Evaluation of A Scalable Peer-to-Peer Lookup Protocol for Internet Applications

SAMER AL-KASSIMI

Master of Science Thesis
Stockholm, Sweden 2005

IMIT/LECS-2005-95

Evaluation of A Scalable Peer-to-Peer Lookup Protocol for Internet Applications

SAMER AL-KASSIMI

Thesis Advisors:
Per Brand and Sameh El-Ansary
Thesis Examiner:
Vladimir Vlassov

Work performed at the Swedish Institute of Computer Science (SICS) and submitted to the Royal Institute of Technology (KTH) in partial fulfillment of the requirements for the degree of Master of Science

Master of Science Thesis
Stockholm, Sweden 2005

IMIT/LECS-2005-95

Acknowledgements

This thesis has been the result of a very, very long and hard work, and not only in terms of scientific and technical labor.
It certainly would not have been possible without the help of many people.
The beginnings are usually difficult, but in this case they were very close to dramatic, and those who know me well understand that this is not just my share of Andalusian blood talking exaggeration.
I am a person who takes pride at being thankful every day for what I am given.
And I have been given a lot during my stay here in Stockholm.
First of all, I have to thank Mats Brorsson at IMIT in KTH for the welcome he gave me nearly 4 years and a half ago, and incidentally, Xavier Martorell from the Computer Architecture Department at UPC, who introduced me to him.
During some of the courses in KTH I met Vladimir Vlassov and Luc Onana.
I am very thankful for all the support that Vlad has given as a new examiner in such a short time and with such a short notice, in a moment when I was nearly desperate thinking that I would suffer another delay yet. It has been a blow of fresh air just when the winter is threatening to fall upon all of us.
Luc Onana is the person that, after a very enjoyable course on Distributed Systems, introduced me to the staff at the Swedish Institute of Computer Science (SICS), and that meant a lot to me. I have the impression that he had very high expectations that I could not fulfill due to many adverse circumstances. For that I am sorry; but most of all, I am grateful of the role he played in all this story.
There at SICS I met the most amazing group I have ever had the pleasure to work with. Beginning with Seif Haridi, all the way with Per Brand and Sameh El-Ansary (my supervisors) and other people that helped me much amongst whom I have to remark Ali Ghodsi for being a great room partner. Being at SICS has been a formidable experience and my only regret is not having been able to extract the best of it. Thank you all for having been there and all your help, especially Per, who has been extremely supportive at all times.
And speaking of support, special mention to my "familia postiza" på Lappis and surroundings. I'm trying to write a few names, and many are going to be lost, there's no place for everybody, and I know this is no excuse, but you all know how bad my memory is...: Cristina Sáez, Rafa Cordones, Carmen Cárdenas, Guillermo Torres, Raul Iglesias, Xavi Gelabert, Rodrigo Sierra, Manolo Mazo, Luis, Estitxu, Maria Selva, Mariansita, David, María José Mesa, Sanna, Rossana Giaconi, Natasha Mouravitskaya, Salva, Jabón (aarrr!), Alicia Soutullo, Fran Marquez, Paco, Juancar, Bego, Jose, Rosa, Elena, Beni, Xavi Gratal, Veera, Bea10, Jaume y Mercedes, Mariquilla, Toñi, Núria, Lisi, Bet, Jordi, Dafne, Andreu Taberner, Kabra, Joan Lusilla, Dario Betrián, Neus, Patricia, Enrico, Patrik, Oscar Sierra... Oh, my! You're so many... But mostly "el núcleo duro": Oscarín, Victor, Sandra, Beatxu and Merche, you are the best.
And I place aside two persons that have meant a lot to me.
Emilio Melero, Kalifa, you know how special this experience has been, and how much we've shared and learnt one from each other. May our paths give us many more chances to learn together and share.
And Pere Oriol, you really can teach so many people how much can one say with little words. Everything is inside. Ets molt més gran per dins que per fora, i això ja és dir molt.
Both of you have made me feel at home and Sweden has a particular homey flavor thanks to you.
And homey home gives me the cue to acknowledge my friends from Spain.

From my home school at UPC I have to mind Susana Ubach, Jordi Camps, Jordi Sola, Gema Gomez, Juan Francisco Fernandez, Jordi Varela, Oriol Mercadé, Roberto López and Maria del Mar Colillas
Those friends who are always there, no matter how bad things are.
Gabriel Lozano, Gabito, crack, eres el mejor. You know you can spell v-a-l-o-n-i-a from the corner of my mouth.
Sara Lanau, tú también has pasado por esto y sabes lo que vale. I'll never get why you didn't want to be Erasmus!
Helena Grau, has sigut una de les persones que més m’ha recolzat amb el teu possitivisme. Moltíssimes gràcies pel teu suport.
Now, getting closer to my family, my Dad and Mum. Papa, gracias por ser tan fuerte. Cuántas veces me he resistido a quejarme al acordarme de tí. Mama, gracias por ser más fuerte todavía. Me habéis dado todo lo que tengo, y soy quien soy gracias a vosotros. My brothers, so similar to me, and yet so very different. The farther away we've lived one from each other, the closer I've felt you. Tamer, sometimes I think you know more about computers than I do (don't be too happy, that doesn't mean much anyway). Amer, you don't know what it feels like when you look down to see your little brother and you realize you have to be looking up. I have so much to learn from you. I'm very proud of both of you.
Grandma Enriqueta, wherever you are, you can see me, you are in our memories. Grandpa Antonio, this achievement has a higher meaning to me because of what it means to you.
To my uncles Paco and Teófilo, aunts Antonia and Salvadora, and all my cousins Ana Mari, Francis, Esther, Arantxa and Sara, os quiero un montón, pensar en vosotros me ha dado fuerzas en los peores momentos. Especially my cousin Montse. Who would have said that after wrecking havoc at your place in my childhood I would adore you the way I do today?
I want to dedicate a line here to all my relatives in Syria, the most of whom I haven't seen in more than 15 years, with special mention to my uncle Osman.
And last, but not least, to the person that snapped her fingers and made it all happen at once. The person that came in my aid when I was worst with myself. The person that made me see the light at the end of the tunnel, where I could see all the sense behind past, present and future. The one that makes me shiver with one look and sends me to unknown places with one touch. The greatest of all these acknowledgements and my deepest gratitude to my fiancée, my friend, my partner, my lover, María José Vicente. I've been through many up and downs all during the realization of this thesis, but there is a definite point of inflection at your arrival. This is the first of many great presents to come from me.
In short, thank you very much, your support means much to me. This accomplishment has a little bit of you in it.

A mi abuelo Antonio,

Abstract

Peer to peer (P2P) systems are, among other models of distributed systems, one of the most fashionable nowadays. Scalability, full decentralization, anonymity, use of the computational power at the edges of the network, mobility and availability of services are, with many other, very desirable properties of such systems.
This master thesis work presents the results of research about Chord. Chord is a project lead by a team from the University of Berkeley and the Massachusetts Institute of Technology that aims at providing location of resources in a network by means of a protocol that addresses some of the features stated above.
The contents of this research include the study of one of the publications by Ion Stoica et al., as a base to further work with Chord. As a complement to this groundwork, a set of software tools has been developed to gather data —through a comprehensive set of simulations— which provides a means for a further, deeper study of Chord's behavior. The aforementioned simulations reproduce certain typical circumstances in order to permit the collection of representative and relevant figures for the subject at hand, that is, measure how the protocol —as implemented here— copes with these particular situations and conditions.
Keywords: *Chord, computer networks, survey, structured, consistent hashing, decentralization, DHT, distributed systems, fault tolerance, Peer-to-peer (P2P), scalability, resiliency, robustness, simulator, traffic generator.*

Table of Contents:

Table of classes in Javadoc

Table of Figures

1 Preliminaries

1.1 Introduction

Peer-to-peer systems' main distinctive feature is the lack of a centralized control or hierarchical organization. Some other desirable properties are redundant storage, permanence, load balance, selection of nearby servers, anonymity, search, authentication or hierarchical, flexible naming. And yet, the main problem to address is the location of items.
This master thesis presents a study of Chord based on a paper published by Ion Stoica and other authors called "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications" [STO-1].
Chord's main goal is the location of entities in P2P environments, namely: documents, files, or generally speaking, any resource that one might want to share in a computer network. It is a distributed lookup protocol that provides such location of entities with some of those very desirable properties. It is done by means of a single operation that maps a given key onto a node. Data location can thus be easily implemented on top of Chord by associating a key with each resource item. Chord shows adaptation advantages when node failures occur and when nodes continuously join and leave the network. Another very desirable feature along with the adaptation of the network is efficient query replies in presence of these events.
Chord might not be the ideal solution for most of the applications in which peer-to-peer technology is used today. However, some of its ideas could be very well put into practice in more general-purpose tools to make systems more efficient.
Chord has a number of advantages when faced to other P2P systems in terms of scalability, performance and simplicity:
While Freenet [FRE-2],[FRE-3] is decentralized and symmetric, and automatically adapts when hosts leave and join, and it does not assign responsibility for shared resources to specific nodes, it does not guarantee retrieval of existing resources or provide low bounds on retrieval costs. Its lookups take the form of searches for cached copies. This allows Freenet to provide a degree of anonymity unlike Chord. But Chord's lookup operation runs in predictable time and always results in success or definitive failure as opposed to Freenet's.
OceanStore [OCS-4], based in work by Plaxton et al. [PLA-5] is perhaps the closest algorithm to the Chord protocol in terms of reliability. It provides stronger guarantees than Chord: queries make a logarithmic number of hops and keys are well balanced; furthermore queries never travel further in network distance than the node where the key is stored, subject to assumptions about network topology. The advantage of Chord is that it is substantially less complicated and handles concurrent node joins and failures well.
Unlike Napster [NAP-6], Chord avoids single points of failure or control; and when compared to Gnutella's widespread use of broadcasts [GNU-7], Chord sports better scalability.

1.2 Related Work

This section presents a survey of data related to Peer-to-Peer systems, beginning with some definitions admitted from scholars and experts in the field of distributed systems. A subsection of how these systems have evolved will follow with also a separate section for a P2P systems taxonomy study. And finally, a summary of trends and research issues closes the section

Much of the data exposed here has been extracted from Sameh El-Ansary's Licentiate Philosophy Dissertation [SEA-8], and completed with information and quotations from other sources and surveys, but mainly from the workgroup of the Distributed Systems Laboratory in SICS that has hosted me under the duration of my thesis work.

1.2.1 Definition

Because Peer-to-Peer systems are relatively young and still evolving, a precise definition is hard to establish. Depending on the sources, the focus at aim and the moment, these definitions have suffered additions or suppressions. At times it has been intended to find a general enough definition, and this ends up categorizing systems that do not purely apply to the idea of a Peer-to-Peer system.

What is common to most definitions is the idea that such systems have resource sharing at aim, they must have certain degree of autonomy and decentralization, the fact that dynamic IP addresses are usually involved, and last but not least, the client-and-server dual role of participants, e.g.:

Oram: P2P is a class of applications that takes advantage of resources – storage, cycles, content, human presence – available at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, P2P nodes must operate outside the DNS system and have significant or total autonomy from central servers. [ORA-9] [ORA-10].

Miller: P2P is a network architecture in which each computer has equivalent capability and responsibility. This is in contrast to the traditional client/server network architecture, in which one or more computers are dedicated to serving the others. However, we need more complex definition: P2P has five key characteristics. (i) The network facilitates real-time transmission of data or messages between the peers. (ii) Peers can function as both client and server. (iii) The primary content of the network is provided by the peers. (iv) The network gives control and autonomy to the peers. (v) The network accommodates peers that are not always connected and that might not have permanent Internet Protocol (IP) addresses. [MIL-11].

P2P Working Group: P2P computing is the sharing of computer resources and services by direct exchange between systems. These resources and services include the exchange of information, processing cycles, cache storage, and disk storage for files. Peer-to-peer computing takes advantage of existing desktop computing power and networking connectivity, allowing economical clients to leverage their collective power to benefit the entire enterprise. [PTP-12]

1.2.2 Evolution

Peer-to-Peer systems have evolved during the last 6 years or so, since the introduction of Napster. This has been a hot topic in research since then, and all through the years the systems have faced changes that depended mostly on two features: decentralization, guarantee of success and scalability.

1.2.2.1 The Beginning:

Napster offered its users a way to share files with the use of a centralized directory service, while the storage was decentralized. This centralization brought two difficulties. First, politically (and legally) it was a problem that most of the material shared in the network had copyright. The directory server was storing and issuing information that ultimately led to what were considered illegal downloads. And second, the technical problems: to start with, the directory server is a single point of failure; moreover, the system is also difficult to scale, given that the load in the directory server increases with linear cost relative to the number of participants in the network.

This central server was the aim of the people who had in mind improving P2P systems. Gnutella [GNU-7] and Freenet [FRE-2],[FRE-3] came up with ideas involving flooding systems in networks where one participant only needed to know about another one peer to start proceedings and gain knowledge of other participants in the network. Similarly, a participant performs a flooding algorithm by asking all of his neighbors about a given query. His neighbors act similarly and the process is stopped by a query embedded Time-To-Live value that prevents further forwarding of queries, and thus, an ultimate collapse of the network due to increasing traffic. With this idea the centralization problem was overcome, but it still remained the issue of scalability which was, if anything, even worse. Some studies [MAR-13],[RIP-14] showed that high network traffic induced by such flooding mechanisms imposed serious restrictions to the growth of such systems. Furthermore, adding a Time-To-Live upper boundary to the path length of a query raised in Gnutella a new reliability problem: that a resource available in the network might not be retrievable by certain peers due to the “distance” between the requester and the node storing the item, or so to call it, the limitation in the scope of search. Freenet follows a slightly better approach which is the document routing model through which a data item *d* is inserted in a node with an identifier that is most similar to the identifier of *d*. During search, a query is forwarded guided by the identifier of the data item. Due to the random nature of the Freenet network, guarantees on finding items are low. An optimization to the flooding/gossiping approach was the introduction of the notion of super-peers that was initially adopted in the Kazaa [KAZ-15] system and later in the Gnutella system as well. The optimization allows for some nodes to act as directory services and thus reduces the amount of flooding needed to locate data.

Scalability was obviously becoming a hot issue in such systems, and the next step in the evolution was to provide a means to conquer that milestone for applications whose popularity was alarmingly increasing.

1.2.2.2 Structure

With that ambition in mind a new idea crawled into the researchers minds: to impose a logical structure to the network topology laying within. And thus the structured Peer-to-Peer systems were born. Their major representatives were Chord [STO-1],[STO-44], CAN [CAN-16], Pastry [PAS-17] and Tapestry [TAP-18]. More have come later, but this thesis aims at focusing on Chord as a good paradigm of these systems.

The technology on top of which these projects are based is known as a Distributed Hash Tables (DHT). A node (Peer) in such systems acquires an identifier based on a cryptographic hash of some unique attribute such as its IP address. A key for a data item is also obtained through hashing. The hash table actually stores data items as values indexed by their corresponding keys. That is, node identifiers and key-value pairs are both hashed to one identifier space. The nodes are then

connected to each other in a certain predefined topology, e.g. a circular space in Chord, a d-dimensional Cartesian space in CAN and a mesh in Tapestry and key-value pairs are stored at nodes according to the given structure. Thanks to the structured topology, data lookup becomes a routing process with low (typically logarithmic) routing table size and maximum path length. Unlike the previously mentioned systems, DHTs provide high data location guarantees because no restriction on the scope of search is imposed. .
Given the desirable properties of scalability and high guarantees while meeting the requirements of full decentralization, DHTs are currently considered in research communities as the most reasonable approach to routing and location in P2P systems. While having a common principle, each system has some relative advantages. e.g., The Chord system has the property of simple design. Tapestry and Pastry address the issue of proximity routing. The most attractive property in all current DHT systems is self-organization. Due to the focus on the absence of central authority, DHTs provide mechanisms by which the structural properties of the network are maintained while the peers are continuously joining and leaving it.
Nevertheless, not only self-organization is at stake. Other "self-" properties [ALI-19] play an important role in the fight for achievement of systems that converge to stability [DIJ-20],[LAS-21] despite of high churn [STU-22] rates.
Periodic stabilization is the system used by Chord, CAN and Pastry. It involves a number of routines being executed in a periodic fashion to correct the routing information that each node maintain.
Adaptive stabilization, also called "self-tuning" in [MAH-23] claims that periodic stabilization consumes too much bandwidth unnecessarily, and is based in the idea that the observation of the behaviour of the system can yield information about how best to tune the amount of information delivered from one node to another to keep routing information up to date with low cost. However, it is not yet clear what parameters are to be observed to effectively tune the probing rate. More importantly, how to make these observations is currently not well understood, given the large scale nature and the high dynamism of the targeted systems. Anyhow, the research on adaptive stabilization show the importance of building systems that self-adapt to observed and current behaviors. Correction-on-use combined with correction-on-change presented in the following paragraphs provide this self-adaption at a low cost.
Correction-on-use is another proposal to overcome the high bandwidth consumption employed by periodic stabilization suggested in [ALI-24]. The technique is basically that the traffic within the network carries information for the nodes to be stored and learn about the topology and status of the network. Its main drawback is that only under certain assumptions of high enough traffic the system is good enough by itself.
Correction-on-change complements correction-on-use by proposing that each time a node joins, leaves or drops from the network some new routing information has to be injected into a number of nodes that will propagate the information according to needs.
The combination of correction-on-change and correction-on-use does not have the high cost of bandwidth that periodic stabilization shows. If there are no changes in the network, no extra traffic is added. Furthermore, the use of this combination adds an extra robustness to the systems that use it that comes from the fact that when a node joins or fails other nodes are pro-actively notified.

1.2.2.3 Conclusion:

Does all this mean that a “battle” between structured and unstructured systems is at stake? From what has been exposed, and thinking in pure terms of evolution, it might seem that DHT systems are superior to the unstructured previously mentioned. The truth is, as in many technologies before, solutions tend to advance into the hybrid compromise. This is the main rebate to the classic criticism that has been placed upon structured systems regarding high churn rates.

However, the second main criticism of structured systems is that they do not support keyword searches and complex queries as well as unstructured systems. Given the current file-sharing deployments, keyword searches seem more important than exact-match key searches in the short term.

Some have justifiably seen unstructured and structured proposals as complementary, not competing. One proposal is Structella [CAS-25], a hybrid of Gnutella and Pastry. Their starting point was the observation that unstructured flooding or random walks are inefficient for data that is not highly replicated across the P2P network. Structured graphs can find keys efficiently, irrespective of replication.

Furthermore, unstructured proposals have evolved and incorporated structure. Consider the classic unstructured system, Gnutella. For scalability, its peers are either ultrapeers or leaf nodes. This hierarchy is augmented with a query routing protocol whereby ultrapeers receive a hashed summary of the resource names available at leaf-nodes. Between ultrapeers, simple query broadcast is still used, though methods to reduce the query load here have been considered. Secondly, there are emerging schema based P2P designs, with super-node hierarchies and structure within documents. These are quite distinct from the structured DHT proposals.

1.2.3 Taxonomy

From what has been mentioned in the previous subsection, and basically considering two variables (decentralization and topology), as done in [AND-26] and [LVQ-27], the following taxonomy (with examples) in Fig. 1.1 IS considered as suitable for Peer-to-Peer systems.

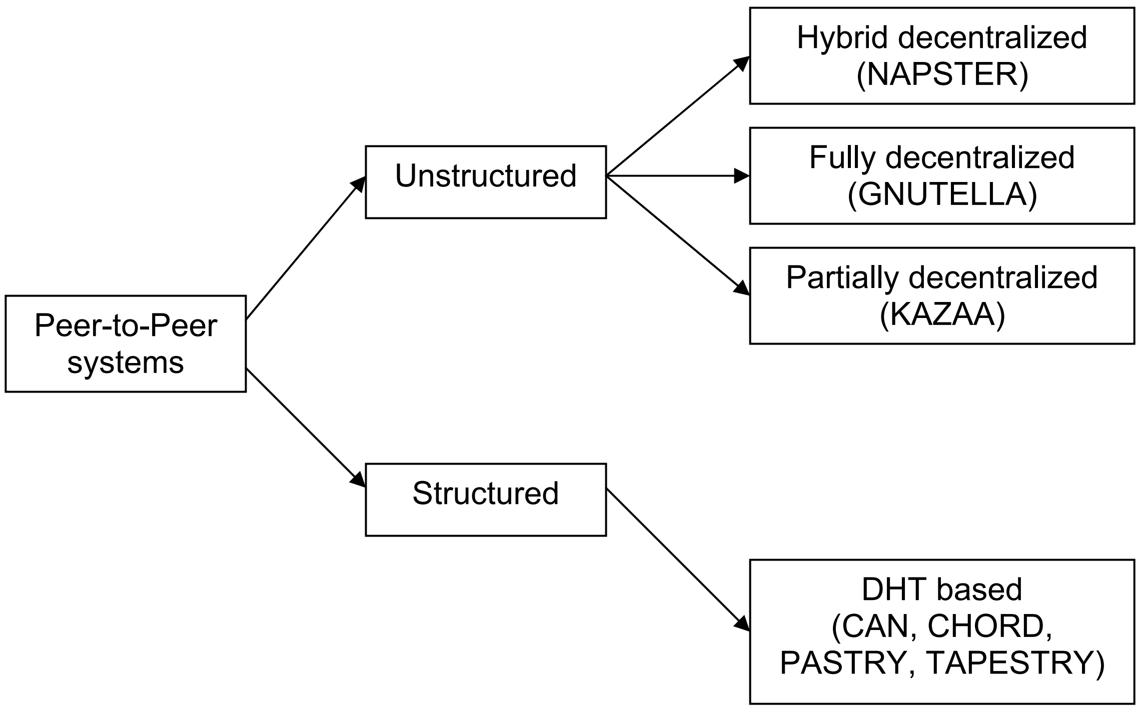

Peer-to-Peer systems
Unstructured
Structured
Hybrid decentralized (NAPSTER)
Fully decentralized (GNUTELLA)
Partially decentralized (KAZAA)
DHT based (CAN, CHORD, PASTRY, TAPESTRY)

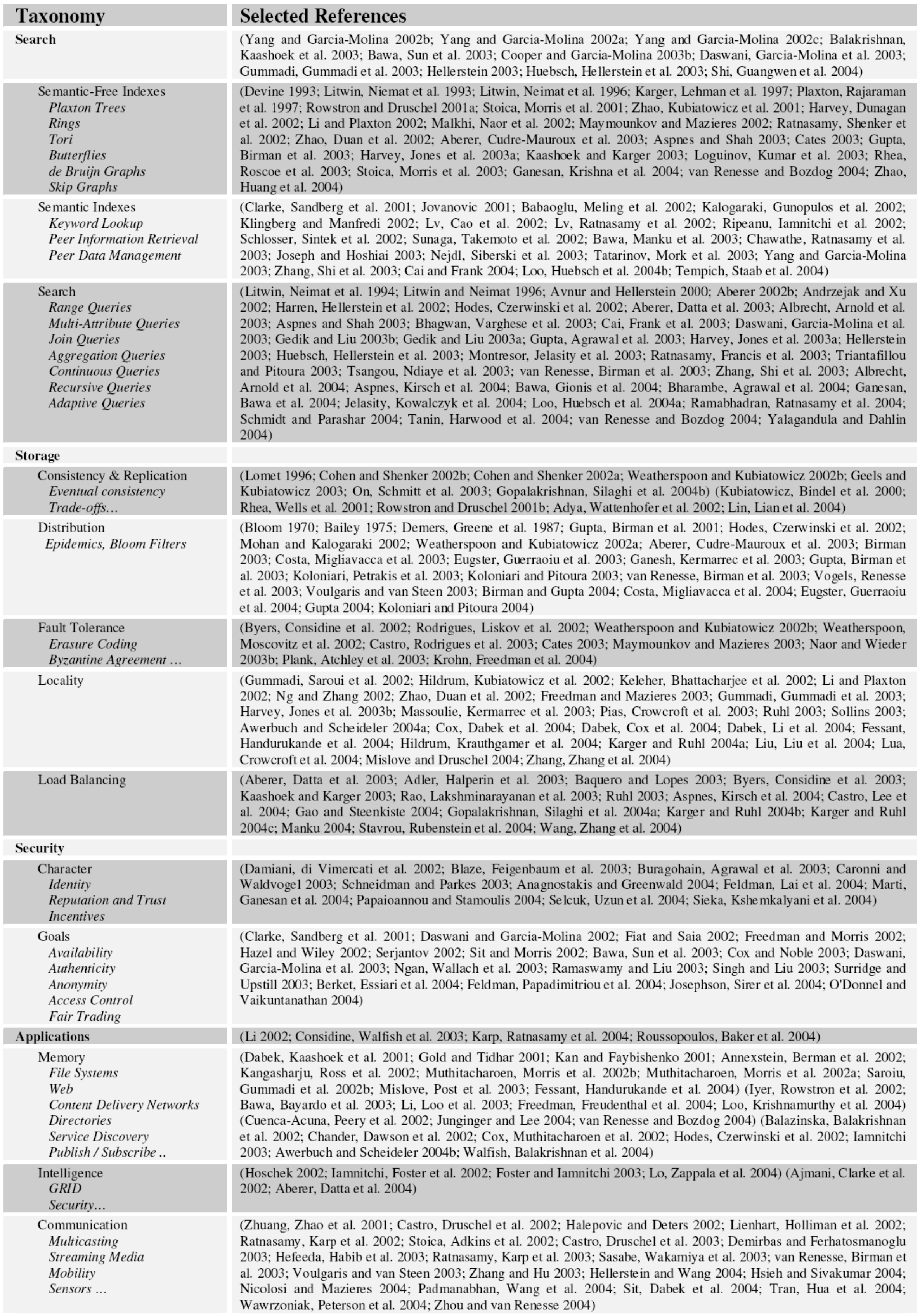
Fig. 1.1 Taxonomy of Peer-to-Peer systems

This taxonomy captures major differences between P2P systems, and is widely accepted by the community

The network structure characteristic aims at looking at systems from the topological perspective. Two levels of structuring are identified: unstructured and structured. In an unstructured topology, an overlay network is realized with a random connectivity graph. In a structured topology, the overlay network has a certain predetermined structure such as a ring or a mesh.

The degree of centralization means to what extent the set of peers depend on one or more servers to facilitate the interaction between them. Three degrees are identified: *Fully decentralized*, *Partially decentralized* and *Hybrid decentralized*. In the fully decentralized case, all peers are of equal functionality and none of them is important to the network more than any other peer. In the partially decentralized case, a subset of nodes can play more important roles than others, e.g. by maintaining more information about their neighbor peers and thus acting as bigger directories that can improve the performance of a search process. This set of relatively more important peers can drastically vary in size while the system remains to be functioning. In the Hybrid Decentralization case, the whole system depends on one or very few irreplaceable nodes which provide a special functionality in one aspect such as a directory service. However, all other nodes in the system, while depending on one special node, are of equal functionality and they autonomously offer services to one another in a different aspect such as storage. Thus, a system of that class is a hybrid system that is centralized in one aspect and decentralized in another aspect.

Anyhow, topology structure and degree of decentralization are not the only parameters that lead to proper classifications of Peer-to-Peer systems. What follows in Fig. 1.2 is a much more specific and focused taxonomy set of properties in which aspects such as security or application issues are taken into consideration. [RIS-28].



