and portable hardware technology are making mobile computing possible. Nowadays, users carrying laptops, personal digital assistants (PDAs) or cellular phones can continue working independently of their physical location. In the more traditional scenario, these users rely on a base station in the fixed network to route messages to other devices. Recently, with the advent of ad hoc networks, these devices can also detach completely from the fixed infrastructure and establish transient and opportunistic connections with other devices that are in communication range.

Designing applications on these dynamic and fluid networks presents many interesting problems [22]. Particularly, coordination is a challenging task. Since a user can find itself 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. Specifically when operating in ad hoc mode, coordination should not assume the existence of any central authority, since the permanent availability of this node can not 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.

Recently, shared space coordination models, inspired by Linda [11], are being considered for communication, synchronization and service lookup in mobile computing systems. The generative communication paradigm introduced by Linda is based on the abstraction of a tuple space. Processes communicate by inserting, reading and removing ordered sequences of data from this space. Tuple retrieving is associative since it is based on a pattern against with a matching tuple is non-deterministically chosen from the space. If a matching tuple is not found, the caller processes is suspended until such tuple is posted.

Communication in Linda presents many characteristics that are desirable in mobile settings. 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 very dynamic and short-lived patterns of communication

In traditional Linda systems, like TSpaces [25] and JavaSpaces [10], 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.

This paper formalizes our attempts to customize and adapt shared space coordination models to applications involving mobile devices with ad hoc network capabilities. The model formalized in the paper, called PeerSpaces, has primitives for local and remote communication, process mobility and service lookup. In order to answer the new requirements posed by ad hoc mobile computing systems, PeerSpaces departs from traditional client/server architectures 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 paper is organized as follows. In Section 2 we informally present the PeerSpaces model, including its main design goals, concepts and primitives. In Section 3 we give the formal semantics of the model in terms of a small language derived from the π-calculus. Besides a precise specification of the model, the semantics presented in this section supports formal reasoning about applications built on PeerSpaces. In Section 4 we use this semantics to prove two properties of PeerSpaces. Section 5 compares the model with similar efforts. Finally, Section 6 concludes the paper.

2 The PeerSpaces Model

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 running process. A host is written $h_g[\,P,\,T]$, where h is the name of the host, P is the process running in the host, T is its local tuple space, and g is the group of the host.

The host-level tuple space has three main purposes. First, it is used for local coordination among processes 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 *resources* and to retrieve the results of *lookup queries*. A resource is any entity available in the host that can be useful to other hosts, such as files, data, hardware devices etc. Resources in PeerSpaces are defined by tuples, whose fields describes the attributes of the resource. Finally, a lookup query is a query performed along the network to discover resources.

In PeerSpaces, the name of a host is different from the name of all other hosts. The model also assumes a infinite set H of possible host names.

Groups Hosts in the model are logically organized in groups. Each group has a name and can also contain subgroups, creating a tree structure. The group of a host is denoted by a tuple $\langle g_1, \ldots, g_n \rangle$, that specifies the path from the root group g_1 to the leaf group g_n where the host is located. For example, the tuple $\langle \texttt{pucminas}, \texttt{cs}, \texttt{proglab} \rangle$ denotes the set of hosts in the `proglab` group, which is a subgroup of the group `cs`, which is nested in the root group `pucminas`. Two

groups can have the same name, as long they are not subgroups of the same group. Groups are used in PeerSpaces to restrict the scope of lookup queries. The idea is whenever possible to look for resources only in the hosts that are members of a specific group.

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. In PeerSpaces, a network with hosts $h_1, h_2, \ldots, h_n$ is denoted by:

$$h_{1\,g_1}[P_1, T_1] \mid h_{2\,g_2}[P_2, T_2] \mid \ \ldots \ \mid h_{n\,g_n}[P_n, T_n] \ , \ E$$

where $g_1, g_2, \ldots, g_n$ are the group of the hosts and $E : H \times H$ is a relation representing connections among hosts. The presence of a pair (h_i, h_j) in E, denoted by $h_i \bowtie h_j$, indicates that host h_i is in communication range with host h_j. This relation is in continuous change to reflect reconfigurations in the network.

PeerSpaces also defines a set of primitives to assemble applications using the previous defined concepts. We spend the rest of this section describing such primitives.

