A Bibliographical Survey on Data Sharing Systems for Mobile Ad Hoc Networks

Etude bibliographique sur les systèmes de partage de données sur réseaux mobile ad hoc

---

Ludovic Martin
Isabelle Demeure

2006D009

décembre 2006

Département Informatique et Réseaux
Groupe Systèmes, Logiciels, Services

Dépôt légal : 2007 – 1er trimestre
Imprimé à l'Ecole Nationale Supérieure des Télécommunications – Paris
ISSN 0751-1345 ENST D (Paris) (France 1983-9999)

Ecole Nationale Supérieure des Télécommunications
Groupe des Ecoles des Télécommunications - membre de ParisTech
46, rue Barrault - 75634 Paris Cedex 13 - Tél. + 33 (0)1 45 81 77 77 - www.enst.fr
Département INFRES

THALES

BIBLIOGRAPHICAL SURVEY ON

DATA SHARING SYSTEMS FOR MOBILE AD HOC NETWORKS

ÉTUDE BIBLIOGRAPHIQUE SUR LES

SYSTEMES DE PARTAGE DE DONNEES SUR RESEAUX MOBILE AD HOC

Ludovic MARTIN

ludovic.martin@enst.fr
ENST Dépt Informatique et Réseaux
46 rue Barrault, 75013 Paris

ludovic.martin@fr.thalesgroup.com
THALES LAND & JOINT SYSTEMS
160 bd de Valmy - BP 82, 92704 Colombes Cedex

Isabelle DEMEURE

isabelle.demeure@enst.fr
ENST Dépt Informatique et Réseaux
46 rue Barrault, 75013 Paris

Rapport technique
Décembre 2006

ABSTRACT

In this report we study existing data-sharing systems for MANETs (Mobile Ad hoc NETworks). We identify eleven candidates data sharing systems: LIME, LIMONE, TOTA, Coda, Ficus, Rumor, Roam, AdHocFS, Ad-Hoc InfoWare, PeerWare and XMIDDLE. Some were specifically designed for MANETs, others target mobile systems but exhibit properties that may be of interest in MANETs. All systems are analyzed in turn following the same analysis chart in order to clearly identify the major functional and non-functional properties of a data sharing system for MANETs. We then do a comparative study of the 11 systems and identify the required properties of an "ideal" data sharing system for MANETs.

RESUME

Dans cette étude, nous analysons des systèmes de partage de données conçus pour les réseaux mobiles ad hoc, appelés MANETs (Mobile Ad hoc NETworks). Nous identifions onze solutions de partage de données candidates : LIME, LIMONE, TOTA, Coda, Ficus, Rumor, Roam, AdHocFS, Ad-Hoc InfoWare, PeerWare and XMIDDLE. Parmi ces systèmes, certains ont été spécialement conçus pour les MANETs, d'autres ciblent des réseaux mobiles et filaires. Ces derniers illustrent des propriétés qui peuvent être intéressantes sur des MANETs. L'examen de chacun des systèmes est basé sur une grille d'analyse commune. Celle-ci identifie clairement les principales propriétés fonctionnelles et non fonctionnelles du partage de données au sein d'un MANET. A partir de cette analyse, nous comparons les 11 systèmes et nous identifions les mécanismes à inclure dans un système de partage de données sur MANET "idéal".

TABLE OF CONTENTS

Abstract i

Table of contents ii

Survey 1

1 Introduction 1

2 Analysis chart and key concepts 2

2.1 Functional properties of a data sharing middleware 3

2.1.1 Data: type, structure and identification 3

2.1.2 Host identifier, presence and communities management 3

2.1.3 Data discovery and searching 3

2.1.4 Basic data management functions 4

2.2 Non-functional properties of a data sharing middleware 4

2.2.1 Data management 4

2.2.2 Resource management in a limited capabilities environment 6

2.2.3 Security 6

2.3 Architecture 7

2.3.1 Host model 7

2.3.2 Communication model 7

3 Existing systems 8

3.1 Tuple-based systems 8

3.1.1 Introduction to the Linda communication model 8

3.1.2 LIME 9

3.1.3 LIMONE 12

3.1.4 TOTA 16

3.1.5 Synthesis on tuple-based systems 19

3.2 Generic data based systems 20

3.2.1 Coda 20

3.2.2 Ficus, Rumor and Roam 23

3.2.3 AdHocFS 27

3.2.4 Ad-hoc InfoWare 31

3.3 Structured data based systems 34

3.3.1 PeerWare 35

3.3.2 XMIDDLE 37

4 Comparison and discussion 41

4.1 System target context 41

4.2 Comparison 42

4.2.1 Functional properties 42

4.2.2 Non-functional properties 43

4.3 Discussion 45

5 Synthesis and conclusion 46

Acknowledgements 48

References 49

Annex: Synthetic comparison 54

SURVEY

1 Introduction

In this paper, we survey the data sharing systems for mobile ad hoc networks. These systems aim to provide a shared space over which a group of hosts can work together on common data.

MANET (Mobile Ad hoc NETwork) is not a new idea [CoM99]. Research began in the early 70's with the DARPA Packet Radio Network (PRNet) project, few years after the introduction of packet switching technology with the ARPANET (1969). In fact, the Departure of Defense immediately understood the potential of ad hoc wireless networks for interconnecting mobile nodes in the battlefield where a wired or cellular structure is lacking or is unusable (e.g. controlled by the enemy). Recently, the rapid advances of radio technologies such as Bluetooth and WiFi, and the spread of mobile devices such as laptops and personal digital assistants equipped with wireless interface(s) are promoting MANET deployment outside the military domain. These recent evolutions are generating a renewed and growing interest in research and development around MANETs. In [CCL03], Chlamtac and al. provide a good overview of this dynamic field.

As defined in [CM90] and [CCL03], MANETs are infrastructure-less networks composed of mobile devices with limited resources and which may or may not be connected to the Internet. In a MANET, the devices must therefore organize themselves in order to create the network. They cannot rely on a pre-configured infrastructure. This is both an advantage and a drawback. The main advantage is the possibility of spontaneously deploying the network anywhere. On the other hand, the design of a data sharing system in this environment must face a number of challenges such as:

- *Communication unreliability*. The wireless medium is significantly less reliable than the wired media.
- *Bandwidth-constrained*, variable capacity links. In spite of advances in radio technologies, wireless links still have significantly lower capacity than their wired counterparts. Moreover, their capacity is time-varying and they can be asymmetric.
- *Limited capacities of handheld devices*. Participants may access the network through small devices with limited resources mainly in terms of battery, memory, and CPU.
- *Dynamic topologies*. Devices are mobile. They may enter and leave the network at anytime. As a consequence, the network topology may change frequently and unpredictably. This may lead to network partitions "meaning that there is currently no sequence of nodes through which a packet could be forwarded to reach the destination" [JMH04].
- *Autonomous and infrastructure-less*. The lack of infrastructure and the dynamically changing topology leads to reconsider the use of traditional client/server architectures (since there is no guarantee that a server would be accessible during a session). This is why the alternative *peer-to-*

peer (P2P) architecture is often used in the systems analyzed in this report

- *Limited physical security*. Mobile wireless networks are generally more prone to physical security threats than fixed-cable nets. The increased possibility of eavesdropping, spoofing, and denial-of-service attacks should be carefully considered.

In this paper, we present eleven data sharing systems. We group them in three categories according to the type of data they manipulate. In fact, the way the information is set can heavily influence the properties of a system and the behavior of users. Therefore, we distinguish the tuple-based systems, the generic data based systems and the structured data based systems.

Tuple-based systems were first introduced in the Linda communication model [Gel85]. A tuple is a sequence of typed fields such as ⟨«abc» ; 52 ; 3.21⟩. LIME [MPR99], LIMONE [FRH04] and TOTA [MaZ03] are based on tuples. LIME and LIMONE are quite similar: they use the Linda primitives and they provide common services. However, LIMONE is based on much lighter protocols and is more adapted to the ad hoc environment. TOTA is quite different. It was designed for a very dynamic network. The goal of TOTA is to simulate some force fields through tuples.

The second group of systems is called the generic data based systems. This name comes from the fact that most data-sharing systems manipulate generic data which correspond to data blobs having an implicit structure neither recognized nor interpreted by the system but only by upper applications. Generally, these data are files. This is the case for Coda [KS92], Ficus [GHM+90], Rumor [GRR+98], Roam [RRP04] and AdHocFS [Bou03]. The only exception is Ad-hoc InfoWare [PAD+04] which can manipulate any kind of object containing a piece of information. Coda and Ficus are historical systems that were among the first to address mobility and partitioned networks. Rumor and Roam are improvements of Ficus. AdHocFS is a file system that provides two levels of consistency and takes particular care of limited capacities of handheld devices. InfoWare aims at sharing information between rescue personnel from various organizations in emergency and rescue situations.

The third group of systems is the structured data systems. Structured data have an explicit and expressive constitution that is recognized and interpreted by systems using them. Systems like XMIDDLE [MCZ+02] and PeerWare [CuP01] can thus offer specific or more adapted services. PeerWare provides its own structured data. This one is very simple. On the contrary, XMIDDLE uses the successful XML document format. Its flexible structure provides finer control over data and adds semantic to the content.

The remainder of this paper is organized as follows. In Section 2, we explain the outline used for describing all systems. We provide definitions of several key concepts. In Section 3, we present some existing data sharing systems dealing with mobility. Some were specifically designed for MANETs, others target mobile systems but exhibit properties that may be of interest in MANETs. In Section 4, we compare these systems and try to point out missing functionalities. This synthesis is greatly aided by the strict methodology introduced in Section 2. Finally, Section 5 concludes.

2 Analysis chart and key concepts

In this section, we introduce the concepts and properties which are to be considered for data sharing systems in mobile ad hoc networks. For this, we follow an outline that will be reused when describing the systems in

Section 3. When necessary, we provide precise definitions of key concepts.

In chapter 2.1, we present the functional properties, namely: "data type and identification", "host identification, presence and communities management", "Data discovery and searching", base functions for data management, data searching". Then, chapter 2.2 deals with non-functional properties, namely: data management, resource management in a limited capability environment and security. Chapter 2.3 concludes with some notions of architecture.

2.1 Functional properties of a data sharing middleware

2.1.1 Data: type, structure and identification

Depending on systems, the data can be typed or not. These types can even be hierarchically classified as in object-oriented programming. Such an approach is used in TOTA to help data processing.

Being typed or not a data container is always structured to be usable by the final application. This structure may be known by the data sharing system. The advantage is that the system can provide more efficient and/or specific services. The structure also determines the sharing unit if it enables to split a data into several parts.

The identifier enables users to access a data. It must be unique, at least in a given ‘domain’.

2.1.2 Host identifier, presence and communities management

A host is a node of the network. It corresponds to a user's mobile device.

When a host enters a MANET it gets a unique identifier. In order to allow point-to-point communications, a presence service maintains, on each host, a list of all accessible hosts. Since modifications in this list indicate changes of context, hosts may be informed of them. Notice that systems also often provide primitives to switch off/on the presence service (e.g. for energy saving).

Finally, hosts can be interested in gathering into *communities of interest* (CoIs). All CoI members share a common property (a subject of interest, a class of employees, etc…). An advantage of CoI is that it may provide a degree of anonymity since members are guaranteed to find “interesting” data and do not need to know each other. Another advantage is that it can improve efficiency of the system by limiting the number of involved hosts. Hosts may participate to several CoI or may move from one CoI to another one.

2.1.3 Data discovery and searching

In MANETs, hosts may not have a complete view of all shared data. They are restricted to the data stored on accessible hosts. There are three ways to discover accessible data:

1. A pre-configured static list is defined inside the system. Entries in this list can be either links to data or data copies.
2. A dynamic local list is maintained by the system. Updates to this list can be notified to the host.
3. The system does not maintain a list. In this case, data searching is performed on demand over accessible hosts (or a subset of them).

The main drawback of the second solution (contrary to the third one) is that maintaining the list is costly in bandwidth, energy and memory even if it can be combined with other data sharing services (e.g. replication). In return, data searching is quick and inexpensive since there is no need to contact remote hosts.

The expressiveness power of data searching languages greatly varies between systems. The research may be based on the content of data or on some meta-information such as file name and ontology.

2.1.4 Basic data management functions

Once hosts are able to discover each other and to communicate with one another, they need to manipulate data. The basic data management functions found in existing data sharing systems are:

- Shared data may be created within the data sharing system or outside of it. In the first case, a shared data creation function is needed. In the second case, data must be made visible from other users and are usable by them. Upon this step, users may be asked to specify some options like access rights and location storage.
- Data reading, data modifying functions.
- Data removing is often limited to local replicas removal.
- Synchronization primitives can be useful to keep data coherent or processes synchronized.

2.2 Non-functional properties of a data sharing middleware

2.2.1 Data management

2.2.1.1 Consistency

There are two kinds of consistency models: the *pessimistic* ones which do not allow replica divergence and the *optimistic* ones where divergences are possible and must be reconciled.

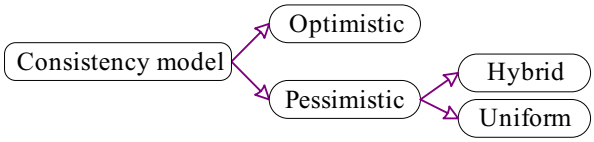


Figure 2-1: Quick classification of consistency models

2.2.1.1.1 Pessimistic models

A huge work on pessimistic consistency has been realized for DSMs (Distributed Shared Memories) [Mos93], [Cor97]. A DSM is a system which gives to a distributed application the illusion that participating processes share a common memory. In this context, Censier and Feautrier defined, in 1978, the consistency like this: “A memory is said consistent if the value returned by a read operation at a memory’s location is always the value corresponding to the last write in this location”.

As shown in Figure 2-1, there are two types of pessimistic models: hybrid and uniform. The *uniform* ones are also called *non-synchronized* because they only use read and write primitives (i.e. they do not use synchronization primitives). The consistency is wholly assumed by the system and not by the programmer. In the contrary, the *hybrid* pessimistic consistency models are also called *synchronized* because their implementations provide synchronization primitive(s) to the programmer who must deal with consistency.

♦ *Uniform models*

In [Mos93], David Mosberger reviews 7 uniform models. The strictest is the *atomic consistency* which behaves exactly like a centralized memory [CeF78]. It is a theoretical model never implemented.

The *sequential consistency*, proposed by Lamport [Lam79], is implemented by several systems. It is still rather strict. It guarantees that all processes see operations in the same order. In fact, read and write operations are logically ordered but not chronologically. So a read access may not return the value corresponding to the last (in time) write. This situation is illustrated in Figure 2-2.

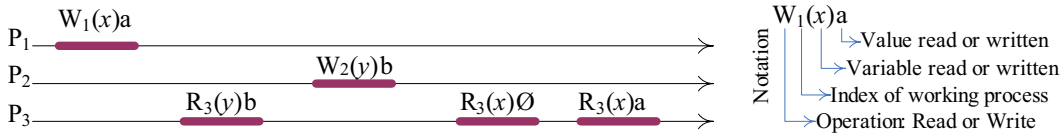


Figure 2-2: Sequential consistency (example)

This example is sequentially but not atomically consistent. The logical order observed by all processes (P_1, P_2 and P_3) is: $W_2(y)b - R_3(y)b - R_3(x)Ø - W_1(x)a - R_3(x)a$. Clearly, it is different from the chronological order.

♦ *Hybrid models*

Hybrid pessimistic consistency allows a programmer to decide himself where critical section must be synchronized. Since other accesses are not monitored, such a system is much more efficient. The three main hybrid models are the *weak consistency*, the *release consistency* and the *entry consistency*. We do not detail them here but interested readers may refer to [Mos93].

2.2.1.1.2 Optimistic models

The key feature that separates optimistic consistency models from their pessimistic counterparts is their approach to concurrency operations. Pessimistic models (both uniform and hybrid) synchronously coordinate replicas during accesses and block other users during an update. Optimistic models let data be accessed without, a priori, synchronization based on the "optimistic" assumption that problems will occur only rarely. Updates are propagated in the background, and occasional conflicts are fixed after they happen.

In fact, optimistic models have good performance and availability in spite of network partition. However, these benefits come at a cost. The reconciliation algorithm is a difficult problem.

The most used optimistic models are the *close-to-open* and *read-your-writes* ones. The close-to-open model is historically used in file system like NFS. It guarantees that:

- When a user opens a file, it gets the most recent version of the file recorded by a close.
- As long as the file is maintained opened, the user only perceives its own updates.

Figure 2-3 shows an example of execution, which is consistent in close-to-open model.

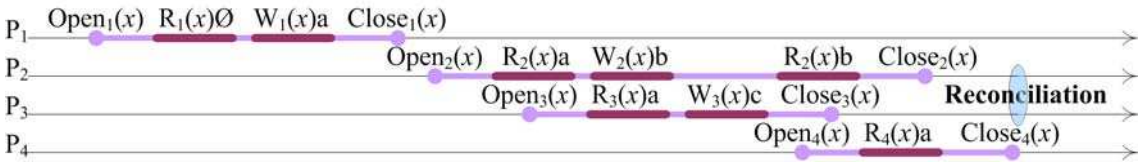


Figure 2-3: Close-to-open consistency (example)

The read-your-writes model was originally introduced by Bayou [DPS+94]. It is less strict than the previous model since it simply guarantees that when a user opens a file, the version of the file that it reads is not older than the version it previously accessed. Once the file is opened, file updates are performed as in the close-to-open model. The schema described in Figure 2-4 is possible (but not unique) in a read-your-writes system.

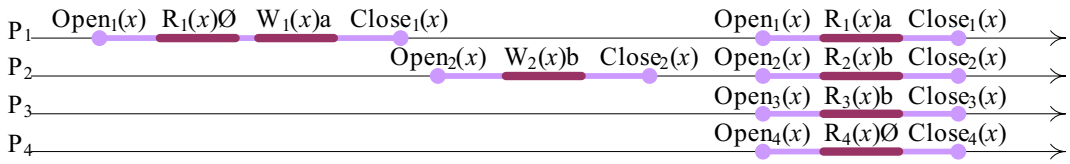


Figure 2-4: Read-your-writes consistency (example)

2.2.1.2 Persistency

Generally speaking, persistency is the characteristic of data that outlives the execution of the program that created it. It is traditionally achieved by storing the data in non-volatile storage (e.g. a file system) in order to prevent data loss when a program exits. An additional problem in MANETs is that data are locally stored on hosts, which can be switched off at anytime: persistency is therefore a real issue.

In addition, in MANET, hosts behave independently from each others. So, when one of them decides to delete a copy of a data (e.g. in order to free memory space), the system should verify that it is not the last copy of this data. In fact, the system should control the number of replicas for each data in order to bear unpredictable copy disappearances. Another constraint is to lose as few updates as possible.

2.2.1.3 Accessibility in mobile ad hoc network context

One of the problems to be faced in MANETs is network partitioning. When the network is partitioned, accessibility may be possible:

- if data are replicated,
- if the replication is such that a replica exists in each partition where a host may access the data,
- and if the consistency model enforced authorizes concurrent operations on replicas belonging to different partitions.

2.2.2 Resource management in a limited capabilities environment

As MANETs are characterized by heterogeneous mobile devices with constrained resources, they should be optimized both locally (i.e. on each host) and globally (i.e. across the entire network). Memory space is crucial for data sharing. An interesting problem is the case of heavy files that can not be stored on a single device. In this case, the data may be split over several hosts. This distribution and the possible replication (to improve persistency and accessibility) of the parts can be done in several possible ways. In the mobile ad hoc context, two other resources are important: the energy and the bandwidth. A positive point is that minimizing the latter automatically saves the former, since, according to [AMR03], in a mobile device, the wireless interface consumes about a third of the system power. Reducing the number and the volume of communications should therefore be a good way to reduce energy consumption. Another energy saving possibility is to simply switch off the wireless interface for a short period [FeN01].

Middleware can also take into account other resources like computational power, which may preclude the use of efficient but too complex algorithms.

2.2.3 Security

In a shared environment, security is crucial, even if more difficult to realize in MANETs than in structured networks. The two most important services are authentication and data access control. The former provides some

degrees of trust between hosts. The latter allows granting data access to remote host requiring access to the data. Some systems also support private space(s). Other security services are confidentiality, integrity and non-repudiation. The latter is rarely addressed in data sharing.

2.3 Architecture

In the remainder of this paper we give a brief overview of the architecture of each system introduced. We do it from two perspectives: host and communication model.

2.3.1 Host model

The host model describes the interactions between components and layers on a given host. Figure 2-5 extracted from [Bak] shows a generic host architecture. In fact, all data sharing systems presented in this paper are located at the middleware layer. They mask heterogeneity of networks, hardware and operating systems. So they are situated above the operating system and below the application layer.

