PeerSpaces: Data-driven Coordination in Peer-to-Peer Networks

Nadia Busi Cristian Manfredini Alberto Montresor Gianluigi Zavattaro

Department of Computer Science
University of Bologna
Mura Anteo Zamboni, 7
40127 Bologna - Italy

{busi,manfredi,montresor,zavattar}@cs.unibo.it

ABSTRACT

Shared dataspaces à la Linda, and the underlying data-driven coordination model, have been successfully exploited in the development of a huge variety of applications, going from parallel computing to web-based collaborative work. In this paper we consider a novel class of applications, namely those developed for peer-to-peer networks à la Gnutella or FreeNet. We discuss the problems which arise when trying to exploit the original Linda coordination model in this new scenario. In order to address these problems, we introduce PeerSpaces, a new coordination model particularly suited for the realm of peer-to-peer network applications, and we present a prototypical implementation of this coordination model based on the JXTA peer-to-peer technology.

1. INTRODUCTION

The rapid evolution of computers and networks is calling for the development of middleware platforms responsible for the management of dynamically reconfigurable federations of devices, where processes cooperate and compete for the use of shared resources. In this scenario one of the most challenging topics is concerned with the coordination of the activities performed by the federated components.

Generative communication, realized by means of the insertion and withdrawal of elements from a shared multiset, is the peculiar feature of a family of coordination languages, of which Linda [4] is the most prominent representative. Generative communication is based on the following principles: a sender communicates with a receiver through a shared data space (called *tuple space*, TS for short), where emitted messages are collected; the receiver can consume the message from TS; a message generated by a process has an independent existence in the tuple space until it is explicitly withdrawn by a receiver; in fact, after its insertion in TS, a message becomes equally accessible to all processes, but it is bound to none.

In the last decades, the shared dataspace approach has been successfully adopted in a huge variety of systems and applications, going from parallel computing to Web–based collaboration system. Recently, this communication mechanism has been adopted also by several proposals of coordination platforms (see, e.g., Sun Microsystems JavaSpaces [16] or the IBM T Spaces [17]) for the management of dynamically reconfigurable federations of devices, where processes cooperate and compete for the use of shared resources.

In this paper we investigate the problem of exploiting this coordination approach to the realm of peer-to-peer networks. Informally, *peer-to-peer* (P2P) networks are distributed systems based on the concept of resource sharing by direct exchange between *peer* nodes (i.e., nodes having the same role and responsibility). Exchanged resources include content, as in popular P2P file sharing applications [14, 7, 8], and storage capacity or CPU cycles, as, for example, in computational and storage grid systems [1, 12, 6].

Modern P2P networks and traditional distributed systems differ in several important aspects. First, P2P applications reach out to harness the outer edges of the Internet and consequently involve scales that were previously unimaginable. Second, P2P by definition, excludes any form of centralized structure; control is required to be completely decentralized, and peers cooperate together by exploiting the locality of their interactions. Finally, the environments in which P2P applications are deployed exhibit extreme dynamism in structure, content and load. The topology of the system typically changes rapidly due to nodes voluntarily coming and going or due to involuntary events such as crashes and partitions. The load in the system may also shift rapidly from one region to another, for example, as certain files become "hot" in a file sharing system. For these reasons, modern P2P systems are required to show a large degree of self-configuration and self-management properties.

Existing distributed implementations of Linda-like coordination infrastructures are based on a client-server architecture, where the dataspace metaphor is intended as a (centralized) repository service. The main proposal breaking the client-server bias is represented by the *transiently shared dataspace* metaphor introduced in Lime [10].

Lime [10] (Linda in a Mobile Environment) is a coordination middleware supporting both logical and physical mobility. The main entities in Lime are agents and hosts; each agent resides on a host, and may migrate to a different host exploiting logical mobility. Each host may communicate with other hosts, provided that they are connected. A host may phisically move, thus changing its relative connections. The whole set of host is partitioned in confederations of connected hosts.

In Lime, the agents coordinate by exchanging tuples as in Linda. Each agent has its own multiset of tuples which moves with the agent. The tuples owned by the agents which are currently in ex-

SAC 2003 Melbourne, Florida, USA

ecution in the same confederation of hosts are merged in a transiently shared dataspace, which represents the medium the agents exploit to coordinate themselves.

The Lime coordination model follows the peer-to-peer philosophy in the sense that it does not assume any client-server relationship. However, peer-to-peer networks like Gnutella [7] or Freenet [8] embody features which are different with respect to those considered in Lime.

