



IP-FP6-015964

AEOLUS

Algorithmic Principles for Building Efficient Overlay Computers

Deliverable D6.1.1

State-of-the-art survey on overlay computers

Responsible Partner: University of Patras (EL)

Report Preparation Date: September 2006

Contract Start Date: 01/09/05 Duration: 48 months

Project Co-ordinator: University of Patras (EL)

Contents

1	Introduction	1
2	Peer-to-peer systems	1
2.1	Unstructured systems	2
2.1.1	Gnutella	2
2.1.2	Freenet	3
2.1.3	FastTrack/Kazaa	4
2.1.4	BitTorrent	5
2.1.5	Overnet/eDonkey	7
2.2	Structured systems	7
2.2.1	Content Addressable Network - CAN	7
2.2.2	Chord	10
2.2.3	Tapestry	13
2.2.4	Pastry	15
2.2.5	Kademlia	20
2.2.6	Viceroy	23
2.2.7	Conclusions	27
3	Distributed Computing	28
3.1	Grid	28
3.2	PlanetLab	32

1 Introduction

Technological advances and algorithmic suggestions in the recent years have led to an explosive growth of the Internet, with the World Wide Web being the driving force. Broadband Internet connections diminished the communication obstacles posed by geographic location and have paved the way for the creation of a new virtual neighborhood. A natural idea is to try to broaden the services offered by personal computers by using remote computers willing to participate and offer computational power, disk storage or useful information. The implementation of this idea has stirred up the interest of the scientific community, about how to accomplish this task in an efficient and transparent way, taking into account the vast heterogeneity that characterizes the Internet. In the AEOLUS framework, we suggest the introduction of an intermediate layer, the *overlay computer*, and plan to investigate the principles and develop the algorithmic methods for building such an Internet-based overlay computer. Although, to the best of our knowledge, the approach followed by AEOLUS is novel, several recent developments in the fields of peer-to-peer and network computing are related. These can be thought of as attempts in order to provide a form of an overlay computer but all of them are targeted at specific properties of an ideal overlay computer.

In this survey we present the current state-of-the-art in the field of overlay networks – computers focusing on systems based on peer-to-peer technology or distributed computing environments. First, we focus on peer-to-peer systems, where the main issues are topology control, fault-tolerance, adaptability to node connections and disconnections and routing. We present a number of popular cases and discuss about their proposed solutions to the above questions. Then, we switch to the field of networks aiming to provide an overlay network targeted at distributed computing, efficient resource usage and load balancing. We overview two popular technologies (namely Grid computing and the Planetlab testbed) which tackle questions also posed in AEOLUS but by following different approaches. We briefly present the main characteristics of each of them.

2 Peer-to-peer systems

We start by classifying several classes of peer-to-peer protocols. A first attempt towards such a classification is based on the degree of decentralization [1]. So, there exist *completely decentralized protocols*, as was Gnutella [16] in its initial form and Freenet [3]. All nodes in these networks have the same capabilities. There is no central server, on the contrary all nodes are server-clients (*servents*). These can be considered as the *pure* peer-to-peer systems. Their advantages are inherent scalability, load balancing and fault tolerance. A second class is formed by *hybrid decentralized* systems, e.g., Napster (see [19] for a brief introduction). In these systems, there exists a central server that assists in the interaction between the participating nodes by keeping a central index that contains the files shared by each participant. When a node is searching for a file, the central index is used in order to

find a suitable node carrying the file and thereafter communication can take place without the participation of the central server. In these systems a failure of the central server causes the failure of the entire network. Actually, this is how Napster was terminated. On the positive side, in this class of protocols access to the desired information is easier and searching takes place over a great range of the network. On the negative side, it is easier to apply censorship and an error in the central server can affect the entire network. Moreover, the central index is not always up to date. A third class contains *partially decentralized protocols* as Gnutella2. The main idea is similar to that of the pure peer-to-peer. The difference is that some nodes become “supernodes”. These supernodes have the extra ability to keep local indexes about the files that are shared by their “subnodes”. The main criterion to determine whether a node becomes supernode is its computational resources. We note that when a supernode fails, the network replaces it by another node, so there is no collapse. These protocols combine the advantages of the first two classes, i.e., the search time is less compared to pure peer-to-peer protocols while the load on the supernodes is less than the load of the central server in hybrid decentralized protocols.

Another way to partition the available protocols depends on the underlying structure; thus, they can be classified to *structured* and *unstructured* networks. In unstructured networks, the location where data resides is independent of the topology. Since we don't know where an object we are searching for exists, searching is being conducted in a random way by asking several nodes whether they have the desired object or not. These systems may differ in the way the topology is formed and the way searching is being conducted. Their advantage is that a sudden significant increase in the size can be accommodated. A drawback is that too many questions must be asked in order to track a desired object.

Structured networks (like Chord, CAN, PAST, Tapestry etc.) were established in an effort to deal with several problems that were inherent in unstructured networks, mainly because of the random search procedure. The topology is strictly controlled and the available data (or pointers to it) is placed in predefined location. Therefore, searching can use specific message routing algorithms. It is common for these networks to maintain a correspondence between an identifier, e.g., file id, in a distributed routing table, so that queries can be efficiently answered by a node holding the desired information. On the negative side, maintaining the required structure is difficult, since nodes connect and disconnect in a continuous fashion. Finally, there are *loosely structured* networks like Freenet. The location where data reside is determined by routing rules which are not strictly defines, therefore sometimes queries fail.

2.1 Unstructured systems

2.1.1 Gnutella

Gnutella [16] is an unstructured decentralized topology system and can be considered as a pure peer-to-peer system. Its internal structure is not fixed but can be modified. Of course, nodes and edges may remain the same but which of them form the network

is constantly changing. It combines an online virtual infrastructure over the physical network infrastructure. There is no central servers and nodes (“peers”) are connected through Internet to a host and send a “PING” message to all Gnutella hosts that are nearby. Those nodes respond with a “PONG” message which means they acknowledged the incoming node. If a user wishes to search for a file, then this request is forwarded to all nearby hosts. In case they have the file they respond and also forward the request to other hosts so that the whole network is progressively informed. The requesting user receives several answers (also called “hits”) and chooses which file he wants to obtain.

Each node (user) in Gnutella is simultaneously a server and a client, i.e., a Servent. Essentially, each node is also a network administrator obliged to forward queries and responds. When a node makes a query, this message obtains a unique 128-bit identifier called UUID and each node that receives the message stores the UUID and then forwards it to the next node. If a loop is formed and messages pass through a node for the second time, then the node does not forward the message. Also, when a node A responds to a query and sends the result, it checks its memory for the UUID and the node B that sent the message and sends a message that contains the initial UUID to B so that it can forward it to the originating node. Finally, each node that sends a message has a parameter TTL (Time To Live) which denotes for how much time the message will be forwarded, so that there is no network congestion.

Gnutella2 is a variant of Gnutella with a more recent architecture. Several applications are based on it including Kazaa, Morpheus, Limewire, Shareaza etc. It relies on the idea of supernodes, which aim to serve a small part of the network offering an index for the files that belong to this part. A mechanism is provided for online selection of supernodes that arranges the network in an interface between SuperPeers and clients. When a computer with sufficient computing power and a fast connection enters the network, it becomes a SuperPeer and establishes connection to other SuperPeers, thus forming an unstructured network consisting of SuperPeers. It also sets the number of clients that a node should serve in order to remain as SuperPeer.

2.1.2 Freenet

Freenet [3] is a decentralized topology network with a loose structure that behaves as a self-organizing system. There is no central server and is very similar to Gnutella. A user forwards a request for a file to the node it is aware of (usually the user’s computer). Requests are messages that can be forwarded to many nodes. The use of remote files is enabled, provided that a message “triggers” such a use. If the node does not have the file, then it forwards the request to the node it thinks has the biggest probability to own the file. The messages form a chain as they are forwarded from a node to another, but there is a limit to the size of these chains, therefore messages’ retransmission stops when either the maximum number of nodes in a chain has been reached or a node responds to the query. The answer is then forwarded to the originating node using the same chain. Each

node locally stores the reply so that it can immediately respond to new queries about the same file. So, it is possible to obtain data from a node that its existence was ignored.

Each node knows only which node forwarded the request to it, and not which originated the request, so a node can only know one or two nodes. This offers anonymity to the users, protects the files and since each file is uniquely specified by a key it is not possible for an attacker to add erroneous files. Though this anonymity is not perfect, since it is possible in theory by going from a node to another to discover which node originated the request, in practice this is impossible since it would require huge amount of resources and computational power in order to monitor the network. Anonymity was the primary goal behind Freenet's design.

It offers a good average case path length of $O(\log N / \log K)$ if every node has K links to other nodes, but the worst case behavior is rather bad. Overall, Freenet is an attempt to simulate the small-world phenomenon [7].

2.1.3 FastTrack/Kazaa

FastTrack [8] is a decentralized file-sharing system that supports meta-data searching. Peers form a structured overlay of supernodes to make search more efficient. Supernodes are peers with high bandwidth, disk space and processing power, and have volunteered to get elected to facilitate search by caching the meta-data. In order to be able to initially connect to the network, a list of supernode IP numbers is stored in the program. The client attempts to contact these, and as soon as it finds a working supernode, it requests a list of currently active supernodes, to be used for future connection attempts. The client picks one supernode as its "upstream" and uploads a list of files it intends to share to that supernode. It also sends search requests to this supernode. The supernode communicates with other supernodes in order to satisfy search requests. The client then connects directly to a peer to download the file; this transfer is done using HTTP. The ordinary peers transmit the meta-data of the data files they are sharing to the supernodes. All the queries are also forwarded to the supernode. Then, Gnutella-typed broadcast based search is performed in a highly pruned overlay network of super-peers. The peer-to-peer system can exist, without any supernode but this results in worse query latency. However, this approach still consumes bandwidth so as to maintain the index at the super-peers on behalf of the peers that are connected. The supernodes still use a broadcast protocol for search and the lookup queries is routed to peers and super-peers that have no relevant information to the query. Popular features of FastTrack are the ability to resume interrupted downloads and to simultaneously download segments of one file from multiple peers. Both Kazaa and Grokster are FastTrack applications. The file sharing application Morpheus originally utilized this network, but was later banished from it.

