International Journal of Cooperative Information Systems

DATA-DRIVEN COORDINATION IN PEER-TO-PEER INFORMATION SYSTEMS

NADIA BUSI ALBERTO MONTRESOR GIANLUIGI ZAVATTARO
Department of Computer Scence, University of Bologna, Mura Anteo Zamboni 7
Bologna, 40127, ITALY. E-mail: {busi,montresor,zavattar}@cs.unibo.it

Peer-to-peer (P2P) has recently emerged as a promising model for supporting scalable networks composed of autonomous and spontaneously cooperating entities. The key concept in P2P is *decentralization*: the resources, the services, as well as the control are not in charge of specialized nodes in the network, but each node (called *peer* in this context) is directly involved in the management of all these aspects. Besides the advantages of decentralization (autonomy, adaptability, collaboration, and dinamicity just to mention few of them) one of the main drawbacks is the impossibility to predict the topology of the network, thus leaving at run-time any decision about the management of the interaction among the peers. For this reason, we consider useful to provide the developers of P2P applications with a high-level coordination language to be exploited to program the coordination among the peers. In this paper, we present PeerSpaces, a new data-driven coordination model suitable for P2P networks, and we describe JPS, an implementation of the PeerSpaces coordination model based on the JXTA peer-to-peer technology.

Keywords: Peer-to-peer information systems, coordination models and languages, shared data spaces, JXTA technology.

1. Introduction

The *Peer-to-peer* (P2P) paradigm is emerging as a promising infrastructure, alternative to the traditional client-server or multi-tier architectures, for supporting cooperative information systems (CIS), i.e. distributed collections of heterogeneous information systems that are able to cooperate by coordinating their actions. Apart from the unquestioned success of P2P file sharing applications like Napster[38] and its successors[20,23], that have rapidly defined a new widely exploited way of exchanging information, the P2P approach is gaining an increasing success in fields traditional for CISs, such as distributed databases[1] and multi-agent systems[32].

Informally, 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). Different form of resources are encompassed: computing power[3], storage capacity[35,19], content[38,20,23] and even human presence[22]. This paradigm differs from the traditional client-server model where only a small number of servers are allowed to satisfy service requests from a potentially large number of clients, thus promoting a clear subdivision of roles in the system.

The concept of P2P is not new: the original Usenet[17], for example, was built over a peer-to-peer dialup network called UUCP. Similarly, in the beginning, all nodes of the Internet were peers that cooperated in routing packets among themselves. As these system grew, however, the original P2P design has been lost. Currently, the Internet has evolved into a more hierarchical, client-server structure, in which a relatively small number of servers provides services to millions of client machines that simply act as service consumers.

The recent renewed interest in P2P can be attributed mainly to the enormous success of applications like Napster[38], and more recently Gnutella[20] and Freenet[23]. These systems enable end-users to establish a file-sharing network for exchanging digital documents including music, movies and software, and have shown to be capable to exploit what has been called "the dark matter" of the Internet, i.e. the huge number of resources (storage, CPU cycles, content) available at the edges of the network. Furthermore, the P2P paradigm has received the attention of both industry and academia. Some big industrial efforts include the P2P Working Group[30], led by many industrial partners such as Intel, HP and Sony; and JXTA[18], an open source effort led by Sun. In the academia, several projects focused on P2P overlay networks have been started[11,31,36,43].

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.

Clearly, self-configuration and self-management have also a significant drawback: each peer must be programmed in such a way that it is able to autonomously manage, at run-time, its interactions and interconnections with its environment. From the point of view of the developer of a P2P application, this means that s/he is responsible for programming the interaction among the peers. The current approach, supported e.g. by P2P platforms such as JXTA[18], is to program these aspects exploiting protocols provided by the underlying infrastructure. For example, JXTA comprises protocols that allow a peer to publish the services it is ready to provide, to discover the peers available in the environment, to open connections with some of these available peers, to close connections that are no longer useful, etc.

Programming the interaction among peers using these low-level protocols is not an easy task due to the dynamicity of the network: its topology cannot be predicted at design time, and it is also difficult to be monitored at run-time. Indeed, new peers can frequently enter and old peers can freely exit the network without any explicit advertisement. Our proposal for alleviating this difficult task is to provide the P2P application developer with a higher-level language for programming these aspects.

We design such a high level language following the typical approach suggested by *coordination models and languages*[16]. According to this approach, the specification of the internal behaviour of the components in a distributed computation or application should be distinct and separated from the specification of their interaction and dependencies. A coordination model defines the medium that the components exploit in order to coordinate, as well as the rules governing the interaction between the components and the coordination medium. On the other hand, a coordination language is a linguistic embodiment of a coordination model, i.e. it is the language that can be used to program the interaction among components according to the coordination model.

As a starting point in the design of this language we consider the most prominent coordination language, i.e. LINDA[15], and its underlying *data-driven* coordination model. LINDA is based on the so-called *generative communication* which is realized by means of the insertion, reading, and withdrawal of elements from a shared bag of data. More precisely, generative communication is based on the following principles. A sender communicates through a shared data space (called *tuple space*, TS for short), where emitted messages are collected. The receiver can read or consume the message from the TS, indicating with a template the kind of message it is interested in. In this way the access to the data is associative, in the sense that data are retrieved according to their contents and not depending on their location or their producer. This form of communication is referred to as generative because a message generated by a process has an independent existence in the TS 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 data-driven approach has been successfully adopted in a huge variety of systems and applications, spanning from parallel computing[9] to Web–based agent systems[10]. Recently, this approach has been adopted also by relevant coordination platforms for distributed Java programming; see, e.g., the Sun Microsystems JAVASPACES[41] or the IBM T–Spaces[42].

When trying to adapt the data-driven coordination model to P2P networks, the first problem which is encountered deals with *decentralization*. Decentralization is the key concept in P2P, whilst it contrasts with the TS that is a logically centralized entity. By logically centralized we mean that the TS has its own identity and its own state independently of the location where the data are actually stored.

In the literature we can find several extensions of the Linda coordination model which support some form of decentralization.* A significant family of coordination models (see, e.g., JavaSpaces[41], KLAIM[14], TuCSoN[28], and LogOp[39]) introduces the possibility to deal with several data spaces located in different locations. According to this approach, in order for a component to access one data space, the knowledge of the corresponding location is required. Moreover, some of these proposals (see, e.g., the extension of the TuCSoN model[33] or LogOp[39]) allow also to access to more spaces at a time.

The form of decentralization embodied in these proposals essentially corresponds to the possibility to support the cohexistence and interaction among different logically centralized data spaces that are possibly located at different sites. A more recent family of decentralized Linda-like coordination models[29,25,26] supports a finer grained form of decentralization. Tuples are not associated to a specific space, but each tuple has its own location. Tuple spaces are then obtained as overlay structures that group together tuples possibly located at different sites. For example, in Lime[29] tuples are associated to agents, which are software components running on hosts. A group of connected hosts form a confederation. All the agents running in the same confederation share the same data space (called Transiently Shared Dataspace); this data space is obtained at run-time grouping together the tuples stored in the agents currently running on the confederated hosts. Logical mobility is supported in the sense that agents can move from one host to another. Physical mobility, on the other hand, is supported in the sense that hosts can move by joining and leaving confederations. When a host/agent leaves (resp. joins) a confederation, the tuples stored in that host/agent leave (resp. join) the corresponding data space.