In a peer-to-peer scenario, the involved entities (peers) are dynamic, i.e., they can frequently connect and disconnect, but they are not necessarily mobile. Mobility is not a peer-to-peer requirement in general. Moreover, a peer usually incorporates a level of autonomy which is different from that of an agent as in Lime. For instance, consider two peers in execution on the same device, each one having different connections to other peers. For this reason, it is not possible to reasonably partition the peers in distinct confederations as assumed in Lime.

In light of this observation, we can see that an adequate data-driven coordination model for peer-to-peer networks is still lacking. In order to cover this gap, we introduce a new coordination model named PeerSpaces, as well as a prototypical implementation of this coordination model based on the JXTA peer-to-peer technology [5].

The paper is structured as follows. Section 2 reports the rationale behind the guidelines we have followed in the design of PeerSpaces; Section 3 presents the formal definition of the coordination model; Section 4 discusses the implementation of the JXTA technology; and finally Section 5 draws some conclusive remarks.

2. TOWARDS PEERSPACES

In this section we present some guidelines we followed in the design of PeerSpaces. To be as general as possible, in our analysis we abstract away from the internal structure of the shared data. In this way, the proposed model can be instantiated to deal with any form of data, such as, e.g., Linda-like tuples, XML documents or JavaSpaces-like entries.

Following the peer-to-peer philosophy we do not assume any centralized storage for the shared data, but we assume that each datum resides on a specific peer. Any datum can be accessed by any other peer, provided that there is a direct connection, or a path of adjacent connections, between the peer reading the datum and the peer on which the datum resides.

The guidelines we report can be classified into two groups: those concerned with the production of data and those related to data retrieval.

2.1 Data production

According to the P2P philosophy, we expect that PeerSpaces supports both *context-aware* and *context-transparent* data production. Context aware applications are those which access both the system configuration context and the data context explicitly. For example, consider a file which is stored on a specific node of a peer-to-peer network and requires, in order to be accessed, the knowledge of its current location. On the other hand, context transparent applications can be developed without explicit knowledge of the current context. Consider, e.g., a retrieval operation in a file sharing application which does not take into account where the file is stored, but the only need is that the file is available in some reachable node.

In order to support context-awareness, it may be useful to locate a new datum on a specified peer. This feature could be used to model resources or information which are strictly connected to an entity of the system, hence they must disappear when the entity becomes disconnected. As an example, consider data which represent resources or services provided by a peer. In this case data production is context aware, in the sense that the programmer of the application explicitly indicates where the new datum should reside.

As far as context independence is concerned, we devise two possible ways for supporting it. The first one is represented by *generic data*. In the spirit of the generative communication approach, a datum belonging to this class has an existence which is completely independent from its producer. Hence, the coordination infrastructure may decide to locate this datum in any of the available storages, as well as to move the datum according to some system or application specific needs. As an example, consider load balancing or accessibility improvement. An interesting aspect related to this form of datum is the so called time- and space-uncoupling: data are accessed independently of both the time when they are produced and the peer which created them. This kind of datum can be useful, e.g., to achieve a form of disconnected master-worker interaction, in which the involved entities connect to the P2P system only when they need to produce or consume job requests, which are processed while disconnected.

The second form of context transparency we consider can be exploited in the case the datum represents an information, or a resource, that cannot be consumed; in other words, data that can never be explicitly removed from the repository. In these cases, the data can be transparently replicated without introducing any consistency problem which typically arises when a withdrawal operation requires to atomically remove all the replicas of a datum. Thus, in these cases, the coordination infrastructure can exploit a transparent replication of the datum in order to improve its availability to the peer community, as well as fault tolerance. We refer to this class as *replicable data*. A typical application which surely benefits from this kind of data is represented by file sharing systems.

2.2 Data retrieval

Concerning data retrieval mechanisms, we observe that it is useful to provide the peers the possibility to define their own visibility horizon of the system, instead of forcing a predefined scope, as it happens, e.g., in Linda and in Lime. In fact, in the first case the scope coincides with the whole dataspace, while in the second one the scope is formed by the union of the contents of the repositories of the currently federated hosts. This idea may be realized by equipping each retrieval operation with an extra parameter, specifying the actual scope to be used. A reasonable metric for scope definition is the Time To Live (TTL), corresponding to the relative distance between the peer hosting the datum and the peer performing the operation in the current topology. This feature adds both inter- and intra-peer flexibility. More precisely: two different peers may have different visions of the global dataspace and the same peer may change its own scope in different retrieval operations.

3. THE PEERSPACES COORDINATION MODEL

Following the approach of [3], we describe the PeerSpaces coordination model providing a formal way to represent the possible configurations of the system, plus a transition system indicating how these configurations may evolve according to the execution of the coordination operations.