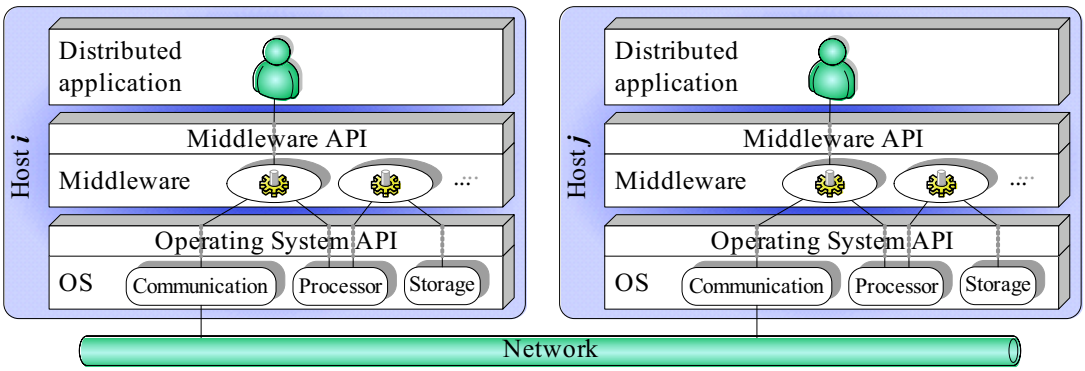


Figure 2-5: Middleware layer

2.3.2 Communication model

The communication model deals with host interaction. The traditional *client-server* model is unsuitable to MANETs since hosts cannot rely on a central entity (this entity may become out of reach or may be switched off). Instead, they must work in a *peer-to-peer* (P2P) fashion where they are both producers and consumers of the data sharing service [Sch01]. Figure 2-6 illustrates the three existing kinds of P2P architectures:

- A *hybrid* P2P architecture (Figure 2-6[a]) uses a central entity for auxiliary service(s) (e.g. security, indexing…). Some systems use this model when the auxiliary service does not require permanent connection to its server.
- A *pure* P2P architecture does not rely on any server. Any entity, called *peer*, which participates to the service can be removed without interrupting the service. Two sub-types are distinguishable:
    - In a *flat*[1] pure P2P architecture (Figure 2-6[b]), all peers have exactly the same role. Although, it does not scale very well, it is the most used model in the presented systems.

[1] This terminology is peculiar to this paper. There is no commonly accepted words to distinguish “flat” and “hierarchical” pure P2P architectures.

- In a *hierarchical* pure P2P architecture (Figure 2-6[c]), some peers, called *super-peers*, have additional functions, which improves global service performances. The difference between a server and a super-peer is that the later is replaceable.

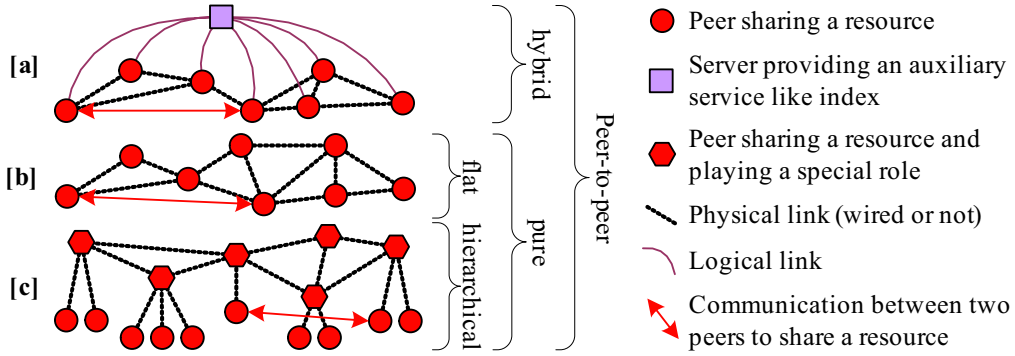


Figure 2-6: Peer-to-peer architectures

3 Existing systems

3.1 Tuple-based systems

Many information sharing systems use *tuples*. This concept was introduced by the Linda communication model. That is why Section 3.1.1 briefly introduces Linda although it does not deal with the subject of this document. Subsequent sections present LIME (3.1.2), LIMONE (3.1.3) and TOTA (3.1.4).

3.1.1 Introduction to the Linda communication model

3.1.1.1 Features

All systems based on tuples stem from the Linda communication model proposed by David Gelertner in 1985 [Gel85]. Linda is a distributed programming language. Its main characteristic is to provide an asynchronous communication mode based on a virtual environment called *tuple space*.

A tuple is a sequence of typed fields such as ⟨«ab» ; 52 ; π⟩. The two main properties of the tuple space are:

- Global accessibility: all processes can access a tuple space from anywhere.
- Persistency: the tuple space is a persistent space independent of the processes that access it. So a tuple that is put into the tuple space remains there until it is explicitly removed.

Interprocess communication is made through the tuple space. For example, after process α writes a tuple to the tuple space, process β may read this tuple when it wishes to. To exchange information, senders and receivers need neither to be in communication nor to know their mutual identity.

These properties (global accessibility, persistent storage and asynchronous communication) make Linda a very powerful data sharing model. Moreover, it provides innovative primitives which directly integrate the consistency proptocol.

3.1.1.2 Specific primitives

Linda interface is very simple. Tuple space access is made through the three following primitives[1]:

- **out**(*t*) writes tuple *t* to the tuple space;
- **in**(*p*) reads and removes from the tuple space a tuple whose parameters match to the pattern *p*;
- **rd**(*p*) reads a tuple but does not remove it from the tuple space.

The primitives **in** and **rd** are blocking: if no matching tuple is available in the tuple space, the process performing these operations is suspended until a matching tuple becomes available. Since it is not always desirable, a typical extension to this synchronous model is the provision of a pair of asynchronous primitives **inp**(*p*) and **rdp**(*p*), called *probes*, that allow non-blocking access to the tuple space.

If multiple tuples match a pattern *p*, **in** and **rd** select one non-deterministically and without being subject to any fairness constraint. Some variants of Linda (e.g. [Row98]) also provide *bulk* operations that can be used to retrieve all matching tuples in one step. Usually, systems only provide a non-blocking version for these operations. In this case, they are called **ing**(*p*) and **rdg**(*p*). But when the two variants are supplied, **ing**(*p*) and **rdg**(*p*) are blocking whereas **ingp**(*p*) and **rdgp**(*p*) are not.

Finally, it must be noted that Linda does not provide a primitive for modifying tuples. The modification must be done in two steps: first, withdraw the tuple (**in**) and then, rebuild it (**out**).

3.1.2 LIME

Lime [LIME] [MPR99] [MPR03] is the result of a joint research effort led by Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman, and has been carried out for the most part at Washington University in St. Louis, MO, USA. The computer science departments at Politecnico di Milano, Italy, also participated to the project. The first publication is dated from 1999 and the last one from 2003. LIME has been implemented in Java. Its sources have been available on SourceForge.net [LIME] through the GNU Lesser General Public License since 2000.

LIME (Linda in a Mobile Environment) is a coordination middleware where distributed processes can exchange information. LIME adapts the Linda communication model to support ad hoc networks. Processes are executed on physically mobile hosts (e.g. a PDA) and can themselves logically move from one host to another[2].

3.1.2.1 Functionalities

♦ *Data: type, structure and identification*

Being based on the Linda model, LIME uses tuples to transport information. Following Linda, it does not impose any special structure inside tuple fields.

The sharing unit is the tuple. Its size is variable and it is indivisible. It means that it is impossible to share only a part of its fields. It also means that a process, which works on a tuple, must manipulate this one wholly. It can't just access some particular fields.

In a LIME *Federated Tuple Space,* which is a sort of Linda-like tuple space, tuples are identified by their

[1] Linda implementations typically include also an **eval** operation which provides dynamic process creation and enables deferred evaluation of tuple fields. However none of studied systems use it

[2] In the LIME terminology, because of their logical mobility, processes are called *agents*. In this document, we will avoid to use this too much connoted word.

content. The identification of Federated Tuple Spaces is made by a unique name. The available published papers do not explain how the uniqueness of these names is guaranteed.

♦ *Host identifier, presence and communities management*

A process is identified, on the one hand, by the identifier of the LIME server that created it, and, on the other hand, by an increasing number assigned by this server. The LIME server is the entity where processes are executed. A host can run several servers[1]. This is why a LIME server is identified by both the IP address of its host and the port number that it uses to load processes.

The presence service described in [RHH01] detects all hosts in the same partition. So LIME allows multi-hop communications. Each accessible host and process is represented by a tuple in a special Federated Tuple Space called **`LimeSystem`**[2] (not shown in Figure 3-1). Disappeared hosts and processes are also indicated in **`LimeSystem`**.

LIME provides a mechanism of *reactions*. A reaction **A.**$\mathbb{R}(s,p)$ executes the *s* fragment code when a tuple matching the *p* pattern appears in the Federated Tuple Space **A**. By this way, it is possible to detect host and process (dis)appearance.

LIME provides communities of interest through the Federated Tuple Spaces (FTSs). A process can access as many FTSs as it wants as shown in Figure 3-1. It is possible to dedicate each FTS to a specific subject. All accessible FTSs are in **`LimeSystem`** (by tuples).

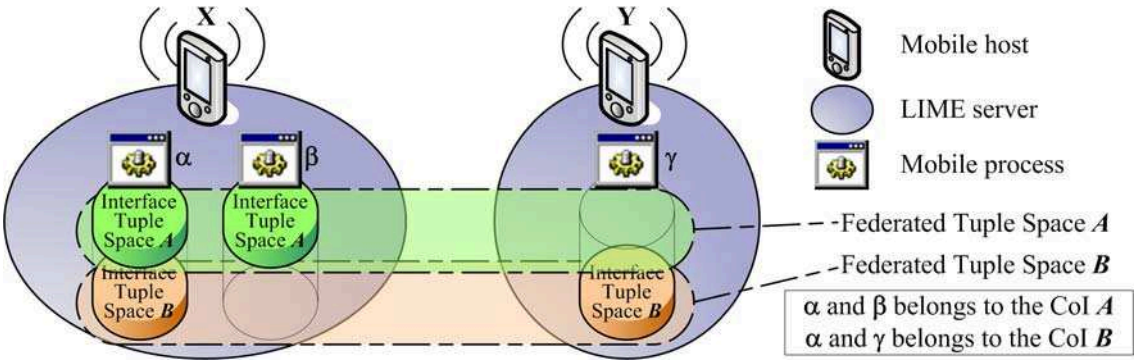


Figure 3-1: LIME communities

In order to save energy, hosts can switch off their presence service and stop their participation.

♦ *Data discovery and searching*

LIME doesn't maintain a list of accessible tuples. However, thanks to the reactions, processes can be notified of the appearance of a desired tuple in a given Federated Tuple Space.

There are two types of reactions. The *strong reactions* are atomic. They are very powerful since they detect a context change and execute *s* in one step. In return, this atomicity is quite restricting. It limits the range of the strong reactions to co-located processes (i.e. hosted on the same host). The *weak reactions* do not guarantee this atomicity. However, the fragment code *s* can interact with all entities of the network. For stability reasons, all blocking operations are forbidden in reactions.

LIME provides a powerful mechanism to search information: the pattern matching. This mechanism is

[1] In order to simplify this document, we only consider the case where a host has only one server.
[2] The Federated Tuple Space **`LimeSystem`** is a special one: on the one hand, for the sake of system stability, it is only readable; on the other hand, it is wholly replicated on each host.

directly integrated into reactions and access primitives (see next section).

♦ *Basic data management functions*

LIME uses the Linda primitives as well as some new primitives to better manage the LIME tuple spaces. In fact, in order to support mobility, LIME breaks the global Linda tuple space into multiple 'sub-tuple-spaces' as shown in Figure 3-2. The smallest sub-tuple space is the ITS (*Interface Tuple Space*). An ITS is always attached to the same process and contains all tuples that this process wants to share with other processes. The union of the ITSs of co-located processes is called a *Host-Level Tuple Space*. Finally the Federated Tuple Space merges together the Host-Level Tuple Spaces of all connected hosts (in the same network partition).

The Federated Tuple Space roughly corresponds to the Linda tuple space except that it is dynamic and that its content evolves according to arrivals / departures of hosts and processes. A quite important consequence is that the Federated Tuple Space is not persistent (contrary to the Linda tuple space).

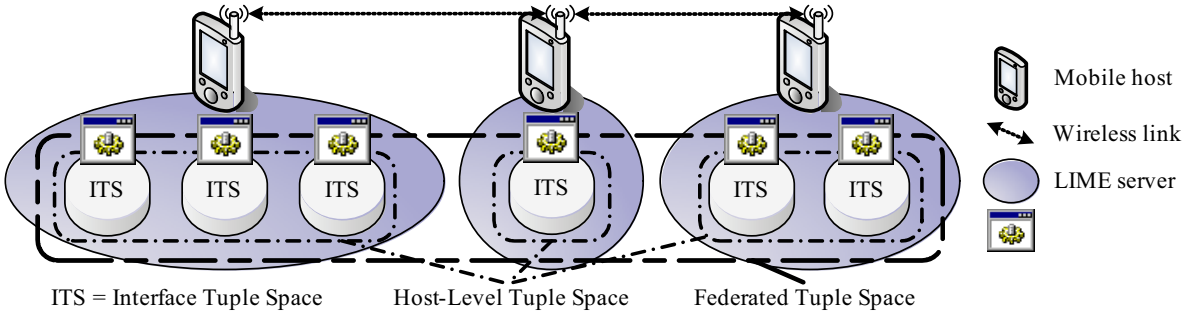


Figure 3-2: LIME tuple spaces' hierarchy

LIME provides the main Linda primitives: **out**(*t*), **rd**(*p*) (and its variants **rdp**(*p*) and **rdg**(*p*)) and **in**(*p*) (and its variants **inp**(*p*) and **ing**(*p*)). They behaves exactly as in Linda except for **out**(*t*) which does not put *t* into the Federated Tuple Space but in the ITS of the process ω which created *t*.

LIME also provides some new primitives for processes, which mind about location of tuples:

- **out**[λ](*t*) to add the tuple *t* into the ITS of the process λ. If λ is not connected at this moment, *t* will be temporary stored in the ω's ITS. During this period, *t* is said to be *misplaced.*
- **rd**[ω,λ](*p*), **rdp**[ω,λ](*p*) and **rdg**[ω,λ](*p*) permit to take misplaced tuples into account or to limit research to some process(es). The parameters ω and λ specify, respectively, current and destination location of the searched tuple(s).
- **in**[ω,λ](*p*), **ipn**[ω,λ](*p*) and **ing**[ω,λ](*p*) use the same parameters for reading and deleting tuples.

As in Linda, LIME, does not provide a primitive to modify a tuple *t*.

LIME does not provide a synchronization primitive. In fact, there is no need for such a primitive since tuples are implicitly synchronized during modifications when they are withdrawn from the Federated Tuple Space.

3.1.2.2 Non-functional properties

3.1.2.2.1 Data management

♦ *Consistency*

One of the advantages of the Linda communication model is to enforce strong consistency. In fact, LIME implements a uniform pessimistic consistency stricter than the sequential consistency. All operations on a tuple are totally ordered and read access always returns the last value of a tuple (or nothing in case of probe primitive).

♦ *Persistency*

At the process level, the persistency of tuples stored in an ITS is not guaranteed by the system. LIME only uses volatile memory. So the extinction of a process permanently deletes all tuples contained in its ITSs. Storing tuples and internal running states on persistent memory is left to the developer. However, this task is facilitated by the current implementation which makes local tuple spaces serializable leading to a form of "*lightweight* persistency".

Since tuples are not replicated, persistency has no meaning from a distributed point of view.

♦ *Accessibility in mobile ad hoc network context*

In LIME, accessibility does not resist to network partitions. Since tuples are not replicated, the tuple *t* is inaccessible to all hosts which are not in the partition of the host on which *t* is stored.

3.1.2.2.2 Resource management in a limited capabilities environment

LIME takes care of neither limited capacities nor heterogeneity of hosts.

3.1.2.2.3 Security

Processes are able to manipulate private tuples in private tuple spaces. For public tuples, LIME does not manage any access rights or other security services.

3.1.2.3 Architecture

The available published papers do not provide a complete picture of the LIME architecture, but it is probably almost similar to the LIMONE one (Figure 3-5 page 15).

LIME communication mechanisms are based on Java socket. They only use TCP streams except for the discovery mechanisms which regularly broadcasts UDP datagram.

In LIME, every host has the same behavior. So LIME is a flat pure peer-to-peer system.

3.1.3 LIMONE

LIMONE [LIMONE] [FRH04] is the result of a joint research effort led by Chien-Liang Fok, Gruia-Catalin Roman (who took part in LIME project) and Gregory Hackmann. It was carried out at Washington University in St. Louis, MO, USA. The only publication is dated from 2004. LIMONE has been implemented in Java. Its sources are available on [LIMONE].

LIMONE (LIghtly-coordinated MObile NEtwork) looks like LIME. This is a coordination middleware where logically mobile processes, hosted on mobile devices, can exchange information over tuple spaces. In fact, it is based on LIME. However, it provides weaker guarantees (notably on operation atomicity) and eliminates all assumptions about network behavior (which can be much more dynamic). In addition, processes use policies to determine with which other processes they interact. This, as we will see below, provides for interesting asymmetric interactions.

3.1.3.1 Functionalities

♦ *Data: type, structure and identification*

Being based on LIME, LIMONE uses tuples to store information. However unlike Linda, LIMONE imposes a structure inside tuple fields. Each of them contains a name, a type and a content. Moreover, the ordered list of fields characterizing tuples in Linda is replaced in LIMONE by unordered collections of named fields. The

following example, given in [FRH04], has three fields:

```
{ ⟨"type" , String , "command"⟩,
  ⟨"device ID" , String , "CD player"⟩,
  ⟨"instruction" , String , "play"⟩ }
```

The sharing unit is the tuple. Its size is variable and it is indivisible.

In the bosom of a tuple space, tuples are identified by their content. LIMONE does not manage several tuple spaces. So, there is no need to identify a tuple space.

♦ *Host identifier, presence and communities management*

Following LIME, a process is identified, on the one hand, by the identifier of the LIMONE server that created it, and, on the other hand, by an increasing number assigned by this server. A LIMONE server is identified both by the IP address of its host and the port number that it uses.

Each process has access to two lists. The first list, **`NeighborList`**, indicates all processes in one-hop neighboring. The second, **`AcquaintanceList`** is a sub-set of **`NeighborList`** and defines all connected processes with a profile selected as "interesting" (see below). Modifications in these lists are not monitored.

LIMONE can make up dynamic communities of interest. Each process has a *profile*. A profile is a collection of triples (property name, type, and value). Additionally to the identifier of the host and process, applications can add their own entries to show their interests. Based on the profile of neighboring processes, each process creates its own community as shown in Figure 3-3. The choice of which profiles are of interest is realized by the Acquaintance Handler according to application specific *engagement*[1] *policies*. A process can only access the tuples of the remote processes in direct communication range with an interesting profile (i.e. in its **`AcquaintanceList`** list). This engagement relation is not symmetrical as shown in Figure 3-3.

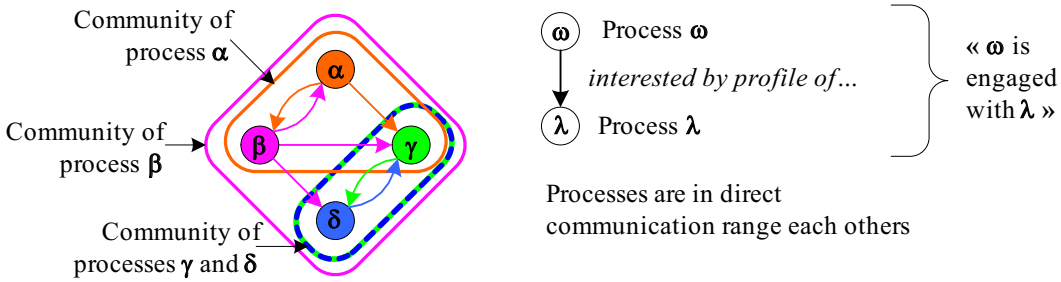


Figure 3-3: LIMONE communities

To conclude, it seems that LIMONE does not allow a process to switch off the presence service.

