Seamless Distribution of Data Centric Applications through Declarative Overlays

Ahmad Ahmad Kassem, Stéphane Grumbach, Eric Bellemon

Seamless Distribution of Data Centric Applications through Declarative Overlays

Ahmad Ahmad-Kassem
Université de Lyon, INRIA, CITI
ahmad.ahmad_kassem@inria.fr

Eric Bellemon
INRIA, LIAMA
eric.bellemon@gmail.com

Stéphane Grumbach
INRIA
stephane.grumbach@inria.fr

Abstract

We present an approach based on peer-to-peer overlays which allows to distribute seamlessly data centric applications defined by queries over a centralized database. We consider applications in which the users have access to views, which contain horizontal fragments of the data of interest to them. The peer-to-peer overlays are defined by simple declarative programs in the Netlog language. The communication relies on implicit addresses, which can be evaluated on the fly, and ensure persistence of data in messages. We demonstrate the technique on a multiplayer online game, written in SQL, with players who connect to a mobile ad hoc network through their portable devices. The overlay is defined by a combination of an ad hoc routing protocol, DSDV, together with a DHT. The application runs on the QuestMonitor platform, which allows to monitor the communication between peers, the evolution of the local data stores, as well as the execution of the declarative code.

Keywords: peer-to-peer overlays; declarative networking; multiplayer online games

Résumé

Nous proposons une approche basée sur les réseaux pair à pair pour distribuer de manière transparente des applications définies par des requêtes sur une base de données centralisée. Nous considérons des applications pour lesquelles les utilisateurs ont accès à des vues qui contiennent des fragments horizontaux des données qui les intéressent. Les réseaux logiques sont définis par des programmes déclaratifs simples dans le langage Netlog. La communication repose sur l'usage d'adresses implicites, qui peuvent être évaluées à la volée, et garantissent la persistance des données dans les messages. Nous montrons le fonctionnement du système pour un jeu en ligne multijoueur, joué par des joueurs qui utilisent leurs terminaux portables connectés à un réseau mobile ad hoc. Le réseau logique repose sur un protocole de routage pour réseau ad hoc, DSDV, couplé à une DHT. L'application tourne sur la platforme QuestMonitor, qui permet de contrôler la communication entre pairs, l'évolution des données locales, ainsi que l'exécution du code déclaratif.

Mots clés: réseaux pair à pair; protocoles déclaratifs; jeux multijoueurs en ligne

[0]This work has been supported by the Agence Nationale de la Recherche, under grant ANR-09-BLAN-0131-01.
[0]CITI Laboratory, INSA Lyon, 6 avenue des Arts, 69621 Villeurbanne Cedex

1 Introduction

Peer-to-peer systems have been widely used to alleviate the burden of servers, by transferring to peers in a network tasks that do not require a centralization of the information. Their architecture can be more or less structured, with nodes playing identical or different functions, and with or without interaction with a centralized server. They have been very successful in various fields such as file sharing (e.g. Napster, Gnutella), and communication networks (e.g. skype). A wide range of applications are now emerging over peer-to-peer systems, such as social networking [5, 19], multiplayer games [14, 11], mobile messaging [22], video broadcasting [15], etc.

Most of these applications are essentially data centric, they rely on exchange of data between peers, and could be expressed by queries over a database. In this paper, we demonstrate how such applications programmed as a collection of queries over a database, can be ported seamlessly, that is without changing the queries, from a client-server architecture, to a peer-to-peer architecture with the appropriate overlay. Moreover, following the trend open by the declarative networking [17, 16], we show that the overlay can be defined using declarative data centric programs, thus resulting in a fully data centric modeling of the peer-to-peer application.

Let us consider for instance online multiplayer games, which constitute a very promising application for peer-to-peer systems. This type of application relies on a scenery from a virtual world, which constitute static data with graphical properties that are out of the scope of this work. Games also involve mutable objects, whose properties can be updated, and avatars representing the players, that can change their attributes. Most of the actions of such games can be captured in a purely data centric perspective, even if like for other applications, additional characteristics are important, such as trust and security issues [7], as well as real time aspects, essential for communication systems [8].

In all these applications, the clients can access data of interest to them, which can generally be defined by views over some horizontal fragments of the data structures. The clients can perform actions, which consist in querying or updating these views (e.g. moving an avatar means updating its position), while the system can perform more general actions such as queries and updates over the whole data. We show that under some restrictions under the views and queries allowed, the application can be ported seamlessly over overlay networks and executed efficiently.

Numerous techniques have been developed to support peer-to-peer overlays, such as Chord [25], or Pastry [24] for instance. As distributed algorithms in general, they require high programming skills, and their correction is very difficult to guarantee. High level programming abstraction, such as data centric programming languages constitute a very promising model in this context [12]. They are more declarative, so facilitate programming, they parallelize well, so facilitate the execution, they manipulate explicitly data structures, so facilitate verification of their properties.

The use of rule-based languages, à la Datalog [1, 2, 26, 23], developed in the field of databases in the 1980's, for distributed applications, was initially proposed in UC Berkeley [18, 16], under the name "declarative networking". It was shown that such languages augmented with communication primitives, allowed to express communication protocols or P2P systems with code about two orders of magnitude shorter than imperative programs, and with reasonable execution models.

We used the rule-based language, Netlog [10], which extends Datalog with aggregation and non-deterministic constructs as well as communication primitives, in the spirit of the declarative networking approach. It has a sound distributed fixpoint semantics, which takes explicitly into account the in-node behavior as well as the communication between nodes. Netlog runs on the *Netquest Virtual Machine*, which is coupled with an embedded Data Management System, DMS, which stores all the data as well as the bytecode of the Netlog programs. The bytecode is obtained by a compilation from Netlog into an SQL dialect. The Virtual machine makes calls to the DMS to evaluate the bytecode of the Netlog programs, which result in updates of the database, and production of messages. Implicit addresses have been added to the Netquest machine, to handle messages to peers that perform server duties. They ensure the persistence of the data even in case of changes in the network due to a node failure or departure.

The Netquest machine has been shown to be portable over small devices, as long as they support an embedded DMS [3]. It runs as well on the QuestMonitor platform [4], which allows to monitor the communication between peers, the evolution of the local data stores, as well as the execution of the declarative code. Moreover, proof techniques have been developed in Coq to certify Netlog programs [9].