Let $Data$, ranged over by d, e, $\ldots$, be the set of the data that can be exchanged by the peers. For each datum d, we denote with d_g (resp. d_r) a generic (resp. replicable) instance of datum d. As described in the previous section, by generic instance of a datum we consider a datum which can transparently migrate from one peer to another one, while by replicable instance we consider a datum

$$(1)\quad (p[\mathbf{write}(\mathbf{d},\mathtt{Here}).\mathbf{P}, DS] \oplus Ps,\ \bowtie,\ MD) \longrightarrow (p[\mathbf{P}, DS \oplus \mathbf{d}] \oplus Ps,\ \bowtie,\ MD)$$

$$(2)\quad (p[\mathbf{write}(\mathbf{d},\mathtt{Gen}).\mathbf{P}, DS] \oplus Ps,\ \bowtie,\ MD) \longrightarrow (p[\mathbf{P}, DS \oplus \mathbf{d_g}] \oplus Ps,\ \bowtie,\ MD)$$

$$(3)\quad (p[\mathbf{write}(\mathbf{d},\mathtt{Rep}).\mathbf{P}, DS] \oplus Ps,\ \bowtie,\ MD) \longrightarrow (p[\mathbf{P}, DS \oplus \mathbf{d_r}] \oplus Ps,\ \bowtie,\ MD)$$

$$(4)\quad (p[\mathbf{write}(\mathbf{d},\mathbf{p}').\mathbf{P}, DS] \oplus Ps,\ \bowtie,\ MD) \longrightarrow (p[\mathbf{P}, DS] \oplus Ps,\ \bowtie,\ MD \oplus \langle\mathbf{d}\rangle_{\mathtt{p}}^{\mathtt{p}'})$$

$$(5)\quad (p'[P, DS] \oplus Ps,\ \bowtie,\ MD \oplus \langle\mathbf{d}\rangle_{\mathtt{p}}^{\mathtt{p}'}) \longrightarrow (p'[P, DS \oplus \mathbf{d}] \oplus Ps,\ \bowtie,\ MD) \qquad \text{if } Route(p,p')$$

Table 1: Data production.

$$(6)\quad (p[\mathbf{read}(\mathbf{d},\mathbf{h}).\mathbf{P}, DS \oplus \mathbf{d}'] \oplus Ps,\ \bowtie,\ MD) \longrightarrow$$
$$(p[\mathbf{P}, DS \oplus \mathbf{d}'] \oplus Ps,\ \bowtie,\ MD) \qquad \text{if } d' \in \{d, d_g, d_r\}$$

$$(7)\quad (p[\mathbf{read}(\mathbf{d},\mathbf{h}).\mathbf{P}, DS] \oplus p'[P', DS' \oplus \mathbf{d}'] \oplus Ps,\ \bowtie,\ MD) \longrightarrow$$
$$(p[\mathbf{P}, DS] \oplus p'[P', DS' \oplus \mathbf{d}'] \oplus Ps,\ \bowtie,\ MD) \qquad \text{if } d' \in \{d, d_g, d_r\} \ \wedge\ p' \in Hor(p,h)$$

$$(8)\quad (p[\mathbf{take}(\mathbf{d},\mathbf{h}).\mathbf{P}, DS \oplus \mathbf{d}'] \oplus Ps,\ \bowtie,\ MD) \longrightarrow$$
$$(p[\mathbf{P}, DS] \oplus Ps,\ \bowtie,\ MD) \qquad \text{if } d' \in \{d, d_g\}$$

$$(9)\quad (p[\mathbf{take}(\mathbf{d},\mathbf{h}).\mathbf{P}, DS] \oplus p'[P', DS' \oplus \mathbf{d}'] \oplus Ps,\ \bowtie,\ MD) \longrightarrow$$
$$(p[\mathbf{P}, DS] \oplus p'[P', DS'] \oplus Ps,\ \bowtie,\ MD) \qquad \text{if } d' \in \{d, d_g\} \ \wedge\ p' \in Hor(p,h)$$

Table 2: Data retrieval.

that can be transparently replicated in different peers. Formally, let $Data_g = \{d_g \mid d \in Data\}$ and $Data_r = \{d_r \mid d \in Data\}$ denote the set of generic and replicable data, respectively.

A peer is a triple, denoted by $p[P, DS]$, where p is the peer identifier, P is the program the peer is executing, and DS is the dataspace local to the peer.

Formally, we denote by Pid the set of the peer identifiers ranged over by $p, q, \ldots$.