As mentioned, Kazaa is based on the proprietary FastTrack protocol which uses specially designated supernodes that have higher bandwidth connectivity. Pointers to each peer's data are stored on an associated super-peer, and all queries are routed to the super-

peers. Although this approach seems to offer better scaling properties than Gnutella, its design has not been analyzed. There have been proposals to incorporate this approach into the Gnutella network. The Kazaa peer to peer file sharing network client supports a similar behavior, allowing powerful peers to opt-out of network support roles that consume CPU and bandwidth. Kazaa file transfer traffic consists of unencrypted HTTP transfers; all transfers include Kazaa-specific HTTP headers (e.g., X-Kazaa-IP). These headers make it simple to distinguish between Kazaa activity and other HTTP activity. The Kazaa application has an auto-update feature; meaning, a running instance of Kazaa will periodically check for updated versions of itself. If it is found, it downloads the new executable over the Kazaa network. A power-law topology, commonly found in many practical networks such as WWW has the property that a small proportion of peers have a high out-degree (i.e., have many connections to other peers), while the vast majority of peers have a low out-degree, (i.e., have connections to few peers). Formally, the frequency f_d of peers with out-degree d exhibits a power-law relationship of the form $f_d \propto d^a$, $a < 0$. This is the Zipf property with Zipf distributions looking linear when plotted on a log-log scale. Faloutsos et al. [5] has found that Internet routing topologies follow this powerlaw relationship with $a \approx -2$. However, Gummadi et al. [?] observes that the Kazaa measured popularity of the file-sharing workload does not follow a Zipf distribution. The popularity of the most requested objects (mostly large, immutable video and audio objects) is limited, since clients typically fetch objects at most once, unlike the Web. Thus the popularity of Kazaa's objects tends to be short-lived, and popular objects tend to be recently born. There is also significant locality in the Kazaa workload, which means that substantial opportunity for caching to reduce wide-area bandwidth consumption.

2.1.4 BitTorrent

BitTorrent [13, 14] is a centralized peer-to-peer system that uses a central location to manage users' downloads. It is designed to distribute large amounts of data widely without incurring the corresponding consumption in costly server and bandwidth resources. This file distribution network uses a strategy known as *tit-for-tat* as a method of seeking. Under this strategy, a peer responds with the same action that its other collaborating peer performed previously. The protocol is designed in such a way so as to discourage peers to take advantage of the protocol and exploit it without further participation (these peers are called *free-riders*), by having the peers choose other peers from which the data has been received. Peers with high upload speed will probably also be able to download with a high speed, thus achieving high bandwidth utilization. The download speed of a peer will be reduced if the upload speed has been limited. This will also ensure that content will be spread among peers to improve reliability.

The architecture is based on a central location, where there is a tracker. When a user downloads a torrent file, it contains information about the file, its length, name, and hashing information, and URL of a tracker. The tracker keeps track of all the peers

who have the file (both partially and completely) and lookup peers to connect with one another for downloading and uploading. The trackers use a simple protocol layered on top of HTTP in which a downloader sends information about the file it is downloading and the port number. The tracker responds with a random list of contact information about the peers which are downloading the same file. Downloaders then use this information to connect to each other. A downloader which has the complete file, known as a seed, must be started, and send out at least one complete copy of the original file.

BitTorrent cuts files into pieces of fixed size (256 Kbytes) so as to track the content of each peer. Each downloader peer announces to all of its peers the pieces it has, and uses SHA1 to hash all the pieces that are included in the torrent file. When a peer finishes downloading a piece and checks that the hash matches, it announces that it has that piece to all of its peers. This is to verify data integrity. Peer connections are symmetrical. Messages sent in both directions look the same, and data can flow in either direction. When data is being transmitted, downloader peers keep several requests (for pieces of data) queued up at once in order to get good TCP performance. This is known as pipelining. Requests which cannot be written out to the TCP buffer immediately are queued up in memory rather than kept in an application-level network buffer, so they can all be thrown out when a choke happens.

Choking is a temporary refusal to upload; downloading can still happen and the connection does not need to be renegotiated when choking stops. Choking is done for several reasons. TCP congestion control behaves very poorly when sending over many connections at once. Additionally, choking lets each peer use a tit-for-tat-like algorithm to ensure that they get a consistent download rate. There are several criteria that a good choking algorithm should meet. It should cap the number of simultaneous uploads for good TCP performance. It should avoid choking and unchoking quickly, known as fibrillation. It should reciprocate service access to peers who let it download.

Finally, it should try out unused connections once in a while to find out if they might be better than the currently used ones, known as optimistic unchoking.

The currently deployed BitTorrent choking algorithm avoids fibrillation by only changing the peer that is choked once every ten seconds. It does reciprocation and the number of uploads are capped by unchoking the four peers with the best download rates and have interest in. Peers which have a better upload rate but are not interested get unchoked and if they become interested, the worst uploader gets choked. If a downloader has a complete file, it uses its upload rate rather than its download rate to decide which to unchoke. For optimistic unchoking, at any one time there is a single peer which is unchoked regardless of its upload rate. If this peer is interested, it counts as one of the four allowed downloaders. Peers which are optimistically unchoked rotate every 30 seconds.

BitTorrent does not offer its users anonymity and it is possible to obtain the IP addresses of all current, and possibly previous, participants. This may expose users with insecure systems to attacks. Another drawback is that BitTorrent file sharers compared to users of client/server technology often have little incentive to become seeders after they

finish downloading. The result of this is that torrent swarms (i.e., groups of peers connected with each other to share a particular torrent) gradually die out, meaning a lower possibility of obtaining older torrents. Some BitTorrent websites have attempted to address this by recording each user's download and upload ratio for all or just the user to see, as well as the provision of access to older torrent files to people with better ratios.

BitTorrent is typically best suited in continuously connected broadband environments. Dial-up users find it less efficient due to frequent disconnects and slow download rates.

2.1.5 Overnet/eDonkey

Overnet/eDonkey is a hybrid peer-to-peer protocol that is based on Kademlia [10] and is structured on a distributed hash table. Overnet [11] has no hierarchy, i.e., all hosts have identical functionality and consists of two layers and aims to provide an information storage network, which is formed by clients and servers that publish and obtain information by creating a file-sharing network. It supports several features that are attractive to the user like downloading a file from several users simultaneously, checking the integrity of a file using hashing etc.

When a new client joins Overnet, it randomly generates an ID for itself. This is the client's ID, and it remains unchanged on subsequent joins and leaves of the client until the user deletes the file containing the client's preferences. For lookup and routing purposes, each host maintains a list of neighbors and their IP addresses. The joining node should know the IP address and the port of another node (server) already existing in the network, and then it connects to the server and registers the files it wishes to share by providing some description. After the registration is completed, a client can initiate a search for a file either by querying the meta-data or by using the file identifier which is unique for each file. Servers provide the locations of object files when requested by clients, so that clients can download the files directly from the indicated locations.

2.2 Structured systems

2.2.1 Content Addressable Network - CAN

Introduction CAN is a pure structured peer-to-peer system. CAN stands for Content-Addressable-Network and is a general term used in order to describe a distributed hash table system. It was proposed by Ratnasamy et al. in [15], where they presented a specific design of a CAN protocol. The main actions are insertion, search and deletion of key-data pairs. Each node stores a part of the hash table, called *zone*, and is informed about nodes residing in neighboring zones. Requests for the main actions are routed through intermediate node until a node whose zone contains the key is reached.

CAN is fault-tolerant since routing can take place even when some nodes have failed. Moreover, nodes store only a small part of the entire state, independent of the total nodes in the system. Each node contains an index for some keys and the corresponding nodes, in

contrast to central indexes where all available information is stored to one node. CAN is a scalable network, since it behaves well even if the number of nodes/messages increases.

CAN uses a virtual d -dimensional coordinates space that corresponds to a d -dimensional torus in order to store a pair consisting of a key K and a value V . This space is segmented in a dynamic way so that each node has a zone. Initially, K is mapped in a deterministic way to a point P of the space. The pair (K, V) is stored in a node that has the zone where P is located. In order to retrieve the entry that corresponds to K , each node can apply the same deterministic procedure to map K to P and then retrieve the corresponding value V from P . If P is not stored in the node to the request, then this request must be routed from node to node until it reaches the node in whose zone P lies. Nodes find out and preserve the IP addresses of are neighboring, in order to enable routing between arbitrary points in the space.

Routing Intuitively, routing in CAN is being conducted by following the straight-line path in the space from the source to the destination. Each node holds a routing table containing information about its neighbors in the space (the IP address and the zone for each neighbor). The neighbors of each node are those nodes that have the same coordinates as the node in contains the coordinates of the destination. Each node routes in a greedy way, forwarding the message to its neighbor that is nearest to the destination. The average path length for a d -dimensional space partitioned in n equal zones is $(d/4)n^{1/d}$ and each node has $2d$ neighbors.

Node connections and disconnections When new nodes enter the system they are assigned part of the space by splitting an existing zone in two. The steps are as follows:

- the new node recognizes an already existing node in CAN using some bootstrap mechanism.
- using CAN's routing mechanism, it randomly picks a point P in the space and sends a JOIN request to the node in whose zone P lies. This zone will split and half of it will be assigned to the new node.
- the new node learns the IP addresses of its neighbors and the neighbors of the zone that was split are notified about the existence of the new node.

A node can find out the IP address of any other node in the system. The initial one maintains a list of nodes that it estimates that are in the system. In order to enter CAN the node should look in the DNS for the domain name in order to retrieve the IP address of the initial node. The initial node can provide the IP addresses of several arbitrary nodes.