Local Primitives The local tuple space of any host is accessed using the traditional **in**, **rd** and **out** primitives from Linda. Furthermore, there is a **chgrp** g primitive, used to change the group of the current host to the one specified by tuple g.

Process Mobility Processes in PeerSpaces are mobile in order to model the behavior of mobile agents. A mobile agent is a process that can move among sites carrying computation and accessing resources locally. In wireless environments, agents are a powerful design tool to overcome latency and to embed autonomous computations [15]. In the model, the primitive **move** $h.P$ is used to move a process to node h, where its execution continues as P. If host h is not connected, the operation blocks until the connection of such host. For the sake of simplicity, we decided to support only mobility of single processes. Support to multithread mobile agents can be added with some effort, as showed in [5].

Remote Primitives Crucial to the scalability and efficiency of any coordination model for mobile computing systems is the design of the remote operations. Thus, from the beginning PeerSpaces departs from the idea of providing seamlessly access to a global and centralized space. Instead, there are primitives that operate in the remote space of a well-known host h: **out** h, v; **in** h, p, x and **rd** h, p, x, where v is a tuple, p is a pattern and x is a variable. These operations are merely remote implementations of the traditional Linda primitives and thus does not impact in the overall performance of the system.

As their local counterparts, the remote **in** h, p, x and **rd** h, p, x primitives are synchronous and thus block until host h is connected and a matching tuple is available. Basically, these operations are used when a process needs a information from a remote host to proceed its execution or wants to know that a host is around.

As its local version, the remote **out** h, v primitive is asynchronous. The primitive is used when a process wants to leave a information to be consumed later in another host. In order to model its asynchronous behaviour, the operation is executed in two steps. In the first step, a tag is added to the tuple v to indicate that it should be transfer as soon as possible to the destination host h. The tagged tuple, denoted by v_h, is then outputed in the local space of host h' that requested the operation. In the second step, tuple v_h is transfered to the space of host h as soon as it is connected to h' and the tag is removed from the tuple. Since both steps are not atomic, while the tuple is "misplaced" in the source node it can be retrieved by an operation like **in** v_h. For example, this operation can be called by a garbage collector process in charge of reclaim tuples that are waiting for the connection of their destination host for a long time.

Lookup Primitive Without a lookup primitive the remote operations described above have little use, since a mobile host may not know in advance the name h of a service provider in its current network. Moreover, since the system is designed to support operation in ad hoc mode, the lookup service must not be centralized in a single host, but must be distributed along the federation of connected devices. In order to accomplish such requirements, there is in PeerSpaces the following primitive: **find** g, p. This primitive queries hosts in group g for tuples matching pattern p in a distributed way. All matching tuples

found in group g are copied asynchronously to the local space of the host that has called the operation.

The semantics of PeerSpaces does not assume any specific routing protocol for propagation of lookup queries. However, the semantics requires that any protocol used in a real implementation must be loop free, i.e., should avoid loops in the propagation of queries. Most of the algorithms that have been proposed for multicast routing in ad hoc networks achieve this property [12, 16].

Continuous Queries Often it is useful to query a group of hosts for a resource and keep the query effective until such resource is available. In this way, a client does not need to periodically send lookup queries to detect new resources 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.

The first fundamental question regarding continous queries is how to stop them. The choice of adding a primitive to revoke queries explicitly is not suitable in mobile settings, since unpredictable reconfigurations in the network can disconnect the host that issued the query from any host in charging of executing it. For this reason, continuous queries in PeerSpaces have a lifetime parameter, used to automatically garbage collect the query after its expiration. Continuous lookup queries are issued adding the lifetime t to the **find** primitive: **find** g, p, t. This primitive will search the hosts of group g for all currently available resources matching pattern p and for resources that may become available in t units of time after the query was issued.

The second fundamental question raised by continuous lookup queries is how to handle engagement of hosts in a group. In such situations, the set of queries in the group and in the new host should be synchronized. In PeerSpaces, this synchronization follows a best effort strategy. 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 destination host.

3 Formal Semantics

The ultimate goal of our research is to deploy a coordination middleware for ad hoc mobile computing systems. In order to achieve this goal we have initially defined the formal semantics of PeerSpaces. The purpose of the semantics is twofold. Firstly, it provides a solid and precise foundation for implementations of the model. Secondly, it can support formal reasoning about applications built on top of those implementations.