We consider four possible out operations: *local*, *remote*, *generic*, and *replicable* (informally described in the previous section). In order to distinguish among these four possibilities, we add to the output operation a parameter which is taken from the set $Target = Pid \cup \{\mathtt{Here}, \mathtt{Gen}, \mathtt{Rep}\}$. The peer identifiers in Pid can be used to denote the target of a remote, while the keywords Here, Gen, Rep denote local, generic, and replicable output, respectively. Let t, t', $\ldots$ range over $Target$.

As far as the data retrieval operations are concerned, we have to provide a way to denote the actual horizon to be used. One may consider different notions of horizon, e.g., all the peers that can be reached in a certain amount of time or within a close region of the network. As discussed in the previous section, in P2P networks a typical metric that is used to denote the proximity of peers is the so called time-to-live (TTL for short). Thus, we denote horizons using h, h', $\ldots$ which range over natural numbers. These natural numbers represent the maximal relative distance (expressed in terms of number of peer connections) between the peer performing the data retrieval operation, and the peer where the datum is retrieved.

We are now ready to introduce the grammar describing the peer programs. Let $Prog$, ranged over by $P, Q, \ldots$, be the set of terms defined by the following grammar:

$$P ::= \quad \mathbf{0} \ \mid\ \mu.P \ \mid\ K$$
$$\mu ::= \quad write(d,t) \ \mid\ take(d,h) \ \mid\ read(d,h)$$

where the term $\mathbf{0}$ denotes the empty program, $\mu.P$ is a program prefixed by a coordination operation, and K, which is taken from a generic set of program constants $Const$, is equipped with a definition $K = P$. Program constants can be used, e.g., for recursive definition of programs. For instance, $Prod = write(a, \mathtt{Here}).Prod$ is a program able to introduce an unbounded amount of instances of datum a in the local dataspace.

Formally, we define the set of peers as follows:

$$Peer \ = \ \{p[P, DS] \mid p \in Pid, P \in Prog, DS \in \mathcal{M}(Data \cup Data_g \cup Data_r)\}$$

where we use $\mathcal{M}(S)$ to denote the set of multisets over S. To lighten the notation, we will sometime omit the parenthesis in the case of singletons (i.e., we denote $\{a\}$ simply with a).

A network of peers (see the formal definition of Net below) consists of a triple composed of a set of peers, a connection relation which indicates whether two peers are currently connected, and a multiset of misplaced data, representing data which have been emitted towards a remote peer and have not reached their destina-

$$(10)\quad \frac{\bowtie\ \mapsto\ \bowtie'}{(Ps, \bowtie,\ MD)\ \longrightarrow\ (Ps, \bowtie',\ MD)}$$

$$(11)\quad \frac{(p[\mathbf{P}, DS] \oplus\ Ps,\ \bowtie,\ MD)\ \longrightarrow\ Ps'}{(p[\mathbf{K}, DS] \oplus\ Ps,\ \bowtie,\ MD)\ \longrightarrow\ Ps'} \qquad \text{if } K = P$$

$$(12)\quad \begin{array}{l}(p[P, \mathbf{DS} \oplus \mathbf{d_g}] \oplus p'[P', \mathbf{DS'}] \oplus Ps,\ \bowtie,\ MD) \longrightarrow \\ (p[P, \mathbf{DS}] \oplus p'[P', \mathbf{DS'} \oplus \mathbf{d_g}] \oplus Ps,\ \bowtie,\ MD)\end{array} \qquad \text{if } LoadBal(p, p', d_g)$$

$$(13)\quad \begin{array}{l}(p[P, \mathbf{DS} \oplus \mathbf{d_r}] \oplus p'[P', \mathbf{DS'}] \oplus Ps,\ \bowtie,\ MD) \longrightarrow \\ (p[P, \mathbf{DS} \oplus \mathbf{d_r}] \oplus p'[P', \mathbf{DS'} \oplus \mathbf{d_r}] \oplus Ps,\ \bowtie,\ MD)\end{array} \qquad \text{if } LoadBal(p, p', d_r)$$

Table 3: Constant and context rules.

tion yet. More precisely, a misplaced datum is represented by a triple, denoted with $\langle d \rangle_p^{p'}$, indicating a datum d emitted from the peer p towards p'.

The connection relation is denoted with $\bowtie$; by $p \bowtie p'$ we mean that the peer p' is in the set of the peers at distance 1 from p.

Formally, we define the set of the peer-to-peer networks Net, ranged over by $Ps, Ps', \ldots$, as follows:

$$\begin{array}{lcl} Net & = & \{(Ps, \bowtie, MD) \mid Ps \subseteq Peer, \bowtie \subseteq Pid \times Pid, \\ & & MD \in \mathcal{M}(Data \times Pid \times Pid)\} \end{array}$$

In order to avoid two peers to have the same peer identifier, we assume that, for each $Ps \in Net$, the following condition holds:

$$\begin{array}{l}(p[P, DS] \in Ps \wedge\ p'[P', DS'] \in Ps\ \wedge\ p = p')\ \Longrightarrow \\ (P = P'\ \wedge\ DS = DS')\end{array}$$

In the following we use $\oplus$ to denote set union as well as multiset union, the actual meaning is made clear by the context.

