SCIENTIFIC and TECHNOLOGICAL COOPERATION
between
RTD ORGANISATIONS in GREECE
and
RTD ORGANISATIONS in U.S.A, CANADA, AUSTRALIA, NEW ZEALAND, JAPAN, SOUTH KOREA, TAIWAN, MALAISIA and SINGAPORE

SecSPeer: Secure and Scalable peer-to-peer computing and communication systems
(Contract no: ΗΠΑ-021**)**

D2.1 "System Design"

Abstract: This document describes our proposed solutions for improving the scalability and security of unstructured Peer-to-Peer systems.

Contractual Date of Delivery	29 June 2005
Actual Date of Delivery	
Deliverable Security Class	Public
Editor	Charalambos Papadakis
Contributors	Elias Athanasopoulos, Evangelos Markatos, Paraskevi Fragkopoulou

The SecSPeer Consortium consists of:

FORTH-ICS	Coordinator	Greece
University of Pittsburgh	Partner	USA
Virtual Trip Ltd.	Partner	Greece

Contents

List of Figures

List of Tables

Introduction

In this document, we desribe the design of SecSPeer, a secure and scalable, unstructured, peer-to-peer system. The document organization is as follows:

Chapter 1 provides an overview of this document.

Chapter 2 provides a brief description of the main architectures used today in unstructured P2P systems.

Chapters 3 and 4 describes the proposals tthat make SecSPeer scalable.

Chapter 5 describes the mechanisms designed to make SecSPeer secure.

Chapter 6 correlates this document with SecSPeer Deliverable 1.1: Systems Requirements document.

The document ends with the Bibliography.

Chapter 1

Overview of the architecture

In this chapter, we describe the overall architecture of the system. At the same time, we present, in brief, the structure of this document.

1.1 System Objectives

In more traditional distributed, content delivery systems, like the World Wide Web, each participant in the system plays a distinct role. Web servers play the role of the content providers, while clients (web browsers) request content from the servers. In a peer-to-peer (P2P) system, each participant offers content to the rest of the participants and at the same time can also request content from them. As a content delivery system, one of the most important functions of any P2P system is the search for a piece of data offered by anyone of the participants. Since, each participant offers any and whatever content he/she likes, that piece of data can be on any participant of the network. For that reason, the prevailing method used is broadcasting (i.e.: trying to ask everyone in the system). Since each participant is aware of only a few other participants, this broadcast is implemented by having each participant that receives the message forward it to all of its neighbors (participants of the network it is aware of and connected to) and so forth. The main Achilles' heel in peer-to-peer system scalability is the number of messages required for these broadcasts. The objectives of SecSPeer is to improve peer-to-peer scalability and also tackle peer-to-peer security issues, such as DDoS attacks.

1.2 System Architecture

The SecSPeer architecture is comprised of several improvements on today's prevailing, peer-to-peer architectures, with the purpose of improving their scalability and security. For this reason, in Chapter 2, we provide a brief description of the main architectures used today in unstructured P2P systems. In the following chapters (3-4-5), we describe the design of SecSPeer, which builds on today's unstructured P2P architectures. Chapters 3 and 4 try to improve the scalability of the

system by reducing the number of ”worthless” messages sent during a broadcast. ”Worthless” messages are messages that do not increase our chance of locating the required piece of data.

In Chapter 3, we describe the first architectural change proposed by SecSPeer, to improve P2P system scalability. As is explained in that chapter, the broadcasting mechanism results in the transmission of redundant messages (i.e.: sending the same message to the same participant more than once). The purpose of the proposed algorithm is the elimination of those redundant messages, in order to reduce the number of worthless messages during a broadcast. The system tries to avoid forwarding a message to a participant that may have already received it by learning from traffic history, through the use of explicit duplicate notification from the receiving participant.

In Chapter 4, we describe another scalability improvement algorithm, which tackles the issue of blind broadcast, by adding semantic information to the network, so as to be able to broadcast to only a subset of participants of the network. This way we try to avoid generating another type of worthless messages, namely messages sent to participants that do not have the data being looked up.

Chapter 5 describes SecSPeer’s proposals of dealing with security issues, such as spam generation and DDoS attacks. Broadcasting creates significant amount of traffic which is shared by all the nodes of the system, on behalf of the broadcasting node. To avoid exploitation, it is essential that amount of traffic load a node injects in the system be relative to other nodes’ amount of traffic load it serves. Spam generation is created from malicious nodes that reply to queries for content they do not have. Since the amount of results they send is arbitrary, this reduces the amount of ”good” results the requesting node receives. This document concludes with some analysis of possible, future directions and a correlation of the design with Deliverable 1.1: System Requirements Document[6].

Chapter 2

Brief Description and Analysis of today's Peer-to-Peer architectures

In this chapter, we provide some background knowledge of existing P2P systems, required for the understanding of the SecSPeer design.

2.1 Introduction

P2P networking has generated tremendous interest worldwide among Internet users as well as computer networking professionals and researchers. P2P software systems like Kazaa and Gnutella clients rank amongst the most popular software applications ever. Numerous businesses and Web sites have promoted "P2P" technology as the future of Internet networking.

The basic notion behind peer-to-peer systems and the one that distinguishes them from more traditional client/server architectures is that those two roles are not cleanly separated. Each user that participates in the network, offers its own pieces of data to the rest of the network, and thus acts as a server, but is also at the same time, able to request data from rest of the users in the network, and thus, acts as a client. This was the idea behind the adoption of the term servent (SERVer-cliENT), to characterize each participant in the network.

2.2 Unstructured systems

2.2.1 Centralized

The first P2P system, to make an impact in the world, was Napster [4]. Napster was not completely decentralized, since the knowledge of the location of every piece of data (which user possesses which data) resided on a single, well-known server (and not servent). Each participant notified the central server for the data it serves, during its bootstrapping process. For that reason, there was no need for a broadcasting technique, since every search for some piece of data was directed to

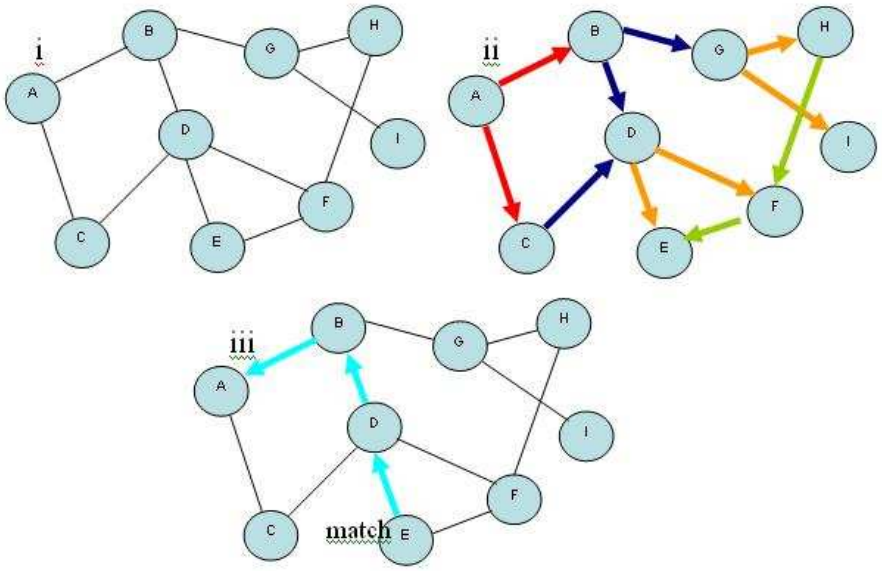

Figure 2.1: Unstructured, decentralized P2P network topology without Ultrapeers.
(i). A is connected to and aware of only B and C.
(ii). A initiates the flooding process, At the first phase of the flooding, A will send its request to B and C. At the second phase, B and C will send it to all of their neighbours, except the one they got it from (A). Since D is both B and C's neighbour, it will receive the message twice (the second message being redundant).
(iii). Assuming that servent E contains the piece of data A is looking for, it will send the reply to D (assuming the message from D arrived before the message from F in (ii)). D will forward it to B (again, assuming that B's message to D in (ii) arrived before C's message to D) and B finally to A.

that server. We use the past tense in describing Napster, since we refer to the "old" Napster, instead of its reincarnation as a subsriber service.

2.2.2 Decentralized

The next generation of unstructured P2P systems includes Gnutella and KaZaA [11] [5]. Both these systems are completely decentralized in the sense that there is no central server like the one in Napster, to facilitate the lookup of data. This made those systems more robust, but also made looking up data more difficult, since each participant in the network was aware only of its own data. In these systems, each servent knows of and is connected to a small subset of other servents in the network. In turn, each servent in that subset is connected to another subset of servent (which includes the first one, since the connections are bidirectional), and so on.

Lookup in decentralized systems - Flooding

Since there is no centralized server to know the location of all data, a servent looking for a piece of data needs to ideally ask every other servent in the network for it. Since each servent is aware of only a few other servents, the broadcast is carried out by all the servents in the network, each sending each request it receives to all of its neighbours (except the one it received it from) and so on. This leads to the philosophy of ”all for one and one for all”, since all the servents cooperate to forward the broadcast of a single servent and every servent forwards every broadcast it receives, from the rest of the network. This results in a flood of messages spreading through the network. Each servent, that contains the piece of data being requested, sends a message back to the requesting servent. Since the query message does not contain the identity of the servent that initiated the flooding, the answering servent sends the reply to the neighbor it received the request from, which in turn forwards it to the servent it received the request from and so forth Figure 2.1.

In the same way, each servent is able to learn of the existance of other servents. Each servent that wishes to connect to the network need only know the existance of one other servent. This it learns from well-known servers, called web-caches. When the address of one servent in the network is known, a broadcast can be initiated, requesting each servent that receives it, to contact the first servent and notify it of their existence.

This broadcast technique used in these systems, called flooding, has two obvious drawbacks. The number of messages during the progress of a single flooding increases exponentially and since two different servents may be connected to the same third servent, that servent will receive the broadcasted message twice. The number of messaged created is bounded by adding to each message a TTL (Time To Live) field. This field is similar to the TTL field of IP packets and contains the number of times the message can be forwarded before it is discarded. This method does reduce the number of messages generated, but also reduces the number of servents that can be contacted.

As explained in Figure 1, a servent may receive a message belonging to the same flood, more that once. A servent that receives the same broadcasted message more than once simply ignores the copies other than the first. Messages belonging to different floodings are distinguished from a global ID. Messages belonging to the same flooding have the same ID.