Taxonomy	Selected References
Search	(Yang and Garcia-Molina 2002b; Yang and Garcia-Molina 2002a; Yang and Garcia-Molina 2002c; Balakrishnan, Kaashoek et al. 2003; Bawa, Sun et al. 2003; Cooper and Garcia-Molina 2003b; Daswani, Garcia-Molina et al. 2003; Gummadi, Gummadi et al. 2003; Hellerstein 2003; Huebsch, Hellerstein et al. 2003; Shi, Guangwen et al. 2004)
Semantic-Free Indexes *Plaxton Trees* *Rings* *Tori* *Butterflies* *de Bruijn Graphs* *Skip Graphs*	(Devine 1993; Litwin, Niemat et al. 1993; Litwin, Neimat et al. 1996; Karger, Lehman et al. 1997; Plaxton, Rajaraman et al. 1997; Rowstron and Druschel 2001a; Stoica, Morris et al. 2001; Zhao, Kubiatowicz et al. 2001; Harvey, Dunagan et al. 2002; Li and Plaxton 2002; Malkhi, Naor et al. 2002; Maymounkov and Mazieres 2002; Ratnasamy, Shenker et al. 2002; Zhao, Duan et al. 2002; Aberer, Cudre-Mauroux et al. 2003; Aspnes and Shah 2003; Cates 2003; Gupta, Birman et al. 2003; Harvey, Jones et al. 2003a; Kaashoek and Karger 2003; Loguinov, Kumar et al. 2003; Rhea, Roscoe et al. 2003; Stoica, Morris et al. 2003; Ganesan, Krishna et al. 2004; van Renesse and Bozdog 2004; Zhao, Huang et al. 2004)
Semantic Indexes *Keyword Lookup* *Peer Information Retrieval* *Peer Data Management*	(Clarke, Sandberg et al. 2001; Jovanovic 2001; Babaoglu, Meling et al. 2002; Kalogaraki, Gunopulos et al. 2002; Klingberg and Manfredi 2002; Lv, Cao et al. 2002; Lv, Ratnasamy et al. 2002; Ripeanu, Iamnitchi et al. 2002; Schlosser, Sintek et al. 2002; Sunaga, Takemoto et al. 2002; Bawa, Manku et al. 2003; Chawathe, Ratnasamy et al. 2003; Joseph and Hoshiai 2003; Nejdl, Siberski et al. 2003; Tatarinov, Mork et al. 2003; Yang and Garcia-Molina 2003; Zhang, Shi et al. 2003; Cai and Frank 2004; Loo, Huebsch et al. 2004b; Tempich, Staab et al. 2004)
Search *Range Queries* *Multi-Attribute Queries* *Join Queries* *Aggregation Queries* *Continuous Queries* *Recursive Queries* *Adaptive Queries*	(Litwin, Neimat et al. 1994; Litwin and Neimat 1996; Avnur and Hellerstein 2000; Aberer 2002b; Andrzejak and Xu 2002; Harren, Hellerstein et al. 2002; Hodes, Czerwinski et al. 2002; Aberer, Datta et al. 2003; Albrecht, Arnold et al. 2003; Aspnes and Shah 2003; Bhagwan, Varghese et al. 2003; Cai, Frank et al. 2003; Daswani, Garcia-Molina et al. 2003; Gedik and Liu 2003b; Gedik and Liu 2003a; Gupta, Agrawal et al. 2003; Harvey, Jones et al. 2003a; Hellerstein 2003; Huebsch, Hellerstein et al. 2003; Montresor, Jelasity et al. 2003; Ratnasamy, Francis et al. 2003; Triantafillou and Pitoura 2003; Tsangou, Ndiaye et al. 2003; van Renesse, Birman et al. 2003; Zhang, Shi et al. 2003; Albrecht, Arnold et al. 2004; Aspnes, Kirsch et al. 2004; Bawa, Gionis et al. 2004; Bharambe, Agrawal et al. 2004; Ganesan, Bawa et al. 2004; Jelasity, Kowalczyk et al. 2004; Loo, Huebsch et al. 2004a; Ramabhadran, Ratnasamy et al. 2004; Schmidt and Parashar 2004; Tanin, Harwood et al. 2004; van Renesse and Bozdog 2004; Yalagandula and Dahlin 2004)
Storage	
Consistency & Replication *Eventual consistency* *Trade-offs…*	(Lomet 1996; Cohen and Shenker 2002b; Cohen and Shenker 2002a; Weatherspoon and Kubiatowicz 2002b; Geels and Kubiatowicz 2003; On, Schmitt et al. 2003; Gopalakrishnan, Silaghi et al. 2004b) (Kubiatowicz, Bindel et al. 2000; Rhea, Wells et al. 2001; Rowstron and Druschel 2001b; Adya, Wattenhofer et al. 2002; Lin, Lian et al. 2004)
Distribution *Epidemics, Bloom Filters*	(Bloom 1970; Bailey 1975; Demers, Greene et al. 1987; Gupta, Birman et al. 2001; Hodes, Czerwinski et al. 2002; Mohan and Kalogaraki 2002; Weatherspoon and Kubiatowicz 2002a; Aberer, Cudre-Mauroux et al. 2003; Birman 2003; Costa, Migliavacca et al. 2003; Eugster, Guerraoiu et al. 2003; Ganesh, Kermarrec et al. 2003; Gupta, Birman et al. 2003; Koloniari, Petrakis et al. 2003; Koloniari and Pitoura 2003; van Renesse, Birman et al. 2003; Vogels, Renesse et al. 2003; Voulgaris and van Steen 2003; Birman and Gupta 2004; Costa, Migliavacca et al. 2004; Eugster, Guerraoiu et al. 2004; Gupta 2004; Koloniari and Pitoura 2004)
Fault Tolerance *Erasure Coding* *Byzantine Agreement …*	(Byers, Considine et al. 2002; Rodrigues, Liskov et al. 2002; Weatherspoon and Kubiatowicz 2002b; Weatherspoon, Moscovitz et al. 2002; Castro, Rodrigues et al. 2003; Cates 2003; Maymounkov and Mazieres 2003; Naor and Wieder 2003b; Plank, Atchley et al. 2003; Krohn, Freedman et al. 2004)
Locality	(Gummadi, Saroui et al. 2002; Hildrum, Kubiatowicz et al. 2002; Keleher, Bhattacharjee et al. 2002; Li and Plaxton 2002; Ng and Zhang 2002; Zhao, Duan et al. 2002; Freedman and Mazieres 2003; Gummadi, Gummadi et al. 2003; Harvey, Jones et al. 2003b; Massoulie, Kermarrec et al. 2003; Pias, Crowcroft et al. 2003; Ruhl 2003; Sollins 2003; Awerbuch and Scheideler 2004a; Cox, Dabek et al. 2004; Dabek, Cox et al. 2004; Dabek, Li et al. 2004; Fessant, Handurukande et al. 2004; Hildrum, Krauthgamer et al. 2004; Karger and Ruhl 2004a; Liu, Liu et al. 2004; Lua, Crowcroft et al. 2004; Mislove and Druschel 2004; Zhang, Zhang et al. 2004)
Load Balancing	(Aberer, Datta et al. 2003; Adler, Halperin et al. 2003; Baquero and Lopes 2003; Byers, Considine et al. 2003; Kaashoek and Karger 2003; Rao, Lakshminarayanan et al. 2003; Ruhl 2003; Aspnes, Kirsch et al. 2004; Castro, Lee et al. 2004; Gao and Steenkiste 2004; Gopalakrishnan, Silaghi et al. 2004a; Karger and Ruhl 2004b; Karger and Ruhl 2004c; Manku 2004; Stavrou, Rubenstein et al. 2004; Wang, Zhang et al. 2004)
Security	
Character *Identity* *Reputation and Trust* *Incentives*	(Damiani, di Vimercati et al. 2002; Blaze, Feigenbaum et al. 2003; Buragohain, Agrawal et al. 2003; Caronni and Waldvogel 2003; Schneidman and Parkes 2003; Anagnostakis and Greenwald 2004; Feldman, Lai et al. 2004; Marti, Ganesan et al. 2004; Papaioannou and Stamoulis 2004; Selcuk, Uzun et al. 2004; Sieka, Kshemkalyani et al. 2004)
Goals *Availability* *Authenticity* *Anonymity* *Access Control* *Fair Trading*	(Clarke, Sandberg et al. 2001; Daswani and Garcia-Molina 2002; Fiat and Saia 2002; Freedman and Morris 2002; Hazel and Wiley 2002; Serjantov 2002; Sit and Morris 2002; Bawa, Sun et al. 2003; Cox and Noble 2003; Daswani, Garcia-Molina et al. 2003; Ngan, Wallach et al. 2003; Ramaswamy and Liu 2003; Singh and Liu 2003; Surridge and Upstill 2003; Berket, Essiari et al. 2004; Feldman, Papadimitriou et al. 2004; Josephson, Sirer et al. 2004; O'Donnel and Vaikuntanathan 2004)
Applications	(Li 2002; Considine, Walfish et al. 2003; Karp, Ratnasamy et al. 2004; Roussopoulos, Baker et al. 2004)
Memory *File Systems* *Web* *Content Delivery Networks* *Directories* *Service Discovery* *Publish / Subscribe ..*	(Dabek, Kaashoek et al. 2001; Gold and Tidhar 2001; Kan and Faybishenko 2001; Annexstein, Berman et al. 2002; Kangasharju, Ross et al. 2002; Muthitacharoen, Morris et al. 2002b; Muthitacharoen, Morris et al. 2002a; Saroiu, Gummadi et al. 2002b; Mislove, Post et al. 2003; Fessant, Handurukande et al. 2004) (Iyer, Rowstron et al. 2002; Bawa, Bayardo et al. 2003; Li, Loo et al. 2003; Freedman, Freudenthal et al. 2004; Loo, Krishnamurthy et al. 2004) (Cuenca-Acuna, Peery et al. 2002; Junginger and Lee 2004; van Renesse and Bozdog 2004) (Balazinska, Balakrishnan et al. 2002; Chander, Dawson et al. 2002; Cox, Muthitacharoen et al. 2002; Hodes, Czerwinski et al. 2002; Iamnitchi 2003; Awerbuch and Scheideler 2004b; Walfish, Balakrishnan et al. 2004)
Intelligence *GRID* *Security…*	(Hoschek 2002; Iamnitchi, Foster et al. 2002; Foster and Iamnitchi 2003; Lo, Zappala et al. 2004) (Ajmani, Clarke et al. 2002; Aberer, Datta et al. 2004)
Communication *Multicasting* *Streaming Media* *Mobility* *Sensors …*	(Zhuang, Zhao et al. 2001; Castro, Druschel et al. 2002; Halepovic and Deters 2002; Lienhart, Holliman et al. 2002; Ratnasamy, Karp et al. 2002; Stoica, Adkins et al. 2002; Castro, Druschel et al. 2003; Demirbas and Ferhatosmanoglu 2003; Hefeeda, Habib et al. 2003; Ratnasamy, Karp et al. 2003; Sasabe, Wakamiya et al. 2003; van Renesse, Birman et al. 2003; Voulgaris and van Steen 2003; Zhang and Hu 2003; Hellerstein and Wang 2004; Hsieh and Sivakumar 2004; Nicolosi and Mazieres 2004; Padmanabhan, Wang et al. 2004; Sit, Dabek et al. 2004; Tran, Hua et al. 2004; Wawrzoniak, Peterson et al. 2004; Zhou and van Renesse 2004)

Fig. 1.2 Taxonomy properties and associated literature

Nevertheless, the taxonomy in Fig. 1.1 has more widespread acceptation and is thereafter more convenient and simple for this thesis' purposes than those that want to take into consideration aspects that have in mind a more pragmatic view of the use in which the P2P systems are going to be put.

1.2.4 Trends

Distributed Hash Tables are a cornerstone of state-of-the-art Peer-to-Peer systems. They mean a remarkable advance in solving the issue of scalability and decentralization, with the added value of determinism and high guarantees. However, this has opened a whole set of new questions that need to be addressed. What follows is a summary of those issues, from [SEA-8]. Quoting:

Lack of a Common Framework Research in DHT systems has been addressed by different research groups. The result was the emergence of systems that are very similar in basic principles. Nevertheless, there is no common framework that allows the common understanding and reasoning about those systems.
Locality Though accounted for in systems like Pastry and Tapestry, locality remains to be an open research issue. Additionally, the loss of locality due to hashing is not always considered a disadvantage. The Oceanstore system [OCS-4] which depends on Tapestry for location and routing, considers loss of locality favorable because replicas of items would be stored at physically apart nodes which renders a system resistant to denial of service attacks.
Cost of Maintaining the Structure Most of the current DHTs depend on the periodic checking and correction (stabilization) for the maintenance of the structure which is crucial to the performance properties of those systems. This periodic activity costs a high number of messages and sometimes unnecessarily in the case of checking stable sections of a routing table. The awareness about this problem motivated research such as e.g., [MAH-23] where a network tries to "self-tune" the rate at which it performs periodic stabilization.
Complex Queries DHTs assume that for each item, there is a unique key and to retrieve the item one must know the key. That is, one can not search for items matching a certain criteria like a keyword or a regular-expression-specified query. The feasibility of the task is questionable [JLI-29]. Some approaches include the insertion of indices [HAR-30] for general queries or using some geometrical constructs that make use of the DHT structure such as space-filling curves [AND-31]. Another approach is to let the hashing be based on keywords or semantic information and not on unique keys [SCH-32].
Heterogeneity While all DHT systems aim at letting all nodes have equal duties and responsibilities, the heterogeneity in physical connectivity makes them unequal. Consequently, nodes with higher latencies constitute bottlenecks for the operation of structured P2P systems. Two approaches were suggested to cope with those problems: i) Cloning: The more powerful nodes are cloned so they can act as multiple nodes and receive higher percentage of the uniformly distributed traffic [DAB-33] ii) Clustering: Nodes of similar latency behavior are clustered together [ZXU-34].
Group Communication Since structured P2P systems offer graphs of known topologies to connect peers, it is natural to start exploiting the structural properties in group communication. The main focus in P2P Group communication is on multicasting. Extensions like [STO-35], [RAT-36], [CAS-37] aim at providing multicast layers to existing DHT systems. Publish-subscribe communication [TAN-

38] is also another form of group communication that was researched in P2P systems [BAE-39].

Grid Integration P2P and the Grid are two fields that share key properties such as being large scale distributed systems and the goal of sharing networked resources. The properties of scalability and self organization provided by recent P2P infrastructures are interesting properties for Grid applications. Actually, both research communities are starting to merge, we can observe that from conferences like the International Conference on Peer-To-Peer Computing [IIC-40] and the International Conference on Cluster Computing and the Grid (CCGRID) [ISC-41]. Additionally, the P2P working group [PTP-12] and The Global Grid Forum [GGF-42], two respective standardization efforts, started to merge their efforts [ROG-43].

1.3 Contribution

The contribution of the research contained in this master thesis to the area of distributed systems — focused on Chord — can be summarized as:

- description of the Chord protocol
- design and implementation of a simulator in Java
- design and implementation of a Chord node in Java
- design of the scenarios that are representative to take measurements
- generation of the data that the experiments need as input for these scenarios
- execution of Chord with these experiments in the simulator in order to review its behavior
- data gathering and representation
- study of data and results interpretation
- validation of the data found in the paper by Ion Stoica et al. [STO-1]

The following chapters in this report include a survey about Peer-to-Peer systems and more specifically, structured P2P systems (ch. 2) followed by some Chord basic principles and internals (ch. 3), a description of the programming that took place before field work (ch. 4) and the description of the experiments that were conducted (ch. 5). Finally, the results are presented (ch. 6) and the future work and overall conclusions stated (ch. 7 & 8). Appendixes can be found at the end of the document (ch. 9), including a glossary, a short user manual of the simulator, the javadoc of the software and references.

2 The Chord Protocol

This chapter presents the mathematic concepts on top of which Chord is sustained, as well as the design of inner data structures and functionalities of the protocol. Most of this is based on the paper by Ion Stoica et al. [STO-1] as well as in their technical report [STO-44]; a more detailed and technical explanation of how the protocol behaves can be found there. What follows is an excerpt of such text, in order to provide some insight on the general ways in which the protocol works and why. Some examples are provided here to make certain aspects more clear, and certain additions that provide better performance are included too.

2.1 Introductory concepts:

There are two mathematical concepts that are basic for the understanding of Chord's behavior: **hash functions** and **modular arithmetic**. What follows is a short introduction on them.
As a starting point, and to simplify calculations, we define the maximum size of any network that we want to build or study to be $\mathbf{N} = \mathbf{2^m}$. This means that this size is power of two. The relevancy of this **m** value will be uncovered later on.
SHA-1 is the hash function that Chord —as described in [STO-1]— uses, but it worth noting that the protocol is not tied to any particular one.

2.1.1 Hash functions

Each node belonging to the network is assigned a number through the use of a hash function. Each item that is going to be made available (searchable, or retrievable) has such a numeric association too.
Hash functions usually convert an input from a (typically) large domain into an output in a (typically) smaller range.
The domain can be any number, or any data that can be represented in a numeric way. In the case of the IDs of nodes belonging to a Chord network, the IP address, or the <IP,port> pair are good candidates as such input, and thus serve as a value from the domain in the hash function. In the case of items or resources to be shared in the network, the name of a file or resource, or even their contents can also be represented in a numeric way, making its hashing possible.
The reason why hashing is used resides in the fact that these functions randomize and disperse values.

- **Randomization**: given a value **X** from the domain, **hash(X)** will be a value from the range of the function with a certain degree of randomness. This only means that small values of **X** will not necessarily mean small (nor specifically big either) values of **hash(X)**.
- **Dispersion**: given two similar or close values of the domain, **X** and **Y**, there is high probability that **hash(X)** and **hash(Y)** will be distant one from each other.

Hence, two nodes with similar <IP,port> values (belonging to a certain LAN/WAN, other factors like geographical proximity, or simply resembling values) will end up having very different numeric values after being applied a hash function. The same holds for resources with similar contents previous to hashing.
More information about hashing properties that apply to our needs can be found in the article about hash functions [CAR-45], a standard about secure hash [FIP-46], the paper by David R. Karger et al. on consistent hashing [KAR-47] and D. Lewin's master thesis about the same issue [LEW-48].

2.1.2 Modular arithmetic

Chord is a protocol whose behavior is based entirely on the topology that the network forms. Modular arithmetic is the cornerstone upon which this topology lies.
As a result of the transformation mentioned in the previous section, the numeric representation of both nodes and items will belong to a certain range of numbers [0,X). This numbers will be operated in a modulo arithmetic (see glossary), which means, in "modulo p": 0+1=1, 1+1=2, (p-1)+1=p=0, p+1=1, and so on; Fig. 2.1 shows an example, in "modulo 3":

```
0 (modulo 3) = 0
1 (modulo 3) = 1
2 (modulo 3) = 2
3 (modulo 3) = 0
4 (modulo 3) = 1
5 (modulo 3) = 2
6 (modulo 3) = 0
7 (modulo 3) = 1
etc...
```

Fig. 2.1 Example of numeric equivalences in "modulo 3"

As it will be seen later, certain operations of the Chord protocol need to perform additions to the identifiers of nodes and items, and those additions will be done following the rules stated above.

2.2 Network topology

Following the last section's contents, Chord's behavior is defined in terms of the way that nodes organize themselves, the so called *topology of the network*. What follows is a description of this topology in two stages.

2.2.1 Basic layout

The two main actors in Chord are nodes and items.
Nodes belonging to the Chord network will be referred to as *node* or its identifier *id*, and shared items as *documents* or *keys*. Any of those is a number belonging to the range **[0,2^m)**.
Each node in the network will be responsible for a set of *keys*. Unlike most other common P2P applications assume, a Chord node is not automatically responsible for the keys it wants to share in the network. When a node shares an item, this item's key will be inserted in the network and will be assigned as a responsibility to (probably) another node.
Now, this is how nodes and documents organize themselves with respect to each other: identifiers are ordered in a **modulo 2^m** ring. Key *k* is assigned to the first node whose identifier is equal to or follows (the identifier of) *k* in the identifier space, regardless of which node was originally the owner of the file (or resource) that generated this key. This node is called the *successor* node of key *k*, denoted by *successor(k)*. If identifiers are represented as a circle of numbers **from 0 to 2^m-1**, then *successor(k)* is the first node whose assigned identifier is k or, in the absence of this, the first node found clockwise from *k* in the ring. In the remainder of this thesis, I will also refer to this circle of identifiers as the *Chord ring*.
Fig. 2.2(a) below illustrates a Chord ring with 10 nodes. Fig. 2.2(b) shows node 14 requesting the insertion of document 24. When inserted, document 24 becomes responsibility of node 32, which is the present successor of key 24, as shown in Fig. 2.2(c).

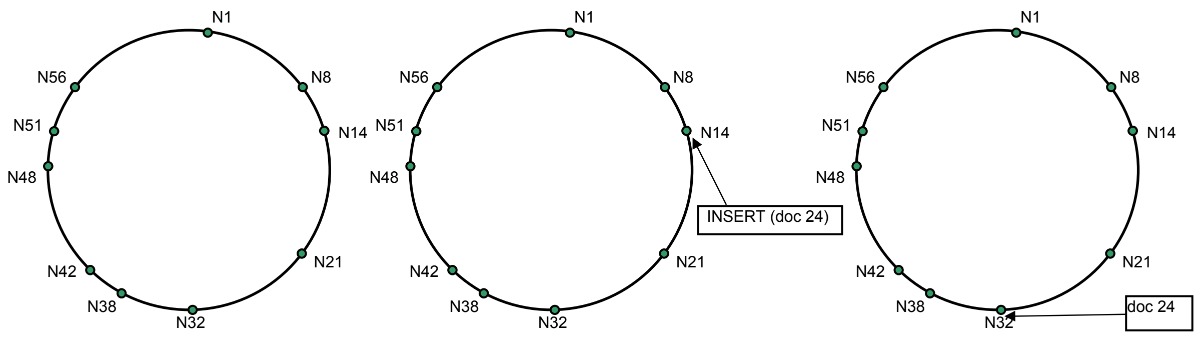
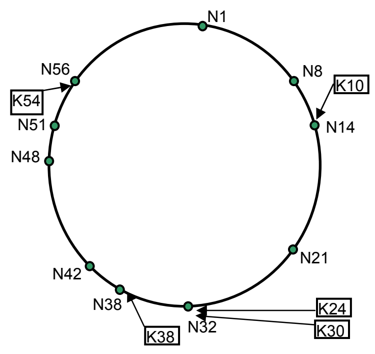


Fig. 2.2 Assignment of responsibilities:
a) Chord ring with 10 nodes b) node 14 inserts key 24 c) node 32 is responsible of key 24

Then again, the basic topology is that every *node* knows its successor, forming the Chord ring. What follows is an example of a Chord ring with **m=6**. Every *identifier* (or *key*) in the network would be a number **$0<=X<=2^6$**, so **$0<=X<64$**, or **$X\in[0,64)$**. The Chord ring in this scenario could accommodate a maximum of **$N=2^m=64$** ***nodes***, each one of them with an identifier **$X\in[0,64)$**. As a clarification, if such a network existed, each one of those nodes would be responsible for one *key* at maximum, the *key* being equal to its node *identifier*, according to what was illustrated in Fig. 2.2.
This example has 10 *nodes*, with identifiers: 1, 8, 14, 21, 32, 38, 42, 48, 51 and 56, as in the previous example. Some of the *nodes* are responsible for a set of *keys* present in the network. We could say that *keys* (or *documents*) 10, 24, 30, 38 and 54 are in the network, available for any peer to be retrieved. And each one of those *documents* is held by its responsible *node*. As explained before, a *node* with identifier *id* is responsible for *document d* if *id=successor(d)*. This whole setup would be enough to provide lookup search capabilities with linear cost (the average number of hops necessary to locate a *key* would be **O(N)**). Fig. 2.3(b) shows the pseudocode for a lookup operation in RPC [WIK-61] format, and Fig. 2.3(c) shows a graphical description of its behavior when *node* 8 requests *document* 54, on the ring previously described and showed in figure Fig. 2.3(a)



```
//Node n asks to find successor of id
n.findSuccessor(id){
  if (id ∈ (n,successor])
    return successor;
  else
    //forward the query around the circle
    return successor.findSuccessor(id);
}
```

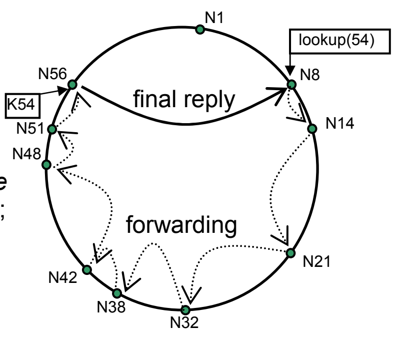


Fig. 2.3 Example of lookup in its simplest form: linear forwarding around the ring
a) the Chord ring b)pseudocode for lookup c) forwarding of the request

2.2.2 Further data structures

As mentioned above, this is enough if what we want is to provide lookup search capabilities, and we are content with linear cost, **O(N)**.
In order to achieve the goals of the protocol (improved efficiency, performance and scalability, fault tolerance, etc), further data structures are required.
Each node has the following data regarding other nodes in the network:

- **A fingers table with *m* entries**. *m* refers to the number of bits that limit the identifier space. Each given entry *i* in the *finger* table holds, *0<=i<m* the identifier of *successor(id+2^i)*. This structure offers lookup performance improvement. Note that *fingers*[0] holds *successor* of (id+1), which means *THE successor*, or said in other words, the next node found clockwise in the Chord ring; so the variable *successor* introduced in the last subsection is not needed anymore, as its equivalent is now part of the *finger table*. This structure is the one that will ensure that lookups will be performed with cost **O($\log_2$N)**, given that the Chord ring has identifiers belonging to **[0,2^m)**, and the size of the network is at most **N = 2^m**. Fig. 2.4 shows a couple of examples of the finger tables of nodes 14 and 38, for the network shown in Fig. 2.3. Given that this network had an identifier space limited by **m=6**, the finger tables have 6 entries:

Finger table for Node 14:

finger level	**aim NodeID+2^{level}**			**successor of aim**
0	14+2^0	14+1	15	21
1	14+2^1	14+2	16	21
2	14+2^2	14+4	18	21
3	14+2^3	14+8	22	32
4	14+2^4	14+16	30	32
5	14+2^5	14+32	48	48

Finger table for Node 38:

finger level	**aim NodeId+2^{level}**			**successor of aim**
0	38+2^0	38+1	39	42
1	38+2^1	38+2	40	42
2	38+2^2	38+4	42	42
3	38+2^3	38+8	46	48
4	38+2^4	38+16	54	56
5	38+2^5	38+32	68	8

Note that successor of 68 is 8.
68 modulo 2^6 = 4. The successor of 4 is 8.

Fig. 2.4 Finger tables for nodes 14 and 38

- **The predecessor**. *p=predecessor(n)* means that *n=successor(p)*. Expressed in the same terms as used for the *successor* definition, given that the identifiers are represented in a circle of numbers **from 0 to 2^m-1**, the *predecessor* of *n* is the first node found counter-clockwise from *n* in the Chord ring. This is necessary for internal management of the topology as the network changes (nodes joining and leaving).
- **A successors list**. This is a list of the next nodes found clockwise. As will be explained later, the longer this list is, the more tolerant to simultaneous failures of nodes is the network. The successors list will be named "sList" when referenced in the pseudocode that appears in following sections.
- **A referrers list**. This is a list of the nodes that are pointing to the node from any of their *fingers*. They are useful in the event of a node leaving the network. When a node will leave, it will let all the *referrers* know, so each *referrer* will be able to substitute the *finger* for an appropriate node (which is always the *successor* of the leaving node). This is an improvement to what is documented by Ion Stoica et al. in their paper [STO-1].

2.3 Operations in Chord

This section contains information about significant parts of the code that the protocol uses to achieve its goals. Certain routines are called periodically. The way this has been implemented is by making calls to a "*schedule*" function which takes care of calling the argument in a future time, e.g.: *schedule(foo)* will make *foo()* to be called in a future. If the last thing that *foo()* does is to call *schedule(foo)*, this will result in *foo()* being called periodically.

2.3.1 Join

When a node joins the network, its successor and predecessor are set to none (*null*).

The first thing a node **X** does when being inserted is to request to any node **Y** present in the network who **X**'s *successor* is. When **X** receives a reply, it stores its *successor*'s id.

The *stabilization* and *fixFingers* routines are called for the first time, and will be executed periodically. This will ensure that the *predecessor*, the *fingers,* as well as the *successors list* (sList) and the *referrers list* stay up to date. Fig. 2.5 shows the most significant part of the pseudocode involved in the creation of a Chord ring with the first node and the join operation.

```
//create a new Chord ring
n.create(){
  predecessor := nil;
  successor := n;
  schedule(stabilize);
  schedule(fixFingers);
}
```

```
//n joins a Chord ring containing node c
n.join(c){
  predecessor := nil;
  successor := c.findSuccessor(n);
  schedule(stabilize);
  schedule(fixFingers);
}
```

Fig. 2.5 Pseudocode involved in the creation of a Chord ring and insertion of nodes

Further details concerning keys should be taken into consideration too. Given that consistent hashing provides the network with the ability to let nodes enter and leave it with minimal disruption, when a node n enters the network certain keys previously assigned as a responsibility to *n*'s successor now should become assigned to *n*, e.g.: if node 32 is responsible for keys 15, 18 and 30, and now node 20 joins the network, it follows that keys 15 and 18 will be now responsibility of the recently joined node. These operations have been included in the final implementation as part of the stabilization procedures.

2.3.2 Lookup

The lookup operation is the heart and core of the protocol: it is what justifies its design. Its performance and reliability stem from the data structures that a node holds and maintains.

Let us assume a node with identifier *n* is interested in locating key *id*: if *id* lies between *n* and *n*'s successor in the identifier circle, the result of the operation is *n*'s successor.

Otherwise, the lookup request is forwarded to the closest preceding node in the network that *n* knows about by inspecting the *fingers table*. This way, by forwarding petitions, the lookup operation will make steps closer and closer clockwise in the identifier circle to reach its destination. Note that at each forwarding step, the forwarder node goes as far away in the identifier circle as its data allows to, and that fact is the one that will ultimately justify the **$O(\log_2 N)$** cost. Later in the analysis of results section it will be justified that the cost is about **$1/2(\log_2 N)$** in average.