In this second approach to decentralization, the P2P philosophy is adopted in the sense that the components of the system are at the same time both clients of the data space (e.g. they produce and retrieve data) as well as servers (because they behave also as data storages). However, P2P networks like Gnutella[20] or Freenet[23] embody features that are different.

In a P2P 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 the other peers. For this reason, it is not possible to reasonably partition the peers in distinct confederations as done in Lime, where the agents running on the same host inherit the same connections from the host itself.

In light of these observations, we can see that an adequate data-driven coordination model for P2P networks is still lacking. In order to cover this gap, we introduce

*See the Related Work section for a more detailed description of related coordination models and languages.

a new coordination model named PeerSpaces. The main novelties of PeerSpaces can be divided in two classes: those concerning data production and those related to data retrieval.

As far as data production is concerned, we permit to associate the produced data to one of three possible attributes:

- the *located* attribute characterizes those data that are strictly associated to a peer: these data will be stored in the local data space of the referring peer until they are explicitly withdrawn;
- the *generic* attribute indicates those data that, on the other hand, can be transparently moved from one peer to another one: this is useful, e.g., in order to support load-balancing mechanisms aiming at locating the data close to those regions of the network in which they are more frequently accessed;
- the *replicable* attribute is associated to those data that can be transparently replicated, thus stored in the local data spaces of several peers: this is useful in order to increase the availability of these data inside the network.

As far as data retrieval operations is concerned, we observe that it is not reasonable to search for a datum in the whole network of peers. Indeed, the size of the network could be very huge and its topology could significantly change during this time consuming retrieval operation. For this reason, we have extended data retrieval operations with an extra parameter which indicates the *horizon* of interest; by horizon we mean a subpart of the peers in the network to be taken into account during the data retrieval operation.

This new extra parameter supports the possibility to explicitly choose among different views of the overall decentralized data space; more precisely, it supports intra- and inter-peer flexible views. By intra-peer flexible views we mean the possibility for a peer to consider different horizons; by inter-peer flexibility we mean the possibility for two different peers to exploit different views.

In order to study the feasibility of our PeerSpaces coordination model we have developed `JPS`, a coordination service for the JXTA[18] infrastructure, which is based on PeerSpaces.

The remaining of 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 presents the implementation of `JPS`; Section 5 discusses the related literature; and finally Section 6 draws some conclusive remarks.

2. Towards PeerSpaces

In this section we present the guidelines we have followed in the design of PeerSpaces. In order to clarify our design choices we will use a running example inspired by the so-called *slashdot effect*[2]. This effect typically occurs in web-based information systems exploiting high volume news web sites (see, e.g., `slashdot.org`, `linuxtoday.org`, and `freashmeat.net`) in order to announce the availability of new resources on the Internet. The slashdot effect corresponds to a spontaneous high rate of requests sent to the server providing the new resource just after its announcement. Due to this huge amount of requests, the availability of the resource is no more guaranteed because the server could be overwhelmed by the requests.

The slashdot effect is a consequence of two forms of centralization: the high volume news web site and the server providing the resource. An equivalent information system based on the P2P philosophy could avoid these two forms of centralization, thus eliminating the slashdot effect and its negative consequences. The dissemination of the announcement inside the network, e.g, could be alternatively obtained as follows: the server communicates to its neigbours the availability of the new resource, each of these neighbours forward the advertisement to its relative neighbours, and so on. The availability of the resource, on the other hand, could be improved simply be relocating (or even replicating) the resource inside the network of peers.

2.1. Shared Data and Coordination Information

As described in the Introduction, we use shared data to represent the information needed for supporting the coordination among the peers. Differently from other coordination models, in PeerSpaces we do not fix a specific predefined form for these data; we assume that the structure of the data is defined during an implementation of the PeerSpaces model. For instance, an implementor could choose among LINDA-like tuples, XML documents, Java objects, etc.

The choice of the structure to give to the data depends on the class of applications for which an implementation is devised. There are several important aspects that should be taken into accout; we identify three of them: (i) the structure of the data should be flexible enough to represent all the kind of information needed during the coordination of the peers, (ii) the format of data should be accessible and readable from all the interested peers, (iii) the size of the data should be light enough to permit an easy and fast exchange of the data among the peers.

As far as (i) is concerned, information such as the services that a peer can provide to the environment, or the description of the resources available in a particular region of the P2P network, should be representable. For instance, in the Introduction we have already discussed the flexibility of the tuple-based approach that have been successfully exploited in very different scenarios spanning from parallel computing to web-based agent systems.

Interoperability issues are concerned with (ii). P2P systems usually support heterogenous devices: all these different kinds of devices should be able to access and to understand the content of the shared data. Data modeled via Java objects or XML documents could tackle this problem.

As far as (iii) is concerned, it is worth to point out that coordination media are usually exploited for sharing only the coordination information: these are typically light information. In the case two entities are interested in exchanging huge amount of data (such as a digital video or part of a database) they can open a direct connection without influencing the data space. In this way, we avoid the introduction of huge-sized data inside the data space. This approach is typically adopted also in P2P applications (see e.g. Napster[38]) where the exchange of huge files is realized out-of-band, i.e. opening a direct and independent connection between the two end-points of the communication.

2.2. Data production

As described in the Introduction we do not assume any centralized storage for the TS, 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.

According to the P2P philosophy, we expect that PeerSpaces supports both *context-aware* and *context-transparent* data production. Context aware applications are those that access both the system configuration context and the data context explicitly. For example, consider a file stored on a specific node of a P2P network and which 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 P2P information system that does not take into account where the information is stored, but the only need is that the information 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 that are strictly connected to an entity of the system, hence they must disappear when the entity becomes disconnected. As an example, consider data representing resources/services provided by a peer: if these data are stored inside the peer itself, it is ensured that the data are co-located and will move with the resources/services they describe. To achieve this, it is necessary to support context-awareness, in the sense that the programmer of the application explicitly indicates where the new datum should reside. These kind of data are named *located data* in PeerSpaces.

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[16]:

data are accessed independently of both the time when they are produced and the peer that created them. This kind of datum can be useful, e.g., to achieve a form of disconnected master-worker interaction. Masters periodically produce tasks that should be executed by one of the available workers. To support this form of interaction in our context, masters can connect to the P2P network when they need to ask for the execution of a task. The request can be expressed in terms of a generic datum. When a worker is ready to execute a task, it can connect to the network and ask for the availability of some generic request: after the reception of a request, the worker can execute the task disconnecting from the network.