To improve the scalability of this algorithm, both systems introduced the notion of UltraPeers [1](SuperNodes in KaZaA), which we describe later on in this document. In short, since servents with low bandwidth reduce the speed at which flooding progresses and thus, the scalability of the system, these servents were removed from the network as such, and instead, were connected to it through another servent with better network capabilities. Since these ”leaf” servents are connected to their parent servent only and not to each other, there is no need to forward any message they receive from the parent servent. What is more, at the time they connect to a parent servent, they send it a hashed index of their contents, so that the

parent servent will only forward searches to them if they may have the piece of data being looked for. Thus, flooding is only used at the parent servents' level. If the UltraPeer/SuperNode of a leaf fails, the leaf reconnects to another one. The leaf receives a list of UltraPeers/SuperNodes from each parent it connects to as alternatives. If this list is empty, Webcaches are consulted.

2.3 Structured systems

Finally, it must be noted that there is a family of decentralized P2P systems that do not require broadcasting a message in every direction, in order to find some piece of information, even though there is no centralized server. This is possible by imposing some order among the stored information. In these, structured P2P systems, even though a requesting servent may not be aware of the exact location of the data it is looking for, it knows in which ”direction” it can be found. This way, lookup is performed by propagating a single message in the right direction, instead of flooding. However, the increased state information required by those systems make them much more vulnerable to failures. SecSPeer aims at making unstructured P2P systems as scalable as structured, in terms of network load, while maintaining unstructured systems' scalability, in terms of robustness. Examples of structured systems are [7], [10] and [9].

Chapter 3

A feedback-based approach to reduce duplicate messages

In this chapter, we present SecSPeer's approach to eliminating redundant traffic during the flooding process. Throughout the rest of this document, we model the topology of an unstructured P2P network as a graph, where each servent is a node of the graph and there is an edge between any node/servent and its neighbours. Since communication is bidirectional on those connections, so are the edges in the graph.

3.1 Problem Description

As mentioned before, one drawback of the flooding mechanism, which is blatant, is the generation of duplicate messages. In Figure 2.1(ii), we gave simple demonstration of how duplicate messages are generated, since B and C nodes both sent the same message to D. What is more, and not readily obvious from the figure, is the fact that D will forward the first message it receives, as soon as it receives it. Assuming that B's message reached D before C's message, this will mean that D will forward the message to all of its neighbours, except the one it received it from (that is B), which includes C. This results in the generation of two duplicates, one from C to D and one from D to C (Figure 3.1). In general, in a network of N servents with d connections per servent, on average, the cost, in messages, of a flood intending to reach every node in the network (boundless TTL) will be d + (N-1)(d-1). This is because every servent will receive the message at least once, and will forward it to every neighbour except one. Only the servent initiating the flood will send it to all of its neighbours. The minimum number of messages required to flood the network is N-1 (the number of servents in the network, minus the servent initiating the flood). Thus, the number of (redundant) duplicate messages is d + (N-1)(d-2). The ratio of duplicate messages versus total messages is (d-2)N + 2 / (d-1)N + 1. For large N, this roughly equals (d-2)/(d-1). The higher the degree, the more the duplicates. This is a problem, because unstructured P2P networks have

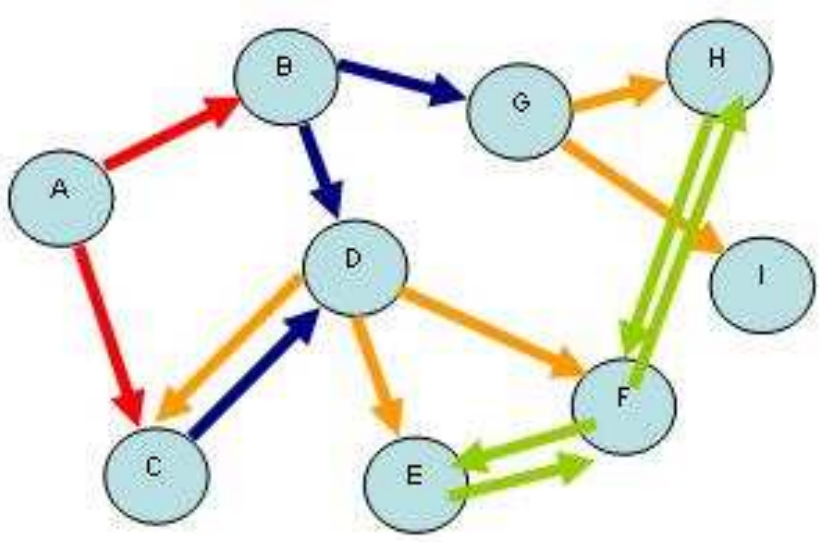

Figure 3.1: Example of duplicates' generation. Duplicates have been generated on whichever node lies at the end-point of more than one arrow.

much to gain from having high degree nodes, with the most important benefit being robustness.

3.2 Problem characteristics/attributes

3.2.1 Introduction to clustering

In a random graph, each node has the same probability of being connected to any of the rest of the nodes. In contrast, in small-world graphs [12], each node is more likely to connect to nodes close to him, as defined by some closeness criterion, than nodes that are far away from him. Notice that this kind of closeness is defined by some other criterion than hops distance. Since, any node is close to most of his neighbors (with high probability) and they are close to their neighbors, it follows that the first node is also close to his neighbors' neighbors, and thus they may also be his neighbors with high probability. The above lead to two interesting conclusions. One is that since in random graphs we have links to any node with the same probability, random graphs have the smallest diameter[1]. In a small -world graph with the same average degree, most of the edges around a node lead to nodes inside the "neighborhood" and so we have very few edges to lead us to the other "neighborhoods", which leads to longer diameter. The extent of clustering inside a graph is measured by a metric called "clustering coefficient". The clustering coefficient of a graph is defined as the average of every node's local clustering

[1]The longest of all the shortest paths between any two nodes in the graph

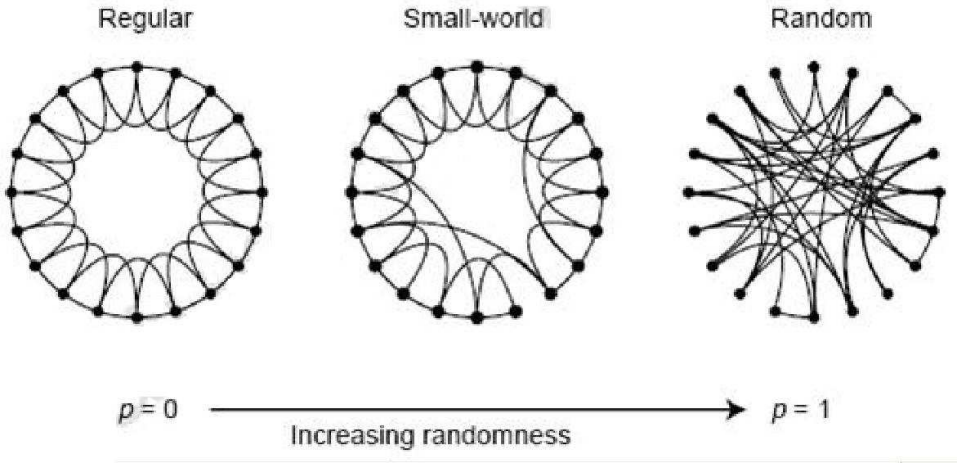

Figure 3.2: p is the probability of rewiring an edge to a random node in the graph. In random graphs, all edges connect random nodes.

coefficient. A single node's local clustering coefficient is the ratio of the number of existent edges among that node's neighbors only, to the maximum number of possible edges, which can exist, between the neighbors. Thus, the clustering coefficient of a node with degree k is:

$$\frac{Number\ of\ edges\ between\ the\ k\ neighbors}{k * (k-1)} \tag{3.1}$$

A sparse random graph has very small clustering coefficient. A clustered, where all edges lead to closeby nodes, has very high coefficient and extremely high diameter. A small-world graph is defined as a graph with high clustering coefficient and yet a diameter comparable to random's. A small world graph is generated by using a regular graph and rewiring a small fraction of its edges to random nodes inside the graph, as in random graphs. This is illustrated in Figure 3.2. Since most of the edges still point to closeby nodes, clustering coefficient remains high. However the rewiring of a few edges is enough to greatly reduce the diameter and bring it close to the diameter of a random graph. This is shown in Figure 3.3. Notice that a dense random graph will have a high clustering coefficient, since there are edges among the majority of all the possible node-pairs.

The important thing to note here is that, in small-world graphs, as noted before, there are not many edges to connect any two "neighborhoods". This means that most shortest paths between nodes of different "neighborhoods" will use the same edge to travel from one "neighborhood" to the other. This is the fact that the horizon criterion that we mention below tries to exploit.

The small-world graphs we used, were generated according to the Strogatz-Watts model. Each node is assigned a number. Two nodes are defined to be close to each other is the difference of their numbers is small. First, each node is connected

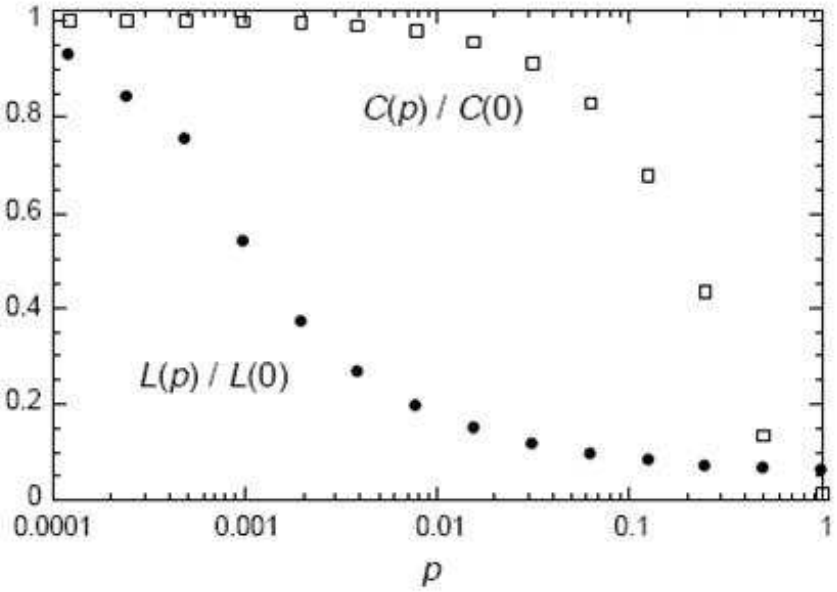

Figure 3.3: The x axis is the rewiring probability. We see that rewiring a few edges to random nodes is enough to greatly reduce the diameter of the graph, without greatly affecting the clustering coefficient.

to the k closest nodes. For any node i, these are all nodes with number i $\pm$ j, where j = 1 to k/2. Then, for each node, we consider only the edges that lead to nodes with higher number, i.e. edges to nodes i + j. For each of these edges, we rewire them to some random node in the graph, with probability p. Setting p equal to 0 generates a completely clustered graph, whereas setting p equal to 1 generates a random graph.