When some node exits CAN, the zone it possessed and the relevant entries in the hash table are transferred to a neighbor. Under normal circumstances, a node periodically sends synchronization messages to each of its neighbors providing the coordinates of its zone, a list of neighbors and their coordinates. If there is a lasting absence of a synchronization

message, the neighboring node understands that there is failure and enters a controlled takeover mechanism. If many neighboring nodes fail, an extended ring search mechanism is initialized by a neighbor in order to find a working node outside the region that contains the failures.

Possible improvements The number of steps mentioned above refers to each path in CAN. These are called application level steps and are different than the IP level steps. The delay in each step is different, because it is possible that two nodes are neighbors in CAN but are in completely different geometric regions (i.e., many IP level steps). Therefore, the average delay for a query is the product of the average number of CAN steps with the average delay in each step. Apart from the main design, some improvements are proposed in order to reduce the total routing delay. By increasing the number of dimensions of the coordinates' space, the length of the routing path is reduced and so does the delay. This comes at a price of a slight increase in the size of the routing table. Fault tolerance is also improved since the number of neighboring nodes is increased; thus, a node has more options to consider when there is a node failure.

Another idea is the use of multiple "realities". We can assume that each node belongs to many independent spaces and to a different zone in each coordinates' space. Each coordinates space is called a "reality". In a CAN system with r realities each node stores r independent sets of neighbors. Assume that a pointer to a file is being stored in the coordinates' space (x, y, z) . With r realities, this pointer will be stored in r different nodes that correspond to the coordinates (x, y, z) in each reality and therefore will be unavailable only if all r nodes are unavailable; thus, leading to an improvement of data availability. Regarding message forwarding, a node checks all of its neighbors in each reality and forwards the message to the neighbor that is closer to the destination. Each node has more options and, therefore, the length of the path is reduced. Moreover, fault tolerance is enhanced since in case of a node failure messages can be retransmitted through other realities.

Routing can be improved if we take into account the underlying IP topology and the connection delay as well as the distance between source and destination in the space. Each node counts the round-trip time at the network layer for each of its neighbors. For a given destination the message is forwarded to the node with the largest progress to RTT ratio. Thus, we can reduce the delay for the independent steps.

Up to now, we have assumed that a zone is tied to a node. We can modify this assumption and allow many nodes to share the same zone; these nodes are defined as peers. Under the new setting, each node maintains a list of its own peers as well as a list of its neighbors. A node should know all peers of the same zone, but it is not required to know all of its neighbors. When a node enters the system, it finds the node whose zone it would take over. Instead of splitting it in two parts as above, it also enters the same zone. If too many nodes are already in this zone then it is split. This modification has several advantages, like smaller path length since placing many nodes in a zone has the

same effect as the reduction of the number of nodes in the system. Even the per step delay is reduced because a node now has several options in choosing its neighboring nodes and, therefore, chooses a neighbor that is close to the destination. Finally, the system can handle more failures, since a zone fails if all of the corresponding nodes fail. This improvement requires each node to maintain a larger list of nodes.

We can use multiple hash functions that map the same key to several different points in the coordinates' space. Then, a key is unavailable when all nodes that possess it are simultaneously unavailable. Queries for a particular entry in the hash table are sent in parallel to all nodes, thus reducing the average delay. However, this leads to an increase of the size of the hash table and of the traffic in the network.

CAN assigns nodes to zones in an arbitrary way, so it is possible that two neighboring nodes are at a distance in the IP network; therefore, it was suggested that the CAN topology should be close to the IP topology. This approach uses the assumption that there exists a set of machines that behave as *landmarks* on the Internet. Each node sorts the landmarks according to the time it would require to reach its actual position starting from each landmark. Hence, nodes that are topologically close should be also close in the space. For this reason, we use the ratio of the delay in the CAN network over the average delay of the IP network; this ratio is called *stretch*.

When a node enters the system, an already existing arbitrary node assigns a zone to it. This node knows the coordinates of its zone and of its neighbors. It compares the volume of its zone to the volume of its neighbors' zones and the zone with the largest volume is split and half of it is assigned to the new node. The volume of a zone is proportional to the size of the database containing the key-value pairs that are stored in a node, and therefore proportional to the workload of the node. There are cases where this technique is not sufficient for an efficient load balancing, e.g., when some keys are more popular the corresponding workload is greater. This problem is similar to the hot-spot problem on the world wide web. In order to solve this problem we can use some caching and replication techniques that are used in the Web. In caching, each node stores in a cache the keys it has recently accessed. If a query is for a key that is in the cache, then it can be served by the node. Thus, popular keys are stored in more than one nodes. In replication, when several queries arrive at a node about a particular key, then that key is copied to the neighboring nodes.

2.2.2 Chord

Introduction Chord [20] is a structured decentralized network that can be easily scaled. It assigns a key to each node and each data, that includes an address, a document, a file or any collection of items. Tracking is taking place by relating the key to data and storing the data/key pair to the node that corresponds to the key (e.g., in the case of a file, the key is the filename and the data is the actual file). Chord offers an efficient solution to the following problems that are typical in peer-to-peer systems.

- Load balancing: It uniformly assigns the data/key pairs to the nodes.
- Decentralization: All nodes have the same capabilities and there is no need for a central server.
- Scalability: The cost for searching in Chord increases as the logarithm of the number of nodes, and, therefore, is efficient also for large systems.
- Efficiency: It can handle frequent and extensive input and output of nodes.
- Flexible domain name: It creates a large and well balanced domain name for the keys, so there are no restrictions when naming the data.

Applications mainly contact Chord in two ways. The first is that Chord uses an algorithm that finds the IP address of the node that corresponds to a key, i.e., enables the application to find the node where some required data are stored and the second is that Chord notifies each node about possible changes in the key group that the node is in charge of, so that the application can transfer the corresponding data to the appropriate nodes or remove it from deleted nodes.

Applications are responsible for the use of Chord capabilities for verifying, storing and copying the data. The following classes of programs can efficiently use Chord: Cooperative Mirroring, Time - Shared Storage, Distributed Indexes, Large - Scale Combinatorial Search.

The protocol Chord supports just one functionality; given a key, it maps the key to a node. An application that uses Chord can utilize this functionality in order to store to that node some data that corresponds to the key. Chord uses a hash function, SHA-1 (Secure Hashing Algorithm), to map each IP address and each key to an m -bit identifier. m is sufficiently big so that the probability that two different keys or nodes have the same identifier is infinitesimal. Given the IP address of a node we can obtain the identifier of the node and we can use the key string in order to obtain the key identifier. The network is formed as a sorted ring of the node identifiers modulo 2^m , i.e., when the identifiers are m bits long then the network has size $2^m - 1$ (from 0 to $2^m - 1$). The main rule is that we store each key to the successor of the node, i.e., to an active node whose identifier is greater or equal to the key identifier. When the identifiers are part of a ring, then the successor of a node k is the first clockwise active node on the ring.

When a new node n enters the network, some of the keys corresponding to $\text{successor}(n)$ are transferred to n , while when a node disconnects its keys are transferred to $\text{successor}(n)$. It holds that in a network with N nodes and K keys with high probability each node stores at most $(1 + \epsilon)K/N$ keys and when node $N + 1$ enters or exits the network $O(K/N)$ keys are transferred, where ϵ is bounded by $O(\log N)$ and can be reduced to a very small value if each node creates $O(\log N)$ fictitious nodes with new identifiers.

Searching and routing Searching for a key can be done in several ways. The simplest but slowest is to search the ring in a clockwise manner from each node to its successor until the node that contains the desired data is found. The only required information each node should possess is the identifier of its successor. In order to make searching more efficient, Chord supports another algorithm which demands that each node keeps additional routing information in a table called *finger table*. Each node n keeps a table of m fingers in such a way that the i -th place in the table corresponds to the successor s of the $(n + 2^i)$ -th node (i.e., the first active node at a distance at least 2^{i-1}). In other words, $s = \text{successor}(n + 2^{i-1})$, with $1 \leq i \leq m$. We call node s as the i -th finger of n and define it as $s = n.\text{finger}[i]$. We note that the first finger is the successor. The idea is that searching is forwarded not to the next active node, like in the previous simple algorithm, but to a node which is near the desired one and we have available information about it. If there is available information for many nodes, then searching is more efficient. The function that determines the nodes that are contained in the finger table (i.e., $s = \text{successor}(n + 2^{i-1})$) is efficient since it accelerates searching by keeping additional information for a small number of nodes and since it distributes this information so that more is stored near the desired node and less away from it. To sum up, a node searches its finger table for a node whose identifier is the largest among those that are at most equal to the key identifier. We can prove that with high probability the number of nodes that should be checked in a network of N nodes is $O(\log N)$.

Online behavior Chord supports frequent connections and disconnection of nodes without a system collapse. In order to guarantee stability it uses a protocol that relies on each node's knowledge of its successor and also periodically updates the finger tables. If new nodes enter and a search is required before the system is rearranged there are three cases. The most usual is that the finger tables are correct, so the query is accomplished in $O(\log N)$ steps. Another case is that the finger tables are not up to date but the successor nodes are correct. In this case, search is efficiently handled albeit in slightly more steps. The last case is when the successor nodes are also incorrect and then search might fail. In that case, the protocol can repeat the search after a slight interval (since the time required by Chord to update the tables is small). It can be proven that after some nodes have entered a ring is formed where each node knows its successor. It can also be proven that if the finger tables are not updated, the efficiency of search is not reduced by much, since it is not very important which are the exact nodes that are in the finger tables, but only their ability to forward the query from the issuing node. Therefore, search again takes $O(\log N)$ with high probability.

A node can exit the system either voluntarily or by force. If it exits voluntarily, the disconnecting node transfers all of its keys to its successor and notifies the successor and the predecessor for its exit. Forced exits may lead to a system collapse, therefore each node should keep a list of r successors. If a successor is not responding during a query, then it is replaced by the next in the list. A suitable value for r is $2 \log N$. In this case,

the system collapses only if all r nodes fail. It can be proven that if a list of $r = O(\log N)$ successors is used in an initially stable system and each node exits with probability $1/2$, then with high probability the search for the successor of a key finds the nearest active node of the key in $O(\log N)$ time.

