Università degli Studi di Pisa

Dipartimento di Ingegneria Informatica
Dottorato di Ricerca in Ingegneria dell'Informazione

Ph.D. Thesis

Cross-layer Peer-To-Peer Computing in Mobile Ad Hoc Networks

Giovanni Turi

Supervisor
Prof. Giuseppe Anastasi

Dott. Marco Conti

February 2006

Abstract

The future information society is expected to rely heavily on wireless technology. Mobile access to the Internet is steadily gaining ground, and could easily end up exceeding the number of connections from the fixed infrastructure. Picking just one example, ad hoc networking is a new paradigm of wireless communication for mobile devices. Initially, ad hoc networking targeted at military applications as well as stretching the access to the Internet beyond one wireless hop. As a matter of fact, it is now expected to be employed in a variety of civilian applications. For this reason, the issue of how to make these systems working efficiently keeps the ad hoc research community active on topics ranging from wireless technologies to networking and application systems.

In contrast to traditional wire-line and wireless networks, ad hoc networks are expected to operate in an environment in which some or all the nodes are mobile, and might suddenly disappear from, or show up in, the network. The lack of any centralized point, leads to the necessity of distributing application services and responsibilities to all available nodes in the network, making the task of developing and deploying application a hard task, and highlighting the necessity of suitable middleware platforms.

This thesis studies the properties and performance of peer-to-peer overlay management algorithms, employing them as communication layers in data sharing oriented middleware platforms. The work primarily develops from the observation that efficient overlays have to be aware of the physical network topology, in order to reduce (or avoid) negative impacts of application layer traffic on the network functioning. We argue that a *cross-layer* cooperation between overlay management algorithms and the underlying layer-3 status and protocols, represents a viable alternative to engineer effective decentralized communication layers, or eventually re-engineer existing ones to foster the interconnection of ad hoc networks with Internet infrastructures. The presented approach is twofold. Firstly, we present an innovative network stack component that supports, at an OS level, the realization of cross-layer protocol interactions. Secondly, we exploit cross-layering to optimize overlay management algorithms in unstructured, structured, and publish/subscribe platforms.

To Gaia

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile ad hoc networks (MANETs) represent complex distributed systems made of wireless mobile nodes that can freely and dynamically self-organize into arbitrary and temporary, "ad-hoc" network topologies. This spontaneous form of networking allows people and devices to seamlessly exchange information in areas with no pre-existing infrastructure, like for example disaster recovery environments. While early MANET applications and deployments have been military-oriented, civilian applications are also gaining interest in this emerging form of mobile computing. Especially in the past few years, with the rapid advances in mobile ad hoc networking research, MANETs have attracted considerable attention and interests from commercial business industry, as well as standardization communities. The introduction of new technologies, such as the Bluetooth and IEEE 802.11, greatly facilitates the deployment of the ad hoc technology outside of military domains, generating a renewed and growing interest in the research and development of MANETs for several applicative scenarios like, for example, in personal area and home networking, law enforcement and search-and-rescue operations, commercial and educational applications, and sensor networks.

Currently developed mobile ad hoc systems adopt the approach of not having a middleware, relying on application developers to write code able to handle all the services it needs. This constitutes a major complexity/inefficiency in the development of MANET applications. Indeed, most of the research in this area concentrated on enabling technologies (i.e. MAC), and on networking protocols (mainly routing), while research on middleware platforms for MANETs is still in its infancy [CCL03].

Recently, interesting proposals for MANET middleware appeared in [BCM05] [ZCME02] [MPR01]. Lime [MPR01] uses tuple space, reactive programming and mobile agent abstractions, to support coordination between software agents running on nodes in communication range. XMIDDLE [ZCME02] provides abstractions for data replication and reconciliation, organizing data as XML document trees, and helping applications to carry out disconnected operations. REDMAN [BCM05] deals with the problem of guaranteeing data and service replication in dense MANET environments, providing algorithms and protocols for the identification and election

of replica managers. While these platforms offer different kinds of abstractions and services, they all rely on a fully decentralized application's design and messaging scheme, inspired to the peer-to-peer computational model.

Ad hoc networking shares many concepts, such as distribution and cooperation, with the peer-to-peer (P2P) computing model [SGF02]. P2P systems are usually characterized by the absence of a centralized authority that drives the system's components, and base their functioning on the self-organization of the entities that take part to the system by playing a symmetrical role. For these reasons, the applications best suited for P2P implementation are those where centralization is not possible, relations are transient, and resources are highly distributed [PC02]. Another key aspect of P2P systems is the ability to provide inexpensive, but at the same time scalable, fault tolerant and robust computing infrastructures. For example, file sharing services like Gnutella [KM02], are distributed systems where the contribution of many participants with small amounts of disk space results in a very large distributed database. A further example is the SETI@home project [SET], where users volunteer their CPU resources, making up a large-scale signal processing infrastructure to support the research of extraterrestrial life.

All the above aspects of P2P computing fit well with the distributed and dynamic nature of mobile ad hoc networks, where decentralization, intermittent connectivity, and resource distribution all begin at the network level, where nodes have to self-organize in a transient communication infrastructure, in order to support user applications. This duality suggests to investigate the P2P paradigm as a promising direction towards efficient middleware platforms for MANET environments.

In this Chapter, we recall the characteristics and paradigms of P2P computing that are of particular relevance in the context of mobile ad hoc networks.

1.1 P2P Computing: history, characteristics and paradigms

P2P systems appeared on the Internet to support applications that harness the resources of a large number of autonomous participants (called peers). In many cases, these peers form self-organizing networks that are layered on top of conventional Internet protocols and have no centralized structure. Inspired by the successes of early P2P systems such as Napster, Gnutella, and SETI@home, a large and active research community continues to explore the principles, technologies, and applications of such systems.

P2P and classical distributed computing are both concerned with enabling resource sharing within distributed communities. However, different base assumptions have led to distinct requirements and technical directions [FI03]. P2P systems have focused on resource sharing in environments characterized by potentially millions of users, most with homogeneous desktop systems and low-bandwidth, intermittent

connections to the Internet. As such, the emphasis has been on global fault-tolerance and massive scalability. In contrast, classical distributed systems have arisen from collaboration between generally smaller, better-connected groups of users with different resources to share.

Despite these differences, the long-term evolution of classical distributed computing and P2P seems likely to converge at least in some regards, as distributed systems expand in scale and incorporate more transient services and resources, and as P2P researchers consider a broader class of applications [FI03].

1.1.1 A brief history

P2P networking has divided research circles. The traditional distributed computing community views these young technologies as "upstarts with little regard for, or memory of, the past"; evidence supports this view in some cases. Others welcome an opportunity to revisit past results, and to gain practical experience with large-scale distributed algorithms. An early use of the term "peer-to-peer computing" is in IBM's Systems Network Architecture documents over 25 years ago, but publicly came to fore with the rise and fall of Napster [NAP] file sharing application in 1999.

Beyond the Client-Server Model

P2P systems can be contrasted with asymmetric client/server systems, in which a server – usually a more powerful and better connected machine – runs for long periods of time and delivers storage and computational resources to some number of clients. Thus the server emerges as a performance and reliability bottleneck, which could be mitigated with techniques such as replication, load balancing, or request routing, and significant investments in high-end machines, high-bandwidth connectivity, rack space and so forth. All of that, suggests that the support of a centralized solution is a viable option provided there is an economic incentive or a business model that justifies capital and administrative expenses.

A natural evolution of this thinking is to include the clients' resources in the system, an approach that becomes increasingly attractive as the performance gap between desktop and server machines narrows, and broadband networks dramatically improve client connectivity. Thus, P2P systems evolve from client/server systems by removing the asymmetry in roles: clients are also servers that allows access to their resources, actively participating to the service' supply. Work (be it computation, or file sharing) is partitioned between all peers, so that a peer consumes its own resources on behalf of others (acting as a server), while asking other peers to do the same for its own benefit (acting as a client). As in the real world, this cooperative model may break down if peers are not provided with incentives to participate, which in successful stories like Napster or Gnutella [KM02] turned out to be just the nature of the content being shared.

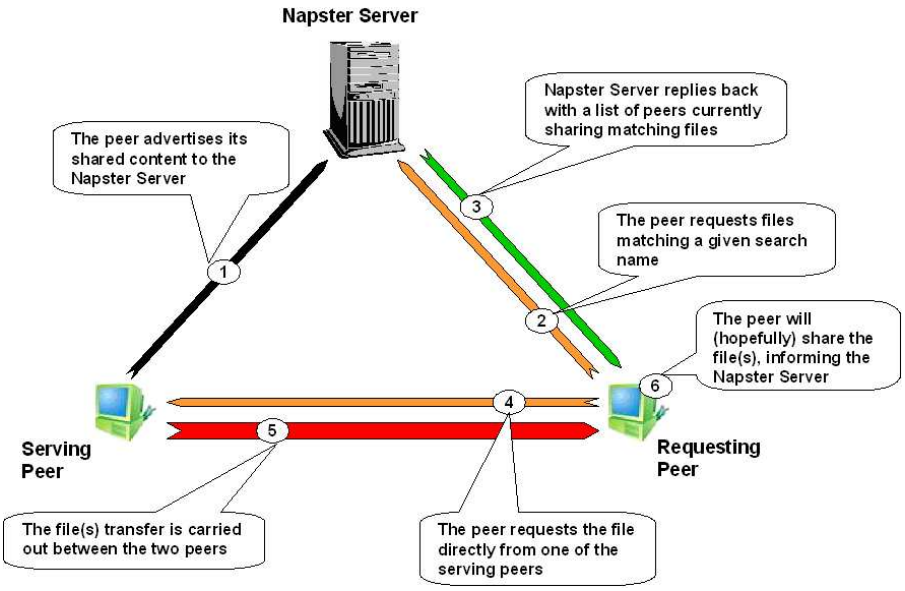


Figure 1.1: Napster, an example of centralized P2P architecture.

Yet another viewpoint from which one can dissect these systems is the use of intermediaries. The Web (and client/server file systems such as NFS and AFS) uses caches to reduce average latency and networking load, but these caches are typically arranged statically. P2P systems partition work dynamically among cooperative peers to achieve locality oriented load balancing. Content distribution systems such as PAST [RD01b] and Pasta [MPH02] use demand-driven strategies to distribute data to peers close in the network to that demand.

The classical distributed systems community would claim that many of these ideas were present in early work on fault tolerant systems in the 1970s. For example the Xerox Network System's name service, Grapevine [MSN84], included many traits mentioned here. Other systems that can be construed as P2P systems include Net News (NNTP is certainly not client-server) and the Web's Inter-cache protocol, ICP. The Domain Name System also includes zone transfers and other mechanisms that are not part of its normal client/server resolver behavior.

Napster

The Napster [NAP] file-sharing system started in 1999 allowing users to "share" audio files (MP3s) stored on their own hard drives. In Napster (see Figure 1.1), peers stored locally their collection of MP3s, while Napster ran a central server storing only the index of files available within the peer community. To retrieve a desired song, users issued keyword-based search requests to this central server and obtained the IP address of peers storing matching files. The user could then download the desired files directly from one of these peers.

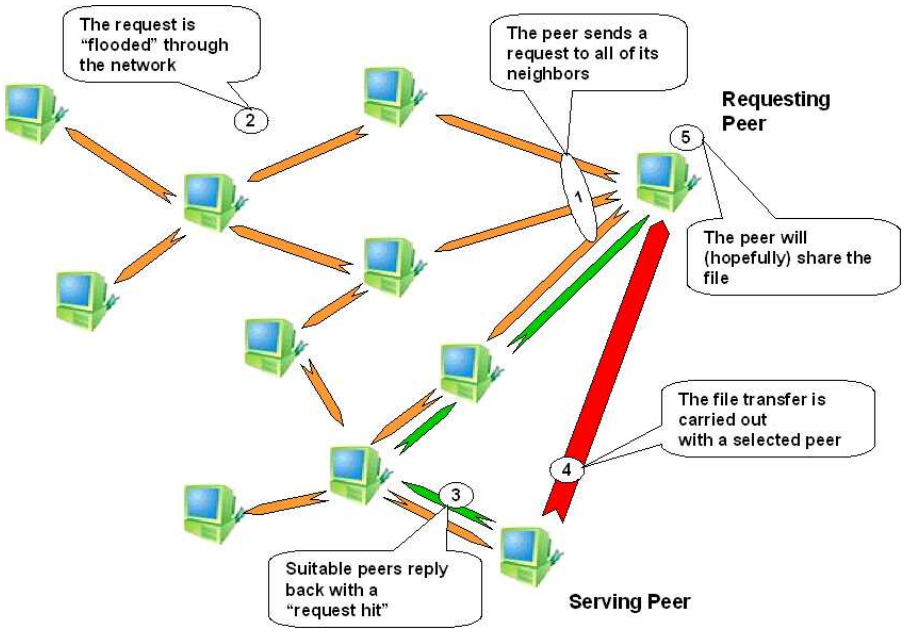


Figure 1.2: Gnutella, an example of fully decentralized P2P architecture.

Clearly, Napster did not come with a "pure" P2P architecture, as only content storage and exchange were distributed among the peers, while file indexing and lookup was on a central location administered by Napster (the company). However, because it dramatically simplified the task of obtaining music on the Internet, Napster became popular, reaching nearly 50 Million user within the first year of service. Over time, Napster's centralized directory became both a severe bottleneck and a single point of failure for legal, economic, and political attacks; and Napster was eventually shut down by court order for helping users infringe copyright.

Napster's success was attributable to online music sharing being a "killer application". Moreover, it demonstrated the potential in harnessing client resources to satisfy their need for a service. With the demise of Napster, there arose a desire within the music-sharing community for a fully decentralized service that would not be susceptible to a similar legal attack. The projects that rose to the challenge stimulated important technical developments in distributed object location and routing, distributed searching, and content dissemination.

The second generation: full decentralization

Gnutella [KM02] is a distributed search protocol adopted by several file-sharing applications, which dispensed with the centralized directory distributing also file indexing and lookup. Gnutella peers locate content sources flooding their neighborhood with search queries messages (see Figure 1.2). Despite measures to limit and restrict flooding, several studies and user experience found that sometimes the vol-

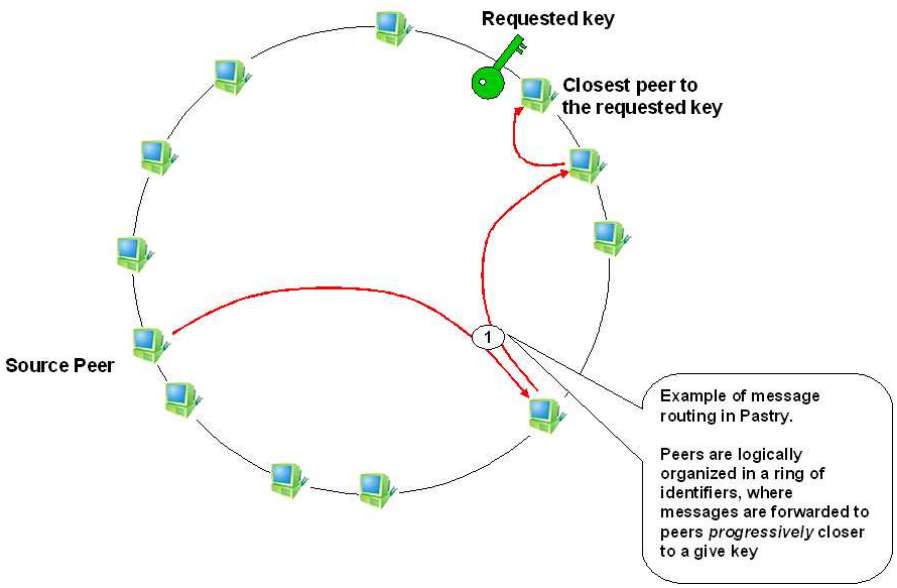


Figure 1.3: Pastry, an efficient message routing substrate.

ume of query and control traffic caused excessive network load,decreasing the chance of satisfying a given query, as well as the amount of bandwidth left for actual file transfers.

Other systems for content location, including Freenet [CSWH01], added mechanisms to route requests to the node where the content is likely to be, in a best-effort partitioning of the networks' content. Systems for file sharing such as Kazaa [LRW03], as well as recent Gnutella evolutions, added structure to P2P file-sharing networks by dynamically electing nodes to become super-peers, caching and serving common queries or content. These schemes take advantage of the observed Zipf-like distribution of object popularity and mitigate the difficulties of passing queries through hosts on high latency, low bandwidth, dial-up connections.

The third generation: efficient routing substrates

Although the range of applications for P2P techniques remained limited by the end of 2001, a common requirement had emerged. In order for each peer to make a useful contribution to the global service, a reliable way of partitioning workload and addressing the responsible nodes was needed. Further, the emphasis on scalability, and the corresponding observation that in global-scale system peers will be joining, failing, and leaving continually, required these functions to be performed with knowledge of only a fraction of the global state on each peer, maintained with only a low communication overhead in the underlying network.

These observations inspired a generation of P2P routing substrates that provided a distributed message passing, object or key location service. The most popular

approaches adopt a virtual address space, in which nodes are assigned a unique pseudo-random identifier that determines their position in the space (see Figure 1.3). Messages are then routed toward keys in the same address space, and are delivered eventually to the closest node. According to the way in which applications use this service, a message destined for a given key represents a request to provide a given service with respect to that key. As requests' keys must be mapped on to the key space pseudo-randomly, these platforms offer effective partitioning of the work between peers. Different variants of this basic approach differ as to the structure of information on nodes and the way messages (or sometimes requests for routing information) are passed between peers.

1.1.2 Properties and Issues

We now survey various properties of P2P computing, exploring aspects that are present in current systems, as well as some that are still subject of ongoing work. In each case, we discuss their relevance in the context of mobile and distributed computing.

Harnessing Resources

The P2P fault model of an unreliable infrastructure and mutually distrustful participants leads P2P systems to treat resources as homogeneous and peers as individually dispensable. Therein lie many of the strengths and weaknesses of the approach. Key-based routing platforms embody these assumptions in their design. Any node is equally likely to be responsible for one particular key, so it is assumed to be equally suitable to carry out a task related to it. Nodes carry similar numbers of keys, so it is assumed that their resources for storing or managing these keys are also equal. Peers, however, are unlikely to have similar resources either in quantity or in quality. Nor are peer resources likely to have similar reliability characteristics. This last observation is especially true in the case of mobile settings such as ad hoc networks, where the user population itself is opportunistic, and comes equipped with handheld devices with limited resource capabilities. Systems that recognize these facts may cause benefits in terms of performance and availability. For example, superpeering [KM02][CRB+03] in file-sharing systems takes advantage of well-connected nodes to implement distributed caching and indexing schemes.

User Connectivity

The nature of a peer's network connection is an essential consideration when designing P2P systems for practical use in target user communities. In many contexts, mean connection quality is low but presents also high variance, due to differences between dial-up, broadband, and connections from academic or corporate networks.

Thus, the scope for generic inter-node communication is severely limited, and applications must consider the heterogeneity of their peers' connections. In the case of MANETs, P2P systems should assume peers connections to be extremely transient and intermittent, due to both the mobility of the participants, and the nature of the wireless transmission media, which is prone to unfair access contention, as well as to environmental interferences.

Scalability

P2P systems often add resources as they add customers. Thus, they should scale (at least at the computing and storage resource level, if not networking) linearly, or better, with the number of peers. Scalability is not a trivial consideration. While significant work has been done to achieve scalability in terms of latency and performance of inter-node communication in Internet-like scenarios [RFH+01][CRB+03][RD01a] [CCR03], MANETs would initially employ P2P systems in small-scale setups (e.g., in the order of hundreds of nodes), and could then require different scalability properties, like for example low overhead generation, or tolerance to frequent node churns.

Proximity

Latency is an important consideration when routing in P2P systems. Poor proximity approximation can result in a message bouncing around the network many times before reaching its final destination, with disastrous effects in constrained environments such as MANETs. Several distributed applications aim to automate the gathering of relevant proximity information, for example by estimating point-to-point communication latency [ZPS00]. Issues of "stretch" (i.e., distance traveled relative to the underlying network) become increasingly important in data sharing systems, in which large quantities of data must be transferred between peers. Furthermore, to minimize the load on the network and increase the rate at which data may be obtained, systems should attempt to store or replicate the information located near the place it is accessed.

Availability

P2P networks experience a high rate of nodes joining and leaving, both because of their large dimension and because of the nature of the user communities. Hence, the system cannot rely on individual peers to maintain any essential state on their own. For purposes of redundancy, most systems based on efficient routing substrates attempt to replicate the state at the k nodes with identifiers numerically closest to the associated key. This replication is maintained by local node cooperation, despite nodes joining or failing, and offers automatic fail-over provided that routing tables are correctly maintained. However, as mentioned in [CRB+03], scenarios with frequent node churns, like mobile environments, may increase the overhead associated to replication, and vanish its benefits.

Little attention has been paid to the effect of network partitions on systems in which partially or wholly independent fragments are formed, update their own state, and then later rejoin. Quorum systems (e.g., [CL99]) have been used to enforce state consistency between peers updating replicated data [KBC+00], but the overhead of these schemes is prohibitive. In mobile ad hoc networks, partitions are frequent events that are caused by the mobility of the participants. Hence P2P systems designed to operate in such settings are expected to handle partitions as normal events, nicely tolerating frequent split and rejoin of system fragments.

Decoupling

One important capability of P2P systems, especially in data sharing applications, is their ability to *decouple* peers' interaction. As discussed in [EFGK03] in the case of publish/subscribe systems, decoupling can be decomposed along the following three different dimensions:

Identity Peers do not need to know each other a priori in order to interact, as it is the system to route service messages from "consumers" toward "producers". Once a consumer discovers one or more locations of interest for the requested service, it could establish direct connections in order to use it. In any case, neither producers nor consumers hold each other references.

Time Peers do not need to be actively participating in the interaction at the same time, as long as the service is *replicated* and *distributed* somewhere in the system. Picking up data sharing applications as an example, a producer peer can put some data in system while an interested consumer is disconnected, and conversely, the consumer might issue queries and retrieve the data when the original producer is disconnected.

Synchronization The production and consumption of data does not happen in the main control flow of producers and consumers, and do not therefore happen in a synchronous manner. This form of decoupling allows a consumer to issue a query passing a reference to a call-back function. In this way, it could be asynchronously notified (using an event-based style of interaction) when somebody in the system produces data that matches the query.

Decoupling the production and consumption of information increases scalability by removing all explicit dependencies between the interacting participants. In fact, removing these dependencies strongly reduces coordination and thus synchronization between the different entities, and makes the resulting communication infrastructure well adapted to distributed environments that are asynchronous by nature, such as mobile ad hoc environments [HGM01].

1.1.3 Paradigms

This thesis mainly investigates efficient techniques of information exchange among nodes of a MANET. As discussed before, the "duality" between ad hoc networks and P2P systems suggests to organize data sharing in a P2P fashion. In this section, we revisit the major paradigms of P2P information sharing, which will be considered in the rest of the thesis.

A key challenge to the usability of a P2P data sharing system is implementing efficient techniques for search and retrieval of shared data. The best search techniques for a system depend on the needs of the distributed application. Some applications, like for example web caches or archival systems, focus on *performance* and *availability*. These requirements usually come at the expense of flexibility, as they are met by using indexes that speed up search procedures. In contrast, other kinds of application focus on *flexibility*, requiring the ability to issue rich queries (e.g., regular expressions or logic predicates), and at the same time respect the autonomy of individual peers, for example without imposing the establishment of a search index. These requirements relax performance and availability assumptions, tolerating situations in which the data is not retrieved even if actually present in the system.

Structured data sharing

Structured platforms are such that peers organize themselves in a distributed search index, which has the function of mapping each piece of data to its exact location, in a way often similar to a distributed hash table (DHT). In these systems, each peer maintains only a partial knowledge of the index, but establishes link relationships with other peers to ensure a complete coverage of the search structure (i.e., a structured overlay). The main idea is to virtually place peers and data on a single logical space of identifiers, assuming that each peer gets "responsible" about data that is close in the logical space. This approach is well suited for a *subject-based* system' organization, allowing for lookup procedures that take data identifiers as input parameters. Various are the proposals in the area of structured P2P computing. For example, Pastry [RD01a] and Chord [SMLN$^+$03] organize the overlay as a ring of identifiers, while CAN [RFH$^+$01] uses quad-tree index organization on an n-dimensional space. All these platforms achieve optimal lookup performances with logarithmic bounds, and require each node to establish a little number of relationships in the overlay.

Unstructured data sharing

Unstructured platforms are such that peers do not organize a distributed search index, and are not required to maintain relevant information about shared content owned by other entities. Peers establish link relationships in a pseudo-random fashion (i.e., an unstructured overlay), starting from a given entry point, and base

their search procedures on message flooding among peers. This approach does not match availability requirements like in the case of structured platforms, but allows for *content-based* lookup procedures using regular expressions or logical predicates. Content-based lookups are directly applied on the published content, and assume that a large number of peers get hit by search requests, thanks to query propagation schemes based on flooding. Platforms like Gnutella [KM02] or KaZaa [Ltd01], witness the flexibility offered by this approach, and the success they had in supporting large-scale file sharing applications on the Internet.

Event Publish/Subscribe

Alternatively to the above paradigms, modern distributed applications focus the interaction among their components on a event-based communication model. The *publish/subscribe* paradigm provides entities with the ability to express their interest (i.e., to subscribe) in an event or a pattern of events. When other entities publish information that match the subscribers interest, the system delivers it to the correct subscribers in a transparent fashion. In other words, a publish/subscribe platform implements a "software bus" [EFGK03], where producers publish information and consumers subscribe to what they want to receive. The information is typically denoted by the term event, and the act of delivering it by the term notification. These platforms provide full decoupling in time, space, and synchronization between the interacting entities, and their reactive style of communication makes them suitable for highly dynamic scenarios like MANETs and peer-to-peer systems.

In the context of P2P computing for mobile ad hoc networks, it is advisable to consider all the above approaches, as they could better support one or the other kind of application. Furthermore, an evaluation of the capacity and the performance of existing platforms in MANET environments would provide an important starting point for further discussion and new proposals.

1.2 Deploying Services by Overlaying

A common characteristic of the aforementioned P2P systems, is that they provide abstractions for service interactions between peers by means of *overlays*. In short, overlays are virtual networks of nodes and links built on top of existing infrastructures, with the purpose of deploying new services. An overlay may be as simple as a collection of static IP-in-IP tunnels, or as complex as a full dynamic VPN (virtual private network). For example, the Resilient Overlay Network (RON) system [ABKM01] is composed by sites that collaborate to find "IP level" paths that are longer than those provided by the infrastructure, but deliver better properties (such as throughput or loss) at the application level. This goal is achieved by routing traffic over a set of dynamically created tunnels among peer machines that run RON instances. Another example is the Internet itself. In principle the Internet is

an overlay network that aims at connecting local area networks using the Internet Protocol (IP).

Introducing new distributed services by overlaying has clear advantages. This approach allows service deployment without the installation of new hardware equipment, or modification of existing software and protocols. Hence, overlays enable fast service bootstrapping in target networking environments. Moreover, services based on overlays do not necessarily need to be deployed on every node to succeed in providing the functionality. This gives freedom to node owners who might be not interested in joining the service all the time, or might have hardware that is not appropriate to support additional workloads.

However, overlaying presents also drawbacks in terms of overhead and complexity. Overlays cause additional overhead as they are implemented as new software layers in networking stacks, and therefore introduce additional packet headers and redundant processing. Just to give an example, IP packet headers contain both IP and Ethernet addresses, but the latter are also part of Ethernet frames. Complexity is another concern, as the introduction of new software layers does not eliminate complexity, but simply attempts to manage it. However, more layers of functionalities also augment the possibility of unintended interaction between layers. As an example, let us consider the behavior of TCP in wireless networks. Frequently, packet dropping due to corruption is misinterpreted as a situation of congestion, and has negative effects on transfer rates [CGM05].

In this thesis, we deal with the problem of overlay networking in mobile ad hoc environments. More in details, we aim at investigating innovative methodologies for the implementation of effective overlay networks, which could then be used to support P2P systems and applications in ad hoc settings. We intend to study overlay management protocols under the following aspects:

Overhead and complexity. The devices which form a mobile ad hoc network have to do a conservative use of both computing and networking resources, as the processing power, the communication bandwidth, and the available energy are usually limited. Therefore, overlay management protocols that do not adhere to these constraints would fail in these settings.

Self-organization and fault-tolerance. The extreme dynamics of MANET settings require a higher level of self-organization and fault-tolerance with respect to standard Internet settings. Therefore overlays should be able to self-organize without the need of user information (e.g., a bootstrapping peer), and should handle node churns or network partitions as "normal" conditions, and not as worst-case situations.

Compatibility with existing protocols. MANETs are expected to complement Internet wireless coverage beyond the first hop, other than supporting users opportunistically grouped in an area without infrastructure. Therefore, it

is important to maintain compatibility with existing platforms, in order to guarantee continuity to the supply of P2P Internet services.

We argue that especially in mobile wireless settings, the above properties can be satisfied by mapping the overlay to the layer-3 network, making it *aware* about the current state of the network topology. This approach requires a "strict" cooperation with protocols at the network layer responsible for discovering and maintaining the network topology (i.e., routing and forwarding agents). Awareness about the underlying layer-3 network would deliver good performance in terms of *stretch* (ratio of delay to shortest path delay), and *stress* (number of duplicate transmissions over a physical link). However, a strict cooperation with underlying protocols and data structures would *couple* overlay agents with network layer agents, spoiling the modularity and flexibility of classical stack architectures. For these reasons, our approach is twofold:

1. We propose a flexible networking architecture that enables the design and easy deployment of *cross-layer* interactions, but at the same time maintains loose coupling between the interacting agents;

2. We show how the cross-layer architecture could be used to improve the quality and the performance of overlays supporting unstructured, structured, and publish/subscribe platforms.