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 explicitly consumed. For instance, an event such as the final result of a football match is an entity that has a starting time, but it has no termination time. Moreover, this kind of information does not change during its lifetime. Due to these features, these data can be freely replicated without any problems related to consistency. Other kinds of data, e.g. the description of an available resource, cannot be freely replicated because when the resource is removed from the system, also all the data describing it should be withdrawn. In order to take advantage of the possibility to freely replicate these data, we intend to support in our coordination model a specific class of data that we name *replicable data*. The advantage of explicitating this kind of data is that the coordination infrastructure can exploit a transparent replication of these data in order to improve their availability to the peer community.

As described above, due to their nature the replicable data cannot be explicitly withdrawn. In order to support the garbage collection of replicas no longer of interest (e.g. because outdated), our coordination model PeerSpaces permits to associate to each datum an expiration time. After expiration, the coordination infrastructure can freely decide to remove the expired data in order, e.g., to free storage resources.

As we will describe in the next section, we do not fix any pre-defined model for specifying the expiration time in the PeerSpaces coordination model. Indeed, we think that each implementation of the model could exploit a different approach. For instance, a typical approach introduced by JavaSpaces[41] is to consider data as leased resources: the producer of a datum explicitly indicates its time of duration, after which the datum can be removed. Other models could, on the other hand, take specific events under consideration: for instance, remove a datum only when a more updated one is produced.

Now we discuss a possible exploitation of the different kinds of data in our running example. The announcement of the availability of a new resource provided by a peer (say the *provider*), could be encoded in terms of a *replicable* datum containing the description of the resource and the name of the provider. This datum will be transparently replicated and each replica will be transparently moved inside the network in order to increase the availability of the announcement.

When a peer (say the *user*) accesses the advertisement, it becomes aware of the new resource and of the name of the provider. The client can send a request of usage of the resource simply by producing a datum which is *located* in the local data space of the provider. The server may produce an answer as result of the execution of the request: we consider two different cases. If the answer is of interest only for the user, it can be encoded in terms of a datum which is *located* in the local data space of the user only; on the other hand, if the answer is of interest for all the network, it can be encoded in terms of a *generic* datum that will be transparently moved inside the network (in order to become available to the maximal number of peers) until it is explicitly withdrawn.

2.3. 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 data space, 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.

The possibility to explicitly associate different horizons of interest to the data retrieval operations adds two levels of flexibility. The first level, that we call *intra-peer*, permits to the same peer to take under consideration different groups of peers during different phases of its interaction with the environment. For example, if a coordination information is needed within a short time, only directly connected peers can be introduced in the horizon of interest (TTL $= 1$). On the other hand, a larger group of peers (TTL > 1) can be considered while retrieving information that describes remote regions of the network. Similarly, two peers can consider two different horizons at the same time: we refer to this possibility as *inter-peer* flexibility.

In our running example, we can exploit two different kinds of data retrieval operations: the former performed by the user to read the advertisement, the latter executed by the provider in order to consume the request. In the first case, it could be convenient for a client to perform a read operation considering a large horizon, i.e., an high TTL number; in this way, the possibility to find the advertisement is increased. In the second case, the provider performs a data consumption operation in its local data space; this corresponds to a destructive read operation performed with TTL equal to 0.

3. The PeerSpaces Coordination Model

Following the approach of Busi et al.[7], 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.

In our opinion, the definition of a formal framework represents a necessary step in the development of a coordination model. The definition of the formal semantics permits to specify in a rigorous way what is the intended behaviour of a system and to clarify some subtleties that can arise in the model. This formal specification turns out to be very useful both for the user – which can figure out what are all the possible outcomes of a system – and for the implementor – which can know the constraints that the implementation must satisfy. Although this aspect is important also in the field of sequential programming, the need for a precise specification of the semantics becomes critical in the field of concurrency, where the nondeterminism provoked by the variety of possible interplays between the components of the system could underhand lead to unexpected or unwanted behaviours. As witnessed by [7] and by the literature quoted therein, a rigorous specification of the semantics turns out to be very useful to compare and to discriminate the expressive power of alternative design choices of various coordination primitives. A further advantage of the formal semantics could be to open the possibility to adapt verification techniques developed for process algebra (such as, e.g., bisimulation and model checking) to the analysis of the behaviour of coordination models.

In the definition of the formal semantics of a model, some choices have to be performed, regarding, e.g., the degree of concreteness in the representation of some features of the system. As far as the modeling of data is concerned, in [7] the authors were interested in modeling and comparing different coordination languages, with particular emphasis on the coordination primitives; to this aim, in that work we completely abstracted away from the structure of data. Although we are not interested in committing to a particular structure of data, in the present work we are interested in modeling the possibility to embody some relevant information (such as, e.g., peer identifiers) inside data. To this aim, we will adopt the modeling of communication described below.

Let $Data$, ranged over by d, e, $\ldots$, be the set of the data that can be exchanged by the peers. As we abstract away from the specific structure of the data, to lighten the formalization we suppose that the set $Data$ contains representations for both the elements contained in the data space and the templates specifying the kind of data required by a data retrieval operation. For example, if we consider the LINDA data model, a datum consists in a sequence of values, whereas a template is a sequence of both values and variables. To this aim, we provide our model with two abstract operations on data, which will be instantiated when a particular data representation is chosen: the predicate $Match(d, d')$ checks if the template of a data retrieval operation matches a datum, whereas the operation $P\{d'/d\}$ updates the continuation of a data retrieval operation with the new information obtained from the datum that has been withdrawn (or read) from the data space. For example, in the LINDA data model $P\{d'/d\}$ substitutes each variable appearing in the template d with the corresponding value in the datum d', in the continuation P.

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 that can transparently migrate from one peer to another one, while by replicable instance we consider a datum 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 data space local to the peer. Formally, we denote by Pid the set of the peer identifiers ranged over by p, q, ...

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 output, while the keywords `Here`, `Gen`, `Rep` denote local, generic, and replicable output, respectively. Let t, t', ... 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', ... ranging 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, ..., be the set of terms defined by the following grammar:

$$\begin{array}{rcl} P & ::= & \mathbf{0} \mid \mu.P \mid P|P \mid K \\ \mu & ::= & write(d,t) \mid take(d,h) \mid read(d,h) \end{array}$$

where the term $\mathbf{0}$ denotes the empty program, $\mu.P$ is a program prefixed by a coordination operation, $P|P$ is the parallel composition of two programs, 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 data space.

The order in which parallel programs are composed has no importance; moreover, the empty program $\mathbf{0}$ is not relevant when composed in parallel with other programs. To deal with this formally, we reason up to rearrangement of the order of composition, as well as garbage collection of the program $\mathbf{0}$. Let $\equiv$ be the least congruence on programs satisfying the following axioms:

$$P|Q \equiv Q|P \qquad (P|Q)|R \equiv P|(Q|R) \qquad P|\mathbf{0} \equiv P$$

As we will discuss in the following, we do not distinguish two programs P and Q when $P \equiv Q$. This is guaranteed by rule (15) of Table 3 ensuring that two congruent programs give rise to the same computation steps.

