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:)

D4.1 “Deployment - Evaluation”

Abstract: This document describes the deployment and evaluation of the scalable and secure unstructured Peer-to-Peer system described in Deliverable 3.1: ”System Implementation”.

Contractual Date of Delivery	29 June 2006
Actual Date of Delivery	23 October 2006
Deliverable Security Class	Public
Editor	Charalambos Papadakis, Elias Athanasopoulos
Contributors	Evangelos Markatos, Paraskevi Fragkopoulou, Alexandros Labrinidis, Dimitris Tsigos

The SecSPeer Consortium consists of:

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

Contents

List of Figures

Chapter 1

Introduction

This document comprises the report for the Deliverable 4.1: System Deployment and Evaluation, of the SecSPeer project. The purpose of this document is to describe the evaluation of the system described in Deliverables 2.1: System Design and 3.1: System Implementation. The structure of this document is as follows:
Chapter 2 provides a description and analysis of the changes in the system imposed by the real world deployment of the system.
Chapter 3 provides an overview on the architecture of the system.
Chapter 4 provides details on the evaluation of the scalability mechanisms of SecSPeer.
Chapter 5 describes the evaluation of the security mechanisms of SecSPeer.
Chapter 6 provides some conclusional remarks. The document ends with the Bibliography.

Chapter 2

Changes in the SecSPeer Architecture

2.1 Introduction

As one can see from the previous two Deliverables [7], [8], SecSPeer is based on the partitioning of the search space to reduce the cost of the search for some particular item. This reduction of the search space is accomplished by dividing the peers in the system in subnetworks, depending on some categorization of the shared content. One of the requisites of any categorization criterion to be used is the fact that it creates roughly equal subnetworks, in size. In Deliverable 3.1 was described the use of the character encoding used in the filenames as a categorization criterion. Although this criterion has several advantages, during the evaluation it proved to have a drawback that affected the efficiency and the scalability of the system. This disadvantage was the fact that the Latin-based (particularly English-based) content and queries consisted of the largest part in the system. For this reason, the Latin-based subnetwork was roughly, in size, equal to the original network, thus offering no improvement in the cost of Latin-based queries, which constituted the majority of them.

2.2 Changes in the architecture

The change implemented in the SecSPeer architecture aims at producing subnetworks, which will be roughly in size. As described in Deliverable 2.1: System Design, the design of SecSPeer remains the same and is based on the partitioning scheme based on some disjunctive criterion. The implementated functions described in Deliverable 3.1: System Implementation also remain all the same, with the exception of the *getCategories()* function, which is replaced by a uniform hash function described in the next chapter. In essence, the categorization scheme described in the next chapter characterizes keywords instead of files.

Another change, which will be analyzed in more depth in the next chapter, is that each peer may belong to all subnetworks. This does not mean that the size of each subnetwork will essentially be equal to the size of the original network. This is because SecSPeer employes a two-tier architecture similar to the one of the FastTrack network [2]. Thus, each subnetwork is comprized of Ultrapeer and not Leaf peers. Each Leaf peer participating in a certain subnetwork simply connected to some Ultrapeer of that subnetwork. This means that the only requirement of a Leaf connecting to all subnetworks is to connect to a number of Ultrapeers equal to the number of subnetworks, which is trivial for any peer even for a number of subnetworks in the order of tens.

Chapter 3

The Partitions Scheme

3.1 Introduction

The aim of the partitions scheme is to improve the partitioning scheme by defining subnetworks, which are, roughly, comparable in size. The proposed method partitions the Ultrapeer overlay network into distinct subnetworks. A novel index splitting technique is further employed. In general unstructured P2P networks are indirectly supplied with some information about the possible location of each resource. Using a simple hash-based categorization of keywords the Ultrapeer overlay network is partitioned into a relatively small number of distinct subnetworks. By employing an index splitting technique each Leaf peer is effectively connected to each different subnetwork. The search space of each individual flooding is restricted to a single partition, thus the search space is considerably limited. This reduces the overwhelming volume of traffic produced by flooding without affecting at all the accuracy of the search method (network coverage). Preliminary experimental results demonstrate the efficiency of the proposed method.

3.2 Background Information

In order to better understand the partitioning scheme, we shall describe a few techniques used in unstructured P2P systems today.