3.3 Related Work

We have yet to find some other work whose main goal is to reduce the duplicates generated during the process of flooding. Today, duplicates are only reduced by the fact that flooding is bounded by the TTL field, which means it covers a small part of the network. Since the paths traveled by the flood messages are short, there is small probability that those paths will form circles and thus generate duplicates. However even this does not hold for small-world graphs as discussed below.

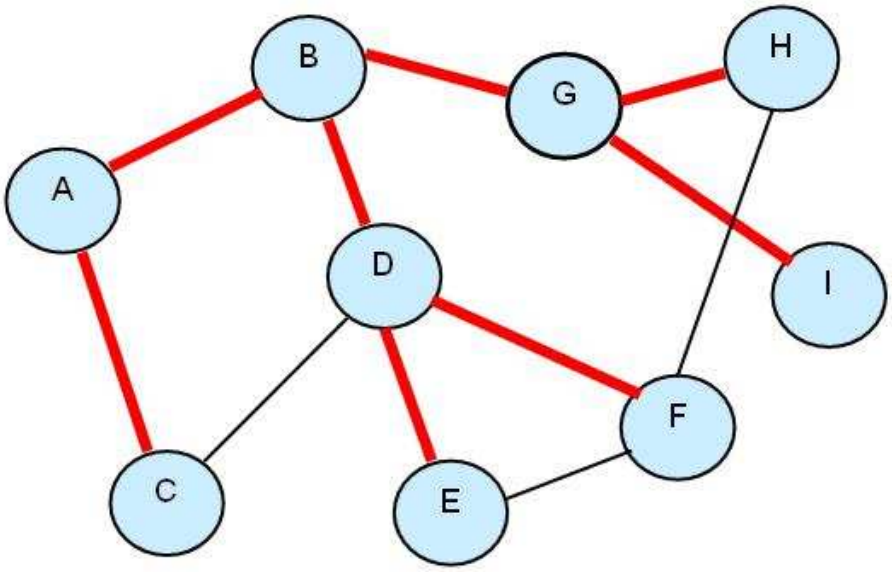

Figure 3.4: Node A's shortest path tree. The red, thick edges form A's shortest path tree.

3.4 Algorithm description and preliminary results

3.4.1 Introduction

During a single flooding process (originating from any node X), if a message traveling over and edge, reaches a node Y, which has yet to receive a message of this flood, this edge is part of X's shortest path tree in that graph of the network. The reason for this is that if there was some other shorter path from X to Y, not including the aforementioned edge, some other message would have traveled that path and reached the node first. On the other hand, if the node that receives the message has already received another message of the same flood, it means that the edge traveled by the duplicate message to reach the node is not part of X's shortest path tree. (See Figure 3.4 for a simple example of a shortest path tree). Even if there are two shortest paths from a node X to a node Y, Y will process the messages that arrive at the same time, sequentially, which means that the path used by the first message to be processed, will be deemed shortest. Any message sent over an edge which is not part of the shortest path tree of the node that initiated the query, will be a duplicate. Each node has a different shortest path tree and this spanning tree does not change, if the network structure does not change.

Since each node has a different shortest path tree, any node that receives a message, must be aware of the identity of the originator node, to route it correctly.

In order to eliminate the duplicate messages during a flooding, each node need not be aware of the shortest path tree of each distinct node X it may receive a

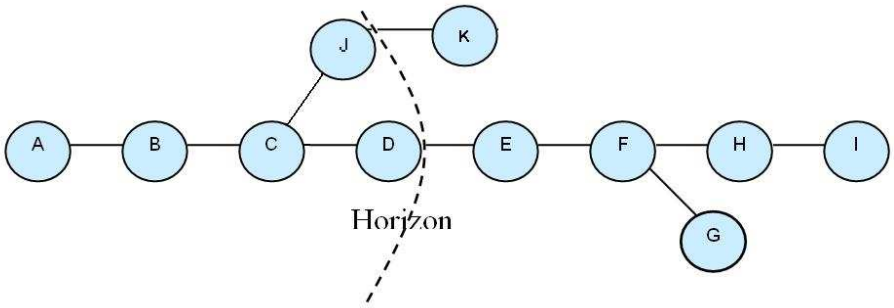

Figure 3.5: Example of Horizon's criterion.
Messages from B to A will be assigned to category B.
Messages from C to A will be assigned to category C.
Messages from D, E, F, G, H, I, J will all be assigned to category D.
Messages from J, K will both be assigned to J.

message from, but rather, which of its edges are part of the shortest path tree of X.

However, this design is also not very scalable in the sense that in requires that each node I hold information equal to the size of the P2P network multiplied by the degree of the node, in the worst case[2]. In the following pages, we describe a scalable version of the above algorithm, with the assumption, for the time being, that the network structure never changes. We present and analyze/explain the efficiency of the algorithm for a wide range of parameters and graphs. The reason for this separation is that we wanted to analyze the efficiency of the algorithm in its ability to correctly construct the spanning trees information, unaffected by changes in the structure since we believe this factor to be orthogonal. Since servents are elected as Ultrapeers/Supernodes only if they usually have a large uptime, each node can benefit, from the information it collects for the topology of the network , for a long time before it becomes stale.

3.4.2 Categories

As mentioned before, in order to construct the local information of every spanning tree in the network, each node I needs to keep a state of size equal to O(N*n), where N is the number of nodes and n the degree of the node. This is because, for each one X of the N different shortest path trees and for each one Y of its n neighbors, I maintains a duplet $\langle X, Y \rangle$ with value either 0 or 1, showing whether edge (I, Y) is part of X's shortest path tree. If I has this information, then when it receives a message originating from node X, it sends the message to neighbor Y only if the value of the duplet $\langle$ X, Y $\rangle$ in I equals 1. X can have N distinct values, whereas Y can have n. This leads to N*n duplets. Since n does not change (each client in

[2]Notice however, that techniques can be used to make the information size in order of the size of the network (N) and not equal to N*n. Still, this information is quite large too

Gnutella and other P2P systems tries to maintain, on average, a constant number of neighbors), it is storing the identity of those N nodes that limits the scalability of the algorithm.

In this algorithm, each message was distinguished based on the originating node, thus leading to N "categories", one for each node in the network. To make the algorithm scalable, we would like to find some other criterion which defines a constant number of categories, regardless of the network size, to distinguish between messages, rather than the node that initiated the flood. Since the number of the nodes may grow arbitrarily while the number of those categories should remain constant or at least grow very slowly with N, this will mean that more than one node will belong to the same category. In the previous algorithm, since there was only one shortest path per category, a distinct value of X represented one shortest path tree. Any one neighbor edge (I, Y) either did belong or did not belong to that tree, and so the duplet $\langle$ X, Y $\rangle$ had value either 0 or 1. However, now that more than one different shortest trees belong to the same category, the value of the duplet is the percentage of the shortest path trees that fall under X category, that edge (I, Y) is part of.

In order for this scheme to work correctly, the criterion should assign in the same category those nodes, all of whose shortest path trees either contain or do not contain the edge from I to Y. This means that more than one category may contain nodes whose shortest path trees, for instance, do contain the edge from I to Y[3], but we wouldn't want to have categories which contain nodes, some of whose shortest path trees contain the edge and some of whose do not[4]. In the case of such mixed categories, since now we distinguish the incoming messages based on the category they belong to and not the node that initiated the flood, we would not know if forwarding the message to node j will cause a duplicate or not. However, even if we do have mixed categories, the categorization will work if most of the traffic (messages) we receive is not assigned (categorized) to those mixed categories, either because not every category is assigned the same number of nodes ("good" categories contain most of the nodes) or some nodes, which belong to "good" categories, generate most of the traffic. There are two ways to distinguish traffic, other than the node it originated from (i.e.: the node that initiated the flood).

3.4.3 Horizon

One of those ways to distinguish messages arriving at node I, is based on the node from which they come from, up to some maximum distance away from I, measured in hops. We call this distance, horizon. If the shortest path from the node that initiated the flood to I is longer than the horizon, we categorize the message as coming

[3]This would mean that we have two categories where we could have just one instead. However, the scalability of the algorithm stems from the fact that the number of categories is constant and not from the number of the categories itself.

[4]I.e.: Some messages coming from those nodes will produce duplicates of sent to j and some will not.

from the furthest node from I, in the shortest path, which is also inside I's horizon. It follows that the distance in hops from us to that node is equal to the horizon. If the shortest path is smaller than the horizon, the node that initiated the flood is within I's horizon and so we categorize the message as coming from that node. It follows that only messages from this node will fall into the category for this node, whereas in the category defined by a node i, which is on the horizon (as opposed to within the horizon) will fall all messages coming from nodes whose shortest paths to us, enter our horizon through node I (Figure 3.5). It also follows that if we use the diameter of the network as horizon, then every node is within the horizon. The number of categories for each edge will be equal to the number of nodes in the network, with one category per node, leading us to the aforementioned, non-scalable algorithm. In all cases, the number of categories created by this criterion is equal to the number of nodes inside (and on) our horizon. In total, the per node state is O(n* number of nodes in horizon).

This algorithm tries to take advantage the degree of clustering in the graph, since, with high clustering, nodes whose shortest paths enter our horizon from the same node, will have the same behavior (i.e.: common shortest paths after the node on our horizon. The paths will differ on the first hops of the flood).

3.4.4 Hops

Another way to categorize the traffic is the distance between I and the node that originated the flood, that is, the number of hops already traveled to reach us. When a message comes from a node X further away, it is more likely that there is another path from X to I's neighbor Y, which is shorter than the path from X to I and on to Y. Using this criterion, we avoid putting in the same category messages that come from further away with messages that come from close by, which will create a mixed category, as described before (2.2). The number of categories created by this criterion is O(radius of the graph). For a random graphs, this is log(N)/(2*log(d)) where N is the size of the graph and d its average degree. Thus, as before, the per node state is O(n*radius of the graph).

This criterion works better in random graphs, where messages on the first few hops almost never produce duplicates. In contrast, almost all of the messages generated during the last hops of the flood, are duplicates. (Figure 3.6). This means that in the experiment shown in the figure, categories for hops 1 to 4 and 6 to 7 will not be mixed categories. Notice that even if we bound a flood with some small TTL, in small-world graphs, most of the messages will be duplicates. On the other hand, in random graphs we won't have many duplicates. More in-depth analysis of this, is provided in the evaluation part, later on.