1.3 Thesis outline

The central focus of this thesis is the design and evaluation of cross-layered P2P systems, to support efficient distributed computing in MANET environments. A fundamental component of the work is an innovative network stack architecture, which enables easy and portable exploitation of cross-layer interactions between protocols. In addition, we show how to integrate Lime [MPR01], a middleware for mobile and distributed computing based on tuple space programming, on top of structured platforms. Accordingly, the remainder of the thesis is organized as follows:

Chapter 2: An Architecture for Flexible Cross Layering
Cross layering has recently emerged as a new trend to cope with performance issues of mobile ad hoc networks. The concept behind this technique is to exploit local information produced by other protocols, so as to enable optimizations and deliver better network performance. This Chapter addresses cross layering from an architectural standpoint, providing a basis for tackling the semantic problems of interfering optimizations and correct system implementation. In particular, we claim that new interactions can be realized maintaining the layer separation principle, with the introduction of a cross-layer interface (XL-interface) that standardizes vertical interactions and gets

rid of tight-coupling from an architectural standpoint. The results related to the cross-layer architecture have been presented in [CMTG04] [CCMT05] [CMT06].

Chapter 3: A Cross-Layer Approach for Unstructured P2P Computing
Chapter 3 investigates the performance of Gnutella, one of the most widely used P2P systems, when put through typical ad hoc conditions like node mobility, frequent network partitioning, etc. We show that a straightforward implementation of the protocol is not satisfactory under the point of view of the produced overhead and the average overlay connectivity. Finally, we propose a cross-layer optimization of Gnutella, which enhances its performance up to the expectations, and makes it more suitable to the degree of self-organization and self-healing required in ad hoc environments. The evaluation of Gnutella and the cross-layer optimization have been presented in [CGT05].

Chapter 4: A Cross-Layer Approach for Publish/Subscribe
Chapter 4 presents Q a publish/subscribe service conceived to operate over MANETs. With Q, the overlay network that routes events from publishers to subscribers dynamically adapts to the changing topology by means of cross-layer interaction. Q also supports content-based filtering of events through mobile code: subscribers can specify in detail the notifications they wish to receive by denying proper filter classes, then binary code of filters is exchanged during runtime by participating nodes. The design of Q, and its preliminary evaluation has been presented in [AVT05].

Chapter 5: Structured P2P Computing
Chapter 5 deals with structured P2P computing and its usage in mobile ad hoc environments. The structured paradigm provides an effective framework to realize decentralized and scalable applications, and has been designed to support large sets of users with bounded costs and fairly balanced workload distribution. In this Chapter, we evaluate the performance of Pastry, highlighting low tolerance to dynamics such as nodes mobility, and topology reconfiguration. Finally, we suggest how the system could be redesigned and adapted to work nicely in emerging mobile computing scenarios. The content of this chapter originates from the work presented in [CGT04] and [CDT06].

Chapter 6: Laying Tuple Spaces over Structured P2P Platforms
Lime (Linda in a Mobile Environment) is a middleware platform for mobile computing, based on coordination concepts inspired by those used in Linda. This platform supports the development of distributed applications for mobile environments, providing the abstraction of transiently shared tuple spaces. While Lime exhibits maturity in the set of exported primitives and coordination algorithms, it uses a basic implementation of messaging between peers, which does not guarantee scalability. Chapter 6 proposes a re-design of Lime

communication concerns, based on structured P2P computing. The new strategy guarantees the semantic defined by Lime, with the prospect of delivering good performance in both MANET and Internet-like settings.

Chapter 7: Conclusions

Chapter 7 draws the conclusions of this thesis and proposes some future work.

Chapter 2

An Architecture for Flexible Cross Layering

Abstract

Cross layering has recently emerged as a new trend to cope with performance issues of mobile ad hoc networks. The concept behind this technique is to exploit local information produced by other protocols, so as to enable optimizations and deliver better network performance. However, the need for a new interaction paradigm inside the protocol stack has to face with the legacy aspects of classical architectures (e.g., the Internet), where layer separation allows for easy standardization and deployment.

In this Chapter, we show that cross layering can be achieved maintaining a clean architectural modularity, making protocols exchange information through a vertical interface. Specifically, we present the design of a cross-layer module, providing a functional analysis and a proof of concepts of its "usability" at different layers of the protocol stack. Finally, we discuss some guidelines for a possible implementation in a real operating system environment.

2.1 Introduction

Cross layering is generally intended as a way to let protocols interact beyond what allowed by standard interfaces. This clashes with the design principles of classical protocol stacks. Just to provide an example, the Internet architecture layers protocols and network responsibilities, breaking down the networking system into modular components. The resulting "strict-layered" system is composed by modules that are independent of each other and interact through well-defined (and static) interfaces, located between adjacent layers. Although this design principle brings important benefits in terms of flexibility and maintenance costs, it suffers from several characteristics of wireless networks (e.g., node mobility or power constraints), degrading

the overall network performance [CCMT05]. Hence, the need of introducing stricter cooperation among protocols belonging to different layers. This last point sets the focus of this Chapter, which aims at investigating cross-layer interactions from an architectural standpoint, in the context of mobile ad hoc networking.

In the ad hoc literature there is much work showing the potential of cross layering for isolated performance improvements. Most of these solutions focus on specific problems, as they look at the joint design of two-to-three layers. For example, in [CSN02] cross-layer interactions between the routing and the middleware layers allow the two levels to share information with each other through system profiles, in order to achieve high quality in accessing data. An analogous example is given in [SGN03], where a direct interaction between the network and the middleware layers, termed Mobile Peer Control Protocol, is used to push a reactive routing to maintain existing routes to members of a peer-to-peer overlay network. In [YLA02] the authors propose an interaction between the MAC and routing layers, where information like Signal-to-Noise Ratio, link capacity and MAC packet delay is communicated to the routing protocol for the selection of optimal routes. In [LSS05], an interaction between forwarding and power management is proposed with the goal of maximizing the amount of delivered packets, with minimum energy consumption. The fundamental idea is to adapt forwarding decisions according to both current energy level and amount of energy needed to transmit the packet. Another example is the joint balancing of optimal congestion control at the transport layer with power control at the physical layer proposed in [Chi04]. This work observes how congestion control is solved in the Internet at the transport layer, assuming link capacities to be fixed quantities. In ad hoc networks, this is not a good assumption, as transmission power, and hence throughput, can be dynamically adapted on each link. Other works propose interactions among multiple layers of the protocol stack to perform optimization involving a richer set of protocols. An example is given in [KKT04], where authors propose cross-layer interactions between physical, MAC and routing layers, to perform joint power control and link scheduling as an optimized objective. Last but not least, a framework to jointly optimize error-resilient source coding, packet scheduling, stream-based routing, link capacity assignment, and adaptive link layer techniques is proposed in [SYZ+05].

All these solutions are clear examples of benefits introduced by cross layering. However, their deployment has to deal with the following issues:

1. *Tight-coupling*: the design of cross-layer optimizations requires direct modification of interfaces, causing the involved protocols to become tightly-coupled, and therefore mutually dependent.

2. *Unbridled stack design*: while an individual suggestion for cross-layer design, in isolation, may appear appealing, combining several of them together could result in a "spaghetti" stack design [KK05], making architectural maintenance a challenging task. Moreover, an uncontrolled combination of isolated cross-layer optimizations may cause mutual interferences, which could lead single

nodes to unstable and degraded behavior, with negative impacts on the entire network.

3. *Correct system implementation*: when introducing new interactions among protocols, special care has to be taken to maintain a correct execution flow, without causing critical problems on the internals of the operating system. In real platforms, network protocols consist of a mixed set of processes executing at both kernel and user levels. For this reason, the implementation of cross-layer interactions should guarantee a correct interleaving of protocols execution, without introducing failure patterns on synchronization and scheduling of local system processes.

This Chapter addresses cross-layering from an architectural standpoint, providing a basis for tackling the semantic problems of interfering optimizations and correct system implementation. In particular, we claim that new interactions can be realized maintaining the layer separation principle, with the introduction of a cross-layer interface (XL-interface) that standardizes vertical interactions and gets rid of tight-coupling from an architectural standpoint. The key aspect is that protocols are still implemented in isolation inside each layer, offering the advantages of:

- allowing for full compatibility with standards, as the XL-interface does not modify each layer's core functions;
- providing a robust upgrade environment, which allows the addition or removal of protocols belonging to different layers from the stack, without modifying operations at other layers;
- maintaining the benefits of a modular architecture (layer separation is achieved by standardizing the usage of the XL-interface).

Engineering the XL-interface presents a great challenge (Section 2.2). This component must be general enough to be used at each layer, providing a common set of primitives to realize local protocol interactions (Section 2.3). To support this novel paradigm, we classified cross-layer functionalities and extended standard TCP/IP protocols in order use them. The result of this effort has been implemented in the *ns*2 Network Simulator (Section 2.4), realizing a simulative evaluation framework for the usability of the XL-interface at different layers. Finally, in Section 2.5 we discuss some implementation guidelines for a real operating system environment, while in Section 2.6 we conclude the Chapter.

2.2 Architectural Functionalities

We designed the XL-interface with two models of interaction in mind: *synchronous* and *asynchronous*. Protocols interact synchronously when they share private data

(i.e. internal status collected during their normal functioning). A request for private data takes place on-demand, with a protocol issuing a query to retrieve data produced at other layers, and waiting for the result. Asynchronous interactions characterize the occurrence of specified conditions, to which protocols may be willing to react. As such conditions are occasional (i.e. not deliberate), protocols are required to subscribe for their occurrences, and then return to their work. The XL-interface is in turn responsible for delivering eventual occurrences to the right subscribers. Specifically, we consider two types of events: *internal* and *external*. Internal events are directly generated inside protocols. Picking just one example, the routing protocol notifies the rest of the stack about a "broken route" event, whenever it discovers the failure of a preexisting route. On the other hand, external events are discovered inside the XL-interface on the basis of instructions provided by subscriber protocols. An example of external event is a condition on the host energy level. A protocol can subscribe for a "battery-low" event, specifying an energy threshold to the XL-interface, which in turn will notify the protocol when the battery power falls below the given value. Note that the host energy controller simply provides the current battery level value, but it is not in charge of checking the threshold and notify related events.

As the XL-interface represents a level of indirection in the treatment of cross-layer interactions, an agreement for a common representation of data and events inside the vertical component is a fundamental requirement in order to guarantee loosely-coupling. To this end, the XL-interface works with *abstractions* of data and events, intended as a set of data structures that comprehensively reflect the relevant (from a cross-layering standpoint) information and special conditions used throughout the stack. A straightforward example is the topology information collected by a routing protocol. In order to abstract from implementation details of particular routing protocols, topology data can be represented as a graph inside the XL-interface. Therefore, the XL-interface becomes the provider of shared data, which appear independent of its origin, and hence usable by each protocol.

How is protocols internal data exported into XL-interface abstractions? This task is accomplished by using *call-back* functions, which are defined and installed by protocols themselves. A call-back is a procedure that is registered to a library at one point in time, and later on invoked (by the XL-interface). Each call-back contains the instructions to encode private data into an associated XL-interface abstraction. In this way, protocol designers provide a tool for transparently accessing protocol internal data.

2.3 Designing the cross-layer interface

In order to give a technical view of the vertical functionalities, we assume that the language used by the XL-interface allows for an object oriented representation of data structures and functions. We adopt the following notation to describe the

XL-interface interface:

$$\texttt{XL\_object.method} : (\texttt{input}) \rightarrow (\texttt{output})$$

As described in the previous Section, the XL-interface does not generate shared data, but simply acts as intermediary. Protocols synchronize on an abstract representation of internal data (namely XL_data) where one *producer* protocol specifies a call-back function to export its private data to the abstract representation.

$$\texttt{XL\_data.seize} : (\texttt{callback}()) \rightarrow ()$$

On the other hand, *consumer* protocols access the shared data with read only permissions, using

$$\texttt{XL\_data.access} : () \rightarrow (\texttt{abstractData})$$

Going back to the example on network topology data, the routing agent plays the role of the producer protocol, exporting routing tables into an abstract graph representation. Consumer protocols living in the scope of other layers, could gather network topology information calling the *access*() method, which in turn invokes the call-back function registered by the routing agent. This makes the interaction between producer and consumer protocols loosely-coupled, avoiding direct protocol dependencies.

The remaining functionalities of the XL-interface cope with asynchronous interactions. In the case of internal events, the role of the XL-interface is to collect subscriptions, wait for notifications, and vertically dispatch event occurrences to the appropriate subscribers. A protocol *subscribes* for a cross-layer event (namely XL_event) by calling the function

$$\texttt{XL\_event.subscribe} : (\texttt{handler}()) \rightarrow ()$$

Note that the subscriber protocol has to specify a handler function, which will be used by the XL-interface to notify occurrences and triggering event handling. So, subscriber protocols play again the consumer role, while producer protocol *notify* event occurrences by calling

$$\texttt{XL\_event.notify} : () \rightarrow ()$$

The XL-interface is in charge of maintaining a subscription list for each kind of cross-layer event, dispatching occurrences to the correct subscribers.

In the case of external events, the XL-interface must additionally act as event notifier. The idea is that some protocols might be interested in conditions that are not directly verified by other protocols. To this end, subscriber protocols instruct the XL-interface on how to detect the event. The detection rules are embedded in a monitor function, which periodically checks the status of the cross-layer abstractions under inquiry. When the monitor detects the specified condition, the XL-interface

dispatches the information to the subscriber protocol. A protocol initiates the *monitoring* of an external event by passing a monitor and a handler function to the XL-interface, through the following method of the target data abstraction

$$\texttt{XL\_data.setMonitor} : (\texttt{monitor}(), \texttt{handler}()) \rightarrow ()$$

The XL-interface serves this call by spawning a *persistent* computation that executes the steps described in Algorithm 1.

Algorithm 1 Cross-layer data monitoring.

```
while true do
  freshData = XL_data.access()
  if monitor(freshData) then
    handler()
  end if
end while
```

2.4 Using the Cross-layer Interface

In order to practice the usage of the XL-interface, we realized a simulation framework, based on the Network Simulator *ns*2 (v. 2.27) [NS2], and a library of objects and abstractions, called ProtoLib, provided by the Naval Research Laboratory (NRL) [NRL].

2.4.1 The Proto Library

The protocol Prototyping Library (ProtoLib) is not so much a library as it is a toolkit. The goal of the ProtoLib is to provide a set of simple, cross-platform C++ classes that allow development of network protocols and applications that can run on different platforms and in network simulation environments. Although ProtoLib is principally for research purposes, the code has been constructed to provide robust, efficient performance and adaptability to real applications. Currently, ProtoLib supports most Unix platforms (including MacOS X) and WIN32 platforms. The version used in this thesis also supports building ProtoLib-based code for the *ns*2 simulation environment.

The approach behind the ProtoLib environment is to provide the programmer with object-oriented abstractions of common networking tools and components, like network addresses, sockets and timers, which are then mapped to a target platform with a specific implementation. Hence, the main idea is to have the programmer writing its networking code using the abstractions, so as to have it working on a significant set of real-platforms as well as simulation environments (see Figure 2.1). This last feature seems to be particularly useful for evaluating and debugging distributed protocols and algorithms (e.g., forwarding agents like REEF) before moving

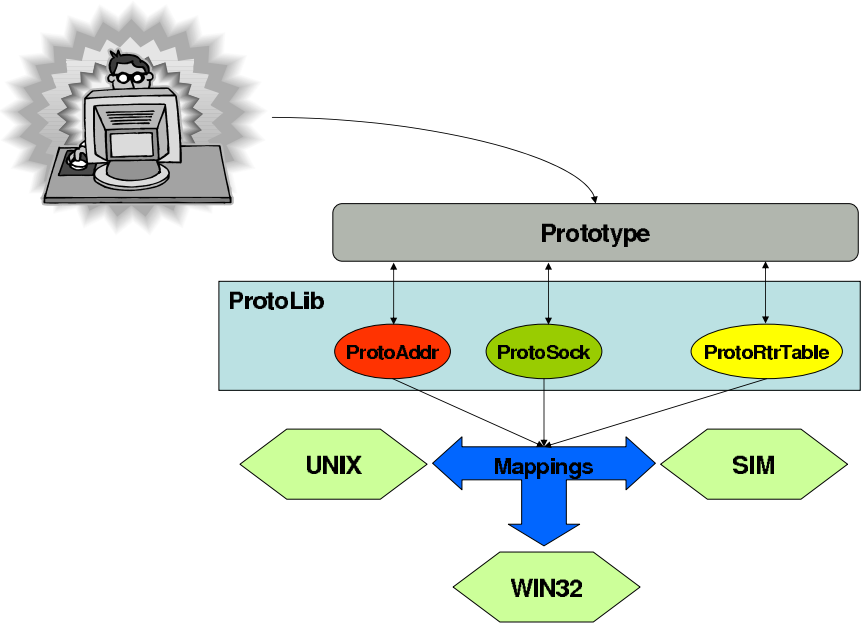


Figure 2.1: The ProtoLib framework.

on to "real" environments, but maintaining code and data structures unaltered. In the following descriptions, we go through the main abstractions provided by the ProtoLib.

ProtoAddress Network address container class with support for IPv4, IPv6, and "SIM" address types. Also includes functions for name/address resolution.

ProtoSocket Network socket container class that provides consistent interface for use of operating system (or simulation environment) transport sockets. Provides support for asynchronous notification to ProtoSocket listener objects. The ProtoSocket class may be used stand-alone, or with other classes described below. Currently, a ProtoSocket may be instantiated as a UDP socket, while TCP sockets will be supported in the future.

ProtoTimer This is a generic timer class which will notify a ProtoTimer listener object upon timeout.

ProtoTimerMgr This class manages ProtoTimer instances when they are "activated".

ProtoTree Flexible implementation of a Patricia tree data structure. It includes a Item object which may be derived from or used as a container for whatever data structures and application may require.

ProtoRouteTable Class based on the ProtoTree Patricia tree to store routing table information. It uses the ProtoAddress class to store network routing

addresses.

ProtoRouteMgr Base class for providing a consistent interface to manage operating system (or other) routing engines. Examples include the routing sockets in UNIX/POSIX platforms.

ProtoApp Provides a base class for implementing Protolib-based command-line applications. Note that "ProtoApp" and "ProtoSimAgent" are designed such that subclasses can be derived to reuse the same code in either real-world applications or as "agent" entities within network simulation environment (e.g., *ns*2, OPNET etc.).

ProtoSimAgent Base class for simulation agent derivations. Currently an *ns*2 agent base class is derived from this class, but it is possible that other simulation environments (e.g. OPNET, Qualnet etc.) might be supported in a similar fashion.

NsProtoSimAgent Simulation agent base class for creating *ns*2 instantiations of Protolib-based network protocols and applications.

Another important feature is that the ProtoLib already delivers an implementation of Optimized Link State Routing protocol (OLSR), compliant with the latest specification [CJ03].

In the framework resulting from the integration of *ns*2 and the ProtoLib, it was a "natural" choice to place the objects of the XL-interface inside the ProtoLib. We engineered them as abstract classes that other protocols can implement in order to share data and exchange local events. Specifically, we realized interfaces for *XL_Data* and *XL_Event* objects, respectively for sharing protocol internal data (i.e., synchronous interactions) and for subscribing/notifying internal events (i.e., asynchronous interactions). In the following, we briefly describe the functionalities of the new objects:

ProtoXLData This is a generic class (see Figure 2.2) that identifies internal data *owned* (*produced*) by a protocol and *shared* to (*consumed* by) the rest of the network stack. It offers methods to declare ownership of the data and to specify a call-back function for "translating" the internal data format used by the owner, in a cross-layer ontology common to the whole stack. Other protocols access instances of this class with read-only permissions.

ProtoXLEvent This is a generic class (see Figure 2.3) that identifies conditions or events detected internally to the protocol, which may result of interest for the rest of the stack. It offers methods to *subscribe* interest in events derived from this class, as well as to *notify* occurrences of them.

In the following, we present two examples of cross-layer optimization based on the XL-interface. We show interactions involving network, transport and middleware

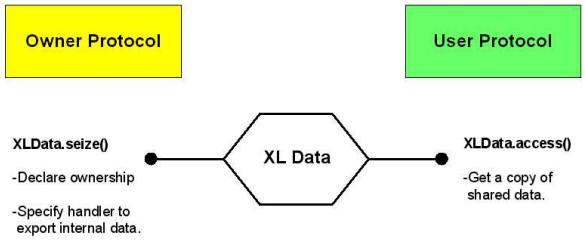


Figure 2.2: Cross-layer data.

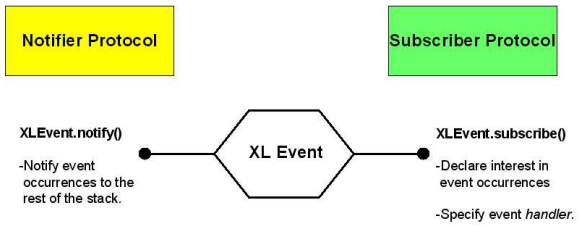


Figure 2.3: Cross-layer events.

layers, to highlight how the objects of the XL-interface suites different levels of the protocol stack. The two case studies implement their cross-layer interactions by specializing the base objects presented in the previous Section.

2.4.2 Improving the performance of data transfer

In this Section, we show how the XL-interface has been used to cope with performance issues of TCP data transfer. TCP performance degrades in ad hoc environments due to losses, which are induced by fault conditions (e.g. network partitions, route failures, and misbehaving nodes), and are erroneously interpreted as effects of congestion. To deal with this problem, we introduce a forwarding mechanism able to improve the performance and reliability of data transfer, also in presence of misbehaving (e.g. selfish) nodes, by means of a cross-layer interaction between the forwarding and transport agents. This mechanism is based on multi-path forwarding and estimates neighbors reliability according to end-to-end acknowledgments. Specifically, in case of reliable data transfer, TCP acknowledgments are used as delivery notifications. The reception of a TCP ack at the transport layer indicates that the corresponding sent packet has been correctly delivered at destination, and hence correctly forwarded by intermediate nodes. Each node estimates only neighbors' reliability, and uses this index to forward packets on most reliable routes, so as to avoid unreliable paths and minimize congestion events. For further details on the forwarding mechanism we point the reader to [CGM05].

The realization of the forwarding mechanism in our evaluation framework involves the introduction of a class of cross-layer events of type *Recv TCP-ack/nack*,

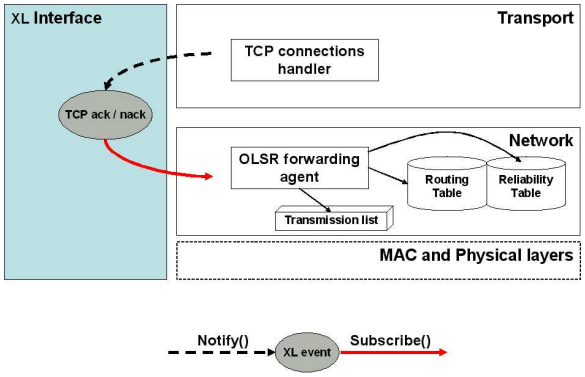


Figure 2.4: Cross-layer interaction between the network and the transport layer.

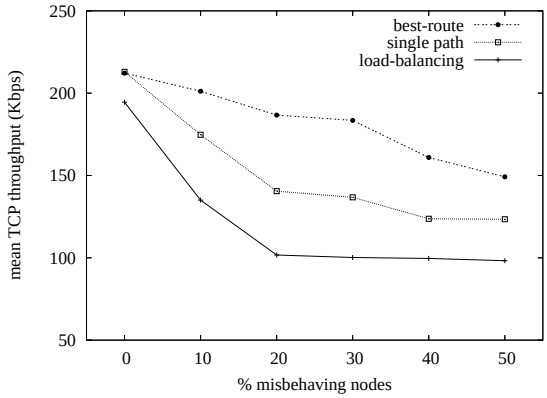


Figure 2.5: Mean TCP throughput as function of the percentage of misbehaving nodes.

to which the forwarding agent subscribes for notifications coming from a local TCP agent (see Figure 2.4). Specifically, the TCP agent notifies a TCP-ack event to the forwarding agent whenever it receives a valid acknowledgment. Instead, TCP-nack events are caused by packets retransmissions. An event notification causes the forwarding agent to update the reliability index associated to the neighbor through which the packet passed. The update is positive for TCP-ack and negative for TCP-nack. Reliability indexes are used to send packets on most reliable routes. Specifically, we implemented a forwarding policy which chooses the route with the smallest route-length/reliability ratio, namely *best-route* forwarding. As the simultaneous use of multiple paths (i.e., *load-balancing*) degrades TCP performance [LXG03], we compare our best-route forwarding policy with the conventional case in which packet forwarding is based on single path routing (like in OLSR), where the shortest route is always chosen (i.e., *single path*).

The simulation study investigates TCP performance by varying the percentage of misbehaving nodes. The simulated network is composed of 20 nodes, with 5 active Telnet sessions. Connection endpoints are generated randomly, and each simulated

scenario is characterized by an increasing number of misbehaving nodes that cooperate to routing, but do not forward TCP traffic. Figure 2.5 shows how best-route outperforms single path forwarding whenever misbehaving nodes are present, while the two method are comparable in cooperative networks. Specifically, the performance gain achieved with our forwarding mechanism grows up to 50% in the case of 20% and 30% of misbehaving nodes. Results also confirms the poor performance achieved by the load-balancing policy, that increase the chances of encountering misbehaving nodes.

2.4.3 Improving the quality of unstructured overlays

In this Section, we show how the XL-interface has been used to improve the performance of Gnutella, a well-known unstructured overlay platform. Although we used Gnutella as a case study, a similar approach could be used for other P2P platforms, so as to make them more reactive and usable in ad hoc environments. Full details about this application of the XL-interface are reported in [CGT05].

By simulating a fully-fledged Gnutella system in ad hoc environments, we identified peer discovery as a critical issue. In summary, discovery procedures based on application layer flooding generate overhead, and decrease the capacity of building the overlay. Moreover, as peer selection is random, Gnutella overlays are significantly sensitive to nodes mobility, and fail to react promptly in scenarios with partitioning or heavy churn rates. Under these observations, we re-designed peer discovery and link selection in order to interact with the routing agent at the network layer. The fundamental idea is to exploit node discovery procedures provided by routing agents, so to jointly perform peer discovery together with gathering topology information. For example, in a proactive routing protocol like OLSR, nodes periodically issue Hello and Topology Control messages, containing information about the neighbors that they currently sense. This information could be enriched with *Optional Information* (OI), containing peer credentials (e.g., IP address and port number of the Gnutella service). This approach saves the network resources consumed by an explicit peer discovery protocol.

We modeled the cross-layer interactions using events between Gnutella peers and OLSR agents (see Figure 2.6). We initially extended the NRL implementation of OLSR to handle new messages for *optional information*, and afterward specialized two classes of cross-layer events: i) *Spread OI* events, to which routing agents subscribe, receiving notifications from Gnutella peers. These events are used to ask OLSR agents to advertise local peer credentials around, along with the next Hello or Topology Control message (see OLSR RFC [CJ03] for details on the protocol); ii) *Recv OI* events, to which Gnutella peers subscribe, in order to receive notifications from underlying OLSR agents. These events are used to notify Gnutella peers about incoming credential advertisements of remote peers. This allowed us to realize peer discovery by making each peer periodically advertise its credentials, and reacting to events of advertisements reception. The overall discovery procedure became simpler

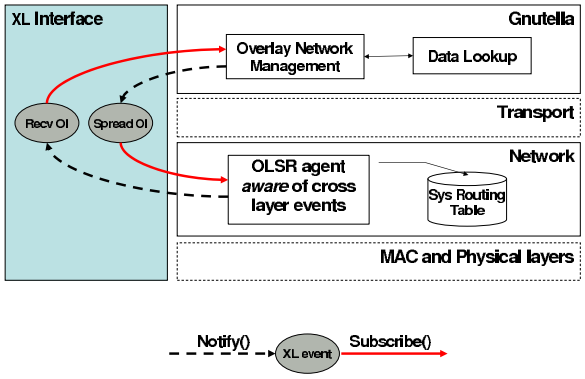


Figure 2.6: Event-based cross layer interaction for Gnutella peer discovery.

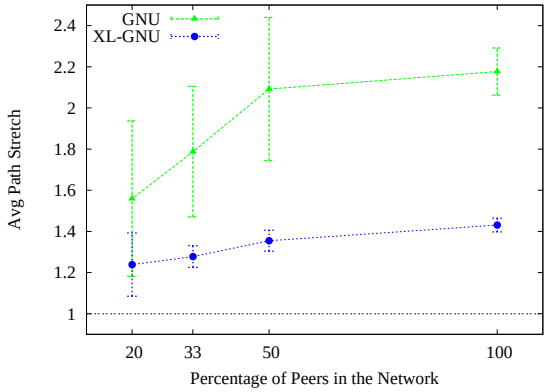


Figure 2.7: Comparison of the average path stretch produced by Gnutella and XL-Gnutella under increasing peer densities.

and easier to control. On receiving cross-layer events, peers were able to fill up a local table of advertisement generated by foreign agents. Moreover, as advertisements travel the network along with routing control packets, it was possible to get accurate estimates of peers physical distances (in number of hops). This topological information enriched the advertisement table, and allowed us to play a smarter overlay formation protocol, introducing a link selection policy based on physical distances. The rational behind was to simply prioritize closer connections over further ones, with the goal of building an overlay topologically closer to the physical network. In order to give a flavor of the benefits introduced by the XL-interface, Figure 2.7 shows the results obtained by studying the *path stretch* generated by the legacy and the cross-layer version of the protocol, defined as the ratio of the number of hops (in the physical network) along the path connecting two peers in the overlay, to that along the direct unicast path. This metric measures how far (from a topological point of view) the overlay is from the physical network, and characterizes the overhead induced by the former on the latter. By configuring an increasing percentage of peers in a network of fixed size, we observed that cross-layer Gnutella (XL-Gnutella) produces better path stretches (e.g., respectively 1.35 against 2.1 of legacy Gnutella

with a 50% of peers), exhibiting a stable behavior with smaller variances.

2.5 Implementation guidelines

In this section, we discuss some issues related to a possible implementation of the XL-interface, intended as an operating system service that supports programmers to realize protocols with cross-layer interactions. We structure our discussion on the base of operating system concepts typical of UNIX/Linux environments.

First of all, we would like to re-emphasize that from an implementation standpoint, a cross-layer interaction is, as a matter of fact, an exchange of data (synchronous or asynchronous) between two processes (or tasks) running on a system. Common OS platforms already come with plenty of support for interprocess communication, so the key role of the XL-interface should be to *coordinate* and *manage* the usage of existing system services to enable cross layering without causing negative interactions on normal system operation. In particular, in this section we would like to address the problem of *scheduling interferences*.