When the *successors list* structure is used, it does not only provide robustness in the event of node failures, but also gives a slight performance improvement. When looking for the closest preceding node to forward a lookup request, this structure can be inspected too in order to save some of the last forwarding hops. What follows in Fig. 2.6 is the pseudocode for the *find_successor* routine — which is what a lookup query is mostly about—, and the *closest_preceding_node* routine, used by the former one to locate the best forwarding candidate among the *fingers table;* at this stage, the use of the *successors list* is obviated for simplicity reasons. Fig. 2.7 shows an example of a lookup query for the same key illustrated in Fig. 2.3, but with the advantage of using the *fingers table* this time (the hops are larger):

```
//ask node n to find the successor of id
n.find_successor(id){
  if (id ∈ (n,successor])
    return successor;
  else
    c = closest_preceding_node(id);
    return c.find_successor(id);
}
```

```
//search the finger table for the
//highest predecessor of id
n.closest_preceding_node(id){
  for i = m downto 1
    if(finger[i] ∈ (n,id))
      return finger[i];
  return n;
  //the successors list can be searched too
  //for the most appropriate candidate
}
```

Fig. 2.6 Pseudocode of the routines involved in the most critical operation: the lookup

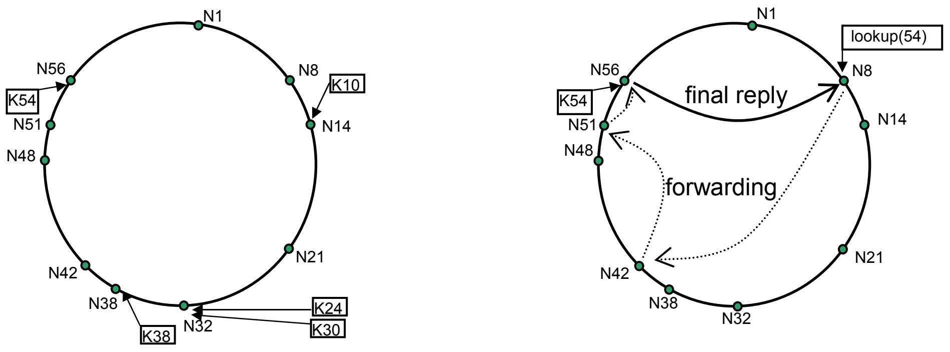


Fig. 2.7 Example of a lookup request: node 8 asks for key 54

2.3.3 Stabilization

The network is kept stable (or converges to a stable network) by means of two operations: *stabilize* and *fixFingers*.

- **The *stabilize* operation:** It ensures that *successor* and *predecessor* pointers are kept up to date. The successor is updated when a new node has been inserted in the identifier circle between the node running the stabilization routine and its successor —this is done by asking for the *successor*'s *predecessor*. Next thing the routine does is requesting the *successors list* of its *successor*, and then build its own by removing the last item and prefixing the successor as first item. Then, the routine lets its *successor* know about its existence, by calling the *notify* procedure. When a node receives a *notify* call, it checks from which node it comes, and updates the *predecessor* pointer if necessary after having checked out that the node who claims to be the predecessor is a better candidate than the existing *predecessor*. The last thing done in the *stabilize* operation is to re-schedule itself to guarantee a periodical execution of the call. What follows is a figure with the pseudocode involved in the stabilization procedures:

```
//called periodically, verifies n's immediate
//successor, and tells the successor about n
n.stabilize(){
  x = successor.predecessor;
  if (x ∈ (n,successor))
    successor = x;
  sList = Shift(successor.sList);
  successor.notify(n);
  schedule(stabilize);
}
```

```
//p claims it might be n's predecessor
n.notify(p){
  if (predecessor == nil OR p ∈ (predecessor,n))
   predecessor = p;
}
```

Fig. 2.8 Pseudocode of *stabilize* and *notify*: these operations ensure that *successor* and *predecessor* pointers are kept up to date. These ultimately ensure correct answers to requests.

- **The *fixFingers* operation:** It updates one entry of the *finger* table at a time, scheduling the next update for a later round of *finger table* correction, which happens periodically. It basically consists of making a call to find_successor, and looking for the best fit existing node in the network for the position of the finger table that is being corrected. If the reply to the find_successor call is the same as the content of the fingers table, no correction is needed; when corrections are made the referrers list of both the old finger and the new finger are updated consequently. What follows in Fig. 2.9 is the pseudocode of the *fixFingers* routine:

```
//called periodically, refreshes finger table entries
//next stores the index of the next finger to fix
n.fixFingers(){
  next = next + 1;
  if (next > m)
    next = $\lfloor \log_2(successor-n) \rfloor + 1$; //first non-trivial finger
  aux = find_successor(n + $2^{next-1}$);
  if (aux != finger[next])
    finger[next].removeFromReferrerList(n);
    finger[next] = aux;
    finger[next].addToReferrerList(n);
  schedule(fixFingers);
}
```

Fig. 2.9 Pseudocode for the fixFingers operation: this ensures that lookup requests are kept efficient

2.3.4 Failure

Given that one of Chord's desired strengths is robustness, the nodes need to have a way to learn about other nodes disappearing from the network —which is denoted as a node failure in this thesis—, regardless of whether this absence is voluntary or not. The way that nodes know about this is with **negative acknowledgement**. A node sends periodically a message to its *predecessor* to see if it is alive. If the sending node does not receive a reply —within a certain timeout—, it means that the receiver is not in the network anymore.

```
//called periodically, checks whether predecessor has failed
n.check_predecessor(){
  if (hasFailed(predecessor))
    predecessor = nil;
}
```

Fig. 2.10 Pseudocode for the verification of the network robustness

The worst case is when a node just drops from the network or disappears without making other nodes in the network notice. In order to make it possible for the network to maintain its invariants of robustness and performance, a node checks periodically for the presence of its *predecessor*. If the predecessor is not present anymore, the predecessor pointer is set to none (*null*), and the *stabilization* routine will make the rest of the job, because nodes send a *notify* call to their *successors* periodically. This is when the *successors list* comes in useful. Fig. 2.11 illustrates the example using the same ring, and focusing on nodes 21, 32 and 38.

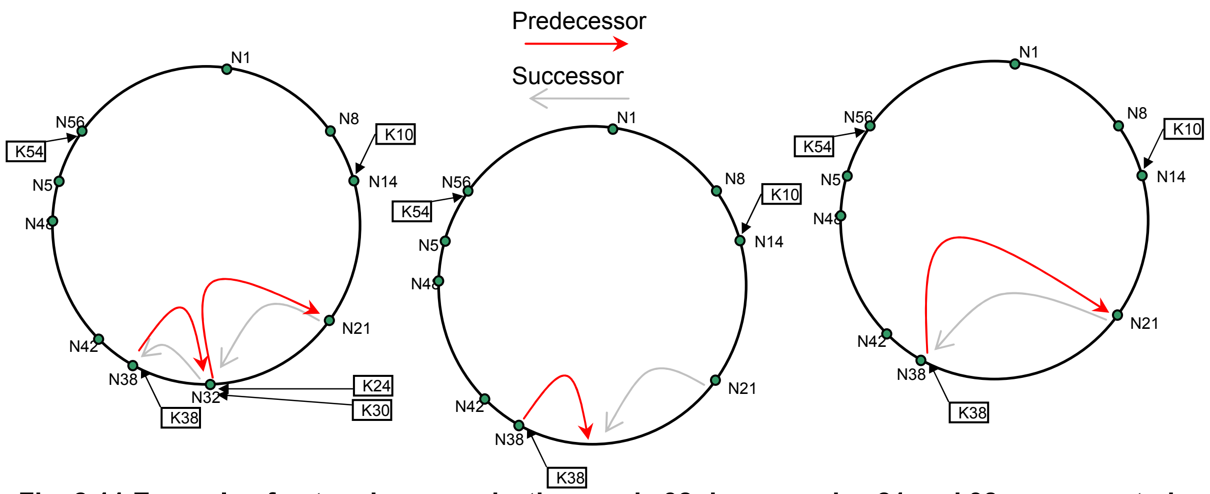


Fig. 2.11 Example of network reorganization: node 32 drops; nodes 21 and 38 are corrected
a) detail of nodes 21, 32, 38 b) nodes 32 fails, and drops c) the chord ring is corrected

If node 32 fails and drops from the network (Fig. 2.11(b)), the next time that 38 checks its *predecessor* it will realize that 32 is no more in the network. 38 will set its *predecessor* to null. And also, next time that 21 runs the *stabilize* routine, it will ask 32 about its *predecessor*, and 32 will not reply; hence, 21 will understand that 32 is not in the network anymore. Being it so, node 21 will remove 32 from the *successors list* and the *finger table* (remember that the first entry of the *finger table* is the *successor*). Instead, the next *successor* it knows about will be used. Say, for example, that every node in the network has a *successors list* of size 3. This means that each node knows about the 3 next nodes found clockwise counting from their own identifier. Thus, node 21 had this *successors list* before 32's failure: {32, 38, 42}. As node 32 has disappeared, the next *successor* that 21 knows about is 38. 32 is then removed from the *finger table* and the *successors list*. Now, *stabilize* will be called again, and 21 will contact 38, asking about its *predecessor*. 38 will reply that its *predecessor* is *null* now, because its former *predecessor* has failed. This results in 21 not changing its *successor* (it has already been updated to 38 when 21 noticed that 32 failed). Next thing 21 does is notifying 38, claiming that it may be a proper *predecessor* for node 38. When 38 receives a *notify* from 21 it checks its *predecessor*; it being *null* now, node 38 will take 21 as its new *predecessor* (Fig. 2.11 (c)). Ion Stoica et al. prove in [STO-1] that a *successors list* with size **$r = \Omega(\log_2 N)$** is enough to make it possible for a network in which nodes fail with probability 1/2 to keep on offering both efficiency and performance “with high probability”. The phrase "with high probability" is justified too in the paper [STO-1] with arguments based on the randomness provided by hash functions, and used all along the discussion of robustness.

As said before, more comments about the choice of SHA-1 (or any other hash function, for that matter) follow. I will illustrate this with two examples of a malicious attack and an accident. These examples, though not representative of the potential Chord's weaknesses, are devised just to clarify the importance of the *successors list* structure for robustness issues.

Malicious attack: let us set the scenario in which an adversary wants to break the Chord ring, and they have the power to take down a set of computers at will. How many nodes, and which ones, should they choose for their attack to result in a destabilization of the network?

- How many: at least the size of the *successors list*.
- And which ones? Any set of nodes present in the Chord network whose Chord identifiers formed a *successor chain*. Note the fact that these identifiers are the result of applying the SHA-1 function to some original data related to the node, e.g. the <IP,port> pair.

In short: a list of **N consecutive nodes**, being N bigger than the *successors list*.
Why? Because the strength of Chord in the event of failures resides in the *successors list* structure.
Now, how difficult would it be for this malicious attacker to do that? Even in the event of this malicious attacker being able to "disconnect" a certain amount of nodes, chances are that they could not choose which nodes to disconnect at will. The most an attacker could do is disconnect, somehow, certain LANs or sub-networks. And even if they could choose individual nodes, the fact that Chord identifiers are the result of a hash function makes it fairly difficult for an attacker to know which IPs are the owners of the IDs that they would like to disconnect, and even more if these IPs are combined with a port number or some other identifying character that might be unknown to the adversary. This is because hash functions are not mathematically reversible. The most the adversary could do would be to map massive amounts of values from the domain to their result after being applied the hash function and try to take advantage of that — what is commonly known as a *dictionary attack*. It is, in general terms, a hard problem for the adversary to solve.

Accident: Let us consider the event of an accident happening in a certain geographical area. This accident implies that a set of computers that were running Chord at that time suddenly become disconnected. How does this affect the Chord network as a whole? Well, let us assume that all these nodes might have (or maybe not) similar IPs, and they are a significant percentage of the nodes forming the whole Chord network. Again, the strength of Chord relies both in the *successors list* structure (which is, remember, a chain of successive id values of the nodes present in the network) and the hash function used to create those ids. In most cases the dispersion achieved by the hash function ensures that nodes with similar IPs (belonging to certain sub-networks, or having the same network prefixes) will end up —most probably— having very distant Chord identifiers. This makes it unlikely that when the whole set gets disconnected the whole Chord network becomes utterly destabilized. The network will likely be underperforming during the time span before it self-stabilizes again. But it will not stop working, unless the percentage of nodes that fail is too big. Again, the bigger the size of the *successors list*, the bigger the set of failed nodes needs to be in order to destabilize the network beyond recovery. If instead of a chosen set of nodes those were random nodes failing, the same argument applies. Randomness and dispersion are qualities that ultimately provide robustness to Chord.

2.3.5 Leave

Regard that given Chord's robustness in the eventuality of failures, a node voluntarily leaving the network can be treated as a node failure, without real need to warn other nodes about it. However, performance can be improved through slight additions.

- A node leaving the network tells its *successor* about it. The *successor* takes advantage of knowing who its *predecessor* will be from that moment on. Also, the node can send to its *successor* the set of resources of which it was responsible upon departure, and that will be assigned to the *successor*. This means that the *successor* needs not wait for the *stabilization* routine in order to fix the *predecessor* pointer, and also increases the positive responsiveness of nodes when being asked about keys (documents) that were present in the network and that might have disappeared if the departing node had not passed them on.
- A node leaving the network tells its *predecessor* about it. The node sends along its *successors list*, and the *predecessor* will use it from then on. This implies that the *predecessor* knows who its *successor* is at once, and does not need to wait for the *stabilization* routine to fix it.
- Every node **X** knows which other nodes in the network (**a,b,c**...) are referring to it. When node **X** leaves, it sends a message to each one of the nodes referring to it in the finger tables (**a,b,c**...) so that these nodes can substitute the reference to **X** for a better one. The substitute of **X** in nodes **a,b,c**... will be *successor(**X**)*, which is a value that **X** will send to these nodes in the same message that lets them know that **X** is leaving.

2.3.6 Insert

Of course, any node belonging to the Chord network can share a new resource and make it available. The way the protocol works is, when a node *n* inserts a key *k*, it is responsibility of the node with *id = successor(k)* to maintain *k*, until departure.

Again, certain discussions about how an adversary could destabilize the network arise. Given a certain hash function, an adversary could choose a set of colliding keys to be inserted in the network, those that map to a single hash bucket, and thus make the network unbalanced, tearing apart fairness and dispersion arguments. The discussion is closed in the paper by Ion Stoica et al. [STO-1] claiming that "we expect that a non-adversarial set of keys can be analyzed as if it were random. Using this assumption, we state many of our results below as 'high probability' results".

Anyway, despite the fact that none of the experiments that were performed for the evaluation of the protocol included the insertion of keys in the network (and therefore, the assignment of the responsibility for the *key* to the *successor* node of that *key*), these features were included in the implementation. As a result, both the traffic generator and the Chord implementation that will be described in the next section take into consideration the possibility to add keys to the store, and act according to this. However, even if keys were inserted in the node's store, the lookup queries do not check the contents of the store before replying as the node implementation stands now, and further improvements like resource redundancy and replication should be taken into consideration for full effectiveness and efficiency of such features.

3 Objectives, Tools and Methodology

This chapter describes the objectives of the study as well as the work that took place to achieve those goals.

3.1 Objectives

The goal of this project is to provide a case study of Chord by means of measuring its behavior under three sets of conditions.
To do this, a simulator and certain other pieces of software were developed to provide the framework in which the experiments were conducted.
What follows is a description of this software and the tools that were used in order to achieve these goals.
Moreover, certain design decisions were taken, and some implementation details are significant enough to deserve being mentioned.

3.2 Equipment

3.2.1 Hardware

The only necessary hardware equipment for the consecution of this work was a desktop workstation. A standard white box PC with 512 Mb of RAM and a AMD Sempron 2400+ CPU was used. No big computation power was needed, although certain simulations took several hours to complete.

3.2.2 Software

The following software packages were used:

- *MS Windows XP Professional* with SP1
- *cygwin* (for GCC use)
- Linux: Debian Sarge Distribution with 2.6.9 kernel
- Borland *JBuilder X*
- Sun Microsystems JDK 1.5.0.02
- XML Spy
- Rational Rose 2000
- GNU Plot
- text editors
- traffic generator (based on original work by Ion Stoica at Berkeley)
- simulator (based on original work by Peep Kungas at SICS)

3.3 The traffic generator

3.3.1 The original traffic generator

The traffic generator is a tool by Ion Stoica, published in a BSD-style, 2-clause license.
In its original form, the traffic generator is a piece of software programmed in "C" that, being fed an input of instructions, generates a text output with events that can be executed by the simulator.
It only compiles with the GNU toolchain, so in order to take advantage of its existence, and instead of programming a brand new traffic generator, there were three alternatives that I could easily handle in order to work comfortably with it:

- Boot *Linux* every time changes in the code were needed or when the generator was to be used (drawbacks: time cost, changing platform every now and then)
- Connect to a remote machine in which I could compile and execute the traffic generator (drawbacks: time and connectivity)
- Install *cygwin* in the local machine under *Windows*. (drawback: disk space, affordable). This was the chosen one, given that Windows was the default work environment.

At the end, it was not necessary to make changes to the code often, but the issue of execution was still there, and the solution proved good anyway.
This is the way the traffic generator works:
The input file must contain lines with three different commands: **events**, **wait** or **exit**
1) **events num avg wjoin wleave wfail winsert wfind**
This command generates **join, leave, fail, insert** document and **find** document events

- **num** - represents the total number of events to be generated for this line command
- **avg** - represents the average distance in time ticks between two consecutive events; this distance is randomly distributed
- **wjoin, wleave, wfail, winsert, wfind** - represent weights associated to each event type; an event of a certain type is generated with a probability proportional to its weight

2) **wait time**
This command generates an event that inserts a pause in the simulation (usually this command is used to wait for network stabilization)

- **time** – the number of time ticks to be idle

3) **exit**
Generates an event to end simulation

When fed with a file that complies with the specification above, the traffic generator returns through *stdout* the list of events sorted by time; *stdout* can easily be redirected to a file, and this file can be used later for simulation purposes.

3.3.2 Changes to the original traffic generator

The traffic generator was modified in order to make it possible that the program accepts an additional parameter to provide in a command line argument the number of bits *m*. This *m* parameter defines the identifier space **$[0,2^m)$**, and the modulo under which the operations within the networks generated are applied. This modification lets the program take *m* as a command line parameter rather than feeding this number into a header file. By doing so, recompiling the program every time this number *m* changes is not needed, as it was in the original tool.

3.4 The simulator

The simulator on which the design was inspired was originally developed by Peep Kungas and adapted with help of Sameh El-Ansary. It was programmed in Oz. After a period of tests and development, I programmed a new one in Java with similar structure. The core of the simulator is, to put it simply, a procedure that counts time ticks, traverses every single node of the network at each tick, and inspects whether the node has pending events to be executed at the current time. If so, those events are executed. Events are modeled as text messages passed from one node to another. When a node receives a message, it executes the routine associated with the message. The actual implementation of the RPC (see glossary and [WIK-61] for a definition) model shown in the various examples of pseudocode seen so far in this thesis is message passing (see glossary, [WIK-62] and [NAP-63]).
The following sections describe different aspects of the simulator, and how it fulfills its specification.

3.4.1 Architecture

The package as a whole goes beyond this basic description, and further data structures are used. Fig. 3.1 shows an UML-like diagram of the classes that form the system with their "use" and "inherits" relationships:

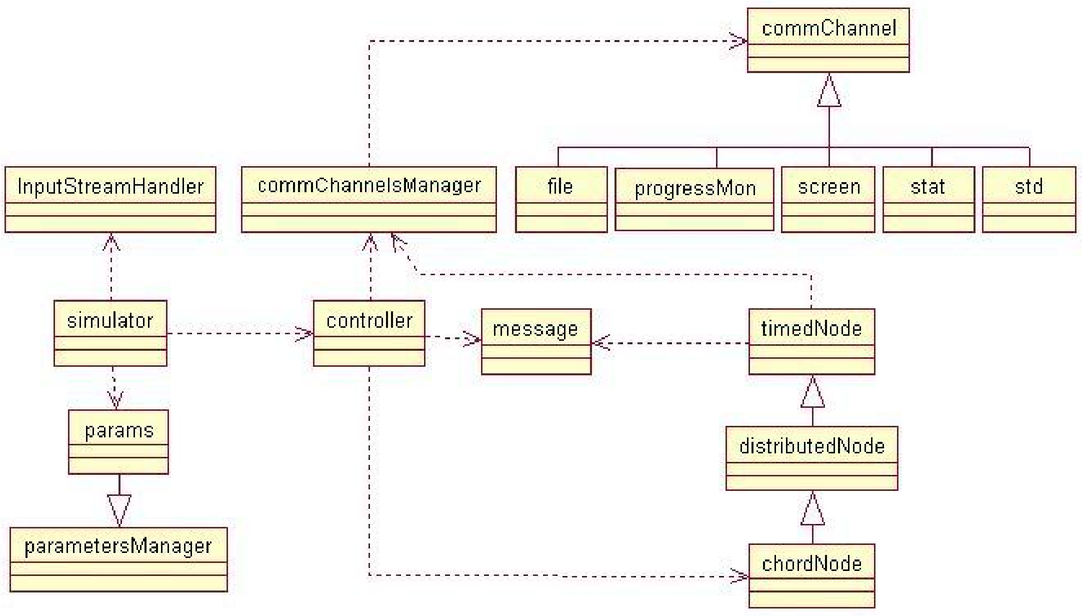


Fig. 3.1 The simulator: UML diagram of classes

What follows is a short description of the most important features of these classes:

- The **simulator** class is the one that holds the *main* routine.
  - Before the simulation starts: Shows on the standard output (*stdout*) the time at which the simulation starts, creates the **params** object, the **controller** object, and initializes the **commChannelsManager**
  - Triggers the start of the simulation on the **controller**
  - After the simulation has finished: Shows on the standard output (*stdout*) the time at which the simulation has finished, and closes the **commChannelsManager** (thus closing each one of the **commChannel** objects that were opened)
- The **params** class is an specialization of the **parametersManager** class. This is an interface that provides easy access to constant like parameters that can be introduced in a XML file. The **simulator** object instantiates one object of class **params** that is then passed to the **controller** as a parameter, and from there it can be used by any of the other components of the program. While **parametersManager** takes care of parsing the XML file and providing a general interface to access the values stored in the XML file, **params** gives more specific type oriented access retrieving methods.
- **InputStreamHandler** is a class downloaded from a website [HAC-49] that I use for *stdin* & *stderr* redirect purposes. When using *Runtime.getRuntime.exec()* to execute an external command (that is the way I use the traffic generator within the simulator, and have them integrated) the input and error streams do not behave cleanly, and this class helps dealing with those streams.
- **commChannelsManager** is the class that initializes and provides clean and transparent access to all **commChannel** objects by means of a common *write* operation. When a **commChannelsManager** object is created, a number of **commChannel** objects is instantiated. The classes that inherit from **commChannel** provide methods to easily log information in various ways. Each one of these channels is created depending on the values found in the XML parameters file previously introduced, accessed through the **params** object. Each one of these communication channels can be declared in the XML file and remain so, but without being really used. The XML entries of type channel have an attribute that can have values on or off. What follows is a brief description of each one of these communication channels. Note that more channels can easily be designed and incorporated in the system; the foundations for that are already laid.
  - **file** communication channels store information into a file whose name can be specified in XML. A user can create as many file **commChannels** as needed for different log purposes, i.e.: log all the messages that nodes send to their predecessor
  - **progressMon** provides the user with a progress bar to supervise the evolution of the simulation, both in 'time ticks' and 'completed percentage' (it does not make sense to have more than one of these in a simulation)
  - **screen** communication channels are pretty much the same as **file** communication channels, with the difference that they are directly shown in a text window as the simulation progresses. When the simulation is complete, the window provides an interface to save its contents into a file
  - **stat** is a channel specifically designed to manage statistical data retrieved during the simulations and generate plots with these data
  - **std** is a channel designed just to let the user set on or off the standard output (*stdout*) for an execution of the simulation. Setting it off or

redirecting it to a certain file makes it easy for the user to execute a number of simulations concurrently without bothering about the verbosity of the simulator.

- **message** is the class that models the messages passed from node to node. A message is basically a command plus a number of parameters. All of them are text based, although they can be later parsed as numeric. These messages can be executed in the nodes upon arrival and depending on the timing of their schedule. This is the implementation of the message passing model that is effective in the simulator instead of the *RPCs* shown in the pseudocode.
- The nodes that form the network whose behaviour is simulated follow this hierarchy:
  - **timedNode** is the topmost node generalization. It takes care of all the timing issues such as ordering the execution of the events of messages that are scheduled for the "now" time moment, and scheduling the execution of messages arrived for a certain future time. One of its inner structures is a queue of events that is filled and emptied as the simulation evolves
  - **distributedNode** inherits from **timedNode** and it satisfies mainly two purposes: it provides the necessary interfaces to pass messages from one node to another and it effectively implements the execution of messages. It also takes care of the necessary methods that simulate the failure of a message delivery (due to the absence of the receiver node in the network, for example)
  - **chordNode** inherits from **distributedNode** and it implements all the characteristics of a real node belonging to the Chord network. It includes the data structures described in the previous chapter as well as the interfaces to execute the commands sent by messages.
- The **controller** is the core class of the simulator. For a start, it gives access to the communication channels and the **params** structure to all the other components of the system that need access to it. For example, the **chordNode** needs access to both of them, and the **commChannelsManager** needs access to **params**. The **controller**'s main features are explained with more depth in the next section.

3.4.2 Internals

All these components that have been mentioned so far fulfill their purpose when they are used as follows:
The first thing the simulator does is loading the XML file that contains the execution parameters through the **params** class.
According to these, a number of communication channels are opened with the **commChannelsManager**.
These two structures are made available to all those components that might need them by means of the core of the application, the **controller** class.
After that, and according to one of the parameters found in **params**, the simulator might generate traffic making an external call to the **traffic generator**. Otherwise, the simulator assumes that there is a file with the instructions as to what traffic is to be used. The name of the file has to be present in **params** too.
The next step is to parse the file produced by the traffic generator (regardless of whether it was just generated or it was done previously). While parsing, the traffic events are inserted into a sorted list. The sorting criteria is the time for which each event has been scheduled by the traffic generator. Given that more than one event can happen at one time tick, the list contains arrays of events.

Once the events have been parsed, a new structure that will contain the nodes is created, the timer is set to zero, and the simulator starts executing its main loop, which looks schematically as follows (Fig. 3.2):

```
time = 0;
while (not end(time)) do
   if(listOfEvents.hasScheduledTrafficNow(time))
      executeEvents(listOfEvents,time);
   endIf
   for (all nodes present in the network) do
      if(node.hasPendingEvents(time))
         executeEvents(node,time);
      endIf
   endFor
   time = time + 1;
endWhile
```

Fig. 3.2 Main loop of the simulator

The events found in the controller's event list are those responsible for creating and destroying nodes, and inserting new documents in the network as well as scheduling the lookup queries. The last event is **exit**, and that marks the last time tick for the loop.
Each one of the nodes has a list of events too, which are modeled by the message structure, and they can be executed by means of an abstraction provided by the programming language.
Given the design previously described, the programmer has the possibility to log certain behaviour to the communication channels.
For example, in the case of this evaluation of Chord, each time a lookup query finds the successor of a certain document, a new entry is written to a chosen channel, storing important information such as the number of hops that the lookup query took to be resolved. This way, it is possible to take these logs later and make statistical studies of them. Another important channels that are used are those that store information about failures, error messages, calls to *stabilization* and *fixFingers* routines, etc.

3.4.3 Extension and customization

The simulator is programmed in such a generic way that it is possible to execute simulations of very different kinds of nodes, not only Chord. So if one wants to experiment the behaviour of networks programmed with different classes of nodes, it is only needed to program the new class of nodes and instruct the simulator about what node class is to be used.
The generality of use of the simulator has been significantly increased by the use of a XML document (and its XSD scheme) that accepts certain number of parameters or program constants to be defined. Examples of these parameters include, among many others:

- the number of bits which limit the identifier space
- the platform on which the program is going to be run (the simulator was tested both on Windows and Linux, and the file paths are platform dependent)
- the paths to files (both input and output)
- the name of the file where the generated traffic is stored
- the fact that the traffic generator should be used or not
- the verbosity of logging
- options that affect the way that graph plots are generated
- etc.

Many other features can be included in this XML file, and tailor the behaviour of the simulator to the needs of the user.
The first thing the **simulator** class does (in the *main* routine) is loading configuration data from the XML file. This file contains certain parameters which the simulator uses in order to initialize the environment for the simulation. For example, as described in previous sections, there is a list of so called “channels”. Each one of these channels is a means to log information for the user.
The events file is generated by an external program, a "traffic generator", described above, too. The paths to access the traffic generator, which are going to be the input and output files, and other options, can also be found in the XML file.