We are now ready to formally define the set of peers as follows:

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

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 destination 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', ..., as follows:

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

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

$$\begin{aligned} & (p[P, DS] \in Ps \wedge p'[P', DS'] \in Ps \;\wedge\; p = p') \implies \\ & (P = P' \;\wedge\; DS = DS') \end{aligned}$$

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

Example 1. *As an example, consider a network composed of a resource provider and three users.*

The provider announces the availability of a new resource by producing a replicable datum; then, it waits for the arrival of service requests in its local data space. Two kinds of service are provided. The first service produces a specific answer, that is of interest only for the user who performed the query. Hence, the datum containing the answer is sent to the user data space. On the other hand, the second service *produces a general answer that could be interesting also for other users; in this case, a generic datum is produced.*

After reading the announce of the availability of a new resource, a user produces a service request in the provider data space. If the answer to the request is of general interest, then the user looks for the answer in a wide horizon; otherwise, the user will wait for the answer in its local data space.

The system is defined by the following term:

$$(p1[Advertiser, \emptyset] \oplus p2[UserSpec, \emptyset] \oplus p3[UserGen, \emptyset] \oplus p4[UserGen, \emptyset],\ \bowtie, \emptyset)$$

where

$$\begin{array}{lcl} Advertiser & = & write(newRes@p1, \mathtt{Rep}).(SpecProvider|GenProvider) \\ SpecProvider & = & take(specReqFrom(x), 0).write(specAnsw, x).SpecProvider \\ GenProvider & = & take(genReq, 0).write(genAnsw, \mathtt{Gen}).GenProvider \\ UserSpec & = & read(newRes@(x), 10).write(specReqFromp2, x). \\ & & take(specAnsw, 0) \\ UserGen & = & read(newRes@(x), 10).write(genReq, x). \\ & & read(genAnsw, 10) \end{array}$$

The templates used in this example are $newRes@(x)$ *and* $specReqFrom(x)$*, whereas examples of data are* $newRes@p1$*,* $specReqFromp2$*,* $specAnsw$ *and so on. In this example, we assume that* $Match(newRes@(x), newRes@p1)$ *holds, as well as* $Match(newRes@(x), newRes@p3)$*; on the othe hand, we assume that* $Match(newRes@(x), specReqFromp1)$ *does not hold. When the template of a data retrieval operation matches a datum in the data space, each occurrence of the variable between parenthesis in the template is replaced by the corresponding part of the datum in the continuation of the process performing the data retrieval. For example, when process* $UserSpec$ *reads the datum* $newRes@p1$*, its continuation becomes* $write(specReqFromp2, x).take(specAnsw, 0)\{newRes@p1/newRes@(x)\} = write(specReqFromp2, p1).take(specAnsw, 0)$.

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. The dynamic aspects of the system are modeled by the 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 consider a small–step evolution of the connection relation, namely, a single pair of peer identifiers can be added or removed from the connectivity relation in a single evolution step. The relation $\mapsto$ is defined by the following axioms:

$$\begin{array}{ll} \bowtie \mapsto \bowtie \cup \{(p, p')\} & \text{if } (p, p') \notin \bowtie \\ \bowtie \mapsto \bowtie \setminus \{(p, p')\} & \text{if } (p, p') \in \bowtie \end{array}$$

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.

$$
\begin{array}{ll}
(1) & (p[\mathbf{write}(\mathbf{d},\mathtt{Here}).\mathbf{P}|Q,DS]\oplus Ps,\ \bowtie,\ MD) \longrightarrow \\
 & (p[\mathbf{P}|Q,DS\oplus\mathbf{d}]\oplus Ps,\ \bowtie,\ MD) \\[1em]
(2) & (p[\mathbf{write}(\mathbf{d},\mathtt{Gen}).\mathbf{P}|Q,DS]\oplus Ps,\ \bowtie,\ MD) \longrightarrow \\
 & (p[\mathbf{P}|Q,DS\oplus\mathbf{d_g}]\oplus Ps,\ \bowtie,\ MD) \\[1em]
(3) & (p[\mathbf{write}(\mathbf{d},\mathtt{Rep}).\mathbf{P}|Q,DS]\oplus Ps,\ \bowtie,\ MD) \longrightarrow \\
 & (p[\mathbf{P}|Q,DS\oplus\mathbf{d_r}]\oplus Ps,\ \bowtie,\ MD) \\[1em]
(4) & (p[\mathbf{write}(\mathbf{d},\mathbf{p'}).\mathbf{P}|Q,DS]\oplus Ps,\ \bowtie,\ MD) \longrightarrow \\
 & (p[\mathbf{P}|Q,DS]\oplus Ps,\ \bowtie,\ MD\oplus\langle\mathbf{d}\rangle_{\mathbf{p}}^{\mathbf{p'}}) \\[1em]
(5) & \text{if } p\bowtie p'' \\
 & (Ps,\ \bowtie,\ MD\oplus\langle\mathbf{d}\rangle_{\mathbf{p}}^{\mathbf{p'}}) \longrightarrow \\
 & (Ps,\ \bowtie,\ MD\oplus\langle\mathbf{d}\rangle_{\mathbf{p''}}^{\mathbf{p'}}) \\
(5') & \text{if } p\bowtie p' \\
 & (p'[P,DS]\oplus Ps,\ \bowtie,\ MD\oplus\langle\mathbf{d}\rangle_{\mathbf{p}}^{\mathbf{p'}}) \longrightarrow \\
 & (p'[P,DS\oplus\mathbf{d}]\oplus Ps,\ \bowtie,\ MD)
\end{array}
$$

Table 1: Data production.

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 data space. We impose that the new datum is introduced in the local data space 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 data space of the source peer) immediately after the execution of the output operation. The datum will be moved/replicated in other peer data spaces 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)–(5'), the former to indicate that a remote output operation simply produces a misplaced datum, and the latter two to indicate that, subsequently, the misplaced datum may reach its destination. By axiom (5) the misplaced datum is moved from one peer to another one, provided that the two peers are connected. When the misplaced datum reaches a peer that is connected to its destination peer, by axiom (5') the datum is finally moved to the local dataspace of the destination peer.

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.

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.

The definition of remote data retrieval operations makes use of the auxiliary function $\Delta(p, p')$. This function returns the minimum distance between peers p and p' wrt the connectivity relation, i.e., the number of arcs occurring in the shortest path connecting the two peers. Formally, we define

$$\Delta(p, p') = min\{h \mid \exists\, p_0, \ldots, p_h \ s.t.\ p_0 = p \wedge p_h = p' \wedge (p_{i-1} \bowtie p_i \text{ for } 1 \leq i \leq h)\}$$

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. If a datum, matching the template of the data retrieval operation, is available within the specified horizon, then the continuation of the operation is updated with the new information obtained from the withdrawn (or read) datum.