The connection topology of a peer-to-peer network may evolve, during the lifetime of the system, due to peer disconnections, peer mobility, failure of connections, etc. We model this dynamic aspect of the system by assuming the existence of a relation $\mapsto$: $(Pid \times Pid) \times (Pid \times Pid)$. More precisely, we use $\bowtie\ \mapsto\ \bowtie'$ to denote the fact that the connection relation $\bowtie$ evolves into $\bowtie'$.

We are now ready to introduce the operational semantics of our coordination model as the transition system $(Net, \longrightarrow)$ where $\longrightarrow$ is the least relation satisfying the axioms and rules reported in Tables 1, 2, and 3.

In Table 1 the semantics for data production is defined. Axioms (1–3) define the execution of a local, generic, and replicable output operation, respectively. In all the three cases, the effect is the introduction of the corresponding new datum in the local dataspace. We impose that the new datum is introduced in the local dataspace also in the case of generic and replicable data. This is useful in order to be sure that an instance of the datum is available (at least in the dataspace of the source peer) immediately after the execution of the output operation. The datum will be moved/replicated in other peer dataspaces according to the load balancing policy (see rules (12) and (13)).

As far as the remote output operation is concerned, different interpretations may be considered. For example, one could follow a *synchronous* approach, according to which a remote output operation can be executed only if the target peer is currently connected with the source peer. On the other hand, according to an *asynchronous* approach the operation may be divided in two distinct phases, the emission of the datum and the subsequent introduction of the datum inside the destination peer.

The main difference between the two interpretations is that, under the synchronous one, an output operation may block (in the case the target peer is not connected). Another difference is that, under the asynchronous interpretation, it is not possible to make any assumption on the time needed for a datum to reach its destination.

Due to the asynchronous nature of communication in peer-to-peer networks, we adopt the second approach. More precisely, we use the axioms (4) and (5), the former to indicate that a remote output operation simply produces a misplaced datum, and the latter to indicate that, subsequently, the misplaced datum may reach its destination. The message can be delivered if there exists a route from the sender to the receiver. To be as general as possible, in the definition of our coordination model we abstract away from the adopted routing protocol that will be defined by the implementors of the model. In the formal definition we simply assume the existence of a predicate $Route(p, p')$ which indicates the existence of a route from p to p'.

It is interesting to note that the routing mechanism, formally represented with the $Route(p, p')$ predicate, could be completely independent of the connection relation $\bowtie$ indicating the topology of the peer-to-peer network. A P2P network, indeed, is usually implemented over an underling infrastructure, e.g., the Internet. In this case it could be the case that two nodes are connected in the underling infrastructure (thus supporting a routing path between them) even if not explicitly connected in the P2P network.

The axioms concerning the data retrieval operations are reported in Table 2. The main novelties w.r.t. traditional Linda-like operations is the ability for a reader to specify the horizon of interest. We use the function $Hor(p, h)$ to denote the actual horizon of peer p under a time-to-live h.

A typical way for P2P protocols to visit the horizon is to initially broadcast queries to the peers at distance 1, which subsequently send queries to the peers at distance 2, and so on, until distance h. This is clearly a distributed protocol during which the topology of the network may change.

Different P2P infrastructures usually exploit different protocols to visit a specified horizon. For this reason, we do not consider any specific protocol, but we present an abstract illustrative definition of $Hor(p, h)$ which assumes that the topology of the network