4 Experiments

This chapter describes the design of 4 different sets of input data for the execution of Chord in the simulator. The goal of these 4 different sets of experiments is to analyze the changes of behaviour of a Chord network when there are:

- changes in the network size
- massive simultaneous node failures
- continuous node joins and departures

The input provided to the simulator comes from the traffic generator, when asked to provide the list of events that make the protocol to behave in a fashion that it is interesting for us to study.

4.1 Changes in the network size

This set of experiments is designed to highlight the impact that three chosen variables in the environment of execution of Chord have in its performance.
The main goal of this set of experiments is to focus the attention in how the protocol performs depending on the number of nodes in the network. The performance metric is the **path length** of a lookup request.
After this main goal is met, further experiments are performed related to the **load of requests that a node has to deal with** in function of the size of the network.
First, the **path length** experiments in function of the network size. In addition to this, two further variables have been included to review their influence: whether the *successors list* data structure is used or not for forwarding purposes, and whether the size of identifier space is constant or proportional to the number of nodes in the network.
The results of these experiments will be grouped in subsets, each one of them belonging to the four combinations resulting of these two parameters. Simplifying, the four subsets will be:

- with successors list and constant identifier space
- with successors list and proportional identifier space
- without successors list and constant identifier space
- without successors list and proportional identifier space

Each one of these four subsets involves 12 different runs of Chord with network sizes of **N = 2^k**, with *k* having values in the range 3..14, and requesting 5,000 lookups per experiment. With this, we will be able to study the impact on the network performance when there is an exponential growth in the network size and thus, evaluate the scalability of the protocol.
In order to make it possible to compare the results of the experiments with those shown in the paper by Ion Stoica et al [STO-1], they were designed in a way that was as similar to theirs as possible. The paper defines the experiments that they conducted stating that the network had size **N = 2^k**, and that each node stored, on average, **100 keys**. This follows from the fact that the paper claims that the number of keys stored is **$100 \cdot 2^k$**. For this to be possible, the identifier space in which Chord works will have to fit within these numbers; this means that the identifier space of the network has to be at least **100 times bigger** than the defined network size. The paper does not mention whether all the experiments have the same identifier space, or each one of the identifier spaces is proportional to the number of nodes. That is the reason why the experiments are designed differentiating between constant identifier space or proportional identifier space; it will prove whether this

differentiation has an impact or not on the main performance parameter observed in this research: **the path length of lookup queries**.
Given that the identifier space has to be a **power of 2**, and the number of nodes in the experiments are all powers of 2 too, it follows that the proportion constant that decides the identifier space in the case of proportional growth has to be a **power of 2 as well**. As the number of keys that each node is responsible for in average is 100, the power of 2 that immediately follows is **2^7=128**. Thus, the identifier space when it is proportional to **the network sizes ranges from $2^{7+3}=2^{10}$ to $2^{7+14}=2^{21}$**. When the experiments are with constant identifier space, this is set to this maximum value: **2^{21}**.
As hinted before, the measurements that the experiment will take are the path length of lookup queries, which is the same as saying that the number of nodes that must be visited to resolve a query.
The *successors list* data structure is not expected to prove dramatically useful in absence of node failures or leave operations, because its main reason of existence is robustness. Despite that, the experiments conducted with and without the use of the *successors list* will show what this use provides in terms of performance, when measured by the query path length.
Besides the path length metric, some experiments related to the **load of queries that a node has to deal with** are run. For this set of experiments only one combination of successors list and identifier space is used. As the results on the previous experiments will show, any combination might have been good, and the “with successors list and constant identifier space” is chosen for the next experiments.
These experiments involve 3 subsets of 10 different runs of Chord with network sizes of **$N = 2^k$**, with k having values in the range 3..12, and requesting 10, 20 and 25 lookups per node respectively in each one of the three subsets. With this, we will be able to study the impact on the node processing load when there is an exponential growth in the network size and thus, evaluate the scalability of the protocol. To further clarify this set of experiments: the first subset will have 10 different runs, with network sizes ranging from 8 to 4096 growing in powers of 2, and each node will make 10 lookup requests. The second subset is similar, but 20 lookup requests per node, and the third one will make 25 requests per node.

4.2 Massive simultaneous node failures

This set of experiments is designed to provide data about the impact of different significant amounts of node failures in the performance of the network.

For this measurements, a total amount of 6 experiments were executed; all of them have an initial network size of **N=1000** nodes.

The *successors list* is of size **r=20**, a value that comes from the size of the network as results from taking **$r=2\cdot\log_2 N$**. The reason why this value was chosen is not justified in the original paper [STO-1] , and an explanation was needed based on the available information. What the paper tells is just that "if we use a successor list of length **$r = \Omega(\log_2 N)$** in a network that is initially stable, and then every node fails with probability 1/2, then with high probability *find_successor* returns the closest living successor to the query key". Later on, **$r=2\cdot\log_2 N$** is chosen without further ado.

Now, given that each node fails with probability (1/2), and the probability that the ring breaks is therefore **$p= N\cdot(1/2)^r$**, it follows for the probabilities 1/2, 1/4, 1/8, and any $1/2^x$ that...

$$\begin{aligned}
N\cdot\left(\frac{1}{2}\right)^r &= \frac{1}{2} \\
N\cdot\frac{1}{2^r} &= \frac{1}{2} \\
N &= \frac{1}{2}\cdot 2^r \\
N &= 2^{r-1} \\
\log_2 N &= \log_2 2^{r-1} \\
\log_2 N &= r-1 \\
r &= \log_2 N+1
\end{aligned}
\qquad
\begin{aligned}
N\cdot\left(\frac{1}{2}\right)^r &= \frac{1}{4} \\
N\cdot\frac{1}{2^r} &= \frac{1}{2^2} \\
N &= \frac{1}{2^2}\cdot 2^r \\
N &= 2^{r-2} \\
\log_2 N &= \log_2 2^{r-2} \\
\log_2 N &= r-2 \\
r &= \log_2 N+2
\end{aligned}
\qquad
\begin{aligned}
N\cdot\left(\frac{1}{2}\right)^r &= \frac{1}{8} \\
N\cdot\frac{1}{2^r} &= \frac{1}{2^3} \\
N &= \frac{1}{2^3}\cdot 2^r \\
N &= 2^{r-3} \\
\log_2 N &= \log_2 2^{r-3} \\
\log_2 N &= r-3 \\
r &= \log_2 N+3
\end{aligned}
\qquad
\begin{aligned}
N\cdot\left(\frac{1}{2}\right)^r &= \frac{1}{2^x} \\
N\cdot\frac{1}{2^r} &= \frac{1}{2^x} \\
N &= \frac{1}{2^x}\cdot 2^r \\
N &= 2^{r-x} \quad , \\
\log_2 N &= \log_2 2^{r-x} \\
\log_2 N &= r-x \\
&\boxed{r = \log_2 N+x}
\end{aligned}$$

then if we want to justify **$r = 2\cdot\log_2 N$**, it is because we make the probability of the ring breaking **$p = 1/(2^{\log_2 N})$**, which ultimately ends meaning that bigger networks are meant to be more robust.

Once this successor list issue is clarified, what is left to do is to finish the description of the set of experiments: in each one of the 6 experiments a number of nodes is made to fail (drop from the network) — all at once. This number is a fraction of nodes **from 0% to 50% in intervals of 10%**. Exactly after the nodes fail, and giving the protocol no time to reconfigure the topology of the network (which means that many nodes have indirections in their data structures to nodes that no longer exist in the network) 10,000 lookups are requested at random.
For each lookup we gather, among other useful data:

- **the lookup query path length:** as in the previous set of experiments, this length is a good indicator of the impact that changes have in the behaviour of the network
- **the timeouts experienced by lookup queries:** a timeout occurs when a node tries to contact a failed node. In real life systems, this will be detected by means of a timeout at the transport (TCP) layer in the network stack [TCP-50], when a number of ACKs are not received. In our system this is simulated, and when a node **A** sends a message to an inexistent node **B** (regardless of whether **B** has left, or it has failed), **A** will receive a *failed* message. Upon reception of such a message **A** will react as established by the protocol.
- **whether the lookup got a reply:** lookups can ultimately fail, and the requester may not receive a reply. The success of each lookup query gets logged.

4.3 Constant node joins and departures

The goal of this set of experiments is to evaluate the performance and accuracy of Chord lookups when nodes are continuously joining and leaving the network.
The network has initially 1000 nodes, and the experiment starts taking measures when the network is stabilized. This means that all successors, predecessors and fingers are correctly pointing to the nodes they should, according to the definitions given in Section 2.2.2, Further data structures.
From any moment since the network is stable, nodes start joining, leaving and requesting lookups, and measurements are taken. A total amount of 10,000 lookups is requested.
The paper from which the information of the experiments is extracted [STO-1] states that the lookups are generated according to a Poisson process at a rate of one per second, and that joins and voluntary leaves are modeled by a Poisson process with a mean arrival rate of **R**. A rate **R=0.05** corresponds to one node joining and leaving every 20 seconds on average.
The whole set of experiments comprises runs with arrival and departure rates ranging **from R=0.05 to R=0.40 in steps of 0.05**, resulting in a total number of 8 experiments. Thus, the frequency at which nodes arrive and depart is **f=1/(0.5i)** seconds, with **i ranging from 1 to 8**.
Given that the simulator is programmed in a way that everything is counted in time ticks rather than actual time, a correspondence to these parameters needed to be built up. To simplify things, the experiments were run assuming 100 time ticks is one second. This way it was easier to tune the values for the way traffic is generated, the average message delay, and both *stabilization* and *fixFingers* periodicity.
Again, as in the previous set of experiments, the size of the successors list is **$r=2\cdot\log_2 N$**, and the justification stays the same.
For this set of experiments the same information as the previous set was logged, but a new communication channel was devised so every lookup reply was compared to an external view of the Chord ring to asses the correctness of lookups, given that transient state might provide erroneous replies.

5 Results and Analysis

5.1 Changes in the network size

In this section the results of the experiments that relate to changes in the network size are presented, in four different side-to-side configurations, as well as an overall examination of these results, and a comparison to the results published in the paper [STO-1] by Ion Stoica et al.

5.1.1 With successors list VS Without successors list

This first analysis or results compares the impact of use of the *successors list* structure. The importance of this structure is shown when nodes drop from the network, providing thus robustness. But these experiments show how this *successors list* changes results of the length of lookups regardless of not having any node departures.

In the first case, Fig. 5.1 shows the average, 1st and 99th percentile of path length for lookups in **networks having 2^k nodes, with k ranging from 3 to 14**, and given that **the identifier space is proportional to the number of nodes in the network**.

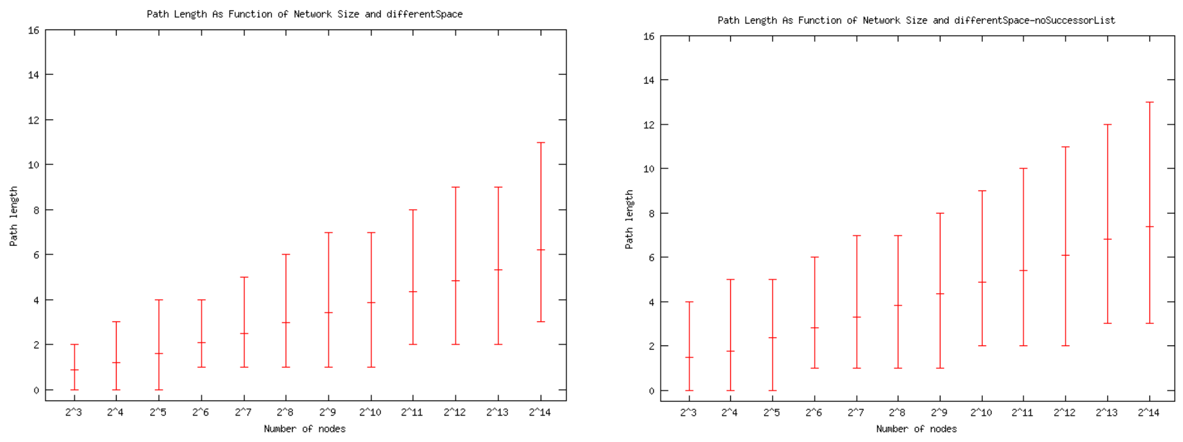


Fig. 5.1 Lookup path length using the *successors list* (left) and not using the *successors list* (right). The identifier space in each of the experiments is proportional to the number of nodes belonging to the network (X axis)

The data shows no dramatic difference, but a slight one. Note that the X axis of the plot grows exponentially, from 2^3 to 2^{14}, which means that the growth shown on the Y axis, linear in appearance, has to be interpreted as logarithmic.

Note that when not using the successors list structure, in Fig. 5.1(right), all the values are slightly higher. This means that the use of the *successors list* structure provides slightly better performance, as expected.

The second comparison in this section shows a similar setup to the last one, but in this case **the identifier space is constant, 2^{21} available keys**. The same range of network sizes: **2^k nodes with k ranging from 3 to 14**:

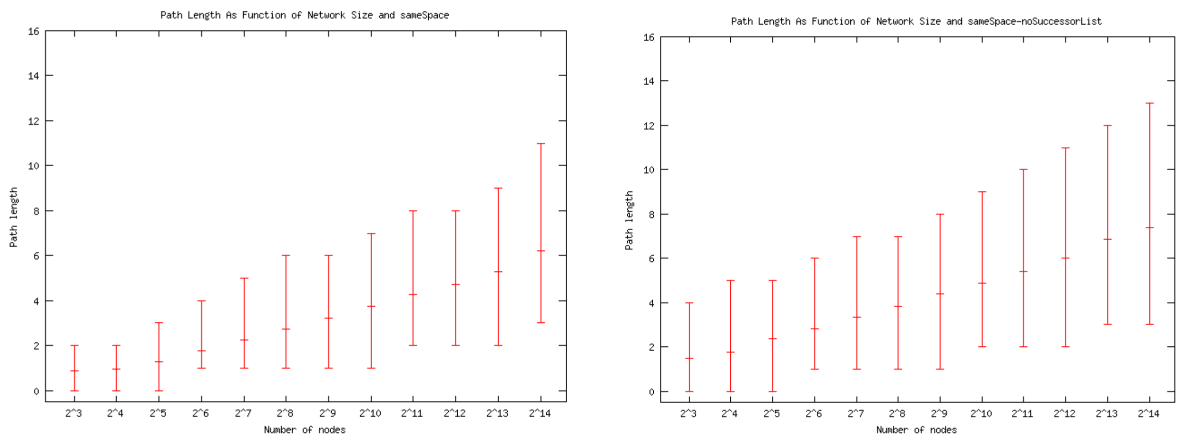


Fig. 5.2 Lookup path length using the *successor list* (left) and not using the *successor list* (right). The identifier space is constant: 2^{21} keys.

As seen before, there is not a big difference between using the *successors list* structure —Fig. 5.2(left)— or not —Fig. 5.2(right)— to improve the path length of lookups, which is the main issue addressed with this set of experiments.
These plots do not really show a dramatic difference. The fact that the X axis that displays the network sizes is in logarithmic scale, and a linear scale is used for the Y axis, is supposed to stress any differences in the Y values. That is why a table (Fig. 5.3) with the actual values, and their differences can illustrate the data in a more exact way:

k	**avg.lgth. (1st,99th perc.) with successors list**	**avg.lgth. (1st, 99th perc.) without successors list**	**difference in avg.lgth.**
3	0.876 (0,2)	1.473 (0,4)	0.597
4	1.182 (0,3)	1.771 (0,5)	0,589
5	1.601 (0,4)	2.382 (0,5)	0,781
6	2.072 (1,4)	2.803 (1,6)	0.731
7	2.491 (1,5)	3.307 (1,7)	0.816
8	2.961 (1,6)	3.822 (1,7)	0.861
9	3.419 (1,7)	4.360 (1,8)	0.941
10	3.876 (1,7)	4.882 (2,9)	1.006
11	4.350 (2,8)	5.391 (2,10)	1.041
12	4.832 (2,9)	6.095 (2,11)	1.263
13	5.317 (2,9)	6.825 (3,12)	1.508
14	6.203 (3,11)	7.382 (3,13)	1.179

Fig. 5.3 Table of values for the average path length for lookups (1st and 99th percentiles too) depending on the size of the network. Last column reflects the differences between using or not using the *successors list*

As can be seen in the table, the difference between the two columns is not constant, but does not grow dramatically either. Regard that k refers to 2^k in the network size.

5.1.2 Constant identifier space VS Proportional identifier space

The goal of this second set of comparisons is to establish how different are the path lengths of lookup requests made in **networks with size 2^k with k ranging from 3 to 14**, but facing the **constant identifier space versus the proportional one**.

Again, two subsets of experiments have been created.

The first one of them, shown in Fig. 5.4, is the one in which the *successors list* is not used.

As shown in the previous section, when the *successors list* is not used, the average path length of lookups is expected to be slightly larger than when it is used. But when facing two sets of experiments in which the *successors list* structure was not used, we can observe that there is absolutely almost no difference. Only the actual numbers given by the simulator differed a bit in some of the experiments, and these are not significant enough to be reflected very much in their graphical representation.

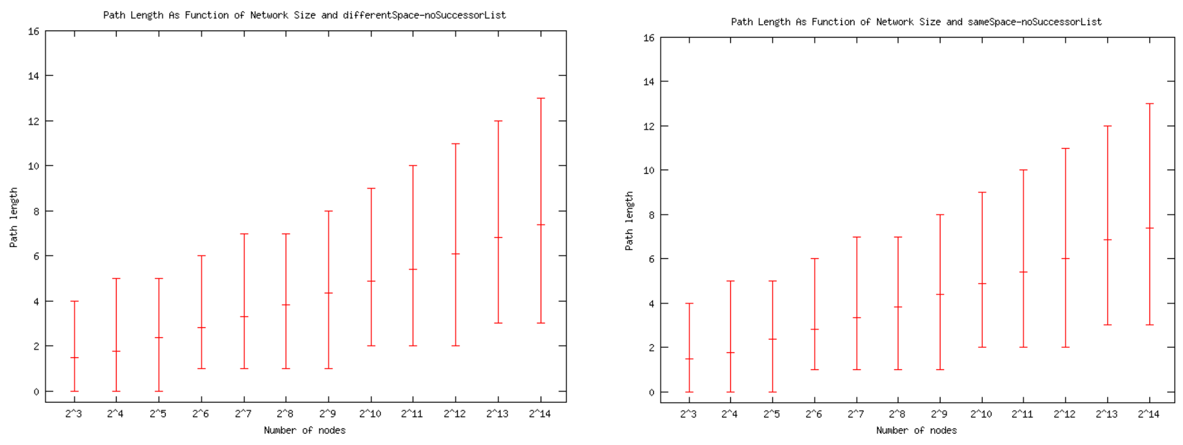


Fig. 5.4 Average path length (including 1st and 99th percentiles) of lookups not using the *successors list*. Proportional identifier space (left) versus constant identifier space (right).

On the other hand, when the experiment was run using the *successors list* structure, which the last section showed that made the performance improve slightly, the results are a little bit different.

If we take a close look to the plots in Fig. 5.5, it can be observed that the only values differing in this experiment are the 99th percentile of the lookup path length in some of the networks tested. This is so because when changing the identifier space, and making it larger, the traffic generated for the experiments changed too. Strangely enough this circumstance did not show up when not using the *successors list* structure.

The 99th percentile of lookup path lengths when they are performed in large identifier spaces fall into a slightly bigger range than when the identifier space is tailored to the size of the network.

Three points need to be remarked about this issue:

- First, that unfortunately it is impossible in this set of experiments to guarantee exactly the same node presence, traffic and lookup requests and responses for the two confronted scenarios. The fact that the identifier space differs has an effect on the way that the network is built and its behaviour.
- Second, that the Y axis is to be interpreted as logarithmic, given that the X axis is exponential, as explained before, which makes this slight difference even less noticeable and insignificant.

- And third, that it is only the 99th percentile that grows, and my interpretation is that it is possibly due to statistical outlier values. 1st percentile and average values are still very much the same. The fact that these percentile values are rounded to the closest integer makes this very slight change more visually noticeable in the plots.

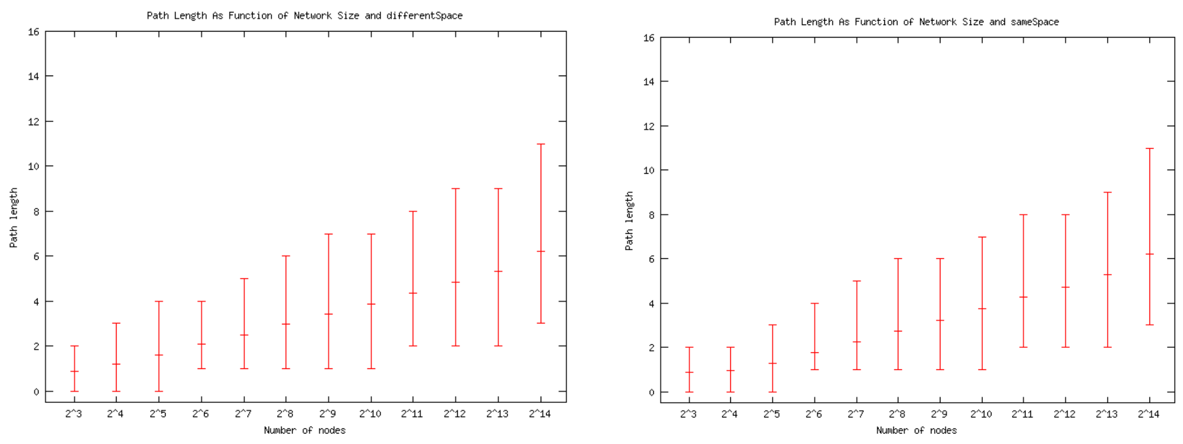


Fig. 5.5 Path length of lookups using the *successors list* with proportional identifier space (left) versus constant identifier space (right).

To further clarify this, another table with values follows:

k	avg.lgth. (1st,99th percentile) proportional identifier space	avg.lgth. (1st,99th percentile) constant identifier space	difference in avg.lgth.
3	0.876 (0,2)	0.876 (0,2)	0.000
4	1.182 (0,3)	0.941 (0,2)	0.241
5	1.601 (0,4)	1.290 (0,3)	0.311
6	2.072 (1,4)	1.751 (1,4)	0.321
7	2.491 (1,5)	2.258 (1,5)	0.233
8	2.961 (1,6)	2.722 (1,6)	0.239
9	3.419 (1,7)	3.240 (1,6)	0.179
10	3.876 (1,7)	3.756 (1,7)	0.120
11	4.350 (2,8)	4.253 (2,8)	0.097
12	4.832 (2,9)	4.722 (2,8)	0.110
13	5.317 (2,9)	5.284 (2,9)	0.033
14	6.203 (3,11)	6.203 (3,11)	0.000

Fig. 5.6 Table of values for the average path length for lookups (1st and 99th percentiles too) depending on the size of the network. Last column reflects the differences between having proportional or constant identifier space

As can be observed, the difference in the average path length is always very low, and does not grow steadily with **k**, which means that both alternatives are good in terms of scalability when performance is measured with regard to the path length of lookup queries.

5.1.3 Overall evaluation of path length metric

It can be considered that the most realistic scenario is that in which the *successors list* structure is used. The experiments in which it was not used were devised precisely to assess how much does the protocol gain when only using the fingers table, which is the structure that really provides most of the performance gains in lookups.
The last graph shows that it is essentially indifferent whether a single Chord instantiation should be made available for use with a huge identifier space or

whether Chord should have a tailored identifier space with regard to the size of the network that is to be used — that is, if we only ponder lookup path length in lookups. It is obvious that if two solutions provide similar results the simpler one is always preferable. That is why the study that should be taken as reference is the one comprising the experiments with the use of the *successors list* and a constant identifier space.
Finally, it is shown in the next lines that the results of these experiments are almost identical to those published by Ion Stoica et al. in page 11 of their paper [STO-1]:

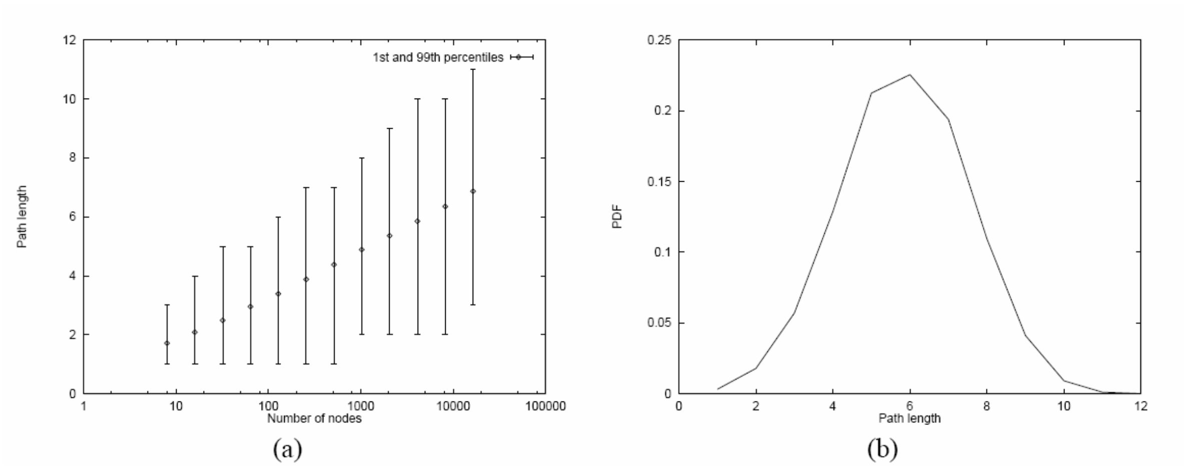


Fig. 5.7 a) Path length as a function of the network size.
b) The PDF of the path length in the case of a 2^{12} node network

Fig. 5.7 is taken from that paper, and from the results of my experiments I produced the two corresponding plots to their respectively equivalent experiments:

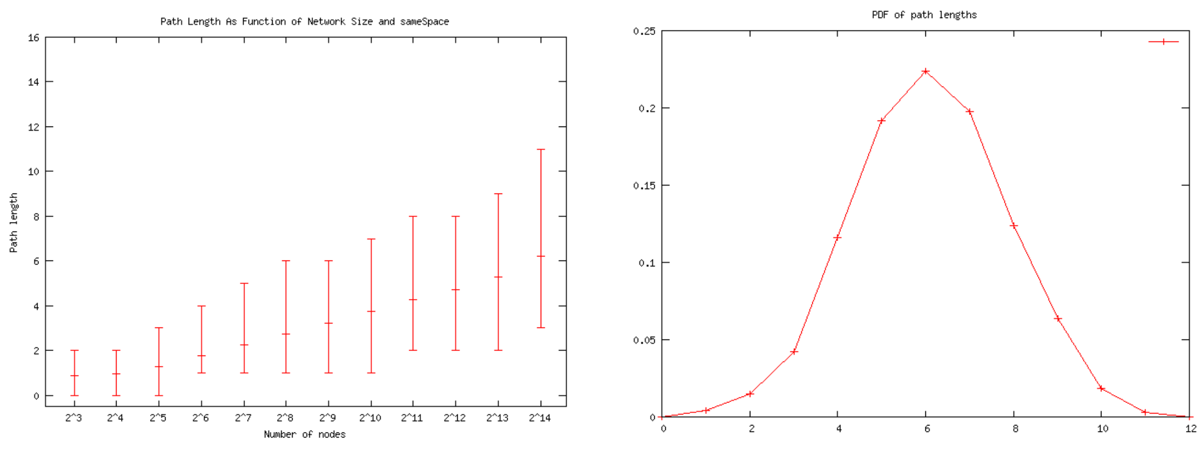
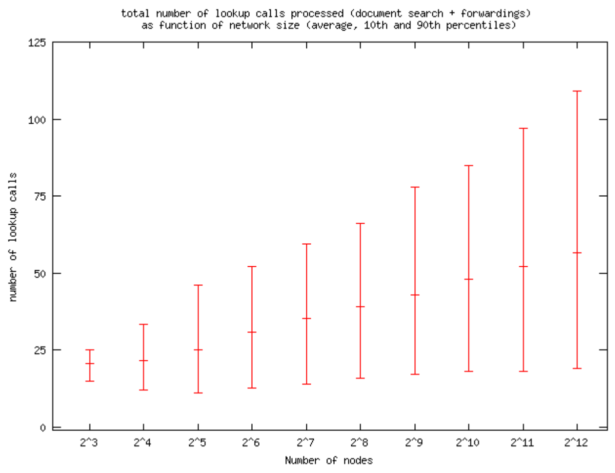


Fig. 5.8 Path length as a function of the network size (left) and
PDF of the path length in the case of a 212 node network (right)

Fig. 5.8(left) shows once again the plot corresponding to the experiments run with the use of the *successors list* and with constant identifier space; Fig. 5.8(right) is a new plot that shows the probability density function (PDF) of the path length for a network with 2^{12} nodes.
Disregarding the difference in which the X axis is scaled, it is evident that the results of the experiments about the evolution of the path length measured with respect to the growth of the size of the network match the ones given by Ion Stoica et al. in their paper [STO-1].