We describe the technique over an example of a multiplayer online game, which can be defined by sequences of queries over a central database, presented in Section 2. Each player has access to views giving the data pertaining either to its avatar in the game (simple views), or to the region where it is involved (geographic views). We consider a game where a player participates to an auction by making a bid, while the system at the expiration of the auction, changes the owner of the object, and updates the bank accounts of both the seller and the buyer according to the price.

So far as the network is concerned, we make the following assumptions on the application. We assume that players participate to the game over a network to which they connect through devices in some short range communication mean (e.g. bluetooth). The players thus form an ad hoc mobile network. They can physically enter or leave the network, as well as move from one place to another, without being disconnected from the application. The players form a pure peer-to-peer system, with nodes playing identical roles, and no centralized server. The overlay network is formed by a distributed hash table, DHT, which is used to distribute both data and computation initially performed by the server.

We assume that each table has (at least) an index attribute. The values of index attributes are mapped to hash keys using a hash function. The corresponding horizontal fragments of the relations are stored on the node which has the largest Id smaller than the hash key. Some restrictions are imposed on the queries to ensure smooth distribution through the peer-to-peer system, making full use of the DHT. Queries should in particular have at least one *where* argument on an index attribute. Moreover join should be performed on index attributes as well. Under such restrictions, we show that the distribution can be done smoothly using the DHT protocols.

For the game application, we propose a DHT constructed over a routing table defined and maintained by a DSDV like protocol [20]. DSDV is a table-driven routing protocol based on the Bellman-Ford algorithm, well adapted to ad hoc networks. We have considered other routing protocols as well such as OLSR [13], AODV, for on-demand routing, well adapted to a network with many changes, [21], or VRR [6] based on a ring, which are useful under other network conditions.

The main contribution of the paper is to show that the whole specification of such applications can be made in a fully data centric approach, relying on simple centralized queries for the application, and data centric programs written in Netlog for the DHT protocols. We show in particular in Section 4 that a few dozen of rules are sufficient to express the DHT for mobile ad hoc networks.

Our experiments with the game are presented in Section 5. The scenario, with a node joining the network, participating to the game, moving physically while playing, and leaving the network before the distributed server handles the updates in the game, shows the robustness of the proposed protocols in Netlog. Our experiments show that the movement of one node does not affect the game, all data are preserved, and the duties ensured by the leaving nodes are distributed to other nodes.

The paper is organized as follows. In the next section, we explain the motivation through the multiplayer games, and show the distribution over an overlay. In Section 3, we present the Netlog language together with the virtual machine to evaluate Netlog programs. Section 4 is devoted to fundamental protocols supporting the overlay. In Section 5, we illustrate over the example, the protocols to distribute the applications specified as centralized queries, and test these protocols over the QuestMonitor system.

2 Motivation

We consider applications which can be described as sequences of database updates performed by clients over a centralized server. The server stores all the data from all clients, while the clients can access and modify only some part of these data, which can be defined by views over the whole data. In this section, we consider such an example to illustrate our technique.

On-line multi-player games over virtual worlds, such as Second Life, World of Warcraft, etc. constitute fundamental applications of this type. Currently, most massively multiplayer games are implemented on a client-server architecture, with a server which handles both client accounts and game states. Various types of clusters are used, for scalability purposes, to support massive numbers, millions, of players at the same time.

If we leave apart the graphical interface in which players evolve, the basic actions they perform can be modeled easily as database updates. Clients generally participate to the game through an *avatar*. The game relies on some stable "landscape", which can be seen by the clients, using their views over the global data. Most games support mutable virtual objects, which can be changed (created, destroyed, exchanged, etc.) by the players during the course of the game.

The server knows at every moment the connected clients, as well as the updates they make on the data (e.g. creating, deleting or moving avatars, exchanging virtual objects, etc.). We illustrate over an example how each of the basic actions of the payers can be described with a set of queries over the centralized database of the server.

The list of avatars of players is stored in the table **Owner**, which contains an authentication key for each avatar. We consider a simple game in which avatars exchange, sell or buy objects through an auction market. Each avatar owns bank accounts (table **Bank**), can register itself into a market (table **Market**), buy or sell their objects (table **Objects**), into an auction market (table **Auction**). Samples of the tables used in the example are shown below.

Table **Owner**

<u>**Avatar**</u>	**Auth**	**Client**
Toto	37	0012
Loulou	54	0193

Table **Bank**

<u>**Name**</u>	**Avatar**	<u>**Account**</u>	**Balance**
WorldBank	Toto	123985642	1524
GlobalBank	Loulou	AB87532	845

Table **Object**

Class	**Avatar**	**Name**	**Price**	<u>**Id**</u>
Pet	Toto	2 yo Mouse	75	1

Table **Market**

<u>**Name**</u>	**Avatar**	**Account**
Catown	Toto	123985642
Catown	Loulou	AB87532

Table **Auction**

<u>**Name**</u>	<u>**Seller**</u>	**Buyer**	**OId**	**Price**	**ExpTime**	**MinPrice**
Catown	Toto	Loulou	1	150	1305273029	100

Each client has a local view of these data, which concerns their avatar, say α. We distinguish between *simple views*, such as the bank accounts or the objects that belong to the avatar α, defined with a *where* condition of type Object.Avatar=α; and *geographic views*, where an avatar can get all the fragment of for instance the market on which it occurs. The local view of table, **Table**, is denoted **VTable**. Views have the same structure as the corresponding table, without the Avatar attribute in the case of simple views.

We distinguish between two types of actions: *client action* and *system action*. The only actions clients can perform is to query or update their local views. Note that some attributes might be system defined (e.g. Account, Balance). An avatar, for example, cannot retrieve the account balance of another avatar neither change the balance of its bank account. When an avatar updates data in a view, the update is sent to the server to be performed. Systems actions consist of queries over the global schema and are triggered and performed by the server. We show below the main actions of the auction game which are defined by simple queries.

