RZ 3433 (# 93700) 07/08/02
Computer Science 91 pages

Research Report

Routing and Data Location in Overlay Peer-to-Peer Networks

Roberto Rinaldi and Marcel Waldvogel*

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

*e-mail: mwl@zurich.ibm.com

Routing and Data Location in Overlay Peer-to-Peer Networks

Roberto Rinaldi and Marcel Waldvogel*

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland
*e-mail: mwl@zurich.ibm.com

Abstract

Peer-to-peer overlay networks offer a novel platform for a variety of scalable and decentralized distributed applications. Systems known as Distributed Hash Tables provide efficient and fault-tolerant routing, object location and load balancing within a self-organizing overlay network. The alternative solution we propose is an overlay location and routing infrastructure that efficiently uses minimal local information to achieve global routing. The main novelty of our approach consists in fitting the overlay network in a hyper-toroidal space and building it with "locality awareness". Thanks to this specific network construction phase, forwarding decisions always take into account "locality preservation" in an implicit manner, leading to significant improvements in end-to-end delays and path lengths.

With this overlay network it is possible to obtain global routing by adding minimal information to each single host and by making only local forwarding decisions. Our analysis shows how the average path length coming from the overlay routing is close to the optimal average path length of the underlaying network: on average, they only differ by a factor of 2. Furthermore, "locality preservation" has a significant impact on the end-to-end latency of the routing process as well.

Such a system can be viewed as novel in the field of peer-to-peer data location and addressing, allowing the development of new applications in a real "low-latency" environment.

Contents

List of Figures

Chapter 1

Introduction

1.1 Scenario overview

The computing world is experiencing a transition from desktop PCs to connected information devices, which will profoundly change the way information is used. The number of devices capable of connecting to the Internet continually increases: laptops, cell phones, car GPS, and uncountable Bluetooth devices. The need for a global system which could add its own reliable mechanisms to the advantages offered by diffusion over the Internet, becomes more significant every day.

The ever increasing size of routing tables represents a significative problem when designing large-scale network protocols. After years of predictable growth, in the first semester of 2001, the size of routing tables exploded, topping 104,000 entries in some cases. Even more troubling is evidence that frequent updates to the routing table entries by network managers are causing instability in the Internet's backbone routing infrastructure. Nobody knows how big or how active routing tables can get before the Internet's core routers start crashing. The exchange of routing information is even larger.

1.1.1 Routing table scalability

The routing table is the complete set of routes that describe the origin of every routed address within the Internet. As new networks connect to the Internet they announce their address prefix into this table. As the Internet grows so does the size of this table. Looking at this table in regular intervals can give us a good idea of what is happening in the routing system.

In routing circles you will often hear talk about the "Big Question": how will the routing scale deal with the demands of tomorrow's Internet? While many aspects of the Internet are prone to scaling pressures, routing appears to be one of the technologies at the tip of the scaling problem, and the issues involved are illustrative of the more general issues of technology design in a rapidly expanding domain.

There is quite a story behind the chart in Figure 1.1, and it can tell us a lot about what is likely to happen in the future. The chart appears to have four distinct phases: exponential growth between 1988 and 1994, a correction during 1994, linear growth from 1995 to 1998, and a resumption of exponential growth in the past two year, and some oscillation in the past few months.

Prior to 1994 the Internet used a routing system based on classes of addresses. One half of the address space was termed class-A space, and used a routing element of 8 bits (or a /8) and the remaining 24 bits were used to number hosts in the network. One quarter of the space was termed class-B space, with 16 bits of routing address (/16) and 16 bits of host address space, and one eighth was the Class-C space, with 24 bits of routing address (/24) and 8 bits of host space. According to the routing system, routed networks came in just three sizes, small (256 hosts), medium (65,535 hosts) and large (16,777,215 hosts). Real routed networks however came in different sizes, most commonly one or two thousand hosts. For such networks a Class-B routing address was a severe case of oversupply of addresses, and the most common technique was to use a number of Class-C networks. As the network expanded so did the number of Class-C network routes appearing in the routing table. By 1992 it was becoming evident that if we did not do something quickly the routing table would expand beyond the capabilities of the routers being used at the time, and by "quickly" we were talking months rather than years.

The solution was termed *CIDR* or *Classless Inter-Domain Routing*. The technique

was elegant and effective. Instead of dividing the network into just three fixed prefix lengths, each routing advertisement was allowed to have an associated prefix length.

CIDR lead to the other change in routing policy, namely that of provider-based addresses and provider route aggregation. Instead of allocating network address blocks to every network, the address registry allocated a larger address block (a /19 prefix) to a provider, who in turn allocated smaller address blocks from this block to each customer. Now a large number of client networks would be encompassed by a single provider routing advertisement. This technique, hierarchical routing, is used in a number of network architectures, and is a powerful mechanism to aggregate routing information.

From 1995 to 1998, the combination of CIDR and hierarchical provider routing proved very effective. While the Internet continued to double in size each year (or even faster), the routing space grew at a linear rate, increasing in size by some 10,000 routes per year. For the routing system this was good news: vendors were able to construct larger routers at a pace that readily matched the growth of the Internet, and a combination of Moore's law in hardware and CIDR and hierarchical routing in the routing system proved very effective in coping with the dramatic growth of the Internet.

But midway through 1998 something changed. The routing system stopped growing at a linear rate and resumed a pattern of exponential growth, at a rate of a little les than 50% each year. While the size of the routing table was some 105,000 routes in mid-2001, in a year's time it could be some 150,000 routes and 225,000 routes the year after, and so on. Within six years, the table will be reach some 1,000,000 routes at this rate of growth. Figure 1.1 shows the latest evolution of active BGP entries in Internet core routers.

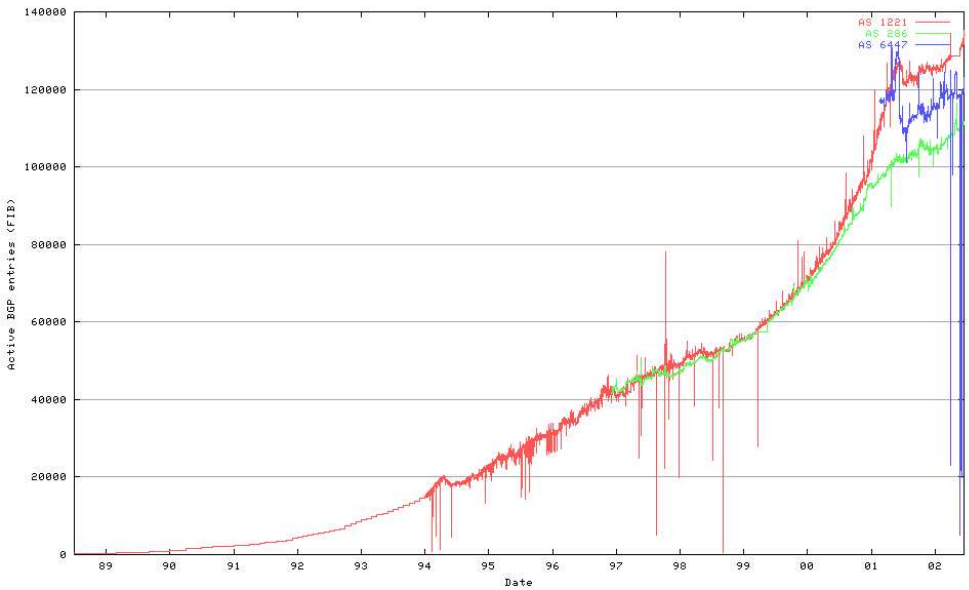


Figure 1.1: Active BGP entries until 14^{th} June 2002 [1].

There are many factors driving this pattern of routing-table growth, including detailed connectivity policies being expressed in fine-grained prefixes, traffic engi-

neering across a dense mesh of interconnectivity, and even some level of operator inattention to aggregation management. But if there was one factor that appears to be the predominant driver of growth, then it would be, in one word, multi-homing. Multi-homing is when an ISP has a number of external connections to other networks. This may take the form of using a number of upstream ISPs as service providers, or using a combination of upstream providers and peer relationships established either by direct links or via a peering exchange. The way in which multi-homing impacts the global BGP table is that multi-homing entails pushing small address fragments into the global table with a distinct connection policy that differs from that of any of its upstream neighbors. What we are seeing in this sharp rise in the size of the BGP table is a rapid increase in the number of small address blocks being announced globally.

Connecting to multiple upstream services and connecting to peering exchanges bith imply the use of more access circuits. When the cost of these circuits was high, the offset in terms of benefit was low enough to negate most of the potential benefits of the richer connectivity mesh. Over the past few years the increasing level of competition in the largely deregulated activity of communications agents provision resulted in reductions in the price of these services. This, togheter with an increasing technical capability in the ISP sector, has resulted in the increasing adoption of multi-homed ISPs. In the quest for ever increasing resilience, we are also starting to see multi-homed customers in addition to ISPs. In the routing system we are seeing the most challenging of environments: a densely interconnected semi-mesh of connectivity, with very fine-grained policies being imposed on top of this connectivity. Any topological or other form of logical hierarchical abstraction is largely lost in such an environment, and the routing system is facing increasing overheads in its efforts to converge the distributed algorithm to a stable state.

If multi-homing becomes a common option of corporate customers, then the function of providing resilience within a network will shift from a value-added role in the network to that of a customer responsibility. And if customers are not prepared to pay for highly robust network services from any single ISP then there is little economic incentive for any single ISP, to spend the additional money to engineer robustness within their service. From that perspective, what the ISP industry appears to be heading into is a case of a somewhat disturbing self-fulfilling prophesy of minimalist network engineering with no margin for error.

But then, as the economists tell us, such are the characteristics of a highly competitive open commodity market. That is quite a story that emerges from a simple chart of the size of the BGP routing table.

1.1.2 Peer-to-peer systems

Peer-to-peer (P2P) Internet applications have recently been popularized through file-sharing applications such as Napster [3], Gnutella [2], and Freenet [10]. While much of the attention has been focused on the copyright issues raised by using these applications, P2P systems have many interesting technical aspects such as decentralized control, self-organization and adaptation. They can be characterized as distributed systems in which all nodes have identical capabilities and responsibilities and all communication is potentially symmetric.

Even though they were introduced only a few years ago, P2P file-sharing systems are now one of the most popular Internet application and have become a major source of Internet traffic. P2P has been generating lots of attention. Like PCs in the

1980s and the Web in the 1990s, industry watchers say it is one of those disruptive technologies that will turn the world of computing upside-down.

P2P represents a class of applications that take advantage of resources (storage, cycles, content, human presence) available at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses[1], P2P nodes must operate outside the DNS system and have significant or total autonomy from central servers.

Until 1994, the whole Internet had one model of connectivity. Machines were assumed to be always on, always connected, and assigned permanent IP addresses. The DNS system was designed for this environment, where a change in IP address was assumed to be abnormal and rare, and could take days to propagate through the system.

With the appearance of Mosaic [29], another model began to spread. To run a Web browser, a PC, not connected before, needed to be connected to the Internet over a modem, with its own IP address. This created a second class of connectivity, because PCs would enter and leave the network cloud frequently and unpredictably.

Furthermore, ISPs began to assign IP addresses dynamically, giving each PC a different, possibly masked, IP address with each new session and justifying it with the reason that there were not enough IP addresses available to handle the sudden demand caused by Mosaic. This instability prevented PCs from having DNS entries, and therefore prevented PC users from hosting any data or net-facing applications locally.

For a few years, treating PCs as dumb but expensive clients worked well. PCs had never been designed to be part of the fabric of the Internet, and in the early days of the Web, the toy hardware and operating systems of the average PC made it an adequate life-support system for a browser, but good for little else.

Over time, though, as hardware and software improved, the unused resources that existed behind this veil of second-class connectivity started to look like something worth getting at. At a conservative estimate, the world's Net-connected PCs currently host an aggregate ten billion MHz of processing power and ten thousand terabytes of storage, assuming only 100 million PCs among the net's 300 million users, and only a 100 MHz chip and 100 Megabyte drive in the average PC.

P2P computing is not exactly new. As many as 30 years ago, companies were working on architectures that would now be labeled P2P. But today, several factors promotethe P2P movement: inexpensive computing power, bandwidth, and storage. P2P computing is the sharing of computer resources and services by direct exchange between systems. These resources and services include the exchange of information, processing cycles, cache storage, and disk storage for files. P2P computing takes advantage of existing desktop computing power and networking connectivity, allowing economical clients to leverage their collective power to benefit the entire enterprise.

In a P2P architecture, computers that have traditionally been used solely as clients communicate directly among themselves and can act as both clients and servers, assuming whatever role is most efficient for the network. This reduces the load on servers and allows them to perform specialized services (such as mail-list generation, billing, etc.) more effectively. At the same time, P2P computing can reduce the need for IT organizations to increase parts of their infrastructure in order to support certain services, such as backup storage.

In the enterprise, P2P is about more than just the universal file-sharing model

[1]Most dialup services are based on dynamic IP (re-)assignment, DHCP and/or NAT protocols.

popularized by Napster. Business applications for P2P computing cover several scenarios.

1. *Collaboration.* P2P computing empowers individuals and teams to create and administer real-time and off-line collaboration areas in a variety of ways, whether administered, un-administered, across the Internet, or behind the firewall. P2P collaboration tools also mean that teams have access to the newest data. Collaboration increases productivity by decreasing the time for multiple reviews by project participants and allows teams in different geographic areas to work together. As with file sharing, if implemented efficiently, it can decrease network traffic by eliminating e-mail and decreases server storage needs by storing the project locally.

2. *Edge services.* This is exactly what you think: Akamai for the enterprise. P2P computing can help businesses deliver services and capabilities more efficiently across diverse geographic boundaries. In essence, edge services move data closer to the point at which it is actually consumed, acting as a network caching mechanism. For example, a company with sites on several continents needs to provide the same standard training across these continents using the Web. Instead of streaming the database for the training session on one central server located at the main site, the company can store the video on local clients, which act essentially as local database servers. This speeds up the session because the streaming happens over the local LAN instead of the WAN. It also utilizes existing distributed storage space, thereby saving money by eliminating the need for local storage on servers (e.g Groove Networks, Lotus Notes).

3. *Distributed computing and resources.* P2P computing can help businesses with large-scale computer processing needs. Using a network of computers, P2P technology can use idle CPU MIPS and disk space, allowing businesses to distribute large computational jobs across multiple computers. In addition, results can be shared directly between participating peers. The combined power of previously untapped computational resources can easily surpass the normally available power of an enterprise system without distributed computing. The results are faster completion times and lower costs because the technology takes advantage of power available on client systems.

4. *Intelligent agents.* Peer to peer computing also allows computing networks to dynamically work together using intelligent agents. Agents reside on peer computers and communicate various kinds of information back and forth. Agents may also initiate tasks on behalf of other peer systems. For instance, Intelligent agents can be used to prioritize tasks on a network, change traffic flow, search for files locally or determine anomalous behavior and stop it before it effects the network, such as a virus.

P2P designs harness huge amounts of resources: the content advertised through Napster has been observed to exceed 7 TB of storage on a single day, without requiring centralized planning or huge investments in hardware, bandwidth, or rack space. As such, P2P file sharing may lead to new content distribution models for applications such as software distribution, file sharing, and static web content delivery.

Unfortunately, most of the current P2P designs are not scalable. For example, in Napster a central server stores the index of all the files available in the entire Napster community. To retrieve a file, a user queries this central server using the

desired file's well-known name or other search criteria and obtains the IP address of a user machine storing the requested file. The file is then downloaded directly from this user machine. Thus, although Napster uses a P2P communication model for the actual file transfer, the process of locating a file is still very much centralized. This makes it both expensive (to scale the central directory) and vulnerable (as there is a single point of failure). Gnutella goes a step further and decentralizes the file-location process as well. Users in a Gnutella network self-organize into an application-level mesh on which requests for a file are flooded with a certain scope. Flooding every request is clearly not scalable and, because the flooding has to be curtailed at some point, may fail to find content that is actually in the system. We soon recognized that central to any P2P system is the indexing scheme used to map file names (whether well-known or discovered through some external mechanism) to their location in the system. That is, the P2P file transfer process is inherently scalable, but the difficult part is finding the peer from which to retrieve the file. Thus, a scalable P2P system requires, at the very least, a scalable indexing mechanism.

There are at least three key problems:

- Firstly, it is necessary to have efficient algorithms for data location and routing within the network. The term *efficient* should be interpreted in terms of *query latency* and *routing table size*.
- Secondly, scalability is not a trivial problem. For an example, look at the Gnutella network: it can have serious scalability problems because the data-location mechanism (based on broadcast to all neighbor nodes) is not efficient at all for large populations of nodes or high query latencies.
- Finally, attention must be paid to the network convergence and self-stabilization: P2P network usually turn out to be very dynamic because most nodes uses dialup services. The continuous joining and leaving of nodes should not have influence the stability of the network. Any kind of *oscillation* or *meta-stability* should be prevented.

1.1.3 Decentralized storage systems

Decentralized systems seem to offer great opportunities (in this research field). Their main advantages are

- the opportunistic sharing of Internet connected computers is a low cost method for achieving high computational power;
- data replication over multiple hosts ensures high resilience and fault tolerance: contents can be retrieved even when nodes go down or misbehave potentially with malicious intent;
- smart content location is a powerful means to improve query latencies, to avoid wasting network bandwidth and to greatly enhance the overall performance;
- preserving the locality of accesses improves the overall performance.

The need for reliable storage systems has been an important and partly open problem in modern communication systems. On the one hand, the Internet expansion has created great opportunities for decentralized, smart systems. On the other hand, it has also created new needs in the fields of security, performance, and reliability. The

growing amount of data exchanged every day on the Internet needs more and more reliable systems. The availability of data is threatened seriously both by the increasing load that most servers are requested to satisfy, and by the increasing number of denial-of-service (DoS) attacks. Some current research efforts are also in the field of data sharing in multiple contexts, i.e. data sharing for mobile users or sharing between home and office.

Even if storage systems constitute a great investment field, they are not capable of guaranteeing a democracy in storage. This means that they cannot share data over multiple hosts in an attempt to distribute the load as much as possible. The research efforts in this field often aim more at eliminating the symptoms than at fixing the root of the problem. In fact there are different points of view of possible solutions, such as

- to prevent overload by increasing capacity of the network;
- to increase security against DoS attack;
- to design complex load-balancing systems suited for each specific scenario;

1.2 Distributed Hash Tables

Even though they were introduced only a few years ago, P2P file-sharing systems are now one of the most popular Internet applications and have become a major source of Internet traffic. Thus, it is extremely important that these systems be scalable. Unfortunately, the initial designs of popular P2P systems have significant scaling problems; for example, Napster [3] has a centralized directory service, and Gnutella [2] employs a flooding-based search mechanism that is not suitable for large systems.

In response to these scaling problems, several research groups have (independently) proposed a new generation of scalable P2P systems that support a distributed hash table (DHT) functionality; among them are Tapestry [40], Pastry [35], Chord [37], and Content Addressable Networks (CAN [32]). In these systems, that we call *Distributed Hash Tables* (*DHTs*), files are associated with a key (produced, for instance, by hashing the file name), and each node in the system is responsible for storing a certain range of keys. An example is shown in Figure 1.2.

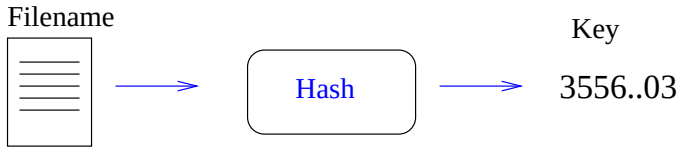


Figure 1.2: Example: from file name to key.

There is one basic operation in these DHT systems, *lookup(key)*, which returns the identity (e.g., the IP address) of the node storing the object with that key. This operation allows nodes to put and get files based on their key, thereby supporting the hash-table-like interface. This DHT functionality has proved to be a useful substrate for large distributed systems; a number of projects are proposing to build Internet-scale facilities layered above DHTs, including distributed file systems [25, 11, 13], application-layer multicast services [41, 33], event notification services [36, 9], and

chat services. With so many applications being developed in so short a time, we expect the DHT functionality to become an integral part of the future P2P landscape.

The DHT nodes form an overlay network, with each node having several other nodes as neighbors. When a lookup(key) is issued, the lookup is routed through the overlay network to the node responsible for that key. The scalability of these DHT algorithms is tied directly to the efficiency of their routing algorithms, given that each one listed above (Tapestry, Pastry, Chord, and CAN) employ a different routing algorithm.

1.3 Project description

The network model analyzed in this project is completely *decentralized* and based on a pure[2] P2P structure.

User-oriented models include systems in which users are conscious of the global complexity and actively perform precise actions to improve reliability and/or performance. A simple example of a user-oriented model is the download of a file available from multiple mirror sites. In this case, even if the availability of the document is guaranteed, the end-user holds the responsibility of choosing mirror. Here, user-oriented models [23] have been discarded because the project goal is to build up a robust system in which end-users do not need to be conscious of data location and of the addressing subsystem. The network itself takes care of locating and retrieving documents. Particular attention has been paid to performance, especially in terms of network latency and memory usage efficiency at each node (small amount of per-request state or, better, a stateless network and small routing tables).

The overlay network designed in our project does not rely on a specific well-defined OSI layer. The only assumption is that at least a point-to-point network protocol is provided. A fully functional IP layer adapts perfectly to the network structure of this project, but also any other kind of basic underlay routing protocol is well suited. If one wants to make a strict classification our overlay network is surely above layer three; probably it best fits into layer five (*Session*), but in general the nature of overlay networks breaks the layering division.

For the routing capabilities, the project proposes a possible alternative to standard routing protocols. Using them as a basic layer, the purpose is to build a higher layer that takes care of data location and routing in such a distributed environment. So the design does not aim at substituting the routing protocols currently used in Internet, but rather at a relatively "new" solution based on them.

The main goal of achieving an efficient protocol of routing has been reached by adding minimal information in each host of the network. The focus is mainly on efficient data-lookup methods in a large-scale distributed environment: a *distributed lookup primitive* able to locate specific data or documents in an efficient manner. Efficiency is "measured" in a low latency environment: data location has to be as fast as possible to result in acceptable performance even in the case of low latency applications.

The main goals of the project can be summarized as follows:

- simplifying routing: efficient mechanism, even using minimal information;
- efficiently content addressing: data location in a purely P2P system;

[2] *Pure* P2P means completely decentralized: the system does not rely on any centralized database, as for example as DNS does.

- topology awareness to improve end-to-end performance when delivering messages.

Finally there are various environments in which this project can be realized. The common feature of these environments is the need of a distributed/decentralized application for data exchange among multiple hosts. In addition to the most common file-sharing applications that made systems such as Napster or Gnutella famous, this project has mainly been created with two environments in mind:

- Distributed robust data storage: multiple contents can be inserted into the network merely by specifying the content name. The network itself is responsible for addressing each content, storing it in a convenient fashion. Whenever a content is requested, our smart network can easily recover it from the best location, usually that is the one closest to the user (in terms of metrics or latencies).
- Multiplayer distributed gaming networks: this is typically an example of a low delay/latency data retrieval. We could even think of completely serverless gaming network, in which each peer can act as player or as routing node. In this case it is not possible to locate where the game code runs. As players are located in a distributed physical space, the game itself runs in a distributed fashion.