♦ *Data discovery and searching*

LIMONE does not maintain a list of accessible tuples. However, processes can detect the appearance of tuples. It is based on *reactions*. A reaction consists of a *reactive pattern* and a *call-back function*. The reactive pattern contains a template that indicates which tuples trigger the reaction and a list of profile selectors that determine which processes the reaction should be propagated to. This reactive pattern is sent to all processes in the **`AcquaintanceList`** with a profile matching at least one profile selector of the reaction. The call-back function is stored in the table **`ReactionRegistry`** of the process ω which created the reaction. It is executed atomically when a tuple *t* matches the reactive pattern. If *t* is a remote tuple, then a copy of *t* is sent to ω. In order

[1] In LIMONE terminology, the fact that the process ω is interested by the profile of the process λ is said "ω is engaged with λ".

to prevent deadlock, the call-back function cannot perform blocking operations.

Like LIME, LIMONE provides a data searching service based on pattern matching. However, the structure of its tuples (named and unordered fields) allows handling situations in which a tuple's arity is not known in advance. For example, the following pattern matches the tuple given page 13:

```
{ ⟨"type" , String , valEql("command")⟩,
  ⟨"device ID" , String , valEql( "CD player")⟩ }
```
[1]

This mechanism is directly integrated into reactions and access primitives (see next section).

♦ *Basic data management functions*

In LIMONE, each process is bound to one (and only one) local tuple space. Furthermore, there is not any kind of "global tuple space" which (virtually) gathers the tuples of all local tuple spaces. Figure 3-4 illustrates the LIMONE tuple spaces.

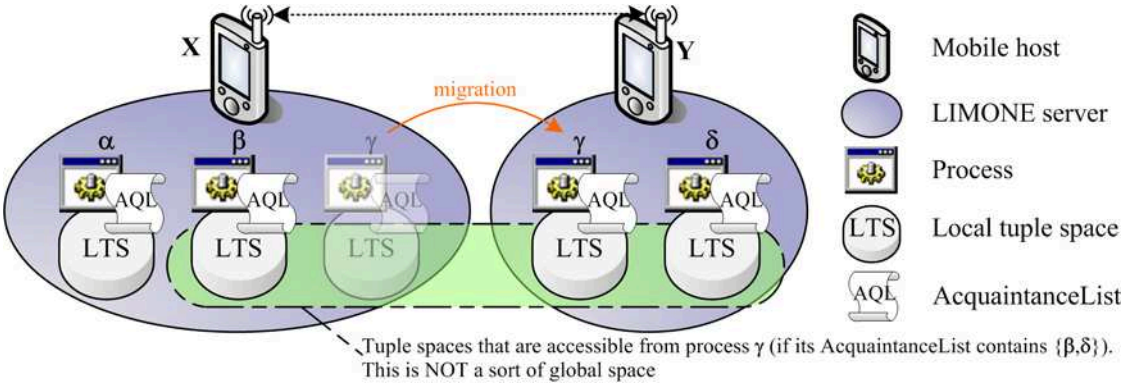


Figure 3-4: LIMONE tuple spaces

On its local tuple space, a process can perform all traditional Linda operations: **out**(*t*), **rd**(*p*) (and its variants **rdp**(*p*), **rdg**(*p*) and **rdgp**(*p*)) and **in**(*p*) (and its variants **inp**(*p*), **ing**(*p*) and **ingp**(*p*)). They behave exactly as in Linda (although limited to the local tuple space of a process).

In order to access a remote tuple space (e.g. the tuple space of the process λ), LIMONE provides the primitives **out**(λ,*t*), **rdp**(λ,*p*), **rdgp**(λ,*p*), **inp**(λ,*p*) and **ingp**(λ,*p*). Blocking operations are not allowed in order to avoid dead lock.

When one of these operations is called, the process on which it is executed sends a request to the remote process λ, sets a timer and remains blocked till a response is received or the timer times out. When the process λ receives the request, it passes it to its Remote Operation Manager, which may reject or approve it. If rejected, an exception is returned to distinguish between a rejection and a communication failure. If accepted, the operation is performed atomically on the λ's local tuple space, and the results are sent back to the initiating process.

LIMONE does not provide any primitive for synchronizing processes.

3.1.3.2 Non-functional properties

3.1.3.2.1 Data management

♦ *Consistency*

Following LIME, LIMONE implements a uniform pessimistic consistency (see page 11).

[1] **valEql**(*p*) is a constraint function that returns *true* if the value within the field is equal to *p*.

♦ *Persistency*

LIMONE provides exactly the same persistency as LIME does (see page 12).

♦ *Accessibility in mobile ad hoc network context*

As in LIME, tuples cannot be replicated. So their accessibility wholly depends on host connectivity (i.e. which hosts are in one-hop neighboring).

3.1.3.2.2 Resource management in a limited capabilities environment

LIMONE does not provide direct mechanisms to deal with limited resources.

However, profiles can be used to adapt the participation of hosts according to their resources. For example, a process can indicate in its profile its free memory and the autonomy of its battery. According to this profile, many scenarios can be considered:

- Migration of processes from a host with low battery level to a host with high battery level.
- Storage of tuples in a process with more free memory.
- Etc.

3.1.3.2.3 Security

The Remote Operation Manager can be used to implement some security mechanisms such as confidentiality and integrity control. However, initially, it only provides a flexible access control. LIMONE does not let a process ω directly manipulate the tuples stored into another process λ. In fact, ω can only *request* λ to perform the operation for it. Then, the Remote Operation Manager of λ uses application specific policies to authorize or not the ω's request.

3.1.3.3 Architecture

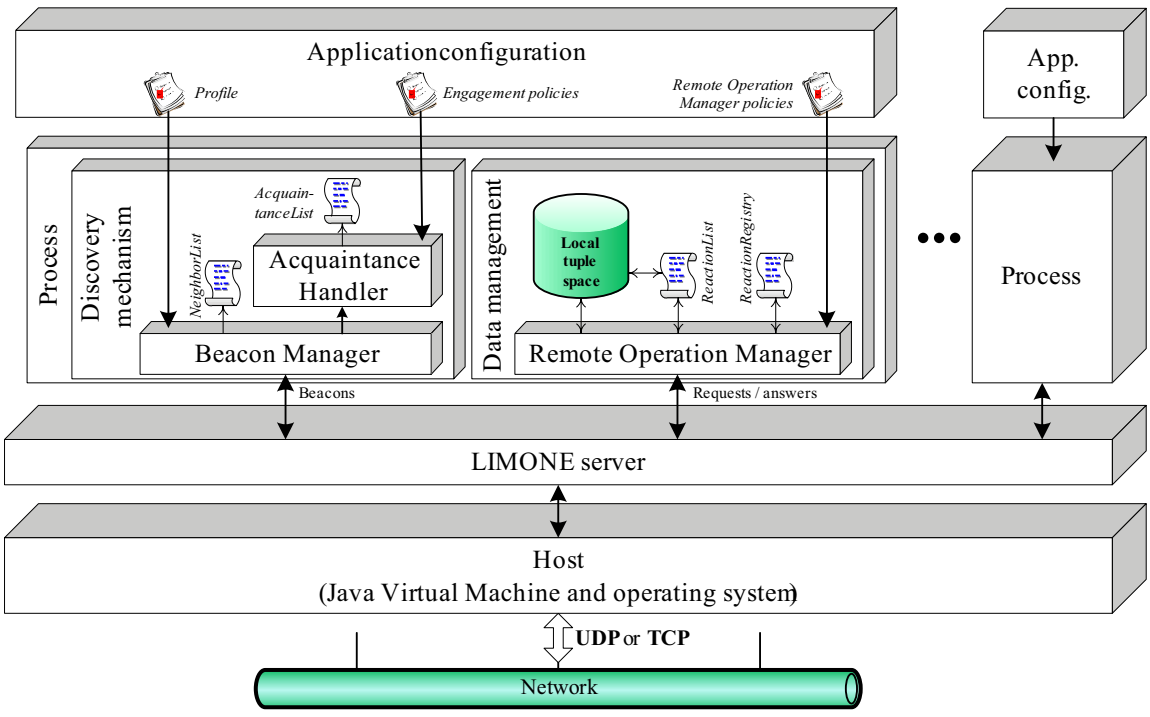


Figure 3-5: LIMONE architecture

Figure 3-5 represents the LIMONE architecture with its main components. We can see that processes are

executed over a LIMONE server which is itself executed over a Java virtual machine. The main components of a process are the Beacon Manager, the Acquaintance Handler and the Remote Operation Manager. The last two have already been presented earlier.

The Beacon Manager deals with the discovery mechanism. It first includes the profile of its process into a beacon. Then it passes the beacon to the LIMONE server which will concatenate profiles of all hosted processes before broadcasting them (for better performances). The Beacon Manager also receives beacons from other processes. Thanks to these beacons, it keeps the **`NeighborList`** table up-to-date. Then it extracts profiles of remote processes from beacons and passes them to the Acquaintance Handler.

The current implementation can transport messages in either TCP or UDP packets. The choice is made when the LIMONE server is launched on each host.

We can notice that every process and every host have the same behavior. So LIMONE is a flat pure peer-to-peer system.

3.1.4 TOTA

TOTA [TOTA] [MaZ03] [MaZ04] is the result of a joint research effort mainly led by Marco Mamei, Franco Zambonelli and Letizia Leonardi at Modena and Regio Emilia University in Italy. The first publication is dated from 2002 and the last one from 2004. TOTA is implemented in Java and runs on laptops and PDAs equipped with 802.11b interface. Its sources are available on [TOTA].

Like Linda, LIME and LIMONE, TOTA (Tuples On The Air) is a coordination middleware based on tuples. However, it uses a quite different approach to provide coordination between distributed processes. It does not maintain any kind of shared tuple space. Instead, it allows tuples to propagate, according to given policies, through the network from node to node. It also offers a strong context awareness mechanism to enable the distributed tuples to self-maintain despite environment dynamism.

3.1.4.1 Functionalities

- *Data: type, structure and identification*

TOTA tuples are much more complex than Linda ones. In TOTA, a tuple *t* is a couple (*C*,*P*) which describes a content *C* and a propagation rule *P*. The content *C* is an ordered set of typed fields representing the information carried on by the tuple, a bit like LIMONE tuples. The propagation rule *P* determines how the tuple should be distributed and propagated in the network. This includes determining the "scope" of the tuple and how such propagation can be affected by the presence or the absence of other tuples in the system. In addition, the propagation rules can determine how tuple's content should change while it is propagated. So, in theory, a transmitted tuple looks like:

$$\Big(\underbrace{\{\langle field\_name_1, type, value\rangle,\ldots,\langle field\_name_n, type, value\rangle\}}_{\text{Content}} \;;\; \underbrace{\{\text{`}rule_1\text{'},\ldots,\text{`}rule_n\text{'}\}}_{\text{Propagation}}\Big)$$

However, in practice, in order to use less bandwidth, the tuples really transmitted are much shorter:

$$\Big(\underbrace{\{\langle field\_name_1, value\rangle,\ldots,\langle field\_name_n, value\rangle\}}_{\text{Content}} \;;\; \underbrace{type\_of\_tuple}_{\text{Propagation}}\Big)$$

The drawback is that each application must construct its own types of tuples. In addition, tuples with an unknown type cannot be managed. A particularity of TOTA is that tuple types are hierarchically organized in

order to write them more efficiently (exactly like in Object-Oriented Programming).

TOTA cannot identify a tuple by its content since it is likely to change. In consequence, it uses a separated identifier (invisible at the application level) composed by the identifier of the host which created it and an incremental counter maintained by this host.

♦ *Host identifier, presence and communities management*

Each host executes only one TOTA server on which a single (non-mobile) application process is running. So host, server and application are all addressed by the same identifier. This identifier is directly built from an IP address. The available papers also evoke the possibility to identify hosts with their MAC address.

TOTA maintains an internal table **`neighborList`** which indicates all hosts in direct communication range. However, this table is not accessible from applications. In return, modifications in **`neighborList`** generate specific events. So an application can be kept aware of them if it subscribes to these events. To do this, TOTA provides a primitive **`subscribe`**(*p*) where *p* is a pattern which filters interesting events. The primitive **`unsubscribe`**(*p*) removes a subscription.

It is important to notice that, contrary to previous Linda-based systems, in TOTA it is the tuples and not the processes that are concerned with subscription. In fact, the 'intelligence' in a TOTA network lies mostly in tuples which maintain themselves according to the environment modifications. So, at the application level, the context awareness of TOTA comes from the content of tuples more than from the event/reaction mechanism.

TOTA does not manage communities of interest and does not allow the presence service to be turned off.

♦ *Data discovery and searching*

TOTA also maintains a list of all accessible tuples (i.e. in direct communication range) under the form of a virtual tuple space. As illustrated in Figure 3-6, a virtual tuple space is constituted by copies of the tuple spaces of neighboring hosts. Modifications in the virtual tuple space also generate events.

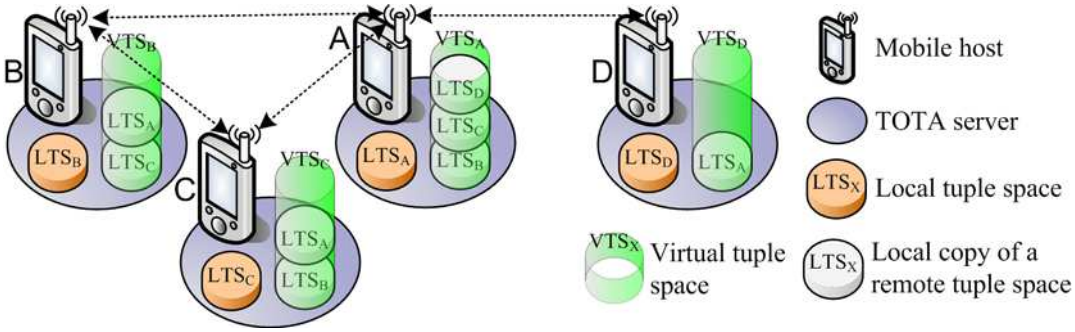


Figure 3-6: TOTA virtual tuple space

TOTA provides two ways for searching tuples. On the one hand, like other tuple-based systems, it supplies a pattern matching mechanism. This one is a bit more constraining than in previous systems because a tuple *t* can match a pattern *p* only if *t* has the same type or a sub-type as *p*. On the other hand, TOTA allows search on the tuples identifier. The first mechanism is directly integrated into subscription and access primitives (see next section). The second is only provided by access primitives.

♦ *Basic data management functions*

Since most of the power lies in tuples and not in applications, the primitives provided by TOTA for managing tuples are quite simple and do not allow complex operations. We can notice that, contrary to LIME and LIMONE, TOTA does not reuse Linda primitives.

TOTA provides several primitives to create a tuple *t* depending on where *t* should be stored. The main primitive is **inject**(*t*). It puts *t* in the middleware and lets it propagate according to its propagation rules. The primitive **store**(*t*) only stores *t* in the local tuple space of the host without propagating *t* to other hosts. On the contrary, the primitive **move**(*t*) propagates *t* to others hosts without storing it in the local tuple space.

TOTA provides the primitives **read**(*p*) and **keyrd**(*t*) to read in the local tuple space and **readOneHop**(*p*) and **keyrdOneHop**(*t*) to read in the virtual tuple space. The primitives **read**(*p*) and **readOneHop**(*p*) returns all tuples that match the pattern *p* while **keyrd**(*t*) and **keyrdOneHop**(*t*) return tuples with the same identifier as the tuple *t*.

TOTA is unable to delete all replicas of a tuple. However, it provides the primitive **delete**(*p*) to remove a replica from the local tuple space.

TOTA does not provide any primitive to modify a tuple. In fact, TOTA tuples are supposed to modify themselves (in reaction to context modification).

TOTA does not provide any primitive for synchronizing processes.

3.1.4.2 Non-functional properties

3.1.4.2.1 Data management

♦ *Consistency*

The TOTA middleware itself does not provide data consistency. However, tuples come with their own consistency: tuples content remains consistent according to the propagation rules.

♦ *Persistency*

At the host level, tuples are only stored in volatile memory (so are lost when the host switches off).

At the distribution level, the number of copies of a tuple (its persistency) depends on the propagation rules.

♦ *Accessibility in mobile ad hoc network context*

Since tuples are replicated in all hosts (except if the propagation rule specifies the contrary), their accessibility (in read only mode) is not influenced by network partition, not even by host disconnection.

3.1.4.2.2 Resource management in a limited capabilities environment

From a memory point of view, the main default of TOTA lies in its management of virtual tuple spaces: a host must store not only its own tuples but also tuples of all its neighbors which is unrealistic in a dense network.

The good side of such a mechanism is that it decreases bandwidth use for frequently accessed remote tuples. However this benefit is offset by the fact that even the tuples that are never accessed are transmitted.

3.1.4.2.3 Security

TOTA does not manage security.

3.1.4.3 Architecture

Contrary to LIMONE, the application process is quite simple. Most of the complexity lies in the definition of application specific types of tuples during the development phase and in the TOTA server, during execution. The main task of an application process is to implement a function **react**(*e*) that processes the events it wanted to receive.

The TOTA server is split into three parts. The TOTA API is the main interface to access the middleware. It

provides primitives seen in chapter 3.1.4.1. The Event Interface is the component in charge of asynchronously notifying the application of subscribed events (via its **`react`** function). The TOTA Engine is the core of TOTA. It is in charge of maintaining the local tuple space, the virtual tuple space and the list of neighbors. It manages the tuples' propagation by sending and receiving them and by calling their **`execute`** function which defines their behavior (i.e. whether they are executed, modified, propagated and locally stored). Finally it broadcasts all modifications of its local tuple space to remote hosts in order that they maintain their own virtual tuple space and **`NeighborList`** table.

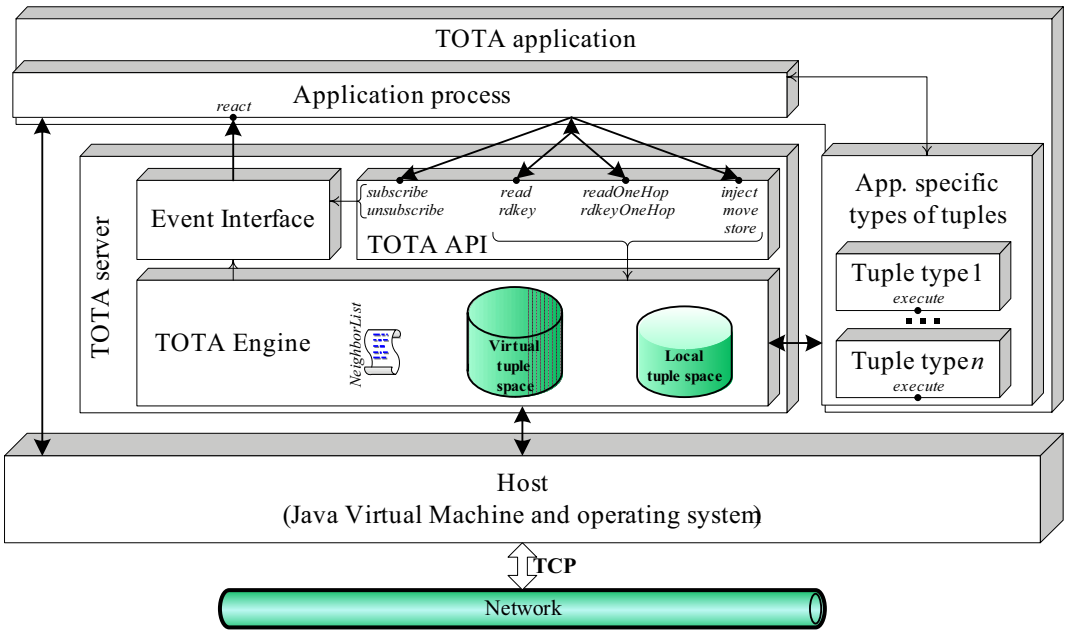


Figure 3-7: TOTA architecture

The current implementation casts messages and beacons in TCP packets.

This architecture is used in all hosts which all have the same behavior. So TOTA is a flat pure P2P system.

3.1.5 Synthesis on tuple-based systems

We have studied three middleware based on tuples (LIME, LIMONE and TOTA) but there exists many other tuple-based systems. In [CFL+06], the authors of TOTA realized a taxonomy covering most of them. We can notice from this paper that all these systems are mobile but very few support MANETs. Moreover, from the above presented study, we can conclude that systems supporting the more dynamic network topologies have the less in common with the original Linda model. For instance, LIME closely follows the Linda model, but, although designed for MANETs, it is unable to run over them in practice. In fact, it requires rather stable networks because it provides each host with a complete view of its partition. Therefore, LIME offers one of Linda's most interesting features: the global tuple space which allows users to view all shared data (almost) anywhere at any time. In order to be more dynamic, LIMONE renounced this feature by looking only after direct neighborhood. Finally, TOTA is the system best adapted to MANETs. It only considers direct neighbors and uses only local mechanisms. However, it has almost nothing in common with Linda. Even its tuples are very specific since TOTA allows several instances of a tuple to have different values. TOTA uses the difference of values rather than the value itself. These tuples seem more useful for context awareness than for data sharing. However, the concept of propagation rules associated with data is very interesting for making data autonomous.