3.4.5 Horizon + Hops

Since the horizon criterion works well for clustered graphs and the hops criterion works well for random graphs, using those two criterions together should reduce

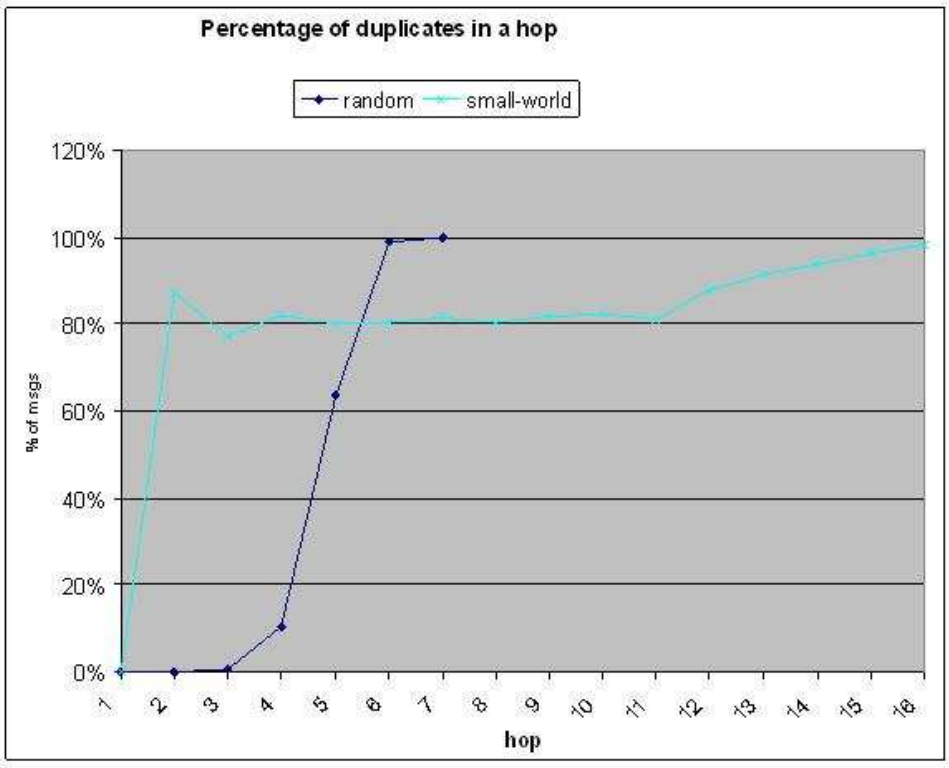

Figure 3.6: Percentage of messages generated at each hop, that were duplicates. Hops 2 to 11 in the small-world graph have an almost constant percentage of duplicates. In contrast, in random graphs, small hops have no duplicates and large hops almost only produce duplicates.

the number of duplicates produced in both types of graphs (Figure 3.7). When using horizon with hops, every category created by the horizon criterion will be divided into more categories, based on the hops criterion. For messages coming from a node inside the horizon (and excluding the nodes on the horizon), the number of hops is always the same and so those categories do not get divided. For instance, in Figure 3.5, messages belonging to category C can only come from node C and so always arrive with hops equal to 2. Categories of nodes on the horizon will be divided into so many categories, as the number of possible hops outside the horizon, that is, at most, the diameter of the graph, minus the length of the horizon. Table 1 shows the categories of node A, based on Figure 3.5. In this simplified example, since we only have one node on the horizon. In general, the number of categories will be at most: (number of node inside the horizon) + (number of nodes on the horizon)*(diameter of the graph - horizon + 1).

Table 3.1: A's categories.

Horizon Criterion Categories	Hops Criterion Categories	Nodes whose messages will fall into each category
B	1	B
C	2	C
D	3	D
	4	E
	5	F
	6	G, H
	7	I
J	3	J
	4	K

3.4.6 Implementation

In order to be able to categorize an incoming message according to the horizon criterion, we need to know the identities of a number of the last nodes visited by that message, equal to the horizon value. This means that each message must contain information of the last hops of its route so far. Gnutella messages already contain information about the number of hops traveled so far. For each category and each neighbor, each node stores two numbers. The first number is the number of messages of that category that were forwarded to that node. The second number is the number of messages of that category that were forwarded to that node and turned out to be duplicates. In order to know which of the messages sent to each node were duplicates, we need to have each of our neighbors explicitly inform us, when it receives a duplicate from us. The ratio of the second number versus the first is the percentage of messages of that category, sent to that neighbor, that were duplicates. If the criterions work well, most of those percentages will be either close to 100 or 0 (i.e.: no mixed categories). Based on some threshold, we can decide to stop sending a certain category of messages, to a certain neighbor, if its percentage exceeds the threshold. To illustrate, let us assume that node I is connected to nodes A, B, C and D. Suppose that the first message I receives, is a message sent by a node well outside our horizon. Let us assume that the message entered I's horizon from node X, and has traveled y hops so far (y $\rangle$ horizon). From node X, the message was forwarded until it reached A and then I. Thus I needs to forward the message to neighbors B, C and D. Suppose we format the categories as such: $\langle$ Horizon node, hops, neighbor, messages sent to that neighbor, duplicate sent to that neighbor $\rangle$. This means that categories $\langle$ X, y, B, 0, 0 $\rangle$, $\langle$ X, y, C, 0, 0 $\rangle$ and $\langle$ X, y, D, 0, 0 $\rangle$ will become $\langle$ X, y, B, 1, 0 $\rangle$, $\langle$ X, y, C, 1, 0 $\rangle$ and $\langle$ X, y, D, 1, 0 $\rangle$. If the message I forwarded was a duplicate for neighbors B and D, after they notify I, it will update the three categories as such: $\langle$ X, y, B, 1, 1 $\rangle$, $\langle$ X, y, C,

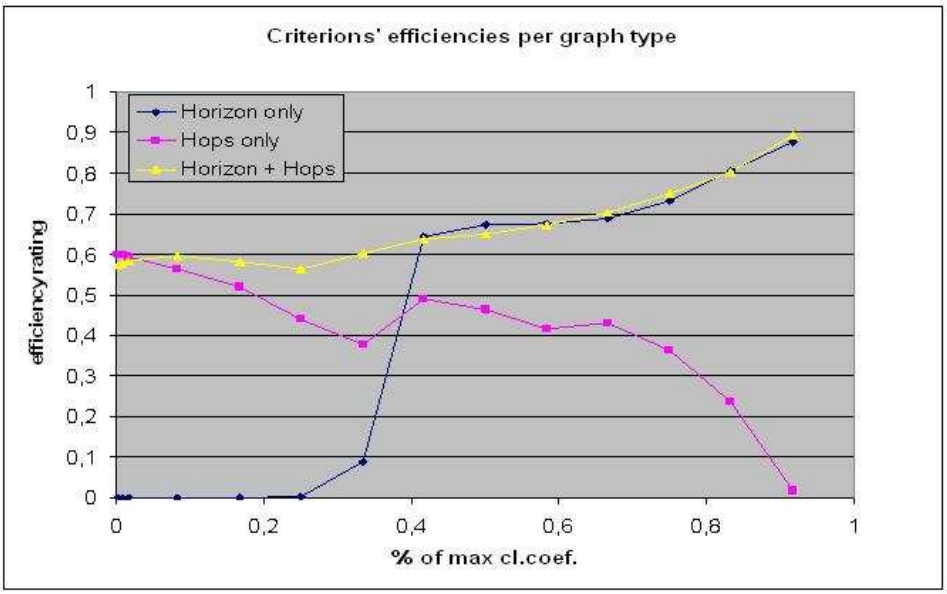

Figure 3.7: Efficiency of each criterion for different Clustering Coefficients. The X axis is the percentage of the maximum clustering coefficient (clustered graph). When the Horizon criterion is used, the horizon distance used is 1.

Table 3.2: Simulation parameters. One experiment per each combination of values of the four parameters.

Horizon	Not used 1, 2, ..., diameter
Hops	Not used, used
Cut-off threshold	75, 100
Clustering Coefficient	0.001, 0.005, 0.01, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55

1, 0 ⟩ and ⟨ X, y, D, 1, 1 ⟩ .

3.4.7 Evaluation of categories

Simulations were conducted using a sP2Ps (simple P2P simulator), developed by us. The parameters of the graphs used were 2000 nodes and 6 degree per node, on average. We conducted experiments for several clustering coefficient values, ranging from 0.001 to 0.6 (the maximum clustering coefficient value for graphs with 2000 nodes and 6 average degree).

For each graph, we run experiments for several parameters of the algorithm. Horizon values ranged from 0 (not using the horizon criterion) to the diameter of the graph, using the hops criterion or not and two threshold values. All the experiments conducted are summarized in table 2.

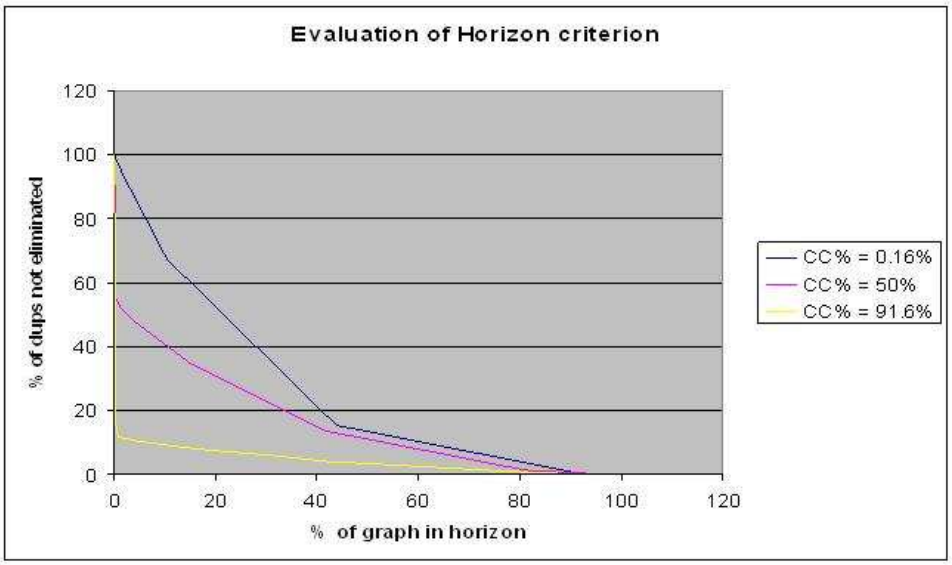

Figure 3.8: Evaluation of horizon. Y axis: % of duplicates not eliminated. X axis: percentage of graph in horizon. One line per clustering coefficient percentage (of max cl.coef.). Threshold is 100.