1.4 Related Work

In this section we review some of the existing routing algorithms. As input all of them take a key and, as output, route a message to the node responsible for that key. The keys are strings of digits of some length. Nodes have identifiers, taken from the same space as the keys (i.e., same number of digits). Each node maintains a routing table consisting of a small subset of nodes in the system. When a node receives a query for a key for which it is not responsible, the node routes the query to the neighbor node that makes the most "progress" towards resolving the query. The notion of progress differs from algorithm to algorithm, but in general is defined in terms of some distance between the identifier of the current node and the identifier of the queried key.

1.4.1 Landmark routing

Plaxton et al. [31] developed what probably was the first routing algorithm that could be scalably used by DHTs. Although not intended for use in P2P systems, because it assumes a relatively static node population, it provides very efficient routing of lookups. The routing algorithm works by "correcting" a single digit at a time, for example: if node number 36278 receives a lookup query with key 36912, which matches its first two digits, the routing algorithm forwards the query to a node that matches the first three digits (e.g., node 36955). To do this, a node needs to have, as neighbors, nodes that match each prefix of its own identifier but differ in the next digit. For a system of n nodes, each node has on the order of $O(n)$ neighbors. As one digit is corrected each time the query is being forwarded, the routing path is at most $O(n)$ overlay (or application-level) hops. This algorithm has the additional property that if the n^2 node-to-node latencies (or "distances" according to some metric) are known, the routing tables can be chosen to minimize the expected path latency and, moreover, the latency of the overlay path between two nodes is within a constant factor of the latency of the direct underlying network path.

The Plaxton location and routing system provides several desirable properties for both routing and location:

- *Simple Fault Handling:* Because routing only requires that nodes match a certain suffix, there is potential to route around any single link or server failure by choosing another node with a similar suffix.
- *Scalable:* It is inherently decentralized, and all routing is done using locally available data. Without a point of centralization, the only possible bottleneck exists at the root node.
- *Exploiting Locality:* With a reasonably distributed namespace, resolving each additional digit of a suffix reduces the number of satisfying candidates by a factor of the ID base b (the number of nodes that satisfy a suffix with one more digit specified decreases geometrically). The path taken to the root node by the publisher or server S storing O and the path taken by client C will likely converge quickly, because the number of nodes to route to drops geometrically with each additional hop. Therefore, queries for local objects are likely to quickly run into a router with a pointer to the object's location.
- *Proportional Route Distance:* Plaxton has proven that the total network distance traveled by a message during both the location and the routing phase is proportional to the underlying network distance, assuring us that routing on the Plaxton overlay incurses a reasonable overhead.

There are, however, serious limitations to the original Plaxton scheme:

- *Global Knowledge:* To achieve a unique mapping between document identifiers and root nodes, the Plaxton scheme requires global knowledge at the time that the Plaxton mesh is constructed. This global knowledge greatly complicates the process of adding and removing nodes from the network.
- *Root Node Vulnerability*: As a location mechanism, the root node for an object is a single point of failure because it is the node that every client relies on to provide an object's location information. Whereas intermediate nodes in the location process are interchangeable, a corrupted or unreachable root node would make objects invisible to distant clients who do not meet any intermediate hops on their way to the root.
- *Lack of Ability to Adapt:* While the location mechanism exploits good locality, the Plaxton scheme lacks the ability to adapt to dynamic query patterns, such as distant hotspots. Correlated access patterns to objects are not exploited, and potential trouble spots are not corrected before they cause overload or cause congestion problems in a wide area. Similarly, the static nature of the Plaxton mesh means that insertions could only be handled by using global knowledge to recompute the function for mapping objects to root nodes.

1.4.2 Tapestry

Tapestry [40] uses a variant of the Plaxton algorithm. The modifications ensure that the design, originally intended for static environments, can adapt to a dynamic node population. The modifications are too complicated to describe them in this short review; for details, the reader is refered to [40]. However, the algorithm maintains the

properties of having $O(\log n)$ neighbors and routing with path lengths of $O(\log n)$ hops.

1.4.3 Pastry

In Pastry [35], nodes are responsible for keys that are the numerically closest (with the key-space considered as a circle). Keys and node IDs are from the same name space. The neighbors consist of a Leaf Set L which is the set of $|L|$ closest nodes (half larger, half smaller). Correct, but not necessarily efficient, routing can be achieved with this leaf set. To achieve more efficient routing, Pastry has another set of neighbors spread out in the key space (in a manner not further described here but somehow similar to the Plaxton approach). Routing consists of forwarding the query to the neighboring node that has the longest shared prefix with the key (and, in the case of ties, to the node with the identifier that is numerically closest to the key). Pastry has $O(\log n)$ neighbors and routes within $O(\log n)$ hops.

1.4.4 Chord

Chord [37] also uses a one-dimensional circular key space. The node responsible for the key is the node whose identifier most closely follows the key (numerically); that node is called the key's successor. Chord maintains two sets of neighbors. Each node has a *successor list* of k nodes that immediately follow it in the key space. Routing correctness is achieved with these lists. Routing efficiency is achieved with the *finger list* of $O(log\, n)$ nodes spaced exponentially around the key space. Routing consists of forwarding to the node closest, but not past, the key; path lengths are $O(log\, n)$ hops.

The basic idea of this routing protocol is known as *Interval Routing* [15]. It has the interesting properties that any node in the network can be reached in a logarithmic number of steps. The drawback is that node IDs cannot be random, they must have a well-defined assignment: in this case a circular ID space, as shown in Figure 1.3, is necessary.

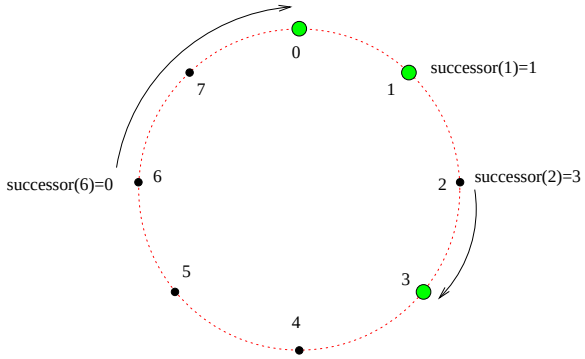


Figure 1.3: Circular ID space and routing in Chords.

A hash function maps nodes onto an m-bit circular identifier space. It defines also a successor of a node u as the node which has the smallest ID greater or equal to u. To implement the successor function, all nodes maintain an m-entry routing table called the *finger table*. This table stores information about other nodes in the system; each

entry contains a node identifier and its network address (consisting of an IP address and a port number). The number of unique entries in the finger table is $O(\log n)$. The finger table can also be thought of in terms of m identifier intervals corresponding to the m entries in the table. Each interval has a logarithmicaly increasing size.

Each node also maintains a pointer to its immediate predecessor. For symmetry, we also define the corresponding immediate successor (identical to the first entry in the finger table). In total, each node must maintain a finger table entry for up to $O(\log n)$ other nodes. As each node maintains information about only a small subset of the nodes in the system, evaluating the successor function requires communication between nodes at each step of the protocol. The search for a node moves progressively closer to identifying the successor with each step.

1.4.5 CAN

CAN [32] stands for Content Addressable Network. It is another distributed infrastructure based on hash-table functionality. It differs from the Chords approach in the ID structure. CAN chooses its keys from a toroidal space of d dimensions. Its approach consists of partitioning this virtual space and assigning a small distinct area to each node, completely tiling the entire space. So a single node is responsible for a certain bounded region.

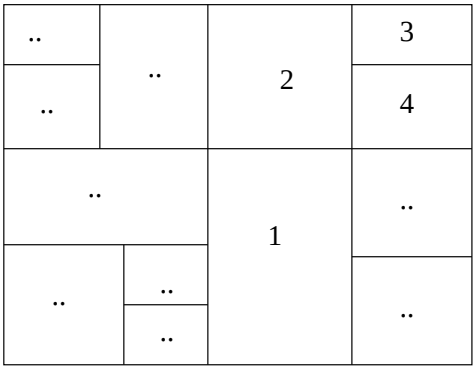


Figure 1.4: ID space (2 dimensions) partitioning in CAN approach.

Intuitively, routing in a CAN works by following the straight line path through the Cartesian space from source to destination coordinates. A CAN node maintains a coordinate routing table that contains the IP address and virtual coordinate zone of each of its immediate neighbors in the coordinate space. In a d-dimensional coordinate space, two nodes are neighbors if their coordinate spans overlap along $d-1$ dimensions and abut along one dimension. Using its neighbor coordinate set, a node routes a message towards its destination by simple greedy forwarding to the neighbor with coordinates closest to the destination coordinates. To allow the CAN to grow incrementally, a new node that joins the system must be allocated its own portion of the coordinate space. This is done by an existing node splitting its allocated zone in half, retaining half and handing the other half to the new node.

CAN has a different performance profile than the other algorithms; nodes have $O(d)$ neighbors and path lengths are $O\left(dn^{\frac{1}{d}}\right)$ hops. Note, however, that when

$d = log\, n$, CAN has $O(\log n)$ neighbors and path lengths $O(\log n)$ like the other algorithms.

1.4.6 Bloom Filters

Bloom filters[34] are compact data structures for probabilistic representation of a set in order to support membership queries (i.e. queries that ask: Is element X in set Y?). This compact representation is the payoff for allowing a small rate of *false positives* answers in membership queries; that is, queries might incorrectly indicate an element being a member of the set.

A Bloom filter is a bit-field of a fixed width w and a number of associated independent hash functions. Each hash function maps from an object ID to an integer in $[0, w-1]$. Inserting an object in a Bloom filter involves applying the hash functions to the object and setting the bits in the bit-field associated with the hash function results. A bit that was already set, remains set. To see if a bloom filter contains an object, one applies the hash functions to the objects and checks the filter for the appropriate bits . If any of those bits are unset, the Bloom filter definitely does not contain the object. If all of the bits are set, the Bloom filter may contain the object. The false positive rate of a Bloom filter is a well-defined function of its width, the number of hash functions and the number of objects inserted. After having inserted n keys into a filter of size m using k hash functions, the probability that a specific bit is still 0 is:

$$p_0 = \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - e^{-\frac{kn}{m}} \tag{1.1}$$

The assumption is to have perfect hash functions that leads to an homogeneous occupation of the space $\{1...m\}$. In practice, good results have been achieved using *MD5* and other cryptographic hash functions[34].

Hence, the probability of a false positive (the probability that all k bits have been previously set) is:

$$p_{err} = (1 - p_0) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \tag{1.2}$$

Even if in practice only a small number (k) of hash functions are used, p_{err} in 1.2 is minimized for $k = \frac{m}{n}\, ln\, 2$ hash functions. An important remark is that the average query cost is inversely proportional to the Bloom filter width. So it can be kept as close to optimal as desired by increasing the width (up to a certain threshold).

An attenuated Bloom filter [25] of depth D can be viewed as an array of D normal Bloom filters. In a storage context, the first Bloom filter can be a record of the documents contained locally on the current node. The ith Bloom filter is the merger of all of the Bloom filters for all of the nodes at a distance i on any path from the current node. If each node has Bloom filters for its (physical) neighbors, messages can be routed easily towards the node who is supposed to have the document. One drawback is that this design only works well in small neighborhoods.

1.4.7 OceanStore

OceanStore [25] is a highly distributed storage architecture providing continuous access to persistent information. It mainly uses attenuated Bloom filters and Plaxton

routing, firstly to create a consistent distributed directory of objects and secondly to route object location requests.

It uses a double routing mechanism:

1. Attenuated Bloom filters as primary step; this allows the queried content to be retrived efficiently with high probability.

2. Plaxton routing whenever the first algorithm fails.

The first mechanism can fail because Bloom filters can sometimes be misleading owing to false positives. In OceanStore the misleading behavior happens when attenuated Bloom filters indicate that two or more routes can lead to the object requested. This conflict cannot be avoided entirely, but it is possible to lower the probability of misleading behavior by choosing appropriate parameters for the Bloom filter such as number of hashing functions or filter size (See section 1.4.6). Let us look at a simple example of the OceanStore object location mechanism (Figure 1.5).

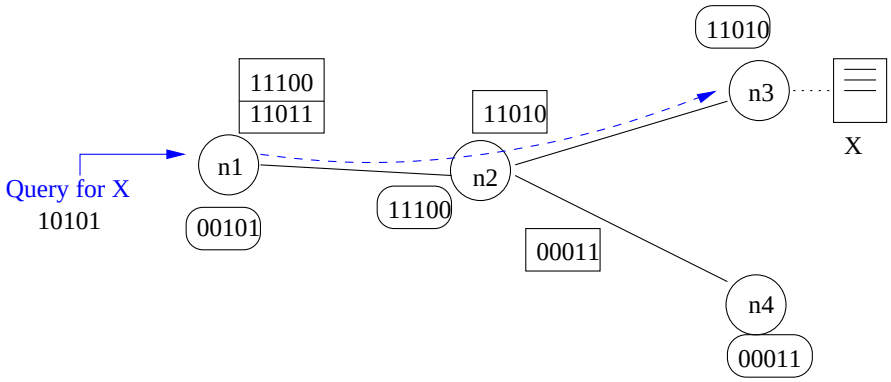


Figure 1.5: Bloom filters in OceanStore.

The replica at n_1 is looking for object X, whose document ID hashes to bits 0, 1, and 3. The local Bloom filter for n_1 (rounded box) shows that it does not have the object, but its neighbor filter (unrounded box) for n_2 indicates that n_2 might be an intermediate node en route to the object. The query moves to n_2, whose Bloom filter indicates that it does not have the document locally, that its neighbor n_4 does not have it either, but that its neighbor n_3 might. The query is forwarded to n_3, which verifies that it has the object.

Attenuated Bloom filters are a good solution either for keeping track of replicas or to manage caches in distributed networks. However they need to be carefully tuned to minimize the number of *false positive* answers.

1.4.8 HyperCast

The HyperCast software [27] builds and maintains logical overlay networks between applications, and supports data transmission between applications in the overlay. Applications self-organize into a logical overlay network, and transfer data along the edges of the overlay network using unicast transport services. Each application communicates only with its neighbors in the overlay network. Using the overlay, services

for one-to-many transmissions ("multicast") and many-to-one transmissions ("incast" or also known as "concast") can easily be implemented.

HyperCast builds topologies graphs with well-known properties. Currently, HyperCast can build two types of overlay network topologies:

- logical hypercubes, and
- logical Delaunay(HyperCast) triangulations.

An important common property to both topologies is that once the topology is established, packet forwarding in these overlay networks can be performed without the need for a routing protocol, just based on location.

The *logical hypercube* overlay network topology organizes the applications into a logical n-dimensional hypercube. Each node is identified by a label (e.g., "010"), which indicates the position of the node in the logical hypercube. In an overlay network with n nodes, the lowest n positions of a hypercube are occupied (according to a Gray ordering [18]) as shown in Figure 1.6.

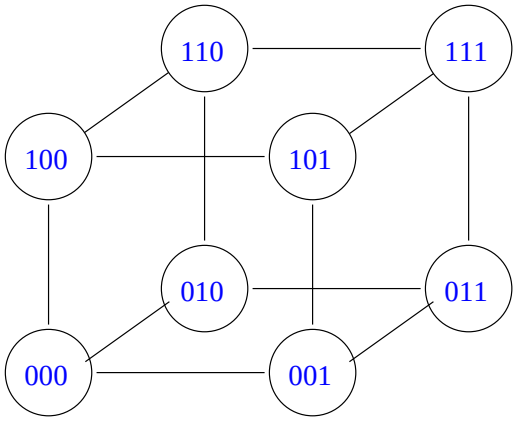


Figure 1.6: Hypercube in the HyperCast design.

One advantage of using a hypercube is that each node has only $\lceil log\, n \rceil$ neighbors, where n is the total number of nodes. Also, the longest route in the hypercube is $\lceil log\, n \rceil$. A disadvantage of hypercubes in the HyperCast design is that the physical network infrastructure is completely ignored. Another disadvantage is that the hypercube construction must be done sequentially, i.e. one node at a time. Therefore, it can take a long time for large groups before the overlay network is completed. Also, the departure of a single node may require substantial changes to the overlay topology.

The *Delaunay triangulation* is a special type of triangulation. Its main characteristic is that for each circumscribing circle of a triangle formed by three adjacent nodes, no other node of the graph is inside the circle. Each node in a Delaunay triangulation has (x, y) coordinates depicting a point in the plane. In Figure 1.7, we show a Delaunay triangulation of five nodes and the circumscribing circles of some of its triangles.

An advantage of the Delaunay triangulation overlay network topology is that it can be constructed in a distributed fashion. Therefore, Delaunay triangulations can

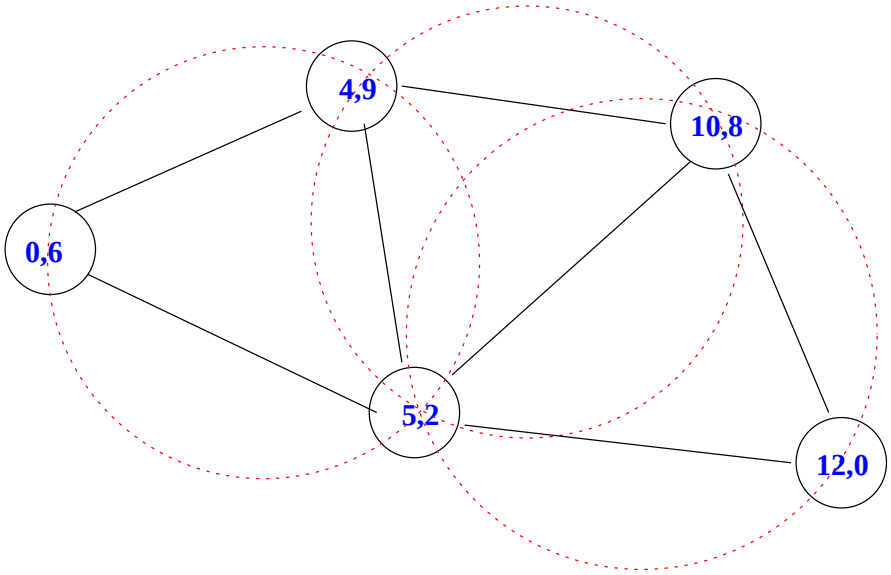


Figure 1.7: Delaunary triangulation in the HyperCast design.

be built very quickly. In a Delaunay triangulation, each node has an average of six neighbors. In the worst-case, however, the number of neighbors is $n - 1$, where n is the total number of nodes.

If the coordinates of a node in the Delaunay triangulation reflect the node's geographical location, then nodes in the overlay network are likely to be neighbors if their geographical location is close. However, Delaunay triangulations in the HyperCast design are not aware of the underlying network infrastructure. It also requires a mechanism to find neighbors by ID for the construction, which is (IMHO) impossible to do in a distributed fashion, before the network has been built.

Chapter 2

The overlay network

2.1 Layering

In order to separate the functionality of a pre-existing routing layer from the new advantages of our protocol, the division into two layers is necessary (see Figure 2.1). The pre-existing network layer, which we refer to as *underlying layer*, is supposed to have efficient routing capabilities; that means it is supposed to be able to deliver any message from a source address to a destination address efficiently. This is the case for the IP network layer: its diffusion throughout the entire Internet is an optimum environment for building up our data location protocol.

The core of the project is what we refer to as *Hypercube routing protocol*, and it is located in a higher layer built directly on top of previous layer. The efficiency of this layer is measured as the capacity to find, given an ID request, the best destination address to which the request is to be routed. As the system is completely P2P, this decision must be done using only local routing information.

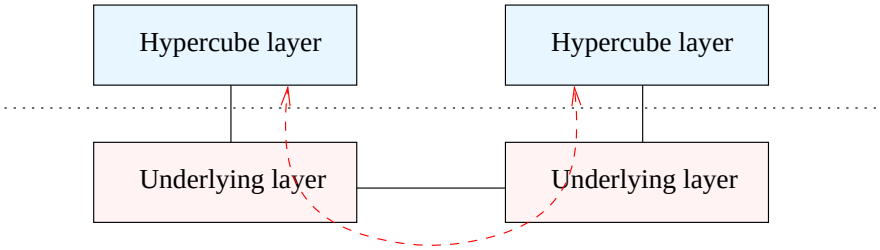


Figure 2.1: Layer division.

The *overlay* network turns out to be an abstract set of hosts that play an active role in the data location process. As shown in Figure 2.2, the hosts in the overlay network are a subset of the total number of hosts present.

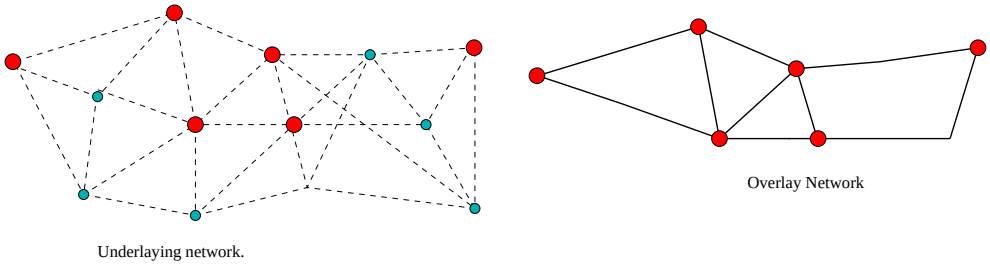


Figure 2.2: Overlay network example.

Note how end-to-end paths between two hosts can be different in the two layers. In particular a single edge in the overlay graph can include multiple edges in the corresponding underlying one. This lack of path correspondence between the two layers makes the routing process more difficult: a route retained as optimal in the overlay graph could be not optimal in the underlying graph.

This problem is one of the most significant in the current implementations of DHTs. On the one hand, most designs take forwarding decisions at each hop, based on the neighborhood relationship in the overlay network. Depending on how the

overlay network has been built, this can lead to "catastrophic" results. The neighbor of an overlay node can potentially be located at the opposite side of the planet. On the other hand, routing paths between two hosts that are physically close, can include jump to hosts far away. As a consequence, the protocol performance suffers in terms of routing-path length and total end-to-end latency. This explains why we decided to pay great attention to *locality preservation* during the routing process.

The interaction of the two layers is perhaps the greatest novelty of our approach. In fact the overlay routing layer is built using the information about the local topology taken from the underlying layer. In other words, every time a new node joins the network, some information must be accessed locally to determine the best point for fitting the new node into the hypercube structure. We assume latency to be a good metric, but we do not restrict the access of latency information to any particular protocol; we just assume this measurement is possible somehow.