In conclusion, Chord offers a completely decentralized protocol that supports easy store and search of data and can sustain a significant change in the number of nodes.

2.2.3 Tapestry

Introduction Tapestry [21] was designed by Ben Zhao, John Kubiatowicz and Anthony Joseph from Berkeley University. It offers a routing architecture for a self-organizing, scalable, robust, large-scale infrastructure that can successfully route requests under high load and node failures. Tapestry provides decentralized storage and routing of objects (DOLR – Decentralized Object Location and Routing) that focuses on message routing to the terminals (like nodes or object copies). Together with Pastry, Chord and CAN, Tapestry belongs to the second generation of peer-to-peer protocols that guarantee a limited number of steps for routing and can scale and self-organize. Pastry and Tapestry utilize the locality information they maintain so as to guarantee the shortest possible path for message routing. Tapestry allows applications to store objects according to their needs, in contrast to some systems that impose restrictions on the number and the place of the objects.

Algorithms For every node or object, Tapestry creates identifiers from a large range using uniform distribution (NodeIDs and GUIDs - Globally Unique Identifiers respectively). Nowadays, Tapestry uses a domain with identifiers of 160 bits in a globally defined base, (e.g., hexadecimal, that allows Ids with 40 hexadecimal characters). NodeIDs and GUIDs are constructed in a distributed and uniform way using a hash function, e.g., SHA-1 (Secure Hashing Algorithm). An application identifier (*Aid*) is also defined as a parameter in each message, a fact that allows many applications to share the same network. That is, every message is targeted at a specific application using the corresponding identifier as a parameter.

The functions that Tapestry uses to communicate with the applications are

- PUBLISHOBJECT(OG,Aid): Announces object OG to the network and makes it accessible.
- UNPUBLISHOBJECT(OG,Aid): Withdraws object OG from the network.
- ROUTEOBJECT(OG,Aid): Routes a message to the location of object OG.
- ROUTEONNODE(N ,Aid,Exact): Routes a message to application Aid at node N .

Tapestry's routing and object tracking mechanism is as follows. Tapestry maps to each identifier G an active node N that is called *root* of the identifier and is denoted by G_R .

That means that if a node N exists with $Nid = G$, then N is the root of G and routing in order to find a node whose identifier is G ends when node G_R is discovered. We note that if G is an object identifier, node N does not store the object itself, but knows the nodes where the object is stored.

Each node stores a routing table that contains the identifiers and the IP addresses of all nodes that it can contact. These identifiers are organized in levels according to their distance from N ; this fact is reflected on the digits of the identifier. Those that differ only in the last digit are the nearest ones, then it is those that differ in the last two digits and so on. In every level, nodes from all possible values in the digit that is different in that level should exist in the routing table. In general, the content of the i -th position in the j -th level is the closest node whose identifier starts with the prefix $(N, j - 1) + i$.

The routing protocol is as follows: each node checks its routing table and finds a node at its level that has an identifier which is closer to the message destination. If no active node is found that matches exactly, then it finds the nearest. The number of steps required for routing is $O(\log_\beta N)$, where β is the base of the identifiers. This procedure is called *surrogate routing* and replaces an inactive node with an active having a similar identifier.

In order to maintain an effective network even under node failures, each node apart from its routing table also stores some additional nodes starting with the same prefix. If there are c nodes, at step n the neighboring nodes that are kept in the table differ only in the n -th bit, therefore there are β such nodes for a total of $c \cdot \beta$ nodes, while the total size of the table is $c \cdot \beta \cdot \log_\beta N$. Each node also stores the nodes that are connected to it (*backpointers*); these nodes are also $c \cdot \beta \cdot \log_\beta N$.

One of Tapestry's properties that distinguishes it from Chord is that the node G_R that an object identifier points to, is not the node that has the object but the node that knows the node where the object is stored. This provides flexibility to the applications with respect to object storage, multiple copies etc.

To track an object, Tapestry has a *Publication* procedure that we describe in the following. A server S that has an object O with an identifier O_G and a root node O_R periodically announces the object by sending a special message to the root node. Each node on the path that this announcement traverses stores the server's location. When there are copies of the object in different servers, each of them receives a message. Furthermore, when a node receives messages from different servers for the same object, it stores the servers' position in a sorted way with respect to their distance from it.

When a client asks for object O , it sends a message to O 's root node, i.e., O_R . Each node that receives the message checks if it has already stored information about O and in case it does, this node forwards the message to the nearest server it had previously stored. In the extreme case, the message does not pass through such a node and reaches O_R that has definitely stored the location of the server because the messages that are created during Publication must reach O_R . In practice, the message meets a suitable node rather early and in this way Tapestry uses the locality information that it stores, so that the nearer a node is to the server, the faster the message arrives.

Tapestry also allows online connection and disconnection of nodes. The algorithm for connection is not simple and each addition might need a significant time interval. The following constraints should be satisfied in the case of the addition of a node N . Some nodes should be notified about the new connection and N should become the root node for some objects whose identifier is close to N . Moreover, a valid routing table should be created and stored at N . Finally, neighboring nodes are notified so that they modify their routing tables, if necessary.

The complete procedure is as follows: N finds its surrogate node S (i.e., the node that a message with the identifier N_{id} goes in the not yet updated network). S finds p , i.e., the number of common digits of its prefix with N_{id} 's prefix. Then, S sends an Acknowledged Multicast message that reaches all nodes with the common prefix. These nodes add N to their routing tables and send reports for local object root nodes in order to satisfy the first two constraints. The nodes that receive the message are connected to N and become the initial nodes that form N 's routing table. N searches for its neighbors starting from level p , finds the k nearest neighbors at this level and fills its routing table accordingly (k is a coefficient chosen according with the network settings). Then, N asks these nodes to send it the nodes that are connected to them (backpointers) at the same level. Afterwards, N applies the same procedure to other levels until its routing table is complete and the third constraint is satisfied. Nodes that are contacted by N update their routing tables if necessary, so that the fourth constraint is also satisfied.

When a node N wishes to exit the system, it should notify all nodes that are in its backpointer list and also send them the nodes that will replace it at each level. N also sends a message to the objects for which N is root node and notifies them for their new root nodes. In order to cope with forced exit from the system, there should be redundant information in the routing tables and in the reports about the location of the objects. For this reason, nodes periodically send messages to their neighbors to check for failures. In case a failure is detected, they adjust their routing table and start reassigning the stored information about objects' location.

In conclusion, Tapestry is a nice second generation protocol that allows changes in the number of nodes and exploits locality. On the negative side, node connection is rather difficult and, therefore, Tapestry is not a suitable architecture for constantly changing networks.

2.2.4 Pastry

Introduction Pastry [17] was developed by Antony Rowstron and Peter Druschel in the framework of Microsoft's project PAST. It is completely decentralized, fault tolerant, self-organizing and can adjust itself to cope with node connection, disconnection and failure. It forms an underlying network for large scale peer-to-peer applications and provides distributed object location and routing mechanisms. Applications exploit those mechanisms in several ways, e.g., PAST creates a fileId for each file by using the filename and

the owner's name using a hash function and stores copies of the file to k nodes in Pastry that have numerical Ids as close as possible to the fileId. If a client wishes to search for a file, then it sends a message having the fileId as a key. Pastry will, unless all k nodes have failed, route the message to one of those k nodes at which the file is stored. Apart from locality, Pastry also guarantees that the message will reach a node among the k nodes that is closer to the client.

Design Each node in Pastry has a unique identifier of 128 bits called nodeID (i.e., the domain is from 0 to $2^{128} - 1$). This identifier is randomly decided in a uniform distributed way when a node enters the system. This random way (using a hash function on the IP address) implies that nodes with neighboring nodeIDs can be geographically distant. When a message and a key appear, one node routes the message to a node having nodeID as close to the key as possible among all active nodes. In a network with N nodes, routing uses less than $\log_{2^b} N$ steps (where b is a configuration parameter; a typical value is 4). The protocol guarantees that routing will succeed even if $L/2$ nodes with neighboring nodeIDs fail simultaneously (L is a configuration parameter with typical values being 16 or 32). In order to facilitate routing, each nodeID and each key are represented in a 2^b -base alphabet.

Routing Routing is conducted from each node to the one with the one having a nodeID as close as possible to the key. This can be done by organizing the nodeIDs as digits. When a message arrives at a node, the key is compared to the nodeIDs of all known nodes and the message is forwarded to the node that has at least $n + 1$ common digits in its prefix with the key, where n is the number of common digits that the prefix of the current node's nodeID with the key. For example, if a message with key = 10221103 arrives at a node with nodeID = 10233102 and there is a discovered node with nodeID = 10222302, then the message is forwarded to the second node. If no such node exists, then the node with n common digits in its prefix is found. Therefore, each node keeps 3 routing tables (a *leaf set*, a *routing table* and a *neighborhood set*).

The routing table consists of $\log_{2^b} N$ rows and $2^b - 1$ columns and contains the nodeIDs and the IP addresses of the nodes. The n -th row (where the first row is the 0-th) contains all nodes whose nodeID's prefix has n common digits with the node's nodeID, while the next element is the number of the column. The neighborhood set M contains the nodeIDs and the IP addresses of the nodes that are geographically closer to the node. These are not used for routing but to determine some of the local settings for the node. The leaf set L is the $L/2$ nodes with nodeIDs that are close to the node's nodeID from below and the $L/2$ nodes with nodeIDs that are close to the node's nodeID from above. This list is used to route messages when no node with $n + 1$ common digits is found. Typical values for M and L are 2^b or 2^{b+1} .

A slight description of the routing algorithm follows. Initially, we check if the key is in the leaf set. If this is the case, we forward the message to a node in the leaf set whose

nodeID is as close as possible to the key. Otherwise, we follow the procedure described above for the routing table, i.d., we find the node whose nodeID has at least $n + 1$ common digits in its prefix with the key. If we cannot find such a node, we search for one with n common digits. This algorithm guarantees that the message is correctly forwarded since at each step it is sent either to a node with more common digits or to a node with the same number of common digits but numerically closer to the key. Given updated routing tables and assuming that there are no recent node failures, this algorithm requires $\log_{2^b} N$ steps. In case of node failures, in practice the number of steps increases proportionally to the number of nodes that fail.