TTL is always set to infinity, so a normal flooding (without using the algorithm) will reach every node in the graph and generate the maximum number of duplicates. Two metrics are used to rate the efficiency of the algorithm in each experiment. The number of duplicates sent and the coverage of the floods (i.e.: the percentage of the graph covered). Notice that in experiments that use a threshold of 100, the algorithm will eliminate those categories that contain ONLY duplicate messages. Thus, is that case, we have no loss of coverage. However, we do not eliminate that many duplicates either.

For each experiment, we run one flood from each node in the network and collected the average results (coverage, messages sent and duplicates sent) over all floods. Each experiment is named after its parameters. Thus, the name format will be [horizon distance]-[HOPS — NOHOPS]-[threshold value]. For instance, experiment 2-HOPS-75 means that the algorithm used a horizon distance of 2 hops away, used the hops criterion and eliminated all categories with duplicates greater than 75.

Evaluation of Horizon Criterion

Figure 3.8 shows that in random graphs, benefit is closer to the linear curve than in more clustered graphs. In clustered graphs, we have a benefit relative to the extent of clustering, just by using the minimum possible horizon distance value (=1)(See Figure 3.12). What is more, even though the percentage of the graph in the horizon lowers if the size of the graph increases, the extent of the benefit will remain the

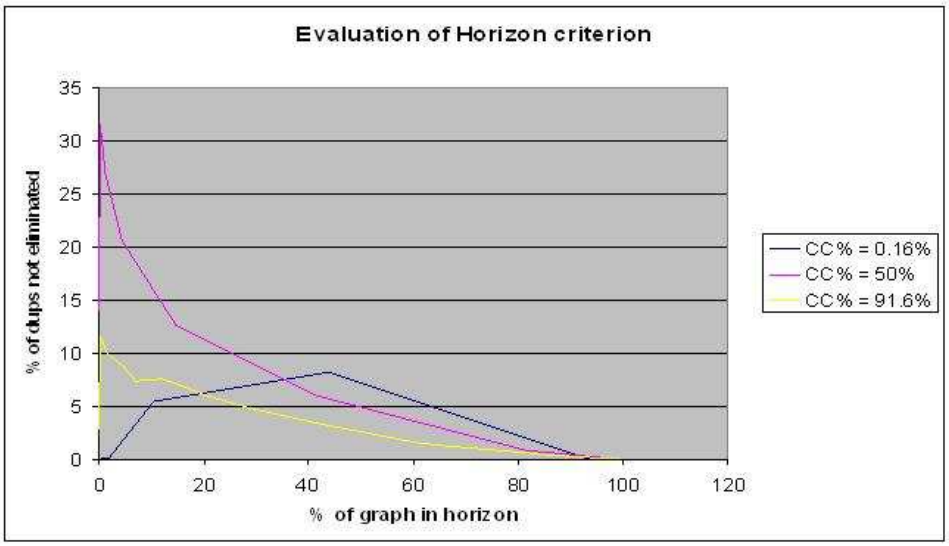

Figure 3.9: Same figure with Figure 3.8 but with cutoff threshold 75%. Y axis: % of duplicates not eliminated. X axis: percentage of graph in horizon. One line per clustering coefficient percentage (of max cl.coef.).

same, since in sparse graphs, the clustering coefficient does not change with size. In contrast, in random graphs, if the size of the graph increases, one would need to increase the number of nodes in the horizon to maintain the same percentage of nodes in the horizon, and thus the same efficiency. Notice that, in these figures, we have used a threshold of 100 and thus have no coverage losses.

Figures 3.9, 3.10 and 3.11 show the algorithms efficiency, when it uses a threshold of 75. Notice that in with a threshold of 100%, in random graphs, the algorithm eliminated almost no duplicates, when the horizon was small. In this case, almost all the duplicates are eliminated, along with all the coverage of the algorithm. This means that the horizon criterion still does not work well for random graphs. This is shown in Figure 3.11, where we combine both metrics (coverage and duplicates eliminated) in a single, simple metric. If C is the coverage percentage and D is the percentage of the duplicated that were eliminated, this metric (rating) is defined as C^2*D. Notice how this is similar with the inverse of Figure 3.8.

Figure 3.13 shows again the efficiency of horizon = 1, like Figure 3.12, but again this time with a threshold of 75%. Notice that the efficiency rating is not linear this time with the extent of the clustering. This is because the 75% threshold is not the optimal for every clustering coefficient. We have developed a modification of the algorithm to allow for the optimal threshold to be computed at run time, instead of being an algorithm parameter and will update the above results as soon as possible. Then we believe we will again have a efficiency curve relative to the extent of the clustering.

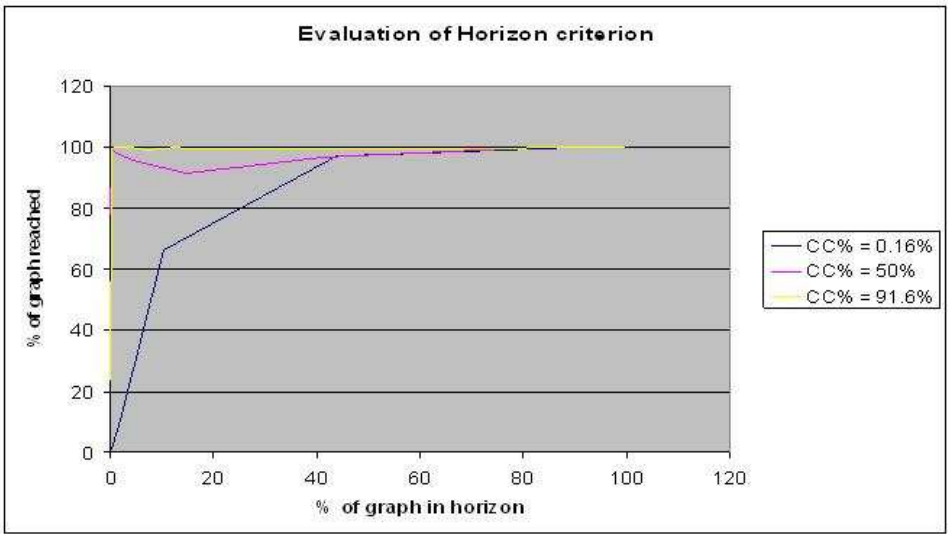

Figure 3.10: Coverage of the algorithm with cutoff threshold of 75%.

Evaluation of Hops Criterion

Figure 3.14 shows results of experiments using the hops criterion but not using the horizon criterion (horizon = 0), with different graphs. As mentioned before, the horizon criterion works better for random graphs. This efficiency is the same for any graph size. The number of possible hops do increase, if the size of the graph increases, albeit much more slowly (Recall that the diameter of a random graph is log(N)/(2*log(d)) where N is the size and d the average degree). In contrast, if N increases in random graphs, for the horizon criterion to maintain its efficiency, in needs to maintain the percentage of graph in the horizon. Thus if N doubles, so must the number of nodes in the horizon. This is not so however in the case of small-world graphs, as we have seen above. The horizon criterion takes full advantage of the clustering of the graph by using a horizon distance of 1 hop, regardless of the percentage of the nodes of the graph in that horizon. Thus we can see that both criterions are scalable.

Efficiency of both criterions.

In Figure 3.15, we see the efficiency of the algorithm that uses both criterions. As mentioned before, the algorithm that uses both criterions works well in any type of graph. We believe that by using a way of calculating the optimal threshold at run-time, as mentioned before, would lead to even better results in the case of random graphs, thus making the above lines more flat.

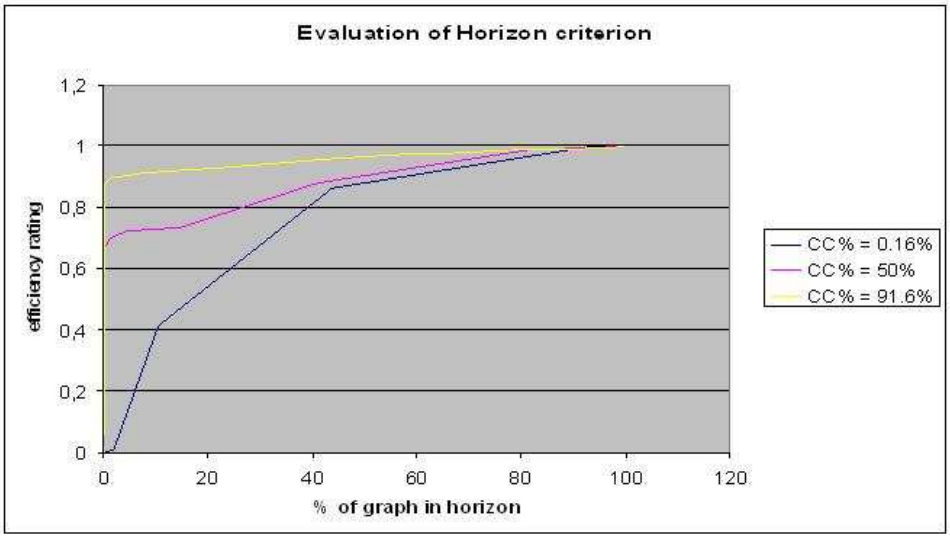

Figure 3.11: This figure cobines results of Figures 3.9 and 3.10.