The first such technique is 1-hop replication. One-hop replication dictates that each peer should inform all of its immediate neighbours of the files it contains. Using this information during the last hop propagation of a request at the Ultrapeer level, the request is forwarded exclusively to those last hop Ultrapeers that contain the requested file. One-hop replication reduces number of messages generated during the last hop of flooding [4]. However, the traffic generated during that last hop constitutes the overwhelming majority of the traffic generated during the entire flooding. Simple calculations show that 1-hop replication requires d times fewer messages to spread to the whole network compared to naive flooding, where d is the average degree of the network (average number of connections for each Ultrapeer).

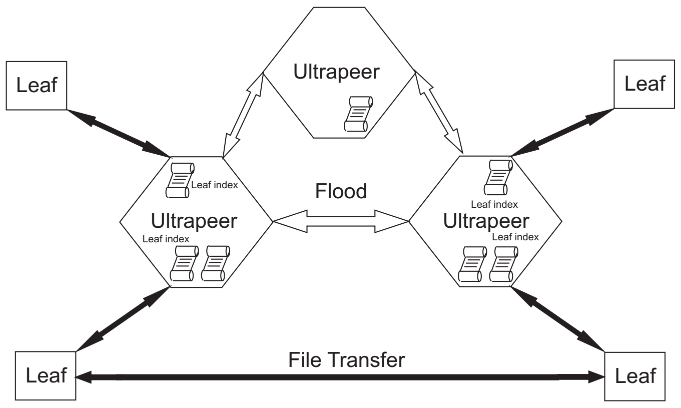


Figure 3.1: The Gnutella 2-tier architecture

It is easy to prove that in order to flood an entire, randomly constructed, network that employs 1-hop replication, one needs only reach $3/d$ of the peers during all hops but the last. In today's Gnutella, where the average degree is 30, one would need to reach 10% of the peers and then use 1-hop replication to forward the query to the appropriate last hop peers, in order to reach the entire network.

Most of the structured P2P systems today implement 1-hop replication by having peers exchange bloom filters of their indices. A Bloom filter [5] is a space efficient way to represent a set of objects (keys). They employ one or more uniform hash functions to map each key to a position in an N-sized binary array, whose bits are initially set to 0. Each key is mapped through each hash function to an array position which is set to 1. To check for the participation of some key in the set, the key is hashed to get its array position. If that array position is set to 1, the bloom filter indicates key membership. Bloom filters require much less space than the actual set, there is thus some loss of precision translated in the possibility of *false positives*. This means that a bloom filter may indicate membership for some key that does not belong to the set (more than one keys mapped to the same position). It cannot however indicate absence of a key which is in the set (false negative).

In Gnutella 2 [1] which uses a 2-tier architecture, each Leaf node sends its index (list of keywords) in the form of a bloom filter to all Ultrapeers it is connected to. Each Ultrapeer produced the XOR of all the bloom filters it receives from its Laves (approximately 30 Leaf nodes per Ultrapeer) and transmits this collective bloom filter to all its neighboring Ultrapeers to implement the 1-hop replication.

3.3 The Partitions Design

The scheme we propose allows for the partitioning of any type of content. More specifically, we propose the formation of categories based on easily applicable

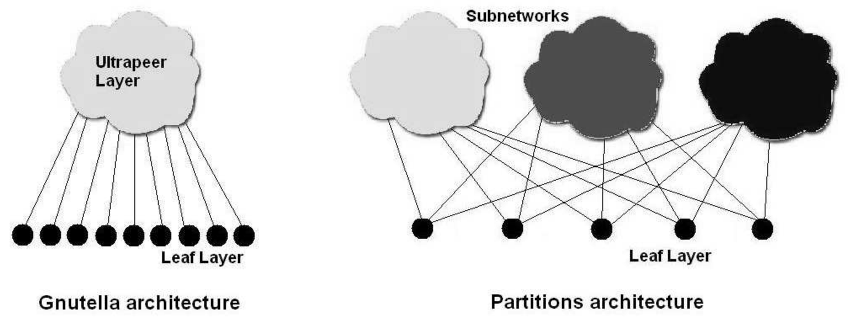


Figure 3.2: Illustration of the Gnutella network and the Partitions design

rules. Such a simple rule is to apply a uniform hash function on each keyword describing the files. This hash function maps each keyword to an integer, from a small set of integers. Each integer defines a different category. We thus categorize the keywords instead of the content (files) itself. Given a small set of integers, it is very likely that each peer will contain at least one keyword from each possible category.