2.2 Exploiting network proximity

While there are algorithmic similarities among the proposed DHTs, one important distinction lies in the approach they use to consider and exploit proximity in the underlying Internet. Considering network proximity is important, because otherwise a lookup to a *key-value* pair that is stored on a nearby node may be routed through nodes that are far away in the network (on different continents in the worst case). Three basic approaches have been suggested for exploiting proximity in these DHT protocols:

- *Geographic Layout* The node IDs are assigned in a manner that ensures that nodes that are close in the network topology are close in the node ID space (e.g. GPS coordinates).
- *Proximity Routing* The routing tables are built without taking network proximity into account, but at each hop the routing algorithm chooses a nearby node from those in the routing table. Routing strikes a balance between making progress towards the destination in the node ID space and choosing the closest routing table entry according to the network proximity.
- *Proximity Neighbor Selection* Routing table construction takes network proximity into account. Routing table entries are chosen such that they refer to nodes that are nearby in the network topology among all live nodes with appropriate node IDs. The distance traveled by messages can be minimized without increasing the number of routing hops.

Proximity neighbor selection is used in Tapestry [40] and Pastry [35]. The basic Chord [37] and CAN [32] protocols do not consider network proximity at all. However, geographic layout and proximity routing have been considered for CAN, and geographic layout and proximity neighbor selection are currently being considered for use in Chord. HyperCast [27] with Delaunay triangulation is actually based on the geographical layout approach.

2.2.1 Geographic Layout

Geographic layout was explored as one technique to improve the routing performance in CAN. The technique attempts to map the d-dimensional space onto the physical

network such that nodes that are neighbors in the d-dimensional space (and therefore in each other's routing tables) are close in the physical network. In one implementation, nodes measure the RTT between themselves and a set of landmark servers to compute the coordinates of a node in the CAN space. This technique can achieve good performance but has the disadvantage that it is not fully self-organizing; it requires a set of well-known landmark servers. In addition, it may cause significant imbalances in the distribution of nodes in the CAN space, leading to hot-spots.

When considering the use of this method in Chord, Tapestry and Pastry, additional problems arise. While a geographic layout provides network locality in the routing, it sacrifices the diversity of neighboring nodes in the node ID space, which has consequences for failure resilience and availability of replicated key/value pairs. Both Chord and Pastry have the property that the integrity of their routing fabric is disrupted when an entire leaf set or a successor set fails. Likewise, both protocols replicate key-value pairs on neighboring nodes in the name-space for fault tolerance. With a proximity-based node ID assignment, neighboring nodes, because of their proximity, are more likely to suffer correlated failures.

In other designs, such as in HyperCast [27] or in various ad-hoc mobile networks [26], geographic layout is fully adopted by the use of GPS devices. This assumption cannot be justified in the distributed environment of the Internet.

2.2.2 Proximity Routing

Proximity routing was first proposed in CAN [32]. It involves no changes to routing table construction and maintenance because routing tables are built without taking network proximity into account. But each node measures the RTT to each neighbor (routing table entry) and forwards messages to the neighbor with the maximum ratio of progress in the d-dimensional space to RTT.

As the number of neighbors is small ($2d$ on average) and neighbors are spread randomly over the network topology, the distance to the nearest neighbor is likely to be significantly larger than the distance to the nearest node in the overlay. Additionally, this approach trades off the number of hops in the path against the network distance traversed in each hop: it can ever increase the number of hops. Because of these limitations the technique is less effective than geographical layout.

Proximity routing has also been used in a version of Chord [37]. Here, a small number of nodes are maintained in each finger table entry rather than only one, and a message is forwarded to the topologically closest node among those entries whose node ID is closer to the message's key. As all entries are chosen from a specific region of the ID space, the expected topological distance to the nearest of the entries is likely to be much larger than the distance to the nearest node in the overlay. Furthermore, it appears that all these entries have to be maintained for this technique to be effective because not all entries can be used for all keys. This increases the overhead of node joins and the size of routing tables.

In conclusion, proximity routing offers some improvement in routing performance, but this improvement is limited by the fact that a small number of nodes sampled from specific portions of the node ID space are not likely to be among the nodes that are closest in the network topology.

2.2.3 Proximity Neighbor Selection

The locality properties of Tapestry [40] and Pastry [35] derive from mechanisms to build routing tables that take network proximity into account. They attempt to minimize the distance, according to the proximity metric, to each one of the nodes that appear in a node's routing table, subject to the constraints imposed on node ID prefixes. Pastry ensures the following invariant for each node's routing table:

> Proximity invariant: *Each entry in a node's routing table refers to a node that is near, according to the proximity metric, among all live Pastry nodes with the appropriate node ID prefix.*

As a result of the proximity invariant, a message is normally forwarded in each routing step to a nearby node, according to the proximity metric, among all nodes whose node ID shares a longer prefix with the key. Moreover, the expected distance traveled in each consecutive routing step increases exponentially, because the density of nodes decreases exponentially with the length of the prefix match. From this property, one can derive two distinct properties of Pastry with respect to network locality:

- *Total distance traveled* - The expected distance of the latest routing step tends to dominate the total distance traveled by a message. As a result, the average total distance traveled by a message exceeds the distance between source and destination node only by a small constant value.
- *Local route convergence* - The paths of two Pastry messages sent from nearby nodes with identical keys tend to converge in the proximity space at a node near the source nodes. To see this, observe that in each consecutive routing step, the messages travel exponentially larger distances towards an exponentially shrinking set of nodes. Thus, the probability of a route convergence increases in each step, even if earlier (smaller) routing steps have moved the messages farther apart. This result is of significance for caching applications layered on Pastry.

The routing algorithms in Pastry and Tapestry claim that they allow effective proximity neighbor selection because there is freedom to choose nearby routing table entries from among a large set of nodes.

CAN also proposed a limited form of proximity neighbor selection in which several nodes are assigned to the same zone in the d-dimensional space. Each node periodically gets a list of the nodes in a neighboring zone and measures the RTT to each of them. The node with the lowest RTT is chosen as the neighbor for that zone. This technique is less effective than those used in Tapestry and Pastry because each routing table entry is chosen from a small set of nodes.

2.3 Overlay structure

Each node of our overlay network is assigned an unique identifier (ID). IDs are not scalar, they are vectors in a d-dimensional space.

This space has the form of an hyper-torus, that means, it is wrapped on itself. IDs can be considered as points in this space with d coordinates:

$$ID_1 = (x_1, x_2, \ldots, x_d)$$

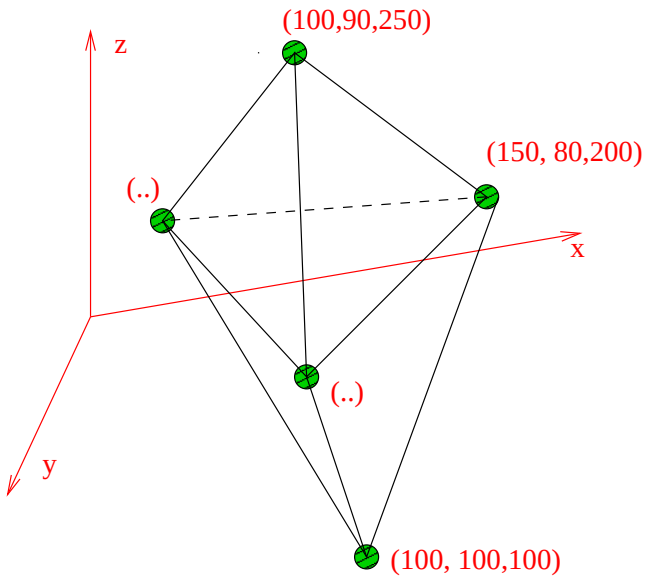


Figure 2.3: Example of ID positions in a threes-dimensional space.

Each coordinate is expressed with a certain number of bits (b), so that

$$x_i \in \left[0, 2^{b-1}\right]$$

As the space is wrapped on itself, each coordinate is circular. As shown in Figure 2.3, this network structure can be easily seen from a geometrical point of view if the number of dimensions is less than four (if you consider the space not wrapped).

As an abstraction, when we say that a node has a certain "position", we mean that its ID has a certain position in the virtual ID space. In the same way, when speaking about the distance between two nodes the distance between their IDs is meant.

2.3.1 Neighbor placement

Two possible approaches are possible for the neighborhood relationship among nodes. In order to explain this two approaches well, it is worth considering a two-dimensional (2D) ID space. The two cases are shown in Figure 2.4:

- Case (a) is the simplest neighborhood relationship: using the node itself as origin of a d-dimensional Cartesian system, neighbors are placed along the axes of this system. The number of neighbors in this case is $2d$.

- In case (b), neighbors are placed by partitioning the space in 2^d *quadrants*, so the number of neighbors is 2^d.

Using this neighborhood relationship, it is possible and useful to extend the word *"quadrant"* to a d-dimensional space as well. Figure 2.5 shows a generalization to a three-dimensional (3D) space:

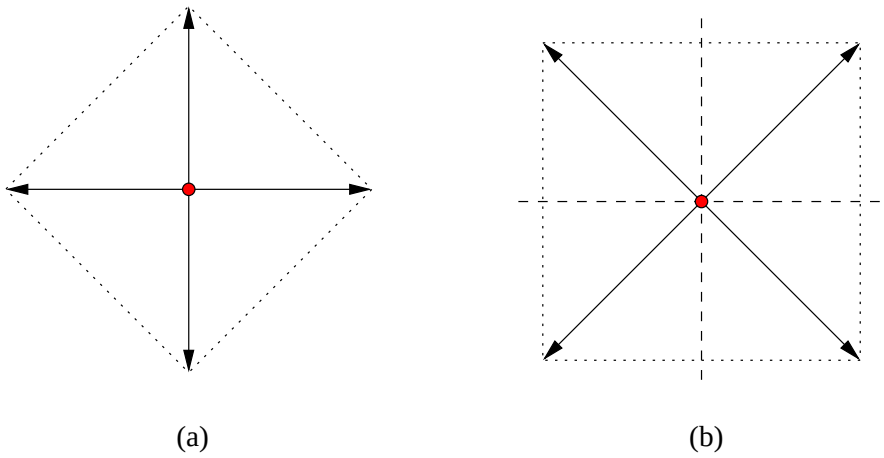


Figure 2.4: Possible neighbors placement in a 2D ID space.

- In case (a), neighbors of a certain node are placed in the vertex of a kind of octahedron centered around the node itself;
- In case (b), the neighbors of a node are placed in the vertex of a cube whose center is the node itself. Here you can see that the number of neighbors is 2^d.

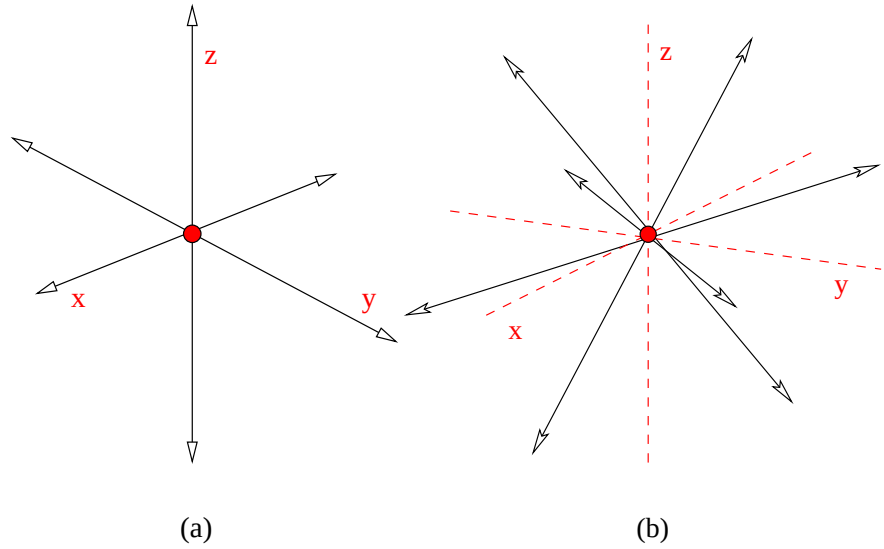


Figure 2.5: Possible neighborhood relationships in a 3D world.

In our design, the second approach has been chosen because

- it offers good flexibility in terms of node positioning: neighbors of node are not obliged to be on the axis extendng from that node but can be placed anywhere in each quadrant;

→ quadrants can be used as "directions" during the routing process, so that the routing path is as close as possible to a line and does not present strange loops.

Figure 2.6 shows the space division induced by this choice for the neighbors' placement around a certain node. Note how the space has been partitioned in "octants" (for the sake of simplicity, we will always call them "quadrants" when the space has d dimensions).

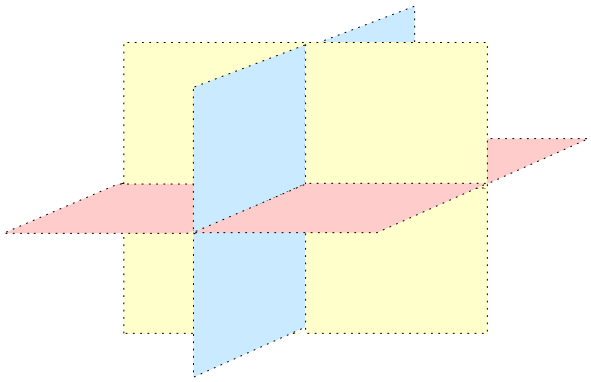


Figure 2.6: 3D example. Given a certain node as the origin, there are 2^3 regions where neighbors can be placed.

2.3.2 Distance notion

Given the d-dimensional ID space, if the space were not wrapped, the Cartesian distance between two points would be:

$$\underline{A} = (x_1, \ldots, x_d)$$

$$\underline{B} = (y_1, \ldots, y_d)$$

$$d_{AB} = \|\underline{A} - \underline{B}\| = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots + (x_d - y_d)^2}$$

The fact that the space is wrapped leads to some modifications (see Figure 2.7).

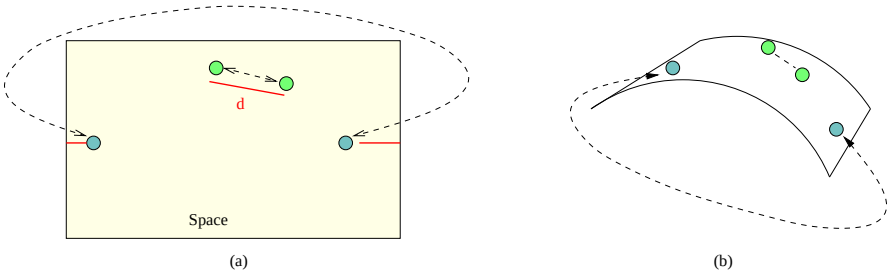


Figure 2.7: Scalar distance in a 2D wrapped space.

It is still possible to give a scalar distance definition; denoting by $M_x = 2^b$ (where b is the number of bits used for each coordinate) the maximum value of each coordinate x, the new distance definition is

$$d_{AB} = \sqrt{\min\left[(x_1 - y_1)^2, (M_x - (x_1 - y_1))^2\right] + \ldots + \min\left[(x_d - y_d)^2, (M_x - (x_d - y_d))^2\right]} \tag{2.1}$$

It is possible to provide also a kind of non-scalar distance definition, if we define the concept of *"distance in a quadrant"*. As the space is wrapped, the relationship between two nodes can change depending on the observer's angle of observation.

As shown in Figure 2.8, the position of node B with respect to A is not well defined. If an observer is placed in A and looks at B, the distance from B to A would appear different, depending on in which direction the observer is looking. In the example in Figure 2.8 we have

$$\underline{A} = (x_A, y_A) \quad \underline{B} = (x_B, y_B)$$

$$\underline{d_{AB}} = (d_1, d_2, d_3, d_4)$$

$$d_{AB} \Rightarrow \begin{cases} d_1 = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2} \\ d_2 = \sqrt{(x_A - x_B)^2 + (M_x - (y_A - y_B))^2} \\ d_3 = \sqrt{(M_x - (x_A - x_B))^2 + (M_x - (y_A - y_B))^2} \\ d_4 = \sqrt{(M_x - (x_A - x_B))^2 + (y_A - y_B)^2} \end{cases}$$

$$M_x = 2^b - 1 \Rightarrow \text{maximum for } x/y \text{ coordinate}$$

From this example it is easy to generalize to a general d-dimensional space.

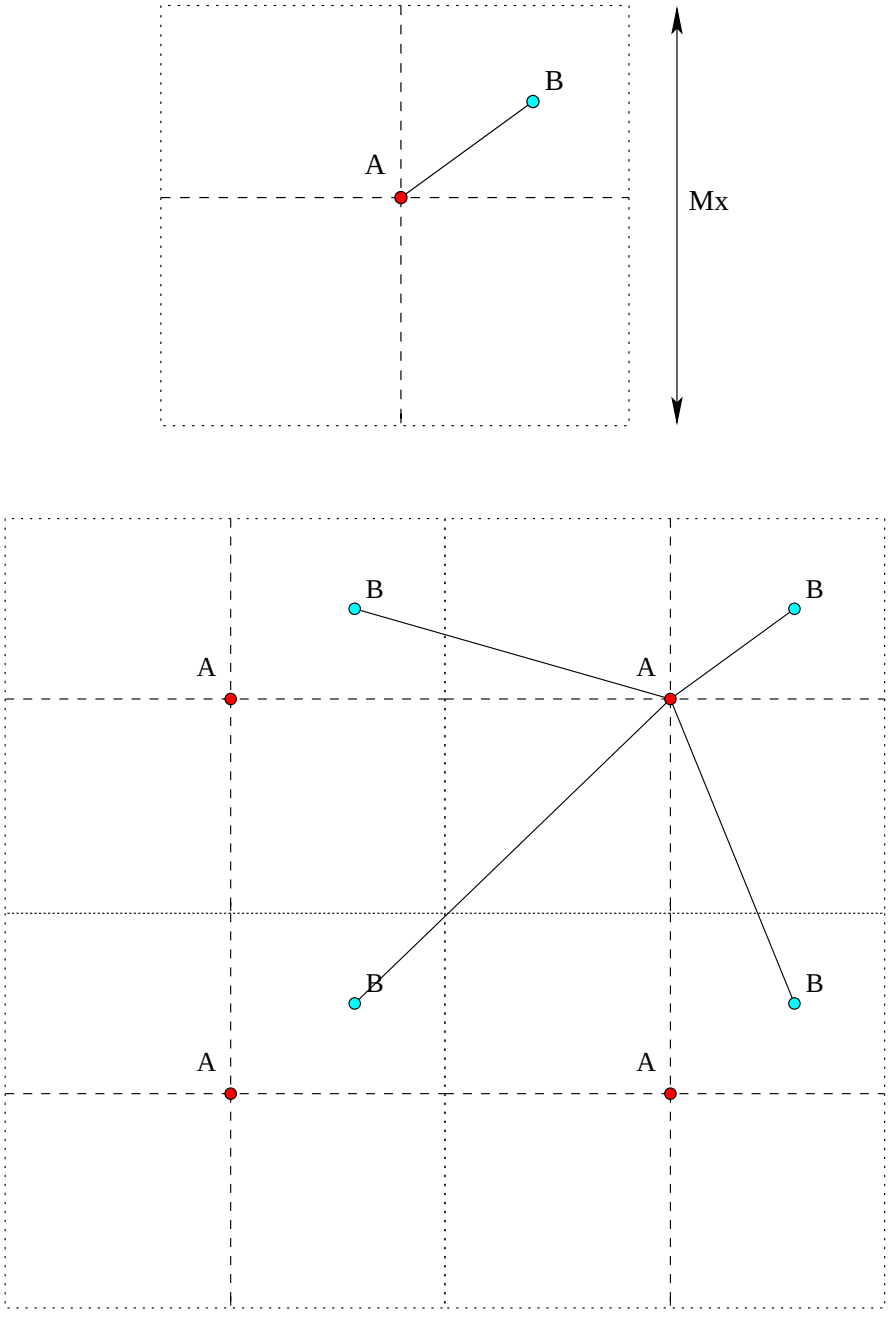


Figure 2.8: Simple distance (above) and extended distance concept in a wrapped space (below).

2.3.3 Links

Once that the quadrants, a neighborhood relationship and the distance notion have been defined, we describe how nodes set up links to each other. The criterium wheter to establish links is the notion of *closeness* in the virtual ID space.

In our design two main link rules stand:

- Each node must have at least one link per quadrant. That means that quadrants without any link are not allowed.
- Links are allowed to be "one way" as explained below.

The closeness criterium used to establish links is as follows

> *"Each node has a link in each quadrant to the closest neighbor in that quadrant."*

The relationship of closeness is in terms of scalar distance as defined in the equation 2.1.

The "one way" properties of links are due to the inner nature of quadrants: given two nodes, the relationship "to be the closest in a certain quadrant" is not reciprocal. Figure 2.9 can clarify this concept.

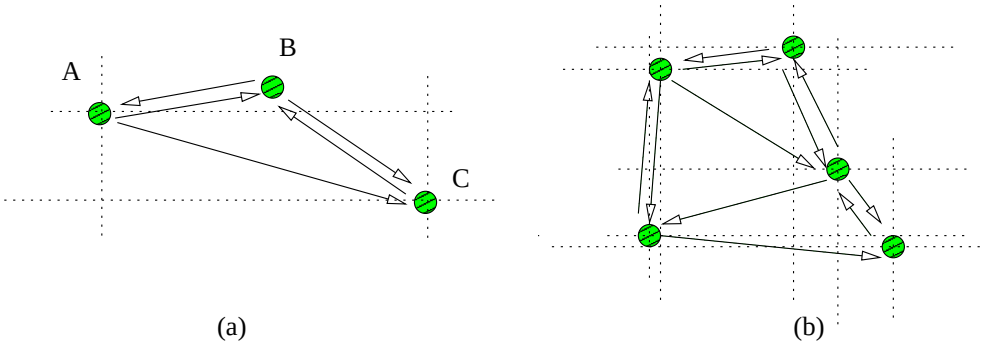


Figure 2.9: Example of links between nodes in a 2D space.

The link between A and C is a "one way" link because

- considering the south-east quadrant around A, C is the node closest to A in that quadrant;
- considering the north-west quadrant around C, B is the node closest to C in that quadrant;

This approach combines a relative simplicity of the joining protocol with the capacity of keeping the message forwarding among nodes efficient. In fact this structure assures that at each step of the routing process, messages are forwarded to the closest neighbor in a certain direction. Routing remains efficient, even in the case of simple greedy forwarding decisions.

Moreover, in order to avoid the possibility of some nodes remaining isolated, we "force" a certain symmetry with a new rule:

> *"Whenever a node B is referenced by A, B is obliged to reference A as well."*

This new rule breaks the "one way" properties and introduces "symmetrical" links with a preferential direction. It means that each link can be walked in both directions, but one direction can be more weighted and more important than the other.

Nodes are also allowed to keep pointers to more than one neighbor in each quadrant. One of these pointers always represents the closest node in that quadrant and is the favorite "next hop" when forwarding messages, whereas the others have a redundancy role, so that they can be used when the first one fails.