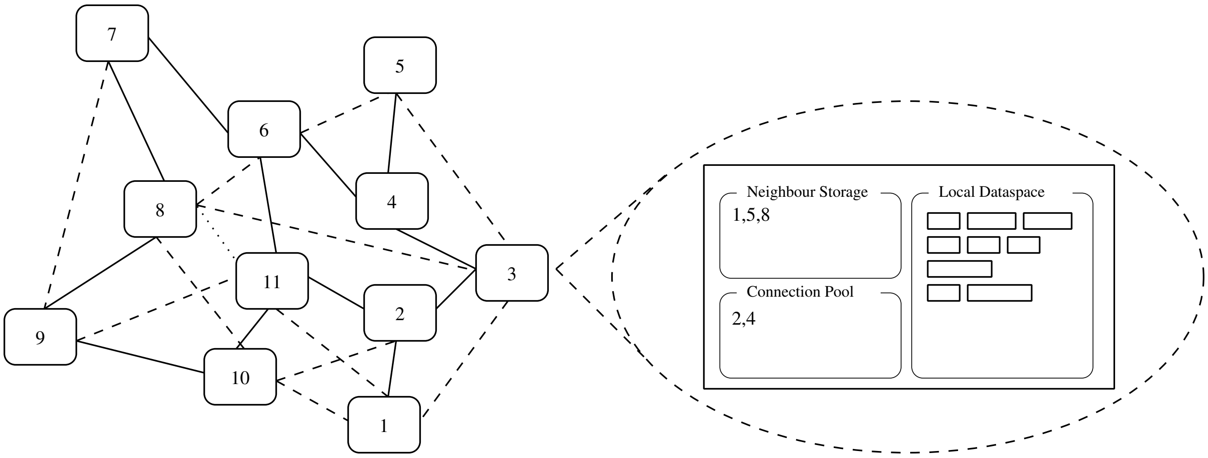

Figure 1: An example of JPS network

is $\bowtie$, and that it does not change during the visit of the horizon. Formally:

$$\begin{array}{rcl} Hor(p,h) & = & \{p' \mid \exists\, p_0,\dots,p_n\ s.t.\ p_0 = p \wedge p_n = p' \wedge \\ & & (p_{i-1} \bowtie p_i \text{ for } 1 \le i \le n) \wedge n \le h\} \end{array}$$

Axioms (6) and (7) define the semantics of the non-consuming data retrieval operations executed locally or remotely, respectively. Similarly, the axioms (8) and (9) define the semantics for the consuming operations. Following the design choices discussed in the previous section, in the case of non-consuming operations, also generic and replicable instances of the datum of interest can be read. On the other hand, in the case of consuming operations, replicable instances are not taken into account.

In Table 3, the remaining axioms and rules are reported. Axiom (10) indicates that the topology of the network may change according to the relation $\mapsto$. Rule (11) simply states that a program constant has the ability to execute the same operations as the corresponding definition. The last two axioms, (12) and (13), indicate the way generic and replicable data may move inside the network, according to some load balancing rules. To be as general as possible, we do not fix any load balancing rule but we assume defined a predicate $LoadBal(p, p', d)$ which indicates whether it is appropriate to introduce in the dataspace of the peer p' the datum d currently available in the dataspace of the peer p. Observe that, in the case of a generic datum, the original datum is removed; on the other hand, in the case of a replicable datum, a new instance is produced and the original one is kept.

4. A JXTA SERVICE BASED ON PEERSPACES

In order to show the feasibility of the PeerSpaces model, we have developed a prototype implementation of the specification illustrated in Section 3. This implementation, called JPS, is written in Java and is based on the JXTA project [5]. JXTA is an open-source project promoted by Sun Microsystems, whose aim is to establish a network programming platform for P2P systems by identifying a small set of basic facilities necessary to support P2P applications and providing them as building blocks for higher-level functions. JXTA is completely platform-independent; communication among peers is based on a set of XML-based protocols, and implementations in different languages exist.

The JXTA middleware is composed of three layers. At the bottom is the *JXTA core*, that deals with low-level functions such as peer establishment, peer discovery, communication management, routing and basic security facilities. The *JXTA services* are built on top of the core and deal with higher-level concepts, such as indexing, searching, and file sharing. These services, although useful by themselves, are used by *JXTA applications* to build high-level applications like chat, auction and persistent storage.

In order to join a JXTA network, a peer must implement a subset of the JXTA core protocols. First of all, the *discovery protocol* is used by peers in a network to discover each other and acquire information about the services they offer. The discovery protocol is based on the concept of *advertisements*, that are XML-based documents describing the main characteristics of a peer or a service. The *membership protocol* and the *access protocol* enable multiple peers implementing the same set of services to be grouped together to form a *peer group*, i.e., communities of peers having common interests. The *resolver protocol* is used to establish mono- and bi-directional communication channels, called pipes.

The benefits of basing our implementation on JXTA are several. For example, JXTA provides the possibility of using different transport layers for communication, including TCP/IP and HTTP, and is capable of handling firewall- and NAT-related problems. The discovery protocol can be used to establish complex PeerSpaces networks, by letting peers to discover each other. This spares our implementation from these low-level details. Furthermore, we also plan to exploit the complex security architecture that is being developed for JXTA.