5.1.4 Processing load

This analysis of results shows the impact of the number of nodes in a network into the processing load of a node belonging to the network.
When a node makes a lookup request for document D, the request is forwarded within the network until the node responsible for D is found. This implies that each node has to process not only the lookup requests that it is issuing itself, but those that come from a peer in the network.
Given a set of networks in which the average amount of requests that each node issues is known, the question is: How does the size of the network affect the load that each node has to deal with?
Example: in a network with 10 nodes, and each node making 10 requests, the load expected to be met by any node is not really expected to be the same as the load met by a node in a network with 1,000 nodes, each one making 10 requests too. In the first network there are 100 lookups traveling within; in the second example there are 10,000.
How much more (or less) work has a node to deal with when the network is 100 times bigger?
As explained in the previous chapter, 3 subsets of experiments in which each node in the network issues 10, 20 and 30 lookup requests in average are run.
What they have in common is that each subset involves 10 different runs of networks with sizes $\mathbf{N = 2^k}$ with k ranging from 3 to 12. That means networks with sizes that grow exponentially from 8 to 4096.
What follows is a series of three plots from the three runs. Fig. 5.9 a) represents the plot of data corresponding networks of sizes from 8 to 4096 in which each node has made 10 requests. Next to it, Fig. 5.9 b) has values of path length (P) and the number of lookup calls that each node processes (L). In between these columns there is a calculation of the number of lookups (10 for this first subset of experiments) multiplied by the average path length (P) of queries for each network size. Fig. 5.10 and Fig. 5.11 show the corresponding plots and data to similar experiments but with 20 and 25 document lookups per node respectively.
For each network size the average and the 10th and 90th percentiles of number of lookup calls are shown. Regard that these lookup calls include both the originating call from the client that makes the search for a document as well as the forwarding lookup calls.

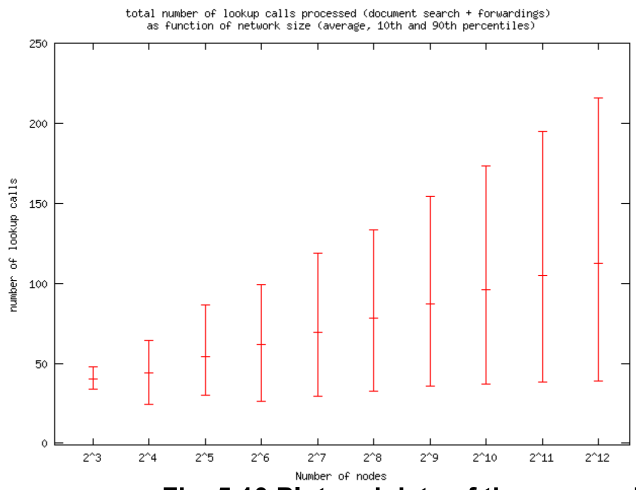


k	$N=2^k$	P	#lookups·P	L_{10}
3	8	1.01	10.1	20.5
4	16	1.21	12.1	21.75
5	32	1.70	17.0	25.0625
6	64	2.08	20.8	30.96875
7	128	2.46	24.6	35.125
8	256	2.92	29.2	39.23438
9	512	3.34	33.4	43.06055
10	1024	3.79	37.9	47.99609
11	2048	4.23	42.3	52.23193
12	4096	4.67	46.7	56.63184

Fig. 5.9 Plot and data of the processing load for networks in which nodes make 10 documents searches in average

a)
Average, 10th percentile and 90th percentile of number of lookup calls when nodes make an average of 10 document searches each.

b)
Table with data of load (L) experienced by nodes in number of lookup calls, and comparison with the multiplication of path length (P) times the number of lookups

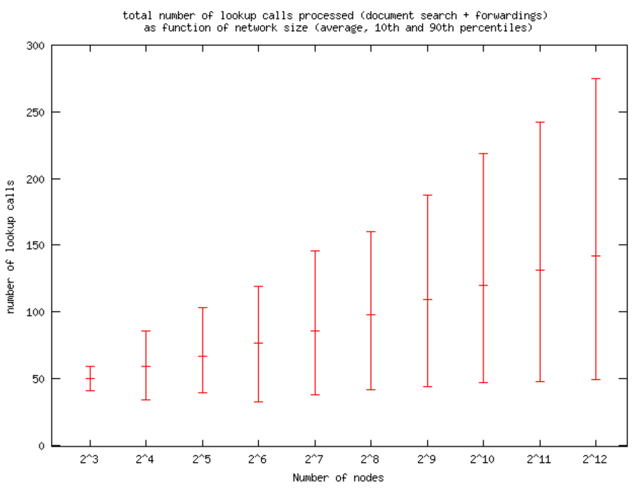


k	$N=2^k$	P	#lookups·P	L_{10}
3	8	1.01	20.2	40.25
4	16	1.21	24.2	44.1875
5	32	1.70	34.0	53.96875
6	64	2.08	41.6	61.70313
7	128	2.46	49.2	69.16406
8	256	2.92	58.4	78.44531
9	512	3.34	66.8	86.96094
10	1024	3.79	75.8	95.81348
11	2048	4.23	84.6	104.6104
12	4096	4.67	93.4	112.5942

Fig. 5.10 Plot and data of the processing load for networks in which nodes make 20 documents searches in average

a)
Average, 10th percentile and 90th percentile of number of lookup calls when nodes make an average of 20 document searches each.

b)
Table with data of load (L) experienced by nodes in number of lookup calls, and comparison with the multiplication of path length (P) times the number of lookups



k	$N=2^k$	P	#lookups·P	L_{10}
3	8	1.01	25.25	49.625
4	16	1.21	30.25	58.6875
5	32	1.70	42.5	66.8125
6	64	2.08	52	76.90625
7	128	2.46	61.5	85.85156
8	256	2.92	73	97.875
9	512	3.34	83.5	108.8477
10	1024	3.79	94.75	119.9619
11	2048	4.23	105.75	130.8804
12	4096	4.67	116.75	141.668

Fig. 5.11 Plot and data of the processing load for networks in which nodes make 25 documents searches in average

a)
Average, 10th percentile and 90th percentile of number of lookup calls when nodes make an average of 25 document searches each.

b)
Table with data of load (L) experienced by nodes in number of lookup calls, and comparison with the multiplication of path length (P) times the number of lookups

These plots give the general idea that when networks grow exponentially, and document search rate is kept more or less constant in a per node basis, the number of calls that each node processes grows linearly. That implies, as with the previous sets of experiments, logarithmic growth of load for each node, which can be considered, in general terms, good.
However, can this load be expressed in terms that let us predict the behaviour of a node in a network of which we know the size and the rate at which nodes make document searches?
The answer is yes, by taking a look at the tables of values that produced the figures above.
Given that:

- P is the average path length of lookups for a given network size
- L is the number of lookup calls that each node was found to make

It does not take too big a deal of guessing to come up with the formula that gives the expected load L by a node:

$$\mathbf{L} = (P+1) \cdot \#lookups$$

That is explained as follows: each node makes its #lookups calls and is also in charge of making P lookup calls for other nodes' sake, as belonging to indirections needed to solve queries.

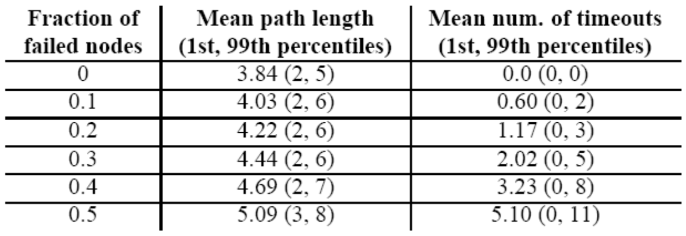
5.2 Massive node failures

This section presents the results of experiments that show how the network reacts in the event of massive node failures. As explained in the previous chapter, the network has initially 1,000 nodes, and for each one of the experiments a fraction of the network is made to fail before requesting 10,000 lookups. The next table in Fig. 5.12 presents the numbers referring to the average path length and the average number of timeouts experienced by a lookup, as well as their 1st and 99th percentiles.

Fraction of failed nodes	Avg. path lgth (1st & 99th perc.)	Avg. num. timeouts (1st & 99th perc.)
0	3.85 (1,8)	0.00 (0,0)
0.1	4.11 (1,9)	0.44 (0,2)
0.2	4.40 (1,10)	0.79 (0,3)
0.3	4.64 (1,11)	1.12 (0,3)
0.4	5.00 (1,12)	1.50 (0,4)
0.5	5.54 (1,13)	2.07 (0,7)

Fig. 5.12 Table of values of average path length and the number of timeouts encountered (including 1st and 99th percentiles) in lookup queries as a function of the fraction of failed nodes

Once again, the results match those produced by Ion Stoica et al. in the paper that this thesis studies [STO-1] with minor differences. What follows in Fig. 5.13 is the table corresponding to those results:



Fraction of failed nodes	Mean path length (1st, 99th percentiles)	Mean num. of timeouts (1st, 99th percentiles)
0	3.84 (2, 5)	0.0 (0, 0)
0.1	4.03 (2, 6)	0.60 (0, 2)
0.2	4.22 (2, 6)	1.17 (0, 3)
0.3	4.44 (2, 6)	2.02 (0, 5)
0.4	4.69 (2, 7)	3.23 (0, 8)
0.5	5.09 (3, 8)	5.10 (0, 11)

Fig. 5.13 Path length and number of timeouts experienced by a lookup as function of nodes that fail simultaneously.

What follows is a more visual analysis of the behaviour of the protocol when massive node failures occur. Fig. 5.14(a) shows the Probability Density Function (PDF) of the lookup path length for the first experiment —when no failures occur at all— which resembles the typical bell shape. Next to it, Fig. 5.14(b) shows the data when 30% of the nodes have failed at once. The shape is mostly the same, but slightly biased to the right, and with the right tail extended. This has to be interpreted as a higher path length for the lookups under this circumstance, and more occurrences of long path lengths than when the network is stable. When it is not stable, there is an amount of nodes whose *fingers* and *successors lists* are not correctly up to date, and that yields retries, which ultimately increases the path length of the queries. This is because there are timeouts when trying to contact a node that does not exist anymore in the network, and a retry is in order.

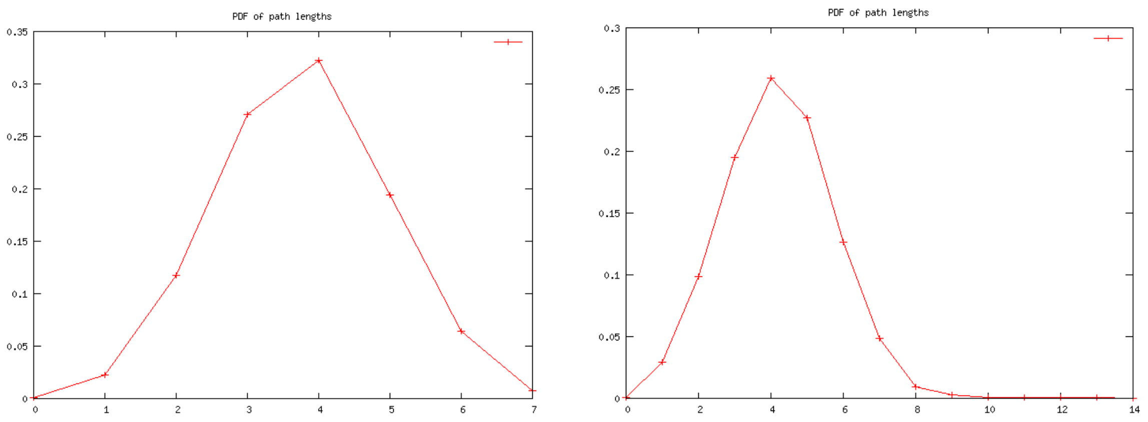


Fig. 5.14 Comparison of the PDF of the lookup path length in a network with 1,000 nodes (left) and the same network when 30% of the nodes have simultaneously failed.

Last, but not least, an important issue to be remarked is the fact that not a single lookup request did *undershoot*. All lookups did eventually get a reply. That is a very outstanding feature of Chord that other P2P systems can not provide along with both the path length performance and the fact that the network does not become flooded with traffic.

5.3 Constant node joins and departures

This section presents the results of the most representative set of experiments within the thesis. They are so because these experiments portray scenarios that are very paradigmatic of the behavior that reality often presents. The network has 1,000 nodes, and that number is maintained more or less stable by the constant arrival and departure of nodes at frequencies of $\mathbf{f=r^{-1}=(0.05i)^{-1}}$ seconds with **i** ranging from 1 to 8. Another way to see it is that the rate **r** ranges from 0.05 to 0.40 in steps of 0.05. In terms of interleaving time between successive events, nodes join and leave one each 20 seconds for **i=1**, 10 seconds for **i=2**, and so on, and 2.5 seconds for **i=8**.

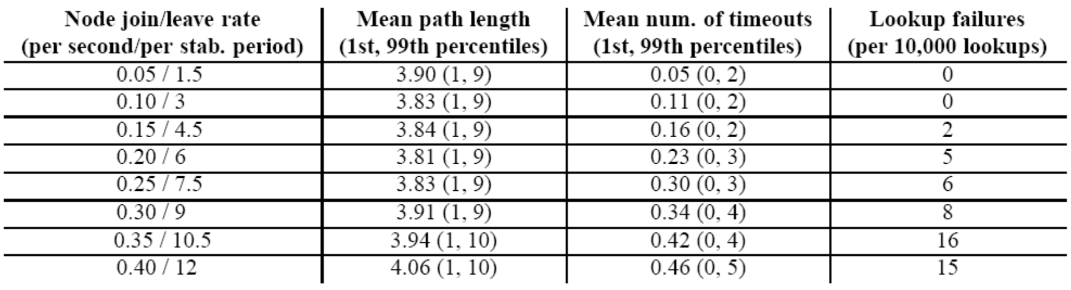
The previous chapter described that what these experiments have in common is that 10,000 lookups are requested at a rate of one per second.

What follows in Fig. 5.15 is a table with the values of average path length and number of timeouts occurred per lookup (and their 1st and 99th percentiles) as well as the total number of lookups failures after the 10,000 lookups that are requested in each experiment and the number of non-resolved lookups.

Node join/leave rate	Avg. path lgth. (1st & 99th perc.)	Avg. number of timeouts (1st & 99th perc.)	Lookup failures (per 10,000 lookups)	Non-resolved lookup queries
0.05	3.88 (1,8)	0.0001 (0,1)	0	0
0.10	3.86 (1,8)	0.0003 (0,1)	0	0
0.15	3.85 (1,8)	0.0004 (0,1)	1	0
0.20	3.84 (1,8)	0.0005 (0,1)	1	0
0.25	3.83 (1,8)	0.0010 (0,1)	1	0
0.30	3.84 (1,8)	0.0006 (0,1)	2	0
0.35	3.87 (1,8)	0.0013 (0,1)	6	1
0.40	3.89 (1,8)	0.0012 (0,1)	3	1

Fig. 5.15 Table with average path length, number of timeouts, failures and undershooting for lookup requests in a network with 1,000 nodes and in function of the arrival/departure rate

As it can be seen by comparing these values to those provided by Ion Stoica et al. in their paper [STO-1] in Fig. 5.16, they do not match completely, but there are significant similarities.



Node join/leave rate (per second/per stab. period)	Mean path length (1st, 99th percentiles)	Mean num. of timeouts (1st, 99th percentiles)	Lookup failures (per 10,000 lookups)
0.05 / 1.5	3.90 (1, 9)	0.05 (0, 2)	0
0.10 / 3	3.83 (1, 9)	0.11 (0, 2)	0
0.15 / 4.5	3.84 (1, 9)	0.16 (0, 2)	2
0.20 / 6	3.81 (1, 9)	0.23 (0, 3)	5
0.25 / 7.5	3.83 (1, 9)	0.30 (0, 3)	6
0.30 / 9	3.91 (1, 9)	0.34 (0, 4)	8
0.35 / 10.5	3.94 (1, 10)	0.42 (0, 4)	16
0.40 / 12	4.06 (1, 10)	0.46 (0, 5)	15

Fig. 5.16 The path length and the number of timeouts experienced by a lookup as function of node join and leave rates.

What follows is a description of certain irregularities in the way that the experiments were laid out. And after that, a description and an analysis of the values of the variables measured during the experiments, as well as a comparison with the values from the table above, and the explanation of these differences.

To start with, it should be noted that the timing conditions are specially significant for this set of experiments —not that they were not in the previous ones, but even

more in this case. At first, it was assumed that one time tick was one second, but this led to some misleading results. This is because the traffic generator does not generate events with fractional average times, and the ratios of events are not exactly what expected. To overcome this problem the parameters that depend on timing were changed to values that match an equivalence of 1 second to 100 ticks. Another reason that backs up this decision is the fact that message delivery times are way smaller than one second, thus making it impossible for the simulator to properly emulate such events. Then again, 100 ticks per second allowed setting the average message delay to 5 time ticks — 50 milliseconds. The main effect this had on the simulator is that each experiment run took much longer, up to durations rounding 5 hours.
Second, the traffic generator does not exactly give an output according to what was expected. The differences are not huge, and certainly not significant enough to invalidate the research, but it adds a source of error that needs to be quantified. The following examples related to the first and second experiments illustrate the case.
In the first experiment, 1,000 nodes are inserted in the network, by feeding the line "**events 1000 100 100 0 0 0 0**" to the traffic generator (see section 3.3 where the traffic generator is explained to get an idea of what this line actually means). In short, this creates 1,000 nodes. So far so good. Then, the simulator is instructed to wait for a certain amount of time that, by experience, we know is enough to make the network stable (that is, all fingers, predecessors, and successors lists are up to date), with the line "**wait 2000**". Now, by inserting the line "**events 11000 100 500 500 0 0 10000**" trouble begins: this instructs the traffic generator to generate a total number of 11,000 events, with an average separation from one to the next of 1 second (100 time ticks). 500 of those 11,000 have to be join operations, 500 have to be leave operations and the rest —10,000— are lookups. All these are randomly mixed up, they do not come in any specific order. The last two lines are "**wait 2000**" and "**exit**", just to give the protocol time to resolve the last queries and exit from the simulation.
What actually happens when the traffic generator is executed is that only 9931 lookups are requested, as well as 569 joins and 500 leaves. As we can see, the leave operation count is correct, and the total amount of events is also correct, but there is a lack of lookups that has been transferred to join operations.
Second experiment: all of the lines fed to the traffic generator are exactly the same except the third, "**events 12000 100 1000 1000 0 0 10000**", which instructs the traffic generator to generate a total number of events of 12,000, of which 1,000 are joins, 1,000 leaves, and the other 10,000 should be lookup queries. This time what actually happens is that 10,010 lookups are requested, 1030 joins and 960 leaves. There is another bias in the way that the traffic generator provides the output!
A detailed inspection to the rest of traffic generations provides evidence that the traffic generator is not accurate enough to produce the results expected. What follows is a table (Fig. 5.17) with the values of expected count of operations and detail of the count of operations found in the output, with the error calculated from these values, for this third set of experiments:

Node join/leave rate	*join* node count (expected), error%	*leave* node count (expected), error%	*lookup* count (expected) – error%
0.05	569 (500), 13.8%	500 (500), 0%	9931 (10000), -0.7%
0.10	1030 (1000), 3.0%	960 (1000), -4.0%	10010 (10000), 0.1%
0.15	1547 (1500), 3.1%	1476 (1500), -1.6%	9977 (10000), -0.2%
0.20	1997 (2000), -0.2%	2049 (2000), 2.4%	9954 (10000), -0.5%
0.25	2528 (2500), 1.1%	2539 (2500), 1.6%	9933 (10000), -0.7%
0.30	3081 (3000), 2.7%	3087 (3000), 2.9%	9832 (10000), -1.7%
0.35	3562 (3500), 1.8%	3595 (3500), 2.7%	9843 (10000), -1.6%
0.40	4070 (4000), 1.8%	4071 (4000), 1.8%	9859 (10000), -1.4%

Fig. 5.17 Table of error rates found on the generation of events by the traffic generator.

In all cases, the total expected amount of events is met, but with a noticeable (although totally affordable) bias to one or another type of event, except in the first experiment, in which the leave count is exactly what expected, but the error bias of join operations is way above 5%. Except for this particular case, the error rates are all more or less not too significant. An inspection to the rest of traffic generations of the thesis shows that this is the only set that incurs into such errors. It is fair to deduce that this is because the only call to the traffic generator that produces errors is that of an "events" with assorted weights, which means calling for different kind of randomly mixed events in a single line.

Another factor that yields inexactness to the way that experiments are performed is the actual choice of weights. For example, the controversial call to the "**events**" command in the last experiment is like "**events 18000 100 4000 4000 0 0 10000**". This means, a total of 18000 events, separated each one of them from the next with 1 second (or 100 time ticks) in average, of which 4000 are joins, 4000 are leaves and the remaining 10,000 are lookups. And this is precisely what the traffic generator does, within a 2% margin of error. But that does not exactly mean that 1 lookup is made every second, nor does this mean that nodes arrive and depart at a rate of one every 2.5 seconds (1/0.4=2.5).

After some calculations a decision was taken concerning this issue, and instead of requesting a constant interleaving time of 100 time ticks, this amount was calculated to approximately yield traffic generations that met the expectations. Some compromises needed to be taken though, given that due to the way the traffic generator is built there is no possibility to really generate traffic complying with the specifications. This is another reason to look at the numbers that are presented in the paper by Ion Stoica et al. [STO-1] with certain degree of skepticism if the traffic that they use for their simulations was generated by this particular traffic generator —which is not such a bad assumption considering that this is their tool too.

So the results of the experiments performed are not really comparable to those found in the paper by Ion Stoica et al. [STO-1], but it is fair to mention that they give a good idea of how the protocol behaves in the event of continuous node arrivals and departures with increasing rates.

Now, regarding the differences found on the values, which cannot be attributable completely to any of the reasons exposed before or any combination of them, a couple of more determinant factors should be stated:

On one hand, the number of timeouts found per lookup shows noticeable better (lower) values in these experiments than those exposed by the paper [STO-1] at Fig. 5.16. More than 3 decimal digits are needed to spot the differences among them. This is because of the use of a data structure that they do not include in their work: the *referrers list*. Each time a node departs voluntarily from the network, it

lets all those nodes in the network that have fingers pointing to it know so, and gives information about the best substitute, which is always its *successor*. That is a notably good improvement to the network stability.
Regarding path length performance, 3 out of 8 simulations show slightly worse figures than those found in the paper. What is worth noting is the fact that the path length does not grow so much in networks with high change rates, and that the values are closer one to each other than those showed by the paper.
The last two columns of the tables correspond to the system's reliability. A lookup failure is defined as the event of receiving a reply that is not correct, meaning that node **X** makes a lookup request for key **K**, and the received reply is that **Y** is **K**'s successor when there is actually another node **Z** that is in the system, and **Z** is responsible for key **K**. Again, this values are slightly better than the ones given by Ion Stoica et al. in their paper, and it is appropriate to assume that the use of the *referrers list* has a saying in this issue too.
Stoica et al. claim in their paper [STO-1] that failures occur due to transient state phenomena. A description of how this lookup queries can be incorrect follows: Suppose that at time t, node n knows both its first and its second successor, s_1 and s_2, both present in n's successor list. A few time units later, t', a new node s joins the network between s_1 and s_2, and that s_1 leaves before n had the chance to discover s. Once n learns that s_1 has left, n will replace it with s_2, the closest successor n knows about. As a result, for any key id$\in(n,s)$, n will return node s_2 instead of s.
However, two issues have to be taken into consideration: first, that these incorrectness is only partial because it has to do with the fact of either s_2 or s having the associated resource to the key being searched, and that is a matter of replication and redundancy that should be taken care of if robustness wants to be combined with correctness. And second, that this is a transient state: the next time n invokes stabilization for s_2, n will learn its correct successor s. It could be argued whether an application developer should include code for retrying the search if the successor has been found but no resource associated to the key was there.
Anyway, further research in the issue showed that not only this case of a node leaving the network displays such behavior; but also nodes joining the network may yield failed lookup replies too.
Regarding the last column, it is worth mentioning that networks with high change rate are prone to lose lookup requests due to the fact that a node can disappear from the network before a message has arrived and subsequently been replied or forwarded. This is something that could easily be corrected into a higher application level by retrying the request, if so was desired. The reason behind these faults is that when a node **X** sends a message to **Y** at time t, this message takes a while to arrive. If **Y** is present in the network the message is by all means delivered from the point of view of **X**, but it could very well happen that **Y** disappears from the network before the message arrives (or the procedure call is executed). That results in the message being lost, and it ultimately yields non resolved lookup queries. Fortunately enough, this only happens in networks with very high amount of joins and departures.

6 Future Work

The area of Peer-to-Peer computing is boiling with activity, and there is a massive amount of areas from which to choose to research about.
From the most general point of view, and focusing into the work that I have carried out, I would say that further work could be done regarding:

- Research about new uses that can take advantage of P2P architecture's capabilities, not just file sharing. Maybe sharing CPU power, memory, or other resources in general could be investigated.
- The DSL group at SICS has been working in DKS (distributed k-ary search), a framework for P2P systems that provides advantages such as those found in Chord, but in such a general way that the performance gain is increased by the use of further data structures that provide costs of O($\log_k$N) instead of just O($\log_2$N). The complexity of the algorithms and the data structures required to do so is bigger than the one found in Chord, but the principles are similar, and the gains in the face of network growth and change are enormous.
- Now, focusing in the work presented in this thesis, I would encourage improving the behavior of the traffic generator. Even a total redesign (and implementation) and a more general and configurable approach could well pay off.
- The simulator itself can be improved too, by:
  - including a GUI, so the interaction with the user is done by means of menus rather than text files
  - using object interfaces instead of parsed text messages. This is not too difficult due to the fact that all the code is strictly modular and very well commented and documented (javadoc is available).
  - improve the memory consumption of the text windows, by devising a way in which the content of a window does not need to be continuously kept in memory (perhaps by disabling scrolling back?)
- The code implementing the Chord node can be extended in many ways. To start with, I strongly suggest to focus on key insertion and acquisition, which are the main purposes of resource sharing. Replication and redundancy are natural extensions of said enhancements to the code, and should be taken care of too.
- And finally, a broad variety of other experiments can be devised to further extend the knowledge of Chord and its behaviour.

7 Conclusions

What follows is a summary of the conclusions drawn after all the work. First I present some ideas about the simulator and its use. And finally a recapitulation of the main conclusions about Chord's behaviour and some of its main advantages such as scalability, robustness, simplicity, performance and accuracy follow.

7.1 The simulator

The simulator is a highly customizable easy-to-use tool that can be improved by further programming, or just be used the way it is now. The XML configuration interface is clean and easy to manage, as well as easy to extend too. Simulations of networks can easily be run once how it works is understood, and there is no actual need to go deep in the understanding of the tool to increase its capabilities.
The simulator is well documented and there is a very detailed javadoc style documentation as well as a simple UML diagram explaining the relationships between entities. Its verbosity can be tailored to the needs of the person that works with it depending on the stage of work that they are going through, and the style of experimentation that is being performed. Batch processes can easily be launched, and all output channels can be redirected to files. It is sound, robust and easy to understand and extend or modify. It is a very valuable tool for network simulation and emulation.
Its resource requirements depend on the use that it is given. The only time that I found memory limitations was when running simulations that created huge logs on AWT text windows, due to the fact that the buffers are not flushed automatically and for some reason, each time a method call is instantiated upon an AWT window object, the whole object, with its contents (maybe megabytes of text) is brought into memory.
That is about the only limitation that I have found to the use of the simulator, and I ended up using files instead of text windows for logs that were big in nature.

7.2 Chord

In general terms, it can be said that the results of the experiments run match those found in Ion Stoica's et al. paper [STO-1], the only exception being those of the third set. These could not be compared anyway because of a certain number of field conditions, a part of which were beyond the aim of this thesis to adapt, and the others would have not yielded more satisfying results at all.
However, this last set of experiments is the one that I consider most relevant. Most P2P applications nowadays are focused on non corporative file sharing. In this set of applications, it is usual to see networks with thousands of nodes in which nodes continuously arrive and depart, which is the scenario that this third set of experiments pictured. The fact that Chord requires that a node responsible for a key is required to maintain that key may lead to disregard this protocol for such uses. Despite of that, the scenario is still very significant, and gives a very good idea of **how smoothly and accurately Chord behaves in the face of network changes**.