Common UNIX/Linux environments organize computation in processes, making a clear distinction between kernel and user computations. A computation is said to live in *kernel space*, if it is executing code that comes from the kernel image or a module image. The code has full permissions to do whatever it wants (i.e., privileged execution). This may be a task or an invoked interrupt. If it is a task, it is not pre-emptible, meaning that no entities can schedule the CPU away from it. If it is an interrupt, it just temporarily uses the CPU. Alternatively, a computation is said to live in *user space*, if it is executing code that comes from a normal process image. This code runs in unprivileged mode, and is protected from affecting other process's or the kernel's address space. In modern architectures, a user process can enter kernel space by means of *system calls*, well known points that a process must invoke in order to enter the kernel. When invoking a system call, the hardware is switched to the kernel settings, where the process will be executing code from the kernel image. At this stage, unlike when in user space, it has full powers to wreak havoc, and becomes no longer pre-emptible. Transfer of data between user and kernel space only make sense through system calls, and usually requires direct memory copies.

Given the above considerations, let us come to processes and tasks that make up a networking stack. As illustrated in Figure 2.8, some protocols run as processes in user space, while others are implemented as tasks and data structures in kernel space, and are accessed by system calls. Typically, networking middleware and applications are user space processes that make use of transport services implemented in kernel. As underlined in [Kaw04], the networking layer is conceptually split in two functionalities: i) *forwarding* consists of taking a packet, consulting the system forwarding table, and sending the packet towards its destination as determined by that table; ii) *routing*, refers to process of building the forwarding table, discovering

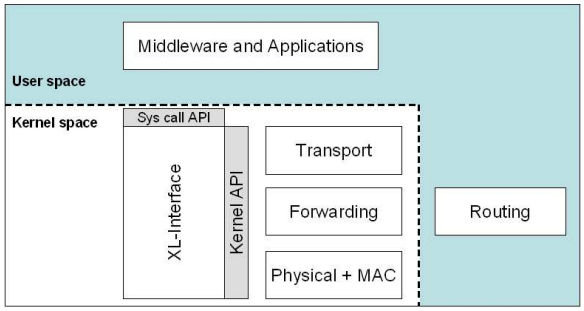


Figure 2.8: Schematic representation of a system network stack.

routes towards given destinations. While the former requires very fast execution, and therefore lives in kernel space, the latter is a distributed algorithm that lives in user space. As a result, when discussing how *synchronous* or *asynchronous* interactions should be implemented, we have to distinguish four different cases:

Kernel-To-Kernel when both the producer/notifier and consumer/subscriber protocols are tasks in the kernel space;

Kernel-To-User when the producer/notifier protocol is a kernel space task, while the consumer/subscriber is a user space process;

User-To-Kernel when the producer/notifier protocol is a user space process, while the consumer/subscriber is a task living in kernel space;

User-To-User when both the producer/notifier and consumer/subscriber protocols are processes in user space.

In this system picture, the XL-interface lives in kernel space, probably implemented as a loadable module, which exports a set of systems calls for processes in user space, as well as a kernel API for protocol implemented as kernel tasks. In the following, we discuss the above four cases for synchronous and asynchronous interactions.

2.5.1 Synchronous interactions

Let us begin with synchronous interactions, recalling that they represent sharing of internal protocol data. For this kind of interactions we can differentiate depending on the nature of the producer protocol. This is because the XL-interface has to guarantee that a consumer process does not get pre-empted in accessing a shared object.

A *Kernel-To-Kernel* interaction requires the simplest handling and management. When both protocols are kernel tasks, they run in privileged and not pre-emptible mode, and can directly pass pointers to internal data buffers and functions. So

an access to a shared object causes a call to a XL-interface function in the kernel API, which in turns invokes the call-back registered by the producer protocol. All of this happens without changes in context for the consumer task. A *Kernel-To-User* interaction can be handled in an analogous way, with the difference that the consumer process will begin from a system call, that switches it from user to kernel space mode, making it not pre-emptible. Additionally, this case surely requires a copy of the abstracted data from kernel to user space, which can be handled at the end of the system call.

A different discussion applies for *User-To-Kernel* and *User-To-User* interactions. As the producer protocol is a user space process, invoking its call-back would inevitably cause the pre-emption of the consumer protocol. A solution to this problem would be to reserve a shared memory segment for each cross-layer object, and work around the call-back invocation, having the producer protocol proactively *refreshing* the abstract content. To this end, we require an additional system call like

`XL_Data.refresh`()

which is invoked whenever the producer protocol modifies the cross-layer object. In this way, the shared object may be accessed from the consumer protocol by *attaching* the memory segment from either user or kernel space. Probably, in the *User-To-User* case, some form of locking on the segment is required (e.g., typically system semaphores).

2.5.2 Asynchronous interactions

Avoiding scheduling problems in the case of asynchronous interactions, can be solved with the same mechanisms used by modern kernels to handle interrupts. Any serious interrupt handling is divided in two halves: a *top* half (i.e., the registered interrupt routine) catches the interrupt, copies over any relevant data, and schedules a *bottom* half for execution at safer times in the future. This basically means to postpone longish and complex interrupt data handling, so that the interrupt service routine itself quickly finishes the job and minimizes the probability of loosing newly incoming interrupts. Modern OS's do provide support for these situations, and particularly UNIX/Linux kernels export services like `task_queues` and the more recent `tasklets`, to postpone task execution at safer time.

In the case of asynchronous interactions, the XL-interface clearly doesn't need to install interrupt service routines, but can use task queues or tasklets to postpone the delivery of a cross-layer event from its notification, and work around scheduling issues.

Given the above, the notification of a XL_Event is translated into the creation and scheduling of a tasklet with instructions to wake up the right subscriber and deliver the event. In *User-To-User* and *User-To-Kernel* cases this top-half handling is done inside the appropriate system call `XL_Event.notify()`. The bottom-half

remains pretty simple for both cases, with the only difference that in one case it wakes up (probably) a thread in a user process, while in the other a kernel task. For *Kernel-To-Kernel* and *User-To-Kernel* cases, the difference stands in the top-half processing, which is done in a kernel API function instead of system call.

2.6 Conclusions

Cross layering represents a trendy solution to overcome performance limitations of mobile ad hoc environments. Current proposals testify the effectiveness of cross layering in delivering better protocol performances, but they tackle single cases without prospecting any form of coexistence from an architectural standpoint. The contribution of this Chapter is the design of an interface able to support several cross-layer solutions, using common interaction models. This approach decouples interacting entities, and preserves the flexibility and modularity features of legacy architectures.

In order to evaluate the usability of the proposed interface, we considered two case studies, verifying that the cross-layer primitives could be used at different layers of the stack, for different purposes.

The next step is to deploy the cross-layer interface on a real platform, according to the guidelines for the guarantee of a clean execution pattern, without mutual scheduling interferences.

Chapter 3

A Cross-Layer Approach for Unstructured P2P Computing

Abstract

In recent years, the Internet has witnessed the introduction of many peer-to-peer systems designed to realize large-scale data sharing. These platforms exhibit interesting features like self-configuration, self-healing and complete decentralization, which make them appealing for employment in ad hoc environments as well. However, the impact of ad hoc dynamics on the performance of these protocols, and the different set of constraints which this emerging networking paradigm imposes, haven't been yet carefully evaluated. This Chapter investigates the performance of Gnutella, one of the most widely used peer-to-peer systems, when put through typical ad hoc conditions like node mobility, frequent network partitioning, etc.. We show that a straightforward implementation of the protocol is not satisfactory under the point of view of the produced overhead and the average overlay connectivity. Finally, we propose a cross-layer optimization of Gnutella, which enhances its performance up to the expectations and makes it more suitable to the degree of self-organization and self-healing required in ad hoc environments.

3.1 Why Gnutella on ad hoc?

The future information society is expected to rely heavily on wireless technology. Mobile access to the Internet is steadily gaining ground, and it could easily end up exceeding connections from the fixed infrastructure. Picking just one example, ad hoc networking is a new paradigm of wireless communication for mobile devices. While initially targeted at military applications, as well as stretching the access to the Internet beyond the first wireless hop [CMC99], ad hoc networks are now expected to be employed in civilian applications. Making these systems working

autonomously to support opportunistic communications among users, still presents challenging issues on topics ranging from wireless technologies to middleware platforms and applications. This Chapter focuses at the middleware layer, investigating on how information sharing could be efficiently realized in ad hoc environments through P2P systems.

Information (or data) sharing is an application layer task that allows different devices to share data in order to carry out distributed computations and satisfy user needs. Data sharing is considered of fundamental importance for mobile ad hoc networks, since they have been proposed to enable data access for mobile devices in the absence of an infrastructure. The big variety of applications that could benefit from an efficient data sharing technique, suggests to cope with this problem at the middleware layer. One of the primary roles of middlewares is to provide application programmers with a set of high-level instruments and abstractions, in order to hide the complexity of underlying systems, properly managing resources and critical operations. In [MCE02] the authors provide a discussion on the characteristics of middleware for mobile computing, identifying among others the organization of distributed services with fair workload distribution, tolerance to dynamics such as the variability of participants to the distributed service, and correct handling of intermittent connectivity and devices mobility. Some, but not all of these features are already offered by P2P file sharing systems recently proposed in the Internet. Open platforms like Gnutella [KM02], but also research proposals like Pastry [RD01a], distribute data (e.g., knowledge, music etc.) without the need of any central server. Under this aspect, ad hoc networks can be considered a computational duality with P2P systems, where peers self-organize in *logical* networks (i.e., overlays) and cooperate independently to make libraries of digital content. For both P2P systems and ad hoc networks, the lack of cooperation or an unfair workload distribution could easily determine severe performance degradation.

The success of a data sharing P2P system lies on the ability to search for and lookup data efficiently. The best way to search in a given system depends on the needs of the application. For example, subject-based search techniques (e.g., Pastry) are well-suited for applications focused on availability, like file or archival systems, because they guarantee location of content if it exists, with bounded costs. To provide these goals, subject-based system tightly control the placement of data among peers, as well as the topology of the overlay network. These features come at the expense of flexibility, as currently search can only be done by subjects (i.e., unique identifiers associated to data items). In contrast, content-based systems, such as Gnutella, are designed for more flexible applications with richer queries. These platforms therefore operate under a different set of constraints, avoiding assumptions on data placement and overlay connectivity, and relaxing both cost bounds and availability guarantees.

In both subject-based and content-based platforms, peers maintain a set of logical links among them, which all together form the *overlay* network. Queries, query results and other control messages are all sent exclusively along overlay links, inde-

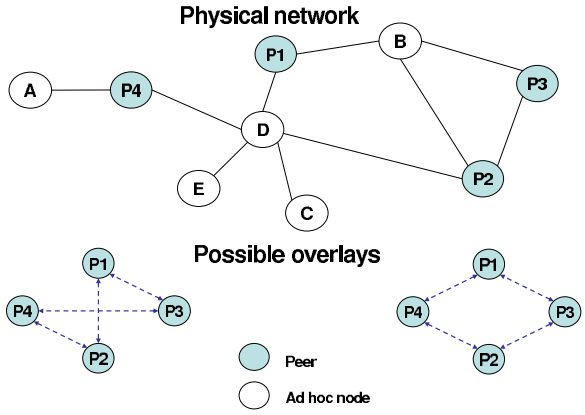


Figure 3.1: Different overlays on the same physical network.

pendently from the physical location of the involved peers. In principle, the graph topology of an overlay network may look significantly different from that of the underlying physical network (see Figure 3.1). This depends on the way peers discover themselves and on the policies used to choose and establish logical links. Clearly, this process has direct impact on the workload imposed on the underlying network, as each logical link spans across an arbitrary number of physical hops. What is the impact of current overlay management algorithms on ad hoc networks? Are platforms like Gnutella or Pastry, ready to be used in infrastructure-less and mobile environments? Is there any need or room for performance improvement?

This Chapter investigates these questions by looking at the performance of Gnutella, an open and widely used systems, in ad hoc environments, and proposes a cross-layer optimization of the protocol to address the last question. We claim that a cross-layer interaction between a P2P platform and the routing agent at the network layer, simplifies overlay management and improves the quality of the resulting overlay. The key idea is to exploit the topological knowledge collected by on-board routing agents at the network layer to simplify peer discovery procedures, and enable a smarter construction of the overlay. For example, peers may favor the establishment of closer relationships, instead of peering with far away entities. In particular, the main targets addressed by this work are: i) to decrease the network traffic (i.e., overhead) generated for overlay management purposes, making it stable against increasing mobility and node churns; ii) to improve the quality of the resulting overlay, making it closer to the underlying network topology and able to deliver high query success rates; iii) to improve features like self-organization and self-healing, especially against typical ad hoc situations like network partitioning. Our contributions are organized as follows:

- We present an overview of the Gnutella protocol (Section 3.2), based on the specification written by the Gnutella Development Forum [KM02]. Using simulation (Section 3.2.3), we demonstrate how Gnutella, if implemented in a straightforward way, can have serious performance problems when employed

in ad hoc environments.

- We identify the key issues behind the poor scalability of Gnutella, and propose a *cross-layer* interaction with the routing agent at the network layer (Section 3.3) to enhance the system performance. We underline the importance of a full cross-layer protocol stack architecture that makes feasible the implementation of our proposal in real systems.

- Through further simulation (Section 3.3.2), we demonstrate how the cross-layer approach enhances the performance and adaptability of Gnutella, making it suitable for usage in ad hoc environments.

3.2 The Gnutella protocol

In this section, we describe the Gnutella protocol for overlay maintenance and data lookup. For more details, please refer to the latest specification [KM02]. Some of the information in this section is not part of the original protocol (e.g., the behavior of a peer accordingly to its connectivity in the overlay), but represents implementation details added for clarity. Note that the Gnutella specification makes distinction between *ordinary* and *super* peers. Super-peers are those making up the overlay and providing the search infrastructure, and are usually represented by nodes with permanent Internet connection. In contrast, ordinary (or leaf) peers have intermittent connectivity, so they don't take part to the overlay formation, but simply attach themselves to an arbitrary number of super-peers, proxying queries through them. In the context of this Chapter, we are interested in the general properties of the overlay formation protocol, so we don't consider ordinary peers in our simulation models.

3.2.1 State maintenance

Gnutella operations rely on the existence of an *unstructured* overlay network. Peers open and maintain application layer connections among them, and messages dedicated to peer discovery, link control and data lookup, are sent exclusively along the overlay. As each peer is allowed to open only a limited amount of connections, message forwarding is a necessary co-operative task in order achieve a broad coverage of the overlay. The lifespan of messages is controlled by assigning bounded Time To Live (i.e., application layer TTL), which decrements at each logical hop.

To establish a connection, a peer P_1 initiates a handshaking procedure, sending a request message (see Figure 3.2) to another peer P_2. The handshaking ends up successfully for P_1 when it receives an accept from P_2, and for P_2 when it receives confirmation from P_1. The Gnutella specification assumes that each peer is given a *boot-server* as an entry point (i.e., first connection) in the overlay. Usually, bootserver credentials (i.e., network address and port number) come directly from

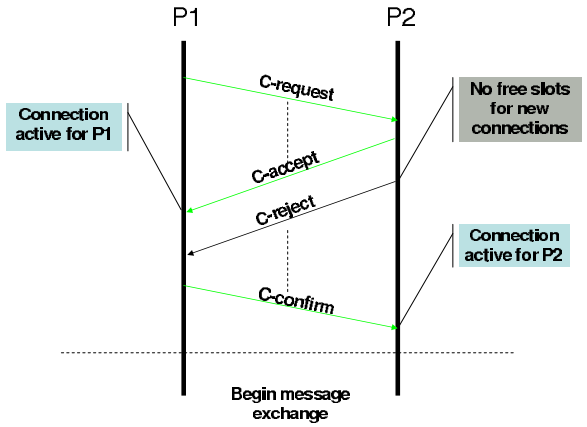


Figure 3.2: Gnutella handshaking procedure.

the user, or through a lookup against a Gnutella Web Cache[1], where on-line peers publish themselves.

As shown in Figure 3.2, two peers carry out a handshaking procedure to connect each other. The Gnutella specification assumes that each peer is given a *boot-server* as an entry point (i.e., first connection) in the overlay. Usually, bootserver credentials (i.e., network address and port number) come directly from the user, or through a lookup against a Gnutella Web Cache[2], which lists currently on-line peers. After the connection to a boot-server, each peer is required to establish more links in the overlay. The specification imposes no strict limits on the number of per-peer connections, but suggestion are given to pro-actively open a minimum amount of connections (e.g., 2 or 3) and then accept incoming connection requests to fill the remaining slots, assuming that each peer has an upper bound for the number of links to open. This guarantees that each peer maintains a minimal amount of connectivity (i.e., a lower bound LB), without overdoing it (i.e., exceeding an upper bound UB) and consequently abusing network resources. This suggests, as also outlined in [CCR04], to model the behavior of a Gnutella peer through the following states:

1. While the peer has less than LB active connections, it stays in a *connecting* state, where it performs peer discovery, initiates connections towards newly discovered peers, and accepts connection requests coming from foreign peers.

2. As the peer reaches LB connections, it enters a *connected* state, where it stops doing peer discovery, but keeps accepting incoming connection requests, as long as it has free slots available. Clearly, if the number of active connections falls back down under LB, the peer goes back in the *connecting* state.

3. Finally, when the peer reaches UB connections, it enters a *full* state, and stops accepting incoming connection requests. Again, by falling down under UB the

[1]See `www.gnucleus.com/gwebcache/` as an example.
[2]See `www.gnucleus.com/gwebcache/` as an example.

Name	Default Value
Probe interval	30 sec.
Probe retries	2
Discovery Ping interval	3 sec.
Pong cache threshold (PT)	5
Pong cache entry lifetime	10 sec.
Lower bound (LB)	4 connections
Upper bound (UB)	8 connections

Table 3.1: Gnutella protocol parameters.

peer returns in the *connected* state.

In each of the above states, the system looks up for data on demand (i.e., driven by the user), and periodically probes active connections, using dedicated probe messages (Ping messages with TTL equal to 1). Connections get terminated after a specified number of probe Pings expire (i.e., see Probe retries and Probe interval in Table 3.1), or upon an intentional Bye message.

3.2.2 Peer discovery, pong caching and queries

We now briefly describe how peers discover each other, pointing at Algorithm 2 for a pseudo-code description. After having established their first connection, peers discover other agents by issuing multi-hop discovery Ping messages (i.e., TTL equal to 7). At each hop in the overlay, the TTL field of discovery Pings gets decremented, before the message is forwarded to each active connection listed by the current peer (except the one where the Ping came from). This bounds the horizon of the flood to 7 hops, "the edge" where discovery Pings are discarded due to expiration. Peers receiving a valid discovery Ping, reply back with a Pong message containing their credentials. Note that this last step is executed only if the peer is not in state *full*: in this state it could not even accept incoming connection requests, and hence it avoids spreading its own credentials around. Pong replies are given enough TTL so that they are able to reach the Ping originator, which can then use the embedded credentials to open new connections. Pong replies are back propagated along the path followed by the Ping. This is possible because Ping originators associate unique identifiers to discovery messages, allowing intermediate peers to remember where Pong replies should be back propagated.

The standard discovery procedure can be enhanced with Pong caching. On receiving Pong messages, a peer stores the embedded credentials in a local cache. Incoming Ping messages could then be directly answered if the local cache contains enough items (see Pong cache threshold in Table 3.1), without further forwarding the discovery Ping. In this case, a certain number of items are selected from the Pong cache and directly replied to the originator. Otherwise, if the cache does not

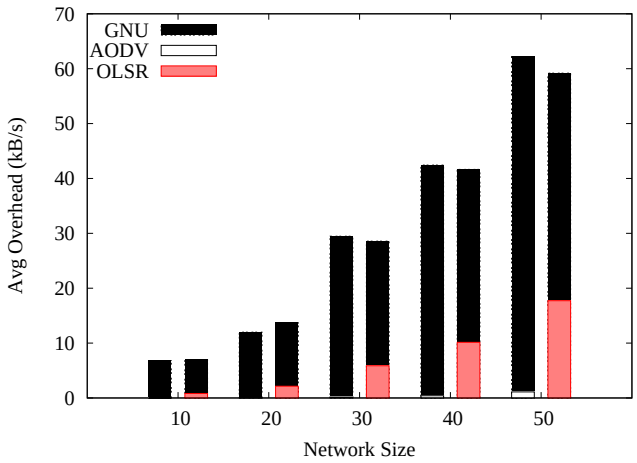


Figure 3.3: Classification of the average overhead, as a function of the network size. The bars show the amount of traffic generated by Gnutella and the routing protocols in each set of scenarios.

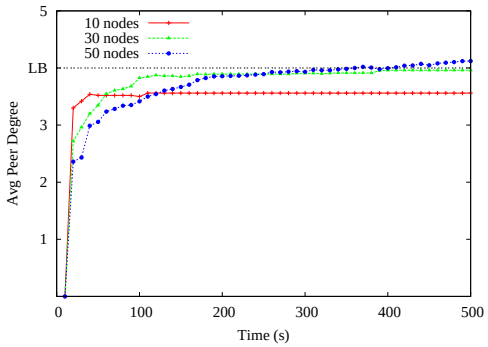
contain enough items, the peer performs standard Ping forwarding. This caching scheme significantly reduces the discovery overhead.

Queries are handled similarly to discovery Pings. On receiving a Query message, a peer looks up the locally shared content using the constraints contained in the Query. If one or more matches are found, the peer replies back with a Query Hit, providing pointers to local results. In any case, the peer decrements the Query TTL field, and forwards it to its neighbors if it is still positive. Please note that subsequent data downloads are carried out outside the overlay through direct file transfers.

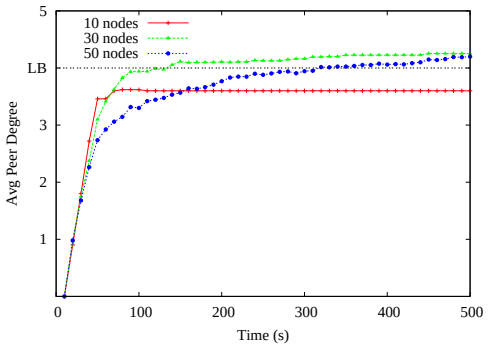
3.2.3 Performance evaluation

To evaluate the performance of Gnutella in ad hoc environments, we used the latest version of the Network Simulator 2 (version 2.27). In this Section, we study the overhead produced by peer discovery and overlay maintenance procedures, as well as the average peer degree (i.e., average number of per-peer active connections) of the resulting overlays. We implemented the relevant networking and state maintenance behavior as specified in [KM02], using the PROTOLIB framework [NRL] provided by the Naval Research Laboratory. PROTOLIB extends ns2 introducing abstractions for network addresses, sockets, timers and others objects, which are then mapped onto simulator objects, as well as onto real platforms like UNIX and Windows. This extension gives the flexibility of writing protocol prototypes that are suitable for both simulation and real experimentation, with minimal porting efforts.

In order to have our Gnutella implementation achieving the best performance in ad hoc environments, and ensure a fair comparison with the cross-layer enhance-



(a) Average Gnutella peer degree under increasing network sizes on AODV.



(b) Average Gnutella peer degree under increasing network sizes on OLSR.

Figure 3.4: Behavior of the Gnutella peer discovery protocol with different routing schemes, as a function of the network size.

ments (see Section 3.3), we implemented Pong caching as described in the previous Section, and we tuned the protocol up by introducing further optimizations. First of all, as peers in state *full* do not accept incoming connection requests, we programmed Gnutella to locally remember about fully connected peers, so to not consider them again in subsequent connection phases. Secondly, we allowed peers to re-consider unconnected boot-servers as possible candidates for establishing new links, when they already have an entry point in the overlay, but are still in state *connecting*. In fact, the Gnutella specification suggests to consider the bootstrap server only as entry point (i.e., no other links available). These enhancements significantly reduced overlay partitioning, due to the limited amount of peers that an hoc network can provide (dozens of peers, against hundreds of thousands of Gnutella overlays in the Internet), and the high latencies introduced by ad hoc routing protocols before finding valid routes towards the boot-servers. Finally, in order to evenly spread the bootstrapping workload, we assigned boot servers uniformly at random, feeding peers at the beginning of each simulated scenario. Table 3.1 lists the default settings used for our Gnutella implementation. In particular, taking into account the small size of ad hoc networks with respect to real Gnutella overlays, we sized the connection lower and upper bounds respectively to 4 and 8. The rest of the values have been fixed as suggested by the specification.

To better understand Gnutella capacity and limitations, we carried out simulations by varying network sizes and mobility, and also putting the protocol through typical ad hoc situations like network partitioning. Additionally, one of our main goals was to evaluate the interaction of Gnutella with underlying network protocols, so as to have a clear picture of the mutual impact. For this reason, we let the P2P platform operate over two representatives of the reactive and proactive routing protocol families, namely AODV [PBRD03] and OLSR [CJ03]. We used the imple-

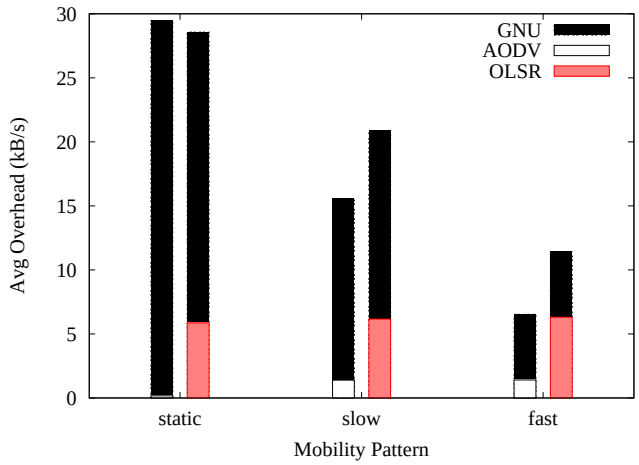
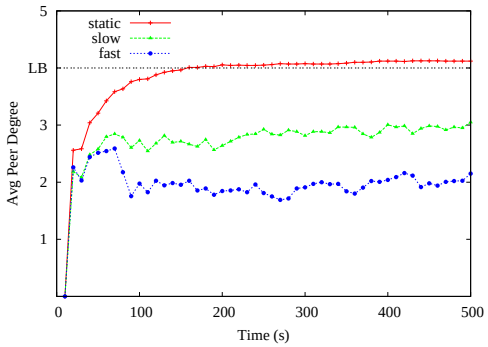


Figure 3.5: Classification of the average overhead, with patterns of increasing nodes mobility. The bars show the amount of traffic generated by Gnutella and the routing protocols in each set of scenarios.

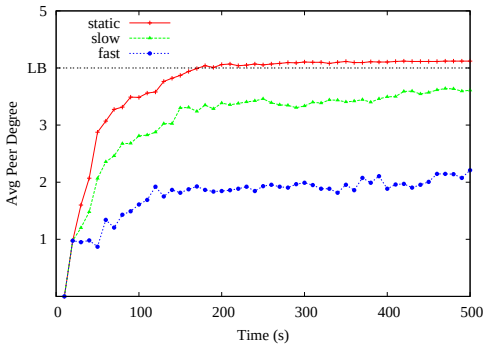
mentation of AODV shipped with ns2, while for OLSR we took the implementation provided by the Naval Research Lab in the PROTOLIB environment. If not differently specified, the simulation time was set to 15 minutes (900 s.), each node was configured with an IEEE 802.11 MAC layer to use the wireless channel, a routing agent and a Gnutella agent running on top. The results hereafter reported are the average of 5 different runs on randomly generated scenarios. In the following we present the performance analysis of Gnutella according to various parameters.

Network size

The first set of experiments aimed at evaluating the effect of increasing network size on the total overhead produced by routing agents and Gnutella peers, and on the average peer degree. To this end, we produced random scenarios with an increasing number of static and equidistant nodes, so as to maintain fixed network densities. We created 5 different classes of scenarios (i.e., 10, 20, 30, 40 and 50 nodes), where each node was configured with a Gnutella agent starting at the beginning of the simulation, immediately after the routing agent. Figure 3.3 presents the total overhead, measured in kBytes per second, produced by routing and P2P agents, with both OLSR or AODV. The figure shows that overall, the system produces a comparable amount of traffic for a given network size, independently from which routing protocol is used. The difference stands in how the overhead is distributed among the two agents. Clearly, AODV itself produces negligible overhead when compared alone with OLSR: 2 kB/s against 18 kB/s in the largest scenario. However, the impact of the two routing protocols on the traffic generated by Gnutella is visibly different: AODV induces up to 50% additional overhead in the 50 nodes scenario



(a) Average Gnutella peer degree with patterns of increasing mobility, and AODV routing.



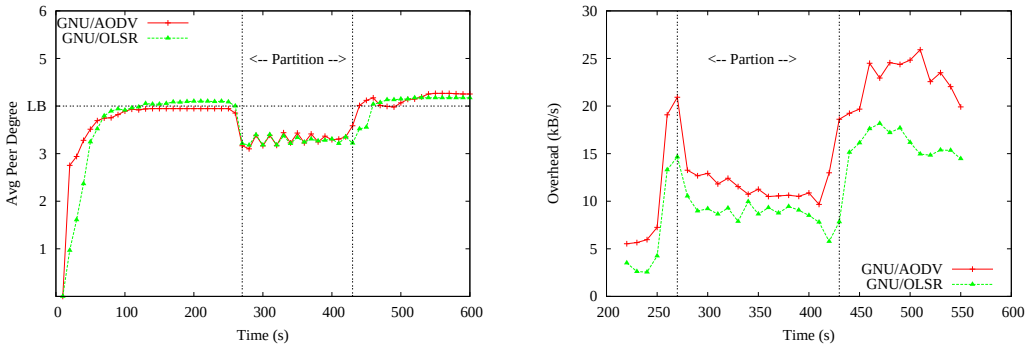
(b) Average Gnutella peer degree with patterns of increasing mobility, and OLSR routing.