1. See the auctions on the market where I am (geographical view)

```
INSERT INTO VAuction
 SELECT Auction.Name, Auction.OId, Auction.Price, Auction.Seller
 FROM Auction, Auction AS Auction2, Owner
 WHERE Auction.Name = Auction2.Name AND Auction2.Seller =
     Owner.Avatar AND Owner.Client = self;
```

2. Propose a new auction (client action)

```
INSERT INTO VAuction
 SELECT 'Catown', Owner.Avatar, Owner.Avatar,
         Object.Id,100,1305273029,100
  FROM VAuction, Owner
  WHERE Owner.Client = self AND VObject.Id= Id;
```

3. Make a bid (client action)

```
UPDATE VAuction
 SET VAuction.Buyer =Owner.Avatar,  VAuction.Price='150'
 WHERE VAuction.Name='Catown' AND VAuction.Seller = 'Toto'
        AND VAuction.OId = 1 AND Owner.Client = self;
```

4. Cancel a bid (client action)

```
DELETE FROM VAuction
 WHERE VAuction.Name='Catown' AND VAuction.Seller = Owner.Avatar
        AND VAuction.OId = 1 AND Owner.Client = self;
```

5. Conclude an auction when the auction has expired (system action)

```
UPDATE Bank
   SET Bank.Balance = Bank.Balance - 150
   WHERE Bank.Account = 'AB87532';
UPDATE Bank
   SET Bank.Balance = Bank.Balance + 150
   WHERE Bank.Account = 'AB123985642';
DELETE  FROM Auction
   WHERE Auction.Name='Catown' AND Auction.Seller = 'Toto'
        AND Auction.OId = 1;
UPDATE Object
   SET Object.Avatar = 'Loulou', Object.Price=150
   WHERE Object.Id = 1;
```

In their seminal paper, Knutsson et al. [14] showed how such a multiplayer game could be developed over a P2P architecture using the Pastry [24] overlay network. We show how more generally any application described by queries and views as presented above can be distributed seamlessly over an overlay.

We consider a network constituted by peers that communicate in a peer to peer fashion (over the Internet, device to device, etc.). They can join and leave the network at any time and participate to the applications with the other connected peers. Collectively they support the tasks of the centralized server, by storing fragments of the data on peers, and by routing the queries (views, client actions, system actions) to the peers in charge. There is no centralized server, and the peers play the same role.

The application is defined over some *schema*, which will be used in the distributed environment exactly like in the centralized one. Each relation has at least one special attribute called *index attribute* (underlined in the tables of our example), such as Avatar in table Owner, and Id in table Object. Attributes of the tables are either client-defined (e.g. Name, Price in table Object), or system defined (e.g. Balance in Bank).

The *views* are expressed as select queries over a single table of the schema. There are two types of views, *simple views* and *geographic views*. Simple views are expressed as select queries with a *where* argument over the index attribute which should be some Id of the client (e.g. the avatar in the game). Geographic views are expressed as select queries with a similar where condition, but allowing a self equijoin over the geographic attribute of the table. The *client queries* are queries over the views available on the client. *System queries* are queries over the global schema. Joins are limited to index attributes.

The data are fragmented and distributed as follows using a DHT. The values of the *index attribute* are mapped to a random uniformly distributed set of circular ID's, used to form the Distributed Hash Table. We assume for simplicity that the node Id's are uniformly distributed, and that the hash values are over the same domain as the node Id's. A circular order is defined over this domain. Each node α is responsible for storing the fragment of each relation corresponding to the tuples whose index value is such that α is the largest node Id smaller than the index value of the tuple.

Several mechanisms can be used to ensure the replication of the data so that nodes can leave the network without perturbing the application. In the sequel we use replication and synchronization methods to ensure the persistence of the data under nodes movement.

3 The Netlog language for distributed protocols

Netlog programs consist of sets of recursive rules of the form *head* :- *body*, where the *head* is derived when the *body* is satisfied. The programs are installed on each node of a network, where they run concurrently. The computation is distributed and the nodes exchange information. The facts deduced from the rules can be either stored on the node on which the rules run, or sent to other nodes.

We present the language through some fundamental examples of programs for network organization, routing, and sensor network monitoring.

In the following *Program tree*, we consider the construction of a spanning tree. The results are distributed so that each node stores the knowledge of its parent in the tree.

Program Tree

$$\updownarrow onST(self) :- Root(self). \tag{1}$$

$$\downarrow ST(\diamond y, self) :- Link(self, y), onST(y), \neg onST(self). \tag{2}$$

$$\updownarrow onST(self) :- Link(self, y), onST(y), \neg onST(self). \tag{3}$$

The variable $self$ is interpreted by the node address. The **store/push operator**, "$\updownarrow$", in front of rules, determines where the results are assigned. The effect of "$\downarrow$" is to **store** the results of the rule on the node where it runs; "$\uparrow$", to **push** them to its neighbors; and "$\updownarrow$", to both store and push them.

The negation is interpreted by local closed world assumption (a fact is not true on a node if it is not stored on that node). The **choice operator** $\diamond$ chooses non-deterministically a parent among the possible choices.

Assume that $Root(\rho)$ holds on a root node ρ exclusively. When a node α is on the spanning tree, it broadcasts $onST(\alpha)$ to its neighbors by Rules (1) and (3). The fact $ST(\alpha, \beta)$, which is deduced by node β, is stored exclusively on node β, and saved as parent the node α, by Rule (2).

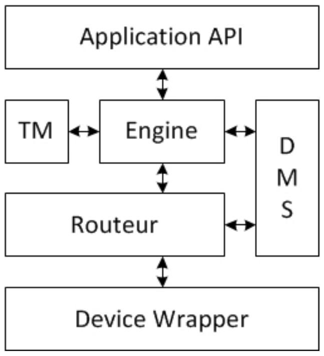
Program Sensor monitoring

$$\uparrow Req(self) : -Temperature(self, t), t > m\_threshold. \tag{4}$$
$$\uparrow Rep(self, @x, t) : -Req(x), Temperature(self, t). \tag{5}$$
$$\downarrow Temperature(x, t) : -Rep(x, self, t), \neg Temperature(x, \_). \tag{6}$$
$$\downarrow TpAvg(x, avg(t)) : -Temperature(x, t). \tag{7}$$

The program *Sensor monitoring* uses to monitor the temperature on sensors. The temperature of the sensors are stored in the relation $Temperature$ with attributes $nodeId$ and $temperature$. The location instruction "@" in the head of rules represents the destination. Rule (5) **unicasts** its results, using the **location instruction** "@", on the second variable of the head, instead of pushing them to all neighbors. The constant $m\_threshold$ in Rule (4) is a *metadata* defined in the header of the program, and the underscore used in Rule (6) denotes "any value".