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 contextual, constant and garbage collection axioms and rules are reported. Axiom (10) indicates that the topology of the network may change according to the modification indicated by the relation $\mapsto$. Rule (11) simply states that a program constant has the ability to execute the same operations as the corresponding definition. Axioms (12) and (13) indicate the way generic and replicable data may move inside the network, to satisfy some load balancing needs. Observe that, in the case of a generic datum, the original datum is removed; on the

(6) if $Match(d, d')$
$(p[\mathbf{read(d,h)}.\mathbf{P}|Q, DS \oplus \mathbf{d'}] \oplus Ps,\ \bowtie,\ MD) \longrightarrow$
$(p[\mathbf{P}\{\mathbf{d'}/\mathbf{d}\}|Q, DS \oplus \mathbf{d'}] \oplus Ps,\ \bowtie,\ MD)$

(7) if $Match(d, d')\ \wedge\ \Delta(p, p') \leq h$
$(p[\mathbf{read(d,h)}.\mathbf{P}|Q, DS] \oplus p'[P', DS' \oplus \mathbf{d'}] \oplus Ps,\ \bowtie,\ MD) \longrightarrow$
$(p[\mathbf{P}\{\mathbf{d'}/\mathbf{d}\}|Q, DS] \oplus p'[P', DS' \oplus \mathbf{d'}] \oplus Ps,\ \bowtie,\ MD)$

(8) if $Match(d, d')\ \wedge\ d' \notin Data_r$
$(p[\mathbf{take(d,h)}.\mathbf{P}|Q, DS \oplus \mathbf{d'}] \oplus Ps,\ \bowtie,\ MD) \longrightarrow$
$(p[\mathbf{P}\{\mathbf{d'}/\mathbf{d}\}|Q, DS] \oplus Ps,\ \bowtie,\ MD)$

(9) if $d' \in Match(d, d')\ \wedge\ d' \notin Data_r\ \wedge\ \Delta(p, p') \leq h$
$(p[\mathbf{take(d,h)}.\mathbf{P}|Q, DS] \oplus p'[P', DS' \oplus \mathbf{d'}] \oplus Ps,\ \bowtie,\ MD) \longrightarrow$
$(p[\mathbf{P}\{\mathbf{d'}/\mathbf{d}\}|Q, DS] \oplus p'[P', DS'] \oplus Ps,\ \bowtie,\ MD)$

Table 2: Data retrieval.

other hand, in the case of a replicable datum, a new instance is produced and the original one is kept.

Axiom (14) performs garbage collection of outdated data. A garbage collection mechanism is necessary to avoid that replicable data – that cannot be withdrawn – saturate the dataspaces. The commitment to a specific modeling of expired data and garbage collection is outside the scope of this work; for this reason, we do not formally specify when a datum can be collected but we simply describe the effect of the consumption of one datum (indeed, axiom (14) simply defines a transparent withdrawal of one of the stored data). A possible approach for defining a collection policy, presented in [4,6], consists in equipping data with a lifetime; the garbage collector will remove only the data whose lifetime has expired.

Finally, rule (15) has been introduced in order to permit to reason up to the structual congruence $\equiv$, i.e., to reason up to rearrangement of parallel composed programs as well as garbage collection of terminated programs.

Example 2. *Consider the network introduced in the previous example, and composed of a resource provider and three users:*

$$(p1[Advertiser, \emptyset] \oplus p2[UserSpec, \emptyset] \oplus p3[UserGen, \emptyset] \oplus p4[UserGen, \emptyset], \bowtie, \emptyset)$$

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

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

$$(12)\quad (p[P, \mathbf{DS} \oplus \mathbf{d_g}] \oplus p'[P', \mathbf{DS'}] \oplus Ps,\ \bowtie,\ MD) \longrightarrow (p[P, \mathbf{DS}] \oplus p'[P', \mathbf{DS'} \oplus \mathbf{d_g}] \oplus Ps,\ \bowtie,\ MD)$$

$$(13)\quad (p[P, \mathbf{DS} \oplus \mathbf{d_r}] \oplus p'[P', \mathbf{DS'}] \oplus Ps,\ \bowtie,\ MD) \longrightarrow (p[P, \mathbf{DS} \oplus \mathbf{d_r}] \oplus p'[P', \mathbf{DS'} \oplus \mathbf{d_r}] \oplus Ps,\ \bowtie,\ MD)$$

$$(14)\quad (p[P, \mathbf{DS} \oplus \mathbf{d}] \oplus Ps,\ \bowtie,\ MD) \longrightarrow (p[P, \mathbf{DS}] \oplus Ps,\ \bowtie,\ MD)$$

$$(15)\quad \text{if } P \equiv Q \text{ and } P' \equiv Q' \quad \frac{(p[\mathbf{P}, DS] \oplus Ps,\ \bowtie,\ MD) \longrightarrow (p[\mathbf{P'}, DS'] \oplus Ps',\ \bowtie',\ MD')}{(p[\mathbf{Q}, DS] \oplus Ps,\ \bowtie,\ MD) \longrightarrow (p[\mathbf{Q'}, DS'] \oplus Ps',\ \bowtie',\ MD')}$$

Table 3: Constant, context, garbage collection and congruence rules.

By axiom (1) the provider announces the availability of the new resource, by producing the replicable datum $newRes@p1$ in the local data space of peer $p1$:

$$(p1[Advertiser, \emptyset] \oplus p2[UserSpec, \emptyset] \oplus p3[UserGen, \emptyset] \oplus p4[UserGen, \emptyset], \bowtie, \emptyset) \longrightarrow (p1[SpecProvider|GenProvider, newRes@p1_r] \oplus p2[UserSpec, \emptyset] \oplus p3[UserGen, \emptyset] \oplus p4[UserGen, \emptyset], \bowtie, \emptyset)$$

At this point, the datum $newRes@p1$ can be replicated in the data space of other peers. Formally, by axiom (13), the system may evolve as follows:

$$(p1[SpecProvider|GenProvider, newRes@p1_r] \oplus p2[UserSpec, \emptyset] \oplus p3[UserGen, \emptyset] \oplus p4[UserGen, \emptyset], \bowtie, \emptyset) \longrightarrow (p1[SpecProvider|GenProvider, newRes@p1_r] \oplus p2[UserSpec, \emptyset] \oplus p3[UserGen, newRes@p1_r] \oplus p4[UserGen, \emptyset], \bowtie, \emptyset)$$

At this point, the datum $newRes@p1$ *can surely be read from process* $UserGen$ *at peer* $p3$*; it can also be read by processes at peers* $p2$ *and* $p4$*, provided that the datum resides in the data space of a peer within the horizon specified in the data retrieval operation.*

For example, if $\bowtie = \{(p2, p1), (p4, p2)\}$*, then the datum can be read both from process* $UserSpec$ *at peer* $p2$ *and from process* $UserGen$ *at peer* $p4$ *(both of them are willing to perform a read operation with horizon 10). However, if* $\bowtie = \{(p2, p1)\}$*, then the datum is accessible only to the process* $UserSpec$ *at peer* $p2$*.*