Figure 3.6: Behavior of the Gnutella peer discovery protocol with different routing schemes, when nodes move with patterns of increasing mobility.

when compared to OLSR. Figures 3.4(a) and 3.4(b) show the average number of peer connections over the time, for scenarios with 10, 30 and 50 nodes. For each size, independently from the routing protocol, the average peer degree converges to the lower bound LB as the time goes by. This was the expected behavior as peers stop doing peer discovery when they reach the *connected* state. Note that in the smaller scenarios (i.e., 10 nodes), peers stabilize around an average of 3.7 connections without reaching the LB, because the number of available peers is not enough to cope with eventual overlay partitioning. This is a problem, as some peers permanently remain in state *connecting* and keep on doing peer discovery, wasting energy and network bandwidth without realizing that there are no chances to discover new peers. Finally, we observe that AODV allows a slightly shorter Gnutella bootstrapping time, as suggested by the slopes in the first 100 seconds. However, peer connectivity stabilizes with comparable latencies in both cases: around 50 seconds for 10 nodes, 100 seconds for 30 nodes and around 200 seconds for 50 nodes scenarios. To conclude, we can observe that probably these implementations of OLSR and AODV are such that the former generates routes of better quality, as Gnutella achieves comparable results with less effort. This is also verifiable in upcoming experiments with the two routing protocols.

Nodes mobility

In the following set of experiments we wanted to study the effects of node mobility on the P2P platform. To this end, we generated random way-point mobility scenarios using the *set-dest* utility shipped with ns2, considering a population of 40 nodes moving at increasing speed, and making shorter pause times inside a rectangular



(a) Effects of network partitions on the average Gnutella peer degree.

(b) Effects of network partitions on the overhead generated by Gnutella peers during discovery phases.

Figure 3.7: Analysis of the effect of network partitions on the behavior of Gnutella peer discovery procedures.

area of 1600 by 700 square meters. We created two sets of mobility scenarios: 1) a *slow* scenario with node speed uniformly ranging inside $[1, 5]$ m/s and pause times up to 10 seconds; 2) a *fast* scenario with speed uniformly ranging inside $[5, 15]$ m/s and pause times up to 5 seconds. We compared the slow and fast scenarios with a static network of 40 nodes equidistantly distributed over an area of the same size. Figure 3.5 shows the trend of the total average overhead produced by routing and P2P agents. This time AODV slightly suffers nodes mobility generating traffic up to 2 kB/s in the fast scenario, while OLSR produces a constant overhead of about 6 kB/s. In any case, the total overhead is largely dominated by the traffic generated by Gnutella agents, which almost halves at each scenario. This can be explained by looking at the trends of the average peer degree (see Figures 3.6(a) and 3.6(b)): the "slow" mobility scenario stabilized the average peer degree under LB, around 3.7 per-peer connections with OLSR and 3 with AODV; the "fast" mobility scenario made the situation even worse, stabilizing the peer degree around 2 per-peer connections with both AODV and OLSR. As we observed in the trace files, nodes mobility caused a proportional amount of link reconfigurations: the faster the mobility scenario, the more frequent the link reconfiguration. This directly influenced the validity of routes discovered at the network layer, which resulted less effective as the mobility increased. The effects on the Gnutella layer were represented by frequent overlay link drops, which caused severe overlay partitioning. So even if peers remained in state *connecting*, they issued discovery Pings on little overlays, resulting in smaller amounts of generated traffic.

Network partitioning

The last set of experiments consisted in stressing Gnutella with network partitioning, in order to see the effects on the overlay. These experiments were carried out for 600 seconds simulation time, leaving a static network of 30 peers stabilize their activity in the first 200 seconds, forcing a network partitioning around time 270, and finally resuming the original network around time 430. This was achieved by placing the 30 nodes in mutual visibility on a static grid, and letting the nodes in the center move in opposite directions, so as to break the network in two unconnected groups. Figure 3.7(a) shows the effects of the partitioning on the traffic generated by Gnutella agents, as well as on the average peer connectivity. Regarding the overhead (see Figure 3.7(b)), we observed bursts of networking activity in correspondence of the beginning and termination of the partitioning: up to 300% of traffic increase in the former, and up to 200% in the latter case. Again, as OLSR provides routes of better quality, the overhead generated by Gnutella on top of AODV results bigger of a 10% to 20% factor. The straightforward explanation for this behavior is that peers transit back in state *connecting* while the network breaks up in two halves, and perform even broader discovery procedure when the original connectivity is restored at the physical layer. These results were confirmed by the analysis on the average peer degree, which showed almost overlapping behavior for Gnutella over AODV and OLSR. In both cases, the average peer degree dropped from LB down to 3 in correspondence of the beginning of the partitioning, but completely recovered when the network resumed at its original state. This was possible by having peers re-considering unconnected bootstrap servers, as explained at the beginning of this Section. Finally, we wish to underline that this experiment was a proof of concept for the *self-healing* capabilities of the protocol: Gnutella tolerates network partitioning at the expense of a rise in the generated traffic overhead.

3.2.4 Key issues

From this preliminary set of experiments, we could verify that Gnutella meets important requirements for the management of data sharing overlay network in ad hoc environments. In particular, we mention the lack of centralization points, the co-operation of peers in forwarding each other messages, the self-organization of the overlay network given the entry points for bootstrapping, and finally the capability to recover from network partitioning.

However, Gnutella was not designed for ad hoc networks, and suffers from node mobility, as shown in Section 3.2.3, causing peers not to achieve minimum connectivity requirements. Moreover, as shown in the experiments with network partitioning (see Section 3.2.3), the protocol generates traffic bursts in correspondence of topological reconfigurations. This is not desirable in situations where the network partitions frequently, or where groups of nodes enter and leave the network (node churns). Finally, we have to observe that even if Gnutella peers are able to self-organize in an

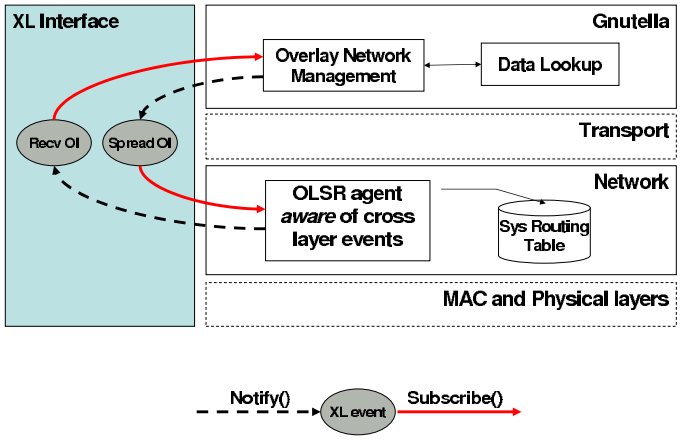


Figure 3.8: Cross-layer (XL) architecture used to optimize Gnutella performance.

overlay network, they still need the identification of a boot-server to start with. Our concern is that in opportunistic environments such as autonomous ad hoc networks, this information could be hard to retrieve. To work around this problem, a solution could be to perform an initial expanding ring search phase, where peers discover closer agents to attempt the first connection.

In the next Section we propose an alternative approach based on cross layering, which addresses the above issues providing satisfactory performances at lower costs.

3.3 Cross-Layer Gnutella

Ad hoc networking heavily relies on the capacity to form a spontaneous communication infrastructure at the network layer. Ad hoc nodes are autonomous *routers*, which discover other entities pro-actively, like in the case of OLSR, or driven on-demand by the needs of application layer requests, like in the case of AODV. In other words, ad hoc nodes have already built-in discovery procedures that "silently" work in the background to fill out system routing tables with topological information. Can Gnutella, or other P2P platforms, exploit this feature? For example, the proposal described in [KP02], is an extension of the AODV routing protocol, where service route request and response messages are added to the original routing protocol specification, in order to discover with a single broadcast those nodes offering a particular service and the routes towards them. We claim that a similar concept can be introduced for proactive protocols like OLSR. In OLSR, nodes periodically issue Hello and Topology Control messages, which contains information about the neighbors that are currently sensed. Ideally, one could add a new type of message that allows nodes to broadcast *optional information* (OI), and let the other nodes react to them as they do for link state messages. A Gnutella peer could then exploit one or the other extension, depending on which routing protocol is in use, and perform peer discovery in conjunction with route discovery at the network layer.

The question is then how such a cross-layer interaction could be feasibly implemented in standard protocol stacks. To solve this problem, we refer to Chapter 2, and the work presented in [CMTG04][CCMT05] and [CMT06], where a *vertical* stack component is introduced to extend protocols interaction capabilities beyond standard layer interfaces. This solution aims at maintaining a clean stack design, providing protocols with cross-layer interaction primitives to:

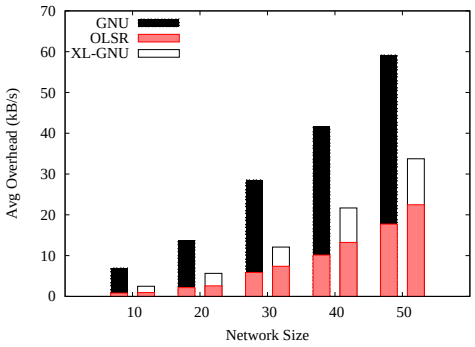
- *synchronize* on shared data structures owned by single protocols, like for example routing tables;
- react *asynchronously* to cross-layer events generated at different layers, through a local publish/subscribe framework.

Using a cross-layer interface (XL-interface) inspired from this model, Gnutella peers could initiate, through cross-layer events, peer discovery requests to an on-demand routing protocol, or ask a proactive routing agent to spread their own credentials around together with control packets. In the same way, routing agents could notify local peers about the reception of discovery requests and reply back when necessary, or, in the proactive case, notify local peers about the reception of fresh peer credentials along with incoming link state updates.

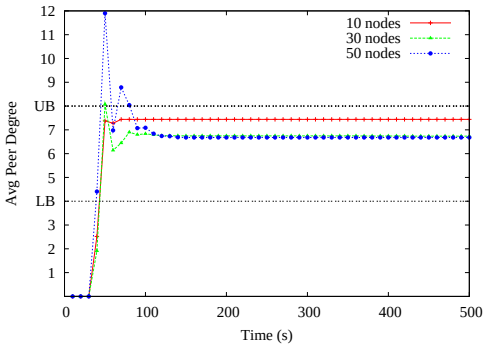
In this work, we adopted the proactive approach by extending the Naval Research Lab implementation of OLSR to handle new messages for optional information (OI). Our choice fell on OLSR because it provided the best results during the performance evaluation described in Section 3.2.3, and because the protocol itself is designed to be easily extended with new messages. Additionally, it was a natural choice to implement the cross-layer interface inside the PROTOLIB, so as to provide OLSR and Gnutella with platform independent objects for cross-layer interactions. Figure 3.8 shows the resulting system architecture, with the cross-layer communication between Gnutella and OLSR. We introduced two classes of cross-layer events:

1. *Spread OI* events, to which the routing agent subscribes, receiving notifications from the Gnutella platform. These events are used to ask the OLSR agent to advertise local peer credentials around, together with the next Hello or Topology Control message, respectively if the node is a Multi Point Relay or a leaf node (see OLSR RFC [CJ03] for details on the protocol).
2. *Recv OI* events, to which the local Gnutella peer subscribes, in order to receive notifications from the underlying OLSR agent. These events are used to notify the local peer about the credential advertisement of a remote peer, received together with a routing control message.

This cross-layer framework, allowed us to re-design the peer discovery procedure of Gnutella, making each peer periodically advertise its credentials, and reacting to events like the reception of new (or the refresh of old) advertisements, and the expiration of peers.



(a) Comparison of the average overhead generated by Gnutella and the XL-Gnutella under increasing network sizes.



(b) Average XL-Gnutella peer degree under increasing network sizes.

Figure 3.9: Analysis of the benefits of cross-layering on the overlay management, as a function of the network size.

3.3.1 Peer discovery and link selection

By adopting the cross-layer framework, we replaced the peer discovery procedure of Gnutella based on Pings flooding, with the management of cross-layer events. The overall procedure became simpler and easier to control. On receiving cross-layer events, peers fill up a local table of advertisement generated by foreign agents. This advertisement table took the place of the Pong cache of the legacy implementation. Moreover, as advertisements travel the network along with routing control packets, we were able to get each time an accurate estimation of the physical distance (in number of hops) of the peer originating the advertisement. This topological information enriched the advertisement table, and allowed us to play a smarter overlay formation protocol, introducing a link selection policy based on the physical distance of discovered peers. In other words peers were able to prioritize the establishment of closer connections over further ones, with the goal of building an overlay network topologically closer to the physical network. Note that this deterministic selection of overlay links is meaningful in ad hoc environments, where one could eventually assume a small number of opportunistic (and hence heterogeneous) participants to the Gnutella network. Additionally, the network participants get reshuffled by mobility and by the arrival of new nodes. The same rational would not apply on the Internet, where overlay formation with random walks has been proved more effective than deterministic approaches [CCR04, CRB+03]. Apart from the new peer discovery approach, we left unaltered the rest of the Gnutella protocol (i.e., connection handshaking, link probing and queries), as well as the states identified by the peer degree and the LB and UB bounds.

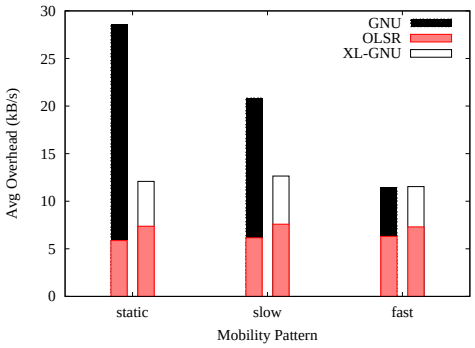
We modified our Gnutella peers to react on three kinds of situation, in corre-

spondence of the updating of the advertisement table, as shown in Algorithm 3. On receiving an advertisement from a *new* peer, the agent always attempts a connection request if it is in state *connecting*. If the current state is *connected*, the agent checks the physical distance of the new peer. In case this is closer than any of the already connected peers, the agent attempts a connection request to it. The same procedure triggers in state *full*, but in this case the peer additionally drops the furthest connection, once the new one becomes active. Similar steps are performed for an advertisement refresh, because with nodes mobility, a peer can receive subsequent advertisements from the same foreign agent at different distances. The third situation is the expiration of an advertisement due to consecutive miss of refreshes. This event is detected internally by the advertisement table, which then directly notifies the peer, causing a connection drop in case the Gnutella protocol didn't detect it using probe Pings.

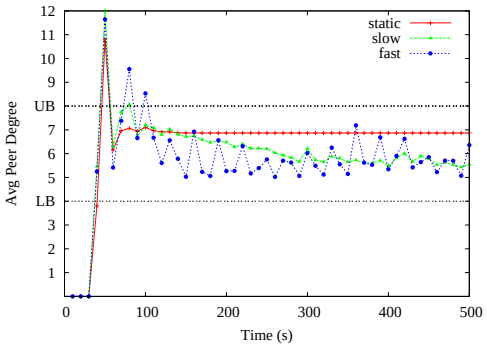
The cross layer approach addresses the issues raised in Section 3.2.4, by eliminating the need of bootstrap servers, reducing and stabilizing the overhead of the protocol, and finally improving the connectivity and the quality of the generated overlays. The next Section proves the validity of these claims, evaluating the cross-layer version of Gnutella (XL-Gnutella) under scenarios similar to those used in Section 3.2.3. XL-Gnutella uses the same settings reported in Table 3.1, replacing discovery Pings with peer advertisement issued every 30 seconds.

3.3.2 Performance evaluation

In order to study the properties of XL-Gnutella and verify our statements, the protocol was subjected to the same set of experiments reported in Section 3.2.3, and compared with the legacy Gnutella running on OLSR. Additionally, we prepared scenarios where the overlays underwent to bursts of increasing number of node churns (i.e., node replacements), in order to study the reaction of the two maintenance protocols. Additionally, we evaluated the *path stretch* produced by the overlays, defined as the ratio of the end-to-end delay (measured in number of hops in the physical network) along the path connecting two peers in the overlay, to that along the direct unicast path in the physical network. The path stretch measures how far (from a topological point of view) the overlay is from the physical network, and characterizes the overhead induced by the former on the latter. By definition, the direct unicast between two nodes in the physical network has a path stretch of unity. The closer the path stretch of a P2P platform to unity, the better. We used this metric to verify the effectiveness of our topology-aware link selection policy. During this set of experiments we were also able to retrieve the overlay *partitioning ratio*, defined as the fraction of unreachable peers pairs with respect to the total number (considered without repetitions). Finally, we performed a simple study on the *query success rate* to see the effects of cross layering on the information discovery capacity of Gnutella overlays. In the following, we present the performance analysis of XL-Gnutella, in each set of experiments. Unless specified, we used the same scenarios



(a) Comparison of the total overhead generated by Gnutella and XL-Gnutella under increasing nodes mobility.



(b) Average XL-Gnutella peer degree under increasing nodes mobility.

Figure 3.10: Study of the benefits of cross-layering on Gnutella overlay management, when mobility comes into play.

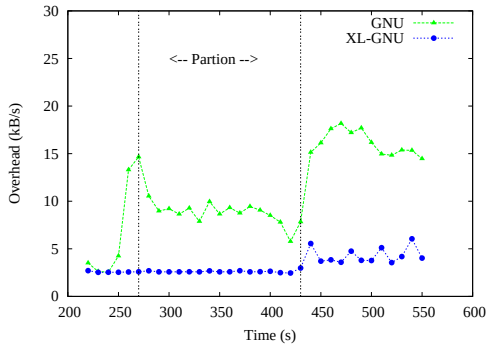
used for the evaluation of Gnutella.

Network size

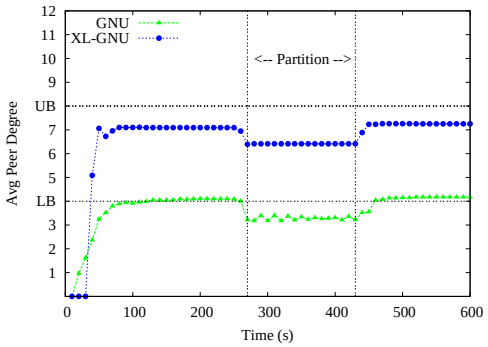
The results obtained by stressing XL-Gnutella with networks of increasing size, suggest that the cross-layer variant achieves better overlay connectivity at significantly lower costs. As shown in Figure 3.9(a), the cost of spreading around peer credentials with OLSR control messages, produces an average increase in size for OLSR messages, which translates to a 25% to 30% added overhead for the routing protocol. However the P2P traffic clearly reduces when compared to the legacy Gnutella, inducing nearly a 60% saving for XL-Gnutella, and hence a 40% global saving. Figure 3.9(b) shows the degree achieved by cross-layer peers, which stabilizes around 7 connections on average (i.e., close to UB), and crosses the lower bound 40 seconds after the beginning of the simulation, independently from the network size. The initial oscillations, which let the average peer connectivity crossing the UB, are due to our selection policy, which first opens the connection with newly discovered peers, and then drops the most disadvantageous.

Nodes mobility

We identified nodes mobility as an important issue for Gnutella (see Section 3.2.3), so we considered important to stress the cross-layer variant to the same mobility scenarios: 40 nodes running on a 1600x700 squared meter area, at increasing speed and with shortening pauses. Figure 3.10(a), shows that the overhead generated by XL-Gnutella is unaffected by nodes mobility, and remains constant around 5 kB/s.



(a) Comparison of the overhead generated by Gnutella and XL-Gnutella across network partitions.



(b) Comparison of effects of network partitioning on the average Gnutella and XL-Gnutella peer degree.

Figure 3.11: Cross-layering helps Gnutella overlay in dealing with network partitions.

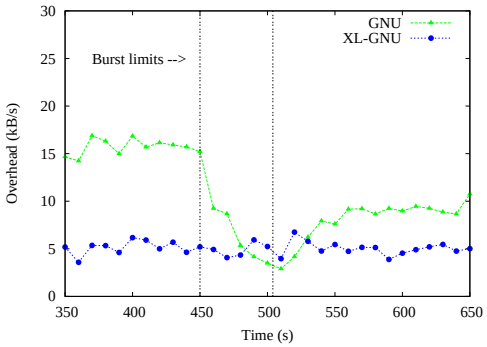
This is confirmed by the behavior of the average peer degree (see Figure 3.10(b)), which falls down from 7 to 5-6 connections per peer as the mobility increases, but remains in the $[LB - UB]$ range. However, fast node mobility induces frenetic oscillations of the average degree.

Network partitioning

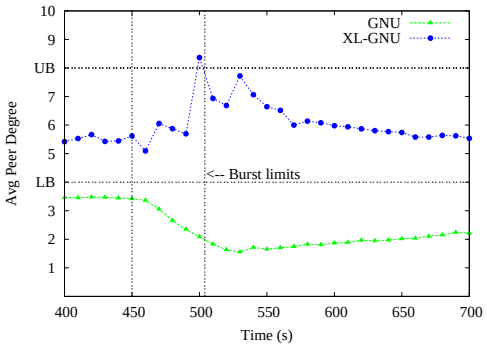
By stressing XL-Gnutella with the network partitioning scenarios, we obtained again interesting results. As show by Figure 3.11(b), also cross-layer peers loose connectivity when the network splits in two halves (see the sharp drop in the plot), but they timely recover the original level when links are re-established at the network layer. Additionally, this happens with no traffic bursts in correspondence of topological reconfigurations, which is not the case for the legacy protocol (see Figure 3.11(a)).

Node churns

In this set of experiments, we stressed the protocols with bursts of node churns scheduled in the middle of the simulation. We created scenarios where 40 peers move around for 900 seconds in an area of 1600 by 700 square meters, using random way-point in "slow" configuration. We let the overlays stabilize, scheduling at time 450 a node replacement every 2 seconds. A replacement is done by picking one node n_1 at random for shut down, and creating a new node n_2 to take the place of n_1. We introduced 30%, 50% and 70% of node replacements, and studied the actual overhead produced and the average peer degree in correspondence of the bursts. For space reasons, we only report the results regarding the 70% burst. As shown in Figure 3.12(a), the amount of traffic generated by XL-Gnutella (i.e., around 5



(a) Comparison of the effects of a 70% node churns burst on the overhead generated by Gnutella and XL-Gnutella.



(b) Comparison of the effect of a 70% node churns burst on the average Gnutella and XL-Gnutella peer degree.

Figure 3.12: Cross-layering improves tolerance to high rates of node churns in Gnutella overlays.

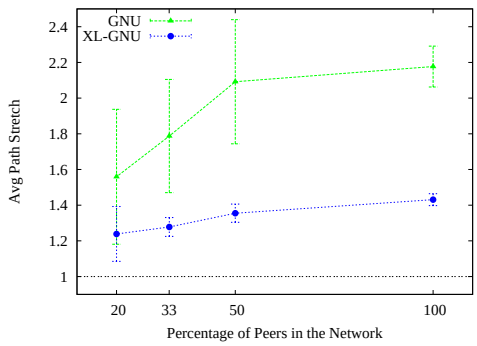
Peer density	Gnutella	XL-Gnutella
20%	8.5%	0%
33.%	25%	0%
50%	29.4%	0%
100%	62%	0.1%

Table 3.2: Comparison of Gnutella and XL-Gnutella average overlay partitioning rate under increasing peer densities.

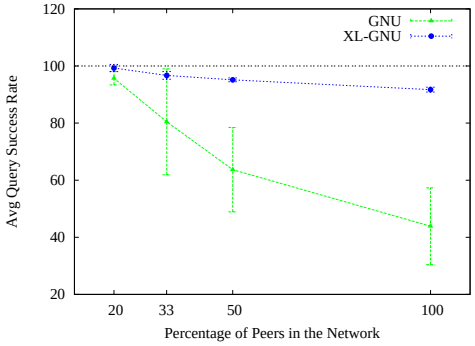
kB/s) is almost unaffected by node replacements. In contrast, Gnutella registers a drop in the amount of generated traffic from 15 kB/s down to 5 kB/s, afterward stabilizing around 10 kB/s. This is due to a resulting partitioning of the overlays, which lasts even after the end of the burst as many among the remaining nodes loose the bootstrap servers. Figure 3.12(b) confirms this, as Gnutella registers during the burst a drop from 3.5 down to 2 on the average peer degree, and never recovers the original value, while cross-layer peers maintain the initial degree (oscillations during the burst are again due to the link selection policy).

Path stretch and query success rate

In the last set of experiments, we wanted to prove the effectiveness of our topology-aware link selection policy. To this end, we studied the path stretch generated by XL-Gnutella and Gnutella overlays. In order to perform this analysis, we turned on periodic query issues with TTL equal to 7, to get multi-hop overlay messages also for XL-Gnutella. We prepared scenarios with 40 static nodes uniformly distributed on a rectangular area, configuring an increasing percentage of them as peers (respectively



(a) Comparison of the average path stretch produced by Gnutella and XL-Gnutella under increasing peer densities.



(b) Comparison of the average query success rate produced by Gnutella and XL-Gnutella under increasing peer densities.

Figure 3.13: Analysis of the path stretch and query success rate generated by Gnutella and XL-Gnutella, as a function of increasing peer densities.

20%, 33%, 50% and 100%). Figure 3.13(a) shows the obtained average path stretch with 95% confidence interval. Not only XL-Gnutella produces overlays significantly closer to the underlying networks when compared to the legacy Gnutella (e.g., respectively 1.35 against 2.1 with a 50% of peers), but the cross-layer protocol exhibits a more stable behavior with smaller variances. Moreover, by looking at the average overlay partitioning rates (see Table 3.2), it was important to notice that with a one-to-one nodes/peers correspondence, 62% of Gnutella peers weren't able to reach each other in the overlay, compared with the 0.1% for XL-Gnutella on the same density. Finally, the numbers shown for overlay partitioning were directly verifiable in a simple experiment on the query success rate. We used the same simulation scenarios as for the path stretch, but additionally distributed shared content on the peers, before making them issue queries on it. This allowed peers to reply with hit messages when reached by query constraints matching their local shared content. We loaded a different file name on each peer, forcing it to issue one query for each file shared by the other peers. The results are reported in Figure 3.13(b), where the Gnutella query success rate clearly degrades accordingly to the overlay partitioning rate: from nearly 95% with 20% peer density (and 8.5% of partitioning), down to 43% with 100% peer density (and 62% partitioning). In the same Figure, the 95% confidence interval highlights once again the stability exhibited by XL-Gnutella in satisfying queries on existing content.

3.4 Related work

In this Section we go through some related work in the area of P2P computing and data sharing in mobile ad hoc networks. Among the many interesting works in the area of large-scale P2P computing, we would like to mention [CRB+03], which proposes to optimize an unstructured platform like Gnutella replacing flooding techniques with random walks, and [YVGM04], which evaluates the performance of GUESS, an iterative and non-forwarding technique to improve Gnutella data discovery. Other works, such as Pastry [RD01a], introduce the concept of structured overlay introducing constraints on the logical location of shared data (or indexes built on top of it). However, all of these works focus on large-scale systems (i.e., thousands of peers), carrying on evaluations under typical Internet workloads and connectivity characteristics.

In the area of mobile ad hoc networks, there have been recent proposals to tailor file-sharing systems for better performance in small-size and mobile scenarios. In [KLW03] the authors present ORION, a file-sharing platform where overlay links and routes are maintained on-demand, with procedures similar to those used by AODV to build and maintain routes at the physical layer. The work in [SGN03], proposes a cross-layer interaction between the network layer and a file-sharing platform at the middleware layer, to force a reactive routing protocol, such as AODV, to build and maintain routes towards other peers and decrease the latency of data look-up. The work in this Chapter spins off the ideas reported in [CGT04], where similar cross-layer interactions are carried out through a vertical interface to define an efficient ring overlay for subject-based routing in ad hoc networks. However, in this Chapter we provide a detailed performance evaluation of an existing and largely used platform like Gnutella, showing its limitations when used in ad hoc environments, and how it could be optimized via cross layering to i) meet ad hoc usability constraints, but at the same time ii) remain fully compatible to the legacy system.

3.5 Conclusions

In this Chapter we promote a cross-layer interaction between overlay peers and routing agents at the network layer. This interaction allows the realization of an efficient peer discovery procedure and link selection policy, determining a low-cost construction of topology-aware overlay networks. We experimented this concept proposing a cross-layer version of Gnutella (XL-Gnutella) as a viable alternative for data sharing in mobile ad hoc environments. From the obtained results, we conclude that XL-Gnutella nicely tolerates typical ad hoc dynamics, like nodes mobility, network partitioning, and node replacements. In each of these conditions, our protocol was able to maintain the required level of overlay connectivity, without generating traffic bursts. Additionally, it presented lower bootstrap latencies and higher rates of query successes, outperforming the legacy version of the protocol. In order to

feasibly realize the interaction between the P2P and the routing agents, we adopted a full cross-layer protocol stack design, where a vertical interface extends the standard layers functionalities, enhancing the local cooperation between protocols. In the future we would like to study similar optimizations for structured overlays, as well as analyze the performance of lookup procedures under typical data distribution models and more realistic mobility patterns.

Algorithm 2 Gnutella peer discovery and Pong caching.

```
AddPong(pongmsg) ⇒ add Pong to cache
PurgePongCache() ⇒ delete stale cache entries
...
HandlePing(pingmsg, connection)
  if (pingmsg.isProbePing()) then
    pong = newPongMsg(local credentials)
    pong.ttl = 1
    SendPong(connection, pong)
  else if (pingmsg.isDiscoveryPing()) then
    PurgePongCache()
    if (PongCache.size() ≥PT) then
      pongs = PongCache.SelectPongs(PT)
      for all pong in pongs do
        pong.ttl = pingmsg.hops + 1
        SendPong(connection, pong)
      end for
    else
      if (PeerTable.state ≠ FULL) then
        pong = newPongMsg(local credentials)
        pong.ttl = pingmsg.hops + 1
        SendPong(connection, pong)
      end if
      if (pingmsg.ttl > 1) then
        pingmsg.ttl − −
        pingmsg.hops + +
        for all c in PeerTable such that c ≠ connection do
          ForwardPing(c, pingmsg)
        end for
      end if
    end if
  end if
...
HandlePong(pongmsg)
  if (pongmsg.ttl + pongmsg.hops ≤ TTL_MAX) then
    AddPong(pongmsg)
    if (pongmsg.ttl + pongmsg.hops > 1) then
      connection = GetMsgOriginator(pongmsg.msgID)
      ForwardPong(connection, pongmsg)
    end if
  end if
```

Algorithm 3 XL-Gnutella peer discovery and link selection.