Chapter 3

Overlay Network Construction

3.1 Hypercube Layer Integration

The interface between the hypercube layer and the lower layer is defined through some primitives such as

- ***sendMessage*** *([Destination Address], [Message])* : sends a message to a specified destination address;
- ***measureDelay*** *([Destination Address])* : returns the delay between the current node and the specified destination;
- ***measureHops*** *([Destination Address])* : returns the path length between the current node and the specified destination (in number of hops);
- ***processMessage*** *([Message])* : processes a message arrived from the lower layer;
- ***getAddress*** *()* : prompts lower layer for its own address, as the higher layer normally ignores low-level information;

3.2 Join protocol

The need of a specific joining phase is justified by the critical impact of the ID assignment on the overall behavior of our protocol. The ID assignment takes place during the phase of overlay network building. Assuming that somewhere a small hypercube network is already in place, any new node that wants to join should set up links to some nodes of the existing network. At this moment, it should be able to fix its own ID in order to fit into the global hypercube structure, built in the ID space. At the same time, some nodes of the existing network should set up links to the new node, so that it does not remain isolated from the network. From this brief it is easy to observe, how two main phases can be identified when building the hypercube structure from scratch:

- Firstly, a small network must be constructed. Having some nodes, opportunely placed in the global ID space, helps not only in placing future nodes but also provides a better and more efficient occupation of the ID space. We will call these few nodes *"landmarks"*, because they serve as points of reference for any new joining node. In the physical world, landmarks should be placed in areas that are sufficiently far apart. This assumption is not essential for the funciton-ing of our protocol, butit is strongly recommended: nodes that are reasonably distributed in the physical space remain reasonably distributed also in the ID space. The allocation of the ID space to new nodes is important, and attention should be paid so that no portions of this space are wasted (Section 3.3).
- Secondly, any new node that wants to join should know an "entry point", that is, the address of any node already belonging to the network. Starting from this entry point, the new node searches that node in the network which is closest to itself. Then it tries to join up with it and the neighbors of the node found (Section 3.4).

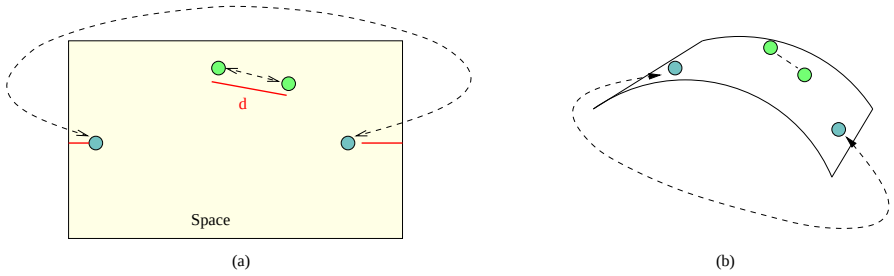


Figure 3.1: Distance example in 2D wrapping space.

3.3 Landmark placement

Assuming that landmark position in the physical world is a "good" distribution among areas covered by the overlay network, we want to place all landmarks in the ID space so that their position reflects more or less this distribution. The only rule, on which the placement is based, is that the distance between two landmarks in the ID space should be proportional to distance metric measured between them in the physical world.

3.3.1 "Spring forces" algorithm

The first phase of the joining protocol, the landmark placement, consist of an algorithm commonly used in graph visualization onto a 2D surface [14]. In the beginning all landmarks are placed randomly throughout the ID space. After the latency between two nodes has been measured, we intend to move these two nodes so that their distance is proportional to the measured latency by a factor applicable to all nodes. In other words, nodes are connected by springs whose strengths are inversely proportional to their lower-layer distance. The system is then released until a stable state is reached.

Forces are applied to each node in order to move it in the right direction. The intensity of each force is proportional to the difference between the current distance between two nodes and the distance they should have.

For example, given two nodes A and B (Figure 3.2), the measured latency between them is L, and each has its own ID. Each ID is a point in an n-dimensional coordinate space:

$$\overrightarrow{ID_A} = [x_1, x_2, .., x_n] \text{ and } \overrightarrow{ID_B} = [y_1, y_2, .., y_n]$$

The Euclidean distance between them is

$$D = \left\| \overrightarrow{ID_A} - \overrightarrow{ID_B} \right\|$$

$$D = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots + (x_n - y_n)^2}$$

but as we are working in a wrapped space (Figure 3.1), where $M_x = 2^b$ is the maximum value for each coordinate x:

$$D = \sqrt{\min\left[(x_1 - y_1)^2, (M_x - (x_1 - y_1))^2\right] + \ldots + \min\left[(x_n - y_n)^2, (M_x - (x_n - y_n))^2\right]} \tag{3.1}$$

By fixing a common constant c for all nodes, we can calculate the two forces to be applied to A and B:

$$\left\|\overrightarrow{F_A}\right\| = c \cdot L - D$$

$$\overrightarrow{F_A} = (c \cdot L - D) \cdot \frac{\overrightarrow{ID_A} - \overrightarrow{ID_B}}{\left\|\overrightarrow{ID_A} - \overrightarrow{ID_B}\right\|} \cdot \Gamma = \left(\frac{c \cdot L}{D} - 1\right) \cdot \left(\overrightarrow{ID_A} - \overrightarrow{ID_B}\right) \cdot \Gamma$$

$$\overrightarrow{F_B} = -\overrightarrow{F_A}$$

Γ is a diagonal matrix so that $\|\Gamma\| = 1$ and

$$\Gamma = \begin{bmatrix} s_1 = \pm 1 & 0 & \ldots & 0 \\ 0 & s_2 = \pm 1 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & s_n = \pm 1 \end{bmatrix},$$

with

$$s_i = sign\left[(x_i - y_i)^2 - (M_x - (x_i - y_i))^2\right]$$

$$s_i = \begin{cases} +1 & \text{if } \min\left[(x_i - y_i)^2, (M_x - (x_i - y_i))^2\right] = (x_i - y_i)^2 \\ -1 & \text{if } \min\left[(x_i - y_i)^2, (M_x - (x_i - y_i))^2\right] = (M_x - (x_i - y_i))^2 \end{cases}$$

Once the forces have been calculated, each landmark is moved a little bit into the direction that results from the sum of all forces applied to that node (see Figure 3.2).

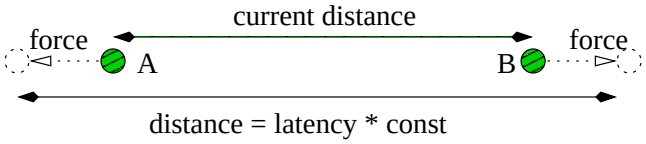


Figure 3.2: Example of the "Spring forces" algorithm.

We borrowed our approach from "Simulated Annealing" [24]. Simulated annealing is a Monte Carlo approach for minimizing such multivariate functions. The term simulated annealing derives from the roughly analogous physical process of heating and then slowly cooling a substance to obtain a strong crystalline structure. In simulations, a minimum of the cost function corresponds to this ground state of the substance. The idea is that there is a cost function H (in physical terms, a Hamiltonian) which associates a cost with a state of the system, a "temperature"

T, and various ways to change the state of the system. The algorithm works by iteratively proposing changes and either accepting or rejecting each change. Having proposed a change we may evaluate the change δH in H. The proposed change can be accepted or rejected by a certain criterion: if the cost function decreases ($\delta H < 0$) the change is accepted unconditionally; otherwise it is accepted but only with a certain probability, which is a constant value in our case, and $\exp\left(-\frac{\delta H}{T}\right)$ for the original simulated annealing algorithm.

Care has been taken to make the algorithm converge consistently and quickly:

- nodes are moved one by one in round-robin fashion; moving all nodes together leads to a great instability, and convergence is never reached;
- at each step only a little part of the force is applied, this avoids that nodes are moved too fast and improves convergence;
- decreasing the partial intensity of the forces exponentially at each step helps convergence because moves tends to become more precise;
- at the beginning a fixed number of iterations are performed to help "hill climbing". In fact in the earlier phase, when the system is far away from its stable state, this solution helps prevent oscillations and moving from local to global minima;
- the algorithm has a statistical probability to stop: it continues as long as the general "energy" decreases, otherwise it stops with a certain probability. This solution further helps small "hill climbing".
- if the energy increases too much, then algorithm is reinitialized, and nodes are again placed randomly.

This algorithm converges to a good landmark placement in an finite time. Figure 3.3 shows a screen-shot of landmark placement in a 2D space. Each node is marked with

`[node number][coordinates][force],`

so that for example $16\,(426, 386)\,(-157, -114)$ means that the node number is 16, it has coordinates $(426, 386)$ in the 2D ID space, and the resulting force on it is the vector $(-157, -114)$.

Note that this approach can be seen as a generalization of a *triangulation* method. In fact, in a 2D space the absolute position of a point can be determined if the distances from three other points are known (see Figure 3.4).

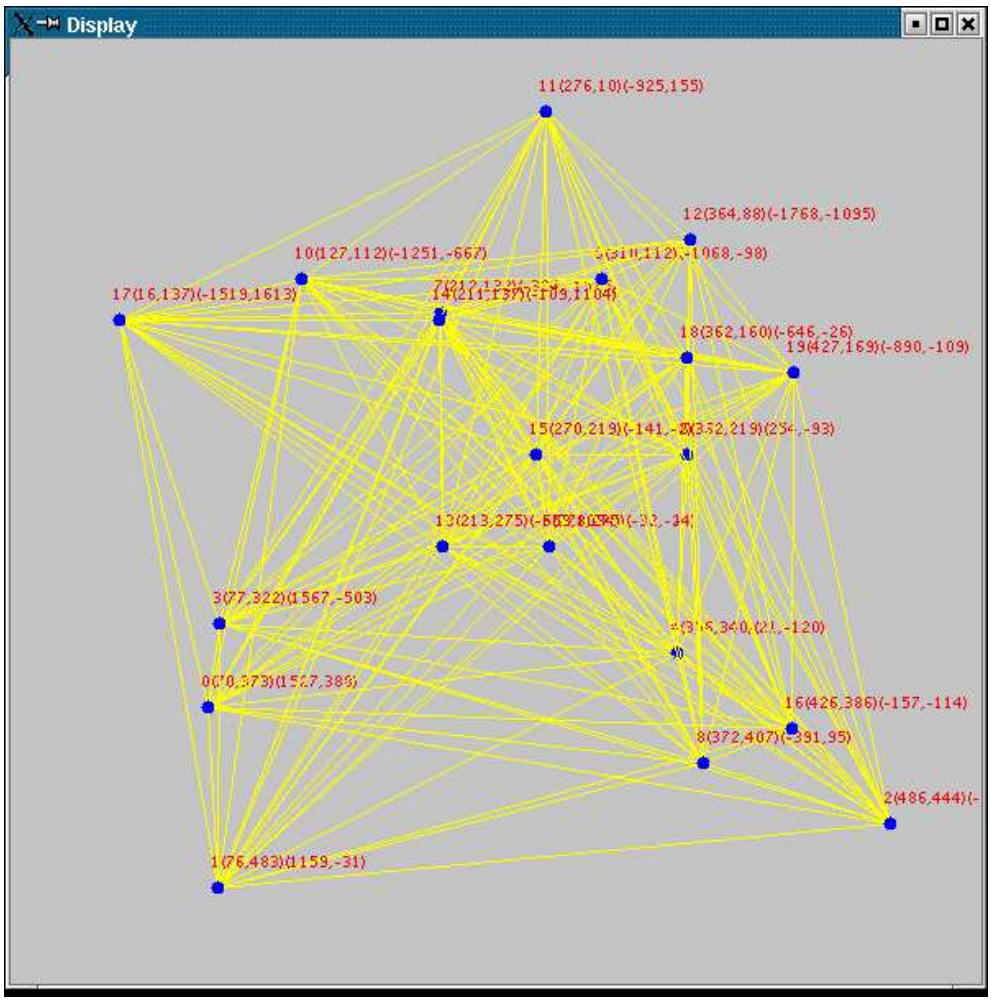


Figure 3.3: Screen-shot of the working algorithm.

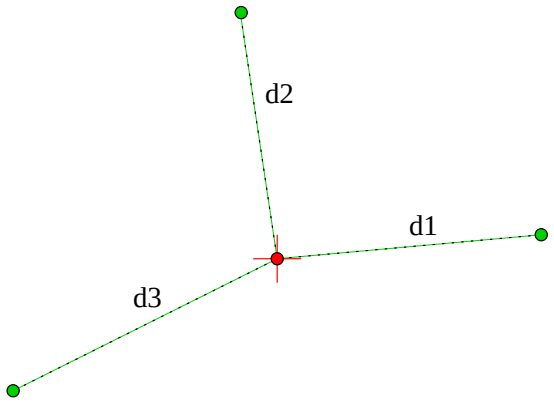


Figure 3.4: Example of triangulation in a 2D space.

In a d-dimensional space at least d+1 distances are necessary to determine uniquely the position of a point. Whenever there are more than $d+1$ distances, it is possible that no exact solution of the problem exists: there is no point whose coordinates can satisfy all the relationships given by the known distances. Given a certain number of known distances $n_d > d+1$, the coordinates of the point to place must satisfy n_d conditions. Triangulation methods offer an exact solution of the problem when $n_d = d+1$. The "Spring forces" algorithm is a numerical method to calculate these coordinates so that they fit best and satisfy the n_d conditions even when $n_d > d+1$.

3.4 Node Join Phase

This second phase occurs whenever a new node wants to join the small landmark network. This phase can be divided in four sub-phases:

1. Latency minimum search: from an entry point the search moves to the best area in which the new node can join.
2. Determine node ID: with the area and a certain neighborhood in that area chosen, a "spring forces algorithm" can set the node ID by taking into account relative distance metrics to neighbors.
3. Set up links: the new node must know its closest neighbor in each quadrant, so there is a specific search mechanism to do that;
4. Network update: all nodes that could have a reference to the new node should be informed of its presence.

3.5 Latency Minimum

The new node must know an entry point address from which the join phase can start. It aims to discover which is the best point it can actually join; here "best" means closest in terms of distance metric, so the search aims find that node in the network that is closest to the new node joining.

It proceeds by walking through the existing network using the latency measurement as an heuristic function: it walks in the direction in which the latency measurement decreases. Using this procedure, the new node normally finds a "good" (local) minimum among the latency measures. This minimum is assumed to be the closest neighbor in the hypercube network as well (see Figure 3.5).

As this phase has a critical impact on the global behavior of our protocol, we added a kind of "overshooting" to the minimum search process: even when a (local) minimum has been found, the search process does not stop but tries to localize an even better minimum. The direction in which the search continues is determined by the behavior of the "slope" of the distance metric function used (in our case latency) in that point. The search continues in the direction having the smaller slope; it ends up either with a better minimum found (from which the search starts again) or after a constant predefined number of steps. Given a fixed maximum number of steps n_s of the overshooting search[1], the complexity overhead of this phase is only $O(n_s)$.

Once the closest node has been found, the area in the ID space where the new node is to be placed is well defined. In order to place the new node correctly in that area,

[1] We chose a value ranging from 3 to 10 for this constant.

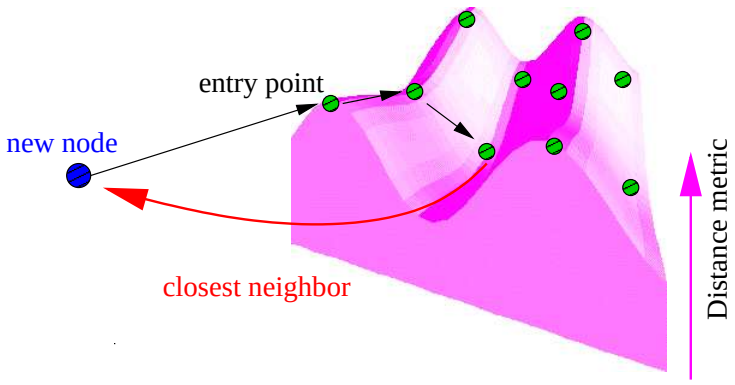


Figure 3.5: 3D representation of distance gradient following.

the "Spring Forces" algorithm introduced above is run once more. In this case the algorithm acts over a small part of the network, moving only the new node, relative to its new neighbors, as determined by the closest node found and its immediate neighbors. Finding the position of the new node in the ID space means to assign it a new ID.

The same nodes as used for the "Spring Forces" algorithm are also used to add the new node to the network, i.e. to determine the neighbors and establish connectivity.

3.6 Setting up links

Once that the position of the new node has been fixed in the virtual ID space, it must set up links to neighbors in the overlay network. This phase should ensure that the node is efficiently integrated into the network. The node should be reachable from anywhere and should be able to make accurate forwarding decisions for messages traveling through it.

As described in Chapter 2, the particular structure of the overlay network relies on the affirmation that each node in each quadrant is linked to the closest neighbor in that quadrant. This phase of the joining protocol aims to find the closest neighbor of a certain node in a given quadrant.

The starting scenario comprehends a node, with a its new ID, and a list of neighbors that it retrieved from the first phase of the joining protocol. However, nodes advertised in this list

- are not guaranteed to be the closest node in any quadrant;
- may not even be placed in one of the quadrants around the new nodes;

The process of finding the closest node in each surrounding quadrant is not trivial. Given a joining node A, and selecting a specific quadrant q among the 2^d possible quadrants adjoining A, the aim is to find a node B that is the closest neighbor for A in quadrant q (this scenario is illustrated in Figure 3.6).

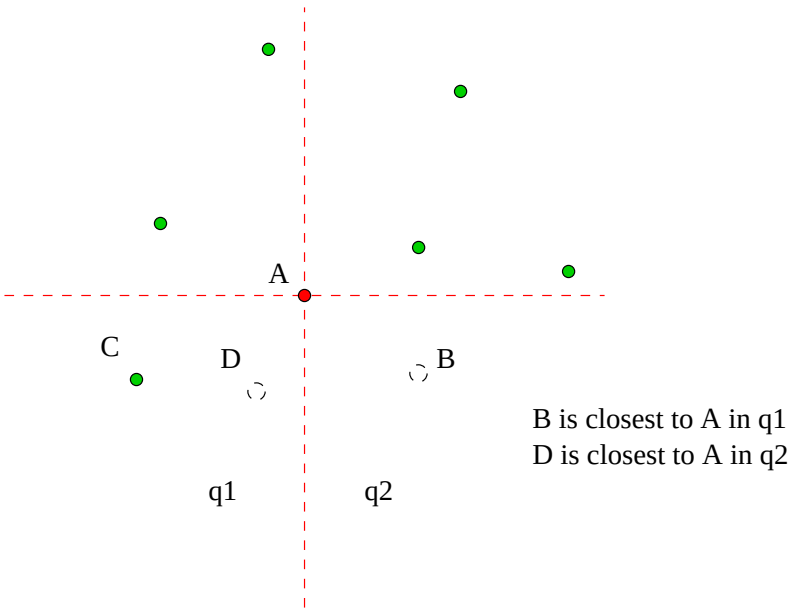


Figure 3.6: Scenario example.

There are two cases that require particular attention, and they are both shown in Figure 3.6:

- There is a node B in quadrant q_2 but A does not know any node in that quadrant;
- A knows a node C in quadrant q_1 but it is not the closest node (D is the closest) in that quadrant.

In our design, we provide a general first phase in which a node tries to get as close as possible to a certain quadrant. Then the second phase takes care of getting inside this quadrant and of moving as close as possible to the original node (A).

3.6.1 First connection sub-phase: "getting closer to a quadrant"

Supposing that a node A wants to find its closest neighbor in a given quadrant q, but does not know any node in that quadrant. The first phase of the neighbor-discovering process consists in getting as close as possible to quadrant q. This can be done by sending probing messages to known nodes and moving from them to their neighbors in the desired direction. Figure 3.7 shows the strategy used.

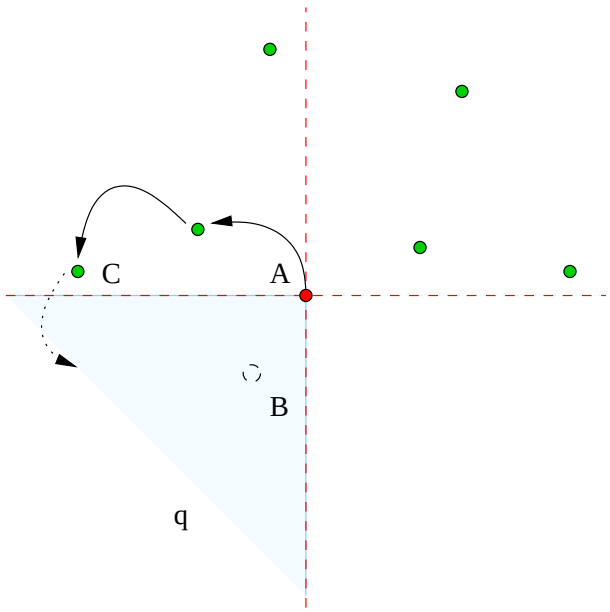


Figure 3.7: Example of strategy used to get "close" to a quadrant.

The "strategy" consists of moving clockwise from node to node until there are no more improving steps. That means that node A tries to locate a node that is as close as possible to quadrant q. It beginsby probing the neighbors it knows and proceeds hop by hop until the last hop found (C) does not have any neighbor closer than itself to quadrant q. This algorithm always converges because at each step

- either C knows a node that is better suited than itself, and which accordingly is the next hop for the search,
- or C does not know any node that could improve the situation, so that C is considered the final result of this search.

Once that a node C that satisfies the conditions above has been found, one can be sure that C has at least one neighbor inside quadrant q. In facts if it is not true, then C would have a neighbor C' outside q, but C' would be closer to q than C; in that case the algorithm would have passed from C to C' before, and C' would have been chosen as the closest node to q.

This first phase can be skipped whenever node A already knows a neighbor C that is already inside quadrant q.

3.6.2 Second sub-phase: "Finding the closest neighbor"

As shown in Figure 3.8, this phase starts when the search is already in quadrant q, but it has not yet found the closest node in that quadrant.

Firstly, in a simple 2D case (as in Figure 3.8) we have an *invariant* that allows the search phase to converge to the closest neighbor. Given a node C in quadrant q, which is the current node being evaluated by the search process, we have that

- C is the closest node or C knows about a closer node;

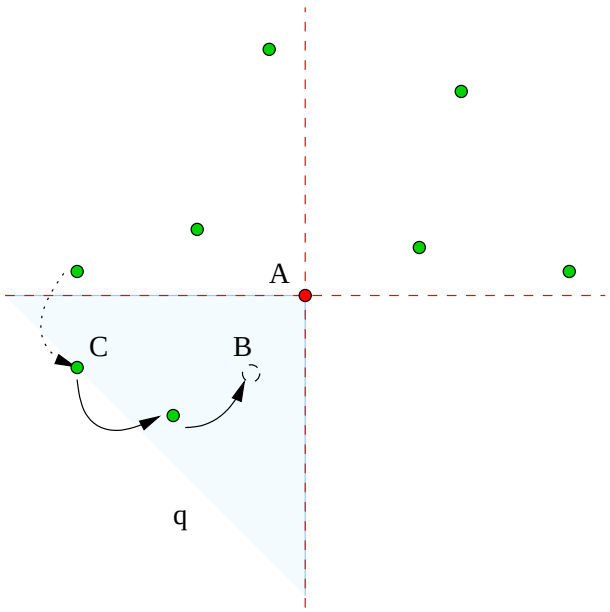


Figure 3.8: Search example inside a quadrant q.