In both cases, by axiom (7) the datum can be read from process $UserSpec$ *at peer* $p2$*:*

$$\begin{array}{l}(p1[SpecProvider|GenProvider, newRes@p1_r] \oplus p2[UserSpec, \emptyset] \oplus \\ p3[UserGen, newRes@p1_r] \oplus p4[UserGen, \emptyset], \bowtie, \emptyset) \longrightarrow \\ (p1[SpecProvider|GenProvider, newRes@p1_r] \oplus \\ p2[write(specReq, p1).take(specAnsw, 0), \emptyset] \oplus \\ p3[UserGen, newRes@p1_r] \oplus p4[UserGen, \emptyset], \bowtie, \emptyset)\end{array}$$

By axiom (4), the process at $p2$ *performs a remote output operation by producing a misplaced datum to be delivered to the data space of peer* $p1$*:*

$$\begin{array}{l}(p1[SpecProvider|GenProvider, newRes@p1_r] \oplus \\ p2[write(specReqFromp2, p1).take(specAnsw, 0), \emptyset] \oplus p3[UserGen, newRes@p1_r] \oplus \\ p4[UserGen, \emptyset], \bowtie, \emptyset) \longrightarrow \\ (p1[SpecProvider|GenProvider, newRes@p1_r] \oplus p2[take(specAnsw, 0), \emptyset] \oplus \\ p3[UserGen, newRes@p1_r] \oplus p4[UserGen, \emptyset], \bowtie, \langle specReqFromp2 \rangle_{p2}^{p1})\end{array}$$

If $p2$ *is connected to* $p1$*, then by axiom (5') the misplaced datum reaches its destination:*

$$\begin{array}{l}(p1[SpecProvider|GenProvider, newRes@p1_r \oplus] \oplus p2[take(specAnsw, 0), \emptyset] \oplus \\ p3[UserGen, newRes@p1_r] \oplus p4[UserGen, \emptyset], \bowtie, \langle specReqFromp2 \rangle_{p2}^{p1}) \longrightarrow \\ (p1[SpecProvider|GenProvider, newRes@p1_r \oplus specReqFromp2] \oplus \\ p2[take(specAnsw, 0), \emptyset] \oplus p3[UserGen, newRes@p1_r] \oplus p4[UserGen, \emptyset], \bowtie, \emptyset)\end{array}$$

At this point, process SpecProvider at $p1$ *withdraws the request* $specReqFromp2$ *(axiom (8)):*

$$\begin{array}{l}(p1[SpecProvider|GenProvider, newRes@p1_r \oplus specReqFromp2] \oplus \\ p2[take(specAnsw, 0), \emptyset] \oplus p3[UserGen, newRes@p1_r] \oplus p4[UserGen, \emptyset], \bowtie, \emptyset) \longrightarrow \\ (p1[write(specAnsw, p2).SpecProvider|GenProvider, newRes@p1_r] \oplus \\ p2[take(specAnsw, 0), \emptyset] \oplus p3[UserGen, newRes@p1_r] \oplus p4[UserGen, \emptyset], \bowtie, \emptyset)\end{array}$$

and, by axiom (4), provides the answer in a misplaced datum specAnsw that will be delivered to the data space of peer $p2$*:*

$$\begin{array}{l}(p1[write(specAnsw, p2).SpecProvider|GenProvider, newRes@p1_r] \oplus \\ p2[take(specAnsw, 0), \emptyset] \oplus p3[UserGen, newRes@p1_r] \oplus p4[UserGen, \emptyset], \bowtie, \emptyset) \longrightarrow \\ (p1[SpecProvider|GenProvider, newRes@p1_r] \oplus p2[take(specAnsw, 0), \emptyset] \oplus \\ p3[UserGen, newRes@p1_r] \oplus p4[UserGen, \emptyset], \bowtie, \langle specAnsw \rangle_{p1}^{p2})\end{array}$$

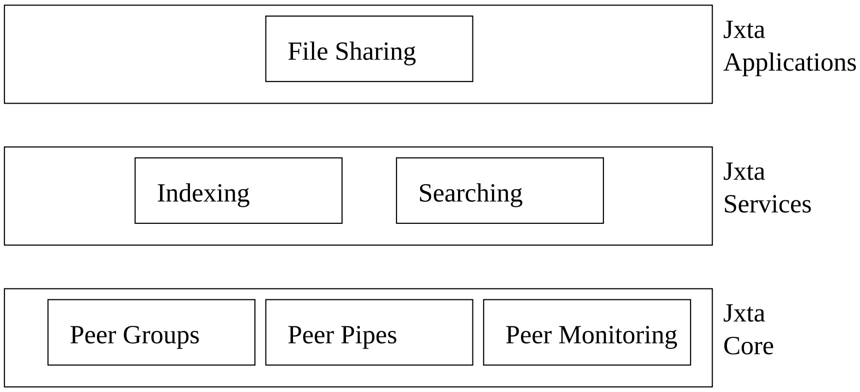

Figure 1: The architecture of the JXTA middleware

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 the previous Section. This implementation, called `JPS`, is written in Java and is based on the JXTA project[18]. 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 as well as on different infrastructures exist.

The choice of JXTA is motivated by the fact that it is currently the unique open-source project that provides a general infrastructure for the construction of P2P services. Other systems exist[21], but they do not provide the generality offered by JXTA.

Figure 1 shows the architecture of the JXTA middleware. The bottom layer is the *JXTA core*, that deals with low-level functions such as peer establishment, peer discovery, communication primitives, routing with firewall and NAT[12] handling, and basic security services. 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.

Several protocols are included in the JXTA core. 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

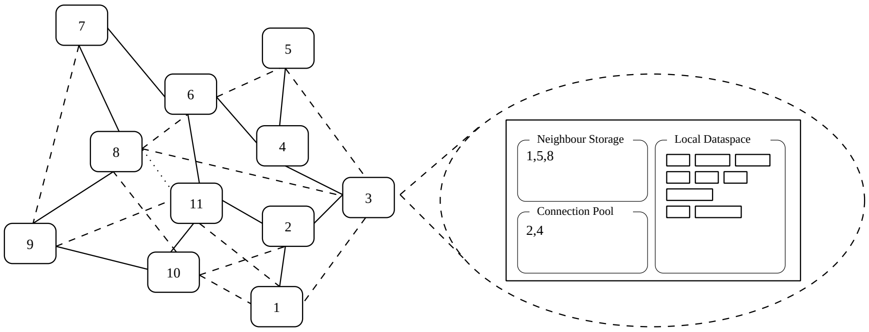

Figure 2: An example of JPS network

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 send generic requests to other peers, while the *pipe protocol* is a more generic communication service that can be used to establish mono- and bi-directional communication channels, called pipes. Finally, the *monitoring protocol* is used to obtain information about the state of other peers.

Peers are networked devices implementing a subset of the JXTA core protocols. Each peer is characterized by a given *peer ID*, and a set of network interfaces (endpoints). A peer ID along with its set of associated peer endpoints, uniquely identifies a peer in the JXTA network. It is not necessary for peers to be directly connected with one another to communicate, since intermediary peers can be used for routing purposes. Peers implementing a common set of services may become member of a *peer group*.