Figure 1 shows a JPS network composed of a collection of peers connected together. In the current implementation of JPS each peer in the system maintains three main data structures, as illustrated in the magnified node on the right of the figure. The first data structure is the *local data space*, whose task is to maintain the multiset of tuples controlled by the local node. The *neighbour storage* maintains information about the set of JPS peers that are known to the local node. This set is dynamic, as new peers may be discovered and existing peers may crash. In the figure, peers included in the neighbour storage of a peer p are represented as dashed lines connecting p with those peers. Finally, the *connection pool* maintained at each peer contains the set of peers with which the local node is currently connected. The reason for having a connection pool distinct from the neighbour storage is that the neighbour storage may grow to become a very large set, and it would be unfeasible to maintain a connection with each of these neighbours. Together

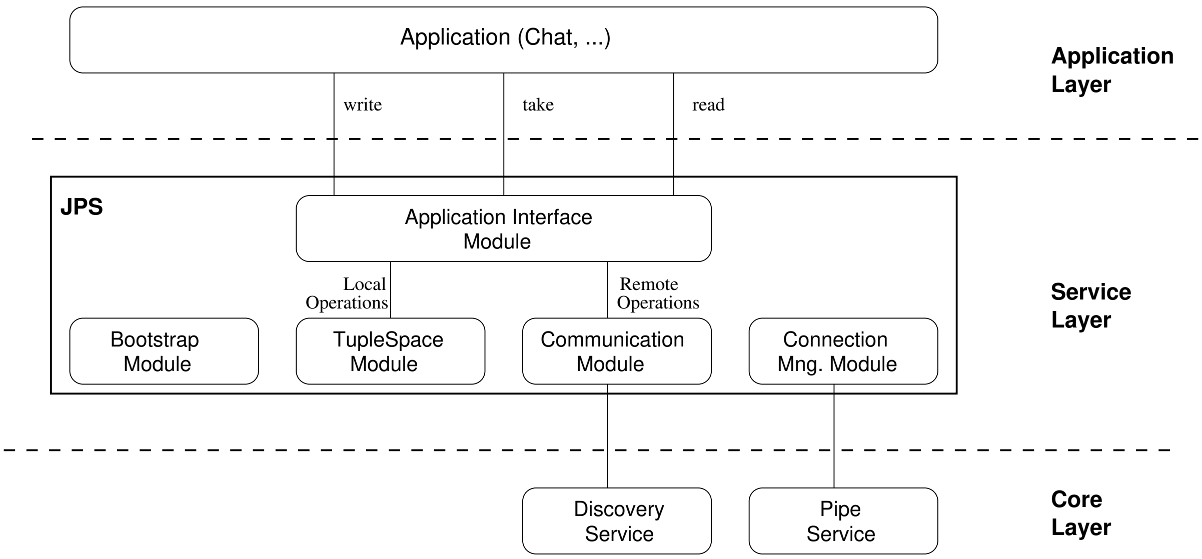

Figure 2: The architecture of a JPS node

with the information needed to establish a connection with the selected peers, the *connection pool* maintains additional information about peers, such as several statistics about the responsiveness of neighbours to previous distributed queries.

Figure 2 shows the architecture of a JPS peer and its relation with the other components of the JXTA platform. JPS has been designed as a JXTA service, and thus is conveniently located between the JXTA core layer and the JXTA application layer. JPS exploits the discovery and communication facilities provided by the discovery and pipe protocols included in JXTA. On the other hand, JPS may provide its tuple-space facility to the other services residing in its layer, or to JXTA-based applications residing in the application layer.

The architecture of JPS is subdivided in the following modules:

- The *Bootstrap Module* is responsible for bootstrapping the JPS service. First of all, the other modules are located. Then, a set of advertisements are created and subsequentially published through the discovery protocol, in order to advertise the presence of the JPS service to other peers. The JPS peer group is created and joined, in order to establish a community of nodes willing to participate in the JPS network.

- The *Application Interface Module* provides PeerSpaces interface to JXTA applications and services. Coordination operations are appropriately dispatched by this module to the local data space or to the communication module, depending on the target parameter of write operations and the horizon parameter of the read and write operations.

- The *TupleSpace Module* contains the local (non-distributed) data space where local tuples are stored. This module is based on an open-source project called LighTS [9]. LighTS has been developed as the local core for the implementation of the distributed Lime coordination infastructure [10].

- The *Communication Module* is responsible for communication between peers; in particular, it implements remote coordination operations like remote write and read, take operations with horizon parameter greater than 1. Details of how these operations are implemented are described in the following.

- The *Connection Management Module* is responsible for the management of the neighbor storage and the connection pool; it collects information about past connectivity and responsiveness of remote peers, and may occasionally decide to drop an active connection in favor of a connection with a more reliable and responsive peer.

Remote $write$ operations are implemented by creating a direct channel with the remote peer, and by performing a tuple exchange protocol with it. The current implementation of this protocol does not take into consideration problems related to communication failures occurring in the middle of the protocol, meaning that tuples may be occasionally lost. Future version of JPS will be extended with a recovery protocol to guarantee that temporary lost tuples are later recovered.