The formalization presented next uses an operational semantics based on the asynchronous π-calculus [17]. The π-calculus is good basis as it provides a small, elegant and expressive concurrent programming language. The main departure from π in our semantics is the use of generative communication instead of channel-based communication. The same idea has been explored in depth in other works [5, 4, 2, 9].

Table 1 summarizes the syntax of our core language. We assume a infinite set H of names, used to name hosts and lookup queries. Meta-variables h and x range over H. Basic values, ranged over by v and g, consist of names and tuples. Tuples are ordered sequences of values $\langle v_1, \ldots, v_n \rangle$. A tuple space T is a multiset of tuples. We use the symbol $? \in H$ to denote the distinguished unspecified value.

$Prog$	::=	$N\,,\,E\,,\,X$
N	::=	$\varepsilon \mid H \mid N$
H	::=	$h_g[\,P,\,T]$
P	::=	$\mathbf{0} \mid P \mid Q \mid !\,P \mid (\nu\, x)\,P \mid \mathbf{out}\ v \mid$
		$\mathbf{in}\ v, x.P \mid \mathbf{rd}\ v, x.P \mid \mathbf{find}\ g, p, t \mid$
		$\mathbf{chgrp}\ g \mid \mathbf{move}\ h.P$

Table 1: Syntax

A program is composed by the network N, the relation E and a global set of names X. The relation $E\colon H \times H$ represents the connectivity map of the network. The names used over several hosts in the system are recorded in the set X, ensuring their unicity. Each host h is member of a group g and has a running process P and a local tuple space T. Processes are ranged by P and Q. Similar to the π-calculus, the simplest term of our language is the inert process $\mathbf{0}$, which denotes a process with no behavior at all. The term $P \mid Q$ denotes two processes running

Reductions

Linda Primitives

$$h_g[\mathbf{out}\ v \mid P, T] \mid N\,,\, E\,,\, X \ \rightarrow\ h_g[P, v \cup T] \mid N\,,\, E\,,\, X \qquad \text{(L1)}$$

$$h_g[\mathbf{in}\ v, x.P \mid Q, v' \cup T] \mid N\,,\, E\,,\, X \ \rightarrow\ h_g[P\{v'/x\} \mid Q, T] \mid N\,,\, E\,,\, X \qquad \text{(L2)}$$

$$h_g[\mathbf{rd}\ v, x.P \mid Q, v' \cup T] \mid N\,,\, E\,,\, X \ \rightarrow\ h_g[P\{v'/x\} \mid Q, v' \cup T] \mid N\,,\, E\,,\, X \qquad \text{(L3)}$$

The rules are subjected to the following side conditions:
(L2) if $v \leq v'$
(L3) if $v \leq v'$

Structural Congruence Rules

$$P \mid Q \equiv Q \mid P \qquad \text{(SC1)} \qquad (\nu\, x)\,(\nu\, y)\,P \equiv (\nu\, y)\,(\nu\, x)\,P \qquad \text{(SC5)}$$

$$!\,P \equiv P \mid !\,P \qquad \text{(SC2)} \qquad P \equiv Q \Rightarrow (\nu\, x)\,P \equiv (\nu\, x)\,Q \qquad \text{(SC6)}$$

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad \text{(SC3)} \qquad (\nu\, x)\,(P \mid Q) \equiv P \mid (\nu\, x)\,Q \qquad \text{(SC7)}$$

$$P \mid \mathbf{0} \equiv P \qquad \text{(SC4)}$$

$$P \equiv Q \Rightarrow h_g[P, T] \equiv h_g[Q, T] \qquad \text{(SC8)}$$

$$h_g[(\nu\, x)\,P, T] \mid N\,,\, E\,,\, X \equiv h_g[P, T] \mid N\,,\, E\,,\, x \cup X \qquad \text{(SC9)}$$

$$h_g[P, T] \mid N\,,\, E\,,\, X \equiv N \mid h_g[P, T]\,,\, E\,,\, X \qquad \text{(SC10)}$$

The rules are subjected to the following side conditions:
(SC7) if $x \notin fn(P)$
(SC9) if $x \neq h,\ x \notin fn(N),\ x \notin X$

Pattern Matching Rules

$$v \leq v \qquad ? \leq v \qquad \frac{v_1 \leq v_1' \ \ldots\ v_n \leq v_n'}{\langle v_1 \ \ldots\ v_n \rangle \leq \langle v_1' \ \ldots\ v_n' \rangle}$$