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 2 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

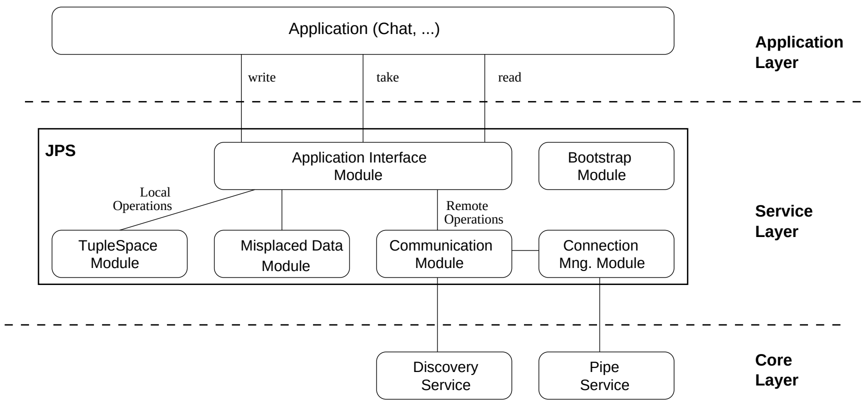

Figure 3: The architecture of a JPS node

peer contains the set of peers with which the local node is currently connected. The peers included in the connection pool are a subset of those included in the neighbour storage. The reason for having a connection pool distinct from the neighbour storage is that the latter may grow to become a very large set, and it would be unfeasible to maintain a connection with each of these neighbours. Together 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 their responsiveness to previously distributed queries.

Figure 3 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 the 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 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[24]. LighTS has been developed as the local core for the implementation of the distributed LIME coordination infastructure[29].
- The *Misplaced Data Storage Module* is used to temporarily store tuples produced by a remote output operation, for which it has not been possible to complete the output operation.
- 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.

As outlined in the previous Section, data produced by local, generic, and replicable output operations are forwarded by the Application Interface Module to the local TupleSpace Module, where they are stored. On the other hand, when a remote output operation is performed, the communication module tries to deliver the datum at the destination, that responds with an acknowledgement. If this is not possible, the Datum is moved to the Misplaced Data Storage Module, that periodically attempts to deliver the datum at the destination. In the current implementation of this delivery mechanism, the delivery is performed only in case of direct reachability between the source and the destination of the datum; i.e., only if a direct connection between them is present. We are considering alternative approaches, that will make use of the routing facilities of JXTA.

Remote *read* and *take* operations are implemented by performing Gnutella-like[20] broadcast searches with the specified time-to-live parameter. The reason behind the choice of using broadcast-based searches is that each peer controls its own data spaces, and is responsible for the data contained in it. No indexing service can be used, as complex queries may be requested; in other words, 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 leases, 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.

In the current prototype of JPS, no load-balancing mechanism has been implemented. We are planning to adopt the scheme described in a related work[27], which is capable to balance the load (in our case, defined as the number of generic tuples maintained in a data space) among a collection of peer nodes. Moreover, we have not implemented any policy for supporting the collection and withdrawal of outdated tuples. We plan to adopt the leasing model as in JavaSpaces[41] with a modified expired tuples collection mechanism. In JavaSpaces data are removed exactly when they expire; we intend to follow a less restrictive policy according to which expired data are removed after expiration only when it is necessary, e.g., to free storage resources.

5. Related Work

In the P2P research area, an emerging approach for communication and coordination among peers is the *distributed hash table* (DHT) paradigm[11,31,36,43]. In this approach, every resource in a P2P environment (documents in file-sharing application, blocks of files in a distributed file system, topics in publish-subscribe tool) is associated with a unique key defined over a virtual address space. Each node in a P2P network is assigned to a subset of the address space and it is responsible to host information on resources whose key falls in this subset. The assignment of keys to hosts is performed dynamically, based on virtual topology defined by the address space and on the dynamics of hosts joining and leaving the system. The only primitive for communication in DHTs enables P2P nodes to send messages to resources, i.e. to the hosts maintaining information on them. With this primitive, it is possible to construct complex coordination mechanisms where communication is decoupled from hosts and it is based on the actual resources present in the P2P network.

PeerWare[13] introduces an alternative approach for supporting the sharing of data in a P2P environment. Each peer holds its own data structure, expressed in terms of a tree of documents. When a group of peers becomes connected, a global virtual data structure is dynamically generated. This structure is obtained by "superimposing" all the local data structures belonging to the peers currently connected.

As described in the Introduction, we can see two separate levels of distribution in LINDA-like coordination models: (i) located data spaces and (ii) located tuples. According to (i) distribution is achieved simply by permitting the interaction among data spaces that are located at different sites. On the other hand, according to (ii) the distributed shared data spaces are obtained as overlay structures that group together tuples that may have different and independent locations. Our coordination model PeerSpaces incorporates both levels of distribution: according to (i) we locate a data space on each peer of the network; we follow the approach (ii) during the execution of data retrieval operations when we consider as the research space all the data currently belonging to the peers in a specified horizon.

Some of the most relevant infrastructures based on the notion of located dataspace are JAVASPACES[41], KLAIM[14], TuCSoN[28], and LOGOP[39]. JAVASPACES is a LINDA-like data space designed for Java-based platforms supporting distributed, mobile and ubiquitous computing. KLAIM is a coordination model particularly suited for network-aware programming with a particular emphasis on security: besides the notion of located data space, an access control policy based on capabilities is introduced in order to avoid undesired accesses to the data spaces. TuCSoN introduces the notion of programmable data space, that is the possibility to program the way the coordination medium reacts to the execution of the coordination primitives. The TuCSoN coordination model has been also extended[33] in order to support the programming of reactions involving more data spaces at a time. Another interesting proposal supporting this possibility is LOGOP[39]: logical operators are used to combine operations performed on different spaces in order to, e.g., removes one datum from one space *and* another one from another space.

The notion of global virtual data space (discussed above in relation to PEERWARE) has been introduced for the first time in LIME[29]. LIME (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 hosts is partitioned in confederations of connected hosts. In LIME, the agents coordinate by exchanging data as in LINDA. Each agent has its own multiset of data that moves with the agent. The data owned by the agents which are currently in execution in the same confederation of hosts are merged in a transiently shared data space, that represents the medium the agents exploit to coordinate themselves. As PeerSpaces, LIME comprises both the notion of located data space and the notion of overlay data structure.

Examples of coordination models based on the notion of independent tuples are TOTA[25] and SwarmLinda[26]. Both models are based on independent devices, each one holding a bag of tuples. In TOTA it is possible to associate to each tuple a propagation rule. This rule indicates whether or not a tuple should be moved/copied from one device to a connected one: this supports a form a transparent movement/replication of the tuples inside the network. SwarmLinda, on the other hand, is inspired by ant based systems: tuples represent food while the templates, used in data retrieval operations to indicate the kind of tuples of interest, are the ants that move inside the network looking for appropriate foods. After tuple retrieval, the templates return to the initial device (following their path backward) communicating about the retrieved information.