Pastry's API Pastry responds to these commands from the applications.

- `nodeId=pastryInit(Credentials,Application)`: Connects a node to the network, performs all initializations and returns the nodeID. Application is a handle that enables the node to contact the application.
- `route(msg,key)`: Routes the message to the node having a nodeID as close as possible to the key.

The applications using Pastry should be able to respond to the following commands.

- `deliver(msg,key)`: It is used when a message arrives and forwards it to the application of a node having a nodeID as close as possible to the key.
- `forward(msg,key,nextId)`: Notifies the application before it forwards a message to the next node.
- `newLeafs(leafSet)`: Pastry notifies the application for any change in the node's leaf-set.

Several application have been deployed based on this simple environment.

Node connection and disconnection When a node enters Pastry, it should know a nearby node A that is already in Pastry. It is then assigned a nodeID X (given not by Pastry but by the application using a hash function taking the IP address as input). X sends to A a special connection message having X as key and Pastry routes this message to the node with the nearest nodeID to X . Since X is not yet connected, it finds the numerically nearest node; let Z be this node.

Z and all nodes in the path traversed by the message send to X their routing table and X uses this information to initialize its tables. More specifically, since A is a neighboring node, X uses the neighborhood set as its own neighborhood set. Since Z is the numerically nearest node to X , X uses Z 's leaf set as its own leaf set. Finally, X uses A ' routing table.

Let's assume that X 's and A 's nodeIDs have no digit in common. Then, row 0 in A 's table contains nodes with no digit in common with A and some of them have no digit in

common with X ; therefore, they can be used for X 's row 0. The first node in the path from A to Z has the first digit as X and therefore some elements in the first row of the routing table can form the first row of X . Applying the same reasoning, we get that the second row of the second node in the path from A to Z can be used to form the second row of X 's routing table and so on. After this procedure is completed, X notifies all nodes in its leaf set, its neighboring nodes and those in its routing table for its existence, so that they can update their tables.

A forced exit of a node is being noticed by the system when a node cannot contact it. Nodes that contain the exiting node in their leaf set should replace it and therefore they contact the active node in their leaf set that has the largest index on the direction of the missing node. A list L' is obtained by that node and possible L' contains nodes that are not part of L . Among the nodes in L' a suitable is chosen in order to replace the missing node in L . This procedure guarantees the replacement of a node, unless $L/2$ nodes having close nodeIDs fail simultaneously; a rare event. When a node fails and is part of a routing table, Pastry can continue forwarding messages but the node should be replaced in the routing table.

Locality Locality is a very interesting feature of Pastry. Locality implies that the path chosen for routing is good compared to a distance metric. Such a metric could be the hops between the IP addresses or the geographical distance between the nodes. Applications running over Pastry should have a function to estimate this metric, which is also supposed to be Euclidean, i.e., the triangle inequality holds. This is not always the case (consider for example the number of hops in the IP addresses), but we will use this assumption in the following, where we show that the routing table exhibits the locality property.

As we have already mentioned, when a node X enters the system, it discovers a neighboring node A . X sends to A a special connection message having X 's value as the key. This message traverses a path from A to B etc. until it reaches a node Z which is numerically closer to X . Locality stems from the fact that the nodes in the routing table (that is formed using the i -th row in the routing table of the i -th node in the path) are close to node X . We assume that locality holds for all nodes that were in the system prior to X 's connection. Since A is close to X , it holds that nodes in row 0 of A 's routing table that are used to form row 0 of X 's routing table are close to A and therefore close to X (because of the triangle inequality). Consider another row in the routing table and assume that it was formed by B 's routing table. Then, the distance between B and a node B_1 in B 's routing table is small, but we do not know which is the distance between B and X and, hence, we cannot estimate the distance between B_1 and X . Nonetheless, it holds that the distance between any pair of nodes in each level of common prefix increases exponentially, therefore the distance between B and B_1 is much larger than the distance between A and B and consequently from the triangle BXB_1 since $B - B_1$ is much greater than $B - X$ and $X - B_1$ is large enough; roughly the same size as $B - B_1$. Finally, after X is connected to the network, it checks the routing tables of its neighboring nodes and

the nodes in X 's routing table in order to update its tables and replace some nodes by others that are closer to it.

Locality of routing stems from the locality of the routing tables. Since nodes in the routing tables are close to the node keeping the tables, the message is being forwarded to its destination following the hop of shortest length at each step. Note that this is different than following the shortest path. We can exploit two facts to solve this problem. First, given a routing from A to B (which is at a distance d from A), the message cannot be routed to a node which is at a distance less than d from A . Second, the distance traveled at each step increases exponentially. This holds because at each row l , the nodes that can be part of the table are in a domain of size $N/2^{bl}$, i.e., the size is exponentially reduced at each row and since these nodes are uniformly distributed, their distance from each other increases exponentially at each step. Therefore, hops get larger and larger and therefore the message cannot return to a previous node. This way we can guarantee locality in routing.

Some applications using Pastry, like PAST, copy this information into many (k) nodes that are close to a key (PAST stores files in a way that enables it to use them even if several nodes fail). Pastry by default routes a message to the nearest active node and guarantees that it will reach its destination even if only one out of k nodes is active. Moreover, because of the locality the message will reach a node which is close to the destination. Since Pastry uses local information during routing and because it is based on the fact that the prefixes of the nodeIDs are similar, these k nodes are approximately close. In the worst case, $k/2 - 1$ from the copies are stored in nodes whose prefix has no common digit with the key. Pastry uses a self-learning procedure to overcome this problem. According to this procedure, Pastry learns the time required for the message to reach the nearest of the k copies and routes the message to this node 75% of the time, while it routes it to one of the two nearest nodes 91% of the time.

Drawbacks Two are the main drawbacks in Pastry. First, the case where nodes have not failed but continue reacting to messages, albeit in a wrong way. Because routing is well specified, in such a case repeated questions will pass through the same path and consequently will fail to find the desired node. This problem can be solved if when the next node is decided during routing, this is not deterministically but using randomization between the nodes whose nodeIDs has at least $n + 1$ common digits in their prefix with the key (where n is the number of common digits between the nodeID's prefix and the key).

The second problem is irregularities in IP routing over the Internet that allow some nodes to be reached by some nodes but not by some others. A solution consists of a random search through multicast that Pastry's nodes conduct in their neighborhood. This search can take place when the network is idle.

Conclusions To sum up, Pastry along with Tapestry, Chord and CAN belong to the second generation of peer-to-peer protocols that guarantee a limited number of steps in routing, scalability and self-organization. Pastry and Tapestry take advantage of locality information so that the message routing path is minimized.

2.2.5 Kademlia

Introduction The main idea of this system is a distributed hash table that performs well in environments that are prone to errors. The routing of the queries is conducted through a topology based on a XOR metric. This topology has the property that each nodes “learns” from the queries it receives and thus strengthens the decision making process used for routing. The number of steps is $\log_2 n$. If we assume that the IDs are of size b bits instead of one bit, then the size of the routing table increases to $2^b \cdot \log_2 n$ k -buckets and the number of steps is reduced to $\log_{2^b} n$.

Description In each node we assign an ID of 160 bits. We view the nodes are leaves in a binary tree. The location of a node is defined by the smallest unique prefix of its identifier. For a particular node, we split the tree into consecutive subtrees that do not contain the node. Initially, the subtree at the higher level contains the half binary tree that does not contain the node. The next subtree contains the half subtree from the other half binary tree and does not contain the node, and so on. Kademlia guarantees that in these subtrees each node knows at least another node. Thus, each node can learn the location of another node if it knows its ID.

XOR metric Each message that gets transmitted contains the ID of the sender so that the recipient learns about the existence of the sender. Apart from the nodes, also the keys are of size 160 bits. Kademlia exploits the distance between two identifiers during the keys assignment. This distance is defined as the logical XOR over integers, $d(x, y) = x \oplus y$, where x, y are the identifiers.

In a binary tree of 160-bit identifiers, the distance between two identifiers is the height of the smallest subtree that contains both. When the tree is not complete, the leaf that is nearer to an identifier x is the leaf whose identifier has the largest common prefix with x . XOR is unidirectional in a similar way to the fact the Chord’s ring is traversed clockwise. So, we are assured that all queries for the same key will eventually use the same path independently of the issuing node. XOR is symmetric, i.d., for each x, y it holds that $d(x, y) = d(y, x)$.

Nodes’ configuration For each $0 \leq i \leq 160$, each node stores a list with information (IP address, UDP port, node ID) about each node at a distance between 2^i and 2^{i+1} . These lists are called k -buckets. Each k -bucket is sorted so that at top is the node which has not appeared for the longest period and at the bottom is the node which has appeared

more recently. The lists can reach at a size of k , where k is a system parameter and suitably chosen so that it is almost impossible that any k nodes will fail during an hour. When a node receives a message, it updates the appropriate k -bucket about the sender's ID. If this node is in the list then it is moved to the bottom, otherwise if the list has fewer than k entries it also goes to the bottom. If the list is full then we check if the top node is active. If this is the case, then the new node is rejected and the top node is moved to the bottom, otherwise the new node goes to the bottom. Thus, active nodes never leave the list. This predilection for active nodes is based on the fact that the more a node is online the more likely it is that it remains online in the future [18].

The protocol The protocol contains the following 4 RPCs:

- PING: checks whether a node is online or not.
- STORE: Stores a key and its value at a node.
- FIND NODE: It has a 160-bit identifier as input. The recipient of the RPC returns the fields [IP address, UDP port, node ID] for the k nodes that it knows to be closer to its identifier.
- FIND VALUE: It performs the same as FIND NODE except when the RPC recipient has received a STORE RPC for the key; in that case it returns the key value.