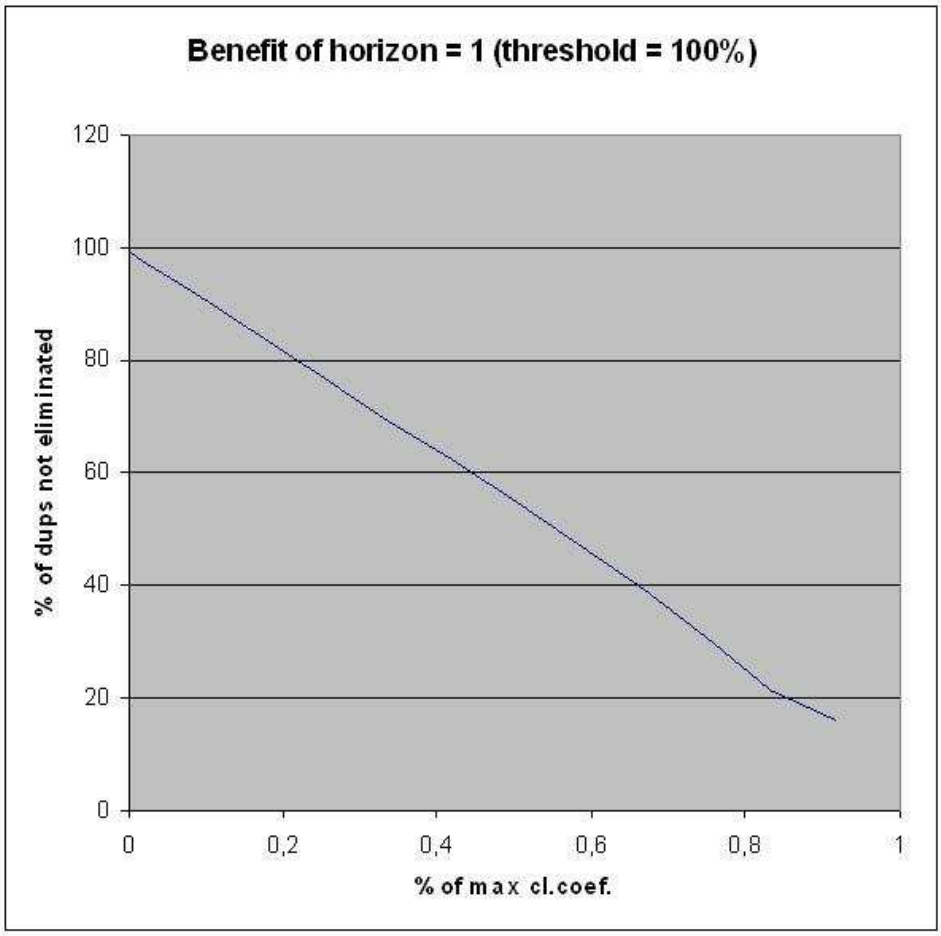

Figure 3.12: Graph of benefit of horizon =1 from clustering extent. Hops was not used and threshold was 100, so there is no coverage loss.

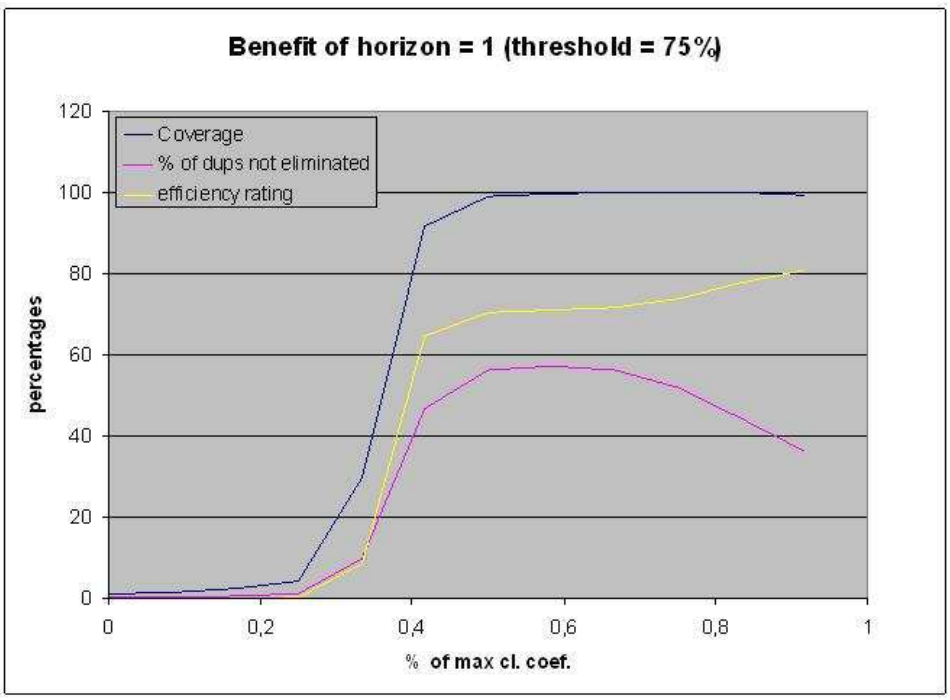

Figure 3.13: Same with Figure 3.12. However, in this case while we still do not use Hops, we use a threshold of 75% instead of 100%, which leads to coverage losses too.

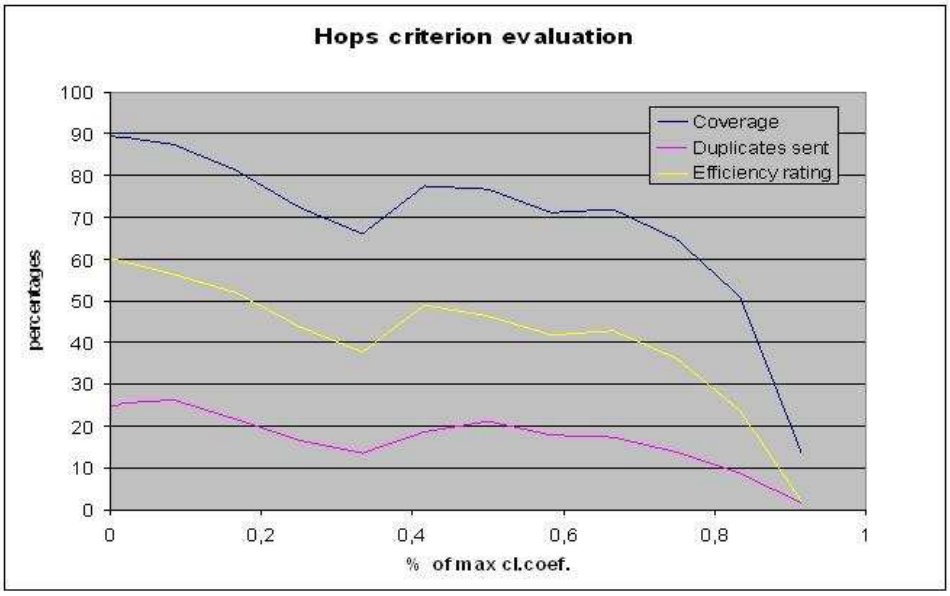

Figure 3.14: Evaluation of hops. Y axis: percentage of duplicates not eliminated. X axis: clustering coefficient percentage.

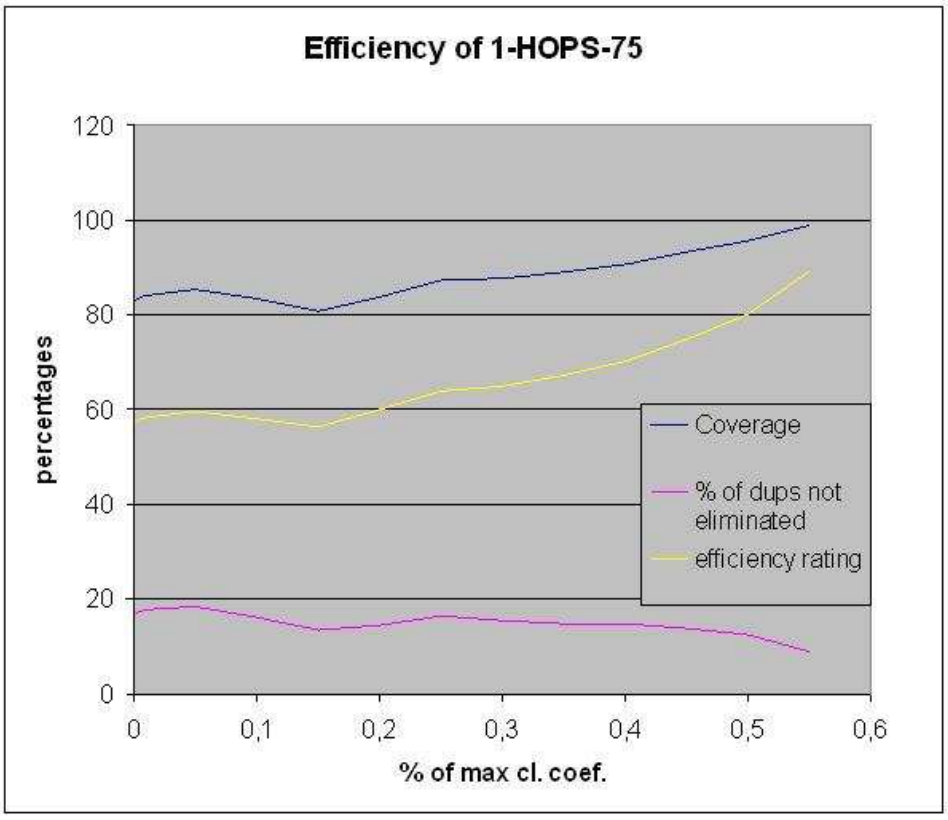

Figure 3.15: Efficiency of 1-HOPS-75.

Chapter 4

Flood Driving algorithms - Divide and Conquer

In this chapter, we describe the proposals for adding such semantic information to the network, as to enable us not to avoid sending duplicate messages, but to avoid sending messages altogether, to nodes which most surely will not contain the piece of data we are looking for.

4.1 Problem description

As mentioned before, the flooding mechanism becomes scalable by using the TTL field, at the expense of greatly reducing coverage and thus making locating less popular items (items that are stored on very few number of nodes in the network) very difficult. The only way to increase coverage by using the same amount of messages, is to ensure that every message reaches a new servent (i.e.:no duplicates) and thus, that we do not waste messages. However, the goal of reaching as many nodes as possible is a consequence of the fact that every node has the same chance of containing the piece of data we are looking for. If the information we are looking for is popular, (i.e. is replicated to many nodes in the network), flooding will locate it quickly (i.e. even with a small TTL). If however, the information resides at just one node, for instance, flooding would have to reach almost every node in the network to locate it. This means that in unpopular searches, a lot of bandwidth is wasted contacting nodes that do not have the information we need. If there was some way of knowing which servents are less likely to contain the information, we could use some way to avoid wasting messages by sending them to those servents.

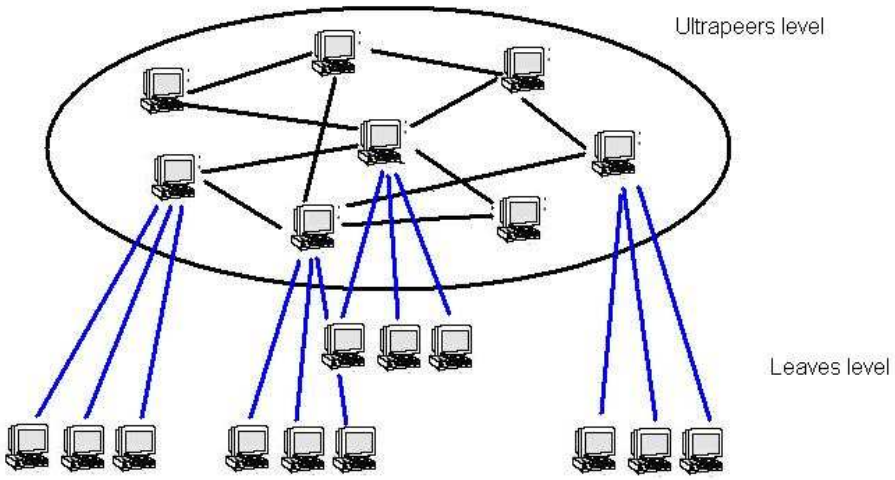

Figure 4.1: Ultrapeers/Supernodes architecture. Black lines correspond to connections between Ultrapeers. Blue lines correspond to connections between Leaves and Ultrapeers. Flooding is performed only at the Ultrapeer level.