Another characteristic of the previous systems is that none of them provide advanced mechanisms for non-functional properties. For instance, data consistency is guaranteed by forbidding either replication or write operations (which greatly simplify the problem). Similarly, security and limited capabilities management are seldom addressed.

3.2 Generic data based systems

Most data sharing systems use generic data. We use the term “generic” to emphasize the fact that these systems manipulate data with an implicit structure neither recognized nor interpreted by the middleware. Generally, these generic data are files as in Coda, Ficus and AdHocFS. The only exception is InfoWare which claims to manage “any kind of data that's storable” [SGP04].

3.2.1 Coda

Coda [CODA] [SKK+90] [Lee00] is an advanced distributed file system. It was developed at Carnegie Mellon University from 1987 to 2000 by the team of Mahadev Satyanarayanan in the SCS (School of Computer Science) department. The first publication is dated from 1987 and the last one from 2000. Coda is implemented in Java. Its sources are available on [CODA] under the terms of the GNU General Public License Version 2. Binary versions exist for many platforms and are still maintained nowadays (the version 6.0.12 has been released in 2005/09/21).

The Coda file system is a descendant of the Andrew File System [MSC+86]. Like AFS, Coda offers a location-transparent access to a shared Unix file namespace that is mapped onto a collection of dedicated file servers. But Coda represents a substantial improvement over AFS because it offers considerably higher availability in the face of server and network failures. The improvement in availability is achieved using the complementary techniques of server replication and disconnected operation.

Notice that Coda follows a client-server architecture where each file is replicated on servers and where clients contact available servers to access files. It was not designed for MANET and is not well suited to them. It is given there for comparison reason because it represents the first distributed system supporting mobile computing with total disconnection from servers.

3.2.1.1 Functionalities

- *Data: type, structure and identification*

Coda is a distributed file system, thus it manipulates ordinary files. It does not interpret their structures. Their types are potentially used for calling external applications during file reconciliation (see later).

The sharing unit, at the client (i.e. user) side, is the file. Its size is variable. On the local file system, the file is split into blocs, however, from a replication point of view, files are indivisible. It means that a host locally replicates all blocs of a file or none. In other words, if a host wants to access a file, it must replicate this file wholly into its memory.

Coda uses two kinds of file identifier. At user level, files are uniquely identified by a pathname. At low level, Coda uses a file identifier called *fid*. This one is composed of two parts: a 32-bit RVID (Replicated Volume Identifier) and a 64-bit file handler. The first part uniquely identifies the *volume* which the file belongs

to[1]. A volume is a set of files. It is the unit for server-side replication. The uniqueness of the RVID and of the file handler is guaranteed by sets of central servers. When these servers are not accessible, new files are associated to temporary fids which are supposed to be unique. During reconnection step, temporary fids are translated into permanent fids which are guaranteed to be unique.

♦ *Host identifier, presence and communities management*

Because of the wired context of Coda, no special host identifiers are needed. Hosts are probably identified by the network layer (e.g. IP address or host name).

Because of the client-server architecture, Coda does not provide host presence service since hosts only communicate with servers. In return, it offers a server presence mechanism which has the specificity to mask the location of servers. In addition, there is no use to turn off this presence service.

Volumes used by Coda can theoretically store any kind of files. However, in practice, volumes correspond to a collection of related files. So volumes can be used to simulate communities of interest.

♦ *Data discovery and searching*

Before searching files (and more generally accessing them) clients must mount the volume they belong to. Available published papers do not explain how volumes are discovered. It is also not clear if Coda maintains on clients a local list of the files of the volume. It seems it does not. Coda also does not provide specific data searching services because it is compatible with traditional system searching tools.

♦ *Basic data management functions*

Volumes are created and shared by servers. They are mounted / unmounted by clients with a kind of UNIX *mount* / *umount* functions. Notice that volume files are not locally copied when the volume is mounted.

Coda behaves as a traditional network file system. It does not provide any primitive to manipulate file but intercepts operating system function calls (mainly **`open`** and **`close`**) as illustrated in Figure 3-8 extracted from [TaS01]. In this figure, system calls are intercepted and transmitted to Venus which is the Coda middleware at the host-side. According to the context, Venus satisfies user requests either by using local file or contacting servers. If the accessed file is not locally stored and no server is accessible, Venus returns an error.

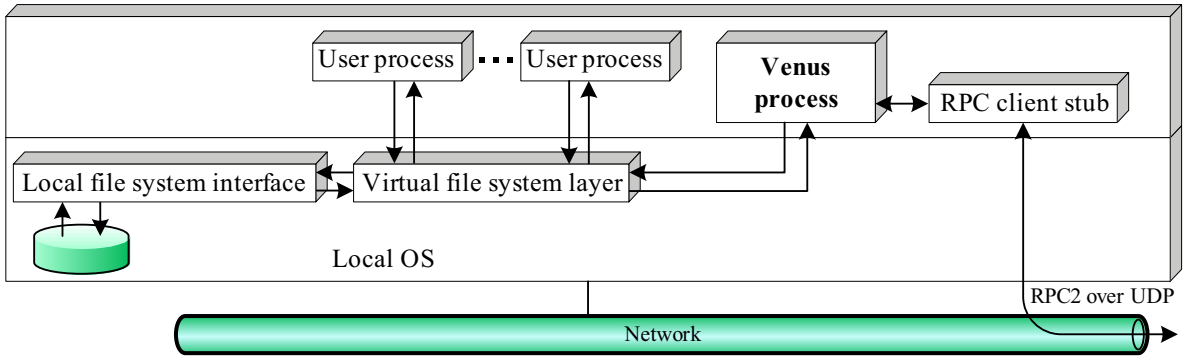


Figure 3-8: Coda interception of system calls

[1] A shared file belongs to one and only one volume.

3.2.1.2 Non-functional properties

3.2.1.2.1 Data management

♦ *Consistency*

Coda internally distinguishes two functioning modes: the *connected mode* if Venus can contact at least one server and the *disconnected mode* if it cannot (notice that these modes are transparent to the user). However, contrary to many other systems which make such a distinction, it provides only one consistency model. In all cases, Coda offers an optimistic read-your-writes consistency model. More precisely, it guarantees that when a user opens a file *f*, it gets the most recent version of *f* that is accessible. The only advantage of the connected mode is that servers notify their clients when one of their cached files is modified by another client.

Since Coda authorizes concurrent writes, notably when servers are partitioned, replicas of a file can diverge. So Coda provides a mechanism to detect these divergences by using version vectors[1]. A particularity of Coda, is that it is lazily realized by hosts (but transparently to users) and not by servers.

When two replicas (of the same file) have different version vectors, it means either that a replica is more recent than the other one, or that there is a conflict. In the first case, last updates are automatically propagated. In the second case, Coda can resolve the inconsistent file replicas with the aid of an *Application-Specific Resolver* (ASR). An ASR is provided by someone who understands the file format used by an application. During resolution, all involved replicas are blocked, i.e. cannot be accessed. Irresolvable files are marked inconsistent and are unavailable until they are manually repaired.

♦ *Persistency*

Coda guarantees strong persistency at the host level since files and even functional states are stored in persistent memory. So after extinction, a host can restarts and continues where it left off.

At the network level, file persistency relies on servers.

At the server level, traditional techniques for wired networks are used.

♦ *Accessibility in mobile ad hoc network context*

Coda accessibility is not influenced by network partition but by the fact that at least one server is reachable. In disconnected mode (no server is accessible), Coda maintains a good accessibility thanks to a hoarding mechanism which pre-loads (during connected mode) replicas of files which may be needed in future. Performances of this mechanism can be improved if users specify their needs in a *hoard profile*. In addition, the optimistic consistency model does not constrain possible operations in disconnected mode.

The Coda accessibility is decreased during reconnection phase. In fact, during reconciliation of inconsistent replicas, all involved replicas are blocked for read and write accesses. The accessibility is also limited when the hoarding mechanism failed because non-locally stored files cannot be accessed even if other (accessible) hosts have these files.

3.2.1.2.2 Resource management in a limited capabilities environment

Coda is not designed for working in limited capacities environment. So it doesn't deal with energy and bandwidth. However, it tries to minimize memory usage, notably in disconnected mode.

[1] For more details on version vectors, see [Lee00], chapter 2.2

3.2.1.2.3 Security

In connected mode, Coda provides authentication of servers and clients. So, on the one hand, clients are sure to access valid files and on the other hand, servers can guarantee the respect of access rights based on access control lists. For reasons of simplicity and scalability, it associates an access control list only to directories and not to files. All normal files in the same directory (i.e., excluding subdirectories) share the same protection rights. Coda distinguishes access rights with respect to the types of operations (e.g. file creation). In addition, it supports the listing of negative rights to specify that a specific user is *not* permitted to certain accesses.

ASR is also secured because they represent a great danger to compromise files.

3.2.1.3 Architecture

Figure 3-8 page 21 illustrates the architecture of a Coda client. It is composed of two modules:

- The virtual file system layer is directly integrated into the operating system. It allows intercepting system calls of user processes.
- The Venus process executes in the user application layer. In connected mode, Venus roughly forwards user request to Coda servers. In disconnected mode (also called *emulation* mode), it performs many actions normally handled by servers by allowing accesses to file locally replicated. Venus is also responsible of resolving inconsistent file replicas after server reconnection.

Clearly, Coda communication model follows a client-server architecture.

Coda uses RPC2 which provides synchronous communications. Its particularity is to be able to send multiple requests (but does not use multicast). This protocol is UDP based.

3.2.1.4 Synthesis and conclusion

With its client-server architecture Coda is clearly not designed for MANETs. However, it provides some solutions:

- To anticipate disconnection with its hoarding mechanism;
- To work autonomously and transparently with the Venus process which emulates servers and logs operations in an optimized way for space saving;
- To diffuse and reconcile concurrent updates based on version vector and operation replay.

3.2.2 Ficus, Rumor and Roam

Ficus [GHM+90] [Rat95] is a research project headed Gerald Popek at UCLA (University of California, Los Angeles) in the Laboratory for Advanced Systems Research (LASR). Publications are spread from 1987 to 1995. Ficus has been implemented in SunOS 4.1.1 and was used in UCLA. Its sources are not available.

Ficus is a transparent, reliable, distributed file system designed for very large scale networks with geographically dispersed hosts and wired communications. Like Coda, it permits updates during network partitions. But it goes farther by removing any server: hosts communicate directly among themselves.

3.2.2.1 Functionalities

♦ *Data: type, structure and identification*

Ficus manipulates ordinary files like Coda.

In its first version, the sharing unit of Ficus was the volume which groups sub-trees of files. Later, as

explained in [Rat95], a *selective replication* service has been added to Ficus. It provides a finer control over replication. So, from a user point of view, the sharing unit is the file.

A volume is identified by the pair ⟨*creator_id*,*volume_id*⟩ which corresponds to the name of the host that created it and a name this host has attributed to the volume. Individual volume replicas are further identified by a *replica_id*. Within the context of a volume, a file is identified by a *file_id*. So a fully specified identifier for a file replica is ⟨*creator_id*,*volume_id*,*file_id*,*replica_id*⟩. The nature of these *xxx_id* is not explained, neither how their uniqueness is maintained.

♦ *Host identifier, presence and communities management*

Because of the wired context of Ficus, no special host identifiers are needed. So hosts are probably identified by the network layer (e.g. IP address or host name).

Available published papers do not precise how the presence of other hosts is detected.

Ficus does not manage community of interest. But like in Coda, volumes can be used for this purpose.

♦ *Data discovery and searching*

Available papers do not explain how volumes are discovered. Inside a (mounted) volume, users can use traditional system research tools because, as explained in the next section, Ficus is totally transparent to the user.

Internally, Ficus maintains a partial list of the files of a volume in order to make access to remote file replicas faster. In fact, for each locally stored object (i.e. file, directory or symbolic link), the parent directory is also stored locally. In addition, for each locally stored directory, all its direct-child objects must be either locally stored or substituted by statue value which optimistically indicates which hosts store a replica of this object.[LM1]

♦ *Basic data management functions*

Ficus is integrated directly inside the OS kernel. It is able to intercept data management system calls. So, most of the time, it is totally transparent to users and/or final applications. The only difference occurs when a file is unavailable (when it is stored remotely on disconnected hosts). In this case, Ficus returns a specific error code.

The decision of sharing a volume (i.e. a set of files) is taken by a single host. This operation is not well documented but it may provide a way to define initial replication degree (where the volume will be replicated). Once the volume is shared, remote hosts which want to access its files must previously use a kind of UNIX *mount* operation. Then file access are transparent. Ficus also provides a kind of UNIX *unmount* operation.

Data management is realized via UNIX system primitives (mainly *open*, *read*, *write*, *close* and *remove*). These accesses can be local or remote. The accessed file replica is chosen by Ficus. The local file replica has priority if it exists; otherwise Ficus uses the first accessible (remote) replica it finds. It also permits the host to explicitly select a particular remote replica for (very) specific cases.

By integrating the selective replication, Ficus introduced two new primitives in order to create and delete a local file replica. This is useful, if a user or a specific application is able to predict the need of a particular file in the future. In this case, creating a local replica lessens the impacts of network partitions and disconnections. On the contrary, if a file is rarely accessed, its local replica can be deleted in order to free resources. However, the file is always accessible via a remote replica.

It is important to distinguish the previous file replica deletion from a file removal (by the UNIX *remove* function). The latter aims to globally remove a file. It is based on the garbage collection service provided by Ficus. This mechanism is described in [Rat95]. It consists of two parts:

- First, it checks that all links on this file have been removed. In other word, it verifies that all hosts having a replica of this file want to globally remove it. If it is not the case, the process is halted.
- Then, it really removes all replicas thanks to the Guy's algorithm [GPP93].

Since Ficus wants to be user transparent, it provides no synchronization primitive.

3.2.2.2 Non-functional properties

3.2.2.2.1 Data management

♦ *Consistency*

Ficus allows concurrent accesses and uses two mechanisms to maintain an eventual consistency:

- Periodic reconciliation based on version-vectors in a multi-ring topology [Rat95]
- Broadcasting of updates to other hosts. Such a mechanism does not guarantee consistency because of network partitions but it increases reactivity of geographically close hosts.

The model of consistency depends on whether the host stores a replica of the file. In this case, Ficus provides a read-your-writes consistency (like Coda). In the other case, the consistency is weaker.

Like Coda, Ficus uses version-vectors to detect divergence of two replicas. Many of these divergences can be automatically resolved as explained in [RHR+94]. Others can invoke specific resolvers to attempt to handle file conflicts.

In order to quickly reach eventual consistency, Ficus uses a multi-ring topology using gossip-transferal of information. This consists in associating an adaptive ring with each file as illustrated in Figure 3-9 extracted from [Rat95]. An adaptive ring is a circular sequence of hosts storing a replica of the same file. It is capable of dynamic reconfiguration face to network partitions and adaptation to creation / deletion of replica of this file. Reconciliation is organized as a pull of information from the following host in the adaptive ring.

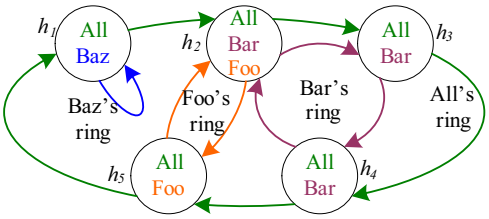


Figure 3-9: Multi-ring topology example

In a not-too-dynamic network, the multi-ring topology seems a good compromised between the number of messages which must be exchanged between all hosts and the length of time required to accomplish a user-visible action.

♦ *Persistency*

At the host level, file and, at least, a part of functional states are stored on persistent memory.

From the distributed system point of view, the persistency of a file is compromised by the UNIX *remove* primitive and the file replica deletion. In the former case, the garbage collection mechanism guarantees that replicas will not be removed before reaching a consensus between all hosts storing them. So when the file disappears, it can be considered as useless. In case of a local replica deletion, a process called *reverse-reconciliation* enforces the reconciliation with another host having a replica of the same file. This prevents to lose potential last updates and to delete the last replica. However, the reverse-reconciliation can be bypassed if

the possibility of losing update is less important than freeing disk space.

♦ *Accessibility in mobile ad hoc network context*

Ficus provides two policies to automatically replicate a new file:

- By default, on every host which stores a replica of its parent directory.
- In some cases, notably at the root level, the default behavior can induce too many replicas. So users can use per-directory replication mask where they can specify where the directory's children should be replicated among hosts storing a replica of this directory.

These policies are interesting but they are not adaptive to network volatility. So they cannot guarantee that at least one replica exists in each network partition. Although users can explicitly create local replica in order to lessen disconnections, Ficus wants to be user-transparent. For this reason, the system SEER [Kue94] [SEER] was designed to work over Ficus. It provides hoarding mechanism which is able to predict future needed file and to create local replicas in consequence.

3.2.2.2.2 Resource management in a limited capabilities environment

Ficus was not designed for a limited capabilities environment. It is obvious at the communication level. For instance, the consistency mechanism which broadcasts updates is quite expensive although not really useful. Another example is the multi-ring topology. It is build from host identifiers and does not reflect host proximity.

From a memory point of view, the garbage collection is quite long to free disk space. So memory can easily become full of "garbage" waiting to be collected.

3.2.2.2.3 Security

Ficus does not take care of security. In fact, authentication, confidentiality, integrity and access rights are handled in Truffles (TRUsted Ficus FiLE System) [RPP+93]. This system, jointly developed by UCLA and TIS (Trusted Information Systems) enhances Ficus with easy-to-use file sharing security. It is based on the TIS/PEM (Privacy-Enhanced Mail) system standardized in RFC 1421, 1422, 1423 and 1424.

3.2.2.3 Architecture

Available papers on Ficus architecture only speak about *stackable layers*. This is a modular structuring paradigm where all layers use the same interfaces. Although interesting, this concept has no real impact on data sharing. Little additional information on architecture is provided except that, Ficus is deeply integrated into the SunOS kernel and that it uses the protocol NFS (Network File System) [SGK+85] for remote file accesses. Since it is the version 2 of NFS, communication is based on UDP datagram.

All hosts have the same behavior. So Ficus is a flat pure P2P system.

3.2.2.4 Evolutions: Rumor and Roam

Ficus has been upgraded by the same research team of UCLA. Two new versions have been developed some years apart: Rumor [GRR+98] [RUMOR] (≈ Ficus v2) and Roam [RRP04] [Rat98] [ROAM] (≈ Ficus v3).

Rumor is based on Ficus and reuses many of its mechanisms (selective replication, reconciliation with multi-ring topology and garbage collection). The main differences are:

- It is implemented at the application level (not inside the kernel). This makes Rumor more portable across different platforms.
- It enforces entirely optimistic consistency without immediate update propagation. This drastically

decreases message exchanges without compromising file consistency.

Roam is based on Rumor. Its main improvement is the use of a hierarchical pure P2P communication architecture for increasing its scalability[LM2]. This architecture groups close hosts into clusters called *wards* (Wide Area Replication Domains). Inside a ward, consistency is maintained like in Ficus or Rumor (with a multi-ring topology). In each ward, a *master* is elected. An adaptive ring of ward masters guarantees consistency between distant hosts. This two-level hierarchy is illustrated in Figure 3-10 extracted from [Rat98].

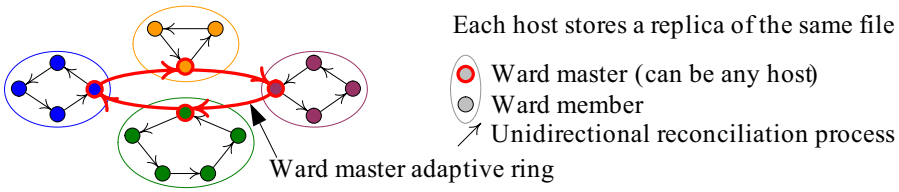


Figure 3-10: Roam hierarchical pure P2P communication architecture

Most mechanisms of Roam are adaptations of those of Rumor to this communication model. The really different mechanisms in Roam are:

- The garbage collection does not wait a global consensus to free disk space. However, during the reconciliation process which propagates a file removal, in order to prevent data loss, the garbage collection only frees memory if the version of the local file replica is equally or less up-to-date than that of the initially removed file replica. In the opposite case, the file replica is moved into a special directory and the potential data loss is notified to the user.
- The version vectors are maintained dynamically and integrate a compression algorithm whose principle is to remove an element of a version vector if all hosts agree on its value. This mechanism is adapted to the hierarchical ward model but does not seem to withstand network partitions.