→ there is a node further in counterclockwise direction that is the closest node or knows about a closer node.

The first rule is intuitive: it allows jumping from one node to the next by evaluating the distance from the original node A.

The second rule takes into account the relationship of "neighbors of a neighbor". This concept is necessary to deal with the inner nature of the quadrant structure. In a 2D space the second rule could be seen as *"try to proceed in counterclockwise direction until either a closer neighbor B is found or you exit from quadrant q"* similar to the routing mechanism used in GPSR [22]. When the number of dimensions is greater than two, this rule is not useful becaus clockwise and counterclockwise have no natural definitions there.

The generalization to the d-dimensional space, with a generic node C situated in the quadrant q, is as follows:

1. *"Whenever a node C' knows of a node in q and closer to A than C, then C' is the next node to be evaluated",*
2. *"whenever none of the neighbors of C satisfy rule* 1*, then look in quadrant q for a neighbor C'' that could potentially have a neighbor closer to A than C (always in q). If C'' exists then C'' is the next node to be evaluated but the message keeps a reference to C as current best match.*

Figure 3.9 shows two examples of application of rules 1 and 2.

From the radius of the circles we see that

→ in the first case the jumps from C to C' and from C' to the final destination B satisfy the conditions $\overline{AC'} < \overline{AC}$ and $\overline{AB} < \overline{AC'}$, so that following this path the distance from A always decreases;

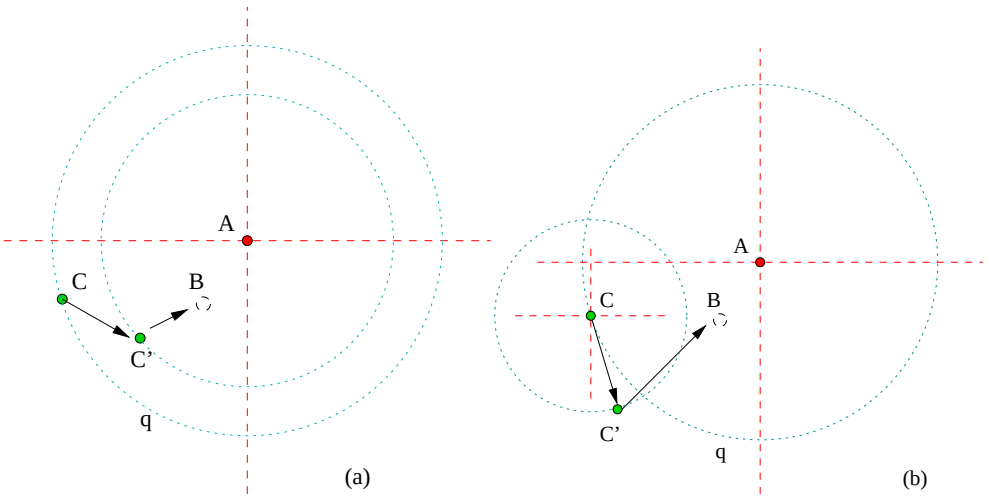


Figure 3.9: Examples of search rules application.

- in the second case note that $\overline{AC'} > \overline{AC}$ but $\overline{AB} < \overline{AC}$, so that B is the closest neighbor for A in quadrant q but C does not have a direct link to B. Therefore one cannot jump directly from C to B following the first rule. In other words, C does not know anything about B, it only knows a neighbor C'. Analyzing the position of A and C', C can deduce whether C' *could potentially* know a neighbor B closer than itself to A.

The sentence *"C can deduce whether C' could potentially know a neighbor B closer than itself to A"* needs a more formal explanation. The basic assumption is that each node has a link (not only, but at least) to its closest neighbor in each quadrant around it.

Given an hyperspace with coordinates $x_1, x_2, \ldots, x_d$, given the distance $\overline{AC}$, the sphere or the hypersphere with center in A and radius AC is

$$(x_1 - x_{1A})^2 + (x_2 - x_{2A})^2 + \ldots + (x_d - x_{dA})^2 = \overline{AC}^2 \quad . \tag{3.2}$$

Any node C' which is inside this hypersphere allows the search to pass from C to C' following the first rule: in fact in this case C' is closer to A than C.

C', being a generic neighbor of C, *"could potentially* know a neighbor closer than itself to A" if

- C' is outside the hypersphere of Eq. 3.2;
- C' is in the same quadrant as C when considered from the point of view of A;
- C' is in the same quadrant as B when considered from the point of view of C.

This conditions define an unique region in the ID hyperspace in which C' can be placed. This region is a kind of *dark zone*, intended as an area in which the first rule cannot be applied. It is represented in Figure 3.10, where the space has only two dimensions.

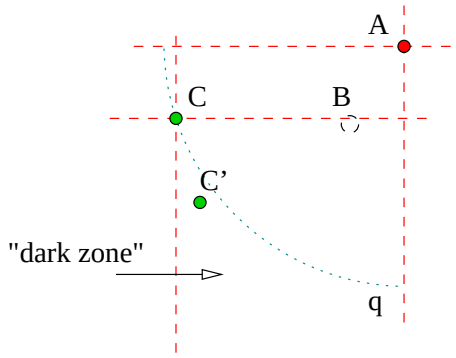


Figure 3.10: Example of *dark zone* in a 2D space.

On the one hand, note that the condition to be in a "dark zone" makes C' a good "next hop" in the searching phase. On the other hand, if B is not there, a node C' in a "dark zone" just represents a false path. In that case the only possibility is to revert to C and consider it as the closest neighbor to A in q.

3.6.3 Algorithm to recognize "dark zones"

During the search phase, whenever a node C and its neighbors are analyzed, it is not trivial to establish whether C' is in a "dark zone", especially when the number of dimensions of the ID space exceeds 2. Given C' a generic neighbor of C, then another hypersphere is defined having the center at C and with radius $\overline{CC'}$, as shown in Figure 3.11 and 3.12:

$$(x_1 - x_{1C})^2 + (x_2 - x_{2C})^2 + \ldots + (x_d - x_{dC})^2 = \overline{CC'}^2 \tag{3.3}$$

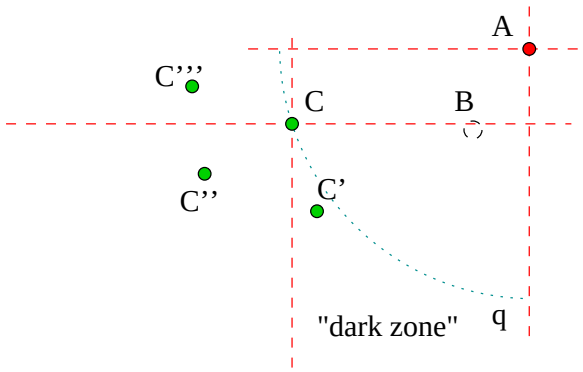


Figure 3.11: Determining a dark zone.

Through deduction, we can demonstrate that C' has to be in a dark zone when the intersection of the two hyperspheres (3.2) and (3.3) passes through quadrant q_c, where q_c is the quadrant in which C' is placed from the point of view of C.

In a d-dimensional space the intersection of two hyperspheres is a hypersphere of $(d-1)$ dimensions. If we calculate the intersection of the two hypersphere with the plane (not hyperplane) defined by the three points A, C, C', we always obtain the situation shown in Figure 3.12. This plane defined by the three points A, C, andC' can easily be written in a parametric form:

$$A\left(x_{1A},\ldots,x_{dA}\right) \quad C\left(x_{1C},\ldots,x_{dC}\right) \quad C'\left(x_{1C'},\ldots,x_{dC'}\right)$$

$$\underline{X} = u\left(\underline{A}-\underline{C}\right)+v\left(\underline{A}-\underline{C'}\right)$$

$$\begin{cases} x_1 = u\cdot\left(x_{1A}-x_{1C}\right)+v\cdot\left(x_{1A}-x_{1C'}\right) \\ x_2 = u\cdot\left(x_{2A}-x_{2C}\right)+v\cdot\left(x_{2A}-x_{2C'}\right) \\ \qquad\qquad \ldots \\ x_d = u\cdot\left(x_{dA}-x_{dC}\right)+v\cdot\left(x_{dA}-x_{dC'}\right) \end{cases} \tag{3.4}$$

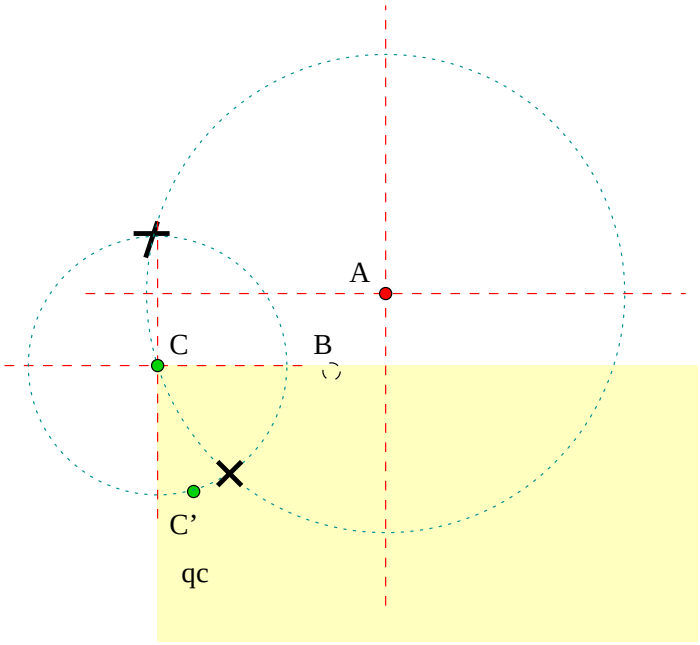


Figure 3.12: Intersections in a 2D space (hyperspheres become circles).

Independent of the number of dimensions n, the intersection of (3.4) with (3.2) and (3.3) always generates a 2D scenario as the one in Figure 3.12, yielding the system:

$$\begin{cases} x_1 = u\cdot\left(x_{1A}-x_{1C}\right)+v\cdot\left(x_{1A}-x_{1C'}\right) \\ x_2 = u\cdot\left(x_{2A}-x_{2C}\right)+v\cdot\left(x_{2A}-x_{2C'}\right) \\ \ldots \\ x_n = u\cdot\left(x_{dA}-x_{dC}\right)+v\cdot\left(x_{dA}-x_{dC'}\right) \\ \left(x_1-x_{1A}\right)^2+\left(x_2-x_{2A}\right)^2+\ldots+\left(x_d-x_{dA}\right)^2 = \overline{AC}^2 \\ \left(x_1-x_{1C}\right)^2+\left(x_2-x_{2C}\right)^2+\ldots+\left(x_d-x_{dC}\right)^2 = \overline{CC'}^2 \end{cases} \tag{3.5}$$

This system contains $n+2$ equations with n unknowns and 2 parameters. It always has two real solutions that are the intersection points of the two circles in Figure 3.12. We are interested in only one of these two solutions, the one which (eventually) falls into quadrant q_c.

The entire pseudo-code for the algorithm to recognize dark zones is:

---

currently evaluating a node C, define the hypersphere $\mathbb{S}_A$ of center A and radius $\overline{AC}$

for each neighbor C' of C placed in quadrant q_c with regards to C and in quadrant q with regards to A:

define the hypersphere $\mathbb{S}_C$ with center C and radius $\overline{CC'}$
define the plane $\mathbb{P}_{ACC'}$ through the three points A, C, C'
calculus of the two intersection of $\mathbb{S}_A$, $\mathbb{S}_C$ and $\mathbb{P}_{ACC'}$
if one of the two intersection calculated falls inside the quadrant q_c then:

C' is in a dark zone

else

C' is NOT in a dark zone

3.6.4 A Shortcut

The exact coordinates of the two intersections in Figure 3.12 can be found by solving system (3.5).

The process of searching a node in a quadrant does not need to know these coordinates exactly: it only needs to known whether one of these intersections falls into a specific space area (quadrant q_c, as explained before). Therefore solving the whole system is a waste of resources.

We found that there is a more efficient way to determine whether one of the intersections of the two circles falls into q_c. It is possible to calculate the projections of C' onto the axis that define quadrant q_c with origin in C (Figure 3.13). The empiric rule we use, which can be intuitively demonstrated, is

- *if at least one of the projections of C' onto the axes that define quadrant q_c with origin in C falls inside the hypersphere with center A and radius $\overline{AC}$, then the two hyperspheres discussed above have an intersection in q_c.*

The evaluation of this rule requires only the calculus of 2^d (or even fewer if we consider only the semi-axes around quadrant q_c) projections of C' in a d-dimensional space, and the evaluation of their distance from A.

As a consequence, whenever the above condition is satisfied, then point C' *is* in a dark region defined by A, B, q.

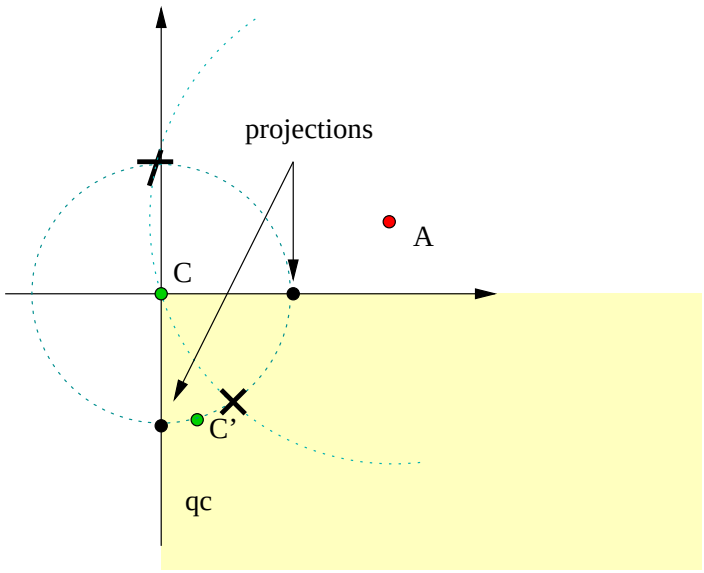


Figure 3.13: Projection of C' onto the axes defining q_c with origin in C.

3.7 Network update

The last phase of the node-joining protocol consists of updating the area in which the new node joins so that other nodes can have a reference to the new node arrived. This is done by sending some specific messages (such as "JOIN" or "NOTIFY_CLOSEST_CHANGED") so that

- each node A in each quadrant has a reference to its closest neighbor;
- each node A keeps references to nodes claiming A as their closest neighbor in a quadrant.

On the one hand this mechanism assures that nodes always have outgoing links in each quadrant around them. On the other hand, nodes are always referenced by other nodes (even if they are not "closest") so that isolated nodes cannot exist in the overlay network.

3.7.1 Updating Phase

This phase takes place whenever a new node (A) joins the network. It aims to inform all network nodes that could benefit. Nodes to be notified are all those that would consider the new node A as the closest in a quadrant. As shown in Figure 3.14, the number of nodes that can potentially satisfy this condition is $O(n)$, and the expected number is less than two per quadrant.

The new node joining, A, sends a specific update message through each quadrant to its closest neighbor in that quadrant. These 2^d (one per quadrant) messages travel through the network with the following considerations (see Figure 3.15):

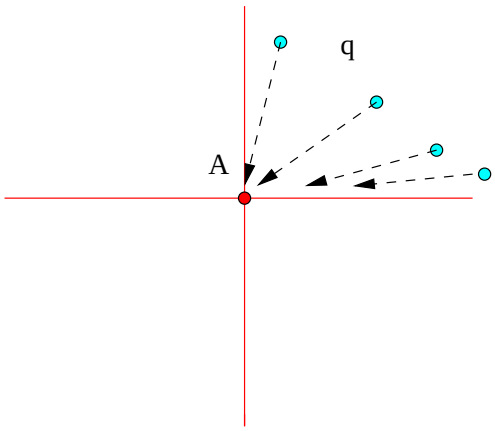


Figure 3.14: Example of nodes who claim A as the closest in A's quadrant q.

1. Given the hypersphere of center A and radius r, as in Figure 3.15, A sends the UPDATE message to its closest neighbor B, so that the radius r is the distance $\overline{AB}$.
2. B analyzes its neighborhood, and replicates the UPDATE message to neighbors trying to stay as close as possible to the surface of that hypersphere.
3. A d-dimensional space must be explored in $d-1$ dimensions to find all the neighbors wanted.
4. Message replication continues until the boundaries of the quadrant are reached in each $d-1$ direction.
5. This kind of exhaustive search always find all nodes who could claim A as the closest, and keeps strong locality properties. In fact, the nodes explored always lie *near* to A, and owing to the overlay network construction phase of our protocol, the closeness relationship keeps its value in the underlaying topology as well.

3.7.2 Joining Serialization

Another important mechanism we want to emphasize is the "serialization of multiple joins". If multiple nodes are relatively close in the underlying network and join the network simultaneously, it is likely they will occupy the same area in the ID space of the overlay network. In this case serialization is needed to prevent the protocol from misbehaving. The two conditions

- two or more nodes joining have a common potential neighborhood;
- two or more nodes joining have potential relationship among themselves in one or more quadrants;

represent the two necessary events to recognize when serialization is needed.

How to detect the need for serialization in practice and how to implement the serialization mechanism currently are open research areas.

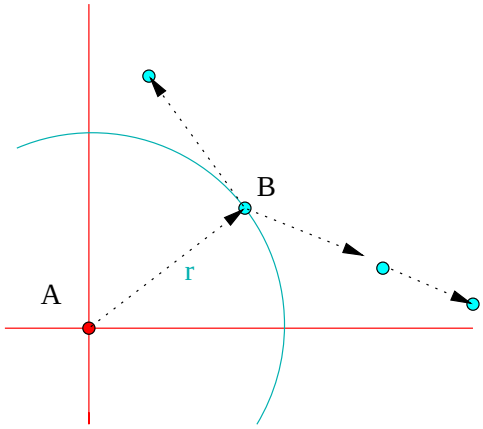


Figure 3.15: 2D example of message replication with update purposes.

3.8 Message exchange.

Figure 3.16 shows the joining phase of a new node into the overlay network we designed. This is the general behavior of the protocol; some exceptions can shorten this flow of messages: they have been successfully implemented in our simulator (see Chapter 4), but are not shown here. Examples of exceptions are

- the phase of locating a neighbor outside a certain quadrant can be skipped if the new joining node already knows a neighbor in that quadrant;
- the JOIN message may not require any notification so that there is no message NOTIFY_CLOSEST_CHANGED.

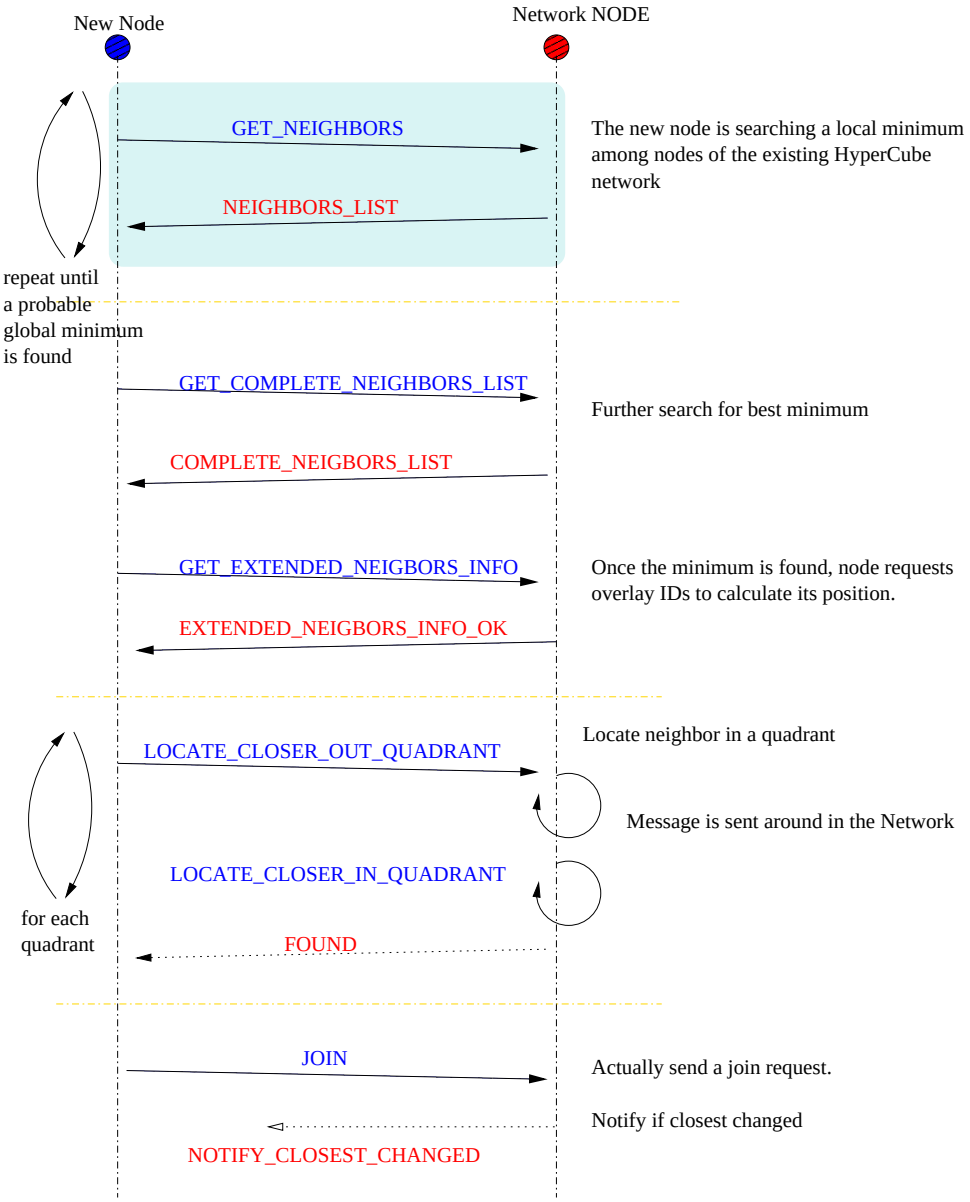


Figure 3.16: Example of message exchange.

Chapter 4

The network simulator

4.1 Requirements

The development of our own network simulator was motivated by our specific needs. Some common network and/or routing simulators (such as NS [4], MaRS [30], Ant [38], etc.) do not suit our requirements:

- Capacity of dealing with large networks, whose size is comparable to the entire Internet.
- In most of our scenarios, only a single message travels through in the network. The capacity of handling a large number of messages simultaneously, typical for routing simulators, turns out not to be useful, hence it would be a waste of system resources.
- Measurement and analysis focus on some specific parameters, so that we do not care about low-level details such as mean latency in each forwarding step, process time in each hop, etc. On the one hand, most simulators, particularly NS, offer a lot of detail about the network internals. On the other hand, the price to be paid for this measurement precision is worse performance, which unfortunately limits the size of the network to simulate.