Table 2: Core Language Operational Semantics

in parallel. The term $!\,P$ denotes a infinite number of copies of P, all running in parallel. The restriction operator $(\nu\, x)\,P$ ensures that x is a fresh and unguessable name in the scope of P. Similar to Linda, the primitive operations **out**, **in** and **rd** provide access to the local tuple space. Since the **out** operation is asynchronous it does not have a continuation P. The same happens to the **find** and **chgrp** primitives. We assume that non-continuous lookup queries can be simulated by defining the lifetime equal to zero. Finally, the **move** operation simulates the behavior of single thread mobile agents.

The operational semantics of our calculus is summarized in Tables 2 and 3. The semantics is defined in terms of a reduction relation $\rightarrow$, a structure congruence $\equiv$ between processes and a set of pattern matching rules.

Table 2 summarizes the core language semantics, which is basically Linda with multiple tuple spaces. A reduction $N\,,\, E\,,\, X \ \rightarrow\ N'\,,\, E'\,,\, X'$ defines how the configuration $N\,,\, E\,,\, X$ reduces in a single step computation to $N'\,,\, E'\,,\, X'$. Initially, there are three reduction rules describing the effects on the configuration of each standard Linda primitive. The output operation, **out** v, asynchronously deposits a tuple in the local space (rule L1). The input, **in** $v, x.P$, and read, **rd** $v, x.P$, operations try to locate a tuple v' that matches v (rules L2 and L3). If one is found, free occurrences of x are substituted for v' in P, denoted as $P\{v'/x\}$. In the case of the input, the tuple is removed from the space.

The next set of rules defines a structural congruence relation $\equiv$ between processes (SC1 to SC7) and hosts (SC8 to SC10). As in the π-calculus, such rules define

Reductions

PeerSpaces Primitives

$$h_g[\textbf{find}\ g', p, t \mid P, T] \mid N\ ,\ E\ ,\ X\ \rightarrow\ h_g[(\nu\, k)\, P, \langle k, g', p, t, h\rangle \cup T] \mid N\ ,\ E\ ,\ X \quad \text{(P1)}$$

$$h_g[\textbf{chgrp}\ g' \mid P, T] \mid N\ ,\ E\ ,\ X\ \rightarrow\ h_{g'}[P, T] \mid N\ ,\ E\ ,\ X \quad \text{(P2)}$$

$$h_g[\textbf{move}\ h'.P \mid Q, T] \mid {h'}_{g'}[P', T'] \mid N\ ,\ E\ ,\ X\ \rightarrow\ h_g[Q, T] \mid {h'}_{g'}[P \mid P', T'] \mid N\ ,\ E\ ,\ X \quad \text{(P3)}$$

Query Propagation

$$h_g[P, \langle k, g'', p, t, h\rangle \cup T] \mid {h'}_{g'}[P', T'] \mid N\ ,\ E\ ,\ X\ \rightarrow\ h_g[P, \langle k, g'', p, t, h\rangle \cup T] \mid {h'}_{g'}[P' \mid P'', \langle k, g'', p, t, h\rangle \cup T'] \mid N\ ,\ E\ ,\ X \quad \text{(Q1)}$$

Network Reconfiguration

$$\frac{E\ \Rightarrow\ E'}{N\ ,\ E\ ,\ X\ \rightarrow\ N\ ,\ E'\ ,\ X} \quad \text{(N1)}$$

The rules are subjected to the following side conditions:
(P3) if $h \bowtie h'$
(Q1) if $(h \bowtie h') \wedge (g'' \preceq g') \wedge (\langle k, g'', p, t, h\rangle \notin T') \wedge P'' = !\,(\textbf{rd}\ p, x.\textbf{out}\ h, x)$

Group Matching Rule

$$\frac{g_1 = g'_1\ \ldots\ g_n = g'_n}{\langle g_1\ \ldots\ g_n\rangle \preceq \langle g'_1\ \ldots\ g'_n \ldots\ g'_m\rangle}$$

Table 3: PeerSpaces Operational Semantics

how processes can be syntactically rearranged in order to allow the application of reductions. In such rules, we write $fn(P)$ to denote the set of names free in process P. The definition of pattern matching, written $v \leq v'$, allows for recursive tuple matching. Values match only if they are equal or if the unspecified value occurs on the left hand side.