4.2 Existing approaches

4.2.1 Directed Breadth First Search

The first approach to make blind flooding more efficient, was Directed Breadth First Search (DBFS). The flooding mechanism is often referred to as BFS, because it traverses the graph in a BFS manner. DBFS avoids propagating each message to all the neighbours. Instead, each node rates its neighbours according to some metric. The metric mostly used is the ratio of the results received from that neighbour, versus the number of searches propagated to that neighbour (recall that results are propagated to the node that initiated the search in the reverse path travelled by the search). Each node propagates each message (flood) it receives to the top-K neighbours, so that k/degree equals some percentage. This approach enables a node to have the same number of results by using an even smaller TTL, which reduces the number of messages for popular searches even more. However, this approach makes finding unpopular data even more difficult, since it may reduce the coverage of a search.

4.2.2 Ultrapeers/Supernodes

The most important approach, so far, in addressing the problem, is the introduction of Ultrapeers in the Gnutella 2 network [Ultrapeers] and Supernodes in the FastTrack network. Ultrapeers' purpose is two-fold. First, it removes all nodes

with low bandwidth from the network on which flooding is performed. Flooding generates a lot of messages and was slowed down by nodes with low bandwidth, which comprise the majority of the users and were not able to route the increased traffic. Nodes with high bandwidth are elected Ultrapeers. The rest of the nodes are called Leaves, and connect to one to three of the Ultrapeers. There are no connections between leaves (Figure 4.1). Every Ultrapeer contains a hashed index of the files residing on each of the leaves connected to it. A leaf can initiate a search by forwarding it to its Ultrapeer. The Ultrapeer floods the search to the rest of the Ultrapeers. Any Ultrapeer forwards a query to one of its leaves only if the index of its leaf shows that it may contain some file(s) that satisfy the query. This means that i)leaves do not propagate messages (they are only the end points of a flood) and ii) leaves will only receive those queries that there is a high probability that they can satisfy. The hash index an Ultrapeer receives from each one of its leaves, allows it to know if the leaf cannot satisfy the query, most of the time. This index uses Bloom filters, thus false positives are possible, but not false negatives.

4.2.3 Semantic Overlay Networks

Crespo and Garcia-Molina propose a thematic partitioning of the network [2]. They propose some mechanism to automatically thematically, classify music files, based on their filenames (which must contain information about the artist and the title) and by using an existent online, categorized, index of songs. They manipulate the extent of granularity in the classification, in order to balance the population of each category in the classification. Then, they propose the partitioning of the network into subnetworks, according to the categorization. When searching for a file, one needs to classify it and issue a query in the appropriate subnetwork. However, they do not define how one can contact the appropriate subnetwork or whether everyone is aware of every subnetwork.

4.3 Algorithm Description

SecSPeer adopts the basic idea of the SON proposal and defines i) how the semantic overlay networks are formed and ii) how search is performed through those networks.

The idea in the core of our design is the partitioning of the Gnutella network in semantically independent subnetworks. This partitioning will be based on some categorization of the files in the network. Each subnetwork shall be comprised only of servents that contain a certain number of a certain category of files. For instance, one subnetwork can be comprised of nodes that contain at least one rock song, or perhaps at least five rock songs. The networks are semantically independent in the sense that two servents in the same network are only connected to one another through a number of servents in the same network. However, it follows from the above descriptions that a single servent can participate in more that one subnet-

works. When this is the case, that servent maintains a different set of connections for every subnetwork it is a part of.

The categories mentioned above, each of which will be the basis for a distinct subnetwork, must be defined so that each category must not contain a large portion of the total number of servents. If some category is large, it can divided it into subcategories. On the other hand, each category's population must not be smaller in size than a certain percentage of the total number of servents. The reason for this is explained below.

Since Ultrapeers/Supernodes are considered to contain not only their own items but also those of their leaves, their files would belong to many different, categories. In the extreme case, every Ultrapeer would belong to every subnetwork, thus either ruining every benefit of the scheme or requiring of us to have very fine-grained categories, in order to exclude some Ultrapeers from some subnetworks. Instead, for the time being, we assume that there are no Ultrapeers/Supernodes. We shall add those later on in the design since they are a very important factor in the scalability of unstructured P2P systems.

4.3.1 Formation of the subnetworks

Each servent can obtain the categories definitions from a well-known source. Since those categories will change very slowly (perhaps once a year?) this does not limit the scalability of the system. Even now, most of the unstructured P2P systems use web-caches to learn the location of at least one peer to connect to. Using the categories definitions, each servent decides which subnetworks to connect to, based on the items it contains. This of course requires some way of knowing for each data item, the category it belongs to. This, for the time being, can be done manually. The client might present the user with a list of all the categories and ask to pick one for each item. This information will be stored by the client and be provided to every other client, which downloads that item, so that its user need not categorize it again. This way, only items injected in the system through external sources (CDs, downloading through other applications, etc) will require categorization.

Connecting to a subnetwork can be done the same way servents connect to the p2p networks today. A webcache can be maintained for each of the subnetworks defined by the categories.

4.3.2 Searching the subnetworks

The reason of the creation of those subnetworks was to be able to initiate a search via flooding inside the right subnetwork and thus avoid sending the message to servents of other subnetworks. However, since having each servent be part of every subnetwork beats the reason for having subnetworks, it follows that if a servent wants to initiate a query in a subnetwork it is not a part of, it needs to locate at least one servent participating in that subnetwork. This is the reason we require all sub-networks be small (so that rare items can be locatable), but not TOO small (so

that sub-networks can be locatable). Each servent initiated a flood query in each of the subnetworks it is a part of, sequentially, looking not for the required item, but for a node belonging to a particular subnetwork. When it locates one, it sends a query to that servent, requesting the particular item it is looking for.

4.3.3 Subnetwork size

We said before that subnetworks must be both small but not quite small. The question is if there is a size that will be small enough to enable us to locate rare items, but large enough to enable us to locate at least one servent in a subnetwork. Since locating just one servent of a certain subnetwork is enough, it can easily be shown that we can have as small subnetworks as to enable us to exclude the vast majority of the total servents and still be very easy to locate them. However, the fact is that some peers contain lots of items. Thus, subdividing a category a lot of times gives diminishing returns in turns of subnetwork size reduction and also would require several servents to maintain a very large number of open connections. Here, implementations need to find a trade-off between categories granularity and number of connections. Notice that servents that have lots of items generally sport much higher bandwidth than the average servent.

What is more, when a servent, during a search, locates a servent of the subnetwork it wants to search, it can store its IP address and port in a cache, for future use, thus eliminating the need to lookup that subnetwork in future, searches. It is obvious that this can be extended to store more than one servents for each subnetwork it has contacted (or only each subnetwork it contacts frequently, to reduce required space). This way, it can periodically contact one of them to learn of new servents in the network and update the cache entry. By using this scheme, subnetwork lookups can be all but eliminated, since when a servent performs a subnetwork lookup, it can be satisfied by its neighbors, without the flooding even reaching a single servent of that subnetwork.

4.3.4 Re-introducing Ultrapeers/Supernodes

Ultrapeers can be added on a per subnetwork basis using the same mechanism used today to elect Ultrapeers in P2P networks. The difference is that each Ultrapeer's leaves will be nodes from the same subnetwork. What is more, leaves which belong to more than one subnetworks will connect to as many Ultrapeers as the number of subnetworks they belong to. This is not a problem since leaves may then have more connections (than today) and yet receive the same number or even less queries. Each leaf will send a subset of its index to each Ultrapeer, containing only the items that belong to the category the Ultrapeer belongs to. A single servent belonging to more than one subnetworks may be elected as Ultrapeer in more than one of them.

Chapter 5

System Security

5.1 Known threats in unstructured P2P systems

Unstructured peer-to-peer systems have no central authority to maintain the overall system's behavior. Thus, each participating peer has the freedom to send or route messages in its own will, abusing the system's rules. That is, each peer can construct messages with fake TTL and HOPS fields pretending that it routes actual traffic. There is no limit for the traffic each peer generates. The faster and well connected peer wins. Whenever a peer responds with a message there is no easy way to judge if the response is valid; that is if the response contains valid payload. For example, consider an unstructured P2P system for exchanging Stock Market's related information. Each peer is assigned with a share. When a peer asks for the current price of a share it generates a Query message, which propagates in the system. In normal circumstances, the peer responsible for the share in question should reply with the true share's price. However, since there is no central authority to coordinate the system a number of different things can happen. A peer may respond in favor of another, a peer may respond with a fake price, a peer may respond with the price of another share, etc.

All the above, can lead to a number of different attacks from malicious peers that target the system or even computers outside the system.

5.2 Related work

Daswani end Garcia-Molina[3] propose a number of strategies for Query traffic balancing in the Ultrapeer level, so as to limit Query flooding attacks from malicious peers. The results have been exported after modelling the Gnutella traffic.

Mayank Mishra[8] describes extensively a number of existen attacks in peer-to-peer systems and he proposes a new protocol, Cascade. One of the main features of Cascade is iterative search. In iterative search, a peer controls the Query flow. In contrast with pure flooding, iterative search forwards the Query to a peer's neighbors and requests the neighbors of each neighbor. Then, it procceeds on connecting

to them and perform recursively the Query.

Demetrios Zeinalipour-Yazti[13] describes in detail the Spam generation in Gnutella and proposes each peer to perform a direct connect to the peer it wants to download from, via the system protocol and not the download protocol, requery the peer and then perform the download.

5.3 Spam generation

A known malicious behavior, observed in the Gnutella system, is the generation of Query Replies for each Query received by the malicious peer. That is, a malicious peer can monitor every Query packet which is routed to it, parse its Search Criteria and produce a Query Reply packet with imaginary embedded responses. The responses are created by adding a known file extension to the original Search Criteria and by performing a type of frequently used capitalization. For example, if someone searches for 'foo', the malicious peer can respond with imaginary filenames such as 'foo.mpg', 'F O O.mp3' and so on. Although, the responses have imaginary filenames, the files have a valid content, which is an advertisement message[1].