The first set of experiments has shown how well the protocol behaves no matter how big the network is. When there is an exponential growth of the number of nodes belonging to the network we observe just linear growth of the lookup path length. That is translated as logarithmic growth of cost against linear growth of the network, and in plain terms this is just **very good scalability**. Regarding the load of lookups that each node has to deal with, it might seem too ideal to think about a network setup in which all nodes provide the same load (meaning how much work is delivered to the network to perform). But in essence, the idea is that the work is more or less evenly balanced, regardless of the fact that some nodes might be loading the network more than others. What counts for this calculation is how many lookup queries are done in the network, and how many nodes are there to share the load.
And last, but not least, the second set of experiments has shown the **robustness and accuracy** of Chord, by not failing to reply a single one of the 10,000 requests despite of the fact that up to 50% of the nodes in the network would fail. Note that this is so because lookups start being requested when nodes have stopped failing. But the fact is that while the network is being reconfigured, the lookups continue their course along the Chord ring and yield results after all.
If the scenario was that nodes continue falling from the network after the lookup request has been delivered, it could happen as with the third set of experiments under the hardest conditions: some requests could very well be lost.
Anyway, a final summary of Chord's strengths based on the data provided by the previous study and the simulations state that the protocol is **simple, robust, reliable** and **scalable**.

8 Appendix

8.1 Glossary

Peer-to-peer (P2P)

A P2P computer network is one that relies on computing power at the edges (ends) of a connection rather than in the network itself. P2P networks are used for sharing content like audio, video, data or anything in digital format. P2P network can also mean grid computing. A pure peer-to-peer network does not have the notion of clients or server, but only equal peer nodes that simultaneously function as both “clients” and “servers” to the other nodes on the network. This model of network arrangement differs from the client-server model where communication is usually to and from a central server.
Source: Wikipedia [WIK-51]

Often referred to simply as peer-to-peer, or abbreviated P2P, a type of network in which each workstation has equivalent capabilities and responsibilities. This differs from client/server architectures, in which some computers are dedicated to serving the others. Peer-to-peer networks are generally simpler, but they usually do not offer the same performance under heavy loads.
Source: Webopedia [WEB-52]

Scalability

The ability of a software program to continue to function smoothly as additional volume, or work is required of it.
Source: LSoft.com [LSO-53]

The capacity of a system to increase performance under an increased load when resources (typically hardware) are added.
Source: Wikipedia [WIK-54]

The capacity of a network to keep pace with changes and growth.
Source: Cisco [CIS-55]

Fully Distributed System

Distributed computing or Distributed system is the process of aggregating the power of several computing entities to collaboratively run a single computational task in a transparent and coherent way, so that they appear as a single, centralized system.
Source: Wikipedia [WIK-56]

Distributed Hash Tables

Distributed hash tables (DHTs) are a class of decentralized, distributed systems and algorithms being developed to provide a scalable, self-configuring infrastructure with a clean programming interface. This infrastructure can then be used to support more complex services. DHTs can be used to store data, as well as route and disseminate information. DHTs are named after hash tables because they assign responsibility for a piece of data based on a hash function (often SHA-1); each node acts like a bucket in a hash table. A DHT provides an efficient lookup algorithm (or network routing method) that allows one participating node to quickly determine which other machine is responsible for a given piece of data.
Source: Wikipedia [WIK-57]

Modulo

In computing, the modulo operation finds the remainder of division of one number by another. When given two numbers, *a* and *n*, *a modulo n* is the remainder, *r*, on division of *a by n*. Although typically performed with *a* and *n* both being integers, many computing systems allow other types of numeric operands.
Source: Wikipedia [WIK-58]

When integers are taken "modulo m", one neglects multiples of m and considers only the remainder. Therefore, 17 (modulo 5) = 2 because 17 = 5 x 3 + 2.
Source: The Mathematics Lair [MAT-59]

An arithmetic scheme in which the result is the remainder after division.
Source: Uniform code council [UCC-60]

Remote Procedure Calls (RPC)

A remote procedure call (RPC) is a protocol that allows a computer program running on one host to cause code to be executed on another host without the programmer needing to explicitly code for this. When the code in question is written using object-oriented principles, RPC is sometimes referred to as remote invocation or remote method invocation.
Source: Wikipedia [WIK-61]

Message passing model

Message passing is a style of parallel programming where instead of using shared memory and locks or other facilities of mutual exclusion, different threads of execution communicate via passing messages.
Source: Wikipedia [WIK-62]

A method of communication between processes that involves one process sending data and the other process receiving the data, via explicit send and receive calls.
Source: National Academies Press [NAP-63]

8.2 Simulator User Manual

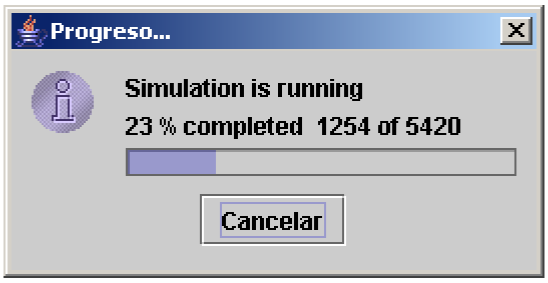
The simulator is a piece of software designed to aid in the task of studying peer to peer systems.
It has been programmed in Java and this document includes both an UML diagram in Fig. 3.1 as well as the whole Javadoc documentation in the next section of this chapter.
What follows is a short description of its requirements, some hints about how to tune up the behaviour of the simulator through the configuration XML file, and how to start it to make it work.

8.2.1 System Requirements

The simulator requires, of course, a functioning Java Virtual Machine.
Log files might need disk space, and how much depends in how much verbosity the user requires from a run of the simulator.
A typical simulator install would have a structure like this:

```
/some_path_to/simulator/                    (installdir)
                        /classes            (classdir)
                        /doc                (docdir)
                        /src                (srcdir)
```

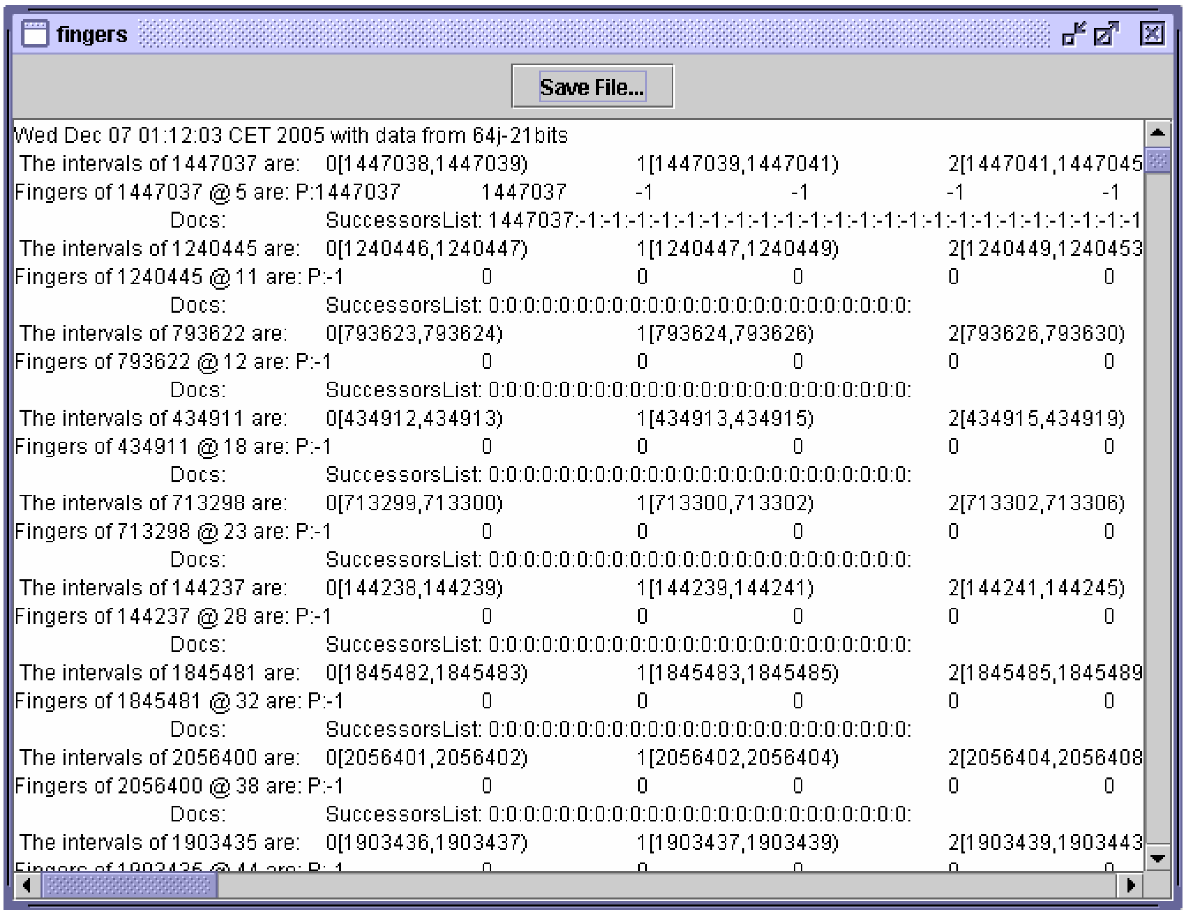
The documentation and source directories are, of course, not needed for the simulator to work.
The fact that the package is named “chord” might lead to the misconception that the simulator has been designed with only Chord in mind.
The truth is, it can be used to simulate the behaviour of various peer-to-peer networks. The next subsection includes a short description of what needs to be done in order to use the simulator to study other networks.
The installation of the simulator needs a XML configuration file called “simulator.xml” and its required schema and document type definition, which are “simulator.xsd” and “simulator.dtd” to be stored all in installdir
The simulator requires JDom and Xerces libraries in order to be able to parse the configuration XML file, and they have to be available either in the $CLASSPATH environment variable of the Operating System, or to feed their locations into the command line when calling the Java Virtual Machine to load the program.
One of the options available in the XML configuration file is a communication channel of type “progressMon”. This needs a graphical environment to be shown (X-windows or a MS-Windows desktop). If the simulator is run in text mode, it is better to set this option off. Fig. 8.1 shows a screenshot of the progress monitor with information of what percentage of the simulation has been completed as well as a detail of how many tick times out of the total have been already simulated.



Simulation is running
23 % completed 1254 of 5420

Fig. 8.1 Progress monitor of the simulator while running

The log files can either be shown as they are generated in a window (that requires a graphic environment too) or they can also be directly written to disk. If the graphic version is chosen, the window provides buttons to save the file if wanted. If the file is not saved, the information is lost. Fig. 8.2 shows a screenshot of a text window log file with some information:



```
Wed Dec 07 01:12:03 CET 2005 with data from 64j-21bits
 The intervals of 1447037 are:   0[1447038,1447039)          1[1447039,1447041)          2[1447041,1447045
Fingers of 1447037 @ 5 are: P:1447037          1447037      -1           -1           -1           -1
                 Docs:           SuccessorsList: 1447037:-1:-1:-1:-1:-1:-1:-1:-1:-1:-1:-1:-1:-1:-1:-1:-1:-1:-1:-1:-1:-1
 The intervals of 1240445 are:   0[1240446,1240447)          1[1240447,1240449)          2[1240449,1240453
Fingers of 1240445 @ 11 are: P:-1              0            0            0            0            0
                 Docs:           SuccessorsList: 0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:
 The intervals of 793622 are:    0[793623,793624)            1[793624,793626)            2[793626,793630)
Fingers of 793622 @ 12 are: P:-1               0            0            0            0            0
                 Docs:           SuccessorsList: 0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:
 The intervals of 434911 are:    0[434912,434913)            1[434913,434915)            2[434915,434919)
Fingers of 434911 @ 18 are: P:-1               0            0            0            0            0
                 Docs:           SuccessorsList: 0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:
 The intervals of 713298 are:    0[713299,713300)            1[713300,713302)            2[713302,713306)
Fingers of 713298 @ 23 are: P:-1               0            0            0            0            0
                 Docs:           SuccessorsList: 0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:
 The intervals of 144237 are:    0[144238,144239)            1[144239,144241)            2[144241,144245)
Fingers of 144237 @ 28 are: P:-1               0            0            0            0            0
                 Docs:           SuccessorsList: 0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:
 The intervals of 1845481 are:   0[1845482,1845483)          1[1845483,1845485)          2[1845485,1845489
Fingers of 1845481 @ 32 are: P:-1              0            0            0            0            0
                 Docs:           SuccessorsList: 0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:
 The intervals of 2056400 are:   0[2056401,2056402)          1[2056402,2056404)          2[2056404,2056408
Fingers of 2056400 @ 38 are: P:-1              0            0            0            0            0
                 Docs:           SuccessorsList: 0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:
 The intervals of 1903435 are:   0[1903436,1903437)          1[1903437,1903439)          2[1903439,1903443
Fingers of 1903435 @ 44 are: P:-1              0            0            0            0            0
```

Fig. 8.2 Example of a text window with information of a channel of type "screen"

The way the simulator is implemented right now, it makes a call to the Traffic Generator (see section 3.3 of this document). So the Traffic Generator must be present at its expected location. It is not difficult to include a new entry in the configuration file to feed the simulator with the proper information about what traffic generator to use, if necessary.
One of the classes in the package, "*stat.class*", provides statistics and tries to make a plot with them if GNU-Plot is found in the system. This can also be easily customizable with the configuration file, and make it possible for the plot to be drawn or not depending on an additional option.
So, in short, in order to get the simulator running the way it is now, it is needed:

- disk space (installation + log files)
- Java Virtual Machine
- XML parsing libraries
- XML, DTD and XSD files
- Graphic environment
- Traffic Simulator
- GNU-Plot

Regard that the three last are necessary to run the simulator with its current setup. Other options might make them unnecessary.

8.2.2 Configuration

The XML configuration file has three sections:

- <constants> - in which numeric and text constants are kept
- <files> - where file paths and file names are stored
- <channels> - where the verbosity of the simulator logs is controlled

All entries in the XML file, regardless of which section they belong to, have three compulsory fields:

- NAME - is the name of the entry, it must be unique
- TYPE - is either integer or text (if integer, the parser returns an Integer object)
- VALUE - is, obviously, the value that the variable holds

Some examples of the information stored there are:

<constant name="AverageMessageDelay" type="integer" value="5"/>
instructs the simulator that message delivery from one node to another should take 5 time units in average
<constant name="useSuccessorsListUpgrade" type="integer" value="1"/>
means that the successors list should be looked into when looking for a good candidate to forward a message
<constant name="generateEventsFile" type="integer" value="0"/>
tells the simulator NOT to generate the events file with the traffic generator (the traffic might have previously been generated)

<file name="WinXP-eventsFilePath" type="text" value="z:\\exp\\arrival\\"/>
this shows the path where to find the events file generated by the traffic generator (or where to store it, if the generator has to run too) in a Windows environment
<file name="Linux-eventsFilePath" type="text" value="/etc/sim/exp/arrival/"/>
the same, for a Linux environment

<channel name="stderr" type="std" value="on"/>
sets the standard error channel on
<channel name="stdout" type="std" value="off"/>
sets the standard output channel off
<channel name="time" type="progressMon" value="on"/>
sets the progress monitor on (like in Fig. 8.1)
<channel name="commTrace" type="file" value="off"/>
makes that the information logged as “commTrace”, which would be written to a file if the channel was ON, is disregarded (no file is created at all)
<channel name="fingers" type="file" value="on"/>
makes that the information logged as “fingers” will be written to a file
<channel name="documents" type="screen" value="on"/>
makes that the information logged as “documents” will appear in a text window (as shown in Fig. 8.2)

8.2.3 Starting the simulator

The simulator has its main method in the “*simulator.class*“ file, so it has to be started with a call to the Java Virtual Machine like follows:

```
/path_to_jdk/bin/java chord.simulator $OPTIONS
```

$OPTIONS must include, if these paths are not in the $CLASSPATH environment variable of the Operating System:

```
-classpath    /path_to/simulator/classes;
              /path_to/jdom.jar;
              /path_to/xercesImpl.jar;
              /path_to/xmlParserAPIs.jar
```

8.3 Javadoc from the Simulator

Package chord		
Class Summary		***Page***
arrivals	Title: stat Description: the module that accumulates statistical data for the arrivals experiment Copyright: Copyright (c) 2005 Samer Al-Kassimi Company:	*58*
chordNode	Title: chordNode Description: This is the main character of the project.	*61*
commChannel	Title: commChannel Description: abstraction of a Communication Channel, it's a way of logging information in different ways Copyright: Copyright (c) 2005 Samer Al-Kassimi Company:	*69*
commChannelsManager	Title: commChannelsManager Description: Is the part of the system that takes care of showing the proper information in the selected communication channels as desired by command of the system's general parameters file.	*70*
controller	Title: Controller Description: controller of the network simulator Copyright: Copyright (c) 2005 Samer Al-Kassimi Company:	*71*
distributedNode	Title: distributedNode Description: A subclass of a timedNode, it offers the interface of a distributed node, with operations concerning the sending of `messages` (including forwarding) and execution of commands the code of which has to be coded into the implementing class.	*72*
file	Title:file Description: this provides the functionality to easily log information into files through `commChannels` Copyright: Copyright (c) 2005 Company:	*74*
InputStreamHandler	Title: Input Stream Handler Description: Prevents the improper use of standard input, output and error channels on a getRuntime() call.	*75*
message	Title: message Description: Calls sent among nodes, they're meant to be executed upon reception This is a way of implementing a RPC abstraction, through messages.	*76*
parametersManager	Title: parametersManager Description: manages constant values Copyright: Copyright (c) 2005 Samer Al-Kassimi Company:	*78*
params		*79*

progressMon	Title: progressMon Description: progress monitor, a bar showing the percentage of the simulation that has so far been completed, it's a subclass of `commChannel` Copyright: Copyright (c) 2005 Samer Al-Kassimi Company:	*80*
screen	Title: screen Description: Copyright: Copyright (c) 2005 Company:	*81*
simulator	Title: Simulator Description: network simulator, container of the main routine Copyright: Copyright (c) 2005 Samer Al-Kassimi Company:	*82*
stat	Title: stat Description: the module that accumulates statistical data and plots it Copyright: Copyright (c) 2005 Samer Al-Kassimi Company:	*83*
std	Title:std Description: this provides the functionality to easily log information out to STDOUT and STDERR channels, in accordance to the rest of the program, making it easy to activate and deactivate this level of information display Copyright: Copyright (c) 2005 Samer Al-Kassimi Company:	*86*
timedNode	Title: timedNode Description: Basic node of a network, provides a queue of events to be executed in timely fashion, and the interface by the means of which those events will eventually be executed Copyright: Copyright (c) 2005 Samer Al-Kassimi Company:	*87*

Class arrivals

chord

```
java.lang.Object
  └ chord.commChannel
      — chord.arrivals
```

public class **arrivals**
extends commChannel
Title: stat
Description: the module that accumulates statistical data for the arrivals experiment
Copyright: Copyright (c) 2005 Samer Al-Kassimi
Company:

Author:
Samer Al-Kassimi

Version:
1.0

See Also:
commChannel, params

Field Summary		**Page**
`protected String`	**`baseFName`** The Strings containing the names of the files	*59*
`protected String`	**`dataFName`** The Strings containing the names of the files	*59*

protected FileOutputStream	**dataFOS** The streams that are used to gather the data necessary to generate the plot	*59*
protected Hashtable	**forwards** The table in which the number of lookups of SEARCH calls and its forwarding steps are stored	*60*
protected float	**forwardsAccum**	*60*
protected Hashtable	**lookups** The table in which the number of lookups of SEARCH calls and its forwarding steps are stored	*60*
protected float	**lookupsAccum**	*60*
protected int[]	**nodeIds**	*60*
protected float	**nodesAccum**	*60*
protected float[]	**sForwards**	*60*
protected float[]	**sLookups**	*60*
protected String	**summary** The Strings containing the names of the files	*60*
protected String	**summaryFName** The Strings containing the names of the files	*59*
protected FileOutputStream	**summaryFOS** The streams that are used to gather the data necessary to generate the plot	*59*

Fields inherited from class chord.commChannel
howMany, name, p

Constructor Summary	***Page***
arrivals(String name, params p) This is the constructor of the class, it is initialized with a name and a `params` object.	*60*

Method Summary		***Page***
void	**close**() Implementation of the abstract method defined in the superclass.	*60*
protected String	**format**(float value)	*60*
protected String	**format2**(float value)	*61*
void	**genDataArrays**()	*61*
void	**generateDataFiles**()	*61*
String	**genSummary**()	*61*
protected void	**initializeFileNames**()	*60*
protected void	**initializeFiles**()	*60*
void	**showStats**()	*61*
void	**write**(String what) Implementation of the abstract method defined in the superclass.	*60*

Methods inherited from class chord.commChannel
close, write

Field Detail

dataFOS

`protected FileOutputStream` **`dataFOS`**

The streams that are used to gather the data necessary to generate the plot

summaryFOS

`protected FileOutputStream` **`summaryFOS`**

The streams that are used to gather the data necessary to generate the plot

baseFName

`protected String` **`baseFName`**

The Strings containing the names of the files

dataFName

`protected String` **`dataFName`**

The Strings containing the names of the files

summaryFName

`protected String` **`summaryFName`**

The Strings containing the names of the files

---

summary

`protected String **summary**`

The Strings containing the names of the files

---

lookups

`protected Hashtable **lookups**`

The table in which the number of lookups of SEARCH calls and its forwarding steps are stored

---

forwards

`protected Hashtable **forwards**`

The table in which the number of lookups of SEARCH calls and its forwarding steps are stored

---

sLookups

`protected float[] **sLookups**`

---

sForwards

`protected float[] **sForwards**`

---

nodeIds

`protected int[] **nodeIds**`

---

lookupsAccum

`protected float **lookupsAccum**`

---

forwardsAccum

`protected float **forwardsAccum**`

---

nodesAccum

`protected float **nodesAccum**`

Constructor Detail

arrivals

```
public arrivals(String name,
                params p)
```

This is the constructor of the class, it is initialized with a name and a `params` object. After initialization (by calling the superclass constructor), it obtains the names of the files that are going to be used and initializes the files so the necessary data can be read or write, as necessary.

Parameters:

`name` - String - the name of the gnuPlot instance (the channel name)

`p` - params - access to the constants

Method Detail

initializeFileNames

`protected void **initializeFileNames**()`

---

initializeFiles

`protected void **initializeFiles**()`

---

write

`public void **write**(String what)`

Implementation of the abstract method defined in the superclass. It updates the statistical data refered by the contents of the parameter

Overrides:

write in class commChannel

Parameters:

`what` - String - the statistical data that needs to be updated

---

close

`public void **close**()`

Implementation of the abstract method defined in the superclass. It closes the communication channel. Before closing, the statistical data is prepared to be inserted into data files. After generating these files, they will be feeded, along with a script, to the gnuPlot program. The script file instructs gnuPlot to generate GIF images containing the plots.

Overrides:

close in class commChannel

---

format

`protected String **format**(float value)`

---

format2

```
protected String format2(float value)
```

showStats

```
public void showStats()
```

genSummary

```
public String genSummary()
```

genDataArrays

```
public void genDataArrays()
                   throws Exception
```

generateDataFiles

```
public void generateDataFiles()
```

Class chordNode

chord

```
java.lang.Object
  └ chord.timedNode
      ─ chord.distributedNode
          └ chord.chordNode
```

public class **chordNode**
extends distributedNode
Title: chordNode
Description: This is the main character of the project. The `chordNode` represents one node belonging to a Chord network.
It provides a series of operations that make it possible the network to achieve its desired topology, as well as maintaining it.
Furthermore, it provides the operations to fulfill the main purpose of such a network, which is to share resources and make them available to the rest of the network.
The main operation in which this project focus is the search of a certain key. Keys represent both node identifiers and shared resources.
In a general approach, we can say that a shared resource is a "document". And so it is possible for nodes to insert documents in the network to be shared, and also to make requests and see if a certain document is present in the network.
The network has a certain maximum size given by two parameters that can be adjusted in the XML file that holds the execution parameters and constants. One of the parameters is named K and in the Chord case is a fixed number: 2. The other parameter is called "number of bits". This name is due to the fact that a key belonging to such a network (K,number of bits) would be belonging to the range [0,(2^number of bits)-1].
Thus, the maximum size of a network is maxNetSize = K^bits.
Other important issues regarding the inner functioning of the `chordNode` are the fact that there are two routines in charge of the maintenance of the network topology. These are the `fixFingers()` and the `stabilization()` routines, and they are executed in a periodical fashion.
The basic `chordNode` is able to form a network and maintain its structure as long as nodes don't fail and drop from the network.
A more advanced `chordNode` implementation takes into consideration the eventuality of a failure, and makes it possible for the network to recover from such failures. This is achieved by the insertion of an extra data structure called "successors list". This "successors list" can provide more advantages to the network more than just robustness. It is possible to make the network slightly faster in terms of number of messages needed to receive a reply for a search request.
This is done by choosing the best of the "pointers" to the next hope from the two structures that hold information about nodes present in the network
which are the "fingers table" and the "successors list".
The use of this advantage is eligible before running a simulation of the network by stating it so in the XML file that holds the execution parameters and constants. A constant called "useSuccessorsListUpgrade" with value "0" (False) or "1" (True) will disable or enable this improvement for the search requests. The "successor list" data structure will be used anyway for robustness purposes.