Table 3 extends the core language with the primitives proposed in PeerSpaces. The **find** g', p, t operation deposits a tuple representing a service lookup query in the local space (rule P1). Such query is a tuple in the format $\langle k, g', p, t, h\rangle$, where k is a fresh name that identifies the query, g' defines the group where the query will be performed, p is a pattern for the desired service, t is the lifetime and h is the name of the current host. The operation **chgrp** g just changes the group of the current host to the one specified by tuple g (rule P2). If such group does not exist, it is created. The **move** $h'.P$ operation changes the location of the continuation process P to host h' if this host is connected (rule P3). Otherwise, the operation remains blocked until the engagement of h'.

Reduction rule Q1 defines how lookup queries are propagated in the network. Basically, any host that holds a query $\langle k, g'', p, t, h\rangle$ can propagate it to a connected host h' in group g', if g'' matches g' and the query is not yet present in h'. If such conditions are satisfied, the query is inserted in the local space of h' and a process P'' is added in parallel with the other processes running in this host. This process continuously read tuples matching the pattern p and then use a remote output operation to send the results to the local space of the host h that has issued the query. Query propagation can be interleaved with any number of reductions representing primitive operations. Furthermore, since queries are stored in the local and persistent tuple space, rule Q1 also handles propagation to matching hosts that further join the network.

The last reduction rule introduces a new type of reduction $\Rightarrow$ used to describe reconfigurations in the network and consequently in the connectivity relation E. Basically, this rule dictates that changes in E should be propagated to the current configuration. However, we left $\Rightarrow$ reductions unspecified in the semantics, since they are dependent on the physical location of each host and on technological parameters

Remote Primitives

$h_g[\mathbf{out}\ h', v \mid P, T] \mid N\,,\, E\,,\, X \ \rightarrow\ h_g[P, v_{h'} \cup T] \mid N\,,\, E\,,\, X$ (R1)

$h_g[P, v_{h'} \cup T] \mid h'_{g'}[P', T'] \mid N\,,\, E\,,\, X \ \rightarrow\ h_g[P, T] \mid h'_{g'}[P', v \cup T'] \mid N\,,\, E\,,\, X, \quad \text{if } h \bowtie h'$ (R2)

$h_g[\mathbf{in}\ h', p, x.P \mid P', T] \mid N\,,\, E\,,\, X \ \rightarrow$
$\quad h_g[(\nu\, k)\,(\mathbf{move}\ h'.\mathbf{in}\ p, y.\mathbf{move}\ h.\mathbf{out}\ \langle k, y\rangle \mid \mathbf{in}\ \langle k, p\rangle, x.P) \mid P', T]\,,\, E\,,\, X$ (R3)

$h_g[\mathbf{rd}\ h', p, x.P \mid P', T] \mid N\,,\, E\,,\, X \ \rightarrow$
$\quad h_g[(\nu\, k)\,(\mathbf{move}\ h'.\mathbf{rd}\ p, y.\mathbf{move}\ h.\mathbf{out}\ \langle k, y\rangle \mid \mathbf{in}\ \langle k, p\rangle, x.P) \mid P', T]\,,\, E\,,\, X$ (R4)

Table 4: Remote primitives semantics

of the subjacent network, such as network standards, power of the transmitters of each device etc.

There is also a special pattern matching rule for groups names. Two groups g and g' matches, written $g \preceq g'$ if all subgroups in g are equal to equivalent subgroups in g', which can also have extra nested subgroups. For example, $\langle \texttt{pucminas}, \texttt{cs}\rangle \preceq \langle \texttt{pucminas}, \texttt{cs}, \texttt{proglab}\rangle$, which means that queries sent to hosts in group $\langle \texttt{pucminas}, \texttt{cs}\rangle$ will also be performed in hosts of the group $\langle \texttt{pucminas}, \texttt{cs}, \texttt{proglab}\rangle$. Similarly, $\langle \texttt{pucminas}\rangle \preceq \langle \texttt{pucminas}, \texttt{cs}, \texttt{proglab}\rangle$, but $\langle \texttt{pucminas}, \texttt{eng}\rangle \not\preceq \langle \texttt{pucminas}, \texttt{cs}, \texttt{proglab}\rangle$. The extra flexibility of group matching makes unnecessary the use of the unspecified value in such rule.