Notice that, like Ficus, both Rumor and Roam can be enhanced with SEER and Truffles.

3.2.2.5 Synthesis and conclusion

Ficus and its evolutions are designed for mobile networks, not for MANET. However, some of their features are interesting and could be adapted to a more dynamic and resource constrain environment:

- The adaptive ring topology could be quite efficient in term of message exchanges. But it must be restricted to a small area. The Roam model, although not sufficient, is a first improvement.
- The combination of optimistic consistency with immediate update propagation could also be interesting if the broadcasting distance is adaptable with the state of the network.
- The automated replication policies for new files provide some leads to increase accessibility. But this mechanism should also take into account at least devices resources.
- Ficus is the only P2P system of this paper which provides a way to globally remove a data.

3.2.3 AdHocFS

AdHocFS [ADHOCFS] [Bou03] [$BI03_a$] [$BI03_b$] is mainly the PhD work of Malika Boulkenafed supervised by Valérie Issarny. Anis Ben Arbia, David Mentre and Animesh Pathak also took part in this research effort led in the ARLES research team, INRIA of Rocquencourt, France. The first publication is dated from 2001 and the last one from 2003. AdHocFS is implemented in Objective Caml 3. Sources are not available.

AdHocFS aims at offering a distributed file system that supports collaborative caching among ad hoc groups

of terminals in the local communication range of each other. Its particularity is the data management in a way that both takes account fair resource usage and enables data replication on mobile nodes that may later access the data, notably when isolated.

3.2.3.1 Functionalities

♦ *Data: type, structure and identification*

AdHocFS manipulates ordinary files and does not deal with their structure and type (not even for calling external applications like in Coda or Ficus).

From the application point of view, a file is indivisible. So, the sharing unit is the file. Its size is variable.

Inside a *security domain* (SD), files are identified by their full path and name, like in conventional file systems. (e.g. `/directory_1/.../directory_n/file_i`). A SD corresponds to a set of files very similar to a CODA volume. In addition, it integrates security mechanisms (described later). A security domain is uniquely identified through the IP address of its home server which stores all files.

♦ *Host identifier, presence and communities management*

Hosts are identified by their IPv4 address.

For efficiency reasons, AdHocFS constructs groups of fully connected hosts belonging to the same security domain as shown in Figure 3-11. These groups are rebuilt regularly, with a dynamic period which adapts itself to the volatility of the network. A host can participate to several groups as long as these groups concern different security domains. Each host of a group has the list of all hosts forming this group (and only them). Modifications are not notified.

A security domain can be viewed as a community of interest since only a set of hosts sharing a common certificate can access these files. It means that these hosts, called members, share a common property or interest. The disadvantage of such a system is that creating a spontaneous community is impossible.

Hosts can leave groups to work alone and save energy.

♦ *Data discovery and searching*

All files shared in a group are indexed on each host in a virtual directory structure. This structure, illustrated in Figure 3-11, shows both local and remote files, as well as where copies are replicated.

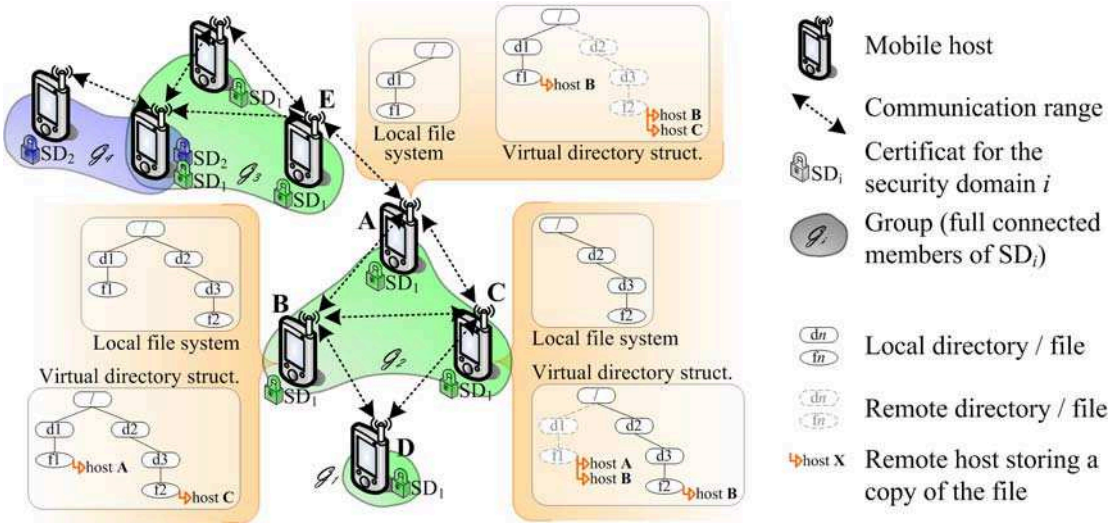


Figure 3-11: AdHocFS groups and virtual directory structures

Data search is based on file names like traditional file systems.

♦ *Basic data management functions*

Base functions for manipulating files in AdHocFS are not described in the published papers available. In addition, the source code is not available. So, we can only make assumptions about the provided primitives.

Since AdHocFS is based on a traditional Unix file system we expect to find the following set of primitives: **`create`**, **`open`**, **`read`**, **`write`** and **`close`**. In addition, all files are protected by locks in the bosom of a group (notice that a disconnected host belongs to a group where it is alone). So **`open`** and **`close`** primitives may include synchronization mechanisms.

File sharing may consist in placing the file into a special directory dedicated to a given SD. The content of this directory is visible from other hosts of the SD.

In order to access to a file, AdHocFS provides **`open`**(*f*,**`Read`**) or **`open`**(*f*,**`ReadWrite`**) to get a **`Read`** or **`ReadWrite`** flag. The first flag guarantees that the host has an up-to-date local copy of *f* and it can read it. The second guarantees the same but with an exclusive access in order to modify *f*. In this case, the flag of other hosts is set to **`Invalid`**.

File reading is normally performed via **`read`**(*f*) when host has a **`Read`** flag. However, it does not, it can also use **`enforceRead`**(*f*) which may return an out of date file. File update is only performed via **`write`**(*f*) when host has a **`ReadWrite`** flag.

The primitive **`close`**(*f*) has only local effect: release exclusive writer lock (if it exists) on the host and physically close the file. In case of an update, this one is not broadcasted to other hosts which keep their **`Invalid`** flag. The update is only sent to the first host which wants to read *f*. In this case, flag of the former writer and the new reader become **`Read`**. Flags of other hosts which have a replica of *f* are set to **`UpdateRead`** which means that these replicas can be read but have to be updated before.

No primitive is provided for removing all replicas of a file. But AdHocFS allows hosts to delete their local replicas. Before deleting a replica (with a system call), AdHocFS verifies that at least another up-to-date copy of this file is available in the group. If not, such a copy is created on another host.

3.2.3.2 Non-functional properties

3.2.3.2.1 Data management

♦ *Consistency*

First of all, the consistency of the replicas of a file is assured only in a security domain. So a file must belong to one and only one security domains. AdHocFS provides two levels of consistency. First, all replicas are eventually consistent following a Read-your-writes consistency model. For detecting unavoidable divergences, AdHocFS associates to each replica a version vector, called Coherency Control List (CCL). CCLs are also used to automatically resolve non-conflicting divergences. Conflicting ones are left to the user responsibility. In the spirit of AdHocFS, the global eventual convergence is assured by the home server of the security domain.

Second, within a group, AdHocFS offers a uniform pessimistic consistency stricter than the sequential model. To do so, it uses a Multiple Readers Single Write (MRSW) mechanism with locks. This is possible because in a group hosts are fully connected.

The choice of this hybrid consistency is justified by the fact that users needing a strong consistency between

their replicas are generally geographically closed whereas far users prefer a high accessibility even if copies diverge.

♦ *Persistency*

At the host level, file copies are stored on persistent memory.

From the distributed point of view, file persistency is guaranteed by the reference copy stored in the home server which is considered as sure.

At the group level, AdHocFS prevents the disappearance of the last up-to-date replica for each file shared on a group by associating a *profile* to each host. A profile represents a level of energy, a level of free memory and an estimation of how much time the host will stay in the group. So, AdHocFS is able to predict the disappearance of a last up-to-date replica and to replicate it to a more reliable host.

♦ *Accessibility in mobile ad hoc network context*

Accessibility is more influenced by group membership than by connectivity. In fact, group level persistency almost guarantees that a remote file *f* available in the virtual directory structure of a host *h* will remain accessible as long as *h* stays in the same group.

However, if *h* leaves the group, it must get a local copy of *f*. If not, the file won't be accessible any longer, even if the host which stores it is always within communication range. If *h* has a replica of *f*, the weak consistency between groups allows it both read and write operations.

3.2.3.2.2 Resource management in a limited capabilities environment

AdHocFS is designed for working on limited capabilities devices. So it allows users to define the amount of memory they wants to share with members of each security domain to which they participate. It also uses profiles to be as fair as possible with respect to free memory use, for instance during replication.

AdHocFS is also concerned with energy saving. So, it tries to minimize communication:

- During group creation a leader is elected to reduce the number of exchanges between hosts. In addition, at regeneration of the group, the leader is always renewed in a concern of equality (unfortunately the energy level of profiles is not taken into account here).
- Updates are lazily propagated and therefore only sent when it is required.

Finally, AdHocFS security services use symmetric keys for limited computational power devices.

3.2.3.2.3 Security

In AdHocFS, hosts are not identified individually but according to their security domains (SD) membership. For each SD they belong to, they get a Digital Certificate (DC) from a trusted third party. So they can prove each others their membership of a given SD and form trusted groups. In order to modify SD access rights, Digital Certificates have a validity period. In consequence, hosts must be able to regularly contact the trusted third party to update their DCs.

Inside a group, hosts have full access to shared data and confidentiality is guaranteed by encrypting all communications with a symmetric key shared during the group creation.

3.2.3.3 Architecture

Figure 3-12 represents the architecture of AdHocFS. It is composed of three layers and an API.

The bottom layer deals with security mechanisms (authentication and communication encryption).

Upper comes the group manager layer. Its first role is the discovery of hosts within communication range

belonging to the same security domain(s). This task mainly relies on the protocol SLP (Service Location Protocol) [Gut99] [GPV+99]. Its second role is the group initialization.

After group initialization, the file manager has at its disposal a virtual directory structure and the profiles of the hosts in the group. This information will be used in consistency, persistency and accessibility mechanisms.

AdHocFS uses operating system interfaces for network communication, monitoring energy and input/output operations on local persistent memory.

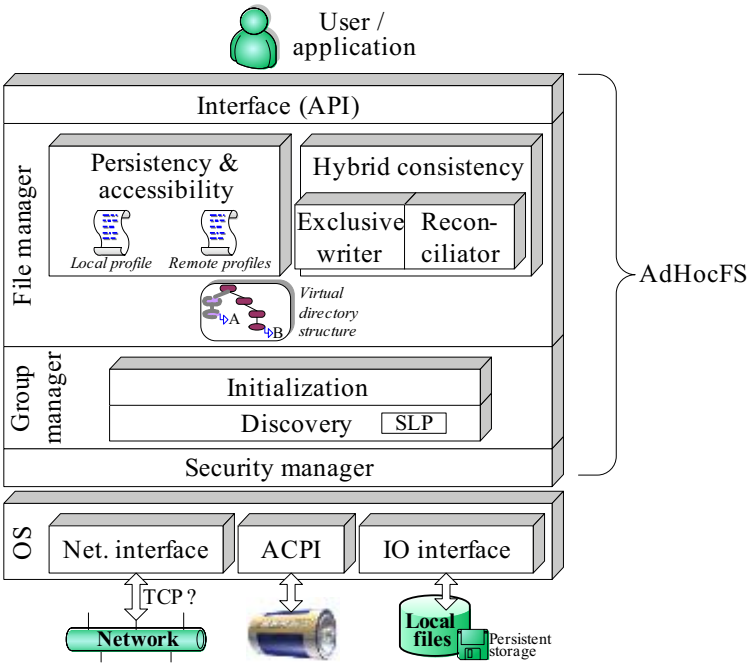


Figure 3-12: AdHocFS architecture

From a group point of view, AdHocFS follows a hierarchical pure P2P middleware. From a general distributed system point of view, AdHocFS enforces a hybrid P2P architecture since it relies on a home server to guarantee the eventual consistency and security.

AdHocFS communications seem to be TCP based.

3.2.3.4 Synthesis and conclusion

AdHocFS is the first middleware, which really accounts for all the non-functional properties addressed in this paper:

- Protocols are designed for energy saving and take into account heterogeneous capabilities.
- Data management offers a strong consistency, persistency and high accessibility into groups.
- Some elements of security are proposed.

The main drawback of AdHocFS is its global consistency which is based on a central server. In addition, its reconciliation mechanism seems to erase most conflicting updates which is not desirable.

3.2.4 Ad-hoc InfoWare

Ad-hoc InfoWare [INFOWARE] [PAD+04], is designed and developed in the Department of Informatics of University of Oslo in partnership with Thales Communications AS and Oregon Health Science University. This is a research project of 4 years from 2003 to 2007 funded by the Norwegian Research Council.

The InfoWare project investigates solutions for information sharing on MANETs used in emergency and

rescue situations. It aims to provide highly available services, security and knowledge management between different rescue teams. Little documentation and papers are available which explains the assumptions made below on the probable system behavior.

3.2.4.1 Functionalities

♦ *Data: type, structure and identification*

InfoWare manages "any kind of data that's storable as data structures in main memory, a file, or system internal tables" [SGP04]. This is much more flexible than file sharing middleware previously seen. However, like these middleware, InfoWare does not deal with data type and structure.

The available papers do not specify the nature of the sharing unit but it is probably the whole data.

One of the main particularities of InfoWare is to use ontology in order to associate semantic with data as explained in [SGM05]. It seems that data do not need to be identified through globally visible identifiers because they are only accessed through semantic researches.

♦ *Host identifier, presence and communities management*

Available published papers do not indicate how hosts are identified. But, since InfoWare uses a MANET routing protocol (AODV [PeR99], OLSR [CJL+01] or a secured variant), it may be based on IP address.

Regarding the range of the host presence service it seems that, at least, a list of one-hop neighbors is maintained as explained in [DPM05]. But if OLSR is used, routing tables could provide a view of hosts in the network partition. Intermediate solutions are also possible: for instance, knowing direct neighborhood and some particular hosts such as DENS nodes (described in the next section).

InfoWare provides a local event notification service, called *watchdog* (WD) and described in [SGP04]. It is probably meant to warn users of modification in the accessible host list.

InfoWare can associate a *profile* to each host and it intends to use them to build *groups* [LM3]of hosts. However, it does not indicate what these groups offer.

♦ *Data discovery and searching*

InfoWare seems to maintain two local lists of data:

- An almost real-time list of direct neighborhood data (like most other systems).
- A view of all data, even these which are not accessible. This view is constructed by merging during meeting of two hosts. So it can be partial and out-of-date.

The specificity of the partial overall view is that it is not a list of data but a "semantic overlay network" [SGM05]. Data searching is realized over this semantic view through XML queries.

InfoWare can also react to modifications in these two lists thanks to its local event notification service. In addition, it provides a publish/subscribe service called DENS (Distributed Event Notification Service). This service allows monitoring any event (such as data appearance) on other hosts. The subscription model is not defined yet in [SGP04] but it is based on content filter on the publisher side, controls access rights before installing and can guarantee persistency of its messages (i.e. at least one copy of each message reaches recipients).

♦ *Basic data management functions*

The current available papers do not explain how data is manipulated. InfoWare does not seem to provide any modification primitive. In return, reading can be done either locally or remotely (i.e. there is non need to

download a data in order to access it).

3.2.4.2 Non-functional properties

3.2.4.2.1 Data management

♦ *Consistency*

The current available papers do not speak about consistency. In fact, data seem to be either immutable or periodically updated by the same host (e.g. sensor measurement) with a validity date.

♦ *Persistency*

At the host level, data can be stored on any kind of memory (volatile or persistent). So data may not outlives the execution of the system.

At the network level, the persistency of some data seems crucial in a rescue situation (e.g. medical records). However, available published papers do not mention it.

♦ *Accessibility in mobile ad hoc network context*

As explained in [DPM05], InfoWare tries to predict future network partitions. In consequence, it claims to be able to maximize accessibility by replicating data depending on "local and remote resource information, group membership descriptions, and any other high-level information available".

3.2.4.2.2 Resource management in a limited capabilities environment

InfoWare deals with device heterogeneity. It provides a modular architecture in which many elements are optional. So limited resource devices can install and run only services they really need.

Energy saving is managed at two levels:

- Minimizing wireless communications. Most of InfoWare protocols have two versions: eager and lazy. The later is less reactive but need less exchanges and can be used when energy must be saved.
- Minimizing processor use. Symmetric cryptography is not computationally too expansive. In return, it needs to distribute secret keys which require wireless communications.

In term of memory, the use of ontology may be quite expensive since its size growths exponentially with its expressiveness.

3.2.4.2.3 Security

InfoWare provides individual authentication. It uses certificates delivered by a commonly trusted authority. The certificates are loaded before entering in the ad hoc network. It is not defined whether they identify devices or users. They are managed just above the MAC layer in order to protect the routing protocol.

Thanks to authentication, access rights can be controlled both in case of direct access and with the publish/subscribe service.

Confidentiality is addressed by a symmetric communication encryption. InfoWare designed the protocol SKiMPy [PAP+05] to share the secret key between authenticated hosts in a network partition.

3.2.4.3 Architecture

Figure 3-13 extracted from [PAD+04] illustrates the modular architecture of InfoWare:

- The Knowledge Manager manages the global semantic view of data (GDDD). Data searching is realized here (Query Management). It also stores profiles.

- The DENS is the publish/subscribe service. Some modules are optional depending on the role of the host: publisher and/or subscriber and/or DENS node (providing message persistency).
- The Watchdogs monitors local events.
- The Resources Manager monitors local and one-hop resources, tries to predict future network partition and proposes replication strategies. The two last functions are optional because they may be too much resource consuming.
- The Security and Privacy Management deals with security.

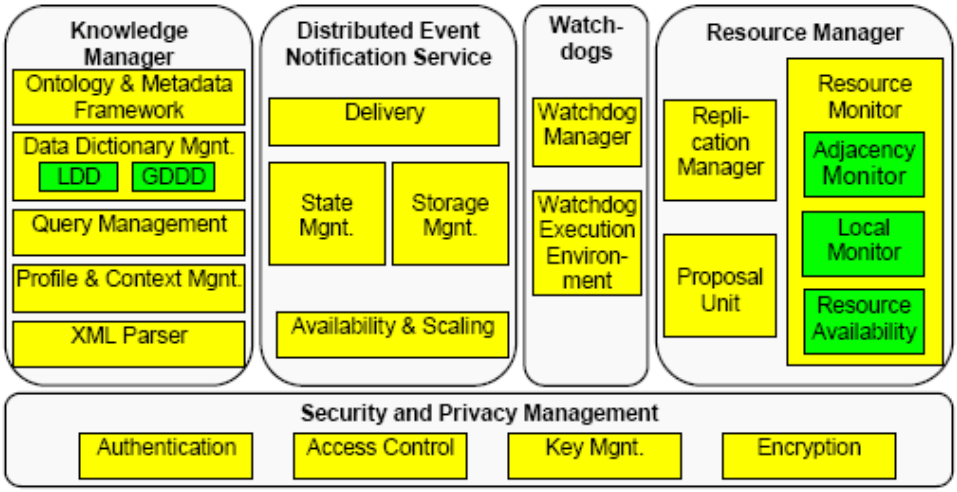


Figure 3-13: InfoWare architecture

InfoWare uses several kinds of communication architecture:

- The Security and Privacy Management uses a hybrid P2P model since it relies on a server to get certificates. But this server is only required before entering in the ad hoc network.
- The DENS uses a hierarchical pure P2P model because message delivery is assured by a subset of hosts. This backbone is dynamic (supports network partition) and may be composed of any host.
- Other modules use a flat pure P2P model.

InfoWare uses a MANET routing protocol: AODV, OLSR or a secured version. The choice is still open.

3.2.4.4 Synthesis and conclusion

InfoWare is designed for a stressful scenario: sharing information in rescue and emergency operations. It integrates many interesting services such as advanced data description (with ontology), network partition prediction, advanced event management and strong security.

Its main weakness is that it does not allow collaborative data edition.