Copyright: Copyright (c) 2005 Samer Al-Kassimi

Author:

samer

Version:

2.0

See Also:

distributedNode, timedNode, commChannelsManager, params, message

Field Summary		*Page*
`protected int`	**`bits`** given that a network has a maximum size of maxNetSize, this is defined as follows: maxNetSize = K^bits both K and bits are values that are taken from the PARAMS constant management, although we always work with K=2	*64*
`protected Hashtable`	**`documents`** storage of the documents inserted in the node	*64*
`protected int[]`	**`end`** superfluous information, end of the interval of responsibility of a certain level of finger	*64*
`protected int[]`	**`fingers`** fingers.get(K) is the first node on the ring such as succeeds ((n + 2^(k-1)) mod (2^m)) being 1 K m and K is a `Double` object	*64*
`boolean`	**`joined`** the existence of the object in the system doesn't necessarily mean that the node has actually joined the network one could say that a node X has joined when there's at least another node Y in the network that know about X this happens when a first stabilization of X has been done	*64*
`protected int`	**`K`**	*64*
`protected int`	**`maxNetSize`**	*64*
`protected int`	**`next`** counter used to perform the fix_fingers routine, pointing to the level that is going to be updated	*65*
`protected int`	**`periodicFixFinger`** time between periodic calls to the fixFinger routine	*64*
`protected int`	**`periodicStabilization`** time between periodic calls to the stabilization routine	*64*
`protected int`	**`predecessor`** identifier of the predecessor node in the identifier circle	*65*
`protected LinkedList`	**`referers`** list for the nodes present in the network in which this node is present at the finger table	*64*
`protected int[]`	**`start`** superfluous information, start of the interval of responsibility of a certain level of finger	*64*
`protected int[]`	**`successorsList`** list of the `#bits` successors	*64*

Fields inherited from class chord.distributedNode
`myClass, myMethods, network, p`

Fields inherited from class chord.timedNode
`commChannels, id, now, queue`

Constructor Summary	*Page*
`chordNode``(int nodeId, int time, Hashtable network, commChannelsManager channels, params p)` Creator operation, initializes a node with its parameters, as described below:	*65*

Method Summary		Page
void	**alone**() When a node is left alone in the network, the pointers to the successor and predecessors need to be reset, as well as it is needed to flush the event queue and thus, re-schedule the periodic calls	67
void	**areYouAlive**(String whoAsks) A node receiving a "areYouAlive" request may have to reply "YES" in real live, but the simulator has a means to simulate the failure from higher layers, sending a "failed" command to the requesting node	67
protected boolean	**belongs**(int candidate, int init, int end, String boundaries)	68
void	**checkPredecessor**() This is how nodes learn that a predecessor has failed, and allow "notify" to reset them If the message fails to be sent, the "failed" routine will take care of it	67
protected int	**closest**(int target, int candidate1, int candidate2)	69
protected int	**closestPreceedingNode**(int whose) Choose the best fit candidate among the nodes that I know about to forward a request.	68
protected void	**correctFingersAndList**(String failedNode) Removes the entries from the successorList and the fingerTable that are a node that is not in the network anymore	68
void	**disconnect**() In order to guarantee the bootstrapping to build feasible networks, this procedure ensures that nodes joining the system find a node within the network	67
void	**failed**(String sender, String receiver, String command, String colonSeparatedParamList) reception of messages is implemented in the superclass as an event scheduled	68
void	**find**(String what)	66
void	**findSuccessor**(String originalSender, String whose, String callerType, String length, String timeouts) This is the main procedure of the protocol.	65
void	**findSuccessorReply**(String whose, String successor, String callerType, String length) I receive the reply of a request that I made in the past.	66
void	**fixFingers**() This is the maintenance routine that keeps the finger table up to date, by updating one row each turn.	68
void	**giveMeMyDocuments**(String whoAsked)	66
void	**giveMeMyDocumentsReply**(String docList)	66
void	**hereIsMySuccessorsList**(String sList, String howManyLeft)	68
void	**iKnowYouExist**(String whoDoes)	67
protected String	**initStaticInfo**() creation of the static information regarding each level of fingers (interval included)	65
void	**insert**(String document)	67
void	**insertDoc**(String document)	67
protected void	**join**(int nexus) Once a node from the network has been selected to be joined to, procedures for the JOIN operation continue.	65
void	**leave**() do whatever a node does to leave nicely the network: • - let the successor and the predecessor know that I'm leaving.	66
void	**myInfo**(String time)	67
protected int	**nexusSelector**() At any time, there's one node in the network marked to be the one contacted for the JOIN operation.	65
void	**notifyPredecessor**(String predecessorCandidate)	67
void	**nTellsItsPredecessorItLeaves**(String myNewSuccessorList) I am the predecessor of a node who has left.	67
void	**nTellsItsRefererItLeaves**(String whoSays, String substitute) I have a finger that leaves, and he tells me so.	67

`void`	`nTellsItsSuccessorItLeaves(String myNewPredecessor, String listOfDocuments)` I am the successor of a node who has left.	*66*
`void`	`preJoin()` bootstrapping: call to select a node from the network, and join to that node	*65*
`void`	`stabilize()` This is how nodes learn about newly joined nodes in the network.	*67*
`void`	`tellMeYourPredecessor(String whoAsks)`	*67*
`void`	`tellMeYourPredecessorReply(String predecessor)`	*67*
`void`	`tellMeYourSuccessorsList(String whoAsks)`	*68*
`void`	`youAreInMyFingersNow(String whoSays)`	*68*
`void`	`youAreNotInMyFingersAnymore(String whoSays)`	*68*

Methods inherited from class chord.distributedNode
`execute, failed, send`

Methods inherited from class chord.timedNode
`execute, scheduleEvent, trigger`

Field Detail

joined

`public boolean joined`

the existence of the object in the system doesn't necessarily mean that the node has actually joined the network one could say that a node X has joined when there's at least another node Y in the network that know about X this happens when a first stabilization of X has been done

bits

`protected int bits`

given that a network has a maximum size of maxNetSize, this is defined as follows: maxNetSize = K^bits both K and bits are values that are taken from the PARAMS constant management, although we always work with K=2

K

`protected int K`

maxNetSize

`protected int maxNetSize`

periodicFixFinger

`protected int periodicFixFinger`

time between periodic calls to the fixFinger routine

periodicStabilization

`protected int periodicStabilization`

time between periodic calls to the stabilization routine

fingers

`protected int[] fingers`

fingers.get(K) is the first node on the ring such as succeeds ((n + 2^(k-1)) mod (2^m)) being 1 K m and K is a `Double` object

successorsList

`protected int[] successorsList`

list of the `#bits` successors

start

`protected int[] start`

superfluous information, start of the interval of responsibility of a certain level of finger

end

`protected int[] end`

superfluous information, end of the interval of responsibility of a certain level of finger

referers

`protected LinkedList referers`

list for the nodes present in the network in which this node is present at the finger table

documents

`protected Hashtable documents`

storage of the documents inserted in the node

predecessor

```
protected int predecessor
```

identifier of the predecessor node in the identifier circle

---

next

```
protected int next
```

counter used to perform the fix_fingers routine, pointing to the level that is going to be updated

Constructor Detail

chordNode

```
public chordNode(int nodeId,
                 int time,
                 Hashtable network,
                 commChannelsManager channels,
                 params p)
```

Creator operation, initializes a node with its parameters, as described below:

Parameters:

`nodeId` - int - the node identification number
`time` - int - time of initialization (entry point) in the network
`network` - Hashtable - the network the node belongs to
`channels` - commChannelsManager - the log manager that takes care of the simulator's debug level
`p` - params - access to the constants

Method Detail

initStaticInfo

```
protected String initStaticInfo()
```

creation of the static information regarding each level of fingers (interval included)

Returns:

String - the static information in printable format

---

preJoin

```
public void preJoin()
```

bootstrapping: call to select a node from the network, and join to that node

---

nexusSelector

```
protected int nexusSelector()
```

At any time, there's one node in the network marked to be the one contacted for the JOIN operation. When the network is empty, the first node to enter marks itself as candidate to be contacted for JOIN operations. Each time a node disappears from the network, this mark is checked out just in case the "token" has to be passed to someone else.

Returns:

int

See Also:

disconnect()

---

join

```
protected void join(int nexus)
```

Once a node from the network has been selected to be joined to, procedures for the JOIN operation continue. If THIS is the only node in the network yet, this operation initializes `fingers` and `successorsList` as well as triggers the `stabilize()` and `fixFingers()` routines. Otherwise, a message to the selected network node is sent asking "who's my successor, I want to join", and these initializations will be done later as a consequence of that call, when the reply comes.

Parameters:

`nexus` - int - the node to which THIS joins

---

findSuccessor

```
public void findSuccessor(String originalSender,
                          String whose,
                          String callerType,
                          String length,
                          String timeouts)
```

This is the main procedure of the protocol. It searches for the successor of a certain `ID`.

Parameters:

`originalSender` - String - The one that expects the reply (if so)
`whose` - String - The element whose successor wants to be found

`callerType` - String - What kind of operation is being performed that needs to know a successor, one of the following:

- JOIN: a node joining the network made the request
- INSERT: a node has requested the insertion of a certain key into the network, the responsible of the key should take care of it
- DOCUMENT: a node has requested the search of a certain key (or resource, or document)
- FIXFINGERS: a node has requested the successor of a certain node to update its fingers table

`length` - String - How many hops are being used to find the successor. It's incremented at each forwarding step
`timeouts` - String - How many timeouts have occurred during the resolution of this findSuccessor call since its original call

---

findSuccessorReply

```
public void findSuccessorReply(String whose,
                               String successor,
                               String callerType,
                               String length)
```

I receive the reply of a request that I made in the past. Upon reception of the reply, the operation that requested the search continues, as it now has the data that it requires.

Parameters:

`whose` - String - what ID was I searching for
`successor` - String - the real content of the reply: WHO is the successor of the key requested
`callerType` - String - What kind of operation is being performed that needs to know a successor, one of the following:

- JOIN: a node joining the network made the request. The node knows now who is its successor, and the operation continues: it requests its documents and schedules the first call to the maintenance routines.
- DOCUMENT: this node has requested the search of a certain key (or resource, or document). The search has been successful, the result is written in the log files
- FIXFINGERS: this node has requested the successor of a certain ID to update the fingers table. The successor is known now, the finger table is updated

`length` - String - How many hops were used to find the successor. It's been incremented at each forwarding step.

---

giveMeMyDocuments

```
public void giveMeMyDocuments(String whoAsked)
```

Parameters:

`whoAsked` - String

---

giveMeMyDocumentsReply

```
public void giveMeMyDocumentsReply(String docList)
```

Parameters:

`docList` - String

---

find

```
public void find(String what)
```

Parameters:

`what` - String

---

leave

```
public void leave()
```

do whatever a node does to leave nicely the network:

- - let the successor and the predecessor know that I'm leaving. As a result:
- --- transfer the documents to the successor
- --- the successor needs to replace it's predecessor
- --- my predecessor will need to remove ME from its successor list, and add MY last successor from the list
- - everything else will go further as if it was a failure

---

nTellsItsSuccessorItLeaves

```
public void nTellsItsSuccessorItLeaves(String myNewPredecessor,
                                       String listOfDocuments)
```

I am the successor of a node who has left. I receive a new predecessor and a ":" separated list of documents

Parameters:

`myNewPredecessor` - String
`listOfDocuments` - String

---

nTellsItsPredecessorItLeaves

```
public void nTellsItsPredecessorItLeaves(String myNewSuccessorList)
```

I am the predecessor of a node who has left. I receive a new successorsList (which means a new successor, too)

Parameters:

`myNewSuccessorList` - String - the new successorsList

---

nTellsItsRefererItLeaves

```
public void nTellsItsRefererItLeaves(String whoSays,
                                     String substitute)
```

I have a finger that leaves, and he tells me so. I should update this finger ASAP

Parameters:

`whoSays` - String - the node that has knows I refer to it
`substitute` - String - the successor of that node

---

alone

```
public void alone()
```

When a node is left alone in the network, the pointers to the successor and predecessors need to be reset, as well as it is needed to flush the event queue and thus, re-schedule the periodic calls

---

disconnect

```
public void disconnect()
```

In order to guarantee the bootstrapping to build feasible networks, this procedure ensures that nodes joining the system find a node within the network

---

insert

```
public void insert(String document)
```

Parameters:

`document` - String

---

insertDoc

```
public void insertDoc(String document)
```

---

myInfo

```
public void myInfo(String time)
```

Parameters:

`time` - String

---

checkPredecessor

```
public void checkPredecessor()
```

This is how nodes learn that a predecessor has failed, and allow "notify" to reset them If the message fails to be sent, the "failed" routine will take care of it

---

areYouAlive

```
public void areYouAlive(String whoAsks)
```

A node receiving a "areYouAlive" request may have to reply "YES" in real live, but the simulator has a means to simulate the failure from higher layers, sending a "failed" command to the requesting node

Parameters:

`whoAsks` - String

---

stabilize

```
public void stabilize()
```

This is how nodes learn about newly joined nodes in the network.

---

tellMeYourPredecessor

```
public void tellMeYourPredecessor(String whoAsks)
```

Parameters:

`whoAsks` - String

---

tellMeYourPredecessorReply

```
public void tellMeYourPredecessorReply(String predecessor)
```

Parameters:

`predecessor` - String

---

notifyPredecessor

```
public void notifyPredecessor(String predecessorCandidate)
```

Parameters:

`predecessorCandidate` - String

---

iKnowYouExist

```
public void iKnowYouExist(String whoDoes)
```

Parameters:

`whoDoes` - String

---

tellMeYourSuccessorsList

```
public void tellMeYourSuccessorsList(String whoAsks)
```

Parameters:

`whoAsks` - String

---

hereIsMySuccessorsList

```
public void hereIsMySuccessorsList(String sList,
                                   String howManyLeft)
```

Parameters:

`sList` - String

`howManyLeft` - String

---

fixFingers

```
public void fixFingers()
```

This is the maintenance routine that keeps the finger table up to date, by updating one row each turn.
The "periodicFixFinger" parameter in the XML parameter file tells how often this routine will be executed.
The whole finger table will be updated each "periodicFixFinger"*bits units of time

---

youAreInMyFingersNow

```
public void youAreInMyFingersNow(String whoSays)
```

Parameters:

`whoSays` - String

---

youAreNotInMyFingersAnymore

```
public void youAreNotInMyFingersAnymore(String whoSays)
```

Parameters:

`whoSays` - String

---

closestPreceedingNode

```
protected int closestPreceedingNode(int whose)
```

Choose the best fit candidate among the nodes that I know about to forward a request.
In the case that the "useSuccessorsListUpgrade" parameter is set to 1 (True) in the XML parameters file, the successors list will be taken into consideration for a better approximation to the closest node.

Parameters:

`whose` - int - the ID of which we want to find the closest node

Returns:

int - the ID of the node of which I know that is closest to the key I'm searching for

---

failed

```
public void failed(String sender,
                   String receiver,
                   String command,
                   String colonSeparatedParamList)
```

reception of messages is implemented in the superclass as an event scheduled

Overrides:

failed in class distributedNode

Parameters:

`sender` - String

`receiver` - String

`command` - String

`colonSeparatedParamList` - String

---

correctFingersAndList

```
protected void correctFingersAndList(String failedNode)
```

Removes the entries from the successorList and the fingerTable that are a node that is not in the network anymore

Parameters:

`failedNode` - String

---

belongs

```
protected boolean belongs(int candidate,
                          int init,
                          int end,
                          String boundaries)
```

Parameters:

`candidate` - double What element is being tested as to belong to the interval

`init` - double Initial boundary of the interval

`end` - double End boundary of the interval
`boundaries` - String Whether the start and end intervals are open or close

Returns:

boolean

---

closest

```
protected int closest(int target,
                      int candidate1,
                      int candidate2)
```

Class commChannel

chord

```
java.lang.Object
  └ chord.commChannel
```

Direct Known Subclasses:

arrivals, file, progressMon, screen, stat, std

---

abstract public class **commChannel**
extends Object
Title: commChannel
Description: abstraction of a Communication Channel, it's a way of logging information in different ways
Copyright: Copyright (c) 2005 Samer Al-Kassimi
Company:

Author:

samer

Version:

3.0

Field Summary		**Page**
protected static int	**howMany** This is a counter that stores the amount of instances of `commChannel` objects there are, for display issues Only those `commChannel` subclasses that are susceptible to be shown in a window may want to access this counter	*69*
protected String	**name** The name of the channel, it will be the address to access, store or display information.	*69*
protected params	**p** Access to the constants	*69*

Constructor Summary	**Page**
commChannel(String name, params p) Constructor of the class.	*70*

Method Summary		**Page**
abstract void	**close**() This is the signature of the call that any subclass must implement for the event of closing the `commChannel`.	*70*
abstract void	**write**(String what) This is the signature of the call that any subclass must implement for the event of logging a message.	*70*

Field Detail

howMany

`protected static int howMany`

This is a counter that stores the amount of instances of `commChannel` objects there are, for display issues Only those `commChannel` subclasses that are susceptible to be shown in a window may want to access this counter

---

name

`protected String name`

The name of the channel, it will be the address to access, store or display information.

---

p

`protected params p`

Access to the constants

Constructor Detail

commChannel

```
public commChannel(String name,
                   params p)
```

Constructor of the class. It initializes the access to the constants as well as the name of the object

Parameters:

`name` - String

`p` - params

Method Detail

close

```
public abstract void close()
```

This is the signature of the call that any subclass must implement for the event of closing the `commChannel`.

write

```
public abstract void write(String what)
```

This is the signature of the call that any subclass must implement for the event of logging a message.

Parameters:

`what` - String - the data that is going to be logged.

Class commChannelsManager

chord

```
java.lang.Object
  └ chord.commChannelsManager
```

public class **commChannelsManager**
extends Object

Title: commChannelsManager

Description: Is the part of the system that takes care of showing the proper information in the selected communication channels as desired by command of the system's general parameters file. It acts as a log debug level manager. By providing the information in the PARAMS file this class will show (or not) information in the desired channels, according to the debug level desired.

Copyright: Copyright (c) 2005 Samer Al-Kassimi

Author:

samer

Version:

2.0

See Also:

params, commChannel

Field Summary		*Page*
protected Hashtable	`ch` Storage of the different channels.	70

Constructor Summary	*Page*
`commChannelsManager(params p)` This is the constructor method: it takes the debug level data from the PARAMS file It creates the channels where the log information will be logged to.	71

Method Summary		*Page*
void	`closeAll()` This method closes all channels opened by the manager.	71
void	`write(String where, String what)` This primitive offers the user a way to log information.	71
void	`writeLine(String where, String what)` This primitive offers the user a mean to log information.	71

Field Detail

ch

```
protected Hashtable ch
```

Storage of the different channels. The keys are `String` objects that represent the name of the channel. The object stored for that key is a `commChannel`. the `execute` method.

Constructor Detail

commChannelsManager

```
public commChannelsManager(params p)
```

This is the constructor method: it takes the debug level data from the PARAMS file It creates the channels where the log information will be logged to.

Parameters:

`p` - params - access to the constants

Method Detail

write

```
public void write(String where,
                  String what)
```

This primitive offers the user a way to log information. It will check out the existence of the channel requested, and if it is active, the information will be written there.

Parameters:

`where` - String - in which channel the information will be written
`what` - String - what information is going to be logged

writeLine

```
public void writeLine(String where,
                      String what)
```

This primitive offers the user a mean to log information. It will check out the existence of the channel requested, and if it is active, the information will be written there. In this case, a line return character will be written at the end of the line.

Parameters:

`where` - String - in which channel the information will be written
`what` - String - what information is going to be logged

closeAll

```
public void closeAll()
```

This method closes all channels opened by the manager.

Class controller

chord

```
java.lang.Object
  └ chord.controller
```

public class **controller**
extends Object
Title: Controller
Description: controller of the network simulator
Copyright: Copyright (c) 2005 Samer Al-Kassimi
Company:
Author:
samer
Version:
2.0

Field Summary		Page
SortedSet	**net**	72

Constructor Summary	Page
controller(commChannelsManager channelsManager, params parameters) Default initialization function	72

Method Summary		Page
void	**executeSimulation**(int exitTime) Executes the simulation.	72
int	**parseEventsFileAndGenerateQueue**() Takes the file specified as an event file from the configuration parameters, and generates the queue of events, inserting multiple entries in the format of `event(parameter[, parameter])`	72

Field Detail

net

`public SortedSet net`

Constructor Detail

controller

```
public controller(commChannelsManager channelsManager,
                  params parameters)
```

Default initialization function

Parameters:

`channelsManager` - commChannelsManager

`parameters` - params

Method Detail

parseEventsFileAndGenerateQueue

`public int parseEventsFileAndGenerateQueue()`

Takes the file specified as an event file from the configuration parameters, and generates the queue of events, inserting multiple entries in the format of `event(parameter[, parameter])`

Returns:

Double the finish time of the simulation. It's assumed that the last event found in the events file is an exit event, and a time for that is provided

executeSimulation

`public void executeSimulation(int exitTime)`

Executes the simulation. For each time tick, it tries to see if there are events pending to execute at that time. If there are, it executes them. Order of execution of events within a given time is not important, therefore a `Vector` of events is provided for each set of events to execute at that given time. Execution ends at the exitTime provided to the function call or at the maxExecutionTime provided in the parameter file, whichever occurs first.

Parameters:

`exitTime` – double

Class distributedNode

chord

```
java.lang.Object
  └ chord.timedNode
      — chord.distributedNode
```

Direct Known Subclasses:

chordNode

abstract public class **distributedNode**
extends timedNode
Title: distributedNode
Description: A subclass of a timedNode, it offers the interface of a distributed node, with operations concerning the sending of `messages` (including forwarding) and execution of commands the code of which has to be coded into the implementing class.
Copyright: Copyright (c) 2005 Samer Al-Kassimi

Author:

samer

Version:

2.0

See Also:

timedNode, message, params, commChannel

Field Summary		*Page*
`protected Class`	**myClass** This is a reference to the class object in order to make it possible for the `java.lang.reflect` package to execute methods called through code>message objects	73
`protected Hashtable`	**myMethods** This is a storage of the available methods to be executed in the implementing node.	73
`Hashtable`	**network** The network that the node belongs to.	73

protected `params`	`p` The parameters object that takes data from the XML configuration file	73

Fields inherited from class chord.timedNode
`commChannels, id, now, queue`

Constructor Summary	**Page**
`distributedNode(int nodeId, int time, Hashtable network, commChannelsManager channels, params p)` Creator operation, initializes a node with its parameters, as described below:	73

Method Summary		**Page**
`void`	`execute(message m)` This is one of the two important operations that this class offers, the execution of a message arrived to the node.	74
`abstract void`	`failed(String sender, String receiver, String command, String colonSeparatedParamList)` This is the signature of the call that any subclass must implement for the event of a failed message.	74
`void`	`send(message msg)` This is the second core functionality of this class.	74

Methods inherited from class chord.timedNode
`execute, scheduleEvent, trigger`

Field Detail

network

`public Hashtable network`

The network that the node belongs to. The keys are `Integer` objects. The `int` value stored in those objects represents a node identificator, and the object stored for that key is a `distributedNode`. This object is accessed locally only upon creation of a new `distributedNode`, and when a `message` is to be delivered

myClass

`protected Class myClass`

This is a reference to the class object in order to make it possible for the `java.lang.reflect` package to execute methods called through code>message objects

myMethods

`protected Hashtable myMethods`

This is a storage of the available methods to be executed in the implementing node. The keys are `String` objects, describing the name of the method. The object stored for that key is a `Method` that, among other things, can be referenced in order to make a call to the `invoke` method, that will actually execute the method referenced by the name.

p

`protected params p`

The parameters object that takes data from the XML configuration file

Constructor Detail

distributedNode

```
public distributedNode(int nodeId,
                       int time,
                       Hashtable network,
                       commChannelsManager channels,
                       params p)
```

Creator operation, initializes a node with its parameters, as described below:

Parameters:

`nodeId` - int - the node identification number
`time` - int - the time of initialization (entry point) in the network
`network` - Hashtable - the network the node belongs to
`channels` - commChannelsManager - the log manager that takes care of the simulator's debug level
`p` - params - the configuration file manager

Method Detail

execute

```
public void execute(message m)
```

This is one of the two important operations that this class offers, the execution of a message arrived to the node. It provides most of the implementation to the translation of RPC to message sending and execution.
Tries to execute the content of an arrived message, if an available operation with the same signature of the message received exists. Otherwise, it logs the failure to the corresponding error channel.
One has to be very careful with the names and signatures that chooses for the methods and messages in order to make it possible for them to be executed without problems. Collision of names and different signatures make the execution unnecessarily complicated.

Overrides:

execute in class timedNode

Parameters:

`m` - message - the message to be executed

send

```
public void send(message msg)
```

This is the second core functionality of this class. This implements the forwarding of `message` objects from one node to another.
Three different cases are contemplated:

- The destination of msg is not a node, (symbolized by -1). This is only meant to be displayed in a Communication Channel.
- The destination of msg is the same as the sender. This will result in the schedule of an event in the next time unit. This means that when all events that were scheduled for time NOW will be executed, the entity in charge of managing all the nodes (namely, the simulator controller) will increase the time counter, and then events scheduled for time NOW+1 will be executed.
- The third, and most usual case of occurrences is when node A sends msg to B. Then, two things can happen: that B is in the network, or maybe B has left the network (either failed or cleanly left). In the case that B is not in the network, a log line will be written on the chosen Communication Channel, and the sender node will be notified of such event. Otherwise, the event will be scheduled in a future time, depending of the DELAY expected for the communications between nodes to take. This DELAY is a parameter that is provided in the PARAMS class.

Parameters:

`msg` - message - the message to be sent

failed

```
public abstract void failed(String sender,
                            String receiver,
                            String command,
                            String colonSeparatedParamList)
```

This is the signature of the call that any subclass must implement for the event of a failed message.

Parameters:

`sender` - String - The sender of the message
`receiver` - String - The receiver of the message
`command` - String - The command to execute on reception of message
`colonSeparatedParamList` - String - The parameters of the command

Class file

chord

```
java.lang.Object
  └ chord.commChannel
      ━ chord.file
```

public class **file**
extends commChannel
Title:file
Description: this provides the functionality to easily log information into files through `commChannels`
Copyright: Copyright (c) 2005
Company:
Author:
samer
Version:
2.0
See Also:
commChannel, params

Fields inherited from class chord.commChannel
`howMany, name, p`

Constructor Summary	***Page***
`file(String name, params p)` This is the constructor of the class	*75*

Method Summary		***Page***
`void`	`close()` Implementation of the abstract method defined in the superclass.	*75*
`void`	`write(String what)` Implementation of the abstract method defined in the superclass.	*75*

Methods inherited from class chord.commChannel
`close, write`

Constructor Detail

file

```
public file(String name,
            params p)
```

This is the constructor of the class
Parameters:
`name` - String - the name of the `CHANNEL`
`p` - params - access to the constants

Method Detail

write

```
public void write(String what)
```

Implementation of the abstract method defined in the superclass. It adds the string passed at the end of the file
Overrides:
write in class commChannel
Parameters:
`what` - String - the content that will be added at the end of the file

close

```
public void close()
```

Implementation of the abstract method defined in the superclass. It closes the output stream
Overrides:
close in class commChannel

Class InputStreamHandler

chord

```
java.lang.Object
  └java.lang.Thread
      — chord.InputStreamHandler
```

All Implemented Interfaces:
Runnable

public class **InputStreamHandler**
extends Thread
Title: Input Stream Handler
Description: Prevents the improper use of standard input, output and error channels on a getRuntime() call.
Copyright: Copyright (c) 2005
Company:
Author:
Al Sutton from http://hacks.oreilly.com/pub/h/1092
Version:
1.0
See Also:
simulator

Method Summary		**Page**
`void`	`run()` Stream the data.	76

Method Detail

run

`public void run()`

Stream the data.

Specified by:
run in interface Runnable

Overrides:
run in class Thread

Class message

chord
```
java.lang.Object
  └ chord.message
```

public class **message**
extends Object
Title: message
Description: Calls sent among nodes, they're meant to be executed upon reception This is a way of implementing a RPC abstraction, through messages. Access to remote object attributes have to be done through the execution of calls for simplicity
Copyright: Copyright (c) 2005 Samer Al-Kassimi
Author:
samer
Version:
2.0

Field Summary		**Page**
`String`	`command` The name of the method that is expected to be found on the receiver in order to be executed	77
`String`	`msg` Complete string containing the description of the call to be executed on the receiver end	77
`String[]`	`parameters` Parameters of the command.	77
`int`	`receiver` Receiver of the call described in the `message`	77
`int`	`sender` Sender of the `message`	77

Constructor Summary	***Page***
`message(int sender, int receiver, String message)` Constructor operation of the `message`, it needs a sender, a receiver and a `String` that must be of the form `nameOfTheOperation(param1[,paramN])` (and no parameters are accepted too)	*77*

Method Summary		***Page***
`boolean`	`equals(message m)` This method states whether two messages are equal.	*77*
`String`	`print()` This operation returns a `String` with a easily readable description of the `message`, for logging and debugging purposes.	*77*

Field Detail

sender

`public int` **`sender`**

Sender of the `message`

receiver

`public int` **`receiver`**

Receiver of the call described in the `message`

msg

`public String` **`msg`**

Complete string containing the description of the call to be executed on the receiver end

command

`public String` **`command`**

The name of the method that is expected to be found on the receiver in order to be executed

parameters

`public String[]` **`parameters`**

Parameters of the command. For clarity and ease of implementation, all of them are `String` objects, though each one of them will be properly parsed depending on the needs of the method. This parsing will be taken care of in the implementation of the node in each one of the different methods.

Constructor Detail

message

```
public message(int sender,
               int receiver,
               String message)
```

Constructor operation of the `message`, it needs a sender, a receiver and a `String` that must be of the form `nameOfTheOperation(param1[,paramN])` (and no parameters are accepted too)

Parameters:

`sender` - int - the sender of the message
`receiver` - int - the receiver of the message
`message` - String - the text of the message (formed by the command ant its parameters)

Method Detail

print

`public String` **`print`**`()`

This operation returns a `String` with a easily readable description of the `message`, for logging and debugging purposes.

Returns:

String - the result of the call, in format `nameOfTheOperation(param1[,paramN])`

equals

`public boolean` **`equals`**`(message m)`

This method states whether two messages are equal. This is done comparing sender, receiver and the message within.

Parameters:

`m` - message - the message to which THIS is compared to

Returns:

boolean - true if the messages are equal

Class parametersManager

chord

```
java.lang.Object
  └ chord.parametersManager
```

Direct Known Subclasses:

params

public class **parametersManager**
extends Object
Title: parametersManager
Description: manages constant values
Copyright: Copyright (c) 2005 Samer Al-Kassimi
Company:

Author:
samer

Version:
1.0

Constructor Summary	Page
`parametersManager()` The constructor of the class, it calls an initialization routine giving as a parameter the name of the XML file going to be used.	78

Method Summary		Page
`void`	`beginManager(String xmlFileAux)` The initialization routine, it is responsible of calling each sequential step for the storage of the constants and its attributes in a `Hashtable` from which they'll be retrieved by the only public operation in the class.	78
`String`	`get(String group, String name, String attribute)` This is, apart of the constructor, the only public operation.	78

Constructor Detail

parametersManager

`public parametersManager()`

The constructor of the class, it calls an initialization routine giving as a parameter the name of the XML file going to be used.

Method Detail

beginManager

`public void beginManager(String xmlFileAux)`

The initialization routine, it is responsible of calling each sequential step for the storage of the constants and its attributes in a `Hashtable` from which they'll be retrieved by the only public operation in the class.

Parameters:
`xmlFileAux` - String

get

```
public String get(String group,
                  String name,
                  String attribute)
```

This is, apart of the constructor, the only public operation. It provides the means by which constants can be retrieved. The access to constants and their attributes can be achieved my multiple forms, depending on the value of the parameters given.