Unstructured P2P systems like Gnutella 2 [1] employ a 2-tier structure. In those systems Ultrapeers form a random overlay network, while Leaf nodes are connected to Ultrapeers only. Each Leaf sends to the Ultrapeers it is connected to its index in the form of a bloom filter. Ultrapeers flood queries to the overlay network on the Leave's behalf. Flooding is only performed at the Ultrapeer level where 1-hop replication is implemented. Whenever an Ultrapeer receives a request this is targetedly forwarded only down to those Leaves that contain the desired information (except in the case of false positives). Fig. 3.1 shows a schematic representation of the 2-tier architecture.

The keyword categorization method is used in 2-tier unstructured systems. In the Partitions design, each Ultrapeer in the system is randomly and uniformly assigned responsibility for a single keyword category, by randomly selecting an integer from the range set of the hash function used to categorize the keywords. Ultrapeers responsible for the same category form a distinct subnetwork. Leaves connect to one Ultrapeer per subnetwork and send to it all the keywords belonging to that category. Thus, an innovative index splitting technique is used. Instead of each Leaf sending its entire index to an Ultrapeer, each Leaf splits its index based on the defined categories and distributes it to one Ultrapeer per category. Notice that peers operating as Ultrapeers also operate as Leaves at the same time (have a dual role). Even though in this design each Leaf connects to more than one Ultrapeers, the volume of information it transmits is the same since each part of its index is send to a single Ultrapeer. We can either send this information in the form of bloom filters or as strings containing the actual keywords. Each Leaf node sends to the Ultrapeer of a category all keywords that belong to the same category.

Each Ultrapeer sends to its neighboring Ultrapeers all the keywords of its Leaf nodes to implement 1-hop replication. Thus the amount of information transmitted from Leaves to Ultrapeers upon connection as well as for the 1-hop replication is increased compared to Gnutella which uses bloom filters. As we will see in the following section this increase is insignificant compared to the gain incurred from the reduced flooding traffic. In Fig. 3.2 we can see a schematic representation of the Partitions design.

This separation of Ultrapeers from content has the benefit of allowing them to be responsible for a single keyword category. The benefit of this is two-fold. First, it reduces the size of the subnetworks since they are completely discrete (at least on the overlay level). Secondly, it allows each Ultrapeer to use all its Ultrapeer connections to connect to other Ultrapeers of the same subnetwork, maintaining the efficiency of 1-hop replication at the Ultrapeer level.

There are, however, two apparent drawbacks to this design. The first one is due to the fact that each Leaf connects to more than one Ultrapeers, one per content category. Even though each Leaf sends the same amount of index data to the Ultrapeers upon connection as before, albeit distributed, however it requires more keepalive messages to ensure that its Ultrapeers are still operating. Keepalive messages however are very small compared to the average Gnutella protocol message. The second drawback arises from the fact that each subnetwork contains information for a specific keyword category. Requests however may contain more than one keywords and each result should match all of them. Since each Ultrapeer is aware of all keywords of its Leaves that belong to a specific category, it may forward a request to some Leaf that contains one of the keywords but not all of them. This fact reduces the efficiency of the 1-hop replication at the Ultrapeer level and at the Ultrapeer to Leaf query propagation. This drawback can be partly ammended in two ways. When we send a bloom filter to the Ultrapeer, that filter contains only keywords of a single type, thus making the filter more sparse and reducing he possibility of false positives. When we send a full index, false positives are completely eliminated and accuracy is increased. Furthermore, the most rare keyword can be used to direct the search, thus further increasing the effectiveness of the search method. The size of the index (set of the keywords) is small compared to the benefit of the additional accuracy.

In order to evaluate the effectiveness of our design and the impact of the drawbacks, we used a simple model that takes into account the aforementioned. Preliminary simulation results are also reported.