In all RPCs the recipient returns a random RPC identifier in order to further guarantee the acknowledgment of the first RPC. Searching for a node is an important procedure in Kademlia since this is how the k nearest nodes are discovered for a particular node. An algorithm running in rounds is being used for this task. Initially, some nodes are chosen among those in the nearest k -bucket and asynchronous FIND are sent to these nodes. At each round, FIND NODE is sent to nodes that were discovered by previous RPCs and are not queried yet. When nodes do not respond they are removed from the search. The search is terminated when the node has received answers from the k nearest nodes it has met.

As we have already mentioned, search is an important functionality because it forms the basis of many other functionalities. When we wish to store a key and its value, a node should find the k nearest nodes to the key and send them STORE RPCs. Also, republishing of the key-value pair that allows us to obtain the most recent information in the network is based on searching. In order to find a key-value pair, a node starts searching for the k nearest to the key nodes. FIND VALUE is used and the procedure terminates when a node returns the value. If the query was successful the node stores the key and the value to the nearest to the key node that did not return the value. This enables the other nodes to know where the key-data pair is located, in case they need it in some future queries.

The node connection procedure is the following. A node u should have an edge connecting it to a node w in the network. Then, u inserts w to the appropriate k -bucket and performs a node search for its own identifier. Finally, u updates all the k -buckets that are further from the nearest neighbor, i.e., constructs its own k -buckets and is inserted in other nodes' k -buckets if necessary.

Routing tables The routing table is a binary tree where the leaves are k -buckets. Each k -bucket contains nodes with a common prefix in their IDs. This prefix also specifies the location of the bucket in the tree. The assignment of nodes to buckets is online and is performed in the following way. Initially, the tree of a node u has a single node. When u discovers a new edge it tries to insert it to the appropriate bucket. If the bucket is available then the connection is successfully completed. Otherwise, if u 's ID is contained in the bucket, then we split the bucket in two new buckets and repeat the connection procedure. If the bucket not containing u is full then the new edge is rejected. A drawback is that the trees produced are not balanced; we can solve this problem by retaining sufficient information for the smallest subtree with at least k nodes that is close to the node.

Key republishing Nodes should republish the keys from time to time in order to guarantee the validity of the key-data pairs. There are two cases where search might fail. In the first one, some of the nodes that own keys may exit the system. Kademlia republishes the keys every hour in order to deal with this case. This would normally demand each node that stores a key to search for a node and send $k - 1$ STORE RPCs, but Kademlia applies a better procedure to improve on it. As soon as a node receives a STORE RPC for a given key-value pair, it assumes that the RPC has been received also by the other $k - 1$ nearest nodes and, thus, the receiver will not republish the key in the next hour. This guarantees that in case the republishing intervals are not properly synchronized, only one node per hour will republish the key-data pair. In the second case, new nodes may enter the network with IDs that are closer to an already published key. Already existing nodes can exploit the information from their neighboring subtrees and find out which keys will be stored in the new node. So, when a node realizes that a new node has appeared, it sends STORE RPCs so that the keys are transferred to the new node. In order to reduce the number of transmitted messages, a node transmits only if its ID is the nearest to the key.

Routing For each k -bucket that covers the range $[2i, 2i + 1)$ we define i as the *bucket index*. We also define $160 - i$ to be the *depth* of a node h , where i is the smallest index of a non-empty bucket. The *height* of node y 's bucket at node x is the index of the bucket that x would insert y minus the index of the x -th least significant empty bucket. We can assume that each k -bucket of any node contains at least an edge if there is a node in the appropriate range. Under this assumption, we can show that searching requires a logarithmic number of steps. Let h be the height of the node that is nearest to the target.

If the h most significant k -bucket are full, then the search will find at each step a node whose distance is shorter by one bit, and thus the target is found in $h - \log k$ steps. If one of the k -buckets is empty then it is possible that the target node is in the range of the empty bucket. The searching will proceed as if the bit of the key that corresponds to the empty bucket is reversed. Thus, the nearest node is always returned by the algorithm in $h - \log k$ steps. After the nearest node is found, the number of steps required to find the other $k - 1$ nearest nodes cannot be greater than the height of the nearest node's bucket at the k -th nearest node, which with high probability is less than $\log n + c$, where c is a constant. Searching requires $\log_2 n$ steps, but if we consider the identifiers as b bits at a time instead of one, then the routing table increases to $2^b \log_{2^b} n$ k -buckets and the number of steps reduces to $\log_{2^b} n$. We have mentioned that a node splits a k -bucket if it is full and contains the node's identifier. However, the implementation also splits buckets that do not contain the identifier up to $b - 1$ levels. For example, if $b = 2$ then half of the identifiers' range that does not contain the node is split once (into two regions), if $b = 3$ then it is split into two levels and 4 regions. Therefore, a node splits a full bucket if its identifier is not contained and the depth d of the bucket satisfies $d \not\equiv 0 \pmod{b}$.

2.2.6 Viceroy

Introduction Up to now we have examined only deterministic protocols, in this section we present a randomized algorithm. Randomized algorithms usually run faster than deterministic ones and sometimes even have upper bounds on the running time that are lower than the lower bounds of the deterministic algorithms. This might look odd but is not unreasonable since by using randomization we obtain a stronger computability model. Another advantage of randomized algorithms is that usually it is easier to describe them. Viceroy [9] is a randomized algorithm that simulates the behavior of the butterfly network. The degree is fixed while the diameter is logarithmic, while node connection and disconnection do not require that all nodes are notified.

Viceroy is a DHT system that allows an efficient object storage and search in large networks and satisfies the first two demands of large DHT systems. Those are the uniform data distribution on active nodes and the storage of routing information among the servers so that a connection or disconnection does not demand that information is broadcasted all over the network. Another advantage of Viceroy is that the cost of such connections and disconnections is constant.

The assignment of key-value pairs to active servers is similar to that of Chord, i.e., all nodes are mapped to an identifier over a unit ring $[0 \dots 1)$, and each node handles all pairs that their identifiers are between the node and its clockwise neighbor. The concepts of successor and predecessor are also used. It also borrows ideas from Kleinberg [7] and Barriere et al. [2] and each node is connected not only to its neighbors but also to a constant number, which is 5, of distant nodes. Based on Kleinberg's model, the distant nodes are randomly chosen with further nodes having a bigger probability of being selected.

In order to improve on its behavior Viceroy follows the approach of the butterfly network, therefore it can be seen as a mix of butterfly and Chord-like ring where there exists an underlying ring topology where each node has an identifier from $[0 \dots 1)$ and is connected to its predecessor and successor as well as to 5 distant nodes in the following way. Each node randomly chooses a level so that $\log n$ levels are formed where n is the number of active nodes. Each node j at level l is connected to two nodes at level $l + 1$; the one is called *down and right* and is at distance of roughly $1/2^l$ and the other is called *down and left* and is the nearest to j at level $l + 1$. If l is greater than 1, j is also connected to its nearest node at level $l - 1$ (*upper level*). Finally, two nodes at level l are connected to j . These are j 's next greatest and next smallest node, thus forming an extra ring at each level. Therefore, each node is connected to 7 nodes.

Routing is performed in three steps; first, an up-link is being followed, then the message is being routed at a lower level following either the “down and right” or the “down and left” edge, depending on whether the node is a distance greater than $1/2^l$ or not. Finally, in the third step if a node without a down-edge is discovered we use the ring search. Routing requires $O(\log n)$ and a node connection or disconnection requires $O(\log n)$ steps and affects the configuration of $O(1)$ servers.

System description

Notation As we have already mentioned, the system is formed as an adaptive group of server that prior to their connection obtain a random identifier from $[0 \dots 1)$ using uniform distribution.

For every real numbers x, y, z we define as $\text{stretch}(x, y)$ the area between x and y with a clockwise direction without including x, y and $z \in \text{stretch}(x, y)$ when z is between x and y . We also define as density $q(x, y)$ the number of active servers in $\text{stretch}(x, y)$.

We define $\text{SUCC}(x)$ to be the first active clockwise neighbor of x and $\text{PRED}(x)$ to be the first active counter-clockwise neighbor. For each server s we denote its level by $s.\text{level}$ and we define as $\text{NLEVEL}_i(x)$ ($\text{PRELEVEL}_i(x)$ respectively) the nearest clockwise (counter-clockwise respectively) active neighbor of x that is at level i . Moreover, we define $\text{NEXTONLEVEL}(x) = \text{NLEVEL}_{x.\text{level}}(x)$ and $\text{PREVONLEVEL}(x) = \text{PRELEVEL}_{x.\text{level}}(x)$. Finally, $q_i(x, y)$ is the number of servers that are contained in $\text{stretch}(x, y)$.

We assume that a server can connect to another if it knows its identifier. We also assume that a server can connect or voluntarily disconnect but cannot be forced to disconnect; also, we assume that there are no simultaneous connections and disconnections. In practice the system can successfully handle such conditions given that the servers that need to update their configuration do not overlap.

Network configuration Each server s is specified by two numbers: an identifier $s.\text{id}$ or simply s , that remains fixed during its entire existence in the network and its level l , an integer that might change as the network evolves. We have already mentioned that

the network consists of three types of connections: a generic ring that every node connects to its successor and predecessor, a level ring where each node connects to the nearest two nodes at its level and butterfly-like connections. Using these connections, a node s that is not a leaf connects to two nodes at level $l + 1$, one called “down and right” and is the nearest clockwise to $s + 1/2^l$ (i.e., $\text{NLEVEL}_{l+1}(s + 1/2^l)$) and one called “down and left” which is the clockwise nearest to s at level $l + 1$ (i.e., $\text{NLEVEL}_{l+1}(s)$). If l is greater than one, s also connects to its nearest clockwise node at level $l - 1$ (i.e., $\text{NLEVEL}_{l-1}(s)$).

Choosing an identifier Each server s selects its identifier independently from a uniform distribution over $[0 \dots 1)$. In the same way it could also select its level from a uniform distribution over $1 \dots \log n$, where n is the number of active nodes, provided it is aware of n . However, this would demand the routing of a lot of information in the network every time the size changes, therefore an approximation is used where $n_0 = 1/d(s, \text{SUCC}(s))$ is computed and then l is uniformly distributed over $[1 \dots \log n_0]$. This way of computing n_0 is plausible since in an ideal network where all nodes are uniformly distributed the distance between two neighboring nodes is $1/n$. A server only needs to update its level when its successor changes.