3.1 Remote Operations

Table 4 defines the semantics of the remote operations. As described, remote output in PeerSpaces is a two step operation. The first step deposits the tuple v with a tag h, denoted by v_h, in the local space (rule R1). In the second step, this tuple is routed to its final destination h', when this host is connected to the network (rule R2). The remote **in** primitive can be explained as a process that moves to the remote host h to perform a local **in** (rule R3). When a matching tuple is found, this process returns to the issuing host h' and outputs the value removed with a key k that identifies the operation. A parallel process that was blocked removes the value and continues as P. The remote **rd** follows the semantics of a remote **in** (rule R4).

3.2 Garbage Collection

In order to garbage collect continuous queries should exist in each host a process that continuously decrement the lifetime of queries stored in its local space. If the lifetime of a query reaches zero, the garbage collector should discard the query and kill the process in charging of executing it (process P'' of rule Q1, in Table 3). For the sake of simplicity and readability, we decided not to add the garbage collector process in the semantics.

4 Properties of the Model

In this section, we use the described semantics to prove two fundamental properties of PeerSpaces.

Proposition 1 *A lookup query can reach any connected member of its target group before its expiration.*

Proof: Directly from rule Q1 (Table 3), which assures that a query can be propagated to any host h' reachable from the issuing host h and that is member of the query target group.

However, accordingly to the best effort semantics adopted in PeerSpaces, we can not guarantee that a lookup query will be propagated to *all* hosts of its target group. For example, a host can join and leave the network without a reduction Q1 being called to propagate a lookup query to it.

Proposition 2 *There are no loops in the propagation of lookup queries.*

Proof: We assume that all communication links are bi-directional. The proof is by induction on the length of the loops.

- *Basis:* The smaller possible loop in any network has length two. Consider a loop connecting host h_1 and h_2 and a lookup query q issued by h_1.

We can use rule Q1 to propagate the query from h_1 to h_2. However, the side condition of rule Q1 prevents the propagation of the query back to h_1, since it is already in the tuple space of this host.

- *Inductive Hypothesis:* Propagation of queries is loop free for connectivity loops of size less or equal n.
- *Inductive Step:* Suppose a loop of size $n+1$. We can use rule Q1 to forward a lookup query to hosts $h_1, \ldots, h_{n+1}$. We can also construct a loop of size n by establishing a communication link between h_n and h_1. However, by the inductive hypothesis, we can not propagate the query from h_n to h_1. Since the query is the same, we can neither propagate it from h_{n+1} to h_1. Thus, it is not possible to create a loop of size $n+1$.

Proposition 2 is fundamental to assure that queries are not indefinitely propagated along the network. However, in order to avoid such loops, the side condition of rule Q1 requires each node to cache the keys of all queries that it has already broadcasted.

5 Related Work

Many characteristics of PeerSpaces have been inspired in file sharing applications popular in the Internet, like Napster [19], Freenet [7] and Gnutella [13]. 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. This explains the choice of Linda shared spaces as the prime coordination infrastructure for PeerSpaces. As described in Section 1, it is well known the advantages of using Linda based models in mobile computing systems. In addition, commitment to shared spaces models differentiates PeerSpaces from Gnutella, where a query is just a string whose interpretation is defined by each node in charging of running it.

Jini [1] is a distributed object infrastructure that adds support to dynamic service registration and lookup to Java RMI [23]. 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 [10]. 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 [25].

Lime [21, 18] 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. A service discovery and provision system for ad hoc networks, built on top of Lime, is described in [14].

The scalability and performance weakness of Lime have motivated the proposal of CoreLime [4], where in name of simplicity and scalability the idea of transiently shared spaces is restricted to the set of mobile agents running in a host. Another work proposing an alternative semantics to the notion of transiently shared spaces is [3].

PeerWare [8] is another recent attempt to solve the well-known problems of Lime. The system relies on the notion of global and virtual data structures (GVDS). Each node in PeerWare has a local data structure in the form of a forest of trees, where the leaves are the shared documents. This structure is similar to the directory tree of traditional file systems. The GVDS created by the system is the "superimposition" of the local trees of connected peers. There is an operation `execute(Fn,Fd,a)` that executes an arbitrary action `a` in the projection of the GVDS de-

termined by functions `Fn` (that filters nodes) and `Fd` (that filters documents). The novelty is the recognition that global atomicity is an impractical assumption in mobile settings. For this reason, the model has variants of its operations that do not assume atomicity. However, the GVDS abstraction only makes sense if consistency is granted, which requires atomicity. If the model does not assure the consistency of the GVDS, it is reduced to remote evaluation.