AdvertiseLocalCredentials() ⇒ periodically generates *SpreadOI* events containing local credentials.
DropFurthestConnOnAccept(ConnHandle) ⇒ selects and drops the furthest connection when ConnHandle becomes active.
UpdateAdvertisementTable(Advertisement) ⇒ updates the advertisement table checking if it is a new advertisement or a refresh.
...
ReceiveRecvOIEvent(E)

```
peerAdv =
newPeerAdv(E.adv, E.credentials, E.phyDistance)
UpdateAdvertisementTable(peerAdv)
HandleAdvertisement(peerAdv)
```

...
HandleAdvertisement($PeerAdv$)

```
if (PeerAdv.isNew()) ∨ (PeerAdv.isRefresh()) then
  if (PeerTable.find(PeerAdv.getCredentials())) then
    return
  end if
  if (PeerTable.state == CONNECTING) then
    SendConnRequest(PeerAdv.getCredentials)
  else
    d1 = PeerAdv.getPhyDistance()
    c = PeerTable.getFurthestConn()
    d2 = c.getPhyDistance()
    if (d1 < d2) then
      cred = PeerAdv.getCredentials()
      cHandle = SendConnRequest(cred)
      if (PeerTable.state == FULL) then
        DropFurthestConnOnAccept(cHandle)
      end if
    end if
  end if
else if (PeerAdv.isExpired()) then
  if (PeerTable.find(PeerAdv.getCredentials())) then
    PeerTable.removeConn(PeerAdv.getCredentials())
  end if
end if
```

Chapter 4

A Cross-Layer Approach for Publish/Subscribe

Abstract

In the context of ubiquitous and pervasive computing, publish/subscribe middleware is gaining momentum due to its loosely coupled communication scheme. In this Chapter, we present Q a publish/subscribe service conceived to operate over mobile ad hoc networks. With Q, the overlay network that routes events from publishers to subscribers dynamically adapts to the changing topology by means of cross-layer interaction. Q also supports content-based filtering of events through mobile code: subscribers can specify in detail the notifications they wish to receive by defining proper filter classes, then binary code of filters is exchanged during runtime by participating nodes.

4.1 Introduction and Background

Publish/subscribe middleware supports the construction of event-based systems, whereby generators publish event notifications to the infrastructure, while consumers subscribe to receive relevant notifications [CRW01]. The interaction model between communicating partners is asynchronous by nature, and is therefore particularly attractive in the context of highly dynamic systems such as mobile ad hoc networks (MANETs). The simplest way to build such infrastructure is through a centralized server, in charge of collecting events and dispatching notifications to interested parties. Clearly this solution, besides its limited scalability, is impractical within MENETs, as these networks are intrinsically peer-to-peer. This suggests to distribute the notification service over the participants, enabling each entity to manage subscriptions and forward events. Unfortunately, most distributed event notification systems assume a fixed network infrastructure, and are not aware of the underlying

network topology, and hence are likely to cause unnecessary traffic when used in MANET environments.

4.1.1 Related Work

Here we summarize some relevant proposals of publish/subscribe middleware for MANETs (for the sake of brevity, we omit those systems that have not been specifically designed for infrastructureless networks).

STEAM [MC02] is an event-based middleware service specifically designed for mobile ad hoc networks. STEAM is based on the idea that communicating entities are likely to interact once they are in close proximity. In other words, events are valid within a certain geographical area surrounding the producer: this limits forwarding of event messages producing positive effects on the usage of communication and computation resources. The system supports three different ways of filtering events: type filtering (on the base of the type, i.e. a name, of events), content filtering (to specify the values of event fields), and proximity filtering (to specify the scope within events are disseminated).

In [VIE03], the authors identify and address the design challenges encountered in the implementation of a Java Message Service (JMS) solution for MANETs. To cope with the lack of readily available routing protocols integrated into the network layer, the system includes an application-level implementation of ODMRP [GPLC00] (a multicast routing protocol for MANETs) as message transportation system.

In [PCM03], the authors propose a technique useful to rearrange efficiently the routes traversed by events in response to changes in the topology of the network of dispatchers. The reconfiguration algorithm assumes that subscriptions are forwarded on an unrooted tree topology that connects all dispatchers.

EMMA [MMH04] is an adaptation of JMS for mobile ad hoc environments where the delivery of messages is based on an epidemic routing protocol. A message that has to be sent is replicated on each host in reach, then if a node that contains a copy of the message, during its movement, gets in touch with other nodes, the message spreads to these nodes as well. Proper mechanisms are used to avoid message duplicates.

4.1.2 Contribution

This Chapter presents the design of Q[1], an infrastructure for publish/subscribe conceived to operate in MANET environments. Q is a type-based system: events are instances of application-defined types, and both publishers and subscribers have to specify the type of events they produce or are interested in.

We briefely recall some properties of type-based publish/subscribe [EG04]. First, it preserves type safety and encapsulation through application-defined event types.

[1] Q is the title of a Luther Blisset's novel. Q is also the name of one of the main characters, a spy.

Second, it easily integrates with object-oriented languages where event types are mapped to classes, and the hierarchical organization of event types is reflected by the inheritance relationship among classes. Third, it does not preclude content-based filtering of events: during subscription, besides specifying the type of events, an application can narrow the set of events of interest by providing a filter, expressed as a set of conditions that must be satisfied by event properties.

Additionally to these features, Q presents two other distinguishing characteristics: reconfiguration through cross-layer interaction and content-based filtering by means of mobile code.

Cross-layer interaction: As in classical protocol stack architectures, ad hoc network activities can be organized in layers. In this model, each layer in the protocol stack is designed and operated separately, with static interfaces independent from network constraints and applications. However, there are some functions which cannot be assigned to a single layer, but are part of an optimization strategy targeting the whole stack. Cross-layering is an innovative form of protocols interaction, placed beside strict-layer interactions, which makes these optimizations possible [CCMT05]. Picking just one example, the routing protocol plays an important role, as it collects network topology information. Sharing topology data, potentially simplifies tasks taking place at other layers (e.g overlay network), avoiding explicit communication to collect it again.

Q interacts cross-layer with routing agents at the network layer, and uses topology information to obtain a self-reconfiguarble overlay that increases communication's efficiency: application-level routes connecting publishers to subscribers closely reflect unicast routes at the network layer.

Content-based filtering through mobile code: Content-based filtering of events can greatly reduce the amount of traffic generated by publish/subscribe systems. This is particularly significant within a MANET scenario where network resources are scarce. In Q, filters are instances of application-defined filter classes. Filter objects, that are issued by event consumers during subscription, are moved towards the publishers of relevant events to maximize the filtering effects. Since filter classes are application-defined, it may happen that a node that receives a filter object is not in possession of the corresponding class. Q is provided with mechanisms that support code mobility, so that filter classes can be downloaded at runtime by participating nodes when needed.

4.2 Q's Overlay Network

Delivery of event notifications requires the cooperation of nodes located between publishers and subscribers, that are involved in the forwarding activity. In the following we will refer to "dispatcher" nodes as those actively involved in event forwarding, while we will say that a node becomes a dispatcher to mean that it starts forwarding event's notifications. For the sake of semplicity, we assume that

all nodes that compose the network are able to participate in the event dispatching activity (i.e. each node runs a Q's instance).

4.2.1 Connecting publishers and subscribers

In a publish/subscribe system communication is anonymous, i.e. publishers and subscribers have no knowledge of each other identity [EFGK03]. A publisher injects an event into the system without specifying the recipients, then the dispatching service is responsible of the delivery of such notification to all interested parties. In the absence of a centralized entity, this poses the problem of establishing routes to forward event notifications from publishers to subscribers. As highlighted in [CRW01], there is the need of broadcasting some information because participants identity is not known. The first solution consists in broadcasting event notifications, which is clearly inefficient since notifications are potentially very frequent, and beside they would be delivered to uninterested nodes. The second alternative broadcasts subscriptions, while notifications get routed through shortest paths. The last alternative, adopted by Q, sees publisher nodes broadcasting event advertisements, and interested nodes replying back with subscriptions. This message exchange sets up the routes that will support event notifications.

In Q, because of the dynamic nature of MANETs, advertisements must be periodically retransmitted, with period T_p, in order to tolerate the loss of messages. Each advertisement contains the publisher's identity (i.e. its IP address and port number) and the type of the events it generates. Upon receiving an advertisement, a node stores in a local table entry i) the type of the event, ii) the ID of the publisher, iii) the ID of the direct (one hop) neighbor from which the advertise message was received. Afterwards, it re-transmits the message to its neighbors. The first two fields are used to keep track of active publishers in the network and the type of produced events, while he third is used to route subscriptions of interested nodes back to the publishers. Entries are discarded after a time equal to $2T_p$.

If subscriber S is interested in the events generated by publisher P, a chain of dispatchers must be established to forward events from P to S. The chain is initiated by S, which uses a subscribe message to register as an event consumer with the neighbor N, from which it received P's advertisement. Afterwards, N registers as an event consumer with the direct neighbor from which it received the advertisement and so on until a path is built from S to P.

As there could be many subscribers interested in the events advertised by P, the system tries to aggregate dispatcher chains whenever possible: a node already acting as a dispatcher for an event type E, does not furtherly forward subscriptions for E. In the scenario shown in Fig. 4.1, the system needs to activate some nodes as dispatchers in order to forward events from P to the three subscribers. In particular, nodes *N1* and *N3* are activated to forward events from P to *S1*. *S3* directly receives events from P, while *N2* and *S3* forward events towards *S2* (i.e. *S3*, besides acting as a sink for events, also operates as a dispatcher).

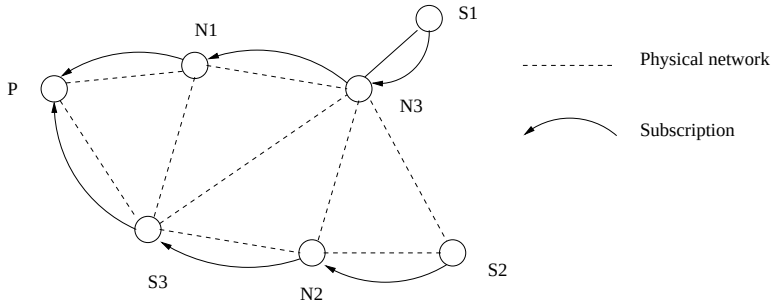


Figure 4.1: Network with three subscribers (*S1*, *S2*, and *S3*) and one publisher (*P*). Arrows show the direction of subscriptions (events flow in the opposite way). One-hop communication links are also shown.

4.2.2 Reconfiguration

As nodes mobility induces topological reconfigurations of the phisical network, Q has to reconfigure the overlay to contain the cost of event dispatching. This is done by means of PING messages, which are periodically generated by each publisher, with period T_{ping}, towards its subscribers. PING messages are forwarded the same way of regular events, but at each hop the dispatcher appends its identifier before sending it to its consumers. This approach makes each dispatcher aware about its level in the forwarding chain (i.e., the publisher is a level 0 dispatcher, which communicates with level 1 dispatchers, and so on).Also, each node becomes aware of the current path from the publisher to itself. This information, together with the content of the routing table, is used to understand if the chain of dispatchers has to be reconfigured or not, as explained in the following.

Cross-layer Interaction

The current implementation of Q is on top of the Dynamic Source Routing (DSR) [JM96] protocol, as it is source-based. This means that the routing table of each node contains the list of nodes that messages have to traverse to reach a given destination.

On receiving the PING message, each node analyzes the list of upstream dispatchers and, for each node in the list, i) calculates the distance at the application level, i.e. the number of intermediate dispatchers, between itself and that node; ii) retrieves from the routing table the distance between itself and that node in terms of physical hops. Then, the node evaluates if a solution better than the current one exists. In that case it performs a partial reconfiguration of the network by unsubscribing from its current upper-level dispatcher and subscribing with another node in the list. In the following we give an intuitive description of the algorithm used to reconfigure the dispatcher chain by means of some examples.

Let us consider the network in Fig. 4.2(a): when *S2* receives the PING message generated by *P*, it sees that *D1* is one hop far away at the network level and three

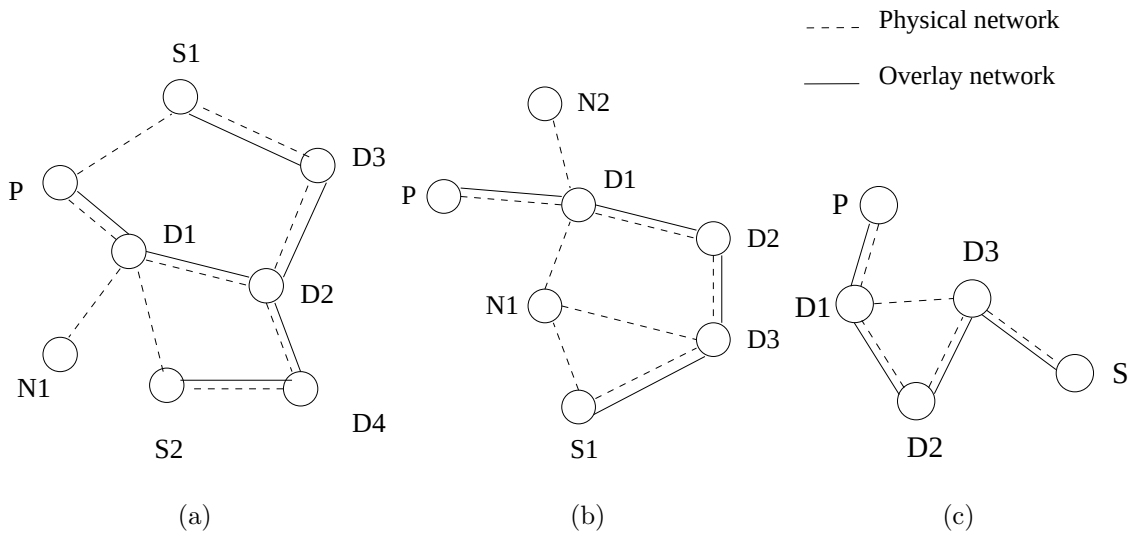


Figure 4.2: Examples of reconfiguration. a) S1 unsubscribes from D3 and becomes a direct consumer of P, while S2 unsubscribes from D4 and becomes a consumer of D1; b) to improve the topology of the overlay network, S1 unsubscribes from D3 and becomes a consumer of N1, which is activated as dispatcher; c) D3, a dispatcher, starts a reconfiguration process by unsubscribing from D2 and subscribing to D1.

hops far away at the level of the overlay network. Therefore, *S2* decides to unsubscribe from *D4* and connects to *D1*. The same way, *S1* discovers that the producer of the events is a direct neighbor. Then, *S1* unsubscribes from *D3* and becomes a direct consumer of *P*. Since *D4*, *D3*, and *D2* are no longer used to route events, they stop serving as dispatchers for that type of events.

The same technique can be applied also when an upstream dispatcher is not directly connected, at the network level, to the subscriber that performs the reconfiguration. In this case, the entry in the routing table indicates how many hops are needed to reach that node and which other nodes are involved. For example, as shown in Fig. 4.2(b), *S1* can understand that *D1* is two hops far away, and that the distance at the application-level is three hops. Therefore, *S1* can improve the topology of the overlay network by unsubscribing from *D3*, and subscribing to *N1*, which in turn must subscribe to *D1*. This operation is achieved through a special message (*Subscribe&Forward*, SF) which is sent by *S1* to *N1*. The SF message contains the list of all nodes on the route between *S* and *P* that must be activated as dispatchers.

As intermediate dispatchers are as a matter of fact "indirect" subscribers, they run the same reconfiguration procedures. For example, as shown in Fig. 4.2(c), on receiving the PING, *D3* decides to unsubscribe from *D2* and attach to *D1*, a direct neighbor.

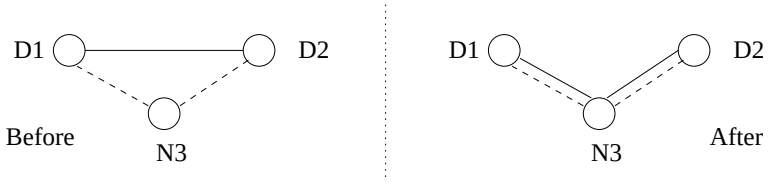


Figure 4.3: Communication between D1 and D2 becomes multi-hop: N3 acts as a router. The overlay network is reconfigured by D2 that unsubscribes from D1 and send a SF message to N3, which becomes a dispatcher.

Communication to a Dispatcher Becomes Multi-Hop

It may happen that two nodes that are adjacent in the overlay network become able to communicate with each other only through the intervention of another node acting as router. For example, as shown in Figure 4.3, dispatchers *D1* and *D2*, that are adjacent in the overlay network, are able to communicate only thanks to node *N3*. Let us suppose that PING messages flow from *D1* to *D2*: when the latter receives the PING, it becomes aware that *D1*, its upstream dispatcher, is reachable by means of node *N3*, which is not active as a dispatcher. Then *D2* sends a SF message to *N3*. *N3* becomes an active dispatcher and subscribes with *D1*. At the same time *D2* unsubscribes from *D1*.

With this strategy, all nodes that route events at the network layer become dispatchers for that type of events (basically, they become routers for that type of events also at the application level). This facilitates the reuse of forwarding chains, as that nodes become possible attach points for other branches.

4.2.3 Node Exit and Failure

Because of the error-prone nature of MANETs, the system must be able to tolerate not only the voluntary exit of a node from the overlay network, but also involuntary disconnections. Voluntary exit of publishers and subscribers is supported by two explicit messages, *stop publishing* and *unsubscribe* respectively. Involuntary disconnections are detected by means of PING messages. If a node in a dispatching chain dies, downstream dispatchers and subscribers will no longer receive PING messages, while the upstream dispatcher will not be able to communicate with it. The system reacts this way: i) downstream dispatchers self-terminate, ii) the upstream dispatcher removes the dead node from the forwarding list and iii) subscribers start a new subscription procedure (this will eventually lead to the creation of new chains).

4.3 Content-Based Filtering through Mobile Code

A prototype of Q has been implemented for the Java 2 SE platform. Besides the mechanisms used to build and maintain the overlay network, the prototype allows

for content-based filtering of events: each subscriber can specify a content filter expressed as a set of conditions on the fields of the type of the event. All events that do not satisfy the conditions specified in the subscription pattern are not delivered to the subscriber.

In many content-based publish/subscribe systems, conditions are expressed by means of specific languages. As pointed out in [EG01], this approach has some drawbacks: the language used to express subscriptions is different from the one used to program the application and syntax errors violating the grammar of the subscription language are detected only at runtime, when conditions are parsed.

With Q, filters are expressed by means of the Java language. More in detail, the programmer can define a new application-specific filter as a class provided with the method *boolean matches(EventType e)* that returns *true* when the event *e* must be delivered to the subscriber, *false* if *e* has to be discarded. The body of the *matches()* method can make use of all public fields and methods of the event object passed as argument.

For example, if a subscriber is interested in receiving events related to the temperature (instances of the *TemperatureEvent* class), but only if the value is greater than a given threshold, the *TemperatureFilter* class could have the following structure (*Filter* is the base class of all filters):

```
public class TemperatureFilter extends Filter {
    private int temp;
    public TemperatureFilter(int t) { temp = t; }
    public boolean matches(EventType e) {
        if (!(e instanceof TemperatureEvent)) return false;
        TemperatureEvent tev = (TemperatureEvent)e;
        if (tev.temp > temp) return true;
        else return false;
    }
}
```

Then, to receive temperature related events when the value is greater than 30 degrees, the subscription operation can be performed as follows:

```
TemperatureFilter tf = new TemperatureFilter(30);
subscribe(tf, ...);
```

4.3.1 Composition of Filters

In many situations, there is no need to build application-specific filters from scratch. More easily, they can be derived from the composition of elementary filters (for example from a library). Q supports the composition of filters by means of specific classes. For example, the *AggregateOrFilter* class allows the programmer to define a new filter as the OR function of an arbitrary number of elementary filters *f1*, *f2*,

..., *fn*. In other words, the *matches()* method of the *AggregateOrFilter* class returns *true* if at least one of the filters *f1*, *f2*, ..., *fn* returns *true*, otherwise it returns *false*. The skeleton of the *AggregateOrFilter* class can be defined as follows:

```
public class AggregateOrFilter extends Filter {
    private ArrayList filterList;
    public AggregateOrFilter() { filterList = new ArrayList(); }
    public boolean matches(EventType e) {
        for (int i=0; i<filterList.size(); i++)
            if (((Filter)filterList.get(i)).matches(e)) return true;
        return false;
    }
    public boolean addFilter(Filter instance){
        return filterList.add(instance);
    }
}
```

The *addFilter()* method is used to add elementary filters. Note that the *AggregateOrFilter* class extends *Filter*, which means that instances of *AggregateOrFilter* can be used in turn as elementaryfilters.

4.3.2 Mobility of Filters

In a distributed implementation of the event notification service, filters can be used to selectively propagate notifications only through those links that are part of a delivery path from publishers to interested subscribers. To ensure maximum traffic reduction, filtering must be performed as close as possible to publishers, so that event notifications that do not match the subscription of any client can be blocked immediately.

Q pushes filter objects as close as possible to publishers. For example, let us suppose that a subscriber S is interested in receiving events of type K that match a filter *f*. Let us also suppose that publisher P produces events of type K and that communication between P and S is guaranteed by a third node, say D. When S starts the subscription process, it subscribes as a direct consumer of D, and the filter object *f* is transferred from S to D. In turn, D subscribes itself as a direct consumer of P and sends to P a copy of the filter object *f*. From now on, all events produced by P that match the filter *f* are forwarded to D, which in turn forwards them to S. Events that do not match filter *f* are dropped by P itself, without generating unnecessary traffic.

Things get slightly more complex if a dispatcher serves multiple subscribers. A dispatcher D that manages subscribers *S1*, *S2*, .., *Sn*, that are interested in events of type K matched by *f1*, *f2*, ..., *fn* respectively, must receive all the event notifications that match at least one of the filters of its subscribers. This is done by creating a filter that is the composition (OR) of the filters *f1*, *f2*, ..., *fn*. Then D forwards

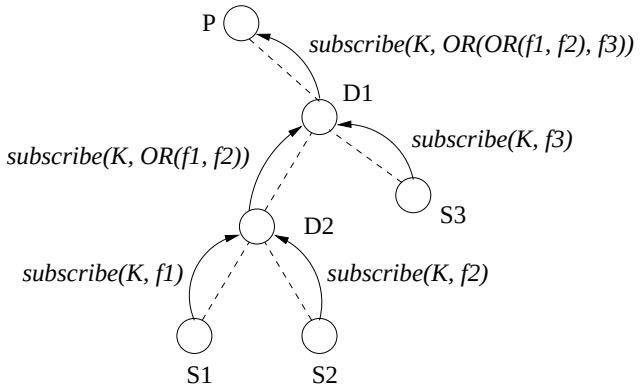


Figure 4.4: Propagation and composition of filters.

the event notifications it receives only to the relevant nodes. An example is shown in Fig. 4.4: *D2* propagates a filter that is equal to the OR of *f1* and *f2*, while *D1* propagates a filter that matches the interests of *D2* and *S3*.

Since new filter classes can be defined by programmers, when a node receives a filter object *f*, instance of class *F*, it may happen that the definition of class *F* is not available in the local context. Q supports the transfer of class definitions from the node that issues the subscription to the node that receives the subscription, where definitions are cached.

4.4 Simulation

The functionalities of the event-notification infrastructure previously described have been validated by implementing the system within the Qualnet [Net03] simulator. Communication between members of the overlay network is based on UDP datagrams. Through simulation we also explored the behavior of the system when varying the following architectural choices:

Event broadcast: Each time a dispatcher *D* has to deliver the same event message to *n* consumers it has to retransmit the same data *n* times. An alternative solution: *D* inserts in the payload of the message the list of consumer addresses, then sends the message only once to the broadcast address, reaching all nodes within its radio range. Every node that receives the message checks if its own address is in the list or not, and decides if the message has to be processed or discarded.

Passive ack for events: If a dispatcher *D1* sends an event message to another dispatcher, *D2*, the latter has to reply with an ack to confirm that the message has been correctly received. Passive acknowledgement can be an alternative solution: if em D2 is not the last dispatcher in the path, it has to retransmit the message to its downstream dispatchers; therefore D1, by listening to messages transmitted by *D2*, can understand that an event message has been correctly received by *D2* if retransmitted.

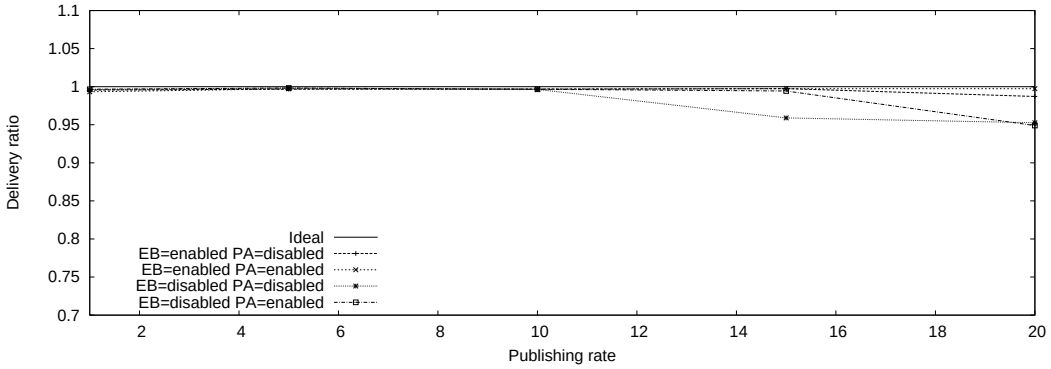


Figure 4.5: Delivery ratio of event notifications with 1 publisher and 1 subscriber

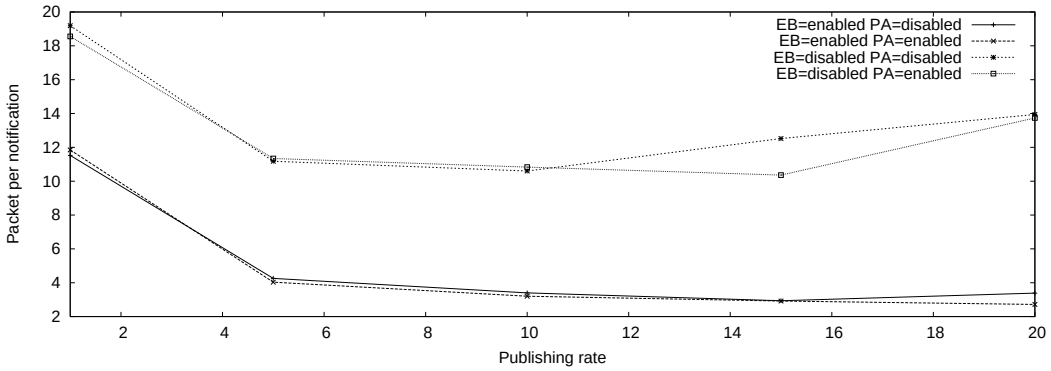


Figure 4.6: Packets per notification with 1 publisher and 1 subscriber

Simulations have been carried out according to a scenario where 50 nodes, equipped with 802.11 radio interfaces (2Mbit per sec.), move in a $1000mX1000m$ area. The model for mobility is random waypoint with pause time equal to $30s$ and max speed equal to $10m/s$. Size of event notifications is $512Bytes$. The capability of filtering events on the base of their content is not considered. The performance indexes are the ratio of successfully delivered notifications and the number of packets generated per delivered notification (the latter provides a measure of the overhead of the protocol). Both indexes are evaluated as functions of the rate of published events.

Figures 4.5 and 4.6 show the performance of the system with one publisher and one subscriber, while Figures 4.7 and 4.8 are related to a scenario with one publisher and five subscribers. The four curves correspond to the alternative solutions mentioned above: the labels indicate if events are sent as broadcast (EB) or not, and if passive acknowledgement (PA) is enabled or not.

In the considered scenarios, the solution that provides the best performance is the one that adopts both event broadcast and passive acknowledgement. This is more evident in the scenario with five subscribers especially at high publishing

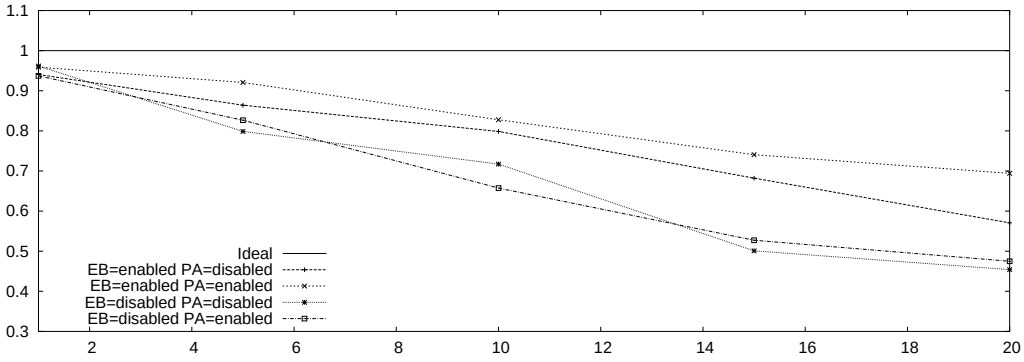


Figure 4.7: Delivery ratio of event notifications with 1 publisher and 5 subscribers

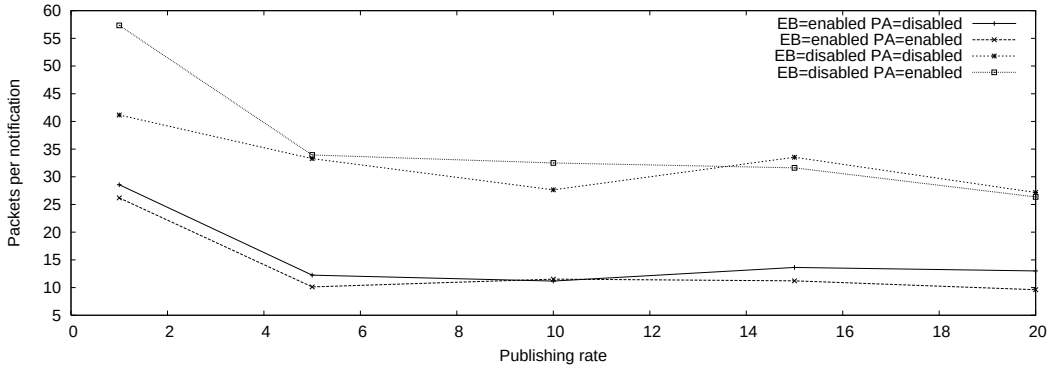


Figure 4.8: Packets per notification with 1 publisher and 5 subscribers

rates, i.e. when the network starts being congested. In particular, when events are sent as broadcast the number of packets per delivered notification is greatly reduced. Instead, the effects of the passive acknowledgement technique, even if positive, are less evident. Probably this is due to the fact that, in the considered scenarios, dispatching chains are rarely longer than 2-3 hops (the radio range is approximately $300m$).