The most vital side effect of the spam generation is the limitation of valid responses. The life time of a typical Query in the Gnutella system does not depend only in its TTL field, but also in the generated Query Responses. That is, an Ultrapeer, which has generated the Query (either by explicitly asking the system for itself, or implicitly asking the system on behalf of one of its Leaves), will likely terminate the querying process, upon it receives a certain amount of results. Now days, in the Gnutella system, the amount of responses per query is upper bounded by a limit of 150-200. Thus, the spam generation can terminate implicitly the querying process, since spam responses are counted as valid responses. This phenomenon can be exaggerated when the Query targets rare content in the system. That is, the Ultrapeer which controls the querying process will likely need to search in a larger horizon with larger possibility to forward the query to a malicious peer. Recall, that one malicious peer is enough to produce hundreds of spam responses, since it can spoof the HOPS field and pretend that the produced responses are generated by another Ultrapeer or by one of its Leaves.

5.4 DoS and DDOS attacks

Another potential security threat in unstructured peer-to-peer systems is Denial of Service (DoS) attacks. A malicious peer can generate random Query traffic; that

[1] A similar malicious behavior, which can be observed rarely in the Gnutella system, is the production of replies with filenames that are constructed with a spam message. For example a Query for 'foo' can issue a Query Hit which embeds a filename 'Free-Movies-AT-www.xxxxxx.org.mpg'. However, this approach is easily captured by the end user, since the spam message is shown in the filename.

is Queries with random Search Criteria, HOPS and TTL fields. Since, each query floods the system via the traditional flooding mechanism, or portions of the system via dynamic querying, a peer can emit to the system unnecessary message traffic, which will eventually grow following exponential rates.

A second type of DoS attack, which nature is completely distributed, can be achieved by emitting Query Responses instead of Queries. This method may target any machine, which listens to a known port and it is connected to the Internet. There is no need for the target machine to be a Gnutella participant. A malicious peer can monitor the Queries it receives and generate responses for every Query message. Each Query Reply packet will carry the IP address and the Port of the target machine. Since, there is no mechanism to indicate if the IP address in the Query Reply message matches the IP address of the machine, which generates the Query Reply message, all generated responses will be routed to the original queriers. Thus, there is a chance that a vast amount of download attempts to a single computer may be performed in a short time of period.

5.5 Existing approaches

In practice, empirical solutions are used in the modern Gnutella system, in order to reduce the impact of the attacks explained above. However, there is no global mechanism to completely solve all the Gnutella security issues.

5.5.1 Spam Generation solution

End users may ban a specific peer, usually via an option in the user interface of the software they use to participate in the Gnutella system, if its responses seem to be spam messages. However, not all spam messages can be easily identified by the end user. Another approach, which is performed mainly by developers of Gnutella servents, is to occasionally scan the network for malicious peers and store them to a public database (hostiles.txt). Each software vendor updates its local database and thus minimizing the chance of contacting malicious peers. However, malicious peers can change their identity quite often and re-enter to the system. On the other hand, scanning the whole network for malicious peers is a slow process and the database updates must be distributed in some way to the participants, which is an even slower process. Updates are published to well known Web sites, or get included to newer software releases.

5.5.2 DoS and DDoS attacks solution

As far as the DoS and DDoS attacks in Natal are concerned, the only way to limit the chance of the attack is via bandwidth limits. Each peer maintains a packet queue adjusted to it incoming bandwidth and eventually drops messages that exceed its queue length in a short time of period. That is, a peer that tries to emit a vast amount of traffic in the system, it will likely get muzzled by its neighbors.

However, this technique can only reduce the traffic emitted by a peer. It can not judge if the peer is malicious or just a fast connected peer with a high degree, nor it can prevent unnecessary message broadcasts in the system. In addition, there is no fair criterion to decide of what Query to drop. That is, Queries generated by malicious peers may dominate in a peer's queue forcing the peer to drop legal Queries.

5.6 Algorithms description

5.6.1 Detecting and preventing spam

As described above a malicious peer that generates spam messages will eventually reply to every Query it receives. That is, the malicious peer will reply to random Queries with a payload consisted of a random combination of search criteria (strings that are not likely to express something meaningful). We propose a strategy in which a legal peer queries with a random Query of TTL=1, at a random time period upon handshaking with a new node, the new node it handshaked with. If it receives a reply for the random Query then it should drop the connection. This strategy has local effect since the decision is taken by one peer and the action regards to the termination of a single connection. We believe that it can have a global effect if the strategy is followed cooperatively by all legal peers of the system. On the other hand, if a legal peer misjudge another legal peer and treats it as a malicious (in the extreme case, where a random string is meaningful) the misjudged peer will loose only a connection. It is rather unlikely that all other peers will treat it also as a malicious one. Further more, the strategy costs only a Query message, which will not be propagated in the system since it has TTL=1.

On the other hand, malicious peers may try to detect the strategy by (a) not responding to Queries with TTL=1 and HOPS=1; that is Queries originated by one of their neighbors (b) not responding to Queries for the first few minutes (c) hide over a legal peer, part of the malicious infrastructure (that is, the legal peer does not follows our strategy), which serves as a gateway to the Gnutella system. Our strategy can be enhanced in order to overcome (a) and (b) by making the HOPS field of the random Query message also random and by requerying peers in random time intervals, respectively. As far as (c) is concerned, we may issue also Queries with TTL=2. If a response of a random Query with TTL=2 is received, then we can safely judge the neighbor as a legal client that hides a malicious one and the connection should be dropped. Someone may argue that dropping the connection in this case leads to a misjudge, since the originator of the response is not known in advance. We can further argue that there is no misjudgment, because if the legal peer was not part of the malicious infrastructure there will not be responses in random Queries with TTL=2. That is, we assume that the legal peer will have been following the original strategy and drop the connections with malicious peers in his first level neighbors.

Peers that have been judged as malicious enough times[2] should be classified as malicious. That is, their IP address should be added to the black list database. Furthermore, legal peers, which route Query Hits should parse the packets and check if a black listed IP is contained. If the check is true the packet should be dropped.

5.6.2 Detecting and preventing DoS attacks

Our spam preventing strategy with random querying could be also used to prevent DoS attacks of the second type, where peers respond to every Query with results that contain the IP address of the target machine. The peer that originates the attack will also answer the random Query and hence will be judged as malicious.

Furthermore, we introduce a complete load-balancing solution based on 'coupon exchanging' between peers, which attempts to prevent Query flooding based attacks[3]. Our load-balancing algorithm prevents peers from generating enormous Query traffic and thus DoS attacks.

Assume two peers, A and B. B establishes a connection with A. Upon handshaking A assigns N coupons to B. Whenever a new Query is sent from B to A, B looses a coupon. Whenever a new Query is sent from A to B, B earns a coupon. That is, if B tries to flood the system via A with a massive amount of Queries, eventually B will exhaust all of its coupons and it must wait to receive new Queries from A, in order to start sending again. Notice, that coupons are exchanged only when Queries are exchanged. This is because a malicious client could issue popular Queries, which generate a vast amount of results and earn a lot of coupons by loosing a single coupon; that is by issuing a single Query.

This load-balancing scheme has also local effect. Coupons are exchanged via direct connections and do not propagate to the system. However, we believe that if the majority of the system follows our algorithm, DoS attacks based on Query flooding will eventually stopped by peers which will get out of coupons.

[2] In the present phase 'enough times' is a number in the order of 10.

[3] Notice also that our duplicate elimination algorithm limits Query traffic and thus has as a side effect the reduction of the scale of a DoS attack based on Query flooding.

Chapter 6

Conformance of Requirements

6.1 Scalability

Unstructured systems are already scalable in terms of maintanance costs, at the cost of the scalability of the searching mechanism. The reason that hinders unstructured P2P systems' scalability is the cost of flooding in messages. Both the proposals of Chapters 3 and 4 address this issue.

6.2 Quality of Service

Up to now, unstructured P2P systems could be alternativly defined as: "Scalability and Quality of Service. Pick one". The reason is that scalability was achieved with the TTL limit in the expense of the ability of the system to locate items. Chapter 4 addresses this issue. It enables

6.3 Expressiveness

Today's substring match based search may lead to presenting as results items of data which are semantically irrelevant to the query. This was somehow rectified by allowing the user to specify the type of data required, as defined by the extension of the filename. The semantic categorization of the data described in Chapter 4 allows the user to be much more specific about the data being looked for.

6.4 Availability

Availability is related to the degree of tolerance of a peer-to-peer system against faulty or unreachable nodes. A peer-to-peer system must be resilient to clients that do not respond to requests or misbehave.

We have described in detail, in the Security Chapter, algorithms, policies and techniques in order to isolate malicious nodes. That is, our proposed system can

remove by itself nodes that may produce internal damage to it and thus gurantee its long time availability.

6.5 Shielding the peer-to-peer infrastructure

We have described in detail, in the Security Chapter, the way a peer-to-peer system can be used to launch DDoS attacks over the Internet. Using the ”random querying” mechanism, healthy nodes of the system can identify the malicious ones and remove them from the system.

References

[1] Christopher Rohrs Anurag Singla. Ultrapeers: Another step towards gnutella scalability, http://rfc-gnutella.sourceforge.net/proposals/ultrapeer/ultrapeers.htm. Technical report, Limewire LLC.

[2] Arturo Crespo and Hector Garcia-Molina. Semantic overlay networks for p2p systems. Technical report, Google Technologies Inc, Stanford university, 2003.

[3] N. Daswani and H. Garcia-Molina. Query-flood dos attacks in gnutella networks. In *ACM Conference on Computer and Communications Security*, 2002.

[4] Napster Inc. Napster, http://www.napster.com.

[5] Sharman Industries. Kazaa, http://www.kazaa.com.

[6] E.P. Markatos. Secspeer deliverable 1.1: System requirements. 2004.

[7] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of IPTPS02, Cambridge, USA, March 2002. http://www.cs.rice.edu/Conferences/IPTPS02/*, 2002.

[8] M. Mishra. Cascade: an attack resistant peer-to-peer system. In *the 3rd New York Metro Area Networking Workshop*, 2003.

[9] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[10] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[11] R. Manfredi T. Klingberg. Gnutella 0.6 specification, http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html.

[12] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of small world networks. In *Nature, Vol.393, 4 June 1998*, pages 440–442, 1998.

[13] D. Zeinalipour-Yazti. Exploiting the security weaknesses of the gnutella protocol. Technical Report CS260-2, Department of Computer Science, University of California, 2001.