3.3 Structured data based systems

The advantage of generic data based systems is to support any kind of data without modifying end-user application. Structured data based systems follow another philosophy: they define their own data structures. In return, they may associate additional information to the shared data and structure them in an efficient way in order to offer new and/or more adapted services. Several kinds of structured data are conceivable. PeerWare creates its own data with a very simple structure whereas XMIDDLE uses XML documents.

3.3.1 PeerWare

PeerWare [PEERWARE], [CuP01] is the result of a joint research effort led by Gianpaolo Cugola and Gian Pietro Picco (who also participated to the design of LIME), which has been carried out at Politecnico di Milano, Italy. PeerWare is the core of the MOTION platform designed for the MOTION IST project [MOTION] which lasted 30 months from February 2000. It was implemented in Java[LM4]. Its sources have been available on SourceForge.net [PEERWARE] through the GNU Lesser General Public License since Mars 2003.

Although data-centric, PeerWare is not a data sharing system. In fact, it aims to provide a minimal set of flexible and extensible primitives sufficient to support any kind of MANET applications. It claims to integrate the best of publish/subscribe and Linda models. As LIME, it also provides code mobility.

3.3.1.1 Functionalities

♦ *Data: type, structure and identification*

PeerWare supports structured data called *documents*. Within addition to its actual content, a document contains a *label* and a *header* in which basic [LM5] meta-information (e.g. an author name) may be stored.

Unfortunately, this structure is not very powerful and Peerware does not really exploit it. In fact, it does not permit a finer control over the data and the sharing unit is the whole document.

Documents are identified by their label and a full path, like in conventional file systems. A document can be associated with several directories[1]. For instance, this paper could be associated with "technical report", "MANET" and "data sharing". Thus a directory can have several valid full paths. Inside a directory, all items must have different labels. However, PeerWare does not explain how the uniqueness is maintained.

♦ *Host identifier, presence and communities management*

Hosts are identified by their IP address coupled with a random number.

Available published papers do not clearly state the scope of a host view (remote hosts can be detected at one or several hops or in the whole network partition). It seems that connected hosts are listed in a local table.

PeerWare does not manage communities of interest.

♦ *Data discovery and searching*

Hosts seem to maintain a local table of all accessible directories (stored locally or on connected hosts). However, they do not manage the equivalent table for documents. In addition, PeerWare does not provide specific function for searching data. Such a function have to be realized from the primitive **`execute`**(F_N,F_D,a) presented in the next section.

PeerWare also provides a publish/subscribe mechanism to discover documents. In fact, all item creation / deletion are automatically notified. In addition, the primitive **`publish`**(e,i) may be manually called to notify any other event e (e.g. an update) on a given item i. At the same time, the primitive **`subscribe`**(F_N,F_D,F_E,c) allows interested hosts to subscribe to events matching the filter F_E and concerning a document matching the filter F_D associated with a directory matching the filter F_N[LM6]. The expressiveness of these filters is not defined in the available papers. But F_D seems to be based on the meta-information of documents.

[1] In the PeerWare terminology, directories are called *nodes* and an *item* refers to either a document or a directory.

♦ *Basic data management functions*

Like the Federated Tuple Space of LIME, PeerWare provides a transient data sharing space, called GVDS (Global Virtual Data Structure) which superposes all directories and documents shared by currently connected hosts. For example, let's take a network partition constituted by hosts X and Y. Imagine that X has a single document d_1 in the directory n_1 and Y also has a single document d_2 in the same directory. Then, in GVDS, the directory n_1 will contain both d_1 and d_2. Contrary to LIME, PeerWare distinguishes the local space from GVDS for efficiency reasons.

PeerWare allows sharing a new item via **`insertNode`**(n,n_f) which inserts the new directory n into the local directory n_f and **`insertDoc`**(d,n_f) which inserts the new document d into n_f. In addition, a document can be associated to several directory with the primitive **`placeIn`**(p,n_f) which links the document pointed by the path p with the directory n_f. Connected hosts can access to all these items.

There is no specific primitive for reading, modifying and synchronizing documents. Instead, PeerWare provides the general function **`execute`**(F_N,F_D,a) which executes the action a over the set of items I matching the filters F_N and F_D. Finally I is returned. A host can use this primitive either over its local space or over the GVDS. This is a blocking operation and **`executeAsynch`** is a probe variant. Another variant which combines **`execute`** and **`subscribe`** is **`executeAndSubscribe`**(F_N,F_D,F_E,a,c).

PeerWare allows removing a local directory n thanks to **`removeNode`**(n) and a local document d from the directory n via **`removeFrom`** (d,n). If there is no other link to d into other directories, d is physically deleted.

3.3.1.2 Non-functional properties

3.3.1.2.1 Data management

♦ *Consistency*

Since PeerWare is designed to provide a minimal set of primitives in order to be integrated into a more specialized platform (initially, the MOTION platform), it does not deal with non-functional properties specific to data sharing. Thus it does not deal with consistency of data. It is not even sure that this problem is addressed with the MOTION[LM7] platform.

♦ *Persistency*

On hosts, documents are stored on lightweight persistent memory.

Persistency is not guaranteed from the distributed point of view.

♦ *Accessibility in mobile ad hoc network context*

PeerWare does not manage accessibility (it seems that documents are not replicated).

3.3.1.2.2 Resource management in a limited capabilities environment

PeerWare does not deal with heterogeneity and limited capacities.

3.3.1.2.3 Security

PeerWare does not deal with security. As shown in Figure 3-14, it relies on the security module of the MOTION platform which provides authentication, access control and confidentiality.

3.3.1.3 Architecture

Available published papers only describe the MOTION platform architecture. This one is illustrated by

Figure 3-14 extracted from [CuP02]. We can see that PeerWare is the core of the MOTION platform. Its main component "Publish/Subscribe and search" provides publish/subscribe and filter services. The module "Repository adapter" offers an interface to access the local and virtual (GVDS) spaces. The security interface has not been implemented. And the code mobility is supported by µcode [MUCODE] which has been designed with LIME.

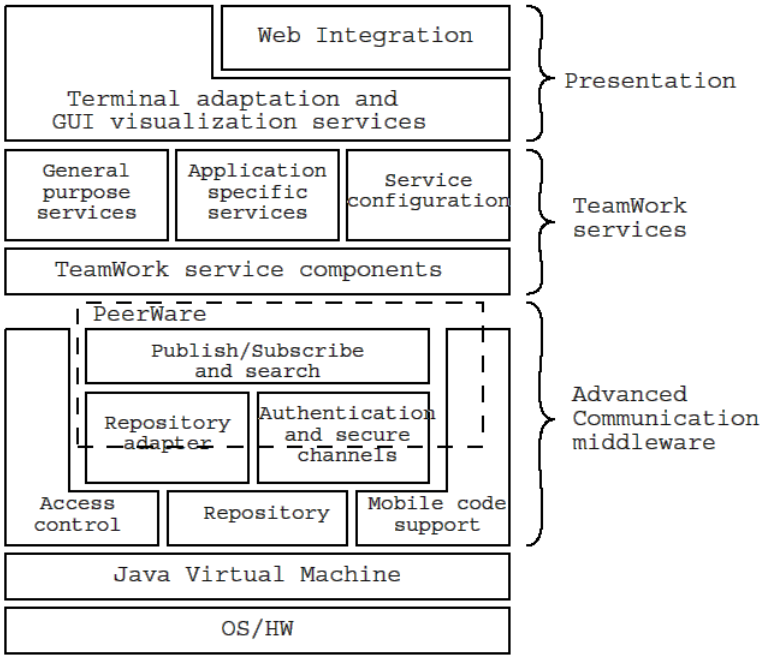


Figure 3-14: The MOTION platform and PeerWare architecture

In PeerWare, every host has the same behavior. So PeerWare is a flat pure peer-to-peer system. However, it relies upon MOTION modules which have been designed for mesh networks. Such a network is a specific MANET with a core of fixed hosts generally less resource constrained. These hosts are relatively stable and reliable. They can be used as servers. So the MOTION platform is not a true P2P system.

3.3.1.4 Synthesis and conclusion

Peerware meets its initial goal of providing a minimal set of primitives for collaborative work in MANETs. Consequently, difficult problems are left to the applications development.

Note that an interesting PeerWare idea is to associate some meta-information to the content inside documents.

3.3.2 XMIDDLE

XMIDDLE [XMIDDLE], [MCZ+02] [Mus02] was designed and developed in the Department of Computer Science of University College London. The first publication is dated from 2001 and the last one from 2002. XMIDDLE is implemented in Java. Its sources have been available on SourceForge.net [XMIDDLE] through the GNU Lesser General Public License since May 2000.

XMIDDLE is a middleware designed to share XML documents between heterogeneous devices on wireless ad hoc network. It offers both on-line and off-line access to shared documents.

3.3.2.1 Functionalities

♦ *Data: type, structure and identification*

The particularity of XMIDDLE is to manipulate XML documents (*eXtended Markup Language*) [XML]. This kind of structured document has two main advantages. On the one hand, it allows content to be enhanced with semantics (e.g. improving search mechanisms, and possibly synchronization mechanism). On the other hand, it offers a tree structure as shown in Figure 3-15 extracted from [MCZ+02].

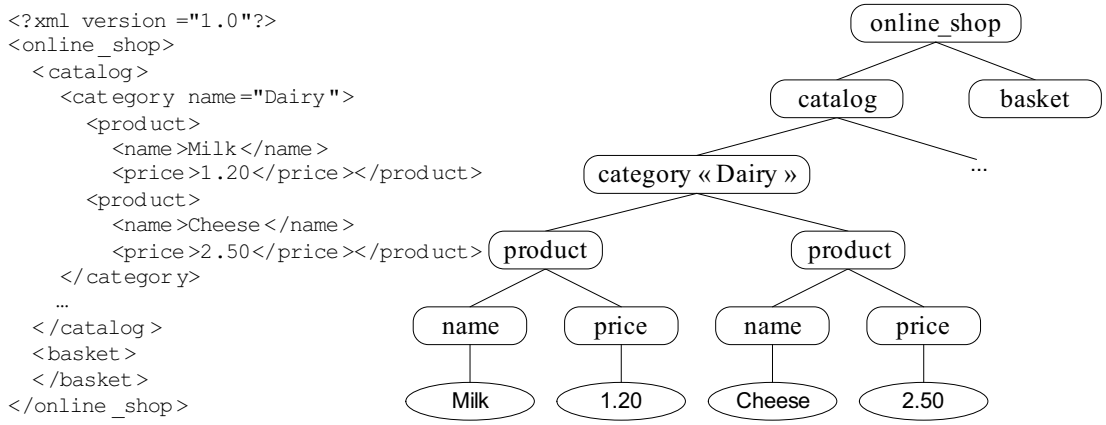


Figure 3-15 : Tree structure of XML documents

The tree structure provides consistent splitting for documents. Each branch of the tree corresponds to an autonomous part of the document. So, in XMIDDLE, it is possible to work on a document fragment.

A document (or a fragment) is identified, on the one hand, by the name of the host that created it and, on the other hand, by an XPath link [XPath]. For instance the identifier of the branch `category "dairy"` could be: `("pc_i.home.net", /online_shop/catalog/category[@name="Dairy"])`.

♦ *Host identifier, presence and communities management*

For host identification, unlike many other systems, XMIDDLE does not use IP addresses. Instead, it uses two types of identification credentials: the primary ID and the secondary ID. The primary ID is assumed to be unique (e.g. a MAC address). In the current implementation, XMIDDLE uses a random integer. The secondary ID is used to identify hosts in a user-friendly manner. It is not required to be unique. So it is not used for host identification. The current implementation uses hostnames as secondary IDs.

XMIDDLE maintains a table **`InReach`** which contains the list of visible hosts in direct communication range. This table is accessible by the primitive **`getHosts`**. Modifications are not monitored.

XMIDDLE does not manage communities of interest. The organization of documents into communities of interest was originally scheduled in a future release of XMIDDLE. Unfortunately, it seems that it was realized.

The primitives **`disconnect`**() / **`connect`**() allow switching off/on the presence service.

♦ *Data discovery and searching*

The table **`InReach`** contains for each accessible host, their list of shared documents (their XPath link).

The available published papers do not mention whether XMIDDLE provides data searching services or not.

♦ *Basic data management functions*

Notice: as already said, XMIDDLE manages XML documents which have a tree structure. So, in the rest of this chapter, the term “shared document” can refer to a whole document or to a fragment (i.e. a branch).

The creation of a shared document *d* is realized in two steps. First, the document must be locally created thanks to a DOM (*Document Object Model*) [DOM] engine which is independent of XMIDDLE. The host which created *d* is called the *owner* of *d*. Then, *d* is shared via the primitive **`export`**. In fact, this one adds the XPath link of *d* to the table **`ExportLink`**. On the host X, this table indicates all documents that are owned and shared by X. In addition, if X is in connected state, **`ExportLink`** is known from all its neighbors.

In order to access a document *d*, a host Y must call the primitive **`link`** which creates a local copy. Then *d* is memorized into the table **`LinkedFrom`** of Y and into the table **`LinkedBy`** of the owner of *d*. The table **`LinkedBy`** inventories for each document all hosts which have copied it and which are currently connected. The table **`LinkedFrom`** indicates for each document which host owns it (even if this one is not accessible).

Once, the shared document is locally replicated, read and update accesses are realized in a traditional file system way. It means that the document must be firstly opened with the primitive **`open`**. Then the second step depends on the nature of the access (read or write) but is always accomplished locally thanks to DOM primitives (such as **`firstChild`**, **`parentNode`**, etc.). Finally, the document is closed with the primitive **`close`**. The **`open`** and **`close`** primitives isolate the host work from other host updates.

In order to remove a replica XMIDDLE provides the primitive **`unlink`**. For instance, if the host Y does not use *d* any longer, it begins by removing its local copy (thanks to DOM primitives) then calls **`unlink`**. This primitive immediately removes the couple ⟨X ; *d*⟩ from the table **`LinkedFrom`** of Y. In addition, it will also remove the couple ⟨Y ; *d*⟩ from the table **`LinkedBy`** of X as soon as X and Y meet again. Available published papers do not mention what happens if a document is deleted by its owner.

3.3.2.2 Non-functional properties

3.3.2.2.1 Data management

♦ *Consistency*

XMIDDLE provides two kinds of consistency, a bit like AdHocFS. From a distributed system point of view, all replicas are eventually consistent (according to the Read-your-writes model). Since in this case, divergences are unavoidable, XMIDDLE is able to detect and repair conflicting updates. This mechanism, based on versioning, takes advantage of the structure and semantic of tree documents for automating the reconciliation. So end-user should never be solicited. In order to minimize divergences, the reconciliation is realized as soon as possible: when two hosts storing a copy of a common[1] sub-branch meet.

In the direct communication range of the owner of a document *d*, XMIDDLE offers a uniform pessimistic consistency forbidding any data divergence. More precisely, the owner of *d* behaves as a server to order all operations and guarantee a sequential consistency[2] to all replicas of *d*.

♦ *Persistency*

In XMIDDLE, hosts only store documents on volatile memory.

At the network level, document persistency is assured by the original copy stored in the owner which seems to never delete its copy (at least, as long as it does not switch off).

---

[1] If the host h_a stores `/a/b` and the host h_b stores `/a/b/c`, the common sub-branch to reconcile is `/a/b/c`.
[2] Consistency is not stricter than the sequential model because of possible delays introduced by communication and owner's response.

♦ *Accessibility in mobile ad hoc network context*

The XMIDDLE accessibility is influenced by the dynamic topology of MANETs because hosts can only view document on one-hop neighbors. XMIDDLE does not have hoarding mechanism like Coda but each accessed document is locally replicated to be accessible in future. In addition, in disconnected mode, the weak consistency allows both read and write operations.

3.3.2.2.2 Resource management in a limited capabilities environment

The main advantage of XMIDDLE is to use XML documents. However, this has several drawbacks. First, it requires a DOM engine which demands computation power and memory (that may be scarce resources in mobile devices).

XML documents have good and bad sides. On the one hand, its tree structure makes it easier to work on a part of the document. But, on the other hand, the reconciliation mechanism must associate several versions to all shared parts of a document which involves much redundancy and is very expensive in terms of storage.

XMIDDLE does not provide energy saving mechanism. However, a host can easily get into disconnected state to decrease its energy consumption by switching off its wireless interface.

3.3.2.2.3 Security

XMIDDLE does not deal with security. It only provides private spaces.

3.3.2.3 Architecture

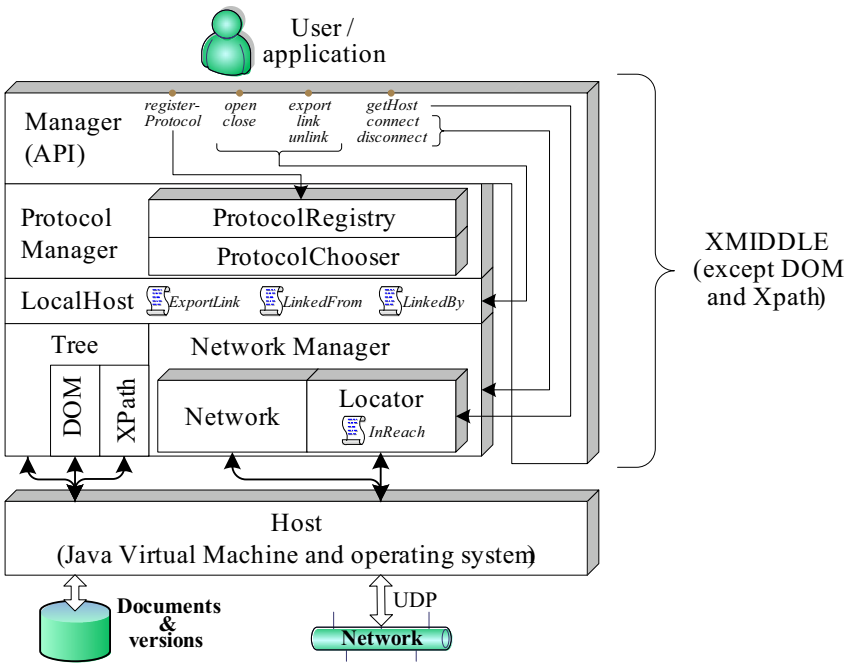


Figure 3-16: XMIDDLE architecture

Figure 3-16 presents the architecture of XMIDDLE. The Manager corresponds to the API. It provides user applications with all the above described primitives. These primitives directly access the three sub-layers which compose XMIDDLE.

The lowest layer concerns the resource access and is divided in two modules. The "Tree" module deals with the local documents operations. It leans on Apache Xerces DOM and Xalan XPath for local operations. It also manages the document parts versions. The Network Manager is the second module of this layer. It is composed

of two sub-modules. The "Locator" module maintains an up-to-date version of the **`InReach`** list of visible hosts by listening their periodical beacons. The "Network" layer handles all connections and receives and sends data.

The upper layer comprises a single LocalHost module that provides a higher level view of the local machine. It manages shared documents by maintaining the tables **`LinkedFrom`**, **`LinkedBy`** and **`ExportLink`**.

Finally the Protocol Manager layer allows registering a new protocol and then chooses the best one according to the context. A protocol is, for instance, a linking or reconciliation mechanism.

The current implementation of XMIDDLE uses UDP socket to communicate both in unicast and multicast.

The type of interaction architecture used in XMIDDLE depends on the consistency model employed. With weak consistency, hosts work in a flat pure P2P way. With strong consistency, they work in a hierarchical pure P2P way.

3.3.2.4 Synthesis and conclusion

XMIDDLE takes advantage of XML documents. It shows that structured document can be used to insert meta-information like semantic. Here it allows improving accuracy of automatic reconciliation. We can also imagine to include other information like versioning which are stored in XMIDDLE separately.

The structure of these documents also proved to be useful when multiple writers want to update different parts of the same document. The counterpart is that concerning limited capabilities, this structure seems to have more disadvantages than advantages.

4 Comparison and discussion

Eleven systems have been analyzed in the previous section, namely LIME, LIMONE, TOTA, Coda, Ficus, Rumor, Roam, AdHocFS, Ad-Hoc InfoWare, PeerWare and XMIDDLE. All these analyses have followed the analysis chart introduced in Section 2. This will facilitate the comparison between the studied systems.

The remainder of this section is organized as follows: we introduce the studied systems according to their target context. We then draw a comparison of the eleven systems following the analysis guideline used for systems presentation. We conclude with a discussion of existing solutions for functional and non-functional properties and an attempt to identify missing features in existing systems.

4.1 System target context

Although this paper focuses on data-sharing middleware for MANETs, not all studied systems were designed for this specific context. In fact, even older and less-constrained environments may provide interesting solutions (possibly after adaptation) for mobile ad hoc networks. Figure 4-1 shows a grouping of the data sharing systems according to their underlying network (mobile wired network, mesh network and MANET).