Laura [24] is a shared space based language with service lookup primitives. It is centered in the notion of a service-space containing forms describing offers, requests and results of services. The system however is targeted to open and wide area distributed systems. The formal semantics presented in Section 3 of the current paper resembles the Ambient Calculus [6]. Unlike the Ambient Calculus, PeerSpaces adopts generative communication primitives and supposes the existence of a dynamic relation representing the configuration of the network.

6 Conclusions

In this paper we have presented and formalized PeerSpaces, a coordination model for mobile computing systems. The model was designed to overcome the main shortcoming of shared spaces 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. Each mobile host in PeerSpaces has its own tuple space, used to local coordination and to advertise services to other hosts. Hosts in the model can discover each other using a decentralized lookup service and communicate using remote primitives. 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.

PeerSpaces can be used as the building block of ad hoc mobile systems like file sharing, groupware, mobile commerce and message systems. A prototype implementation of PeerSpaces is described in [20].

References

[1] K. Arnold. *The Jini Specifications*. Addison-Wesley, 2nd edition, 2000.

[2] N. Busi, R. Gorrieri, and G. Zavattaro. A process algebraic view of Linda coordination primitives. *Theoretical Computer Science*, 192(2):167–199, Feb. 1998.

[3] N. Busi and G. Zavattaro. Some thoughts on transiently shared tuple spaces. In *Workshop on Software Engineering and Mobility. Co-located with International Conference on Software Engineering*, May 2001.

[4] B. Carbunar, M. T. Valente, and J. Vitek. Corelime a coordination model for mobile agents. In *International Workshop on Concurrency and Coordination*, volume 54 of *Electronic Notes on Theoretical Computer Science*. Elsevier Science, July 2001.

[5] B. Carbunar, M. T. Valente, and J. Vitek. Lime revisited. In *5th IEEE International Conference on Mobile Agents*, volume 2240 of *Lecture Notes in Computer Science*, pages 54–69. Springer-Verlag, Dec. 2001.

[6] L. Cardelli and A. Gordon. Mobile ambients. In M. Nivat, editor, *Foundations of Software Science and Computational Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[7] 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.

[8] G. Cugola and G. P. Picco. PeerWare: Core middleware support for peer-to-peer and mobile systems, 2001. Submitted for publication.

[9] R. DeNicola and R. Pugliese. A process algebra based on Linda. In P. Ciancarini and C. Hankin, editors, *1st International Conference on Coordination Models and Languages*, volume 1061 of *Lecture Notes in Computer Science*, pages 160–178. Springer-Verlag, 1996.

[10] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, 1999.

[11] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan. 1985.

[12] S. Giordano. *Mobile Ad-Hoc Networks*, chapter of Handbook of Wireless Networks and Mobile Computing. John Wiley & Sons, 2002.

[13] Gnutella Home Page. `http://gnutella.wego.com`.

[14] R. Handorean and G.-C. Roman. Service provision in ad hoc networks. Technical Report WUCS-01-40, Washington University, Department of Computer Science, 2001.

[15] D. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, 1999.

[16] S.-J. Lee, W. Su, J. Hsu, M. Gerla, and R. Bagrodia. A performance comparison study of ad hoc wireless multicast protocols. In *Proceedings of IEEE INFOCOM 2000*, pages 565–574, Mar. 2000.

[17] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[18] 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.

[19] Naspter Home Page. `http://www.napster.com`.

[20] F. M. Pereira, M. T. Valente, R. Bigonha, and M. Bigonha. Uma linguagem para coordenação de aplicações em redes móveis ad hoc. In *V Simpósio Brasileiro de Linguagens de Programação*, pages 152–165, June 2002.

[21] 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.

[22] 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.

[23] Sun Microsystems. Java Remote Method Invocation Specification, Oct. 1998.

[24] R. Tolksdorf. Laura — a service-based coordination language. *Science of Computer Programming*, 31(2–3):359–381, 1998.

[25] P. Wycko, S. W. McLaughry, T. J. Lehman, and D. A. Ford. TSpaces. *IBM Systems Journal*, 37(3):454–474, Aug. 1998.