Given these reasons, we decided to write our own simulator from scratch. This choice allows us to adapt the program exactly to our needs and to achieve a better tradeoff between the value of results obtained and the complexity of the analysis.

4.2 Programming language

Analyzing the project requirements, the possible choices of the programming language were C, C++ or Java. The choice of Java resulted in some important advantages:

- high code flexibility about code re-usage in different situation;
- high portability to multiple platforms;
- availability of many data structures already implemented in standard libraries;
- improvements in code simplicity and readability thanks to the completely modular organization of classes;
- simple building of a visualization and a user-friendly graphical interface.

The only drawback is that the large amount of data to process requires particular attention in the design of classes. Our simulator should work with hundred thousands of nodes, so any code that is executed multiple times and is replicated multiple times in memory should be particularly efficient in terms of CPU usage and memory occupation (details are given in Section 4.6).

4.3 Simulator Goals

Our main goal is to evaluate the mean delay that a common *find request* accumulates from the source to any destination. This is the main parameter to judge how efficient the protocol is in terms of query latency.

Secondly, we want to evaluate the impact of simulation parameters on the protocol. The main parameters are the total number of nodes, the number of dimensions chosen for the hypercube structure, the size of additional routing information to store at each node, etc. The simulator must provide results about the choices to take in order to improve protocol behavior.

Thirdly, there are various ways to make the protocol work: one can play with the link disposition, with the dynamics of node joining and leaving and so on. This is the main reason for developing a software simulator soon. Simulation results are good guidelines to further protocol design, and help define the protocol details.

4.4 Simulation Layers

The division into two layers has been respected during the simulator development. The two different layers, as described in the Section 2.1, have different requirements from the simulation point of view:

- The basic underlying layer provides basic connectivity among nodes. In other words, whenever a node knows the low-level address (for example the IP address) of a certain destination, then this layer is charged with taking the message to that destination. This layer also simulates the static behavior of some well-known routing protocols such as RIP [6] or OSPF [5].

- The core of the *hypercube routing* protocol is located at the highest layer. At this layer, simulations should deal with the routing decision in the peer-to-peer network. At the same time global end-to-end analysis of all important phases of our protocol can be performed.

4.5 Design Details

4.5.1 Observer design pattern.

The *observer design pattern* [16] turned out to be very useful to implement the scheduler policies. It prevents processor resources from being wasted in continuously polling all nodes; instead when a node requires processing resources it notifies a central unit (the scheduler), which keeps a queue of all nodes to be served.

This design pattern works in two phases:

- *Osbservable* objects register their own observer, so whenever a new event occurs the observer object can be informed, see the example in Figure 4.1;

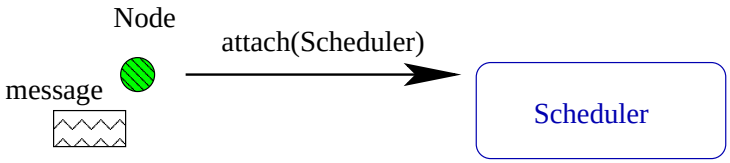


Figure 4.1: Attach observer example.

– Whenever a change occurs, observable objects *notify* their observer; an example is given in Figure 4.2.

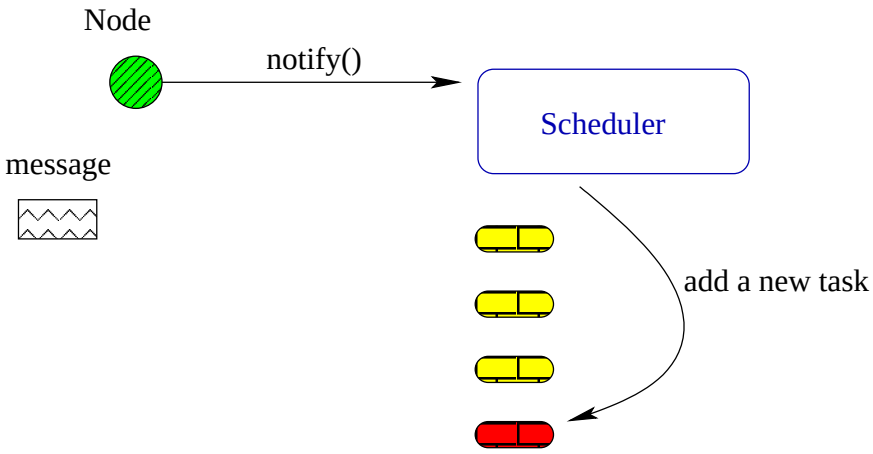


Figure 4.2: Notify example.

A node which requires services from the central scheduler is considered a *task*. Using the observer design pattern, the scheduler always has a updated list of active tasks and can share its resources fairly among them.

4.5.2 Statistics collection

Each message object contains some information fields which are updated, either in each node the message passes through, or at the beginning or at the and of the message path (for details about paths, see Section 4.5.3).

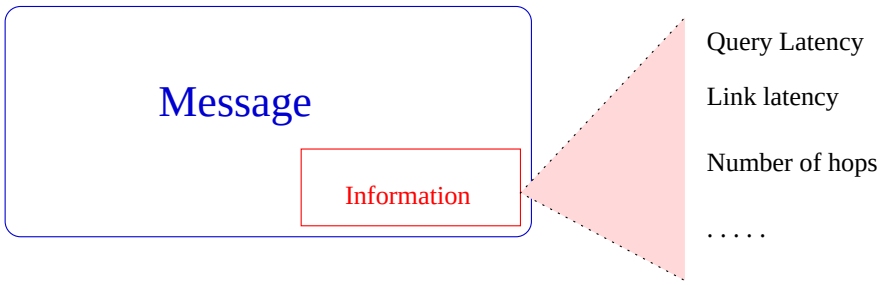


Figure 4.3: Information fields in each message.

4.5.3 Source routing or hop-by-hop

Even if the routing protocol of the lower layer is supposed to be efficient and reliable, there is no need for implementing a real routing algorithm such as RIP [6] or OSPF [5]. As the simulator has a priori complete knowledge of the entire network, the route

for each message can be pre-computed at the beginning, when the message is sent. This approach has the advantage of offering useful statistics as soon as the message is sent. Routes are calculated by creating a shortest path tree from the source node to the destination and then extracting the source-destination path from this tree.

To improve the simulation speed, once the route for a certain message has been computed and statistics for that message have been gathered, the message can be directly sent to the destination, without passing through any intermediate underlying node. Even if in some cases the synchronization among multiple messages is not preserved, the speed improvement obtained largely justifies this choice (which is the *default* working mode). However, we also keep the possibility to route the message in a more "traditional way", that is, step by step. This is suitable for studing the effects of queuing at intermediate nodes and of "hard" message synchronization. For the time being we did not used this secondary mode because we are mainly interested in end-to-end statistics.

4.5.4 Scheduling

The core of the design is an event-driven, discrete-time[1] network simulator. As shown

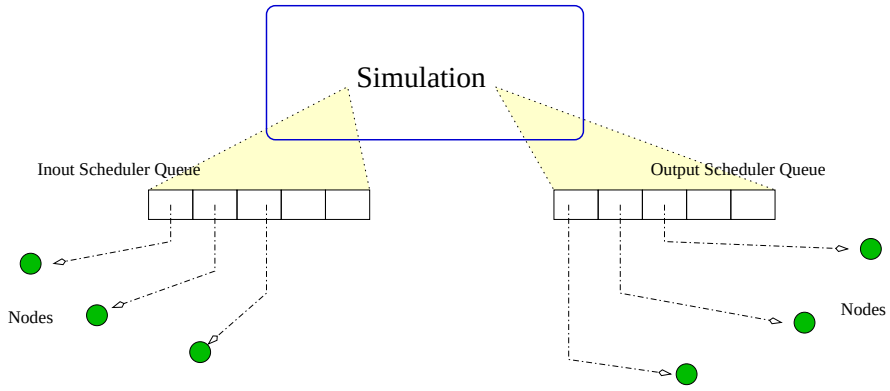


Figure 4.4: Simulation organization.

in Figure 4.4, the simulation is based on a simple task scheduling:

- Nodes that are going to *receive* a message are scheduled as new tasks in a FIFO *input queue.*
- An *output queue* is reserved for nodes that want to send a message.

Then the simulation consist roughly of repeatedly updating these two queues, as shown in Figure 4.5. The processor resources are never wasted: they are always fairly shared among all active tasks. The scheduler performs a Round Robin among active tasks.

[1]Time of simulation is discrete but unspecified: the original order of messages is preserved during the exchange, but there are no references to any numeric timestamps. Messages are ordered in a FIFO queue.

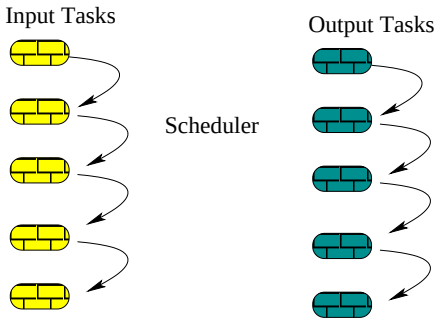


Figure 4.5: Scheduler tasks.

4.6 Simulator Optimization

The necessity of optimizing some aspects of the network simulator arises when dealing with a large amount of data. Our task is to analyze and gather statistics of large-scale networks with more than 100,000 nodes. We can take advantage of the power and flexibility of a strong object-oriented language like Java, and use specific optimizations to keep the performance of the program reasonably acceptable.

4.6.1 Building the shortest path tree.

Figure 4.6 explains the main phases of our algorithm, as also used by Dijkstra[12]:

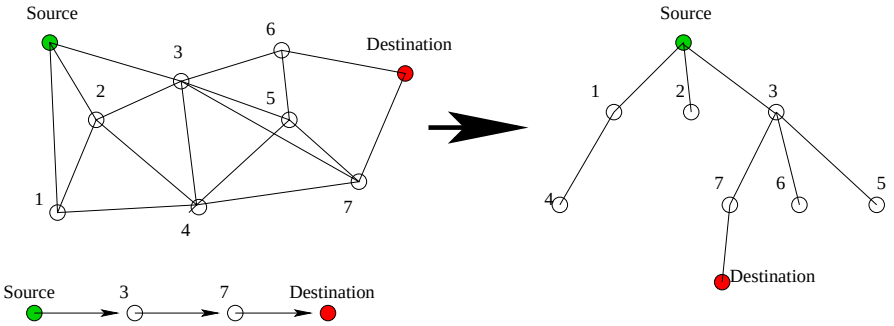


Figure 4.6: Shortest-path computation.

Normally the tree building procedure is $O(n^2)$ with an n-node graph. This is because each time a node is to be inserted into the shortest path tree, it is necessary to check whether it already exists. This is an ordinary (in-order) search in tree and it is on the order of n, so the entire routine is on the order of n^2. To improve the behavior of this routine, we used an auxiliary data structure to reduce the search complexity. Using a hash table to check whether a certain element exist, the search time is constant and the entire tree building procedures remains $O(n)$.

The tree structure is a tree with linked lists among nodes at the same level and parent pointers for each node (see Figure 4.7). The "horizontal" linked lists are necessary because the tree must be constructed in a *breadth-first* manner, that means

level by level. The parent pointers allow us to extract the path from the destination to the source easily, in only $log(n)$ steps, as $log(n)$ is the tree depth. So the time complexity of searching the shortest path between two nodes is $O(n) + O(log\, n) \leqq O(n)$.

After we did a standard implementation in Java, the algorithm turned out to be

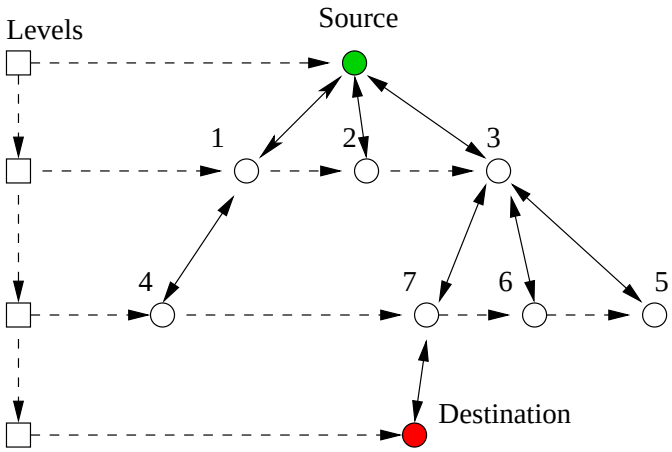


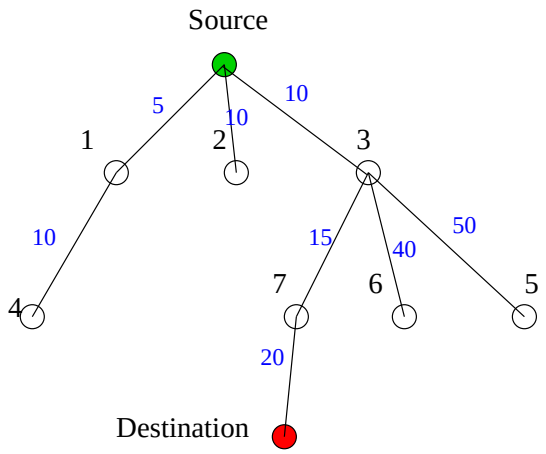
Figure 4.7: Data structure.

extremely slow, so that the exchanges of messages at the lower layer was not nearly as fast as we needed. The main reason was that creating the shortest path tree involves the creation of too many objects. We did a further optimization leading to better results: the creation of dynamic objects can be avoided completely; all local data, as well as the method to do the computation itself have been declared as *static*; the tree has been substituted by a couple of static vectors of integers, and all work is done using only the *integer* data type. In other words, the strong object-oriented features of Java have been substituted by an old procedural style of coding, resulting in a speedup by one order of magnitude.

4.6.2 Locality and caching

Whenever a node sends a message to a certain destination, there is a high probability that further messages will be sent to the same destination and vice-versa, as messages usually require answers. Note that, from the point of view of the simulator, the only purpose of the shortest-path tree calculation is to determine the delay a certain message accumulates in going from its source to its destination. If this information could be cached somewhere, it would not be necessary to recalculate the shortest-path tree for each message to a certain destination. It would be sufficient to calculate it for the first message, and then use the cached information for further messages.

Figure 4.8 demonstrates how we designed local node caches in order to avoid having to repeat the shortest-path tree calculation for each message. When the shortest-path tree has been built, delay information relative to each node is stored in the same hash table used to verify node existence in the tree. For each node the hash table contains the cumulative delay from the source to that specific destination. After the tree and the table have been calculated, the shortest path is extracted from the tree and



Nodes	Delay
1	5 ms
2	10 ms
3	10 ms
4	15 ms
5	60 ms
6	50 ms
7	25 ms
Dest	45 ms

Hash table with cumulative delay from source.

Figure 4.8: Caching.

the tree is discarded, and the hash table is stored in the local node. Whenever a new message has a destination that already is contained in the hash table, it can be sent without recalculate again the shortest-path tree. It is sufficient to pick the delay information from the hash table (with a complexity $O(1)$) and forward the message. Note that the hash table does not contain all destinations possible. This is a consequence of the tree construction method: when the shortest-path tree is built, the building process is stopped as soon as the specific destination is reached. This choice is a further speed-up in shortest-path calculation.

4.6.3 Specific Optimizations

- Avoid method calls in functions where efficiency is critical.
- Avoid thread spanning during the simulation. This is a common approach also in other simulation environment, such as JavaSim [20].
- Limit object creation, growth, and structure size in frequently used code as much as possible .

4.6.4 Profiling

Using the profiler option built into the Jsdk 1.4, it is possible to track down any performance problems including memory leaks, excessive object allocation, performance bottlenecks and inefficient algorithms, right down to the line of source code responsible. We used this feature to find out where particular optimizations were needed (see Figure 4.9). The profiler allowed us to discover the pieces of code that were particularly slow and to concentrate our efforts to some precise aspects. Moreover, whenever a certain section was optimized, it allowed us to compare directly its performace before and after the optimization.

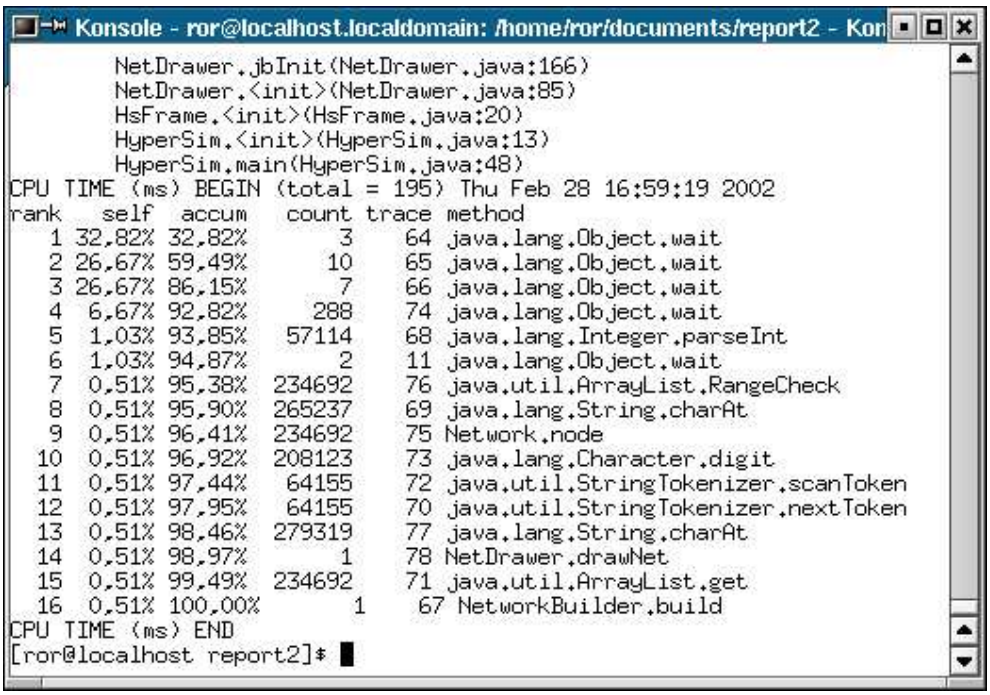


Figure 4.9: Example of profiling output.

4.7 Graphical Interface

The graphical interface has been built from scratch. A complete view of the network is provided, as can be seen in Figure 4.10. The user can easily interact with the program using either the menu bar on the top or the pop-up menu within the window. Any part of the network can be explored with a single mouse click. The viewing window can "fly" over the network, and be moved to any desired point just by dragging the mouse pointer to the new point specified. The 2D graphics are quite sophisticated and include some specific cares: triple buffering[2], resolution reduction during the drag, and some extra-computation to place nodes, draw links, and handle zoom. Moreover, we also developed another version of the simulator, that excludes the Java classes with the graphic interface. This choice brings a further speed-up in simulations that do not requires a "real time" visual output. These "blind" simulations continually output statistics for the benefit of the user (for example mean latencies, number of hops, path lengths, etc.) that can be easily ordered and printed out in graphs and tables using basic plotting tools.

4.8 JNI

JNI is the *Native* programming *Interface* for *Java* that is part of the JDK. Programmers use JNI to write native methods to handle those situations when an application cannot be written entirely in the Java programming language. In our case, we needed

[2]Useful to efficiently handle the dragging of the mouse pointer over the window.

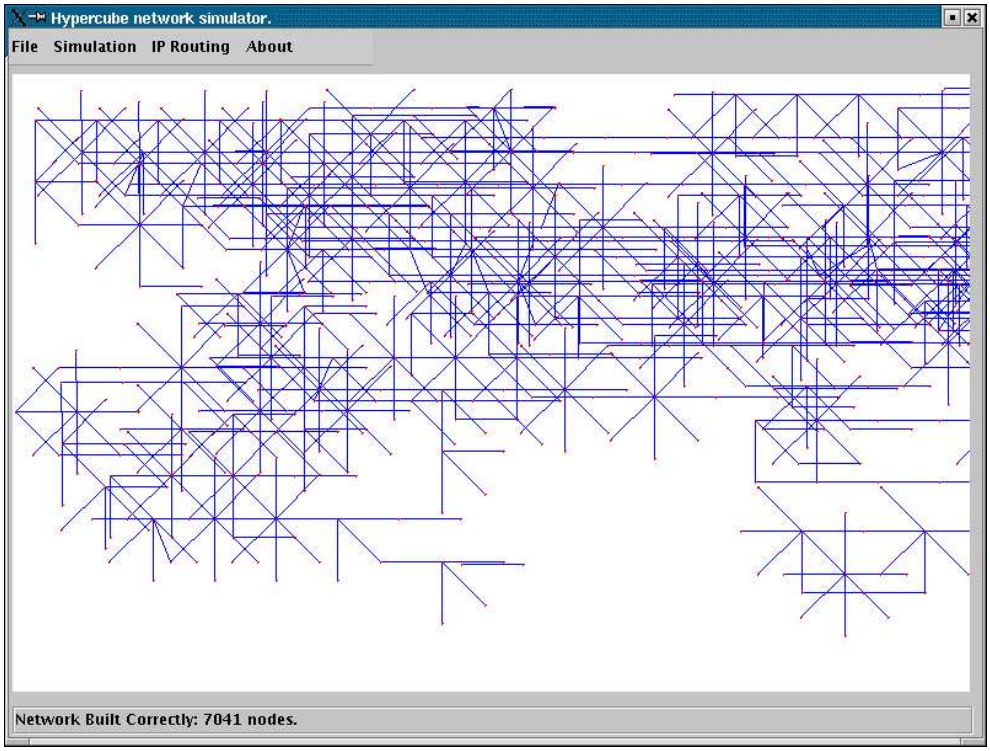


Figure 4.10: Screen-shot example of the network simulator.

to implement a small portion of time-critical code in a lower-level programming language, such as C, and then have the Java application call these functions.

Using the java profiler we found that the standard Java implementation of the *Hashtable* class was too slow for our purpose: we use hash tables to store latency information for each node, so that the shortest-path tree among nodes can be calculated efficiently.

The problems with the standard *Hashtable* class arise with the methods:

```
Object put(Object key, Object value);
Object get(Object key);
```

These two methods do not allow to put and retrieve *int* variables directly into a hash table. The correct code to add and retrieve elements is

```
hashtable.put(new Integer(key), new Integer(number));
int number = (Integer)hashtable.get(key)).intValue();
```

As hash tables are accessed continuously during the simulation and each node in the network has its own hash table, this code turns out to be too slow owing to frequent object creation and access functions.

Using JNI, we coded our own hash table implementation. The result is a shared library which contains a group of functions to handle hash tables efficiently, such as: *init, put, get, contains,* etc., so that the operations now look as follows:

```
hashtable.put(key, number);
number = hashtable.get(key);
if (hashtable.contains(key)) ...
```

The shared library also handles memory allocation for elements in each table, and has its own addressing system to handle multiple hash tables of different nodes.