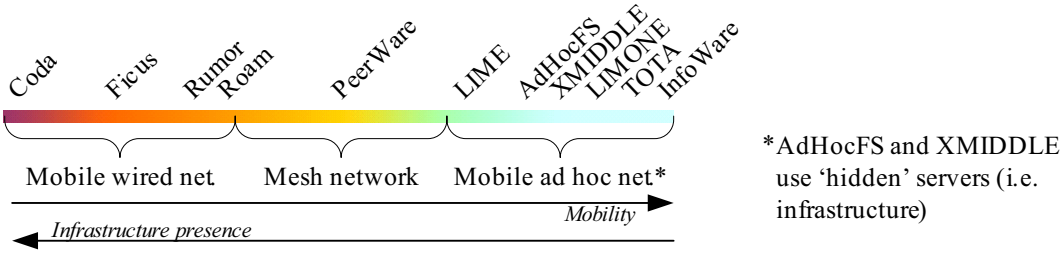


Figure 4-1: Grouping of systems according to their underlying network

During the system comparison, we must keep in mind that systems are not always directly comparable because not implemented for the same target networks.

4.2 Comparison

The comparison of all systems in this paper reveals that functional properties are almost always handled. On the contrary, non-functional properties are much more unequally considered. In addition, no system proposes a complete solution which enables to work in a collaborative and secured way with data shared spontaneously on a mobile ad hoc network composed of heterogeneous and resource constrained devices.

The annex sums up this comparison in a table.

4.2.1 Functional properties

4.2.1.1 Host identifier, presence and communities management

All systems but XMIDDLE use the IP address to uniquely identify hosts. This is realistic neither in MANETs where IP addresses may be duplicated, nor in mobile wired networks where IP addresses may change dynamically. Using random numbers, as it is done in XMIDDLE, is not more relevant. A potential solution, often mentioned in the description of the studied systems, is to use the MAC address.

In MANETs, all systems but LIME (i.e. LIMONE, TOTA, AdHocFS, InfoWare and XMIDDLE) provide a presence service detecting only direct neighbors. LIME claims to detect all hosts in the same partition. However, it does not support really dynamic environments. A particularity of LIMONE and AdHocFS is to restrain the neighboring view according to specific criteria (e.g. AdHocFS constructs fully connected groups of hosts belonging to the same Security Domain). On the contrary, InfoWare increases this limited view by maintaining an extended view possibly out of date and incomplete.

Few systems manage communities of interests: LIME, Coda, Ficus, Rumor and Roam offer an indirect way to simulate CoI. Only LIMONE, AdHocFS, InfoWare and PeerWare (via MOTION) really deal with CoI. In addition, AdHocFS CoIs are much less flexible than others since they are managed by an external server.

4.2.1.2 Data: structure, type and identification

As explained in the introduction, we distinguished between three kinds of data structures:

- The tuple structure (LIME, LIMONE and TOTA) that is well suited for synchronization and context-awareness but not for collaborative work on documents. However, TOTA enhances the basic tuple structure with propagation rules which is an interesting idea for data-sharing.
- The generic structure used in Coda, Ficus, Rumor, Roam, AdHocFS and InfoWare. All these

systems but InfoWare practically use files. This potentially makes reutilization of existing applications easier. In any case, it is well adapted to data-sharing.

- The structured documents (PeerWare and XMIDDLE) may embed some semantic information. In XMIDDLE they also offer a dynamically configurable fine-granularity sharing unit. However, they have many counterparts in a resource constrain environment.

Only TOTA directly uses data types. It even classifies them hierarchically. However, this seems more a programming trick for making development of propagation rules easier.

The unique identification of data may be based on:

- The content (LIME and LIMONE) or its description (InfoWare). This is an interesting semantic approach. But, it does not allow a given data to be precisely identified. So it requires a specific kind of collaborative application.
- A number or a path relative to the host which created the data: TOTA and XMIDDLE.
- A number or a path relative to a well-identified shared space: Coda, Ficus, Rumor, Roam, AdHocFS and PeerWare. This space must be at least as persistent as the data.

4.2.1.3 Data discovery and searching

Most systems do not maintain a local list of accessible data. LIME, LIMONE, PeerWare, Ficus, Rumor and Roam must flood accessible hosts at each data search. However, this flooding can be limited with filters either on hosts (LIME and LIMONE) or on directories (PeerWare, Ficus, Rumor and Roam). In addition, LIME, LIMONE and PeerWare provide a reaction mechanism to be notified when a specified data appear on a remote host. In Coda a list of accessible data is maintained by a server.

TOTA, AdHocFS, InfoWare and XMIDDLE maintain a local list of accessible data. In this list, TOTA stores a full copy of each data, XMIDDLE only stores the data identifier and AdHocFS and InfoWare stores the data identifier and a link toward some replicas.

The expressiveness of data search mainly depends of the data structure: pattern matching for tuples and file name for files. A particularity of InfoWare is to manage ontology for semantic data searching.

4.2.1.4 Data management functions

All systems allow sharing and reading a data. Modifications are impossible in TOTA and InfoWare.

All systems but TOTA (and InfoWare?) permit the deletion of a local data replica. When data replication is supported (this excludes LIME, LIMONE and PeerWare), the deletion of a data (i.e. all its replicas) is only provided by Ficus, Rumor and Roam which implement variations of a complex distributed garbage collector.

Coda, Ficus, Rumor, Roam, AdHocFS and XMIDDLE offer the same synchronization primitives (**`open`** and **`close`**). During a session (between an **`open`** and a **`close`**), remote modifications are ignored and local modifications are not propagated.

4.2.2 Non-functional properties

4.2.2.1 Data management

4.2.2.1.1 Consistency

Among all systems only six provide reusable consistency models. Other systems either forbid replication

(LIME, LIMONE and PeerWare) or forbid updates (InfoWare) or are too specific (TOTA).

The remaining systems are Coda, Ficus, Rumor, Roam, AdHocFS and XMIDDLE. Notice that this corresponds to the systems which use synchronization primitives. These systems enforce the Read-Your-Write consistency model with some enhancements:

- Coda and Ficus notify hosts detaining a replica when the data is modified.
- Ficus, Rumor and Roam use a (1 or 2 level) ring topology to schedule reconciliations.
- AdHocFS and XMIDDLE combine this optimistic consistency with a pessimistic one which is only applied in some specific conditions and inside a geographically restrained area.

4.2.2.1.2 Persistency and accessibility

In a MANET, persistency and accessibility require data replication. LIME, LIMONE and PeerWare (it seems) do not enable data replica creation.

Among systems allowing replication, only Ficus, Rumor, Roam and AdHocFS (inside a local group) use a distributed mechanism to prevent data disappearance. They try to maintain a minimal number of replicas for each data. The use of a reliable server for guaranteeing data persistency (as Coda does) is not a good solution in MANETs. Yet, it is implicitly done by AdHocFS (with the controller of security domain) and XMIDDLE (with the data owner).

All systems enabling replication (except TOTA and perhaps InfoWare) locally replicate data when they are accessed. In addition, Coda, Ficus, Rumor and Roam provide a hoarding mechanism to download data potentially accessed in future. All these mechanisms are individualistic: hosts do not cooperate to maximize accessibility globally according to link reliability and network volatility.

4.2.2.2 Security

Generally, security means authentication, access control and confidentiality. Sometimes, integrity is also handled but non-repudiation is never provided.

Most systems offer no security (LIME, TOTA and XMIDDLE) or delegate it to another middleware (MOTION for PeerWare and Truffle for Ficus, Rumor and Roam). LIMONE provides access control and can implement confidentiality and integrity but without authentication these mechanisms have no real meaning. Authentication seems always guaranteed by using certificates, notably in AdHocFS and InfoWare. Although conceivable in some scenarios, this cannot be applied in all use cases of MANETs. InfoWare is the only system which builds a GKA (Group Key Agreement) protocol for sharing a secret key inside a MANET partition.

4.2.2.3 Resource management in a limited capabilities environment

4.2.2.3.1 Bandwidth and energy

Some systems are more or less bandwidth saving. AdHocFS optimized all its protocols. InfoWare proposes a lazy version for all its mechanisms. XMIDDLE may save bandwidth by allowing sharing only a piece of a big data. And Roam uses a reconciliation topology geographically restricted.

Others are often costly (mainly LIME, TOTA, Coda and Ficus). Notably, an expensive mechanism is the notification of updates when it is systematic and over long distances.

4.2.2.3.2 Memory

AdHocFS takes into account free memory of each host in a group inside several protocols. It does not

explicitly save memory contrary to Coda and Roam which compress their update logs.

The version logs of XMIDDLE are expensive since potentially redundant. But the most memory consuming system is TOTA which maintains on each host a copy of all accessible data.

4.2.2.3.3 Computation power

AdHocFS and InfoWare are able to encrypt all their communications. However, they do not require a high computational power since they use a symmetric encryption. On the contrary, the DOM engine of XMIDDLE is CPU (and memory) consuming.

4.3 Discussion

In this section, we do a synthesis of the comparison drawn in the previous section.

Although it is not required for sharing data, the publish/subscribe communication model seems very interesting for wireless systems. Besides, 5 out of the 7 wireless middleware use such a mechanism to process local and/or remote events (like data creation, data modification, host (dis)appearance…). Moreover, this mechanism provides asynchronous communications to contact hosts in another network partition. This functionality may not be directly used for data sharing: it is nevertheless made available to end users.

The Linda-like data management is more convenient for coordination than for bulky data sharing. In fact, it is particularly well adapted to support context-aware applications. Unfortunately, the Linda primitives (**`in`**, **`rd`** and **`out`**) are not compatible with most of the current applications which prefer the tradition file system primitives (**`open`**, **`read`**, **`write`** and **`close`**). In addition, the Linda model tends to prevent the replication of data required in MANETs. By the way, this is a major weakness of LIME and LIMONE (not of TOTA but it does not provide all Linda primitives).

The support of structured data enables advanced features (with a fine-granularity sharing unit and the use of semantic information). However, XMIDDLE has raised a performance problem when it shows that XML documents may be too memory consuming for limited resource devices. In fact, it is possible to get around the difficulty by designing more adapted structure, for instance, by separating data from the structure description. Another issue of using structured data is that it implies the modification of existing applications or the provision of an adaptation layer. Yet, this adaptation should be rather small since more and more applications already use structured data. For instance, the Microsoft Office 2007 suite will use the Office Open XML format [OOXML]. It will also support the OpenDocument format [ODF] developed by the OASIS [OASIS] consortium. These two formats are structured data being standardized. In addition, OASIS' goal is to standardize open structured formats for any kind of application.

The simplest (but nevertheless efficient) mechanism for maximizing accessibility is the automatic local replication when a user accesses a remote data. The hoarding is a much more complex mechanism for anticipating future accesses. However, its performance seems very dependant on the user behavior and type of application. AdHocFS is the only system that uses a collaborative mechanism for maximizing accessibility between the members of a local neighborhood. Other research teams [Har01] [YiC04] [JYX+04] [SHH+04] [ChN01] have also developed collaborative protocols to increase accessibility. All these protocols rely on the assumption that several hosts which collaborate may achieve a higher accessibility level than if they act independently because of their limited resources.

All systems allowing data replication and write accesses support optimistic consistency. The most often

used model is the Read-Your-Writes consistency or a variation. However, AdHocFS and XMIDDLE enforce their own model with the goal of taking the best of pessimistic and optimistic consistencies. These hybrid models behave pessimistically within a close neighborhood and optimistically with the rest. This is interesting since it may prevent many conflicts at a low cost. However, even these advanced models cannot cover efficiently all possible scenarios. For instance, in a very dynamic network, the hybrid consistency will not be able to form close neighborhood sufficiently stable to be exploited. In this context, a classical optimistic consistency will be more efficient. So the best solution is perhaps to propose several models and indicate users when to use them.

It is interesting to notice that many useful features of AdHocFS (specifically the collaborative accessibility and the hybrid consistency) come from the logical partitioning of hosts. In fact, clustering a network provides some stability to upper applications. In [MM06], several mechanisms are described for building and maintaining clusters. However, the authors conclude that these techniques are not appropriate when networks are too dynamic.

Finally, security is the most lightly treated property. The base service is authentication since it is required by others security services like access control, confidentiality, integrity and non-repudiation. Almost all the analyzed systems as well as other research works like [CBH03] and [KZL+01] tackle the problem in a classical way by authenticating each user individually. However, AdHocFS proposes another approach by authenticating groups of users rather than users. Users can prove their membership of a group. And members of a group trust themselves. This authentication is sufficient for most security services required by data sharing (however, it does not allow non-repudiation). In addition, it probably scales better. Unfortunately, AdHocFS uses a centralized mechanism for group authentication. This allows a strong security but greatly limits the spontaneity of data sharing. In order to distribute such a mechanism, it is possible to use some group key agreement protocols [Bah03]. Very simple protocols like (physical) hand to hand key exchange are also suitable for small networks.

5 Synthesis and conclusion

The most complete system is AdHocFS which offers almost all functional and non-functional properties. Unfortunately, some of its mechanisms bear on a server which must be contacted from time to time. InfoWare, although less complete, provides additional features. The main ones are the use of ontology which associates strong semantic with data and a persistent publish/subscribe service which is able to propagate events outside a network partition. XMIDDLE has many drawbacks but its use of XML documents is very instructive about pros and cons of structured data in general and XML data in particular. Coda, Ficus, Rumor and Roam, although designed for wired network, provide interesting mechanisms for data management (consistency, persistency and accessibility) which can be adapted to MANETs. Finally, tuple-based systems do not seem general enough except TOTA for its propagation rules.

In conclusion, in our opinion, the "ideal" data sharing middleware for MANETs would supply the classical UNIX-like primitives for manipulating structured data. It would maintain local tables in order to limit expensive researches across the network. In addition, data search could use semantic information associated to data for getting more accurate results. This "ideal" middleware would implement a collaborative replication mechanism with the goal of maximizing the global use of memory. It would also propose several consistency models so that the user could choose the most efficient one according to the current situation. Among these models, the

middleware should offer a hybrid consistency and select it by default. In addition, all these consistency models could take advantage of the structure of data. Both replication and consistency protocols would be energy aware. It means they would be able to decrease the quality of their service for minimizing energy consumption. Finally, the middleware would enforce a group (rather than user) authentication. Besides it would manage data access rights according to authenticated groups. If users may belong to several groups, it should be possible to define fine access rights over data.

Today, in the current state of the art, many mechanisms of the "ideal" system remain to be implemented. Concerning data replication, there exists many implementations of individualistic mechanisms (greedy or by hoarding). Collaborative mechanisms are often only evaluated by simulation and hardly implemented. Concerning data consistency, the sequential and Read-Your-Write models have been studied and implemented. Some hybrid models have been proposed but other models (with different properties) may be specified and implemented. Finally concerning access rights, no work has been published to specify and evaluate a fine management based on authenticated groups.

ACKNOWLEDGEMENTS

This survey has benefited from the expertise of Julien N. and Cédric T. in the conclusions relative to the state of the art of security in MANETs.

The authors are specially thankful to Valérie Issarny for her valuable comments about AdHocFS. They also thank André C. and Jérémie L. for their careful global reviewing.

REFERENCES

[ADHOCFS] http://www-rocq.inria.fr/arles/work/AdHocFS.html

[AMR03] Nevine ABOUGHAZALEH, Robert N. MAYO and Parthasarath RANGANATHAN, "Idle Time Power Management for Personal Wireless Devices". In HP technical report, 2003 http://www.hpl.hp.com/techreports/2003/HPL-2003-102.pdf

[Bah03] Raghav BHASKAR, " Group Key Agreement in Ad hoc Networks". INRIA Research Report, RR-4832, May 2003.

[Bak] David BAKKEN, "Middleware". Chapter in Encyclopedia of Distributed Computing, J. Urban and P. Dasgupta, eds., Kluwer Academic Publishers, to appear. http://www.eecs.wsu.edu/~bakken/middleware.pdf

[$BI03_a$] Malika BOULKENAFED, Valérie ISSARNY, "AdHocFS: Sharing Files in WLANs". In proc. of the Second IEEE International Symposium on Network Computing and Applications, p.156, April 16-18, 2003

[$BI03_b$] Malika BOULKENAFED, Valérie ISSARNY. "A Middleware Service for Mobile Ad Hoc Data Sharing, Enhancing Data Availability". In Proc. of the 4th ACM/IFIP/USENIX International Middleware Conference. June 2003, Rio de Janeiro, Brazil.

[Bou03] Malika BOULKENAFED. "Gestion de l'accès aux données dans les réseaux sans fil en mode ad hoc". PhD thesis report, November 2003, University of Paris 6, France.

[CBH03] S. CAPKUN, L. BUTTYÁN and J. P. HUBAUX, "Self-Organized Public-Key Management for Mobile Ad Hoc Networks". In IEEE Transactions on mobile computing, 2003. 2(1): p. 1-13.

[CCL03] Imrich CHLAMTAC, Marco CONTI and Jennifer J-N. LIU, "Mobile ad hoc networking: imperatives and challenges". In Ad Hoc Networks, vol. 1, no. 1, pp. 13-64, 2003.

[CeF78] Lucien M. CENSIER, Paul FEAUTRIER. "A New Solution to the Cache Coherence Problem in Multicache Systems". In IEEE Transactions on Computers December 1978 pp 1112-1118

[CFL^+06] Giacomo CABRI, Luca FERRARI, Letizia LEONARDI, Marco MAMEI and Franco ZAMBONELLI, " Uncoupling Coordination: Tuple-based Models for Mobility ". In book "Mobile Middleware" (CRC Press, London-UK), Paolo Bellavista and Antonio Corradi editors, May 2006. http://polaris.ing.unimo.it/MOON/papers/pdf/MMB05.pdf

[ChN01] Kai CHEN and Klara NAHRSTEDT, "An integrated data lookup and replication scheme in mobile ad hoc networks". In Proc. of SPIE International Symposium on the Convergence of Information Technologies and Communications (ITCom 2001), August 2001.

[CJL^+01] T. CLAUSEN, P. JACQUET, A. LAOUITI, P. MUHLETHALER, A. QAYYUM and L. VIENNOT. "Optimized link state routing protocol". In Proc. of IEEE International Multi-topic Conference (INMIC), 2001

[CKB^+03] Ugur CETINTEMEL, Peter J. KELEHER, Bobby BHATTACHARJEE, Michael J. FRANKLIN. , Deno: "A Decentralized, Peer-to-Peer Object-Replication System for Weakly Connected Environments". In IEEE Transactions on Computers, vol. 52, no. 7, pp. 943-959, July 2003.

[CODA] http://www.coda.cs.cmu.edu/

[CoM99] S. CORSON and J. MACKER, "Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations". RFC 2501, Internet Engineering Task Force, Network Working Group, January 1999

[Cor97] Thierry CORNILLEAU. "Etudes des Cohérences Mémoire Uniformes - Cohérence Causale : mise en oeuvre sur CHORUS et extensions". PhD Thesis. Cnam. Paris. Janvier 1997.

[CuP01] Gianpaolo CUGOLA and Gian Pietro PICCO, "PeerWare: Core Middleware Support for Peer-To-Peer and Mobile Systems". Technical report, Politecnico di Milano, May 2001. Submitted for Publication.

[CuP02] Gianpaolo CUGOLA and Gian Pietro PICCO, "Peer-to-peer for Collaborative Applications". In Proc.of the International Workshop on Mobile Teamwork Support, co-located ICDCS'02, Vienna, Autria, IEEE press, pp. 359-364, July 2002.

[DOM] http://www.w3.org/DOM/

[DPM05] Ovidiu Valentin DRUGAN, Thomas Peter PLAGEMANN and Ellen MUNTHE-KAAS, "Building resource aware middleware services over MANET for rescue and emergency applications". In the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications, International Congress Center (ICC), Berlin, Germany, September, 2005

[DPS+94] A. DEMERS, K. PETERSEN, M. SPREITZER, D. TERRY, M. THEIMER, and B.WELCH. "The Bayou architecture: Support for data sharing among mobile users". In Proc. of IEEE Workshop on Mobile Computing Systems & Applications, Dec. 1994.

[DSB86] M. DUBOIS, C. SCHEURICH, and F. A. BRIGGS, "Memory access buffering in multiprocessors". In Proc. of the Thirteenth Annual International Symposium on Computer Architecture, pages 434–442, June 1986.