We can prove that with high probability the following hold:

- $\log(n/(2 \log n)) \leq \log n_0 \leq 3 \log n_0$: each level l with $l < \log(n/(2 \log n))$ is called *sane*.
- For each server s it holds that $q(s, s + (\log n)/n) = O(\log n)$
- The expected number of steps for a server s to discover its next at a level is $O(\log n)$ and $O(\log^2 n)$ in the worst case.
- For each server s : $\min_{j \leq \log(n/(2 \log n))} q(s, \text{NLEVEL}_j(s)) = O(\log n)$
- For each server s and $i \leq \log(n/(2 \log n))$ it holds that $E[q_i(s, s + (3 \log n)/n)] = 1$ and with probability at least $1/2$ that $q_i(s, s + (3 \log n)/n) \geq 1$.
- For each server s and $i, j \leq \log(n/(2 \log n))$ it holds that $E[q_i(s, \text{NLEVEL}_i(s))] = 1$ and that $q_i(s, \text{NLEVEL}_i(s)) = O(\log n)$.

Node connection and disconnection Each server stores as local information its identifier s , the level $s.\text{level}$ and its neighbors. When a node connects or disconnects, the above data might be updated.

In the case of connections, an identifier s should be selected as was previously described and using the function LOOKUP (defined in the following) we can discover $\text{SUCC}(s)$. Then, the successor and predecessor of nodes s , $\text{SUCC}(s)$ and $\text{PRED}(s)$ should be updated. In the following, s 's successor sends to s all the key-data pairs where the keys are between s 's predecessor and s . s 's level is chosen according to the previous paragraph and $s' = \text{NEXTONLEVEL}(s)$ and $s'' = \text{PREVONLEVEL}(s)$ are discovered.

Finally, $\text{NLEVEL}_{(s.\text{level}+1)}(s)$ is discovered and mapped to $s.\text{left}$, while the nearest clockwise node s' to $(s+1/2^i)$ (where $i = s.\text{level}$) is discovered and also the next clockwise to s' at level $s.\text{level}$, i.e., $\text{NLEVEL}_{s.\text{level}+1}(s')$ and mapped to $s.\text{right}$; also $\text{NLEVEL}_{s.\text{level}-1}(s)$ is mapped to $s.\text{up}$.

When a node should update its level, it follows the last two steps of the above procedure. When a node disconnects, it notifies all servers that connect to it so that they replace it and transfers all of its key-data pairs to its successor.

Simple search We now intuitively describe a simple algorithm $\text{LOOKUP}(x, y)$ that uses only the successor, the predecessor and the butterfly edges, where the idea is that starting from server y we discover the clockwise nearest node to x . The algorithm performs in three stages. First, we find the “root”, i.e., a server of the previous level using the up-connection. Then, starting from the root we find for each level i the node nearest to x that is connected at level i . This node is always at a distance at most $1/2^{i-1}$ from the target. The second stage terminates when either a node without connections is reached or a node that is greater than x . In the last stage, starting from the last node we discover the target.

We can show that the first two stages require $O(\log n)$ with high probability while the third requires $O(\log n)$ on average and $O(\log^2 n)$ in the worst case, therefore the third stage dominates the running time. In order to measure the load on the network we define the load of a server to be the probability that it is involved in the search between two random nodes. We can show that in a network with n servers the expected value of the load for each server is $O(\log n/n)$ and the maximum load is $O(\log^2 n/n)$.

Improved search In the analysis of the third stage in the previous paragraph we obtained that there are at most $O(\log^2 n)$ servers between the current server and the target and since search visits one server at each step, it requires $O(\log^2 n)$. We can improve on it by using the level ring and reducing the time to $O(\log n)$.

Conclusions Viceroy provides a satisfactory object storage and search in large and changing networks. It borrows ideas from Chord (with respect to resource allocation on the nodes) but improves upon it since

- while every node in Chord keeps data for $\log n$ nodes to which it keeps a remote connection, Viceroy needs to keep data for only 7 nodes, thus greatly reducing the cost for node connection and disconnection.
- Also, Chord’s behavior is not concrete as the network evolves, while Viceroy behaves in a very well specified way.

2.2.7 Conclusions

The second generation of peer-to-peer systems was deployed in an attempt to overcome the limitations that were inherent in pioneer systems like FreeNet and Gnutella. The systems in the new generation guarantee the return of an answer for each query in a limited number of steps while they preserve the scalability exhibited by Gnutella and the ability to fix and reorganize the network that both FreeNet and Gnutella had.

Pastry, Tapestry, Chord, Kademia, CAN and Viceroy are members of this second generation. These systems store at each node specific information regarding the other nodes in a strictly predetermined way and route the messages in a secure and fast way, based on this information. They also possess the ability to adjust their routing paths and to reorganize their information in the case of network changes (e.g., several nodes connecting or disconnecting).

Chord was the first system to appear. It is based on DHT (Distributed Hash Table) and its main characteristic is the way it handles the object location in a specific way on which message routing and object tracking rely. This fact poses some limitations on applications both with respect to the location and the number of object copies. Also, Chord does not consider the geographical distance of the nodes that are part of a routing path (locality) and, hence, cannot guarantee that the shortest routing path is being followed. The number of routing steps is $O(\log N)$ where N is the number of the nodes.

Tapestry relies on DOLR (Decentralized Object Location and Routing) and transfers messages to the nodes that have a prefix as near as possible to the message key prefix. It can guarantee that locality is being used so that the messages are sent without delay and without congesting the network. It does not pose constraints on the location and the number of objects that the applications may handle. On the negative side, the restoration of the information being stored at each node in the case of rapidly changing networks is rather difficult. The number of routing steps is $O(\log_{\beta} N)$ where β is the basis of the identifiers.

Pastry uses a routing mechanism that is similar to that of Tapestry (using the prefix in order to determine the next node) but offers a more flexible structure of the information tables stored at each node. This leads to a better response in online events and suggests that Pastry is a suitable system for rapidly changing networks. It also guarantees that locality is used and, thus, routing paths are only slightly longer than the actual distance of two nodes. The number of routing steps is $O(\log_{2^{\beta}} N)$, where β is a network configuration parameter; a typical value is 4. Contrary to CAN where the number of routing steps is $O(dN^{1/d})$, in Pastry the number of steps increases much slower as N increases.

Kademia differs from the rest of these systems because of its XOR metric. XOR is symmetric, thus enabling the participants to receive queries from exactly the same node distribution that is also at their routing tables. Chord does not possess this feature and cannot gather routing information from the queries. Each entry in the index table at a node in Chord stores the exact node of some range in the identifiers' space. Each node

from this range can be away from the node. On the contrary, Kademlia can send a query to each node within a range.

Viceroy was designed as an improvement on Chord. Thus, it follows the directed ring structure, the resource allocation mechanism and the basic routing mechanism (i.e., the concept of successor and predecessor). In order to accelerate routing each node is connected not only to its neighbors on the ring but also to 5 distant nodes in a butterfly-like way. Thus, Viceroy is a mix of butterfly and Chord-like ring. Thus, while in Chord each node stores information for $\log n$ nodes, in Viceroy each node has to store information only for 7 nodes, thus reducing the node connection/disconnection cost, while at the same time using the same number of routing steps as Chord, i.e., $O(\log N)$.

3 Distributed Computing

Apart from peer-to-peer systems, distributed computing is a very natural alternative. The main distinction is that load balancing and accumulating computing power are now the main targets and not topology control and adaptability. Many projects, from searching for prime numbers to analyzing signs of extraterrestrial activity, have been deployed where the target was to efficiently perform intensive computations over large data sets. In the following, we describe two prominent examples.

3.1 Grid

Introduction Grid [6] computing is a rather new computing model that provides the ability to perform higher throughput computing by using the accumulated computing power of many networked computers. The main idea is to establish a virtual computer architecture that will be able to distribute workload across an underlying parallel infrastructure.

Grid exploits the resources of many individual computers connected by an existing network (in most cases the Internet) to solve large-scale computational problems and provides the ability to perform computations on large data sets, by breaking them down into many smaller ones. It also enables the user to perform many more computations at once than would be possible on a single computer, by modeling a parallel division of labor between processes. Today resource allocation in a grid is done in accordance with SLAs (service level agreements).

Grid computing reflects a conceptual framework rather than a physical resource. The Grid approach is utilized to provision a computational task with administratively-distant resources. The focus of Grid technology is associated with the issues and requirements of flexible computational provisioning beyond the local administrative domain.

A Grid environment is created to address resource needs. The use of that resources, which could range from CPU cycles, disk storage and data to software programs and peripherals, is usually characterized by its availability outside of the context of the local

administrative domain. This “external provisioning” approach entails creating a new administrative domain referred to as a *virtual organization* with a distinct and separate set of administrative policies (home administration policies plus external resource administrative policies equals the virtual organization administrative policies). The context for a Grid “job execution” is distinguished by the requirements created when operating outside of the home administrative context. Grid technology, also called middleware, is employed to facilitate formalizing and complying with the Grid context associated with your application execution.

One characteristic that currently distinguishes Grid computing from pure distributed computing is the abstraction of a “distributed resource” into a Grid resource. One result of abstraction is that it allows resource substitution to be more easily accomplished. Some of the overhead associated with this flexibility is reflected in the middleware layer and the temporal latency associated with the access of a Grid (or any distributed) resource. This overhead, especially the temporal latency, must be evaluated in terms of the impact on computational performance when a Grid resource is employed.

Grid architecture The architecture of the Grid is often described in terms of “layers”, each providing a specific function. In general, the higher layers are focussed on the user, whereas the lower layers are more focussed on computers and networks.