When the temperature of a sensor, say α, is greater than the defined threshold, it sends a request to all neighbors to retrieve their temperature in Rule (4). Neighbor sensors, upon receiving the request in Rule (5) , unicast their current temperature to α, which in its turn saves the temperature of their neighbors upon receiving their answers in Rule (6). Finally, sensor α, using the aggregation avg in Rule (7), computes the average temperature of all its neighbors and stores it in the relation $TpAvg$.

The Netquest Virtual Machine executes the Netlog bytecode and manipulates data and messages. It is working as a daemon in the device, and applications can use it to communicate with other devices on the network. The virtual machine is portable and can be installed in small devices with embedded DMS. A previous implementation was done in iMote sensors [3].



The Netquest Virtual Machine is composed of six components:

- Device Wrapper for QuestMonitor: provides an abstraction layer of the network. It receives and sends data over the network, and does the address translation between Netlog internal addresses and the network addresses.

- Data Management System (DMS): provides an access to the data. This module evaluates the bytecode, manipulates data (insertion, update and deletion) and produces messages.

- Router: receives and sends Netquest messages through the device wrapper. It chooses the best route to reach each destination. The strategy to select the route can be easily defined in Netlog.

- Engine: executes the Netlog programs. When a node receives a new message, the engine loads the rules matching the facts of the message and evaluates them through the DMS.
- Timer Manager (TM): manages time event of the system. Netlog programs can create and manipulate timers. These timers are managed and fired by this module.
- Application API: this module is an interface between the virtual machine and applications. An external application can use the Netlog Virtual Machine to send and receive messages over the network.

The generated bytecode is a SQL dialect. A query is built for each operator (store, push and deletion) in a rule.

Consider for instance the following rule which contains the three operators. The @ symbol in the body of the rule followed by the variable a denotes the *location specifier*, where the evaluation of the rule is taken place (on node a). The symbol "!" denotes the deletion of the facts used in the evaluation of the rules.

$$\begin{aligned} \updownarrow Link(a,b) :- !Hello(b,@a), \\ \neg Link(a,b). \end{aligned}$$

This rule is evaluated when the engine receives a $Hello$ message. It is translated into three SQL queries corresponding to each operator.

The first query is the result for the operator push, the second for the operator store and the third for the deletion.

All the keyword beginning by $m\_$ (e.g. m_self) are replaced by the engine during the evaluation of the rule. The negation of $Link$ is translated with the sub-query into the section $not\ exists$.

```
SELECT Hello.a, Hello.b
  FROM Hello
  WHERE Hello.a='m_self'
  AND NOT EXISTS (
    SELECT Link.a, Link.b
    FROM Link
    WHERE Link.a=Hello.a
       AND Link.b=Hello.b);

INSERT INTO Link
  SELECT Hello.a, Hello.b
  FROM Hello
  WHERE Hello.a='m_self'
  AND NOT EXISTS (
    SELECT Link.a, Link.b
    FROM Link
    WHERE Link.a=Hello.a
       AND Link.b=Hello.b);

UPDATE Hello
  SET Hello.deleted=1
  WHERE Hello.a='m_self'
  AND NOT EXISTS (
    SELECT Link.a, Link.b
    FROM Link
    WHERE Link.a=Hello.a
       AND Link.b=Hello.b);
```

Nodes communicate by messages which have the following format: $< dest, payload >$, where $payload$ is the content of the message (a set of facts), and $dest = (exdest, imdest)$, where $exdest$ is an *explicit destination* (e.g. node Id), and $imdest$ is an *implicit destination*, that is a query whose answer is of type node Id.

When a message is received by a device, the device wrapper transfers it to the router. The message is read by the router and the payload is sent to the engine if the device belongs to the destination, otherwise it is forwarded to the destination. If the explicit destination is known to the node, the implicit destination is ignored. Otherwise, the query of the implicit destination is fired on the local store of the node, and the result is used as new explicit destination. As we will see, the implicit destinations reveal very useful to handle the destination obtained by the DHT, which can have left the network, and can be recomputed on the fly.

The engine loads rules from Netlog programs matching the facts contained in the payload and then evaluate these rules using the DMS. The DMS can update or delete data and create messages to be sent. These new messages are sent to the network through the device wrapper.

The Netlog Engine does not execute directly the bytecode. It orchestrates the tasks to be done to treat messages and facts. When receiving facts, a new round starts and a first stage is executed. In this stage, the engine loads and executes rules triggered by these facts. If there are derived facts produced by the engine, a new stage is executed recursively again with these new facts. A round is finished when there are no new derived facts. At the end of a round, produced messages are sent to other nodes in unicast or broadcast mode.

4 Declarative overlays

Netlog is well adapted to the development of networking protocols. In this section, we show how the basic protocols which are used to distribute the server tasks over a DHT can be written. We first present a routing protocol, which defines the routes in the network. Then a ring, that is a circular order, is defined over the node Ids. Each node then constructs the segment of keys it is responsible for. Finally, the replication of the fragments to ensure persistence of the game in the event of failure of a node is defined.

The program DSDV, presented below, is a simplified version of the DSDV protocol [20], which constructs and maintains proactively all possible routes in ad hoc networks. We chose DSDV to support mobile clients participating to a multiplayer game in a device to device network. The routes are stored in relation $Route$ with attributes $dest$, $nextHop$, $nbHops$, $destSN$ and $expirationTime$, where $destSN$ is used for sequence numbers.

The program is composed of four modules. Each module can be triggered by an incoming fact or by a timer. For example, the module ini, composed of Rule (8), is triggered by the timer ini, while the module $UpdRoute$ is composed of Rules (11), (12) and (13) and triggered by the fact $HelloRoute$. When a module is triggered, all the rules of this module are evaluated in parallel.

Each node initially creates a route to itself when the program starts with Rule (8), triggered by the timer ini, which is fired only once. Periodically, using the timer $hello$, each node in Rule (9) broadcasts all its route information to its neighbors, and increases the value of the sequence number $destSN$, of the route to itself, using Rule (10).