4.5 Conclusion

Publish/subscribe middleware is particularly attractive in the mobile computing domain, as it provides a loosely coupled communication scheme. This is particularly useful with mobile ad hoc networks where the quality of communication and connectivity change dramatically in a short time scale.

In this Chapter we presented Q, an infrastructure for publish/subscribe that has been specifically designed for mobile ad hoc environments. In Q, the overlay network used to route event notifications dynamically adapts to the changing topology by means of information extracted from the routing layer. Currently, Q relies on

the abstract representation of the network provided by a reactive and source-based protocol. However, we believe that with few changes it could operate also on top of a proactive routing protocol, which could provide a richer view of the network since it does not limits its knowledge to used routes.

The primary goal of the simulative study that we presented was to demonstrate the feasibility of an event notification service based on a cross-layer approach. However, the results obtained also give insightful suggestions about the efficiency of different implementation techniques, and quantify the benefits achievable through event broadcasting and passive acknowledgement.

Besides its reconfiguration capabilities, Q also supports content-based filtering of events. To support seamless integration of filtering with object-oriented languages, in Q filters are instances of application-specific classes. At runtime, filter objects and code are exchanged by participating nodes, maximizing the benefits of filtering and the flexibility of the system.

Future work will concern the evaluation of the performance of Q in comparison with other publish/subscribe systems.

Chapter 5

Structured P2P Computing

Abstract

This Chapter deals with structured P2P computing and its usage in mobile ad hoc environments. The structured paradigm provides an effective framework to realize decentralized and scalable applications, where *availability* and *performance* are a key constraints. Platforms like Pastry, Chord, CAN etc., have been designed to support a large set of users with bounded overheads and latencies, providing a fairly balanced workload distribution, and guaranteeing the retrieval of existing data. The Chapter initially reports the results of a performance evaluation of Pastry in MANET settings. This study highlights that structured overlay algorithms exhibit low tolerance to dynamics such as nodes mobility, or topology reconfiguration, and suggests a re-design strategy structured overlay management that is based on cross-layering and aims at delivering good performances in emerging mobile computing scenarios.

5.1 Introduction

To complete our perspective on P2P computing in MANET environments, this Chapter focuses the attention on *structured* P2P computing. In particular we analyze the performance of a platform called Pastry [RD01a]. Pastry offers a key-based message routing interface, establishing a ring overlay network among the participating peers. The term *key-based*, refers to the ability of routing data items (or messages) in a distributed virtual space using data subjects (e.g., filenames, object identifiers, group identifiers etc.). Using this technique, application programmers can rely on a simple abstraction to set up distributed shared spaces, provided that application components responsible to produce and consume data, agree on an set of rules that associate subjects (or the keys) to data items in a unique way. Interesting services and applications [FRE] have been built over Pastry, showing that this interface is general enough to build different and more complex abstractions, and therefore demonstrates potential to be exploited also in MANET environments.

Platforms like Pastry provide a way to organize resilient and fully decentralized systems, where the workload is fairly distributed over the set of participants. These are fundamental characteristics also in ad hoc contexts. Resiliency is mainly needed because of nodes mobility and topology reconfigurations, as well as the fact that users may turn services (and devices) on and off at their pleasure. Fair workload sharing would be desirable to avoid central points of failure, and situations with congested nodes. Moreover, the message routing properties of this paradigm guarantees upper bounds on the latencies, and on the number of messages generated to lookup data items. However, Pastry has been designed to support data sharing on Internet scenarios, therefore assuming large number of users, and resources typical of a network with infrastructure. Key-based routing is efficient as each node builds knowledge about a significant portion of the rest of the overlay network, and maintains it coherent with changes. Unfortunately, as shown in Section 5.3.4, the procedures and protocols that guarantee this behavior are expensive in terms of network resources, and even if MANET won't present Internet-scale scenarios, a straightforward implementation of Pastry would have to cope with situations like highly variable network topologies, high rates of node churning and network partitioning.

This chapter, after giving background information on key-based message routing (Section 5.2), reports the results of a performance evaluation of Pastry in MANET environments (Section 5.3.4), and concludes by giving guidelines (Section 5.4) about the design of a platform for efficient key-based massage routing for MANETs, which exploits the cross-layer architecture presented in Chapter 2.

5.2 Background on key-based message routing

In recent years, structured P2P overlay networks have gained popularity as they proved to be good candidates for building resilient, and large-scale systems for distributed computing. Several existing platforms, like CAN [RFH+01], Chord [SMLN+03], Pastry [RD01a], and Tapestry [ZKJ01], just to mention some, have been implemented to provide structured overlay networking. The main idea is to build the overlay with a certain *structure*, which means that links between peers are set up following specific criteria, instead of using random policies like in the case of Gnutella [KM02]. This approach allows peers to locate objects by exchanging a number of messages that scales logarithmically with size of the overlay (i.e., $O(logN)$ messages where N is the number of participants). Structured overlays can be used to construct services such as distributed hash tables (DHT), scalable group multicast/anycast (CAST), and decentralized object location (DOLR). In turn, all these services promise to support novel classes of highly scalable, resilient, distributed applications, including cooperative archival storage, or cooperative content distribution and messaging.

As analyzed in [DZD+03], all the aforementioned platforms offer a common de-

nominator of services based on the concept of key-based message routing (KBR). In other words, given a message M to be routed across the overlay, the final destination, as well as the intermediate peers visited by the message, are chosen accordingly to a key K associated to M, instead of using network information such as the IP address of a final recipient. The association between K and M is done by the peer that creates the message. There are other application-visible concepts common to all structured overlay protocols. Generally, a node represents an instance of a participant in the overlay, and it is identified through a unique logical address chosen uniformly at random from a large space. Application-specific objects are also assigned unique identifiers, called *keys*, selected from the same space. Tapestry, Pastry, and Chord, use a circular identifier space of n-bit integers modulo 2^n ($n = 160$ for Chord and Tapestry, and $n = 128$ for Pastry), while CAN uses a d-dimensional Cartesian identifier space, with 128-bit identifiers that specify points in the space. Each key is dynamically mapped by the overlay to a unique live node, called the *key's root*, following a principle of proximity in the logical space. To deliver messages efficiently to the root, each node maintains a routing table consisting of the logical identifiers and IP addresses of some other nodes, which have been selected to establish links in the overlay. Messages are forwarded across these links to nodes whose logical identifiers are progressively closer to the message's key. Each system defines its policy for mapping keys to nodes. In Chord, keys are mapped to the live node with the closest identifier clockwise from the key. In Pastry, keys are mapped to the live node with the closest identifier (i.e., either clockwise or counterclockwise). Tapestry maps a key to the live node whose id has the longest prefix match, where the node with the next higher id value is chosen for each digit that cannot be matched exactly. CAN follows a different approach, where neighboring nodes in the logical space agree on a partitioning of the space surrounding their ids, and keys are mapped to the node responsible for the sub-space containing the key.

In the following Section, we report details about Pastry's algorithms and data structures, and then we analyze the results of an evaluation of the platform in MANET settings.

5.3 A case-study: evaluation of Pastry

The overlay network defined by Pastry [RD01a] implements a large circular space of $2^{128} - 1$ logical identifiers, and therefore is also called a ring overlay. Each Pastry peer chooses a 128-bit identifier (nodeId), which represents a logical position in the ring. The nodeId is randomly assigned at join time, usually by hashing a physical identifier. The hashing process uniformly distributes inputs in the circular space, minimizing the chances of mapping two different inputs on the same nodeId, and scattering nodes with closer nodeIds far apart in the ring.

5.3.1 Key-based message routing

The fundamental service offered by Pastry allows peers to exchange messages by keys. The idea is to associate a logical key to an application layer message, and route it hop by hop in the ring, until it lands on the peer with the closest nodeId to the message's key. This final peer represents the *root* for the message, and is therefore responsible to process the enclosed content at the application layer. The rules that associate keys to messages and keys usually apply the same hashing process used for mapping nodes to logical addresses, guaranteeing equivalent distribution properties. To give an example, consider a file sharing application where each file is represented by its name. A typical interaction with Pastry would be to create two types of messages, one to *advertise* in the ring the sharing of a file associated to a given name, and another to *lookup* the node (or the nodes) sharing a file with a given name. In this scenario advertise and lookup messages get routed through the same logical identifiers, and the corresponding root peers associate them at application layer.

The key-based routing (KBR) policy used by Pastry is based on a numerical proximity metric between message keys and nodeIds. From an algorithmic standpoint, consider logical identifiers to be represented as a sequence of digits with base 2^b, where the parameter b is defined a priori. At each step of a message routing procedure, a Pastry peer P forwards the message with key K to a peer Q whose nodeId shares with K a prefix that is at least one digit (or b bits) longer than the prefix shared with P. If no such node is known, P tries to forward the message to a peer L that has the same common prefix with K, but is numerically closer to K with respect to P (this can be easily identified by looking at the digit after the common prefix). With this process, the expected maximum number of hops in the overlay between source and destination is equal to $\log_{2^b} N$ in an overlay of N peers.

5.3.2 State representation

To support the above message routing procedure, each peer maintains information about foreign portions of ring overlay, using the following data structures:

Routing table. This structure is organized into $\log_{2^b} N$ rows with $2^b - 1$ entries each (one for each possible digit). Each entry at row n of the routing table refers to a peer whose nodeId shares with the local nodeId the first n digits, but whose $n+1^{th}$ digit differs. If there are no nodeIds with this characteristic, the entry is left empty. In practice, a destination node is chosen, between those known by the local node, based on the proximity of its logical identifier to the value of the key. This choice provides good locality properties, but only in the logical space. In fact nodes that are logical neighbors, have a high probability to be physically distant in the underlying network. In addition, the choice of the parameter b determines a trade-off between the size of this data structure and the maximum number of hops in key-based routing procedures, that is

expected to be in the order of $\log_{2^b} N$, as confirmed by simulations results [RD01a].

Neighborhood set. This structure represents the set of peers that are physically close to the local peer. The neighborhood set is not normally used in routing messages, but could be useful for maintaining physical locality properties among peers.

Leaf set. This structure represents the set of peers with the closest logical identifiers. The leaf set is centered on the local peer P, with half of the identifiers larger than P, and the other half smaller than P. The leaf set represents the "perfect" knowledge that each peer has of its logical contour.

In routing a given message, the peer first checks if the related key falls within the range of nodeIds covered by its leaf set. If so, the message is directly forwarded to the destination peer, namely the leaf set entry whose nodeId is logically closest to the message key. If the key is not covered by the leaf set, then the routing table is used, and the message is forwarded to a peer that shares a common prefix *at least* one digit longer than the local nodeId. Sometimes, it is possible that the appropriate entry in the routing table is empty, or that the associated peer is currently disconnected from the network, but the overlay is still not updated; in this case the message is forwarded to a peer (if any exists) that shares the same prefix as the local node, but is numerically closer to the key. In general, each entry maintains a correspondence between a logical identifiers and the corresponding network credentials (IP address and port number), to allow the establishment of direct connections driven by application needs.

5.3.3 State management

The main procedures used by Pastry to manage the ring overlay (i.e., the above data structures), consists of join and leave operations. First of all, when a new node, say X, decides to join the overlay, it needs to initialize internal data structures, and inform other nodes of its presence. The assumption is that the new node knows at least one of its physical neighbors, say A, which already takes part to the overlay. In Internet settings, the "bootstrap" peer is typically obtained through outside channels (e.g., the user, or the network administrator). Node X then requests A to route a special *join* message with the key equal to X. As usual, Pastry routes the join message to a root peer Z whose id is the closest to X, passing through some intermediate nodes. In response to the join request, peers A, Z, and all the intermediate peers, send part of their internal structures to X. At this point, X processes the received information, and initializes its own structures using the following rules:

- The neighborhood set is initialized with the contents of that of node A, since it is a physical neighbor of X.

- The leaf set is initialized with that of node Z, which has the closest existing nodeId to X.

- The i^{th} row of the routing table is initialized with the corresponding row of the routing table of the i^{th} intermediate peer (B_i), encountered in the logical path from A to Z (as it shares a prefix of length i with X).

At the end, X informs A, Z, the intermediate peers, and all the peers listed in the resulting tables about its arrival, transmitting a copy of its resulting state. This gives a chance to the rest of the peers to update their tables as well.

An important feature of Pastry is that it manages also peer departures. Pastry assumes that peers may fail or depart without warning the rest of the community. In particular, a peer is considered failed when its logical neighbors can no longer communicate with it. To this aim, nodes in the leaf set are periodically probed with ping messages. Leaf entries that do not reply to a series of probe pings are considered failed, and get replaced by new entries in the leaf set. A similar probing mechanism is used to maintain a consistent neighbor set. Instead, a peer discovers failed entries in the routing table, only when it attempts to forward it an application message. This event does not normally delay message routing, as another destination peer could be selected. In any case, to replace the failed entry the peer gathers a suitable nodeId by asking the peers in the same row of the routing table. If none of them has a pointer to a live peer with the appropriate prefix, the process is repeated with the next row in the routing table.

The maintenance procedures explained above, highlight the complexity of the algorithms associated to ring management in a structured overlay network. While this complexity is acceptable in Internet scenarios, its feasibility should be validated in a MANET's context, to understand and characterize its behavior in wireless multi-hop scenarios, with device mobility.

5.3.4 Performance evaluation

To better understand the capacity and limitations of Pastry in mobile ad hoc environments, we implemented the protocol inside the framework described in Chapter 2, using the Network Simulator 2 [NS2] (version 2.27), and performing a set of simulations to put Pastry through typical Wi-Fi based ad hoc conditions. At first, we wanted to study the protocol behavior in static scenarios, but with an increasing number of nodes and peers in the network. To this end, we disposed at random an increasing number of wireless nodes in a grid fashion over the simulation area. The size of the simulation area varied along with the network size, in order to keep constant network densities. Additionally, we introduced different peer densities: either only 50% of of the wireless hosts were also Pastry peers, while the rest simply performed networking activity, or the totality of the wireless devices were also Pastry peers (i.e., 100% density). In a second set of experiments, we fixed the network size

to 30 nodes, and created some scenarios with Random Waypoint mobility [Bet02]. In particular, we varied the speed and pause parameters used by the Random Waypoint algorithm, creating a *slow* mobility scenario, where nodes move at most at 5m/s, and a *fast* mobility scenario where the maximum speed is 15m/s.

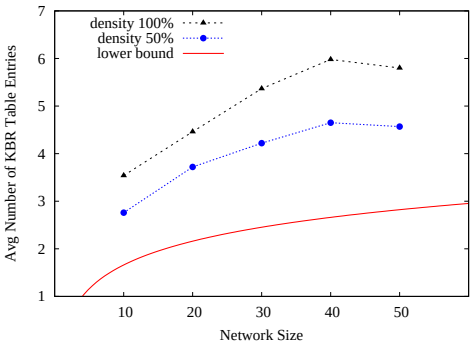
Using the aforementioned parameters, we evaluated Pastry considering the following metrics:

- the capacity of building the ring overlay, verified as the average number of entries in Pastry KBR tables, which should be *at least* the logarithm of the overlay size, if the tables are built correctly [RD01a];
- the average number of hops travelled by each message routed by key in the overlay, to reach its root peer; again this number should be *at most* the logarithm of the overlay size;
- the rate of unsuccessful key-based message routing tentatives, defined as the fraction of the failed tentatives over the total.

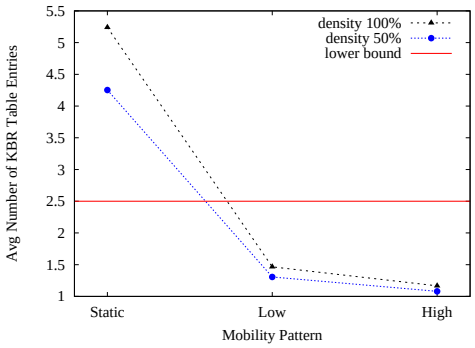
For the last two metrics we engineered a simple data distribution model, in which each Pastry peer was configured to be root for only one key. In this scenario, we programmed each peer to execute a route tentative toward each key K, considering it successful if it eventually lands at the corresponding root peer. The results have been obtained by averaging 10 independent simulation runs of 900 seconds, and are shown in Figure 5.1.

Figure 5.1(a) shows the average number of KBR table entries that Pastry peers registered in static scenarios with increasing network size. The plot demonstrates the validity of the protocol implementation, as peers were able to populate KBR tables above the lower bound (i.e., the logarithm of the overlay size), and build a consistent state of the ring overlay. This is confirmed by the graph reported in Figure 5.1(c), where it is clearly visible that each successful KBR tentative employed (less than) a logarithmic number hops in the overlay. However, increasing network sizes do have a negative impact on the success rate of KBR tentatives, as show in Figure 5.1(e). In fact, the rate of failed KBR tentatives quickly grows with the network size, apparently following a polynomial pattern. This was mainly due to an increased network congestion, which negatively influenced packet loss during the simulations.

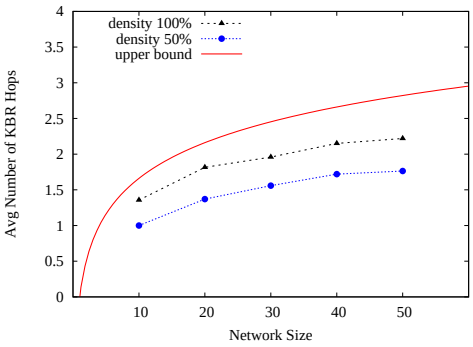
The result associated to the experiments with mobility, reported a less promising picture of the usability of Pastry in MANET environments. Figure 5.1(b) clearly shows that as nodes start to move around, the protocol looses the ability to set up the ring overlay, as the average number of entries in the peers tables are less than expected for an overlay of 30 peers (i.e., around 2.5 which is $\log_{2^2} 30$). This causes a severe ring partitioning, where peers are not able to set up and maintain a unique ring, but only form little rings composed by at most two or three peers. Figures 5.1(d) and 5.1(f) confirm this thesis, showing an average number of hops that falls



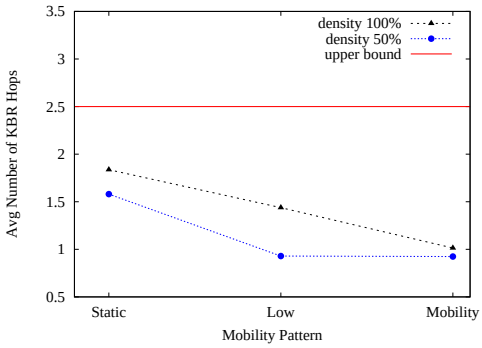
(a) Average number of Route Table entries with increasing network sizes.



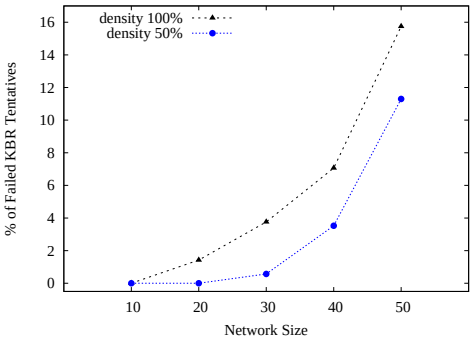
(b) Average number of Route Table entries with increasing mobility.



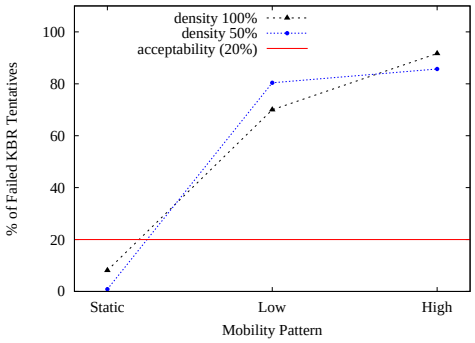
(c) Average number of KBR hops with increasing network sizes.



(d) Average number of KBR hops with increasing mobility.



(e) Percentage of failed KBR tentatives with increasing network sizes.



(f) Percentage of failed KBR tentatives with increasing mobility.

Figure 5.1: Evaluation of Pastry in MANET environments.

down to 1.5 as mobility comes into play, as well as a significant increase of failed KBR tentatives.

As applications based on structured platforms focus on availability, KBR tentatives should maintain very low failure rates, as pointed out in [RD01a], where the acceptability threshold is set around 5% of failed KBR tentatives. Even considering a higher acceptability threshold (e.g., around 20%), the result just discussed do not highlight good properties for Pastry algorithms in mobile ad hoc scenarios. Therefore, in the following Section we give suggestions on how a similar service could be re-engineered using cross-layer interactions with layer-3 protocols, in order to simplify overlay construction as well as KBR procedures.

5.4 Cross-layering and key-based message routing

The simulative evaluation of a Pastry in MANET environments provides us with useful insights on how to improve the performance of structured platforms in presence of MANET dynamics. Pastry implements a basic service that routes application layer messages across the links established by peers to cover a distributed ring of logical identifiers. On top of Pastry overlays, it is easy to set up and deploy key-based distributed applications, which follow a P2P model of computation, however, the efficiency and the usability of such applications strongly depends on the capacity of building consistent ring overlays. As shown during the simulations, this is not the case in presence of device mobility, even when the size of the network (and therefore of the overlay) are several orders of magnitude smaller than the targets of Pastry. As mobility comes into play, peers are not anymore able to consistently fill up their routing tables, causing severe overlay partitioning and consequent high failure rates during lookup procedures, quickly degrading the service performance and usability.

In this Section, we briefly discuss the challenges behind an efficient implementation of KBR abstractions in mobile ad hoc networks. Details are provided using Pastry as a reference case study, but similar concepts can be applied to other KBR platforms.

The logarithmic costs guaranteed by KBR message delivery, are strongly associated to the correctness of the structures used to route application packets through the overlay [RD01a]. These structures are independently maintained in a distributed and autonomous fashion, and represent the knowledge that each peer has about the rest of the overlay. Typically, Internet nodes have little knowledge about the network: apart from routers, which do not participate to overlay rings, normal nodes only know their subnet gateway to make the first hop of locally generated packets. This implies that in order to take part to a ring overlay, peers have to play *discovery* and *maintenance* protocols, which fills out routing structures and adapts them to the network dynamics. In [RD01a] the authors performed a cost analysis for a Pastry overlay of 5000 peers, where a 10% failure rate affects randomly selected nodes over a long period of time. As a result, each node made an average of 57 remote

procedure calls just to repair the tables.

Ad hoc environments will put ring overlays through tougher conditions across short time intervals. For example, nodes mobility or users turning devices on and off, will cause peers to be intermittent, and entire rings to frequently split and rejoin. Even if ad hoc networks are smaller than thousand of nodes, their dynamics could easily degrade the performances of KBR platforms, as highlighted in Section 5.3.4. To overcome this limitation, the idea is to exploit the knowledge about the network topology coming from network layers. Although ad hoc nodes have resource limitations respect to their Internet counterparts, they have significant advantages from a routing standpoint, as they actively participate to routing protocols. If such a knowledge is made available in the stack, for example through the cross-layer architecture described in Chapter 2, then other protocols could benefit from it and optimize their functioning. In the case of Pastry, accessing the routing tables through the XL-Interface translates to having local references on the nodes forming the network, which are a superset of the reachable peers. This suggests a way to save most of the communication associated to peer discovery and table maintenance. In the following, we show how a Pastry system can benefit from cross-layer interactions with a link-state routing protocol such as OLSR [CJ03] at the network layer.

Discovering overlay peers in the physical network

Link-state protocols continuously update the network topology representation, usually by flooding neighbor set information across the network. The XL-Interface makes this representation available to other protocols in the local stack. In this framework, a Pastry peer willing to join an existing ring R, would need to understand which nodes, among those visible through the XL-Interface, are currently part of R. This can be done similarly to what described in Chapter 3, by spreading service information together with link-state updates. The signaling between Pastry and routing processes can be done using cross-layer events for *spreading* and *receiving* optional information.

State management and message routing

By introducing a cross-layer interaction with the network layer, a Pastry peer P sees at each time the subset of the ring R that is visible in the routing tables. The logical addresses of these peers could be easily made available through optional information spread along with routing control messages. The neighborhood set M could be constituted by those peers that are close to P, according to a given proximity metric, as for example the physical distance between the peers. Similar considerations apply for the leaf set and the routing table. In short, the three structures could be merged in a single table T that lists all the peers made visible through cross layering, relaxing the entire overlay concept. Following this approach, key-based routing can be performed without applying prefix-based procedures. In

fact, if P has to route a message with key k, it could directly select the destination peer with the closest logic address to k, and directly forward the message to it with a single unicast.

This strategy has of course advantages and disadvantages. On one hand, it frees P2P agents from the overhead of managing and maintaining overlays. In fact, by exchanging advertisements along with routing control packets, peers have the possibility to discover the available P2P community in parallel with the discovery of the network topology. On the other hand, proactive protocols like OLSR do not guarantee a perfect and immediate view of the physical network. A certain latency has to be considered in order to allow a deep propagation of topology changes. This "hazy" view of the physical network, and therefore of the ring participants, has negative effects on the availability requirements of the structured paradigm. In fact, in scenarios with significant mobility or churn rates, nodes would probably have fairly different views of the network topology, with negative impacts on the convergence of key-based message routing. For example, far apart peers would temporarily identify different roots for a given key, causing messages addressed to the same key, to land on different nodes. A solution to this problem could be to associate temporal information to application messages. For example, a peer D currently selected as root destination for a message m, could cache it long enough to allow a newly entered peer to join the ring, and receive cached messages. In this case, a new peer N would receives m from D's cache if its logical address is closer to message key.

Finally, inferring ring tables from network routing tables brings significant advantages also in case of network (and therefore ring) partitioning and merging. Current implementations of Pastry [FRE] do not tolerate situations in which a ring gets partitioned for a significant amount of time, as well as situations in which two separate rings supporting the same application should merge in a single structure, because the underlying network provide connectivity. In contrast, the usage of the cross-layer approach delivers full tolerance to those situations, which could easily happen in MANET scenarios.

The basic of the above concepts has been implemented in Crossroad [CDT06], a structured platform targeted at mobile ad hoc environments. This platform exports a Pastry-like semantic, routing keys to the live node with the closest identifier in the ring. Crossroad exploits a cross-layer interaction with an OLSR routing agent at the network layer, which enables fast and inexpensive peer discovery, as well as the avoidance of prefix-based message routing, delivering application messages in a single unicast to their roots.

5.5 Conclusions

In this Chapter, we dealt with the problem of providing efficient structured P2P computing in mobile ad hoc networks. After an initial case-study evaluation of

Pastry in scenarios with device mobility and a simple data distribution model, we identified peer discovery and overlay maintenance as key issues for a correct and smooth functioning of platforms adhering to this paradigm.

The solution to these problems comes again in the form of cross-layering. The usage of the XL-Interface allows information sharing and event handling among local protocol agents. In this framework, the topological information collected by a proactive link-state routing protocol, augmented with information related to the ring overlay, reduces the costs associated with overlay structure management, and potentially simplifies key-based message routing.

The proposed solution, while customized to Pastry, is applicable to other structured platforms, and is part of our efforts in providing a global optimization of ad hoc networks functioning.

In the next chapter, we deal with coordination models and tuple spaces, showing how they could be implemented on top of key-based message routing platforms, in order to merge the rich content-based API of coordination languages, with the efficiency of structured platforms.

Chapter 6

Laying Tuple Spaces over Structured P2P Platforms

Abstract

Lime (Linda in a Mobile Environment) is a middleware platform for mobile computing, based on coordination abstractions inspired by those used in Linda. This platform supports the development of distributed and peer-to-peer applications for mobile environments, providing the abstraction of transiently shared tuple spaces. While Lime exhibits maturity in the set of exported primitives and coordination algorithms, it relies on a basic implementation of messaging between peers, which is built on top of multicast sockets, and does not guarantee scalability. This Chapter proposes a redesign of Lime communication concerns, based on the key-based message routing service provided by recent platforms for structured P2P computing. The new strategy implements the semantic defined for Lime's primitives, and makes it usable and efficient in both MANET and Internet-scale scenarios.

6.1 Introduction

Mobile and distributed computing is emerging as a disruptive new trend that challenges fundamental assumptions on networking and software engineering practices. An increasing number of real scenarios foster the realization of applications that exploit and support device mobility, energized by advances in wireless communication, device miniaturization, and new software design techniques.

Coordination is a computing style that emphasizes a high degree of decoupling among application components, along with a clean computational model and an abstract approach to communication. This form of computing was initially proposed with Linda [Gel85], and bases on the concept of information sharing through a globally accessible, persistent, and content-addressable data structure, implemented

as a tuple space. Software agents interacts among them using a simple interface that allows for insertion, removal and duplication of tuples from the space. This persistent data structure, usually implemented in a centralized way [TSp][Jav], provides *temporal* and *spatial* decoupling. Temporal decoupling is achieved because the interacting parties should not be necessarily present at same time when communication takes place, while spatial decoupling is achieved by eliminating the need of knowing each other's identities during agent communication. These features are clearly appealing in the context of mobile computing, specially in the extreme form of MANETs. However, when shifting to mobile contexts, the assumption of having a reliable connection to a persistent data structure is a strong limitation. Moreover, in the case of mobile ad hoc networks, the global tuple space could not be centralized on a single unit. Therefore, the transition to mobility requires the engineering of data distribution schemes that accommodate device mobility and disconnection.