At the base of everything, the bottom layer is the network, which assures the connectivity for the resources in the Grid. On top of it lies the resource layer, made up of the actual resources that are part of the Grid, such as computers, storage systems, electronic data catalogues, and even sensors such as telescopes or other instruments, which can be connected directly to the network. The middleware layer provides the tools that enable the various elements (servers, storage, networks, etc.) to participate in a unified Grid environment. The middleware layer can be thought of as the intelligence that brings the various elements together. The highest layer of the structure is the application layer, which includes all different user applications (science, engineering, business, financial), portals and development toolkits supporting the applications. This is the layer that users of the grid will “see”. In most common Grid architectures, the application layer also provides the so-called serviceware, the sort of general management functions such as measuring the amount a particular user employs the Grid, billing for this use (assuming a commercial model), and generally keeping accounts of who is providing resources and who is using them - an important activity when sharing the resources of a variety of institutions amongst large numbers of different users. The serviceware is in the top layer, because it is something the user actually interacts with, whereas the middleware is a “hidden” layer that the user should not have to worry about.

There are other ways to describe this layered structure. For example, experts like to use the term fabric for all the physical infrastructure of the Grid, including computers and the communication network. Within the middleware layer, distinctions can be made between a layer of resource and connectivity protocols, and a higher layer of collective

services.

Resource and connectivity protocols handle all Grid specific network transactions between different computers and other resources on the Grid. Remember that the network used by the Grid is the Internet, the same network used by the Web and by many other services such as e-mail. Many transactions are going on at any instance on the Internet, and computers that are actively contributing to the Grid have to be able to recognize those messages that are relevant to them, and filter out the rest. This is done with communication protocols, which let the resources speak to each other, enabling exchange of data, and authentication protocols, which provide secure mechanisms for verifying the identity of both users and resources.

The collective services are also based on protocols: information protocols, which obtain information about the structure and state of the resources on the Grid, and management protocols which negotiate access to resources in a uniform way. The services include:

- keeping directories of available resources updated at all times,
- brokering resources (which like stock broking, is about negotiating between those who want to “buy” resources and those who want to “sell”)
- monitoring and diagnosing problems on the Grid,
- replicating key data so that multiple copies are available at different locations for ease of use
- providing membership/policy services for keeping track on the Grid of who is allowed to do what, when.

In all schemes, the topmost layer is the applications layer. Applications rely on all the other layers below them in order to run on the Grid. To take a fairly concrete example, consider a user application that needs to analyze data contained in several independent files. It will have to:

- obtain the necessary authentication credentials to open the files (resource and connectivity protocols),
- query an information system and replica catalogue to determine where copies of the files in question can currently be found on the Grid, as well as where computational resources to do the data analysis are most conveniently located (collective services),
- submit requests to the fabric - the appropriate computers, storage systems, and networks - to extract the data, initiate computations, and provide the results (resource and connectivity protocols),
- monitor the progress of the various computations and data transfers, notifying the user when the analysis is complete, and detecting and responding to failure conditions (collective services).

In order to do all of the above, an application that a user may have written to run on a stand-alone PC will have to be adapted in order to invoke all the right services and use all the right protocols. Just like when trying to import an application on the Web, where a stand-alone application should be adapted in order to run on a Web browser, so too the Grid requires users to invest some effort into modifying their applications to render them suitable for the Grid. Once this is performed, users are able to use the same application and run it on the Grid with minimal effort, using the middleware layers to adapt in a seamless way to the changing circumstances of the fabric.

Roles machines play Typically, machines play different roles in a grid project. For example, the DataGrid testbed [4] consists of approximately 1000 CPUs at 15 sites across Europe, and is built on top of the EU-funded GEANT high-speed communications network. The machines linked on this testbed play one (or more, if possible) of the following different roles:

- Resource Broker, the module that receives users' requests and queries the Information Index to find suitable resources.
- Information Index, which can reside on the same machine as the Resource Broker, keeps information about the available resources.
- Replica Manager, used to coordinate file replication across the testbed from one Storage Element to another. This is useful for data redundancy but also to move data closer to the machines which will perform computation.
- Replica Catalog, which can reside on the same machine as the Replica Manager, keeps information about file replicas. A logical file can be associated to one or more physical files which are replicas of the same data. Thus a logical file name can refer to one or more physical file names.
- Computing Element, the module that receives job requests and delivers them to the Worker Nodes, which will perform the real work. The Computing Element provides an interface to the local batch queuing systems. A Computing Element can manage one or more Worker Nodes. A Worker Node can also be installed on the same machine as the Computing Element.
- Worker Node, the machine that will process input data.
- Storage Element, the machine that provides storage space to the testbed. It provides a uniform interface to different Storage Systems.
- User Interface, the machine that allows users to access the testbed.

3.2 PlanetLab

Introduction PlanetLab [12] is a globally distributed computing platform that, in a similar way to the Internet emerging as an overlay to the telephone system, aims to provide industry and academic researchers a robust and realistic platform with a large overlay network that can tap into today's Internet without disrupting it. PlanetLab currently consists of 694 machines, hosted by 335 sites, spanning over 25 countries. Most of the machines are hosted by research institutions, although some are located in co-location and routing centers. All of the machines are connected to the Internet.

One of PlanetLab's main purposes is to serve as a testbed for overlay networks. Research groups are able to request a PlanetLab *slice* (which is a collection of virtual machines spread around the world) in which they can experiment with a variety of planetary-scale servi network-embedded storage, content distribution networks, routing and multicast overlays, QoS overlays, scalable object location, scalable event propagation, anomaly detection mechanisms, and network measurement tools. There are currently over 275 active research projects running on PlanetLab.

Elements and principals We briefly discuss about the main architectural elements of PlanetLab:

- Node: it is any machine capable of hosting one or even more virtual machines. In the current PlanetLab architecture, there is one-to-one mapping between nodes and physical machines.
- Virtual machine: it is an execution environment in which a slice runs on a particular node. Several types of virtual machine are possible and a given node might support more than one.
- Node manager (NM): it is a program running on each node that creates virtual machines on the node and handles the resources allocated to these virtual machines. There is a one-to-one mapping between nodes and node managers.
- Slice: it is a set of virtual machines; each element of this set runs on a unique node.

There are four main principals involved, *owners*, *service providers*, *management authorities* and *slice authorities*; we briefly describe each of them.

- Owner: it hosts one or more nodes, over which it retains full control, but chooses one management authority to manage its nodes. It also accepts slices on behalf of a slice authority.
- Service provider: it implements and deploys network services on a set of nodes, and also is responsible for those services.

- Management authority: it operates a set of nodes on behalf of their owners. It also installs the required software that runs on these nodes. It is possible for the management authority to actually own a fraction of the nodes it manages.
- Slice authority: it registers a set of service providers, creates slices and binds providers to slices. It should also be able to map a slice to its corresponding providers.

Conclusions The advantage to researchers in using PlanetLab is that they are able to experiment with new services under real-world conditions, and at large scale. The example services outlined above all benefit from being widely distributed over the Internet: from having multiple vantage points from which applications can observe and react to the network’s behavior, from being in close proximity to many data sources and data sinks, and from being distributed across multiple administrative boundaries. PlanetLab also serves as a meta testbed on which multiple, more narrowly-defined virtual testbeds can be deployed. That is, if we generalize the notion of a service to include what might traditionally be thought of as a testbed, then multiple virtual testbeds can be deployed on PlanetLab. For example, we are developing an “Internet-in-a-Slice” service that recreates the Internet’s data plane (IP forwarding engine) and control plane (routing protocols like BGP and OSPF) in a slice. Network researchers can use this infrastructure to experiment with modifications and extensions to the Internet’s protocol suite.

On the negative side, the topology in PlanetLab is rather fixed; an institution should apply in order to enter the consortium and specific actions should take place before it can exploit PlanetLab.

References

- [1] S. Androutsellis–Theotokis and D. Spinellis. A survey on peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335-371, 2004.
- [2] L. Barriere, P. Fraigniaud, E. Kranakis and D. Krizanc. Efficient routing in networks with long range contacts. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*, pp. 270-284, 2001.
- [3] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*, LNCS 2009, Springer, pp. 46-66, 2001.
- [4] The DataGrid Project. Available at <http://eu-datagrid.web.cern.ch/eu-datagrid/>
- [5] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the ACM SIGCOMM '99 Conference on Applications*,

- Technologies, Architectures, and Protocols for Computer Communication*, pp. 251-262, 1999.
- [6] I. Foster, and C. Kesselman. The grid: Blueprint for a new computing Infrastructure. *Morgan Kaufmann*. 2004.
 - [7] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the 32nd ACM Symposium on Theory of Computing (STOC '00)*, pp. 163-170, 2000.
 - [8] J. Liang, R. Kumar and K. W. Ross. The FastTrack overlay: A measurement study. *Computer Networks*, 50(6): 842-858, 2006.
 - [9] D. Malkhi, M. Naor and D. Datajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing (PODC '02)*, pp. 183-192, 2002.
 - [10] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pp. 53-65, 2002.
 - [11] Overnet. Available at <http://www.overnet.org>
 - [12] PlanetLab. Available at <http://www.planet-lab.org/>
 - [13] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips. The Bittorrent p2p file-sharing system: Measurements and analysis. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS '05)*, pp. 205-216, 2005.
 - [14] D. Qiu and R. Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '04)*, pp. 267-278, 2004.
 - [15] S. Ratnasamy, P. Francis, M. Handley and R. Karp. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communicationof (SIGCOMM '01)*, pp. 161-172, 2001.
 - [16] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings of the 1st International Conference on Peer-to-Peer Computing (P2P '01)*, 2001.
 - [17] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware '01)*, pp. 329-350, 2001.

- [18] S. Saroiu, P. K. Gummadi and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking Conference 2002 (MMCN'02)*, 2002.
- [19] S. Saroiu, P. K. Gummadi and S. Gribble. Measuring and analyzing the characteristics of Napster and Gnutella hosts. *Multimedia Systems*, 9(2):170-184, 2003.
- [20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)*, pp. 149-160, 2001.
- [21] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1): 41-53, 2004.