The route information are sent using facts of the form $HelloRoute$. A node updates its routing table according to the received route information from its neighbors as follows: (i) a new route is stored Rule (11) if there is no route to the same destination in the local route table, (ii) the old route is deleted and replaced with a new one, if the new route has a larger destination sequence number, Rule (12), or the new route has the same sequence number as the old one but has a smaller number of hops, Rule (13). Each node sets for each route a timeout $m\_timeout$, in Rules (11), (12) and (13) upon saving it in the routing table. In Rule (14), each node periodically using the timer $checkRoute$ deletes all expired routes.

Program DSDV

module(ini)

$$\downarrow Route(self, self, 0, 1, 0) :-\Big\{\ TimeEvent('ini'). \tag{8}$$

module(hello)

$$\uparrow HelloRoute(self, x, n, s) :-\Big\{\ TimeEvent('hello'), Route(x, y, n, s, \_). \tag{9}$$

$$\downarrow Route(self, self, 0, s, 0) :-\begin{cases} TimeEvent('hello'), \\ !Route(self, self, 0, s', 0), s := s' + 1. \end{cases} \tag{10}$$

module(UpdRoute)

$$\downarrow Route(x, \diamond y, n, s, t) :-\begin{cases} HelloRoute(y, x, n', s), \neg Route(x, \_, \_, \_), \\ n := n' + 1, t := m\_time + m\_timeout. \end{cases} \tag{11}$$

$$\downarrow Route(x, \diamond y, n, s, t) :-\begin{cases} HelloRoute(y, x, n', s), n := n' + 1, s' < s, \\ !Route(x, y', n'', s'), t := m\_time + m\_timeout. \end{cases} \tag{12}$$

$$\downarrow Route(x, \diamond y, n, s, t) :-\begin{cases} HelloRoute(y, x, n', s), n := n'+1, n'' > n' + 1, \\ !Route(x, y', n'', s), t := m\_time + m\_timeout. \end{cases} \tag{13}$$

module(checkRoute)

$$:-\begin{cases} !TimeEvent('checkRoute'), !Route(\_, \_, \_, t), \\ t < m\_time, t <> 0. \end{cases} \tag{14}$$

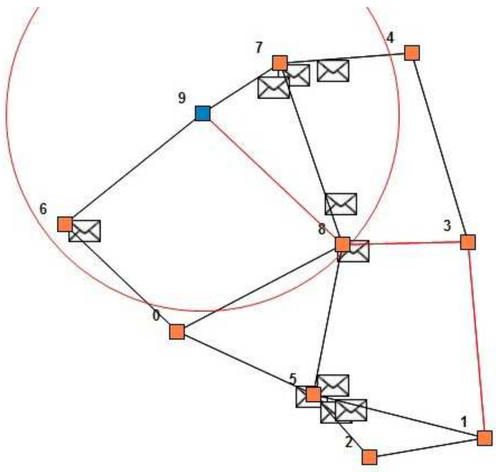
The next figure shows an example of dynamic network in which $node$ 9 is moving between nodes. We monitor and display the content of the routing table of $node$ 9. The left figure and related table represent the network and the content of the table of $node$ 9 before changing the position while the right figure and related table represent $node$ 9 after changing its position. The route between source $node$ 9 and the destination $node$ 1 is colored in red. As we notice, all routes are up to date, with the DSDV protocol, and there is always a route between all the nodes.

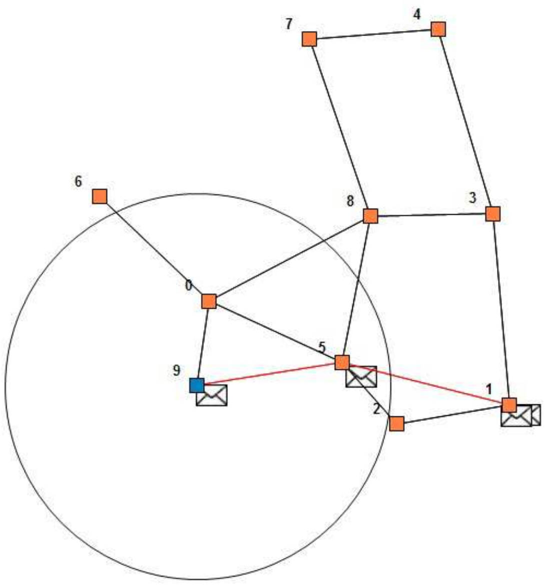
Route

dest	**nextHop**	**nbhops**	**routesn**	**exptime**
0000	0008	2	32	103055929
0001	0008	3	30	103055929
0002	0008	3	30	103055929
0003	0008	2	32	103055929
0004	0007	2	32	103057272
0005	0008	2	32	103055929
0006	0006	1	34	103056834
0007	0007	1	34	103057272
0008	0008	1	34	103055929
0009	0009	0	36	0

Route

dest	**nextHop**	**nbhops**	**routesn**	**exptime**
0000	0000	1	54	103190702
0001	0005	2	52	103189468
0002	0005	2	52	103189468
0003	0000	3	50	103190702
0004	0000	4	48	103190702
0005	0005	1	54	103189468
0006	0000	2	52	103190702
0007	0000	3	50	103190702
0008	0000	2	52	103190702
0009	0009	0	56	0





The following program, RING, defines a circular order on the node Ids (with the smallest node Id, immediate successor of the biggest one), used in the DHT. The index values of each relation are mapped to hash values. Each node is responsible for an interval of these values, according to the value of its identifier, and their predecessor and successor in the ring.

Since we consider mobile ad hoc networks, the next program relies on the previous routing protocol, ensuring that each node has routes to all nodes in the network, which are maintained over time. Any participating node may be part of the DHT and hold data related to the application. The program uses the relations *Intvl* with two attributes to save the *predecessor* and the *successor* of each node, similar for *NIntvl* to calculate the new interval. The relation *Extreme* with two attributes saves the nodes with *minimum* and *maximum* Id in the network.