Lime (Linda In a Mobile Environment) [PMR99][MPR01][LIM] is an interesting response to the challenge of providing a coordination middleware in mobile settings. This platform offers a peer-to-peer computational model, where each peer owns a private and local tuple space. When a wired or wireless link support physical connectivity between two peers, they can share the tuples contained in the local spaces, as they were part of a single *federated* space. In this way, the set of available tuples in a group of peers changes over time, influenced by mobility and connectivity: when hosts come in communication range the set of shared tuples expands, and when they move apart it contracts. Tuple space sharing is done in a transparent way from the application standpoint. Lime applications access the tuples using a Linda-like set of operations, enlarged with constructs that facilitate the reaction to changes in the federated space content.

The current Lime implementation [LIM] realizes tuple space federation by playing coordination protocols on top of both TCP and UDP data transfer. In particular, group messaging is done using UDP datagrams and IP multicasting. A message not addressed to a specific peer is sent in multicast, and received by the entire Lime community. Although this solution might be appropriate for small demonstrative test-beds and did prove the semantic validity of Lime coordination protocols, it is not usable with current ad hoc routing protocols implementations. Moreover, this approach does not scale to large peer communities in networks with infrastructure (i.e., the Internet). For this reason, after a recent re-engineering effort [VIC], Lime splits coordination and communication in two separate layers, and get open to new networking approaches and different forms of group communication.

This work proposes a realization of Lime messaging through key-based message routing. In particular, we show how the federation process can be implemented by having Lime peers participating to a ring overlay. The key idea consists in having each peer indexing the content of the local tuple space, and further distributing the index on the ring overlay. This approach builds a distributed knowledge of the federated space that narrows down the scope of Lime group communications, targeting only those peers whose context is affected by the group message. In addition to

that, we show how the distribution could be programmed in a transparent way from application developers, providing rule templates based on the type of tuples used by the application.

The rest of this Chapter is organized as follows. Section 6.2 provides an introduction to Lime and tuple space programming. Section 6.3 describes the main contribution of this work, showing the main algorithms and data structures used to port Lime over KBR platforms. Finally Section 6.4 concludes the work and highlights future directions.

6.2 Tuple Space programming

The Lime model defines a coordination layer that can be exploited successfully for designing mobile applications. The design criteria underlying Lime springs from the fact that designing mobile applications is primarily a coordination problem [RMP00], and that a fundamental issue to be tackled is the provision of good abstractions for dealing with, and exploiting, a dynamically changing context. To achieve its goal, Lime borrows and adapts the communication model of Linda [Gel85]. After presenting basic concepts about Linda tuple spaces, the remainder of this Section discusses their re-adaptation inside the Lime middleware.

6.2.1 Linda

In Linda, processes communicate through a shared tuple space that acts as a repository of elementary data structures called tuples. A tuple space is a multi-set of tuples that can be accessed concurrently by several processes. Each tuple is a sequence of typed fields, such as `<'foo', 9, 27.5>`, and contains the information being communicated.

Tuples are added to a tuple space by performing an `out(t)` operation, and can be removed by executing `in(p)`. Tuples are anonymous, thus their selection takes place through pattern matching on the content. The argument `p` is often called a template or pattern, and its fields contain either actuals or formals. Actuals are values; the fields of the previous tuple are all actuals, while the last two fields of `<'foo', ?integer, ?float>` are formals. Formals act like "wild cards", and are matched against actuals when selecting a tuple from the space. For instance, the template above matches the tuple defined earlier. If multiple tuples match a template, the one returned by is selected nondeterministically. Tuples can also be read from the tuple space using the non-destructive `rd(p)` operation. Both `in(p)` and `rd(p)` are blocking, i.e., if no matching tuple is available the calling process is suspended until a matching tuple is added to the space. A typical extension to this synchronous model is the provision of a pair of asynchronous primitives `inp()` and `rdp()`, called probes, that allow non-blocking access to the tuple space. Moreover, some variants of Linda (e.g., [Row98]) provide also bulk operations, which can be used to retrieve

all matching tuples in one step. Lime provides a similar functionality through the `ing()` and `rdg(p)` operations, whose execution is asynchronous like in the case of probes.

6.2.2 Overview of Lime

Linda characteristics resonate with the mobile setting. In particular, communication in Linda is decoupled in time and space, i.e., senders and receivers do not need to be available at the same time, and mutual knowledge of their identity or location is not necessary for data exchange. This form of decoupling is of paramount importance in a mobile environment, where the parties involved in communication change dynamically due to their migration or connectivity patterns. Moreover, the notion of tuple space provides a straightforward and intuitive abstraction for representing the computational context perceived by the communicating processes. On the other hand, decoupling is achieved thanks to the properties of the Linda tuple space, namely its global accessibility to all the processes, and its persistence properties that are clearly hard if not impossible to maintain in a mobile environment.

Transparent context maintenance

In Linda, the data accessible through the tuple space represents the data context available during process interaction. In the model underlying Lime, the shift from a fixed context to a dynamically changing one is accomplished by breaking up the Linda tuple space into many tuple spaces, each permanently associated to a mobile unit, and by introducing connectivity-based rules for transient sharing of these individual tuple spaces.

Each peer's tuple space is referred to as the interface tuple space (ITS), as it provides the only access to the data context for that peer. Each ITS contains the locally shared tuples, and access to this data structure uses standard Linda operations, whose semantics remain basically unaffected. These tuples represent the only context accessible to a peer when it is alone.

When multiple peers are able to communicate, either directly or transitively, they form a Lime group. Conceptually, the contents of the ITSs of all group members are merged, or transiently shared, to form a single, large context which is accessed by each peer through its own ITS. The sharing itself is transparent from the peer's standpoint, however as the members of the group change, the content of the tuple space each peer perceives through operations on the ITS changes as well. When a new peer enters an existing group, its local data context is merged with the group context using an engagement procedure. When a peer leaves the group, the opposite process takes place with a disengagement procedure.

In Lime, peers may have multiple ITSs distinguished by a name since this is recognized as a useful abstraction to separate related application data. The sharing rule in the case of multiple tuple spaces relies on tuple space names: only identically-

named tuple spaces are transiently shared among the members of a group. Thus, for instance, when a peer a owning a single tuple space named X joins a group constituted by a peer b that owns two tuple spaces named X and Y, X only becomes shared between the two peers. Tuple space Y remains accessible only to b, and potentially to other peers owning Y that may join the group later on.

Transient sharing of the ITS constitutes a very powerful abstraction, as it provides a peer with the illusion of a local tuple space that contains all the tuples coming from all the units belonging to the group, without any need to know the members explicitly (i.e., spatial decoupling). The notion of transiently shared tuple space is a natural adaptation of the Linda tuple space to a mobile environment. When physical mobility is involved, and especially in the radical setting defined by mobile ad hoc networking, there is no stable place to store a persistent tuple space. Connections among machines come and go, and the tuple space must be partitioned in some way. Lime enforces an a priori partitioning of the tuple space in subspaces that get transiently shared according to precise rules, providing a tuple space abstraction that depends on connectivity (i.e., existence of a functioning communication link).

Adding location context to tuple space primitives

Thus far, Lime appears to foster a style of coordination that reduces the details of distribution and mobility to content changes in what is perceived as a local tuple space. This view is very powerful, and has the potential for greatly simplifying application design in many scenarios, relieving the designer from the chore of maintaining a consistent view of the context as the system changes configuration. On the other hand, this view may hide too much in domains where the designer needs more fine-grained control over the "slice" (or portion) of the context that needs to be accessed. Also, performance and efficiency considerations may come into play, as in the case where application information would enable access aimed at the specific local tuple space, thus avoiding the greater overhead of a query spanning the whole federated tuple space. Lime extends Linda operations to provide fine-grained control over the context perceived by the mobile unit introducing tuple location parameters. This approach gives a way to operate on user-defined projections of the global tuple space. To this aim, all tuples are implicitly augmented with two fields, representing the tuple's current and destination location. The current location identifies the single peer where the tuple is currently placed, and the destination location indicates the peer with whom the tuple should eventually reside.

The `out[L]` operation extends `out` with a location parameter representing the identifier of the host responsible for holding the tuple. The semantics of `out[L](t)` involve two steps. The first step is equivalent to a conventional `out`, the tuple `t` is inserted in the ITS of the host calling the operation, say `O`. At this point the tuple `t` has a current location `O`, and a destination location `L`. If `L` is currently connected, the tuple `t` is moved to the destination location in the same atomic step. On the other hand, if `L` is currently disconnected the tuple remains at the current location

Current location	Destination location	Defined projection
unspecified	unspecified	Entire federated tuple space
unspecified	L	All existing tuples destined to L
O	unspecified	Tuples in O tuple space
O	L	Tuples currently in P but destined to L

Table 6.1: Accessing different portions of the federated tuple space by using location parameters. In the table, O and L are host identifiers.

O. This "misplaced" tuple, if not withdrawn, will remain misplaced unless L becomes connected. In the latter case, the tuple will migrate to the tuple space associated with L as part of the engagement. By using out[L], the caller can specify that the tuple is supposed to be placed within the ITS on L. This extends the default policy of keeping the tuple in the caller's context, enabling more elaborate schemes for transient communication.

Variants of the in and rd operations that allow location parameters are allowed as well. These operations, of the form in[O,L](p) and rd[O,L](p), enable the programmer to refer to a projection of the current context defined by the value of the location parameters, as illustrated in Table 6.1. The current location parameter enables the restriction of scope from the entire federated tuple space (no value specified) to the tuple space associated to a given host (i.e., a slice of the global space). The destination location is used to identify misplaced tuples.

Reactive programming

In real scenarios, the set of available data and hosts could change rapidly according to mobility patterns. Reacting to changes constitutes a significant fraction of an application's activities. At first glance, the Linda model would seem sufficient to provide some degree of reactivity by representing relevant events as tuples, and by using the in() operation to execute the corresponding reaction as soon as the event tuple appears in the space. Nevertheless, in practice this solution has a number of drawbacks. For instance, programming becomes cumbersome, since the burden of implementing a reactive behavior is placed on the programmer rather than the system. Moreover, enabling an asynchronous reaction would require the execution of in() in a separate thread of control, with consequences on the overall system's performance. Therefore, Lime further extends Linda with *reactions*[MR98]. A reaction R(S,p) is defined by a code fragment S that specifies the actions to be executed when a tuple matching the pattern p is found in the tuple space. Informally, a reaction can fire if a tuple matching the pattern exists in the tuple space. After every regular tuple space operation, a reaction is selected non-deterministically and, if it is enabled, the statements in S are executed in a single, atomic step. This selection and execution continues until no reactions are enabled, at which point normal processing resumes. Blocking operations are not allowed in S, as they may prevent it

from terminating.

Lime reactions can be explicitly registered and unregistered on a tuple space, and hence do not necessarily exist throughout the system's lifetime. Moreover, a notion of *mode* is provided to control the extent to which a reaction is allowed to execute. A reaction registered with mode ONCE is allowed to fire only one time, i.e., after its execution it becomes automatically unregistered, and hence removed from the system. Instead, a reaction registered with mode ONCEPERTUPLE is allowed to fire an arbitrary number of times, but never twice for the same tuple. Finally, reactions can be annotated with location parameters, with the same meaning discussed earlier for `in` and `rd`. Hence, the full form of a Lime reaction is `R[O,L](s,p,m)`, where `m` is the mode.

Reactions provide the programmer with very powerful constructs. They enable the specification of the appropriate actions that need to take place in response to a state change, and allow their execution in a single atomic step. In particular, it is worth noting how this model is much more powerful than many event-based ones [RW97], including those exploited by tuple space middleware such as TSpaces [TSp] and JavaSpaces [Jav], that are typically stateless and provide no guarantee about the atomicity of event reactions.

However, this expressive power comes at a price. In particular, when multiple hosts are present, the content of the federated tuple space depends on the content of the tuple spaces belonging to physically distributed, remote peers. Thus, maintaining the requirements of atomicity and serialization imposed by reactive statements requires a distributed transaction encompassing several hosts for every tuple space operation on any ITS – very often, an impractical solution. For specific applications and scenarios, e.g., those involving a very limited number of nodes, these kind of reactions, referred to as strong reactions, would still be reasonable and therefore they remain part of the model. For practical performance reasons, however, Lime currently limits the use of strong reactions by restricting the current location field to be a host, and by enabling a reaction to fire only when the matching tuple appears on the same host. This constraint effectively forces the detection of a tuple matching `p`, and the corresponding execution of the code fragment `S`, to take place (atomically) on a single host, and hence does not require a distributed transaction.

To strike a compromise between the expressive power of reactions and the practical implementation concerns, Lime introduces a new reactive construct that allows some form of reactivity spanning the whole federated tuple space, but with weaker semantics. The processing of a *weak* reaction proceeds as in the case of a strong reaction, but detection and execution do not happen atomically: instead, execution is guaranteed to take place only eventually, after a matching tuple is detected. The execution of `S` takes place on the host where the reaction was registered.

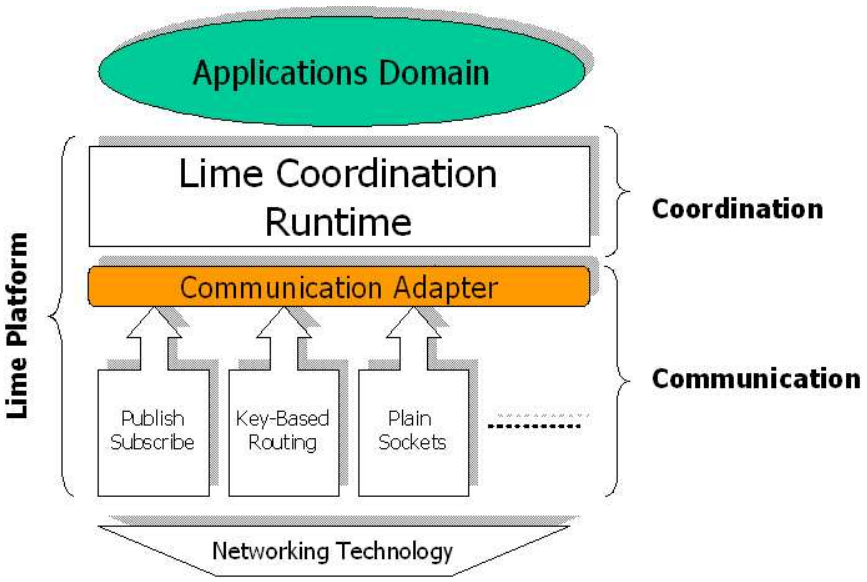


Figure 6.1: Lime architectural overview.

Separating coordination and communication concerns

The transparency of the process of federating several slices in a global tuple space, as well as the semantical guarantees that Lime primitives offer to application programmers, are important features to the adoption and deployment of this middleware. However, in the current Lime implementation [LIM] these features are supported by coordination protocols that make use of TCP sockets for unicast communication, and IP multicasting for group communication. The existing implementation proves the correctness of the coordination policies, but does not provide scalability with large peer groups. In fact, a basic and "coarse-grained" multicasting approach where group messages are sent to all members even if only a small subset of them could benefit from receiving the information, does not scale with peer groups of medium-large sizes. Moreover, current implementations of MANET routing protocols do not support IP multicasting, with direct impacts on the usability of Lime in mobile ad hoc settings. For these reasons, new forms of group communication should be investigated to promote the adoption of Lime.

In the context of project VICom [VIC], Lime has been re-engineered to decouple communication from coordination. As shown in Figure 6.1, the platform now exports a communication interface called Communication Adapter, which separates the coordination protocols from the used message transport technology. This allows to cleanly port coordination on top of different message passing approaches, from publish/subscribe techniques, to structured or unstructured overlay networking, by simply implementing a new Communication Adapter module. The choice of the transport technology to be used by the system, is then left to the application programmer depending on the target environment, or the application constraints.

6.3 Merging tuple spaces and key-based routing

The contribution of this work consists in providing a scalable support for distributed tuple spaces, based on key-based message routing (KBR). Although the work is presented in the context of Lime, showing the handling of both classical tuple space primitives and reactive extensions, it could be applied to other existing tuple space oriented systems.

The general scenario is an arbitrarily large Lime group, where each peer owns a local tuple space. The problem is how to federate the content of local tuple spaces in a global, but distributed view, where peers can all operate in a transparent and scalable fashion. The basic idea is for each peer to index the local space, and then distribute the index over the community using a KBR platform. In this way the system avoids tuples replication, establishing a structured and distributed knowledge of the federated space. This knowledge is then used to narrow down the scope of search primitives, so that the system makes a conservative use of networking resources, and therefore scales up to large peer communities.

An important characteristic of this approach, is that it merges the good features of *content-based* and *key-based* platforms. On one hand, content-based platforms help programmers to create distributed applications, which exploit a rich and expressive set of primitives. This includes the ability to query the system with predicates and patterns, giving flexibility to end-user application. However, this flexibility usually comes at the cost of having no clear guidelines on how to efficiently structure the distributed application, and perform queries in a scalable manner. On the other hand, key-based platforms have been proved to be scalable, being able to route application messages among thousands of peers with bounded costs. However, they impose constraints on data placement, and greatly reduce the flexibility of searches, which could only be performed using keys.

As shown in Figure 6.2 We exploit key-based message routing by means of the CommonAPI proposed in [DZD+03]. This choice enables an evaluation of the resulting system on top of different platforms, possibly operating in different networking environments. Hereafter, we briefly go through the main CommonAPI objects and functions, pointing the reader to the original paper for a complete discussion.

In CommonAPI's jargon, a `Message` is the plain array of bytes produced and consumed by the application layer. Application layer messages are assigned a `Key`, a valid a logical identifier that will drive their routing toward a destination. Peers are represented by `NodeHandle` objects, which are structures containing both a logical identifier and the network information (e.g., IP address and port) of the hosting machine. The function `route(Key K, Message M, NodeHandle N)` is called by the application to initiate the routing of M toward K. Alternatively, the application can indicate the destination peer N, causing the platform to directly deliver M to N. When the message lands on a destination peer N, the platform notifies the application layer by calling the function `deliver(Key K, Message M)`. At this stage the application running on N becomes responsible for handling the message. Finally,

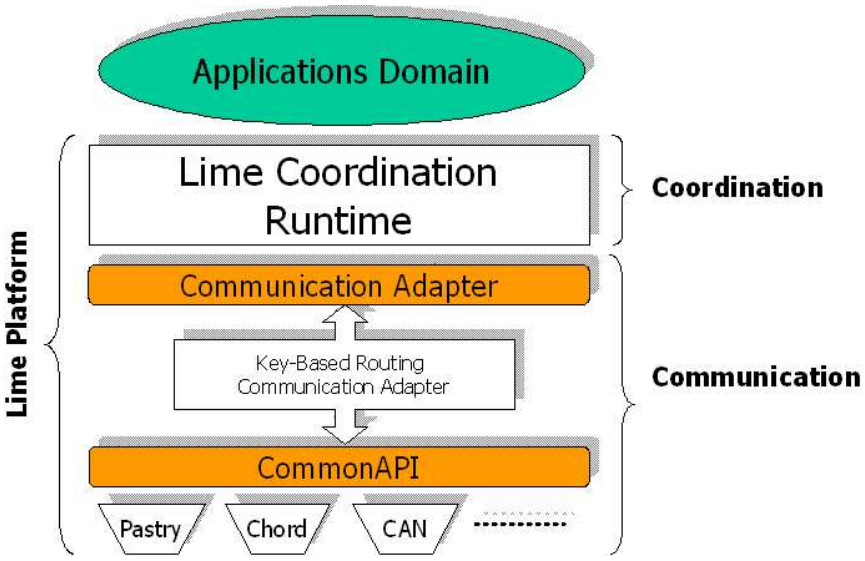


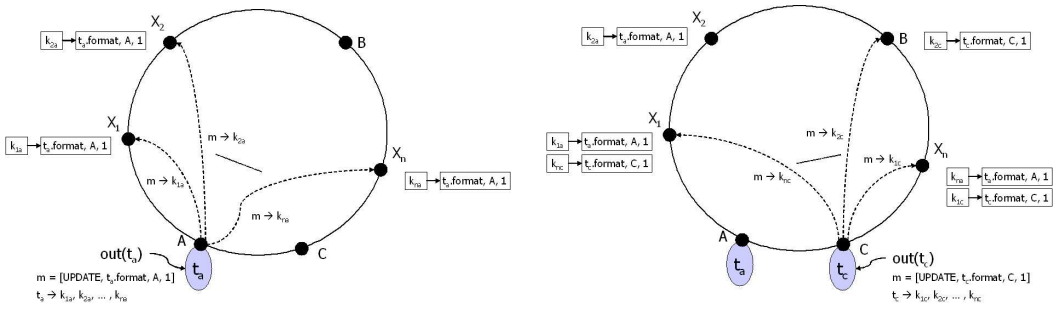
Figure 6.2: Architectural overview of Lime over CommonAPI compliant KBR platforms.

a typical CommonAPI platform provides a strong hash function `H()` that transforms raw byte material into a valid logical identifier. This function is usually called by applications to build keys used for message routing.

6.3.1 A case study with Lime

Given an arbitrarily large community $\{p_1, \ldots, p_n\}$ of peers, each equipped with a local tuple space TS_i, the goal is to federate them into a global space virtually seen as the union $FTS = \bigcup_{i=1}^{n} TS_i$. The result of such a process would allow to extend the scope of tuple space primitives beyond local domains (i.e., the local tuple spaces), and eventually satisfy them with data located on remote peers. The issue clearly consists in guaranteeing system scalability when the federated community is large, and scattered across the network. In the following discussion we will refer to the tuple space TS_A as the *slice* or the *projection* of FTS related to peer A.

Due to the *large* nature of the peer community and the networking constraints of mobile ad hoc networks, the execution of a tuple space primitive on FTS, should only involve a small subset of peers. Just to give an example, suppose peer A executes a `rdp()`. At first, the system will try to satisfy the call with the content on A's slice, but if this initial attempt doesn't succeed, the system should try to satisfy the read probe with the content on the rest of FTS, before returning a `null` tuple. To this end, A has to distribute the operation request in the community, to give other peers a chance to look up their slices and eventually return a tuple copy back to A. At this stage A needs a way to figure out *where* to send the operation request, which means to identify the subset of peers that have good chances of satisfying the



(a) Peer A outs tuple t_a.

(b) Peer C outs tuple t_c.

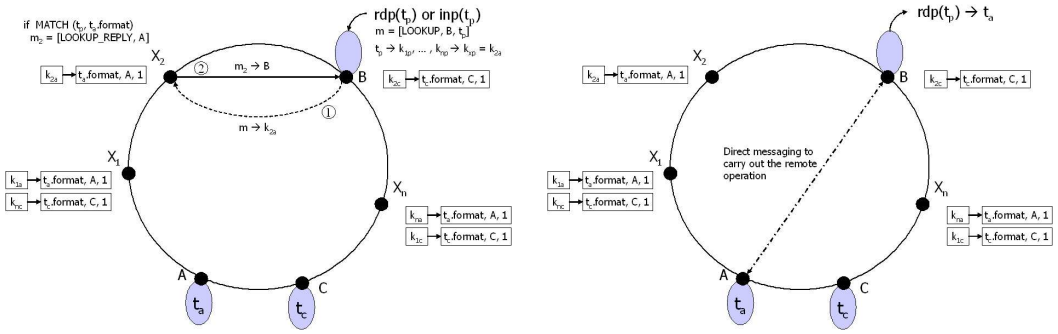
Figure 6.3: Two examples of index updates when peers out tuples in their local tuple space. Upon receiving `UPDATE` messages, peers are able to store information about remote slices. Peers route `UPDATE` messages towards the keys obtained from the new tuples.

read probe. The main idea behind our approach, is to exploit the capabilities of ring overlays to build a distributed knowledge of FTS content, such that it can be efficiently used to select the slices which could satisfy query primitives.

The approach is twofold. Each peer is responsible for indexing the content of its slice, classifying tuples according to their types structure (or format), defined as the sequence of field types. For example, tuples `<'water', 15.5>` and `<'air', 23.6>` have the same format `<String, Float>`, but they are different from tuple `<'fire', 200.9, 5>`, which is `<String, Float, Integer>`, or from `<0.0, 'ice'>` that is `<Float, String>`. As a peer adds tuples to the local slice, it classifies them summarizing the locally available formats, along with the corresponding number of tuples for each format. At the same time, the peer has to distribute the classification outcome, in order to let it available during lookups to the rest of the community. The idea is to provide rules for building keys out of tuples and patterns, so that if a pattern p matches a tuple t, they map to the same set of keys. A KBR platform would then ensure that if peers distribute classification entries using keys calculated from the tuples, and route lookup requests using keys obtained from matching patterns, these will land on the same peers in the overlay. This process makes possible to retrieve information about those peers that can potentially satisfy a given query. In the following sections, let us assume to have rules that coherently build a set of keys out tuples and patterns, postponing a detailed discussion on this topic to Section 6.3.1.

Writing out tuples

As already mentioned, each peer classifies the local slice and, at the same time, distributes the outcome on the ring overlay. It does that in an *incremental* fashion, as



(a) Peer B routes chooses a lookup key, and routes a single `LOOKUP` message towards it. X_2 replies with the list of peers that updated matching entries in its `RemoteTupleIndex`.

(b) Peers B and A perform Lime messaging to carry out the remote operation, and in case of a `rdp()`, B eventually gets back a result.

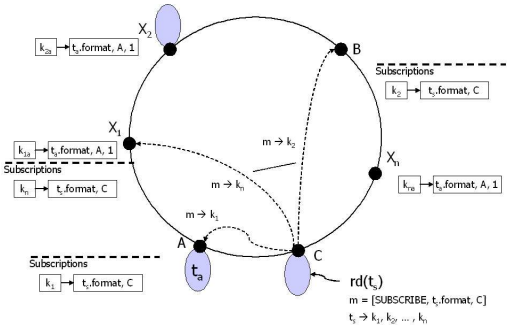
Figure 6.4: Peers *look up* remote indexes before sending instructions for remotely executing a primitive. This approach narrows down the set of peers where the primitive *might* be executed with success.

local agents perform `out()` operations. Basically, the classification is maintained in a `LocalTupleIndex`, and gets updated whenever a new tuple t is written. After the update, the peer takes the modified entries, packs them inside an `UPDATE` message along with its `NodeHandle`, and routes the message using *all* the keys $k_1, \ldots, k_n$ obtained from t. Figures 6.3(a) and 6.3(b) illustrate two examples of this process, showing how peers receiving an `UPDATE` message collect remote classification entries in a `RemoteTupleIndex`.

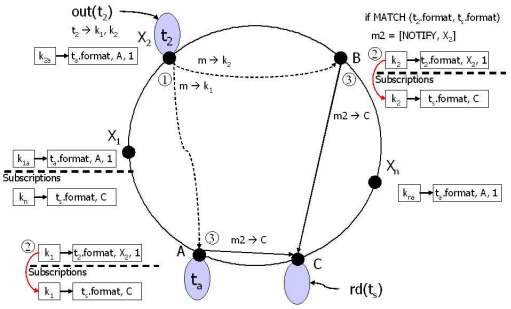
Handling Non-blocking Primitives

The remote execution of non-blocking *probes*, namely `rdp()` and `inp()`, is carried out in two steps. In the first phase (see Figure 6.4(a)), the calling peer retrieves a list of remote slices that can potentially satisfy the probe pattern t_p. To this end it (1) packs a `LOOKUP` message containing t_p and its `NodeHandle`, and routes it only *once* choosing at random one of the keys obtained from t_p. The receiving peer looks up the `RemoteTupleIndex` with the distribution key and the pattern t_p, and eventually sends back (2) a `LOOKUP_REPLY` message containing a list of potential slices. In the second phase (see Figure 6.4(b)), the calling peer B selects a subset of the slices from the list to remotely execute the operation. Finally, for each selected slice, peer B initiates a Lime protocol according to the primitive.

A similar discussion applies for bulk primitives `rdg()` and `ing()`. The lookup phase is performed as above, with the only difference that the querying peer considers the whole list of slices returned from the lookup phase, in order to retrieve all the



(a) Peer C routes a `SUBSCRIBE` message that lands on a set of peers. Those peers will work as *dispatchers* for events of interest for the reaction template.



(b) As peer X_2 routes update messages (1), dispatcher peers (2) have a chance to match tuple formats with reaction templates, and eventually notify (3) C with a `NOTIFY` message.

Figure 6.5: Peers use the KBR platform to set up a subscription scheme for reactions and blocking primitives. Reaction descriptors are routed in the logical space using keys obtained from templates.

tuples matching a given pattern. The maximum execution time for remote bulk primitives is controlled by the querying peer, which waits up to a certain timeout for tuples coming back from each slice.

Handling Blocking Primitives and Weak Reactions

Lime coordination protocols handle blocking primitives and distributed weak reactions in a uniform way. In particular, `rd()` and `in()` operations are implemented and executed as system-defined weak reactions, simplifying their handling inside the distribution scheme.

In principle, a weak reaction from a peer C is handled by *subscribing* to the distributed community, a certain interest of C in the state of the federated tuple space. The interest of C is described by a reaction template, which specifies the format of tuples that should trigger the installation and firing of the reaction code. The community is in charge for maintaining the subscription, as well as for *notifying* the origin peer C about the presence of matching tuples in certain slices of FTS. Once again, the KBR distribution scheme has the role to find out those slices of FTS where there are tuples that might satisfy the reaction.

To this end, each peer is equipped with a `LocalSubscriptionTable` that stores information about the locally originated reactions, as well as a `RemoteSubscriptionTable` that maintains information about remote subscriptions. The local table primarily keeps track of where (i.e., on which slices) reactions have been installed since their creation, so that when the peer needs to cancel them has a way to address a proper

message only to the interested slices. The remote table has instead a dispatching purpose. When a remote slice S sends an `UPDATE` message declaring tuples formats that potentially match the reaction template, the peer notifies the subscriber, and at the same time stores S in a *dispatched* list, to prevent possible duplicate notifications. Just to give an example, consider peer C executing a blocking `rd()`, as illustrated in Figure 6.3.1. Peer C (see Figure 6.5(a)) packs a `SUBSCRIBE` message containing the reaction template and identifier, obtains the distribution keys from the reaction template as usual, and routes the message in the community. Those peers receiving the subscription, store it in their `RemoteSubscriptionTable` according to the key, and begin their dispatching role. When another peer (see X_2 in Figure 6.5(b)) outs a tuple matching the reaction template, the update message will be routed with a similar set of distribution keys, giving the receivers a chance to match the formats, and eventually notify C. To this end they pack a `NOTIFY` message with the reaction identifier and a handle to the remote slice X_2, sending it directly to C. At this stage, C can start the Lime coordination protocol to install the reaction on the notified slices.