One final worth noting observation is the fact that we can remedy the fact that Partitions filter queries using one keyword only some more. This is done by using the most rare keyword when resolving multiple-keyword queries. Since we are using complete indices instead of Bloom filters, the rarity of a keyword in the system can be used effectively. (In the case of Bloom filters, rarity is not as important, due to false positives. This is because each position in the array has equal probability of being set). In order to determine which keyword is more rare, lookup can begin by first contacting one Ultrapeer per keyword type. Given the density of content per Ultrapeer and the fact that each Ultrapeer has complete knowledge of its neighbours' keywords, each Ultrapeer has complete knowledge of the keywords of 9300 Leaves. Such a sample should be more than enough for rarity determination purposes.

3.4 SecSPeer Implementation details

In SecSPeer, we have used a number of ten subnetworks. This number, as will be proven in the Evaluation chapter, was chosen so that the number of Leaf connections will be kept to small levels. This is because the whole philosophy of the two-tier approach is to reduce the load on Leaves. This means that each Ultrapeer serves 300 Leaves, however receiving from them $1/10$th of each one's keywords.

Chapter 4

Evaluation of the SecSPeer System

4.1 Introduction

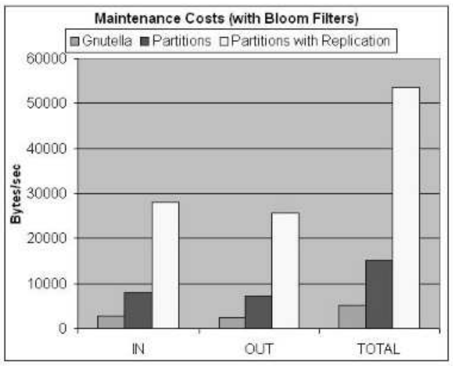
In this section, we shall present the results of measurements we made, in order to evaluate the scalability of SecSPeer. Two basic kinds of measurments were made, one to measure the maintenance costs of the system and one which included the operational costs also (i.e. query load).

4.2 Evaluation

In order to measure the maintenance costs of Gnutella and Partitions, we focus on the operation of a single Ultrapeer, because the load of Leaves is negligible in both systems compared to a Ultrapeers load since flooding is performed at the Ultrapeer overlay. In both cases we measured the costs in a few hours in the life of a single Ultrapeer, with Leaves coming and going. Each time a Leaf is connecting to the Ultrapeer, it sends its index information, which is propagated by the Ultrapeer to its thirty Ultrapeer neighbours. In addition, we assumed that, periodically, each Ultrapeer receives a small keep-alive message from each Leaf and replies with a similar message to each one of them. For each communication taking place, we measured the incoming or outgoing traffic in bytes, in order to estimate the bandwidth requirements.

There are two modifications in this scenario, between Gnutella and Partitions. In Partitions, the number of Leaves is 300.

In addition, the process of computing the size of the index information sent to the Ultrapeer differs greatly. In the case of Gnutella, we have used the code used by LimeWire [3], the most popular Gnutella client, to construct the bloom filter of each Leaf. We first randomly decided on the number of files shared by each Leaf, based on the file sharing distribution per peer presented in [9]. We then extracted this number of files from a list of filenames obtained from the network by



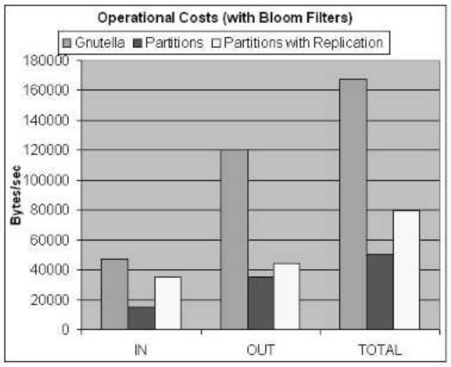
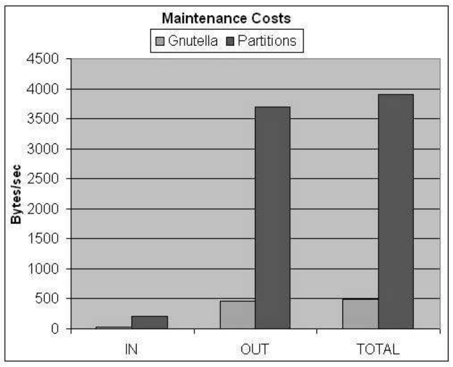


Figure 4.1: Maintenance costs for Gnutella and Partitions using Bloom Filters. Incoming, Outgoing and Total traffic.