Program RING

$\mathbf{module(ini)}$

$$\downarrow Intvl(self, self) :- \Big\{\ TimeEvent('ini'). \tag{15}$$

$\mathbf{module(extreme)}$

$$\downarrow Extreme(min(d), max(d)) :- \Big\{\ TimeEvent('chNgb'), Route(d, \_). \tag{16}$$

$\mathbf{module(interval)}$

$$\downarrow NIntvl(max(d1), min(d)) :- \left\{ \begin{array}{c} !Extreme(\_, \_), Route(d, \_), d > self, \\ Route(d1, \_), d1 < self. \end{array} \right. \tag{17}$$

$$\downarrow NIntvl(y, min(d)) :- \Big\{\ !Extreme(self, y), Route(d, \_), d > self. \tag{18}$$

$$\downarrow NIntvl(max(d), x) :- \Big\{\ !Extreme(x, self), Route(d, \_), d < self. \tag{19}$$

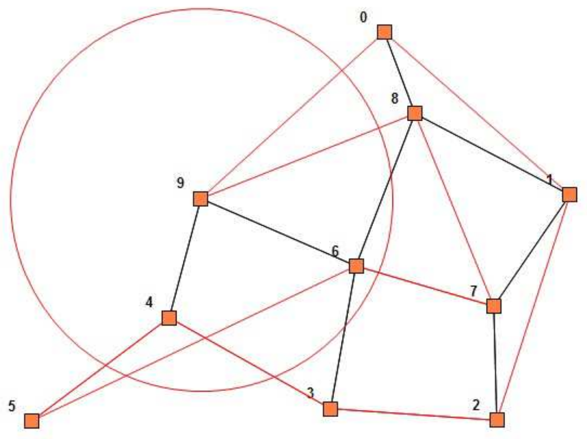
$\mathbf{module(updateInt)}$

$$\downarrow Intvl(np, ns) :- \Big\{\ !Intvl(\_, \_), !NIntvl(np, ns). \tag{20}$$

The program is composed of four main modules. Each node in the network initially sets its successor and predecessor to its node Id when the program starts with rule (15), triggered by the timer ini only once. Then, periodically using the timer $chNgb$ in Rule (16), extreme nodes are specified by using the routing table which has a route to all destination. The third step consists at finding the interval of each node. Rule (17) defines new intervals of intermediate nodes by checking the routing table and finding the new predecessor and successor, while Rule (18) and (19) are used to calculate the interval of extreme nodes. The minimum node, in Rule (18), sets its predecessor to the last node and finds its successor, and the maximum node in Rule (19) sets its successor to the first node and finds its predecessor. Finally, in Rule (20), each node updates its interval upon receiving a new interval.

The virtual ring can be seen on the next figure. The edges in black are the network edges, while red edges form the virtual ring.

	Interval	
Node	**Predecessor**	**Successor**
0000	0009	0001
0001	0000	0002
0002	0001	0003
0003	0002	0004
0004	0003	0005
0005	0004	0006
0006	0005	0007
0007	0006	0008
0008	0007	0009
0009	0008	0000



For reliability purposes, we assume that all data are replicated on two nodes, thus allowing any node to leave the network with no perturbation to the game. We assume that each node, say α, is responsible for the interval of values between its predecessor and successor node in the loop, $[pred_\alpha, succ_\alpha[$. The program REPLICATION ensures the replication of the data on the two nodes responsible for a fragment. It is shown here on the relation Owner, which has three attributes *Avatar*, *Auth* and *Client*, where *Avatar* is the index attribute.

Program REPLICATION

$\mathbf{module(updatecopy)}$

$$\uparrow OwnerUpd(@s,x,y,z) :- \begin{cases} TimeEvent('upd'), hv := hash(x), hv \geq \\ self, Intvl(p,s), Owner(x,y,z), hv < s. \end{cases} \quad (21)$$

$$\uparrow OwnerUpd(@s,x,y,z) :- \begin{cases} TimeEvent('upd'), hv := hash(x), hv \geq \\ self, Intvl(p,s), Owner(x,y,z), s < self. \end{cases} \quad (22)$$

module(**updatemove**)

$$:-\left\{\begin{array}{c} Intvl(op,os), hv := hash(x), hv < self, \\ !Owner(x,y,\_), !NIntvl(np,ns), np > op. \end{array}\right. \quad (23)$$

$$\uparrow Recover(@np,x,y,z) :-\left\{\begin{array}{c} Intvl(op,\_), !NIntvl(np,\_), hv < self, \\ Owner(x,y,z), hv := hash(x), np < op. \end{array}\right. \quad (24)$$

module(**updateOwner**)

$$\downarrow Owner(x,y,z) :-\Big\{ !OwnerUpd(self,x,y,z). \quad (25)$$

$$:-\left\{\begin{array}{c} !OwnerUpd(self,x,y,z), !Owner(r,s,m) \\ hv := hash(r), hv < self. \end{array}\right. \quad (26)$$

$$\downarrow Owner(x,y,z) :-\Big\{ Recover(self,x,y,z). \quad (27)$$

The program is composed of three main modules. Periodically using the timer upd in module $updatecopy$, each node, using Rules (21) and (22), sends a backup to be saved on its successor s for all entries that have index values in the interval $[self, s[$. Rule (22) is used to manage the last node and together with Rule (21) to prevent each node to save on its successor the backup of its predecessor.

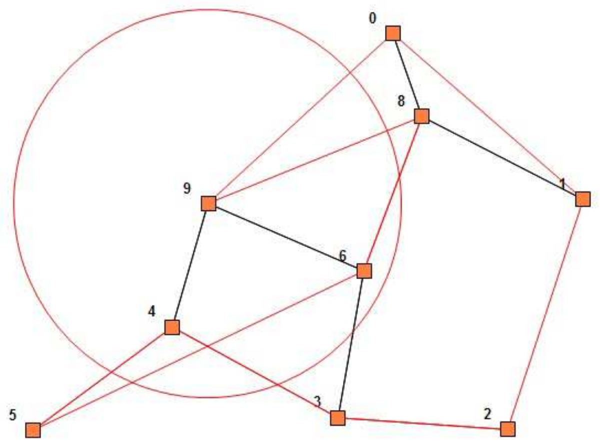
The module $updatemove$ with rules (23) and (24) is used to manage the backup copy in the successor upon the detection of any change in the predecessor. Indeed, if the new predecessor is greater than the old one, all entries that have index values less than $self$ are deleted, Rule (23). However, if old predecessor less than old one, that means the old predecessor leaves the network, and so all entries that have keys less then $self$ should be sent to new predecessor, Rule (24).