Dealing with System Dynamics

Due to the size of the target scenario, the system has to cope with community dynamics. Changes on the slices content, as well as changes in the current set of slices due to peer churns, must reflect on the distributed indexes, in order to properly route tuple space primitives and reactions. Each peer has to cope with two kind of indexes. *Local* indexes (i.e., `LocalTupleIndex` and `LocalSubcriptionIndex`) maintain information about the content shared by the peer, as well as its interests on the status of the distributed tuple space. *Remote* indexes (i.e., `RemoteTupleIndex` and `RemoteSubcriptionIndex`) are the result of the distribution of local indexes owned by remote peers, and are used to find out where tuple space primitives and reactions should be projected.

Local indexes are incrementally built and maintained by the peer when tuples (or reactions) get added, or removed from the local slice. Changes on the local indexes can be updated following two strategies:

- an *eager* strategy causes the immediate generation of an update message for each index change;
- a *lazy* strategy buffers index changes, and updates them in bunches when they reach a predefined number (i.e., $lazy_{size}$), or at periodical time intervals (i.e., $lazy_{time}$).

Lazy strategies are more respectful in terms network resource consumption, but also result in a less reactive system, depending on how the size and time parameters get dimensioned.

Remote indexes receive entries from update messages and organize them in lists in correspondence of the distribution key. They represent a portion of the federated

tuple space knowledge, and therefore should reflect changes on the set of available slices. Once again, their maintenance could be done following two strategies:

- a *lazy* strategy associates *freshness*' timestamps to remote index entries, and considers them stale and ready to be deleted after fixed time intervals. This strategy of course requires a periodical redistribution of local indexes in order to ensure the freshness of remote entries;

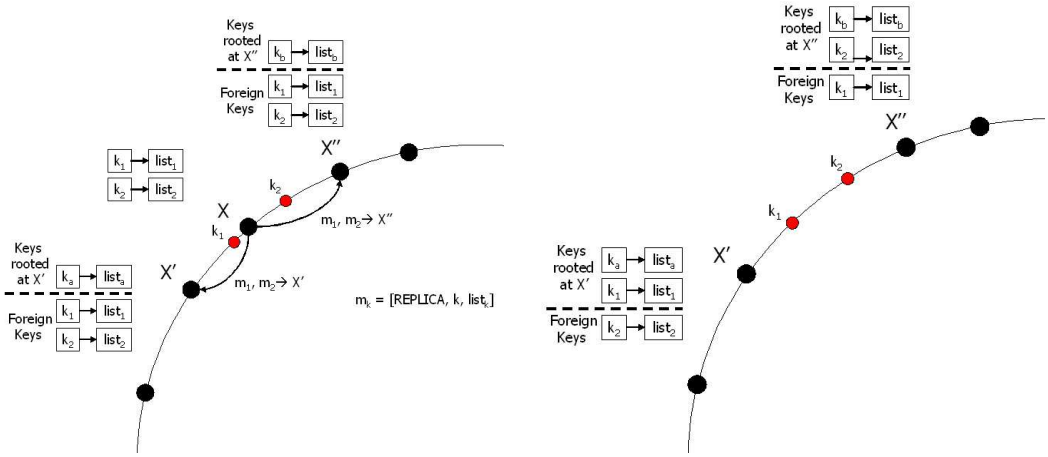
- an *eager* strategy delegates to remote peers the responsibility for maintaining remote entries, and make use of replication techniques as shown in Figure 6.3.1. Each peer knows which key it is root for, and periodically pushes `REPLICA` messages (see Figure 6.6(a)) on its logical neighbors, in order to replicate the associated lists of remote entries. Replicas are assigned a freshness timestamp to bound their lifetime. While the peer replicates those keys for which it is root, it also checks for stale replicas, and eventually deletes them. This strategy guarantees the persistence of remote entries in the system when the root peer leaves (see Figure 6.6(b)).

The eager strategy has advantages in terms of network resources consumption. In fact, as each peer is aware about its neighbors in the logical space, it can directly unicast remote entries. On the other hand, the lazy strategy requires key-based routing for the redistributing local indexes, and therefore potentially generates a logarithmic amount of messages for each list of remote entries.

Engagement and Disengagement — migration of tuples

Lime enriches tuples with a location context, and extends Linda primitives with variants that use a location specifiers. In particular, applications can `out()` tuples specifying a destination slice, with the result of having the tuple migrating to a remote tuple space. In the same way, an application can `in()` or `rd()` tuples from a specified slice. For this reason, each tuple has a *current* and a *destination* location, used to specify in which slice the tuple is currently located, and in which slice it should be located. When these two location specifiers are different, then the tuple is said to be *misplaced*. This happens for example when a peer outs a tuple on a remote slice S that is not present in the system. The idea with Lime is to have misplaced tuples waiting for their destination slice to appear in the system, and then spontaneously migrate during an engagement procedure.

The solution to this problem consists in a coordination scheme similar to what proposed for reactions and blocking primitives. The creation of misplaced tuples designated to S, causes the subscription to an event that signals the join of S in the system. For this purpose the key related to this subscription could be for example the identifier of the destination slice itself.



(a) Peer X periodically sends a `REPLICA` message to its neighbors, for each key rooted at X.

(b) When X quits the ring, neighbors X' and X'' take over for the entries related to k_1 and k_2, previously under the responsibility of X.

Figure 6.6: Peers periodically *replicate* on logical neighbors those entries of their remote indexes associated to keys for which they are root. This approach enhances remote lookup resiliency upon peer departure.

Rules for Building Keys

By default, the system associates at most n update keys to each tuple, using the strong hash function $H()$ provided by the KBR platform. Considering tuples as sequences of `(type, value)` fields, the first key takes into account the whole tuple structure. The remaining keys are obtained by considering only the first $n-1$ fields of tuple. Specifically, each of them is the result of hashing the concatenation of the type specifier, with the position within the tuple. These additional keys index the tuple in correspondence of its field types, and help the system in better distributing remote index entries and lookup workloads in the ring overlay. So given a tuple with m fields $t =< (t_1, v_1), (t_2, v_2), \ldots, (t_m, v_m) >$, k_1 is obtained by hashing the type specifiers' concatenation $(t_1 \bullet t_2 \bullet \ldots \bullet t_m)$, while $k_i = H(t_{i-1} \bullet (i-1))$ when $i = 2 \ldots MAX(n, m)$. Patterns are special tuples made up of actuals and formals, so lookup keys are built exactly in the same way. However, the system considers only one default key during lookup phases, as the list of slices associated to each of them in the distributed index is equivalent.

Additionally to system defined rules, the application programmer has the possibility to specify customized rules for building update and lookup keys. The idea is that the programmer is aware about both the tuple formats and the kind of queries the application will use and do at runtime. Hence, for each tuple format, it can specify one or more index fields, which the system will use to build keys out of

actual values. During update phases, peers will distribute remote entries in correspondence of value-based keys, and when templates show actuals at index positions, lookup phases exploit the same approach to obtain more focused search results.

Just to provide an example, consider a music sharing application where files are stored together with song, band album names, information on the encoding format, file size in KBytes, and a pointer to binary image. In this example, the programmer could structure tuples using a format like `<String, String, String, String, Integer, Binary>`, to organize the song database on each peer. Moreover, it could specify the first three fields as index positions, as meaningful queries will probably be done on songs, albums, or band names. In this way, by performing an `out(<'Talk', 'Coldplay', 'X and Y', 'mp3', 1467, talk_coldplay_xandy.mp3>)`, the system indexes the tuple also with:

$$k' = H(Talk \bullet String \bullet String \bullet String \bullet Integer \bullet Binary)$$

$$k'' = H(String \bullet Coldplay \bullet String \bullet String \bullet Integer \bullet Binary)$$

$$k''' = H(String \bullet String \bullet XandY \bullet String \bullet Integer \bullet Binary)$$

It is worth to note that the whole tuple format is still used along with field values. This is to prevent the generation of identical keys with tuples that have a different format, but an equal value on index fields with the same position.

6.4 Discussion and Related Works

In this Chapter we presented a combined approach to p2p data sharing in mobile and ad hoc environments, based on an integration of tuple space coordination and key-based message routing. Although Lime has been essentially proposed as a middleware for mobile computing, the presented architecture should enable its usage also in large-scale, infrastructure-based environments.

To the best of out knowledge, there is a single proposal, called Comet, which has been designed with similar concepts. Comet [LP05] is a content-based coordination space architecture for wide-area P2P environments, which separates *coordination* and *communication* concerns in two separate layers. The coordination layer implements Linda-like associative primitives, and supports a shared-space coordination model. The communication layer, realized on top of structured data sharing platforms, provides content-based messaging, and is able to manage system heterogeneity and dynamism. The current implementation prototype of Comet is in the framework of the JXTA [JXT] project, and uses Chord [SMLN+03] as for key-based message routing. This system maps tuples on the KBR space using the Hilbert Space-Filling Curve (HSFC), choosing keys from content and tuple descriptors. The system makes no use of indexes, but directly migrates tuples on remote peers when they are added to the space. The authors provide very few details on the actual usage of HSFC to map tuples to points or segments of the node index space, but the

idea behind is to exploit the clustering properties of these class of recursive fractals. The system is described as able to handle queries with incomplete tuple structure information. However, given an incomplete query that maps to a large region of the distributed space, it is not clear how the query propagates to all the nodes in the region (which could be a relatively large set) in an efficient way. Another significant difference is that they encode tuples as XML documents, which provide a minor degree of expressiveness respect to the JAVA classes used by Lime. Finally, while Lime provides primitives for reactive programming, Comet has no mechanisms to help application programmers to react to context changes.

In the future we intend to carry on evaluations of a preliminary prototype of Lime over key-based routing, on Internet-scale test-beds using Planet Lab [PLA].

Chapter 7

Conclusions

7.1 Concluding remarks

Mobile ad hoc networking represents a potential new frontier for wireless communications. Its flexible and infrastructure-less nature, as well as its self-configuration, and self-administration capabilities, make it a prime candidate for becoming the stalwart technology for the future information society. In the past few years, significant research efforts have been spent on networking, as well as on performance enhancements of transport protocols, which also led to the creation of real test-beds for on-the-field protocols' evaluation [APE] [MOB]. However, one important factor for a future deployment of this emerging technology and the creation of potential business opportunities around it, is the provision of suitable middleware platforms to facilitate the design and integration of new services, and foster the development of distributed and mobile applications.

Early proposals of middleware for mobile ad hoc networks focused their attention on information sharing by means of asynchronous communications. Lime [MPR01], as detailed in Chapter 6, makes use of tuple space programming abstractions, while XMIDDLE [ZCME02] realizes data sharing by means of XML documents and operations to support off-line data manipulation. In both cases, the emphasis is on the set of primitives and algorithms to support coordination between interacting parties, assuming a sharp separation with the underlying transport and communication protocols. Instead, this thesis has focused on the communication and networking strategies used to support various data sharing paradigms, with the goal of complementing current proposals with instruments to achieve efficiency and performance.

The investigation began by looking at the problem of building overlays, intended as the communication backbones on top of which information sharing paradigms deliver services in a P2P fashion. A first evaluation of current platforms for structured and unstructured P2P computing, highlighted that overlay management algorithms designed to operate in Internet settings do not tolerate MANET dynamics such as frequent topological reconfigurations, or heavy rates of nodes' churning, and might

generate excessive network overheads with negative impacts to the network capabilities. Therefore, we argued that performance and efficiency could be achieved by having overlay management tasks cooperating with agents at the network layer. Much of the work associated to overlay management consists on peer discovery, link establishment and maintenance. While on the Internet this is done independently from layer-3 activity, as peers are end-user machines and have little or no knowledge about the topology, in MANET settings each node has to cooperatively participate to routing and forwarding, locally collecting topology information that could be exploited by overlays. Moreover, discovering peers means to find out paths toward nodes that are running the P2P service of interest and, in realistic settings, this can't be done more efficiently than the underlying routing protocol. Under this point of view, the discovery procedures used at the network layer represent the "upper bound" in terms of performance, for those used at the application layer by overlays. These considerations, led us to consider *cross-layering* as a viable technique to achieve efficiency and performance.

The work presented in this thesis, is based on a cross-layer architecture that allows interactions between protocols and agents at different (and not adjacent) layers of the stack architecture. The cross-layer architecture is not only a building block for the proposed overlay management techniques, but enables optimizations at different layers of the protocol stack, as shown by the TCP case study in Chapter 2. The distinctive feature stands in the innovative approach of extending the interaction among protocols beyond standard layers' interfaces. Cross-layering can be considered as an alternative way of designing networking software, required by a different networking paradigm such as MANETs. If we compare our cross-layer approach to existing proposals, we conclude that while the latter are isolated solutions that break a clean protocol stack design, the former introduces a "channel" to standardize interactions, maintaining flexibility and modularity. Furthermore, the cross-layer interface identifies the system's component where to introduce semantic controls on protocol interactions, with the final goal of avoiding interfering optimizations [KK05].

The first application of cross-layering addresses unstructured overlays. We proposed a cross-layer version of Gnutella (XL-Gnutella) as an alternative way to organize unstructured data sharing in mobile ad hoc environments. From the obtained results, we conclude that XL-Gnutella nicely tolerates typical ad hoc dynamics, like nodes mobility, network partitioning, and node replacements. In each of these conditions, our protocol was able to maintain the required level of overlay connectivity, without generating traffic bursts. The new proposal presented low bootstrap latencies and high rates of query success, outperforming the legacy version of the protocol, but maintaining compatibility with it.

Another study considered cross-layering in the context of publish/subscribe middleware, as this paradigm is particularly attractive in the mobile computing domain, as it provides a loosely coupled communication scheme. We presented Q, an infrastructure for publish/subscribe that has been specifically designed for mobile ad hoc

environments. In Q, the overlay network used to route event notifications dynamically adapts to the changing topology by means of information extracted from the routing layer. Currently, Q relies on the abstract representation of the network provided by a reactive and source-based routing protocol. However, we believe that it could be easily integrated also on top of a proactive routing protocol. The simulative study demonstrated the feasibility of an event notification service based on a cross-layer approach, where the event dispatching trees reconfigure based on the network topology status.

To complete our perspective on P2P computing in MANETs, we also coped with structured P2P computing and coordination models. An initial case-study evaluation of Pastry, identified severe performance limitations of this platform in MANET environments, and led us to consider the same cross-layer optimization used for unstructured and publish/subscribe platforms. In the structured case, the devised approach is *not* to build and manage an overlay at all. Through a cross-layer interaction with a link-state based routing protocol, peers are able to discover themselves and collect a "view" of the overall ring. The result is that at each key-based message routing procedure, the peer with the closest logical identifier to the message key is elected as root, and the communication reduces to a single unicast stream to the destination. Although this approach delivers good performance, it is hard to engineer its integration with existing platforms, such as Pastry, and therefore produce a system that is able to provide continuous service supply in mixed (i.e., Internet plus ad hoc) settings. However, the approach has proved effective to realize an integration architecture between Lime, a tuple space oriented middleware, and platforms for structured computing. The goal of this work was to provide a functional middleware with an expressive, flexible, and content-based API towards the application programmers, and a scalable communication layer in both large-scale and MANET environments. The proposed architecture shows that structured platforms deliver efficient communication at the expense of flexibility, shifting much of the distributed programming efforts (i.e., organization of distributed indexes, handling of failures and replicas etc.) to application developers.

7.2 Future directions

In a future work, we intend to further refine and evaluate unstructured P2P computing. In particular, we intend to provide a cross-layer overlay formation protocol that builds a middleware communication backbone able to support content-based queries and reactive programming by means of events. Our intention is to carry on refinements in mixed environments (i.e., wireless infrastructure and ad hoc networking), which best match current industrial trends such as mesh networking [BCG05].

Another possible development of the work presented in this thesis, is in the area of wireless content distribution networks. While the presented algorithms for structured and unstructured computing support data discovery in wireless ad hoc

environments, effective solutions should be devised to carry on distributed P2P data delivery, especially when data is of significant sizes (e.g., music or video files). In this area, we intend to explore the capacity and the limitation of platforms like BitTorrent [BIT].

Bibliography

[ABKM01] D. Anderson, H. Balakrishnan, F. Kaashoek, and R. Morris. The Case for Resilient Overlay Networks. In *Proceeding of the 8th Annual Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

[APE] APE Testbed. `http://apetesbed.sourceforge.net`.

[AVT05] Marco Avvenuti, Alessio Vecchio, and Giovanni Turi. A cross-layer approach for publish/subscribe in mobile ad-hoc networks. In *Proceedings of the 2nd International Workshop on Mobility Aware Technologies and Applications (MATA'05)*, Lecture Notes in Computer Science. Springer, November 2005.

[BCG05] R. Bruno, M. Conti, and E. Gregori. Mesh Networks: Commodity Multi-hop Ad hoc Networks. *IEEE Communication Magazine*, pages 123–131, 2005.

[BCM05] P. Bellavista, A. Corradi, and E. Magistretti. Lightweight Replication Middleware for Data and Service Components in Dense MANETs. In *Procceedings of the 6th IEEE Symposium on a World of Wireless Mobile and Multimedia Networks (WoWMoM 2005)*, June 2005.

[Bet02] C. Bettstetter. On the minimum node degree and connectivity of a wireless multihop network. In *Proc. of the 3rd ACM international symposium on Mobile ad hoc networking and computing (MOBIHOC 2002)*, pages 80–91. ACM Press, 2002.

[BIT] The Official BitTorrent Site. `http://www.bittorrent.com`.

[CCL03] I. Chlamtac, M. Conti, and J. J.-N. Liu. Mobile ad hoc networking: imperatives and challenges. *Ad Hoc Networks Journal*, 1(1):13–64, 2003.

[CCMT05] M. Conti, J. Crowcroft, G. Maselli, and G. Turi. A Modular Cross-Layer Architecture for Ad Hoc Networks. In Jie Wu, editor, *Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks*. CRC Press, New York, 2005.

[CCR03] M. Castro, M. Costa, and A. Rowstron. Should we build Gnutella on a structured overlay? In *In Proceedings of HotNets-II*, November 2003.

[CCR04] M. Castro, M. Costa, and A. Rowstron. Peer-to-peer overlays: structured, unstructured, or both? Technical report, 2004. Microsoft Research, Cambridge, Technical Report MSR-TR-2004-73.

[CDT06] M. Conti, F. Delmastro, and G. Turi. Peer-to-Peer Computing in Mobile Ad hoc Networks. In Paolo Bellavista and Antonio Corradi, editors, *Mobile Middleware*. CRC Press, 2006. To appear.

[CGM05] M. Conti, E. Gregori, and G. Maselli. Improving the performability of data transfer in mobile ad hoc networks. In *Proceedings of the 2nd IEEE Conference on Sensor and Ad Hoc Communications (SECON'05)*, Santa Clara, Ca, USA, September 2005.

[CGT04] M. Conti, E. Gregori, and G. Turi. Towards scalable P2P computing for mobile ad hoc networks. In *Proceeding of IEEE PerCom 2004 Workshops*, pages 109–113, Orlando, FL, USA, 2004.

[CGT05] M. Conti, E. Gregori, and G. Turi. A Cross Layer Optimization of Gnutella for Mobile Ad hoc Networks. In *Proceedings of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, Urbana-Champaign, IL, USA, May 2005.

[Chi04] M. Chiang. To Layer or not to Layer: Balancing Transport and Physical Layers in Wireless Multihop Networks. In *Proceedings of IEEE INFOCOM'04*, Hong Kong, China, March 2004.

[CJ03] T. Clausen and P. Jacquet. Optimized Link State Routing Protocol (OLSR). RFC 3626, October 2003.

[CL99] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *In Proceedings of OSDI 1999*, 1999.

[CMC99] M. S. Corson, J. P. Macker, and G. H. Cirincione. Internet-based Mobile Ad Hoc Networking. *IEEE Internet Computing*, 3(4):63–70, July 1999.

[CMT06] M. Conti, G. Maselli, and G. Turi. A flexible cross-layer interface for ad-hoc networks: architectural design and implementation issues. *Ad hoc & Sensor Wireless Networks, An International Journal, (to appear)*, 2006.

[CMTG04] M. Conti, G. Maselli, G. Turi, and S. Giordano. Cross Layering in Mobile Ad Hoc Network Design. *IEEE Computer*, 37(2):48–51, February 2004.

[CRB+03] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making Gnutella-like P2P Systems Scalable. In *In Proceedings of ACM SIGCOMM 2003*, August 2003.

[CRW01] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[CSN02] K. Chen, S. H. Shah, and K. Nahrstedt. Cross-Layer Design for Data Accessibility in Mobile Ad Hoc Networks. *Wireless Personal Communications*, 21(1):49–76, 2002.

[CSWH01] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. *Freenet: A distributed anonymous information storage and retrieval system*. Number 2009 in Lecture Notes in Computer Science. Springer, 2001.

[DZD+03] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkley, CA, USA, Feb 2003.

[EFGK03] Patrick Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[EG01] Patrick Eugster and Rachid Guerraoui. Content-Based Publish/Subscribe with Structural Reflection. In *Proceedings of the 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS'01)*. The Usenix Association, 2001.

[EG04] Patrick Eugster and Rachid Guerraoui. Distributed Programming with Typed Events. *IEEE Software*, 21(2):56–64, March 2004.

[FI03] I. Foster and A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *In Proceedings of IPTPS'03*, February 2003.

[FRE] Freepastry. FreePastry web site: `http://freepastry.rice.edu`.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Computing Surveys*, 1(7):80–112, January 1985.

[GPLC00] M. Gerla, G. Pei, S. Lee, and C. Chiang. On-demand multicast routing protocol (ODMRP) for ad hoc networks. 2000. Internet draft, http://www.ietf.org/proceedings/00jul/ID/manet-odmrp-02.txt, Work in progress.

[HGM01] Y. Huang and H. Garcia-Molina. Publish/Subscribe in a mobile enviroment. In *Proceedings of the 2nd ACM International Workshop on Data Endineering for Wireless and Mobile Access (MobiDE'01)*, pages 27–34, May 2001.

[Jav] JavaSpaces. The SUN Microsystems JavaSpaces Specification web page. http://www.sun.com/jini/specs/js-spec.html.

[JM96] David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. *Mobile Computing*, pages 153–181, 1996.

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

[Kaw04] V. Kawadia. Protocols and Architecture for Wireless Ad Hoc Networks. Ph.D. dissertation, University of Illinois at Urbana-Champaign, sep 2004.

[KBC+00] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *In Proceedings of ACM ASPLOS 2000*. ACM Press, November 2000.

[KK05] V. Kawadia and P. R. Kumar. A Cautionary Perspective on Cross Layer Design. *IEEE Wireless Communications*, February 2005.

[KKT04] U. C. Kozat, I. Koutsopoulus, and L. Tassiulas. A Framework for Cross-layer Design of Energy-efficient Communication with QoS Provisioning in Multi-hop Wireless Netwroks. In *Proceedings of IEEE INFOCOM'04*, Hong Kong, China, March 2004.

[KLW03] A. Klemm, C. Lindemann, and O. P. Waldhorst. A Special-Purpose Peer-to-Peer File Sharing System for Mobile Ad Hoc Networks. Workshop on Mobile Ad Hoc Networking and Computing, in conjunction with WiOpt'03, 2003.

[KM02] T. Klinberg and R. Manfredi. Gnutella Protocol Specification v0.6. `http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html`, June 2002.

[KP02] R. Koodli and C. E. Perkins. Service Discovery in On-Demand Ad Hoc Networks. Internet Draft, October 2002.

[LIM] The Lime Project Website. `http://lime.sourceforge.net`.

[LP05] Z. Li and M. Parashar. Comet: A Scalable Coordination Space for Decentralized Distributed Environments. In *Proceedings of HOT-P2P 2005*, CA, USA, May 2005.

[LRW03] N. Leibowitz, M. Ripeanu, and A. Wierzbicki. Deconstructing the kazaa network. In *In Proceedings of Third IEEE Workshop on Internet Applications (WIAPP'03)*, June 2003.

[LSS05] Q. Li, N. B. Shroff, and R. Srikant. Asymptotically Optimal Power-Aware Routing for Multihop Wireless Networks with Renewable Energy Sources. In *Proceedings of IEEE INFOCOM'05*, Miami, USA, March 2005.

[Ltd01] Sherman Networks Ltd. KaZaa Media Desktop. http://www.kazaa.com, 2001.

[LXG03] H. Lim, K. Xu, and M. Gerla. TCP Performance over multipath routing in mobile ad hoc networks. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 1064–1068, May 2003.

[MC02] René Meier and Vinny Cahill. STEAM: Event-Based Middleware for Wireless Ad Hoc Network. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCSW'02)*, pages 639–644, Washington, DC, USA, 2002. IEEE Computer Society.

[MCE02] C. Mascolo, L. Capra, and W. Emmerich. Middleware for Mobile Computing (A Survey). In E. Gregori, G. Anastasi, and S. Basagni, editors, *Neworking 2002 Tutorial Papers*, LNCS 2497, pages 20–58. Springer, 2002.

[MMH04] Mirco Musolesi, Cecilia Mascolo, and Stephen Hailes. Adapting asynchronous messaging middleware to ad hoc networking. In *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, pages 121–126, New York, NY, USA, 2004. ACM Press.

[MOB] Project MobileMAN. `http://cnd.iit.cnr.it/mobileMAN`.

[MPH02] T. Moreton, I. Pratt, and T. Harris. Storage, Mutability and Naming in Pasta. In *In Proceedings of the International Workshop on P2P Computing at Networking 2002*, May 2002.

[MPR01] A.L. Murphy, G.P. Picco, and G.-C. Roman. LIME: a Middleware for Physical and Logical Mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS'01)*, pages 524–533, May 2001.

[MR98] P.J. McCann and G.-C. Roman. Compositional Programming Abstractions for Mobile Computing. *IEEE Transaction on Software Engineering*, 2(24):97–100, 1998.

[MSN84] D. Michael, A. Schroeder, and R. Needham. Experience with grapevine: The growth of a distributed system. *ACM Transactions on Computer Systems*, 2(1):3–23, February 1984.

[NAP] Napster. napster media sharing system. `http://www.napster.com/`.

[Net03] Scalable Networks. The QualNet Simulator, 2003. http://www.scalable-networks.com.

[NRL] PROTEAN Research Group. http://cs.itd.nrl.navy.mil/5522/.

[NS2] The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[PBRD03] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. IETF RFC-3561, July 2003.

[PC02] Ian Pratt and Jon Crowcroft. Peer-to-Peer systems: Architectures and Performance. Networking 2002 Tutorial Session, May 2002.

[PCM03] Gian Pietro Picco, Gianpaolo Cugola, and Amy L. Murphy. Efficient Content-Based Event Dispatching in the Presence of Topological Reconfiguration. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, Washington, DC, USA, 2003. IEEE Computer Society.

[PLA] PLANETLAB: An open platform for developing, deploying, and accessing planetary-scale services. http://www.planet-lab.org/.

[PMR99] G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In *Proceedings of the 21st International Conference on Software Engineering*, pages 368–377, May 1999.

[RD01a] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[RD01b] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent P2P storage utility. In *In Symposium on Operating Systems Principles*, pages 188–201, 2001.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM'01*, pages 149–160, San Diego, CA, USA, 2001.

[RMP00] G.-C. Roman, A.L. Murphy, and G.P. Picco. Coordination and Mobility. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 254–273. Springer, 2000.

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

[RW97] D.S. Rosenblum and A.L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proc. of the 6th European Software Engineering Conference (ESEC/FSE97)*, number 1301 in Lecture Notes in Computer Science, Zurich (Switzerland), 1997. Springer.

[SET] Seti@Home Project. http://setiathome.ssl.berkeley.edu/.

[SGF02] R. Schollmeier, I. Gruber, and M. Finkenzeller. Routing in Mobile Ad Hoc and Peer-to-Peer Networks. A Comparison. In *Proceedings of Networking 2002 Workshops*, Pisa, Italy, May 2002.

[SGN03] R. Schollmeier, I. Gruber, and F. Niethammer. Protocol for Peer-to-Peer Networking in Mobile Environments. In *Proceedings of 12th IEEE International Conference on Computer Communications and Networks (ICCCN'03)*, Dallas, Texas, USA, October 2003.

[SMLN+03] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *IEEE Transactions on Networking*, 11, 2003.

[SYZ+05] E. Setton, T. Yoo, X. Zhu, A. Goldsmith, and B. Girod. Cross-layer Design of Ad Hoc Networks for Real-Time Video Streaming. *IEEE Wireless Communications*, August 2005.

[TSp] TSpaces. The IBM TSpaces Web Page. http://www.almaden.ibm.com/cs/TSpaces.

[VIC] VICom: Virtual Immersive Communication. http://www.vicom-project.it.

[VIE03] Einar Vollset, David B. Ingham, and Paul D. Ezhilchelvan. JMS on Mobile Ad Hoc Networks. In *IFIP-TC6 8th International Conference on Personal Wireless Communications (PWC 2003)*, volume 2775 of *Lecture Notes in Computer Science*, pages 40–52. Springer, September 2003.

[YLA02] W. H. Yuen, H. Lee, and T. D. Andersen. A Simple and Effective Cross Layer Networking System for Mobile Ad Hoc Networks. In *Proceedings of IEEE PIMRC 2002*, Lisbon, Portugal, September 2002.

[YVGM04] B. Yang, P. Vinograd, and H. Garcia-Molina. Evaluating GUESS and Non-Forwarding Peer-to-Peer Search. In *Proc. of 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 209–218, March 2004.

[ZCME02] S. Zachariadis, L. Capra, C. Mascolo, and W. Emmerich. XMIDDLE: Information Sharing Middleware for a Mobile Environment. In ACM Press, editor, *Proc. of the 24^{th} International Conference of Software Engineering (ICSE 2002), Demo Session*, LNCS 2497, page 712, Orlando, Florida, 2002.

[ZKJ01] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report, 2001. Tech. Rep. UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley.

[ZPS00] Y. Zhang, V. Paxson, and S. Shenker. The stationarity of internet path properties: Routing, loss, and throughput. ACIRI Technical Report, 2000.