Remote $read$ and $take$ operations are implemented by performing Gnutella-like [7] broadcast searches with the specified time-to-live parameters. The reason for using broadcast-based searches is motivated by the fact that peers have control on their own dataspaces, and may select the tuples that have to be stored locally. This means that searches must reach as many peers as possible, in order to enlarge the horizon of peers as requested.

In order to guarantee fault-tolerance, searches are *leased*: this means that every peer reached by a search will store the request associated to the search until the lease for the request expires. If a local data space of a peer contains the desired tuple, or the tuple is inserted in the data space before the lease expiration, the peer will send a response to the local nodes containing a description of the peer. Peers waiting for the completion of $read$ and $take$ operations will periodically renew lease, until one or more responses are received. In the case of $read$ operation, the first response is delivered to the requesting peer and the operation completes. In the case of $take$ operations, the requesting peer executes a peer transfer protocol with the first responding peer; in case of failure (due to the tuple already taken by another peer, or to a communication problem), the next responding peers are contacted, until the transfer succeeds.

It is interesting to observe that during the above protocol, used to search the remote data to be retrieved, the topology of the P2P network may change according to peer connections and disconnections. This does not contrast with the formal semantics of the $read$ and $write$ operations (see Table 2) because the $Hor(p, h)$ function, used to indicate the current horizon to be considered, is not evaluated at the beginning of the execution of the protocol, but at the instant in which the required datum is retrieved.

5. CONCLUSION AND FUTURE WORK

In this paper we have introduced PeerSpaces, a data-driven coordination infrastructure suitable for P2P systems.

As future work, we plan we intend to extend the coordination model with other features such as a support for reactive programming or transaction operations. In the first case, we plan to adapt to our setting coordination primitives such as the $notify$ of JavaSpaces [16] or the $monitor$ of WCL [11]. In the second case, we will investigate either global operations such as *copy-collect* [13] or a more general support for transaction, such as in JavaSpaces, which permits to group a sequence of coordination operations in such a way that they should be atomically executed.

Another interesting extension concerns the possibility to associate an expiration time to the produced data. This could be particularly useful in order to support a garbage collection mechanism for replicable data which, according to the PeerSpaces model, cannot be explicitly removed.

6. REFERENCES

[1] D. Anderson. SETI@home. In A. Oram, editor, *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*, chapter 5. O'Reilly, Mar. 2001.

[2] N. Busi and G. Zavattaro. Some Thoughts on Transiently Shared Dataspaces. In *Proc. on the Workshop on Software Engineering and Mobility (at ICSE 2001)*, 2001.

[3] N. Busi, P. Ciancarini, R. Gorrieri, and G. Zavattaro. Models for Coordinating Agents: a Guided Tour. In *Coordination for Internet Agents: Models, Technologies, and Applications*, pages 6–24, Springer-Verlag, 2001.

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

[5] Project JXTA. http://www.jxta.org.

[6] J. Kubiatowicz et al. OceanStore: An Architecture for Global-Scale Persistent Storage. In *9th International Conference on Architectural support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.

[7] G. Kan. Gnutella. In A. Oram, editor, *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*, chapter 8. O'Reilly, Mar. 2001.

[8] A. Langley. Freenet. In A. Oram, editor, *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*, chapter 9. O'Reilly, Mar. 2001. March 2001.

[9] Project LighTS. http://lights.sourceforge.net/docs/info/intro.html.

[10] G.P. Picco, A. Murphy, and G.C. Roman. Lime: Linda Meets Mobility. In *Proc. 21th IEEE Int. Conf. on Software Engineering (ICSE)*, pages 368–377, 1999.

[11] A. Rowstron. WCL: A web co-ordination language. *World Wide Web Journal*, 1(3):167–179, 1998.

[12] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *18th Symposium on Operating Systems Principles*, Canada, November 2001.

[13] A. Rowstron and A. Wood. Solving the Linda multiple `rd` problem using the `copy-collect` primitive. *Science of Computer Programming*, 31(2-3):335–358, 1998.

[14] C. Shirky. Listening to Napster. In A. Oram, editor, *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*, chapter 2. O'Reilly, Mar. 2001.

[15] M. T. Valente, B. Carbunar and J. Vitek. Lime Revisited. Reverse Engineering an Agent Communication Model. In *Proc. of MA'01*, Lectures Notes in Computer Science. Springer-Verlag, Berlin, 2001.

[16] J. Waldo et al. Javaspace specification - 1.0. Technical report, Sun Microsystems, March 1998.

[17] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.