The backup and recover copy are sent using facts of the form $OwnerUpd(nodeId, avatar, authentication, owner)$ and $Recover(nodeId, avatar, authentication, owner)$. In the module $updateOwner$, the successor saves a backup copy upon receiving the fact $OwnerUpd(self, x, y, z)$ in Rule (25) and at the same time deletes the old copy, Rule (26), while new predecessor saves a recover copy upon receiving the fact $Recover(self, x, y, z)$ in Rule (27).

In the next example, we monitor the impact of the departure of node 7 from the network. Node 7 is in charge of fragments of relations whose index attribute is mapped to a value in the interval $[6, 8[$. The values of the index attributes Toto and Loulou are mapped by the hashing function respectively to 7, and 3. Node 7 is therefore responsible for the fragment of relation Owner with Avatar Toto as seen on the first Owner Table below. When node 7 disappears, its predecessor 6 takes the duty, and hosts the fragment, as shown on the second owner table. The figure shows the new Ring over the network after the departure of node 7.

	Owner		
Node	**Avatar**	**Auth**	**Client**
0007	Toto	37	0009
0008	Toto	37	0009
0003	Loulou	54	0005
0004	Loulou	54	0005

	Owner		
Node	**Avatar**	**Auth**	**Client**
0006	Toto	37	0009
0008	Toto	37	0009
0003	Loulou	54	0005
0004	Loulou	54	0005



The program SYNCHRONIZATION, performs the synchronization of the tables which have more than one index, such as the table Auction of the example, and whose data are therefore duplicated in as many pairs of copies as there are indexes. (This program is omitted in this draft for space reason).

Similar replication and synchronization programs are defined for all the tables of the application. The use of these programs is shown in the next section.

5 A Distributed Server for a multiplayer game

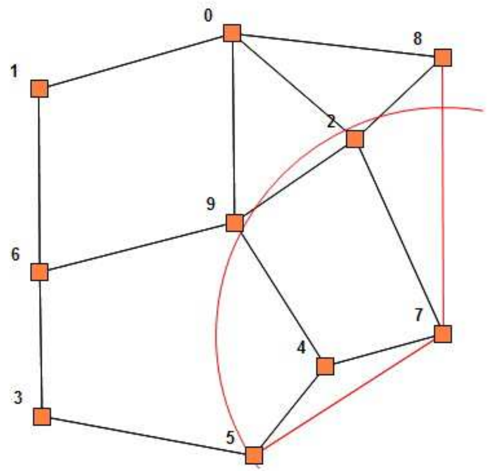
We have implemented the auction game in Netlog over the QuestMonitor system, and made experiments on dynamic networks of several dozen peers participating to the game. We describe below the execution of a simple scenario, where a node joins the network to participate to the game. After retrieving all the information about its avatar, it does a bid on an auction and finally disconnects from the network. At the expiration of the auction, the avatar owns the object, although the peer to whom the avatar belongs has already left the game.

To join the game, a node, say α, goes through four important steps:

- Authentication: Node α sends an authentication message to its neighbors. When the neighbors receive an authentication request, they verify if the authentication key of the avatar is correct, by sending the query to the node which stores the corresponding fragment of the Owner table. If the authentication succeeds, the routing algorithm is allowed to start.
- Route propagation: If node α is allowed to join the game, neighbors propagate their routes to node α and accept incoming route from it. The detailed workflow of DSDV is described in Section 4.
- Insertion in the ring: Node α has to insert itself into the DHT ring. The Ring protocol described in Section 4 is used to make it responsible for the interval $[pred_\alpha, succ_\alpha[$ of index values, and to update accordingly the nodes $pred_\alpha$ and $succ_\alpha$.

- Replication and synchronization: The change in the ring also triggers the replication and synchronization protocols. The predecessor and the successor of node α in the ring send to node α all entries (data about the distributed table) that have index values in the interval $[pred_\alpha, succ_\alpha[$. The new node is now responsible of some fragments of the distributed database.

The process begins as follows when a new node, $node$ 7, joins the game, which is already running between players. The communication between nodes is represented in red in the following figure.



After the authentication and the propagation of the route, $node$ 7 calculates its interval and saves as predecessor $node$ 6 and successor $node$ 8. It then receives all entries of the distributed table that have index values in its interval $[6, 8[$ from both its predecessor and its successor.

Suppose now that $node$ 6 has data with index values in the interval $]7, 8]$. The predecessor of the index values is the node responsible. Thus, $node$ 6 communicates with $node$ 7 to save the corresponding data, and then saves a backup in its successor $node$ 8.

After $node$ 7 has entered the game, it has to retrieve all the information about its avatar (list of owned objects, bank account, etc.) as well as its geographic views (such as the list of auctions on markets where it registers). It uses the index attributes to communicate with nodes that are responsible of the fragments of interest to it. When the node has retrieved all the information about its avatar, and has registered to a market, it is able to bid at an auction. It updates its local view of the table *Auction* and then sends an update (client action 3 in Section 2) to the node responsible of the market. When this node receives the update query, it updates the table *Auction* and the backup copy in its successor (using the replication protocol).

The node can move in the physical space, and so change position in the network. Thanks to the DSDV program, it is always reachable, but the DHT ring is not modified.

Every message sent by a node is composed of an explicit and an implicit destination. The explicit destination is the node responsible of the index values and the implicit destination is a query over relation $Intvl$ that can compute a new destination if the explicit destination becomes unreachable.

The DSDV program updates its routes every three seconds. If a node α sends a message to a disconnected node β, a hop in the path between α and β will eventually be missing and the implicit destination to find a new explicit destination will be evaluated. The new destination is the node newly responsible of the index value. Thus, data in the message is not lost.

Suppose now that the node disconnects before the end of the auction. The DSDV program detects that the node is no more in the network and the route to this node expires and is deleted.

When the route is deleted, the ring, as well as the duplication and replication of the data fragments are updated as seen in Section 4.