Parameters:
`group` - String - the group to which the constant belongs
`name` - String - the name of the constant, as specified in the XML file
`attribute` - String - the special attribute that differs some constants with similar names

Returns:
String - the constant value, it may be parsed if necessary

Class params

chord

```
java.lang.Object
  └ chord.parametersManager
       — chord.params
```

public class **params**
extends parametersManager

Constructor Summary	Page
`params()` Constructor of the class.	79

Method Summary		Page
boolean	`getChannel(String name)` This method gives access to constants of the type `CHANNEL`.	79
String	`getChannelNameByPosition(int position)` This method retrieves the name of the POSITIONth `channel` found in the XML file	80
int	`getChannelsCount()` This method informs of the total number of `CHANNEL` names found in the XML file	80
String	`getChannelTypeByPosition(int position)` This method retrieves the type of the POSITIONth `CHANNEL` found in the XML file	80
int	`getConstant(String name)` This method gives access to numeric constants of the type `CONSTANT`.	79
int	`getDelay()`	80
int	`getDelayE()`	80
String	`getFile(String name)` This method gives access to constants of the type `FILE`.	79
String	`getTextConstant(String name)` This method gives access to text constants of the type `CONSTANT`.	80

Methods inherited from class chord.parametersManager
`beginManager, get`

Constructor Detail

params

`public params()`

Constructor of the class. Creates a `params` object by calling the superclass constructor, and sets the operating system variable.

Method Detail

getFile

`public String getFile(String name)`

This method gives access to constants of the type `FILE`. These are dependent of the operating system, so each entry will need to be duplicated in the XML file that stores the constants, having one entry for each platform

Parameters:

`name` - String - the name of the `FILE` that wants to be accessed (meaning regular file, path, or program)

Returns:

String - the value of the constant

getChannel

`public boolean getChannel(String name)`

This method gives access to constants of the type `CHANNEL`.

Parameters:

`name` - String - the name of the `CHANNEL` that wants to be accessed

Returns:

boolean - TRUE if the channel is going to be available for communication

getConstant

`public int getConstant(String name)`

This method gives access to numeric constants of the type `CONSTANT`.

Parameters:

name - String - the name of the CONSTANT that wants to be accessed

Returns:

int - the value of the CONSTANT requested

---

getTextConstant

`public String getTextConstant(String name)`

This method gives access to text constants of the type CONSTANT.

Parameters:

name - String - the name of the CONSTANT that wants to be accessed

Returns:

String - the value of the CONSTANT requested

---

getChannelsCount

`public int getChannelsCount()`

This method informs of the total number of CHANNEL names found in the XML file

Returns:

int - the number of CHANNEL constants

---

getChannelNameByPosition

`public String getChannelNameByPosition(int position)`

This method retrieves the name of the POSITIONth channel found in the XML file

Parameters:

position - int - a int with the index in which we want to find the constant

Returns:

String - the name of the found channel

---

getChannelTypeByPosition

`public String getChannelTypeByPosition(int position)`

This method retrieves the type of the POSITIONth CHANNEL found in the XML file

Parameters:

position - int - a int with the index in which we want to find the constant

Returns:

String - the type of the found item

---

getDelay

`public int getDelay()`

---

getDelayE

`public int getDelayE()`

Class progressMon

chord

```
java.lang.Object
  └ chord.commChannel
        — chord.progressMon
```

---

public class **progressMon**
extends commChannel

Title: progressMon

Description: progress monitor, a bar showing the percentage of the simulation that has so far been completed, it's a subclass of commChannel

Copyright: Copyright (c) 2005 Samer Al-Kassimi

Company:

Author:

samer

Version:

1.0

See Also:

commChannel, params

Fields inherited from class chord.commChannel
howMany, name, p

Constructor Summary	***Page***
progressMon(String name, params p) Constructor of the class, it initializes the monitor by creating the instance of the superclass and setting various parameters.	*81*

Method Summary		Page
void	**close**() Implementation of the abstract method specified by the superclass.	81
void	**write**(String what) Implementation of the abstract method specified by the superclass.	81

Methods inherited from class chord.commChannel
close, write

Constructor Detail

progressMon

```
public progressMon(String name,
                   params p)
```

Constructor of the class, it initializes the monitor by creating the instance of the superclass and setting various parameters.

Parameters:

`name` - String - the name of the communicationChannel, so that it can be addressed within the program to update the status

`p` - params - the constant parameters manager, so constant values can be addressed if necessary (needed to call the superclass constructor)

Method Detail

write

```
public void write(String what)
```

Implementation of the abstract method specified by the superclass. It provides the information of how much of the task has been completed

Overrides:

write in class commChannel

Parameters:

`what` - String - the text to be shown on the progress bar

close

```
public void close()
```

Implementation of the abstract method specified by the superclass. It closes the monitor.

Overrides:

close in class commChannel

Class screen

chord

```
java.lang.Object
  └ chord.commChannel
       — chord.screen
```

All Implemented Interfaces:

ActionListener, EventListener

public class **screen**
extends commChannel
implements ActionListener
Title: screen
Description:
Copyright: Copyright (c) 2005
Company:

Author:

not attributable

Version:

2.0

Fields inherited from class chord.commChannel
howMany, name, p

Constructor Summary	Page
screen(String name, params p)	82

Method Summary		Page
void	**actionPerformed**(ActionEvent e)	82
void	**close**() So far, SWING windows showing don't need specific handling	82
void	**write**(String what) Specific operation that makes information to be shown on the screen channel	82

Methods inherited from class chord.commChannel
close, write

Constructor Detail

screen

```
public screen(String name,
              params p)
```

Method Detail

actionPerformed

```
public void actionPerformed(ActionEvent e)
```

Specified by:

actionPerformed in interface ActionListener

write

```
public void write(String what)
```

Specific operation that makes information to be shown on the screen channel

Overrides:

write in class commChannel

Parameters:

what - String

close

```
public void close()
```

So far, SWING windows showing don't need specific handling

Overrides:

close in class commChannel

Class simulator

chord

```
java.lang.Object
  └ chord.simulator
```

public class **simulator**
extends Object
Title: Simulator
Description: network simulator, container of the main routine
Copyright: Copyright (c) 2005 Samer Al-Kassimi
Company:

Author:

samer

Version:

2.0

Constructor Summary	Page
simulator() Default initialization function	82

Method Summary		Page
static void	**main**(String[] args) Program launcher, calls the simulator to run	83

Constructor Detail

simulator

```
public simulator()
```

Default initialization function

Method Detail

main

```
public static void main(String[] args)
```

Program launcher, calls the simulator to run

Parameters:

`args` - String[]

Class stat

chord

```
java.lang.Object
  └ chord.commChannel
       — chord.stat
```

public class **stat**
extends commChannel
Title: stat
Description: the module that accumulates statistical data and plots it
Copyright: Copyright (c) 2005 Samer Al-Kassimi
Company:
Author:
Samer Al-Kassimi
Version:
1.0
See Also:
commChannel, params

Field Summary		***Page***
protected String	**baseFName** The Strings containing the names of the files	84
protected String	**dataFName** The Strings containing the names of the files	84
protected FileOutputStream	**dataFOS** The streams that are used to gather the data necessary to generate the plot	84
protected float	**failure**	85
protected float	**failureDocument**	85
protected float	**fixFingersCount**	85
protected float	**fixFingersLengthAccum**	85
protected float	**left**	85
protected Hashtable	**lengths**	85
protected float	**lookupCount**	85
protected float	**lookupLengthsAccum**	85
protected float	**lookupResultCount**	85
protected float	**maxLength**	85
protected float	**maxTimeout**	85
protected float	**minLength**	85
protected float	**notClean**	85
protected String	**scriptFName** The Strings containing the names of the files	84
protected FileOutputStream	**scriptFOS** The streams that are used to gather the data necessary to generate the plot	84
protected float	**stabilizationCount**	85
protected String	**summary** The Strings containing the names of the files	85
protected String	**summaryFName** The Strings containing the names of the files	84
protected FileOutputStream	**summaryFOS** The streams that are used to gather the data necessary to generate the plot	84
protected float	**timeoutCount**	85
protected String	**timeoutFName** The Strings containing the names of the files	85

protected FileOutputStream	**timeoutFOS** The streams that are used to gather the data necessary to generate the plot	*84*
protected Hashtable	**timeouts**	*85*

Fields inherited from class chord.commChannel
`howMany`, `name`, `p`

Constructor Summary	***Page***
stat(String name, params p) This is the constructor of the class, it is initialized with a name and a `params` object.	*85*

Method Summary		***Page***
float	**calculatePercentile**(float[] data, float whichPercentile, int step)	*86*
float	**calculateTimeoutPercentile**(float[] data, float whichPercentile, int step)	*86*
void	**close**() Implementation of the abstract method defined in the superclass.	*86*
float	**doCalculation**(float[] pAccums, float vTCR, int s)	*86*
protected String	**format**(float value)	*86*
protected String	**format2**(float value)	*86*
void	**generateDataFiles**()	*86*
void	**generatePlots**()	*86*
String	**genSummary**()	*86*
protected void	**initializeFileNames**()	*85*
protected void	**initializeFiles**(String initSettingsForFile)	*85*
float	**percentile**(Hashtable data, float whichPercentile)	*86*
float	**percentileTimeouts**(Hashtable data, float whichPercentile)	*86*
float[]	**putDataIntoArray**(Hashtable data)	*86*
float[]	**putTimeoutDataIntoArray**(Hashtable data)	*86*
void	**showStats**()	*86*
void	**write**(String what) Implementation of the abstract method defined in the superclass.	*85*

Methods inherited from class chord.commChannel
`close`, `write`

Field Detail

scriptFOS

`protected FileOutputStream` **`scriptFOS`**

The streams that are used to gather the data necessary to generate the plot

dataFOS

`protected FileOutputStream` **`dataFOS`**

The streams that are used to gather the data necessary to generate the plot

summaryFOS

`protected FileOutputStream` **`summaryFOS`**

The streams that are used to gather the data necessary to generate the plot

timeoutFOS

`protected FileOutputStream` **`timeoutFOS`**

The streams that are used to gather the data necessary to generate the plot

baseFName

`protected String` **`baseFName`**

The Strings containing the names of the files

scriptFName

`protected String` **`scriptFName`**

The Strings containing the names of the files

dataFName

`protected String` **`dataFName`**

The Strings containing the names of the files

summaryFName

`protected String` **`summaryFName`**

The Strings containing the names of the files

timeoutFName

`protected String timeoutFName`

The Strings containing the names of the files

summary

`protected String summary`

The Strings containing the names of the files

lengths

`protected Hashtable lengths`

timeouts

`protected Hashtable timeouts`

maxLength

`protected float maxLength`

minLength

`protected float minLength`

lookupCount

`protected float lookupCount`

lookupResultCount

`protected float lookupResultCount`

fixFingersCount

`protected float fixFingersCount`

fixFingersLengthAccum

`protected float fixFingersLengthAccum`

stabilizationCount

`protected float stabilizationCount`

lookupLengthsAccum

`protected float lookupLengthsAccum`

timeoutCount

`protected float timeoutCount`

maxTimeout

`protected float maxTimeout`

notClean

`protected float notClean`

left

`protected float left`

failure

`protected float failure`

failureDocument

`protected float failureDocument`

Constructor Detail

stat

```
public stat(String name,
            params p)
```

This is the constructor of the class, it is initialized with a name and a `params` object. After initialization (by calling the superclass constructor), it obtains the names of the files that are going to be used and initializes the files so the necessary data can be read or write, as necessary.

Parameters:

`name` - String - the name of the gnuPlot instance (the channel name)

`p` - params - access to the constants

Method Detail

initializeFileNames

`protected void initializeFileNames()`

initializeFiles

`protected void initializeFiles(String initSettingsForFile)`

write

`public void write(String what)`

Implementation of the abstract method defined in the superclass. It updates the statistical data referred by the contents of the parameter

Overrides:

write in class commChannel

Parameters:

`what` - String - the statistical data that needs to be updated

close

```
public void close()
```

Implementation of the abstract method defined in the superclass. It closes the communication channel. Before closing, the statistical data is prepared to be inserted into data files. After generating these files, they will be fed, along with a script, to the gnuPlot program. The script file instructs gnuPlot to generate GIF images containing the plots.

Overrides:

close in class commChannel

format

```
protected String format(float value)
```

format2

```
protected String format2(float value)
```

showStats

```
public void showStats()
```

genSummary

```
public String genSummary()
```

putDataIntoArray

```
public float[] putDataIntoArray(Hashtable data)
```

putTimeoutDataIntoArray

```
public float[] putTimeoutDataIntoArray(Hashtable data)
```

percentile

```
public float percentile(Hashtable data,
                        float whichPercentile)
```

percentileTimeouts

```
public float percentileTimeouts(Hashtable data,
                                float whichPercentile)
```

calculatePercentile

```
public float calculatePercentile(float[] data,
                                 float whichPercentile,
                                 int step)
```

calculateTimeoutPercentile

```
public float calculateTimeoutPercentile(float[] data,
                                        float whichPercentile,
                                        int step)
```

doCalculation

```
public float doCalculation(float[] pAccums,
                           float vTCR,
                           int s)
```

generatePlots

```
public void generatePlots()
```

generateDataFiles

```
public void generateDataFiles()
```

Class std

chord

```
java.lang.Object
  └ chord.commChannel
      — chord.std
```

public class **std**
extends commChannel
Title:std
Description: this provides the functionality to easily log information out to STDOUT and STDERR channels, in accordance to the rest of the program, making it easy to activate and deactivate this level of information display
Copyright: Copyright (c) 2005 Samer Al-Kassimi
Company:
Author:

samer

Version:
 1.0
See Also:
 commChannel, params

Fields inherited from class chord.commChannel
`howMany, name, p`

Constructor Summary	Page
`std(String name, params p)` This is the constructor of the class	87

Method Summary		Page
`void`	`close()` Implementation of the abstract method defined in the superclass.	87
`void`	`write(String what)` Implementation of the abstract method defined in the superclass.	87

Methods inherited from class chord.commChannel
`close, write`

Constructor Detail

std

```
public std(String name,
           params p)
```

This is the constructor of the class
Parameters:
 `name` - String - the name of the `CHANNEL`
 `p` - params - access to the constants

Method Detail

write

```
public void write(String what)
```

Implementation of the abstract method defined in the superclass. It displays the information into STDERR or STDOUT as wished
Overrides:
 write in class commChannel
Parameters:
 `what` - String - the content that will be added at the end of the file

close

```
public void close()
```

Implementation of the abstract method defined in the superclass.
Overrides:
 close in class commChannel

Class timedNode

chord
```
java.lang.Object
  └ chord.timedNode
```
Direct Known Subclasses:
 distributedNode

abstract public class **timedNode**
extends Object
Title: timedNode
Description: Basic node of a network, provides a queue of events to be executed in timely fashion, and the interface by the means of which those events will eventually be executed
Copyright: Copyright (c) 2005 Samer Al-Kassimi
Company:
Author:
 samer

Version:
2.0
See Also:
message, commChannelsManager

Field Summary		Page
commChannelsManager	**commChannels** Tool used to log information through different channels.	*88*
int	**id** Identifier of the node	*88*
int	**now** Current time, it is updated on creation of the node and upon calls to the `trigger` method	*88*
Hashtable	**queue** Queue of events that a node has to perform.	*88*

Constructor Summary	Page
timedNode(int id, int time, commChannelsManager channels) This is the constructor method for `timedNode`.	*88*

Method Summary		Page
abstract void	**execute**(message event) Abstract method to be implemented by a lower class in the inheritance hierarchy of nodes.	*89*
void	**scheduleEvent**(message message, int time) This is the way messages are inserted in the queue of events of a node.	*89*
void	**trigger**(int time) This is the public method offered in order to execute all events that might have been scheduled for a given time.	*88*

Field Detail

id

`public int id`

Identifier of the node

queue

`public Hashtable queue`

Queue of events that a node has to perform. The keys are `Integer` objects. The `int` value stored in those objects represents a time moment, and the object stored for that key is a `Vector`. The elements of this `Vector` are messages, and those messages can be executed with the `execute` method.

now

`public int now`

Current time, it is updated on creation of the node and upon calls to the `trigger` method

commChannels

`public commChannelsManager commChannels`

Tool used to log information through different channels.

Constructor Detail

timedNode

```
public timedNode(int id,
                 int time,
                 commChannelsManager channels)
```

This is the constructor method for `timedNode`. The ID of the node, the time of creation and a valid reference to the Communication Channels tool have to be provided.

Parameters:

`id` - int - identifier of the node
`time` - int - time of creation
`channels` - commChannelsManager - access to the logging means

Method Detail

trigger

`public void trigger(int time)`

This is the public method offered in order to execute all events that might have been scheduled for a given time. The caller has to take care of the correct generation of time sequences for the simulation to make sense.

Parameters:

`time` - int - the time of the actions to be performed

---

scheduleEvent

```
public void scheduleEvent(message message,
                          int time)
```

This is the way messages are inserted in the queue of events of a node. A message is only scheduled for execution if the time at which this message is supposed to be executed is bigger (which means later) that the current time.

Parameters:

`message` - message

`time` - int

---

execute

```
public abstract void execute(message event)
```

Abstract method to be implemented by a lower class in the inheritance hierarchy of nodes. It provides an interface by the means of which messages sent to a node can be executed, if the node provides a valid interface for the execution of the message.

Parameters:

`event` - message - the message to be executed

8.4 References

[STO-1] ION STOICA, ROBERT MORRIS, DAVID LIBEN-NOWELL, DAVID R. KARGER, M. FRANS KAASHOEK, FRANK DABEK, HARI BALAKRISHNAN, Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. ACM SIGCOMM (San Diego, CA, 2001) pp. 149-160

[FRE-2] CLARKE, I. Freenet: A distributed decentralized information storage and retrieval system. Master's thesis, University of Edinburgh, 1999

[FRE-3] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T.W. Freenet: A distributed anonymous information storage and retrieval system. In Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability (Berkeley, California, June 2000)

[OCS-4] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000) (Boston, MA, November 2000), pp. 190-201.

[PLA-5] PLAXTON, C., RAJARAMAN, R., AND RICHA, A. Accessing nearby copies of replicated objects in a distributed environment. In Proceedings of the ACM SPAA (Newport, Rhode Island, June 1997), pp. 311-320

[NAP-6] Napster

http://www.napster.com/

[GNU-7] Gnutella

http://gnutella.wego.com/

[SEA-8] EL-ANSARY, S. A Framework For The Understanding, The Optimization, and Design Of Structured Peer-To-Peer Systems. Licentiate Philosophy Dissertation, Royal Institute of Technology, 2003.

[ORA-9] ORAM, A. Peer-To-Peer: Harnessing the Power of Disruptive Technologies. O'Reilly, first edition, March 2001. ISBN:0-596-00110-X.

[ORA-10] ORAM, A. What is P2P... And What isn't?, November 2000.
http://www.oreillynet.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html.

[MIL-11] MILLER, M. Discovering P2P. Sybex International, November 2001. ISBN-0782140181.

[PTP-12] Peer-To-Peer Working Group. What is Peer-To-Peer?, 2001.
http://www.peer-to-peerwg.org/whatis/index.html.

[MAR-13] MARKATOS, E.P. Tracing a Large-Scale Peer to Peer System: An Hour in the Life of Gnutella. In The Second International Symposium on Cluster Computing and the Grid, 2002.
http://www.ccgrid.org/ccgrid2002

[RIP-14] RIPEANU, M., FOSTER, I., IAMNITCHI, A. Mapping The Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems And Implications For System Design. IEEE Internet Computing Journal, 6(1), 2002

[KAZ-15] Kazaa
http://www.kazaa.com

[CAN-16] RATSANAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., SHENKER, S. A Scalable Content Addressable Network. Technical Report TR-00-010, Berkeley, CA, 2000

[PAS-17] ROWSTRON, A., DRUSCHEL, P. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. Lecture Notes in Computer Science, 2218, 2001.
http://citeseer.nj.nec.com/rowstron01pastry.html

[TAP-18] ZHAO, B.Y., KUBIATOWICZ, J.D., JOSEPH, A.D. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. U. C. Berkeley Technical Report UCB//CSD-01-1141, April 2000

[ALI-19] ALIMA, L.O., HARIDI, S., GHODSI, A., EL-ANSARY, S., BRAND, P. Position Paper: "Self-"properties in Distributed K-ary Structured Overlay Networks. KTH, Royal Institute of Technology, SICS, Swedish Institute of Computer Science, Kista, Sweden.

[DIJ-20] DIJKSTRA, E. W. Self-stabilizing systems in spite of distributed control, Communications of the ACM 17 (1974), 643–644.

[LAS-21] LASZLO, E. Basic constructs of systems philosophy, Systematics 10 (1972), 40–54.

[STU-22] STUTZBACH, D., REJAIE, R. Characterizing Churn in Peer-to-Peer Networks Technical Report CIS-TR-2005-03 University of Oregon June 3, 2005

[MAH-23] MAHAJAN, R., CASTRO, M., ROWSTRON, A. Controlling the cost of reliability in peer-to-peer overlays, LNCS 2735, Proceedings of the Second International Workshop IPTPS 2003 (Berkeley), Springer, 2003.
[ALI-24] ALIMA, L.O., EL-ANSARY, S., BRAND, P., HARIDI, S. DKS(N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications, The 3rd International workshop CCGRID2003 (Tokyo, Japan), May 2003.
[CAS-25] CASTRO, M., COSTA, M., ROWSTRON, A. Should we build Gnutella on a structured overlay? Microsoft Research, Cambridge, CB3 0FB, UK
http://nms.lcs.mit.edu/HotNets-II/papers/structella.pdf
[AND-26] ANDROUTSELLIS-THEOTOKIS, S., SPINELLIS, D. A Survey of Peer-to-Peer File Sharing Technologies. White paper, Electronic Trading Research Unit (ELTRUN), Athens University for Economics and Business, 2002.
http://www.eltrun.aueb.gr/whitepapers
[LVQ-27] LV, Q., CAO, P., COHEN, E., LI, K., SHENKER, S. Search and Replication in Unstructured Peer-to-Peer Networks. In Scott T. Leutenegger, editor, Proceedings of the 2002 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS-02), volume 30, 1 of SIGMETRICS Performance Evaluation Review, pages 258–259, New York, June 15–19 2002. ACM Press.
[RIS-28] RISSON, J., MOORS, T., Survey of Research towards Robust Peer-to-Peer Networks: Search Methods. Technical Report UNSW-EE-P2P-1-1, University of New South Wales, Sydney, Australia, Sep. 2004.
http://www.ee.unsw.edu.au/?timm/pubs/robust_p2p/submitted.pdf
[JLI-29] LI, J., LOO, B.T., HELLERSTEIN, J., KAASHOEK, F., KARGER, D.R., MORRIS, R. On the Feasibility of Peer-to-Peer Web Indexing and Search. In 2nd International Workshop on Peer-to-Peer Systems (IPTPS ’03), Berkeley, CA, USA, February 2003.
[HAR-30] HARREN, M., HELLERSTEIN, J.M., HUEBSCH, R., LOO, B.T., SHENKER, S., STOICA, I., Complex Queries in DHT-based Peer-to-Peer Networks. In The 1st International Workshop on Peer-to-Peer Systems (IPTPS’02), 2002.
http://www.cs.rice.edu/Conferences/IPTPS02/
[AND-31] ANDRZEJAK, A., XU, Z. Scalable, Efficient Range Queries for Grid Information Services. In 2nd International Conference on Peer-To-Peer Computing, pages 33–40, Linköping, Sweden, September 2002. IEEE Computer Society. ISBN-0-7695-1810-9.
[SCH-32] SCHLOSSER, M., STINEK, M., DECKER, S., NEJDL, W. A Scalable and Ontology-Based P2P Infrastructure for Semantic Web Services. In 2nd International Conference on Peer-To-Peer Computing, pages 104–111, Linköping, Sweden, September 2002. IEEE Computer Society. ISBN-0-7695-1810-9.
[DAB-33] DABEK, F., KAASHOEK, M.F., DARGER, D., MORRIS, R., STOICA, I. Wide-Area Cooperative Storage With CFS. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP ’01), Chateau Lake Louise, Banff, Canada, October 2001.
[ZXU-34] XU, Z., MALLIK, M., KARLSSON, M. Turning Heterogeneity into an Advantage in Overlay Routing. Technical Report HPL-2002-126R2, Hewlett-Packard Labs, July 2002.
http://www.hpl.hp.com/techreports/2002/HPL-2002-126R2.html
[STO-35] STOICA, I., ADKINS, D., RATNASAMY, S., SHENKER, S., SURANA, S., ZHUANG, S. Internet Indirection Infrastructure. In The 1st International Workshop on Peer-to-Peer Systems (IPTPS’02), 2002.
http://www.cs.rice.edu/Conferences/IPTPS02/
[RAT-36] RATNASAMY, S., HANDLEY, M., KARP, R., SHENKER, S., Application-level Multicast using Content-Addressable Networks. In Third International Workshop on Networked Group Communication (NGC ’01), 2001.
http://www-mice.cs.ucl.ac.uk/ngc2001/
[CAS-37] CASTRO, M., DRUSCHEL, P., KERMARREK, A.M., ROWSTRON, A., SCRIBE: A Large-Scale And Decentralized Application-Level Multicast Infrastructure. IEEE Journal on Selected Areas in Communications (JSAC) (Special issue on Network Support for Multicast Communications, 2002.
[TAN-38] TANENBAUM, A.S., VAN STEEN, M., Distributed Systems: Principles and Paradigms. Prentice Hall, Inc., 2002. ISBN-0-13-088893-1.
[BAE-39] BAEHNI, S., EUGSTER, P., GUERRAOUI, R. OS Support For P2P Programming: A Case For TPS. In 22nd International Conference on Distributed Computing Systems (ICDCS ’02), pages 355–362, Washington - Brussels - Tokyo, July 2002. IEEE.

[IIC-40] The IEEE International Conference on Peer-To-Peer Computing, Use of Computers at the Edge of Networks P2P, Grid, Clusters.
http://www.ida.liu.se/conferences/p2p/p2p2002/
[ISC-41] The IEEE International Symposium on Cluster Computing and the Grid.
http://www.ccgrid.org
[GGF-42] The Global Grid Forum, 2003.
http://www.gridforum.org
[ROG-43] Relation of OGSA/Globus and Peer2Peer, 2003.
http://www.gridforum.org/4 GP/ogsap2p.htm

[STO-44] ION STOICA, ROBERT MORRIS, DAVID LIBEN-NOWELL, DAVID R. KARGER, M. FRANS KAASHOEK, FRANK DABEK, HARI BALAKRISHNAN, Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. Technical Report TR-819, MIT LCS, Jan 2002

http://www.pdos.lcs.mit.edu/chord/papers

[CAR-45] CARTER, J.L., AND WEGMAN, M.N. Universal classes of hash functions. Journal of Computer and System Sciences 18,2 (1979)

[FIP-46] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, Apr. 1995

[KAR-47] DAVID R. KARGER, E. LEHMAN, F. LEIGHTON, M. LEVINE, D. LEWIN, R. PANIGRAHY Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. 29th Annual ACM Symposium on Theory of Computing (El Paso, TX, May 1997)

[LEW-48] D. LEWIN Consistent hashing and random trees: Algorithms for caching in distributed networks. Masters Thesis, Department of EECS, MIT, 1998

http://thesis.mit.edu

[HAC-49] Hack's at O'Reilly, available at

http://hacks.oreilly.com/pub/h/1092

[TCP-50] Transmission Control Protocol, RFC 793

http://www.faqs.org/rfcs/rfc793.html

[WIK-51] Definition of "peer to peer" at wikipedia:

http://en.wikipedia.org/wiki/Peer-to-peer

[WEB-52] Definition of "peer to peer" at webopedia
http://www.webopedia.com/TERM/p/peer_to_peer_architecture.html

[LSO-53] Definition of "scalability" at LSoft:

http://www.lsoft.com/resources/glossary.asp#S

[WIK-54] Definition of "scalability" wikipedia:

http://en.wikipedia.org/wiki/Scalability

[CIS-55] Definition of "scalability" at CISCO:
http://www.cisco.com/univercd/cc/td/doc/product/dsl_prod/6160/hwguide/glossary.htm

[WIK-56] Definition of "fully distributed systems" at wikipedia:
http://en.wikipedia.org/wiki/Distributed_computing

[WIK-57] Definition of "distributed hash tables" at wikipedia:
http://en.wikipedia.org/wiki/Distributed_hash_table

[WIK-58] Definition of "modulo" at wikipedia:

http://en.wikipedia.org/wiki/Modular_operation

[MAT-59] Definition of "modulo" at the glossary of the Mathematics Lair:
http://www.stormloader.com/ajy/glossary.html

[UCC-60] Definition of "modulo" at the Uniform Code Council:
http://usnet03.uc-council.org/glossary/#M

[WIK-61] Definition of “RPC” at wikipedia:

http://en.wikipedia.org/wiki/Remote_procedure_call

[WIK-62] Definition of “message passing” at wikipedia:
http://en.wikipedia.org/wiki/Message_passing

[NAP-63] Definition of “message passing” at the National Academies Press
http://books.nap.edu/html/up_to_speed/appD.html