6. 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 to continue along two main directions. On the one hand, we plan to develop a complete formal framework for the reasoning about PeerSpaces, comprising adequate observational equivalences, logics for the description of properties, and tools for the automatic verification of these properties. On the other hand, we intend to extend the coordination model with other features such as a (i) support for non-blocking data-retrieval operations, (ii) reactive programming or (iii) transaction operations.

As far as (i) is concerned, we plan to support both the two traditional alternative approaches used for obtaining terminating data retrieval operations: *termination for absence* and *termination for timeout*. The first approach can be used when the considered horizon has a small size: the data retrieval primitive terminates when the horizon is completely visited and no interesting data have been found. The second approach, on the other hand, can be taken into account when a large horizon is of interest: in this case we may consider a timeout which expresses the maximum amount of time for which the peer is willing to wait for an interesting datum.

As far as (ii) and (iii) are concerned, we plan to adapt to our setting coordination primitives such as the *notify* of JavaSpaces[41] or the *monitor* of WCL[34] (for reactive programming), and to investigate either global operations such as *copy-collect*[37], the addition of logical operations such as in LogOp[39], 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 that, according to the PeerSpaces model, cannot be explicitly removed.

Acknowledgement: This paper is based on a preliminary work[8] written in collaboration with Cristian Manfredini, the implementor of JPS. We thank the anonymous referee for his insightful comments and his precious suggestion for improving the presentation.

References

1. K. Aberer and M. Hauswirth. Peer-to-Peer Information Systems: Concepts and Models, State-of-the-Art, and Future Systems In *Proc. of 18th International Conference on Data Engineering* (ICDE), San Jose, California, 2002.
2. S. Adler. The Slashdot Effect: An Analysis of Three Internet Publications. http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html.
3. D. Anderson. SETI@home. In A. Oram, editor, *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*, chapter 5. O'Reilly, Mar. 2001.
4. N. Busi, R. Gorrieri, and G. Zavattaro. Process Calculi for Coordination: from Linda to JavaSpaces. In *Proc. of AMAST 2000*, volume 1816 of *Lecture Notes in Computer Science*: 198-212. Springer-Verlag, Berlin, 2000.
5. N. Busi and G. Zavattaro. Some Thoughts on Transiently Shared Dataspaces. In *Proc. of the Workshop on Software Engineering and Mobility (at ICSE 2001)*, 2001.
6. N. Busi and G. Zavattaro. Expired Data Collection in Shared Dataspaces. *Theoretical Computer Science*, volume 298: 529-556, Elsevier, Marzo 2003.
7. 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.
8. N. Busi, C. Manfredini, A. Montresor, and G. Zavattaro. PeerSpaces: Data-driven Coordination in Peer-to-Peer Networks. In *Proc. of ACM Symposium on Applied Computing (SAC'03).* ACM Press, 2003.
9. N. Carriero. Implementation of Tuple Space Machines. PhD thesis, Yale University, 1987. YALEU/DCS/RR-567.
10. P. Ciancarini,R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche. Coordinating Multiagent Applications on the WWW: a Reference Architecture. *IEEE Transactions on Software Engineering*, 24(5):362–375, 1998.
11. Frank Dabek et al. Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service. In *Proc. of 8th hotos*, IEEE, 2001.
12. K. Egevang, K. Francis and P. Francis. The IP Network Address Translator (NAT). RFC 1631, May 1994.
13. G. Cugola and G.P. Picco. Peerware: Core middleware support for peer-to-peer and mobile systems. Draft, 2002. Available at www.elet.polimi.it/upload/picco
14. R. DeNicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
15. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
16. D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107, 1992.
17. M.R. Horton and R. Adams. Standard for interchange of USENET messages. RFC 1036, December 1987. Available at http://www.faqs.org/rfcs/rfc1036.html
18. Project JXTA. http://www.jxta.org.
19. J. Kubiatowicz et al. OceanStore: An Architecture for Global-Scale Persistent Storage. In Proc. of *9th International Conference on Architectural support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.
20. G. Kan. Gnutella. In A. Oram, editor, *Peer-to-Peer: Harnessing the Benefits of a*

Disruptive Technology, chapter 8. O'Reilly, Mar. 2001.

21. G. Kortuem et al. When Peer-to-Peer comes Face-to-Face: Collaborative Peer-to-Peer Computing in Mobile Ad Hoc Networks. In *Proceedings of the 1st International Conference on Peer-to-Peer Computing*, Linkping, Sweden, August 2001.
22. Groove Networks. http://www.groove.net
23. 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.
24. Project LighTS. http://lights.sourceforge.net/docs/info/intro.html.
25. M. Mamei, F. Zambonelli and L. Leonardi. Programming Context-Aware Pervasive Computing Applications with TOTA. Technical Report University of Modena and Reggio Emilia, No. DISMI-UNIMO-2002-22, August 2002.
26. R. Menezes and R. Tolksdorf. A New Approach to Scalable Linda-systems Based on Swarms. In *Proc. of ACM Symposium on Applied Computing*, SAC 2003, Melbourne, FL, USA.
27. A. Montresor, H. Meling and Ö. Babaoğlu. Messor: Load-Balancing through a Swarm of Autonomous Agents In Proc. of *1st Workshop on Agent and Peer-to-Peer Systems*, Bologna, Italy. July 2002.
28. A. Omicini and F. Zambonelli. Coordination of mobile information agents in tucson. *Journal of Internet Research*, 8(5), 1998.
29. 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.
30. Peer-to-Peer Working Group. http://www.p2pwg.org
31. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM'01*, San Diego, CA. August, 2001.
32. A. Ricci, A. Omicini, and E. Denti Virtual Enterprises and Workflow Management as Agent Coordination Issues. International Journal of Cooperative Information Systems (IJCIS), combined September & December, 2002.
33. A. Ricci, A. Omicini, and M. Viroli Extending ReSpecT for Multiple Coordination Flows. In *Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications*, (PDPTA'02), June 2002, Las Vegas, NV, USA. Vol. III, CSREA Press, 2002.
34. A. Rowstron. WCL: A web co-ordination language. *World Wide Web Journal*, 1(3):167–179, 1998.
35. 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.
36. A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of 18th International Conference on Distributed Systems Platforms*, 2001.
37. 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.
38. 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.
39. J. Snyder and R. Menezes. Using Logical Operators as an Extended Coordination Mechanism in Linda In *Proc. of 5th International Conference on Coordination Models and Languages*, COORDINATION, 2002.
40. 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.
41. J. Waldo et al. Javaspace specification - 1.0. Technical report, Sun Microsystems, March 1998.
42. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.
43. B.Y. Zhao, J. Kubiatowicz, and A.D. Joseph. Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing. UCB/CSD-01-1141, U.C. Berkeley, 2001.