The impact of this approach on the overall behavior of the simulator was significative. The speed-up obtained by using this native library is around 150% (1.5 times faster), even after the optimizations reported in section 4.6.1.

Unfortunately we pay fr this excellent performances in terms of portability. The library using JNI is strictly platform-specific, so it needs to be recompiled whenever the program is "ported" to another platform.

4.9 Testing

Software testing is the process of verifying the functionality and correctness of software by running it. Software testing is usually performed for one of two reasons:

- error detection
- reliability estimation.

The problem of applying software testing to *error detection* is that software can only suggest the presence of flaws, not their absence (unless the testing is exhaustive). The problem of applying software testing to *reliability estimation* is that the input distribution used for selecting test cases may be flawed. In both of these cases, the mechanism used to determine whether the program output is correct (known as an oracle) is often impossible to develop. Clearly the benefit of the entire software testing process is highly dependent on many different pieces. If any of these parts is faulty, the entire process will be compromised.

Testing involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets the requirements [28]. Software is not unlike other physical processes where inputs are received and outputs are produced [19]. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways. By contrast, software can fail in many bizarre ways and detecting all of the different failure modes for software is generally infeasible.

An interesting analogy to the difficulty in software testing is use of the pesticides, known as the Pesticide Paradox [8]: *"Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual"*.

4.9.1 Simulator testing

In our case, the testing phase did not have to be intensive, for the following reasons:

- firstly, we do not plan to ship a public release of our network simulator. Our purpose is only to analyze our protocol;
- secondly, our simulator does not require a large number of inputs. Simulations need a network-structure file (that describes the topology of the network, usually the output of a topology generator) and a few parameters;

- thirdly, our Java implementation guarantees a certain portability[3] among multiple platforms and prevents the presence of bugs linked to specific platforms;

On the other hand we also had valid reasons to perform testing; namely,

- tp guarantee the validity and consistency of the results obtained with the simulator;
- the deep optimization done by using the Java Native Interface led to strict machine- and platform-dependent code. Consequently, we needed to ensure that these parts of code are reasonably "bug free".

4.9.2 Programming with *Assertions*

An assertion is a statement containing a boolean expression that the programmer believes to be true at the time the statement is executed. For example, after "unmarshalling" all of the arguments from a data buffer, a programmer might assert that the number of bytes of data remaining in the buffer is zero. The system executes the assertion by evaluating the boolean expression and reporting an error if it evaluates to false. By verifying that the boolean expression is indeed true, the system corroborates the programmer's knowledge of the program and increases the confidence that the program is free of bugs.

Assertion checking may be disabled to increase performance. Typically, assertion-checking is enabled during program development and testing, and disabled during deployment. Because assertions may be disabled, programs must not assume that the expressions contained in assertions will be evaluated. Thus, it is extremely bad practice for these expressions to have any side effects: evaluating the boolean expression should never affect any state that is visible after the evaluation is complete (it is not, in general, possible for the compiler to enforce a prohibition of side effects in assertions, so the language specification does not disallow the practice).

Similarly, assertions should not be used for argument-checking in public methods. Argument-checking is typically part of the contract of a method, and this contract must be upheld regardless of whether assertions are enabled or disabled. Another problem with using assertions for argument checking is that erroneous arguments should result in an appropriate runtime exception (e.g., *IllegalArgumentException*, *IndexOutOfBoundsException*, *NullPointerException*). An assertion failure will not throw an appropriate exception (again, it is, in general, not possible for the compiler to enforce a prohibition on the use of assertions for argument checking on public methods, so the language specification will not disallow the practice).

Types of assertions include the following:

- *Precondition* - an assertion that must be true before some operation. For example the precondition of x = y/z is that z != 0. If the precondition is false, the code can do anything up to and including starting World War III. If the precondition is true the code should always succeed, i.e. make the postcondition true.
- *Postcondition* - the assertion that will be true after the code completes (if the precondition is true). For example the postcondition of sort is that the output value is the sorted version of the input.

[3] An exception is the library developed with JNI, which needs to be recompiled if the program is ported to platforms different than Linux.

- *Loop invariant* - an assertion which is true on every iteration of a loop. For example one way to sort an array is to sort a[0..i] with i initially 0. If we increase the region that is sorted by 1 each time we go around the loop the array will eventually be completely sorted. In the case the invariant is that a[0..i] is sorted.
- *Loop variant* - an integer expression used to prove that a loop terminates. For sorting, the variant is the number of elements which have not been sorted. If on every loop iteration we reduce this number by 1, and we stop when it reaches 0 (i.e. sorting has finished), then the loop will terminate. Loop variants are used to represent progress in solving the problem.
- *Data invariant* - an assertion about data variables which is always true. For example an integer representing human tallness must be always positive.

Example extracted from our code:

```
static public double distance(Node a, Node b){
long square_sum = 0;
( ... )
assert a != b : a; //print out the value of a if expression is false
assert (a.address != -1) && (b.address != -1);
( ... )
```

The piece of code above belongs to the set of functions responsible for calculating the distance between two nodes in our virtual multi-dimensional space. This example shows that assertions are useful to check whether everything works well.

Chapter 5

Analysis and results

5.1 Design parameters

The overall behavior of the protocol we designed is influenced by some main parameters analyzed in this section.

5.1.1 Hyperspace dimensions

As the overlay network aims to reflect somehow the underlaying topology, one important parameter is certainly the number of *dimensions* (d) of the virtual ID space. This parameter stronlgy influences both node placement in the ID hyperspace and number of links for each nodes. As discussed in Chapter 2, the number of links that each node keeps is $O\left(2^d\right)$ (at least one link in each quadrant). As d has a strong influence on the degree of connectivity of the overlay network, we expect a strong impact on the efficiency of the routing process as well as on the routing table. Figure 5.1 shows some examples of hypercubes in spaces with different numbers of dimensionsto illustrate how the degree of connectivity among vertices changes.

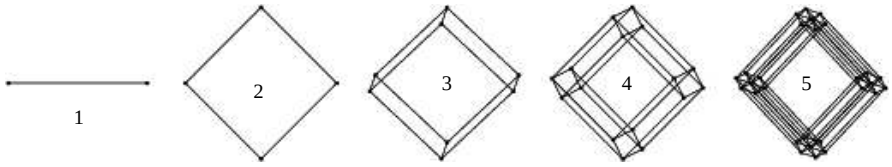


Figure 5.1: Hypercubes in 1, 2, 3, 4, 5 dimensions.

5.2 Local versus Global Minimum

An important parameter, which strongly influences the joining phase in our design, is how well the structure of the overlay network can adapt to and take into account the structure of the underlying network.

Every time a new node joins the global overlay network, it tries to find the best neighborhood. This is an area close to it in terms of the kind of distance metrics used. If, for example, the distance metric used is *latency*, then it will join into the network in an area with nodes having minimal latency to it. We wanted to analyze how much the minimum in latency found by any new joining node (which we call local *minimum*) is related to the *absolute* latency minimum existing in the network for that node. Figure 5.2 shows a typical graph-output that we obtained from a simulation with 1000 nodes in the overlay network. The graph is related to the joining phase of the 1000th node in the network and shows the quality of the local minimum found by this new joining node. Clearly this graph by itself is not significant, because the behavior of the joining phase depends on the node and topology, but this graph is useful to give an idea of what is happening during the joining phase.

As you can see in Figure 5.2 (the star represents the minimum found), the new joining node in this case finds quite a good minimum, located on the part of the curve in which the latency is steeply decreasing.

Moreover, multiple simulations have shown that typically about 40% of the local minima found are also global minima.

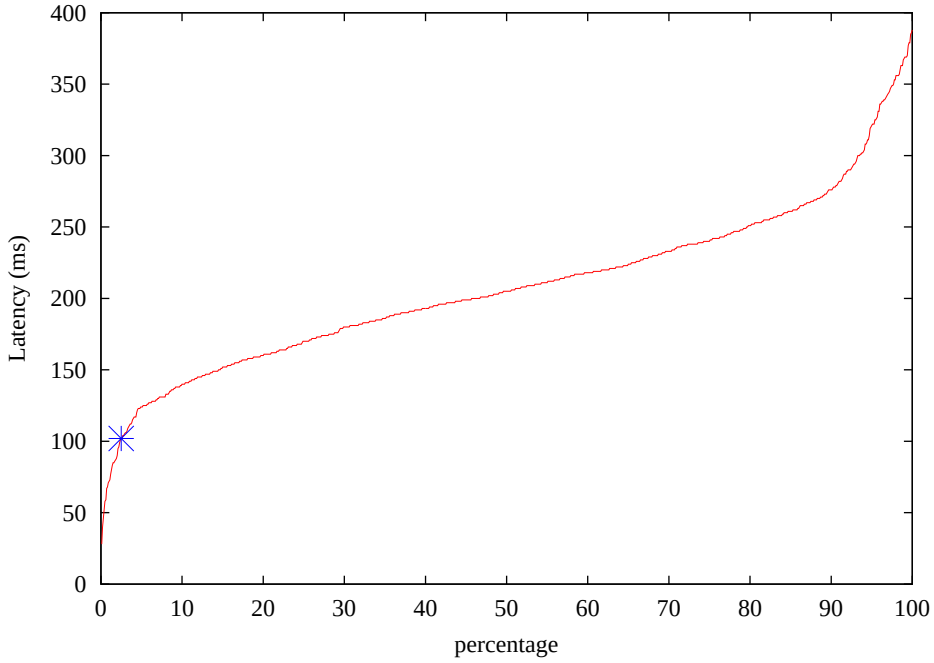


Figure 5.2: Minimum analysis based on latency measurement.

We also analyzed the ratio between the latency of the local minimum and of the global minimum. Figure 5.3 shows the results. This figure has been generated with 300 nodes joining the overlay network one by one; for each we take into account two parameters:

- the delay from the new joining node to the local minimum;
- the delay from the new joining node to the global minimum.

Note that these two parameters are different for each new joining node. Figure 5.3 shows the ratio between these two parameters. Specifically, Figure 5.3 represents the cumulative distribution function of the ratios obtained.

We can easily see in Figure 5.3 that the average ratio is relatively small (less than 5 for more than 80% of the nodes).

Figure 5.4 shows the behavior of the latency minimum found when the number of nodes (n) in the overlay network grows. Note that the behavior does not get worse when n increases: there are a few peaks up to 10 but the average ratio remains around 2.

Figure 5.5 shows the absolute latency difference of the ratio peaks in Figure 5.4. Values of latency ratio grater than 5 are considered as peaks.

5.3 Routing path length

The estimation of the latency minimum is significant because it influences the relationship between the routing path in the overlay network and the one in the underlying

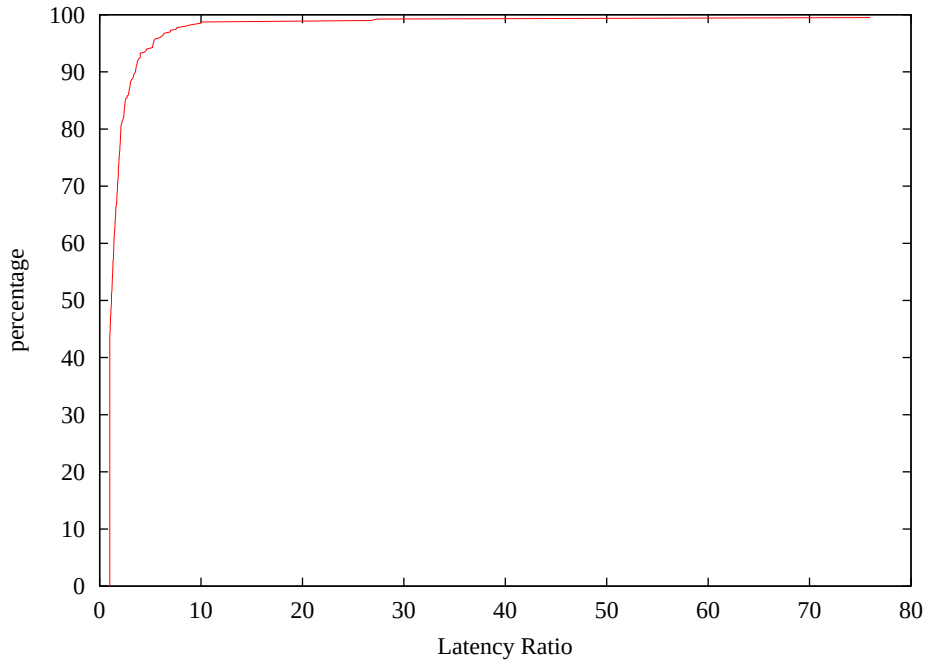


Figure 5.3: Latency ratio from the local/global minimum for each joining node.

network. Figure 5.6 points out the meaning of this relationship.

If we call n_U the number of hops needed to route a certain message from a source to a destination in a basic underlying network (for example, the IP network) and n_O the same quantity defined in the overlay network, we find that

$$n_U \leq n_O \tag{5.1}$$

The ratio $\frac{n_O}{n_U}$ is a fundamental expression of the efficiency of the overlay routing protocol, and the efficiency increases when this ratio becomes closer to one (note that $\frac{n_O}{n_U} > 1$ always).

Figure 5.7 has been plotted in an overlay network of 200 nodes and an underlying network of 10,000 nodes. Each curve represents the ratio $\frac{n_O}{n_U}$ for 10,000 routing requests made in the overlay network. The values obtained are sorted in increasing order of the ratio $\frac{n_O}{n_U}$.

As expected, increasing the number of dimensions leads to better performance in terms of the path-length ratio, because the degree of connectivity of nodes in the overlay network increases, and accordingly the length of the end-to-end routing path tends to decrease.

Note also that when the number of dimensions (d) is four or greater, the path length in the overlay network remains always smaller than twice the one in the underlying network. This results suggests a good value for the global number of dimensions of the virtual ID space: $d = 4$ turns out to be a good choice.

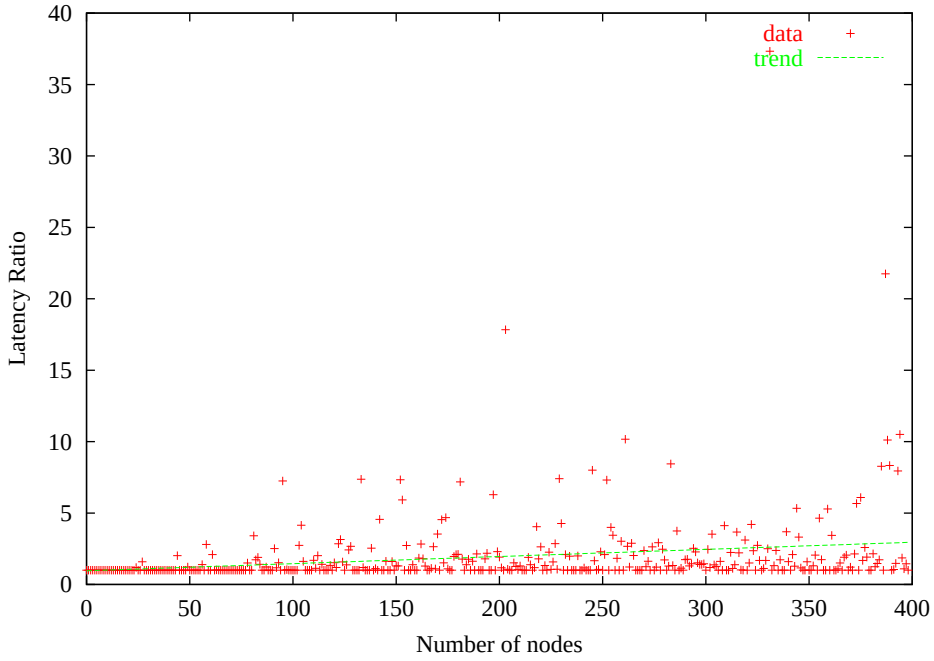


Figure 5.4: Quality of latency minimum found as the network grows.

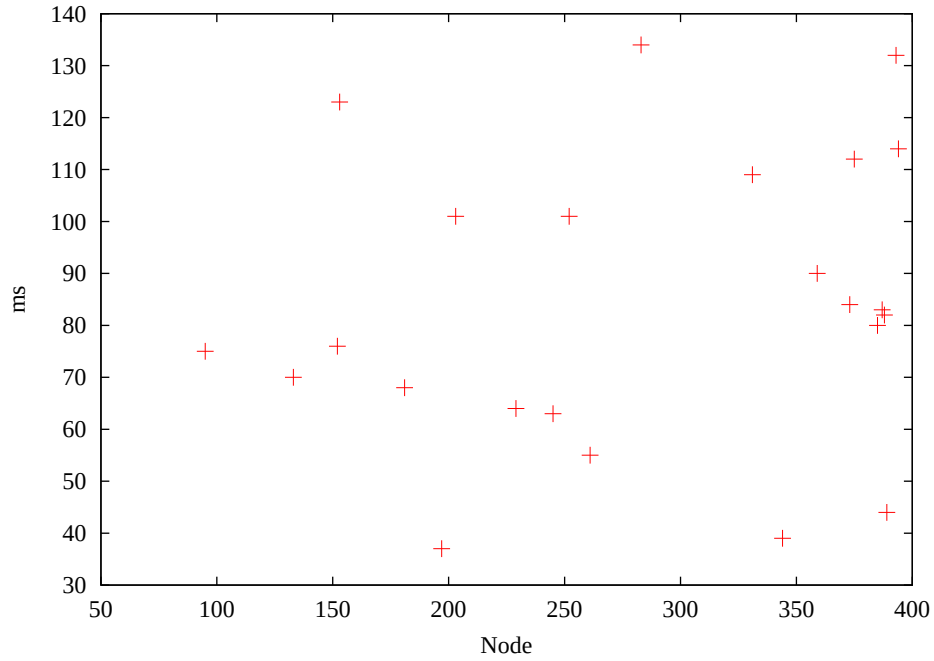


Figure 5.5: Absolute latency difference of peaks in Figure 5.4.

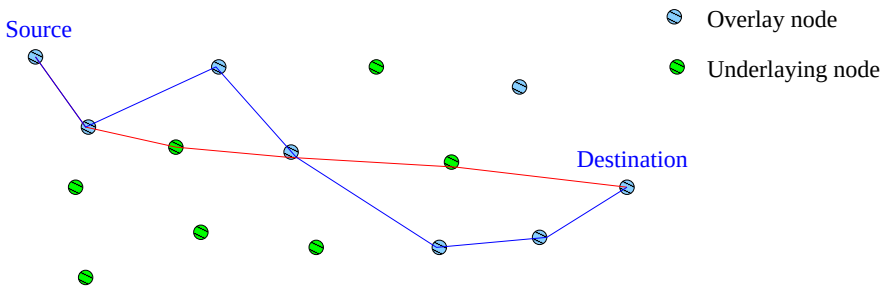


Figure 5.6: Relationship between routing paths of overlay/underlying network.

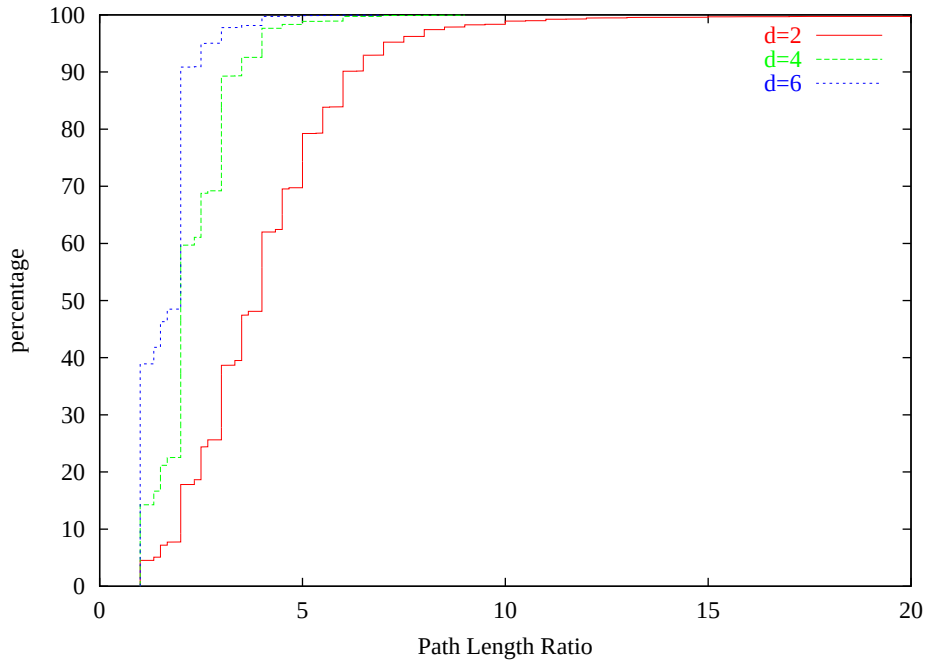


Figure 5.7: Path length ratios with 2, 4, 6 dimensions.

5.4 Memory usage

Using $d = 4$, the average number of neighbors referenced by each node, which is also the number of "routing table entries" in the overlay network, is:

$$2 \cdot 2^d \big|_{d=4} \Rightarrow 32$$

If we choose $d = 6$, then we would have

$$2 \cdot 2^d \big|_{d=6} \Rightarrow 128$$

Given that in the d-dimensional ID space we allocate b bits for each dimension, these previous simulations have been done with $b = 10$. It means that each dimension is defined by 10 bits. The choice of this value is empirical, in order to reach a tradeoff between memory occupation of the IDs and correct space occupation of the "ID cloud".

Given $d = 4$ and $b = 10$, each ID takes $d \cdot b = 40$ bits. Furthermore, each entry of the routing table at the overlay layer merely consists of the ID of the corresponding node. Depending on how the table lookup function is implemented, there can be some other parameters inside the table, but we only aim to achieve a general estimation.

We analyze the follwing cases:

- $d = 4$, having 32 entries of 40 bits each, leads to a table size of 1280 bits;
- $d = 5$, having 64 entries of 40 bits each, leads to a table size of 2560 bits; and
- $d = 6$, having 128 entries of 40 bits each, leads to a table size of 5120 bits.

This amount of memory is very poor, even for mobile devices.

5.5 End-to-end Latency

Whenever a data location request is issued by an host in the overlay network, the end-to-end latency measured for that request can be defined as *"the total delay from the moment in which the request is issued to the instant the final destination is found"*. This delay depends on how efficient the routing process is in the overlay network.

We define L_u to be the latency to route a message between two generic nodes A and B in the underlying network. When the address of B is known to A, L_u is the end-to-end latency for a message traveling from A to B.

When A does not know the address of B, we want the message to B to be sent using the capabilities of the overlay network. In that case, we define L_o as the latency for a message traveling from A to B and using only the routing capabilities of the overlay network.