When the auction expires, the node responsible of the market where the auction is stored has to execute some system actions. These actions are SQL queries on the distributed tables. They consist of a transaction of SQL queries: (i) update the bank account of the seller and the buyer; (ii) update the owner of the object; and (iii) delete the auction from table *Auction*

To update the bank accounts, the node responsible of the auction, say λ, does a join between the fragments of two tables *Auction* and *Market*, which are on the same node because they have the same index attribute, in order to retrieve the account of the avatar. Then, after hashing the account of each avatar, it sends two messages with $explicit$ and $implicit$ destination to nodes responsible (say β, γ for the seller and the buyer respectively) in order to update their balance.

Given the restrictions on the queries, messages are never broadcasted, but unicasted thanks to the index attribute of the DHT table. An important feature of our approach, supported by the Nequest machine, is the use of a pair of $explicit/implicit$ destination at the same time, to handle the possibility that the destination node leaves the network.

To update the object, node λ hashes the object OId from its local table $Auction$ and sends a message with $explicit$ and $implicit$ destinations to the node responsible, say θ. Upon receiving the message, node θ updates the entry related to object OId.

Finally, to delete the auction, node λ directly removes the entry of the auction on the local fragment of the auction table, and updates the replicated version and synchronizes with the protocol Replication and Synchronization.

In the simple game we have considered, the join is done on one single node. More complex distributed joins can be handled as well by the present approach, but require additional protocols to carry on the distribution.

6 Conclusion

Data centric languages facilitate the writing of distributed programs, resulting in much shorter and more declarative code. We have shown that networking protocols could be written as simple, very concise, programs consisting of a few dozen of Netlog rules. We described the rules coding for a DHT in a mobile ad hoc network relying on a DSDV like routing protocol. We then showed that applications coded as queries in a client-server framework could be ported seamlessly, that is without modifying the initial queries, to the distributed environment. The distributed system based on the DHT ensures the tasks of the centralized server in a fully distributed manner, by relying in the peers which handle horizontal fragments of the relations, and communicate with other peers to solve queries. We considered the promising example of multiplayer online games, which can be fully described in a data centric fashion, and showed how it can be seamlessly distributed. The experiments we made on the QuestMonitor platform, with games of over 50 players, demonstrated the robustness of the approach.

We plan to further investigate the protocols supporting peer-to-peer applications and the distribution of the server tasks, by considering other DHTs, as well as additional issues, such as security and real time which are critical. We also plan to verify formally the DHT, and extends our results to ensure the persistence of the game under more stringent conditions, such as the movements of several nodes at a time. It has been shown that the data centric approach facilitates the verification of the programs. Simple Netlog protocols have already been verified [9] using the Coq proof assistant.

Acknowledgment

The authors would like to thank Ben Leong for suggesting us to consider fully distributed multiplayer games, as well as Christophe Bobineau, Christine Collet, Fuda Ma, and Stéphane Ubéda, for fruitful discussions.

References

[1] F. Bancilhon. Naive evaluation of recursively defined relations. In *On knowledge base management systems: integrating artificial intelligence and database technologies*, 1986.

[2] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1986. ACM.

[3] M. Bauderon, S. Grumbach, D. Gu, X. Qi, W. Qu, K. Suo, and Y. Zhang. Programming imote networks made easy. In *The Fourth International Conference on Sensor Technologies and Applications*, pages 539–544. IEEE Computer Society, 2010.

[4] E. Bellemon, V. Dubosclard, S. Grumbach, and K. Suo. Questmonitor: A visualization platform for declarative network protocols. In *MSV 2011: The 8th International Conference on Modeling, Simulation and Visualization Methods, Las Vegas, USA*, 2011.

[5] S. Buchegger, D. Schiöberg, L.-H. Vu, and A. Datta. Peerson: P2p social networking: early experiences and insights. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*, 2009.

[6] M. Caesar, M. Castro, E. B. Nightingale, G. O'Shea, and A. Rowstron. Virtual ring routing: network routing inspired by dhts. *SIGCOMM Comput. Commun. Rev.*, 36:351–362, 2006.

[7] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. *SIGOPS Oper. Syst. Rev.*, 36, December 2002.

[8] D. Chopra, H. Schulzrinne, E. Marocco, and E. Ivov. Peer-to-peer overlays for real-time communication: Security issues and solutions. *IEEE Communications Surveys and Tutorials*, 11(1), 2009.

[9] Y. Deng, S. Grumbach, and J.-F. Monin. A framework for verifying data-centric protocols. In *FORTE 2011: The 31th IFIP International Conference on FORmal TEchniques for Networked and Distributed Systems*, Reykjavik, Iceland, 2011.

[10] S. Grumbach and F. Wang. Netlog, a rule-based language for distributed programming. In *PADL'10, Twelfth International Symposium on Practical Aspects of Declarative Languages, Madrid, Spain, january*, 2010.

[11] T. Hampel, T. Bopp, and R. Hinn. A peer-to-peer architecture for massive multiplayer online games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames '06, 2006.

[12] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.

[13] P. Jacquet, P. Muhlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot. Optimized link state routing protocol for ad hoc networks. In *Multi Topic Conference, 2001. IEEE INMIC 2001. Technology for the 21st Century. Proceedings. IEEE International*, pages 62–68, 2001.

[14] B. Knutsson, M. M. Games, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM*, 2004.

[15] J. Liu, S. G. Rao, B. Li, and H. Zhang. Opportunities and challenges of peer-to-peer internet video broadcast. *Special Issue on Recent Advances in Distributed Multimedia Communications, Vol. 96, No. 1, pp. 11-24*, 2008.

[16] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proc. ACM SIGMOD'06*, 2006.

[17] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proc. SOSP'05*, 2005.

[18] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *Proc. ACM SIGCOMM '05*, 2005.

[19] C. man Au Yeung, I. Liccardi, K. Lu, O. Seneviratne, and T. Berners-Lee. Decentralization: The future of online social networking. In *W3C Workshop on the Future of Social Networking Position Papers*, 2009.

[20] C. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. In *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.

[21] C. E. Perkins. Ad-hoc on-demand distance vector routing. In *In Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, 1999.

[22] P. Persson. Exms: an animated and avatar-based messaging system for expressive peer communication. In *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*, GROUP '03, 2003.

[23] R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23:125–149, 1993.

[24] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[25] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31, 2001.

[26] L. Vieille. Recursive axioms in deductive databases: The query/subquery approach. In *Expert Database Conf.*, pages 253–267, 1986.