[FeN01] Laura Marie FEENEY and Martin NILSSON, "Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment". In Proc. of IEEE Conference on Computer Communications (IEEE InfoCom), Anchorage AK, USA, April 2001

[FRH04] Chien-Liang FOK, Gruia-Catalin ROMAN and Gregory HACKMANN, "A Lightweight Coordination Middleware for Mobile Computing". In Proc. of the 6th International Conference on Coordination Models and Languages (Coordination 2004), De Nicola, R., Ferrari, G., and Meredith, G., (editors), Lecture Notes in Computer Science 2949, Springer-Verlag, Pisa, Italy, February 2004, pp. 135-151.

[Gel85] David GELERNTER. "Generative Communication in Linda". In ACM Computing Surveys, 7(1):80–112, Jan. 1985.

[GHM+90] Richard G. GUY, John S. HEIDEMANN, Wai MAK, Thomas W. PAGE, Jr., Gerald J. POPEK, and Dieter ROTHMEIER, "Implementation of the Ficus Replicated File System". In Proc. of the Summer 1990 USENIX Conference, Anaheim, CA, June 1990, 63-71. 24

[GLL+90] K. GHARACHORLOO, D. LENOSKI, J. LAUDON, Phillip GIBBONS, Anoop GUPTA, and John HENNESSY. "Memory consistency and event ordering in scalable shared-memory multiprocessors". In Computer Architecture News, 18(2):15–26, June 1990.

[Goo89] James R. GOODMAN. "Cache consistency and sequential consistency". Technical Report 61, SCI Committee, March 1989.

[GPP93] Richard G. GUY, Gerald J. POPEK, and Thomas W. PAGE, Jr. "Consistency Algorithms for Optimistic Replication". In proc. of the First International Conference on Network Protocols. IEEE, October 1993.

[GPV+99] Erik GUTTMAN, Charles PERKINS, John VEIZADES, and Michael DAY, "Service location protocol, version 2". RFC 2608, June 1999.

[GRR+98] Richard G. GUY, Peter H. REIHER, David Howard RATNER, Michial GUNTER, Wilkie MA, and Gerald J. POPEK, "Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication". In proc. of Mobile Data Access Workshop, November 1998

[Gut99] Erik GUTTMAN, "Service location protocol : Automatic discovery of IP network services". In Journal of IEEE Internet Computing, July 1999.

[IBM98] IBM Corp. XML TreeDiff; November 1998
http://www.alphaworks.ibm.com/tech/xmltreediff

[INFOWARE] http://www.ifi.uio.no/~infoware/

[JMH04] David B. JOHNSON, David A. MALTZ and Yih-Chun HU, "The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR)". Internet-Draft, IETF, MANET, Feb 2004 http://www.ietf.org/internet-drafts/draft-ietf-manet-dsr-10.txt

[Kel99] Peter J. KELEHER. "Decentralized Replicated-Object Protocols". In proc. of the 18th ACM Symposium on Principles of Distributed Computing, April 1999.

[Kue94] Geoffrey H. KUENNING, "The Design of the SEER Predictive Caching System". In Proc. of Mobile Computing Systems and Applications 1994, Santa Cruz, CA, December 1994.

[KZL+01] J. KONG, P. ZERFOS, H. LUO, S. LU and L. ZHANG, "Providing robust and ubiquitous security support for mobile ad-hoc networks". In proc. of ICNP'01, 2001, p. 251-260.

[Lam78] Leslie LAMPORT. "Time, clocks and the ordering of events in a distributed system". In Communications of the ACM, 21(7):558–565, 1978.

[Lam79] Leslie LAMPORT. "How to make a multiprocessor computer that correctly executes multiprocess programs". In IEEE Transactions on Computers, C-28(9):690–691, Sept. 1979.

[Lee00] Yui-Wah LEE, "Operation-based Update Propagation in a Mobile File System". PhD thesis report January 2000, Department of Computer Science and Engineering, The Chinese University of Hong Kong

[LIME] http://lime.sourceforge.net/

[LIMONE] http://www.cs.wustl.edu/mobilab/projects/limone/

[Mah02] Qusay H. MAHMOUD, "Learning Wireless Java", O'Reilly, 2002.

[MaZ03] Marco MAMEI and Franco ZAMBONELLI, "Self-Maintained Distributed Data Structure Over Mobile Ad-Hoc Network". Technical Report, University of Modena and Reggio Emilia, August 2003

[MaZ04] Marco MAMEI and Franco ZAMBONELLI. "Programming Pervasive and Mobile Computing Applications with the TOTA Middleware". In proc. of 2nd IEEE International Conference on Pervasive Computing and Communication (Percom2004), Orlando (FL), 2004

[MCE01] Cecilia MASCOLO, Licia CAPRA and Wolfgang EMMERICH, "An XML based Middleware for Peer-to-Peer Computing". In proc. of 1st IEEE International Conference of Peer-to-Peer Computing, Linkoping (S), Aug. 2001.

[MCZ+02] Cecilia MASCOLO, Licia CAPRA, Stefanos ZACHARIADIS and Wolfgang EMMERICH. "XMIDDLE: A Data-Sharing Middleware for Mobile Computing". In Personal and Wireless Communications Journal 21(1). Kluwer. April 2002.

[MM06] R. MELLIER and J.-F. MYOUPO. "Partitionnement distribué de réseaux mobiles ad hoc et applications" (chap. 8) in " Réseaux mobiles ad hoc et réseaux de capteurs sans fil". Paris : Hermès, 2006

[Mos93] David MOSBERGER, "Memory consistency models". In Operating Systems Review, 17(1):18–26, January 1993.

[MOTION] http://www.motion.softeco.it/pages/

[MPR03] Amy L. MURPHY, Gian Pietro PICCO and Gruia-Catalin ROMAN. "Lime: A Coordination Middleware Supporting Mobility of Hosts and Agents". Technical report, Politecnico di Milano, Italy, 2003

[MPR99] Amy L. MURPHY, Gian Pietro PICCO and Gruia-Catalin ROMAN. "Lime: Linda Meets Mobility". In proc. of the 21 st International Conference on Software Engineering, May 1999

[MSC+86] J.H. MORRIS, M. SATYANARAYANAN, M.H. CONNER, J.H. HOWARD, D.S.H. ROSENTHAL and F.D. SMITH. "Andrew: a distributed personal computing environment". In Comm. ACM, 29(3):184--201, Mar. 1986.

[MUCODE] http://mucode.sourceforge.net/

[Mus02] Mirco MUSOLESI, "A middleware for data-sharing in ad-hoc networks". PhD thesis report, December 2002, Facolta' di Ingegneria, University of Bologna (Italy)

[OASIS] http://www.oasis-open.org/home/index.php

[ODF] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=office

[OOXML] http://www.ecma-international.org/news/TC45_current_work/TC45-2006-50.htm

[OPENSLP] http://openslp.org

[PAD+04] T. PLAGEMANN, J. ANDERSSON, O. DRUGAN, V. GOEBEL, C. GRIWODZ, P. HALVORSEN, E. MUNTHE-KAAS, M. PUZAR, N. SANDERSON and K.S. SKJELSVIK, "Middleware Services for Information Sharing in Mobile Ad-hoc Networks - Challenges and Approach". In proc. of IFIP Workshop on Challenges of Mobility (WCC 2004), August 2004

[PAP+05] Matija PUZAR, Jon ANDERSSON, Thomas PLAGEMANN and Yves ROUDIER, "SKiMPy: A Simple Key Management Protocol for MANETs in Emergency and Rescue Operations". Technical Report #319, University of Oslo, Department of Informatics, Feburary, 2005

[PEERWARE] http://peerware.sourceforge.net/

[PeR99] C.E. PERKINS and E.M. ROYER, "Ad hoc on-demand distance vector routing". In Proc. of IEEE 2nd Workshop on Mobile Computing Systems and Applications, 1999

[Rat95] David Howard RATNER, "Selective replication: Fine-grain control of replicated files". Master's thesis, University of California, Los Angeles, 1995

[Rat98] David Howard RATNER, "Roam: A Scalable Replication System for Mobile and Distributed Computing". PhD thesis, University of California, Los Angeles, Los Angeles, CA, January 1998. Also available as UCLA CSD Technical Report UCLA-CSD-970044.

[RHH01] Gruia-Catalin ROMAN, Qingfeng HUANG, and Ali HAZEMI. "Consistent group membership in ad hoc networks". In proc. of the 23rd Int. Conf. on Software Engineering, pages 381–388, Toronto, Canada, May 2001.

[RHR+94] Peter L. REIHER, John S. HEIDEMANN, David H. RATNER, Greg SKINNER and Gerald J. POPEK, "Resolving File Conflicts in the Ficus File System". In Proc. 1994 Summer USENIX Conf., pp. 183-195 (1994)

[ROAM] http://www.lasr.cs.ucla.edu/roam98/welcome.html#CurrentStatus

[Row98] A. ROWSTRON. "WCL: A coordination language for geographically distributed agents". In World Wide Web Journal, 1(3):167–179, 1998.

[RPP+93] Peter H. REIHER, Thomas W. PAGE, Gerald J. POPEK, Jeff COOK and Stephen CROCKER, "Truffles - A Secure Service For Widespread File Sharing". In proc. of the Privacy and Security Research Group Workshop on Network and Distributed System Security, Feb, 1993

[RRP04] David Howard RATNER, Peter H. REIHER, and Gerald J. POPEK, "ROAM: A Scalable Replication System for Mobility". In ACM/Kluwer Mobile Applications and Networks (MONET), Vol.9, No. 5, October 2004, pp. 537-544

[RUMOR] http://www.lasr.cs.ucla.edu/rumor/

[Sch01] Rüdiger Schollmeier, "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications". In Proc. First International Conference on Peer-to-Peer Computing, p. 101-102, Aug. 2001

[SEER] http://lasr.cs.ucla.edu/geoff/seer.html

[SGK+85] Russel SANDBERG, David GOLDBERG, Steve KLEIMAN, Dan WALSH, and Bob LYON, "Design and implementation of the Sun Network File System". In proc. of USENE' Conference, p. 119-130. USENIX, June 1985.

[SGM05] Norun SANDERSON, Vera GOEBEL and Ellen MUNTHE-KAAS, "Metadata Management for Ad-Hoc InfoWare - A Rescue and Emergency Use Case for Mobile Ad-Hoc Scenarios". In proc. of International Conference on Ontologies, Databases and Applications of Semantics (ODBASE05), , pp.1365-1380, 2005.

[SGP04] Katrine Stemland SKJELSVIK, Vera GOEBEL, Thomas PLAGEMANN, "Distributed Event Notification for Mobile Ad Hoc Networks". In IEEE Distributed Systems Online, vol. 05, no. 8, Aug., 2004.

[SKK+90] Mahadev SATYANARAYANAN, James J. KISTLER, Puneet KUMAR, Maria E. OKASAKI, Ellen H. SIEGEL, David C. STEERE, "Coda: A Highly Available File System for a Distributed Workstation Environment". In IEEE Transactions on Computers, April 1990, Vol. 39, No. 4

[TaS01] Andrew TANENBAUM and Maarten VAN STEEN. "Distributed Systems: Principles and Paradigms" Chapter 10. Published by Prentice Hall. Sep 26, 2001. http://www.cs.vu.nl/~ast/books/ds1/10.pdf

[TOTA] http://polaris.ing.unimo.it/didattica/curriculum/marco/MAIN/research/tota/tota.html

[XMIDDLE] http://xmiddle.sourceforge.net/

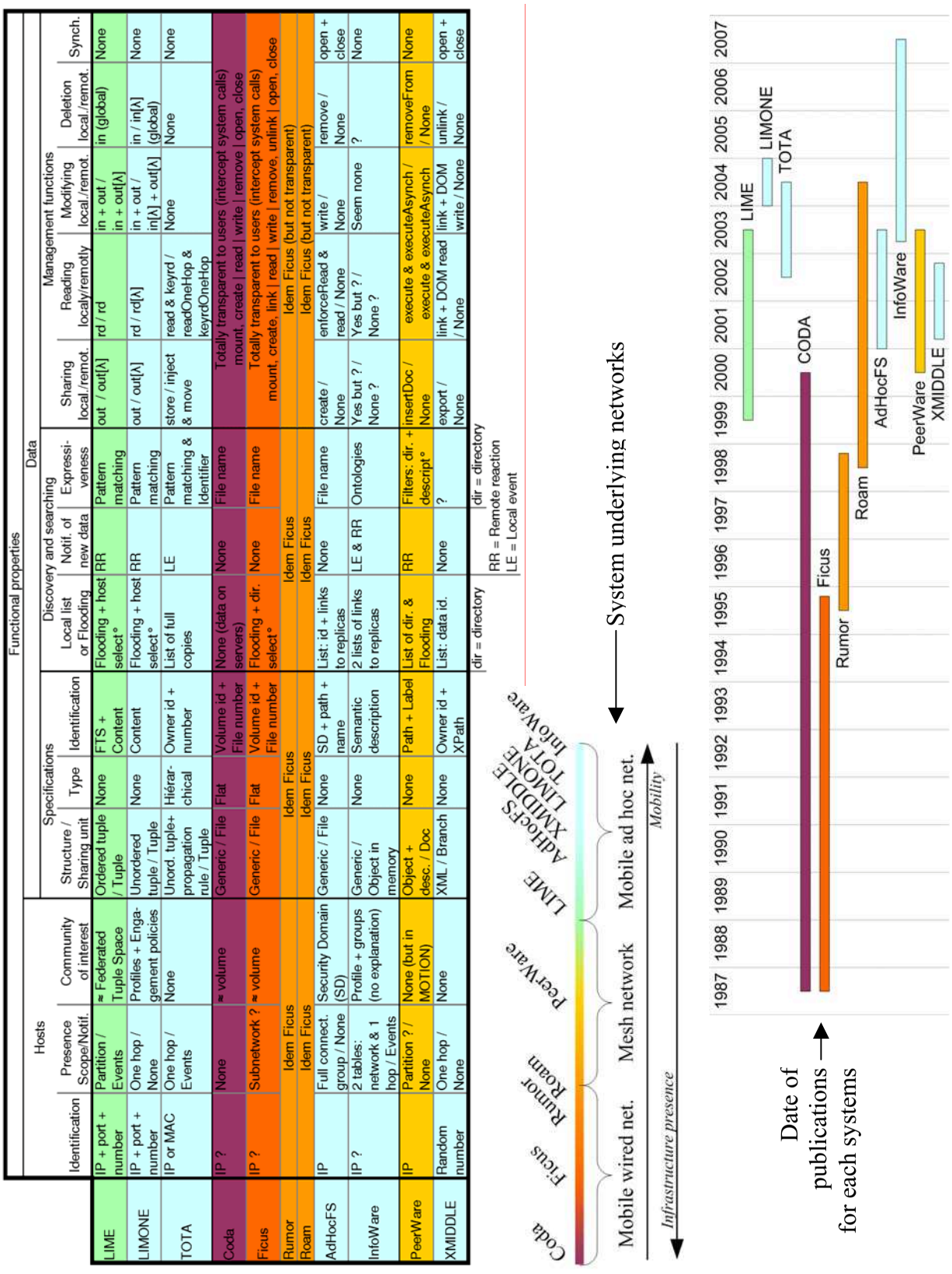
[XML] http://www.w3.org/XML/

[XPath] http://www.w3.org/TR/xpath

ANNEX: SYNTHETIC COMPARISON

	Functional properties														
	Hosts			Data											
				Specifications			Discovery and searching				Management functions				
	Identification	Presence Scope/Notif.	Community of interest	Structure / Sharing unit	Type	Identification	Local list or Flooding	Notif. of new data	Expressi-veness	Sharing local./remot.	Reading localy/remotly	Modifying local./remot.	Deletion local./remot.	Synch.
LIME	IP + port + number	Partition / Events	≈ Federated Tuple Space	Ordered tuple / Tuple	None	FTS + Content	Flooding + host select°	RR	Pattern matching	out / out[λ]	rd / rd	in + out / in + out[λ]	in (global)	None
LIMONE	IP + port + number	One hop / None	Profiles + Enga-gement policies	Unordered tuple / Tuple	None	Content	Flooding + host select°	RR	Pattern matching	out / out[λ]	rd / rd[λ]	in + out / in[λ] + out[λ]	in / in[λ] (global)	None
TOTA	IP or MAC	One hop / Events	None	Unord. tuple+ propagation rule / Tuple	Hiérar-chical	Owner id + number	List of full copies	LE	Pattern matching & Identifier	store / inject & move	read & keyrd / readOneHop & keyrdOneHop	None	None	None
Coda	IP ?	None	≈ volume	Generic / File	Flat	Volume id + File number	None (data on servers)	None	File name	Totally transparent to users (intercept system calls) mount, create | read | write | remove | open, close				
Ficus	IP ?	Subnetwork ?	≈ volume	Generic / File	Flat	Volume id + File number	Flooding + dir. select°	None	File name	Totally transparent to users (intercept system calls) mount, create, link | read | write | remove, unlink | open, close				
Rumor	Idem Ficus			Idem Ficus			Idem Ficus			Idem Ficus (but not transparent)				
Roam	Idem Ficus			Idem Ficus			Idem Ficus			Idem Ficus (but not transparent)				
AdHocFS	IP	Full connect. group / None	Security Domain (SD)	Generic / File	None	SD + path + name	List: id + links to replicas	None	File name	create / None	enforceRead & read / None	write / None	remove / None	open + close
InfoWare	IP ?	2 tables: network & 1 hop / Events	Profile + groups (no explanation)	Generic / Object in memory	None	Semantic description	2 lists of links to replicas	LE & RR	Ontologies	Yes but ? / None ?	Yes but ? / None ?	Seem none	?	None
PeerWare	IP	Partition ? / None	None (but in MOTION)	Object + desc. / Doc	None	Path + Label	List of dir. & Flooding	RR	Filters: dir. + descript°	insertDoc / None	execute & executeAsynch	execute & executeAsynch	removeFrom / None	None
XMIDDLE	Random number	One hop / None	None	XML / Branch	None	Owner id + XPath	List: data id.	None	?	export / None	link + DOM read / None	link + DOM write / None	unlink / None	open + close

dir = directory
RR = Remote reaction
LE = Local event



	Non-functional properties													
	Data management				Security						Ressource management		
	Consistency	Replication		Accessibility	Authenticat°	Access control	Confidentiality	Integrity	Non-repudiat°	Misc.	Bandwidth / Energy	Memory	Comput. Power
			Distr. persist.										
LIME	>SC	No	None	None	None	None	None	None	None	PS	Costly: Presence	None	None
LIMONE	>SC	No	None	None	None	Flexible ctr.	Possible with	Remote Op.	None	None	No optimisation but can use Profiles		
TOTA	Depends on propagation rules	Yes	Depends on propagation rules	Depends on propagation rules	None	None	None	None	None	None	Costly: Replication & Notification	Very costly (locally & globally)	None
Coda	{R-Y-W + VV + log + ASR} & Notification	Yes	Servers	Hoarding & Access replication	Yes	Ctr. over dir. (permition & exclusion)	None	None	None	ASR secu-red & PS	Lightly costly: Notification	Log com-pression	None
Ficus	{<R-Y-W + VV + ring topo resync.} & Notif.	Yes	Prevents del operations	Replicat° rules & Access & Hoard. (Seer)	Delegated to Truffles (upper system)				None	PS	Costly: Notification	Garbage collection very slow	None
Rumor	<R-Y-W + VV + ring topo reconciliation	Yes	Idem Ficus		Idem Ficus						None	Idem Ficus	
Roam	<R-Y-W + vesion vect + 2-level ring topo reconciliat°	Yes	Idem Ficus with 2-geographical-level mechanisms		Idem Ficus						Long distance communicat° minimized	Version vector compress°	None
AdHocFS	{Global: R-Y-W + log + server?} & {Group: >SC}	Yes	Global: server & Grp: prevents loss	Global: access replicat° & Grp: prevents loss	Yes: Certificat	Unrestricted in SD & No access out	GKA in group + encryption	None ?	None	PS?	All protocols take into account host memory and energy and minimize com		Symetric encryption
InfoWare	None ? (No updates ?)	Yes	None	Prevents futur disconnect° ?	Yes: Certificat	Yes	GKA= SkimPy + encryption	None	None	Event secu-red & PS?	Lazy version protocols	Costly?: ontologies	Symetric encryption
PeerWare	None	No ?	None	None	Integrated in the MOTION platform			None	None	None	Data research may be costly	None	None
XMIDDLE	{Global: R-Y-W + log + semantic} & {Near owner: SC}	Yes	Server (doc owner)	Access replication	None	None	None	None	None	PS	Little sharing unit may save bandwidth	Costly: XML redundant version logs	DOM engine

SC = Sequential consistency
R-Y-W = Read-Your-Writes consistency
VV = Vector vector
ASR = Application specific resolver
SAR = Semantically assisted resolver

ctr. = control
dir. = directory

PS = Private space