As in terms of end-to-end latency the routing of the overlay network is not as efficient as the one of the underlaying network, we can expect that in most cases $\frac{L_o}{L_u} > 1$. We cannot state that this ratio is *always* greater than 1. In fact, the routing mechanisms of the underlying network are optimal with regard to the number of hops of the total path between the source and the destination of a certain message.

Figure 5.8 represents the ratio $\frac{L_o}{L_u}$ when the number of dimensions $d = 2$, 4 and 6. This graph has been obtained with 200 nodes in the overlay network and 10,000 nodes in the underlying network. The curves have been drawn by sorting the values of $\frac{L_o}{L_u}$ in increasing order.

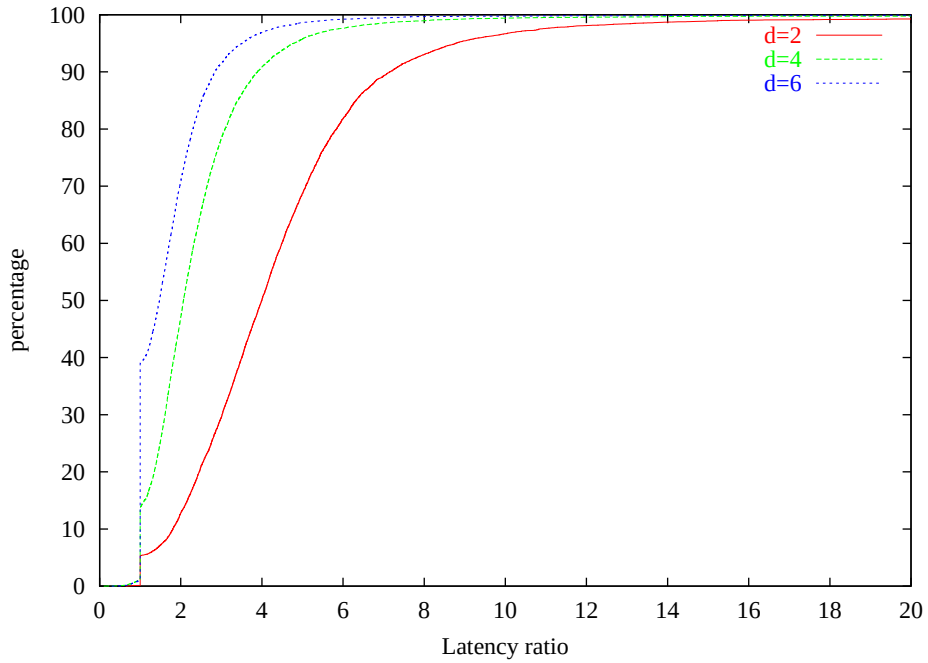


Figure 5.8: Overlay/Underlying network latency ratios for 2, 4 and 6 dimensions.

As we can notice, the overlay network performs well from the point of view of latency. The curves obtained with $d \geq 4$ show that the ratio $\frac{L_o}{L_u}$ is lower than 2 for most requests.

Note how there are even (on the extreme left) some values of $\frac{L_o}{L_u} < 1$: in these few cases the overlay network is even more efficient than the underlying one from the point of view of latency.

5.6 Input datasets

The network structure plays a fundamental role in the overall behavior of the simulation. In order to make our result significant, we examined different types of input datasets. In particular we found that two topology generators are suited for our purposes. Both of them can generate realistic large-scale scenarios that try to emulate the structure of the Internet. We need a kind of dataset that includes topology as well as some kind of "distance" metric. Currently we used latency as distance metric for our simulation but any kind of metric can be used, such as link cost, bandwidth or a combination of delay/cost/bandwidth, etc.

5.6.1 GT-ITM topology generator

The Georgia Tech Internetwork Topology Models [17] (GT-ITM) provides a great number of choices. It includes flat random graphs, n-level hierarchies, and 2-level transit-stub hierarchies. A flat random graph may be generated using a wide variety of edge probability functions. For an n-level hierarchy, the user specifies the number of nodes within each level and the type of probability function to use for connecting edges within each level. The user does not specify connectivity between levels. Rather, intra-level links are resolved into inter-level links; the degree of inter-level connectivity is determined by the edge probability function used within the upper level. For the transit-stub hierarchy, the user specifies a two-level transit hierarchy in the above manner, then lists the average number of stub domains per transit domain and the average number of extra transit-stub and stub-stub edges.

Its output format is a *sgb* file, and it is widely used as input file for NS [4].

5.6.2 *Inet* topology generator

The "Inet" generator[21] produces an AS-level representation of the Internet with qualitatively similar connectivity. It generates random networks with characteristics similar to those of the Internet from November 1997 to June 2000, and beyond. The generator should be used to generate networks of no fewer than 3037 nodes, which is the number of ASs in the Internet in November 1997. It provides connectivity information over a large-scale network and a weight for each link of the network.

We found this solution to be appropriate for our simulator. At the same time we also wrote a simple perl script to convert file formats from *sgb* to *Inet*, so that the GT-ITM generator can be used as well.

Figure 5.9 shows the latency distribution in a network with 10000 nodes generated by Inet. Latency values seems to have a mean around 100 ms as we expected. Most of values are within the interval $[50, 200]$ ms.

5.6.3 Self-collected data

Statistics [39] show that most latency peaks of point-to-point links in the Internet are between 50 and 150 ms. In our first implementation we assigned distance metrics among couples of nodes randomly. We were using a Gaussian distribution centered around 100 ms.

In order to obtain an idea of the current latency distribution in the Internet, we planned to collect latency information by ourselves. Having selected some few source points in several areas of the "planet", we run a simple program that is able to set

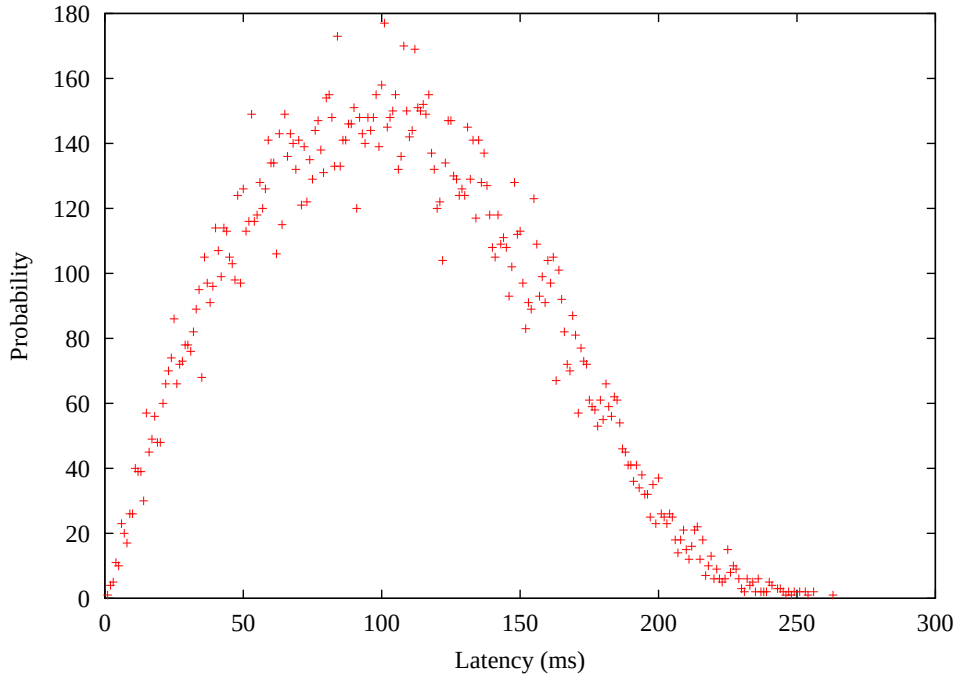


Figure 5.9: Latency distribution from the Inet topology generator.

up TCP connection from there. The program simulates the TCP/IP connection of a Web browser, and if there is a server waiting on the other side, it issues a few simple HTTP commands and then shuts down the connection properly. We are able to collect latency data about four types of TCP/IP interactions:

- delay between the first SYN message and the SYN-Ack message received;
- if the port is not opened on the other side, most systems answer with an ICMP message: "port domain unreachable";
- delay between the first DATA packet sent and the first Ack received;
- delay between the first FIN packet sent to shutdown the connection and the ensuing FIN received from the remote system.

Probing a randomly generated pool of IP addresses (eventually with multiple asynchronous queries) it is possible to dump this huge latency data into a single log file.

This file can be be analyzed to extract a possible correlation or a particular data distribution. In particular, for each address probed, we considered the minimum latency value (whenever more than one type of message are present) obtained among the four different message described above to be the most significant. Figures 5.10 and 5.11, show, respectively, the cumulative distribution function and the distribution function of latency data we collected[1]. The probing program has been run from five different location (the legend in figure shows the five hosts):

[1]Latency values greater than one second are not shown as they represents a timeout or network congestion.

- *athos-arl-wustl-edu* is in the US;
- *wanda-ch*, *pooky-granitsoft-ch* and *zurich.ibm.com* are in Switzerland, but use different ISPs;
- *sfo-openpksd-org* is in Japan.

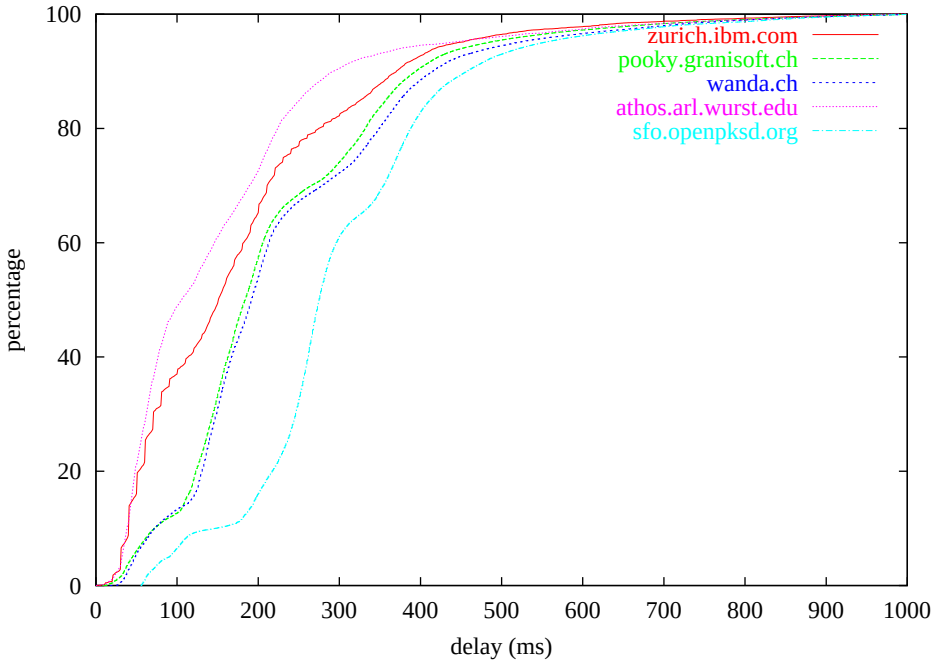


Figure 5.10: Cumulative Distribution Function of the four data sets collected.

As we can see, there are three peaks that can be isolated in all the four distributions. All but the Japan-related location data show peaks around 50, 160 and 350 ms. The data collected for the Japan location has the three peaks slightly shifted to the right, namely to 100, 280 and 380 ms.

On the one hand, Figure 5.10 shows that data collected for the Switzerland location are quite similar. On the other hand, if compared with the other ones, latency for the US location is lower and latency for the Japan location is higher. As expected we can conclude that the latency behavior depends more strongly on the geographic location than on the ISP connection.

We do not plan to go into details about this study, because the analysis of the Internet latency distribution is not part of this project. The aim of the simple program we developed was only to give us a general idea of point-to-point latencies around the Internet and not to perform accurate measurement in this fields.

Figure 5.12 shows the comparison of the latency distribution of real data collected with that of a topology of 10,000 nodes generated by Inet. The behavior of the Inet curve is similar to our data when the latency is less than 150 ms; for larger values the data from the Inet topology generator turns out to be “optimistic”, providing lower latency values than the mean we found.

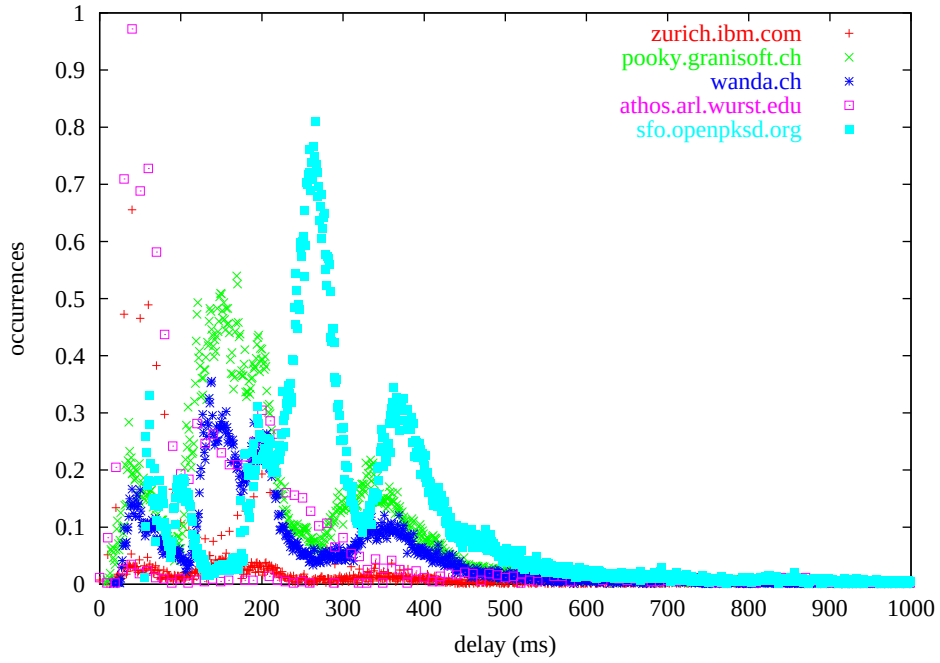


Figure 5.11: Latency distribution of data collected from the four different locations.

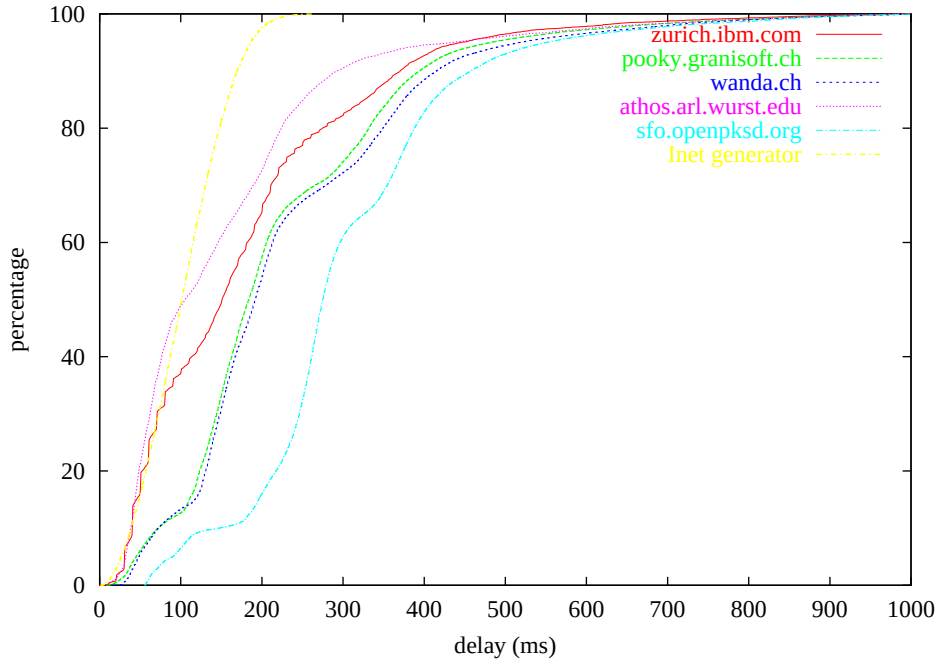


Figure 5.12: Comparison of Inet topology with datasets collected.

Chapter 6

Conclusions

6.1 Summary

Peer-to-peer overlay networks offer interesting possibilities to decentralized distributed applications. The main weakness usually associated to the concept of peer-to-peer networks is the need for some mechanism to locate data in a distributed environment. Distributed Hash Tables (DHTs) address and solve this problem efficiently.

Our work has been to design a distributed infrastructure based on DHTs schemes, in order to provide efficient and fault-tolerant routing, object location and load balancing in a self-organizing overlay network. Furthermore, our project's purpose is to achieve global routing capabilities even if the individual nodes only have a very limited knowledge of the network around them, that means:

- packet/message forwarding decisions are just local,and
- the information overhead due to the overlay layer is minimal.

We payed attention to one aspect that usually reduces the routing efficiency of overlay P2P networks: the lack of "locality preservation" during the routing process. We addressed this problem using the following concepts:

1. our overlay network maps to a d-dimensional hypertorus (each node has an unique identifier that belongs to this hypertorus structure);
2. routing is done by address in this space;
3. we provide an explicit network construction phase with the purpose to place each node at the best-suited location in the overlay network structure;
4. the network construction phase is strongly "locality aware", based on the measurement of some distance metrics in the underlying network (delay, bandwidth, cost or any combination of the previous ones);
5. the problem of establishing efficient routing is moved from the routing mechanism itself to the network construction phase: the routing efficiency depends only weakly on the routing process but strongly on "how good" the overlay network was built.

Fitting nodes into the hypertorus structure is a delicate phase that occurs whenever a new node wants to join the overlay network. Our design provides the means to guarantee the correctness, stability and efficiency of this mechanism.

Our simulations show significant improvements in end-to-end delays and path lengths, so that our overlay network is well suited for the development of new applications in a real "low-latency" environment.

6.2 Future Work

Below is a list of some key aspects related to our design that could be developed in the future:

- Various possible applications could take advantage of the data location infrastructure we focused on, in particular in the field of robust distributed storage protocols. In this case more capabilities should be integrated into our solution, such as handling multiple replicas of documents or taking care of security issues (cryptography, privacy, transactions, etc.).

- The quadrant structure we propose has numerous advantages from the point of view of routing, but also includes some nasty problems in the network construction phase. What we propose is just a possible solution to address the problem of building the overlay network correctly, but the problem remains open to further research;
- an open question is the study of the impact of underlaying network dynamics (e.g. frequent node JOIN/LEAVE) on the overall stability of our network;
- Our work actually consisted in the prototype implementation of a protocol; beyond our analysis and our simulations it would be intersting to test it with a distributed application over a real network or associated to some specific hardware such as network processors, gaming boxes, booster boxes [7], etc.

Bibliography

[1] BGP analysis. *http://bgp.potaroo.net*.

[2] The gnutella protocol specification v0.4. *http://www.clip2.com*.

[3] Napster protocol specification. *http://opennap.sourceforge.net/napster.txt*.

[4] Network simulator ns/nam. *http://www.isi.edu/nsnam/ns/*.

[5] OSPFv2. *IETF, http://www.ietf.org*, (RFC 2178).

[6] RIP (routing information protocol). *IETF, http://www.ietf.org*, (RFC 1528).

[7] D. Bauer, S. Rooney, and P. Scotton. Network infrastructure for massively distributed games. *NetGames 2002, Braunschweig, Germany*, 2002.

[8] Boris Beizer. Software Testing Techniques. *Van Nostrand Reinhold, New York*, (Second Edition), 1990.

[9] L. F. Cabrera, M. B. Jones, and M. Theimer. Herald: Achieving a Global Event Notification Service. In *Eighth Workshop on Hot Topics in Operating Systems, Elmau, Germany*, pages 87–94, 2001.

[10] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Freenet project at freenet.sourceforge.net*.

[11] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[12] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, (1):269–271, 1959.

[13] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pages 75–80, 2001.

[14] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, (42):44–51, 1984.

[15] Pierre Fraigniaud and Cyril Gavoille. Interval routing schemes. *Algorithmica*, 21(2):155–182, 1998.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns. *Addison-Wesley Publishing Co.*, 1995.

[17] College of Computing Georgia Tech. *http://www.cc.gatech.edu/projects/gtitm/.*

[18] E.N. Gilbert. Gray codes and paths on the n-cube. *Bell Systems Technical Journal*, 815-826(37), 1958.

[19] Hetzel and William C. The Complete Guide to Software Testing, 2nd ed. *Wellesley, MA. : QED Information Sciences*, 1988.

[20] http://javasim.cs.uiuc.edu. Javasim. *University of Illinois, Department of Electrical Engineering.*

[21] Cheng Jin, Qian Chen, and Sugih Jamin. INET topology generator. *http://topology.eecs.umich.edu/inet/.*

[22] Brad Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *Mobile Computing and Networking*, pages 243–254, 2000.

[23] P. Keleher and B. Bhattacharjiee. Are clouds of data pie in the sky? *http://motefs.cs.umd.edu/818/papers/motefs.ps.*

[24] M. P. Vecchi Kirkoatrick, C. D. Gelatt. Optimization by simulated annealing. *Science*, (220):671–680, May 1983.

[25] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummandi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. *Proceedings of ACM ASPLOS.*

[26] J. Li, J. Jannotti, D. De Couto, D. Karger, and R. Morris. A scalable location service for geographic ad-hoc routing. In *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking (MobiCom '00)*, pages 120–130, August 2000.

[27] Jorg Liebeherr and Tyler K. Beam. Hypercast: A protocol for maintaining multicast group members in a logical hypercube topology. In *Networked Group Communication*, pages 72–89, 1999.

[28] Myers and Glenford. The Art of Software Testing. *New York, Wiley*, 1979.

[29] NCSA. *http://archive.ncsa.uiuc.edu/SDG/Software/Mosaic/.*

[30] Northeastern University's College of Computer Science. Mars: Maryland routing simulator. *www.ccs.neu.edu.*

[31] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *ACM Symposium on Parallel Algorithms and Architectures.*

[32] Sylvia Ratnasamy, Paul Francis, Mark Handley, and Richard Karp. A scalable content-addressable network. *ACM SIGCOMM*, 2001.

[33] Sylvia Ratnasamy, Mark Handley, Richard M. Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In *Networked Group Communication*, pages 14–29, 2001.

[34] Sean C. Rhea and Wes Weimer. Data location in the oceanstore. *Final report for CS 262a*, (http://www.srhea.net/research/oceanstore/osdl-paper.ps), 1999.

[35] Antony Rowston and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large scale peer-to-peer systems. *Microsoft Research Ltd. Cambridge*.

[36] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[37] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM Sigcomm*, 2001.

[38] Department of Computer Science University of Virginia. Ant routing simulator. *http://www.cs.virginia.edu/ wz5r/cs655/proposal.htm*, 2001.

[39] Rich Wolski, Neil Spring, and Jim Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, (15 (5-6)):757–768, October 1999.

[40] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrasctructure for fault-tolerant wide-area location and routing. *Computer Science Division University of California*, (UCB/CSD-01-1141), April 2001.

[41] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatovicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, 2001.