Figure 4.2: Operational costs for Gnutella and Partitions using Bloom Filters. Incoming,Outgoing and Total traffic.



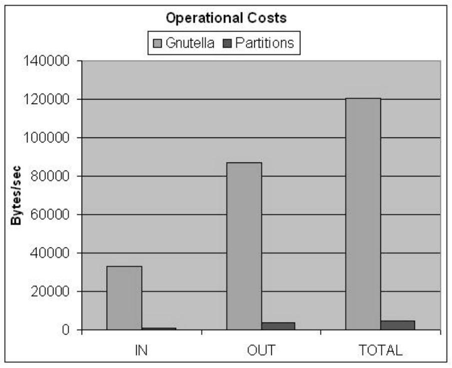


Figure 4.3: Maintenance costs for Gnutella and Partitions, using Full Indices for Partitions and Bloom Filters for Gnutella. Incoming, Outgoing and Total traffic.

Figure 4.4: Operational costs for Gnutella and Partitions using Full Indices for Partitions and Bloom Filters for Gnutella. Incoming, Outgoing and Total traffic.

a Gnutella crawler developed in out lab. Those filenames were fed to the LimeWire bloom filter generation code, which produced the corresponding bloom filter in compressed form, i.e., the way it is sent over the network by LimeWire servents. Thus we constructed the actual bloom filter, although what we really need in this case is just its size.

In the case of Partitions, we likewise computed the number of files to be shared by each Leaf. We extracted again the same number of filenames from the list of available filenames.

However, we also subdivided the Partitions scheme depending on the form of the index information sent by Leaves to Ultrapeers. Two experiments were run with the Partitions scheme using bloom filters. In the first, each bloom filter sent to an Ultrapeer only contained appropriate keywords (of the same type as the corresponding Ultrapeer). In the second experiment, we used replication, i.e. each bloom filter contained all the keywords of the Leaf, regardless of type. In addition, positions of keywords of the corresponding type as the Ultrapeer were set in the bloom filter to the value of two instead of one. (This bloom filter essentially distinguishes between keywords of the appropriate type and the rest types). Another experiment was run using full indices. Here, instead of constructing the bloom filter, we instead compressed the string formed by all filenames fo a Leaf and compute its size. Notice that there is no replication in the full indices experiments. Each Leaf sends a compressed string of all keywords of the same type as the Ultrapeer it sends the string to. Figs. 4.1 and 4.3 show the results of the measurements for the costs of maintaining the structures of Gnutella and Partitions, without any query (flood) traffic. From this figure it is obvious that, as expected, the maintenance cost of partitions is higher than that of Gnutella, but not as much. As we will see in the next paragraph the gains incurred during the operational phase of the two systems outweighs the increased maintenance costs.

We then focused our attention to the query traffic load. Measurements conducted in our lab showed that, on the average, each Ultrapeer generates 36 queries per hour (i.e., queries initiated by itself or its Leaves). This adds up to approximately 2000 queries per second generated anywhere in the Gnutella network. In our measurements, we assumed that the aim of each flood (both in Gnutella and Partitions) is to reach the entire network. As we mentioned before, such a flood would need to reach $\frac{1}{10}$th of the Gnutella's network (or a Partitions' subnetwork). This means that the Ultrapeer in our measurements has a probability of 0.1 to receiving each query. In addition, every time this does not occur, it has another opportunity to receive the query during the last hop, depending on its bloom filter (or index) (in case the searched keywords match its index or bloom filter). Should the Ultrapeer receive a query, it is assumed to propagate it to its Leaves, again depending on their bloom filters or index (again depending on a possible keyword match by the bloom filter or the index). Figs. 4.2 and 4.4 show the comparison in the traffic load of Gnutella and Partitions, including maintenance and query traffic. We used a size of 40 bytes for each query. In reality, the size of a query can be up to a few hundred bytes, if XML extensions are used. In addition, for every 1400 bytes for each message sent, we added 40 bytes for the TCP and IP header. From these figures it is evident that Partitions outperform Gnutella in operational costs, in every case.

4.3 Conclusions

In this Chapter, we have shown that the SecSPeer servent, while employing the Partitions scheme, faces up to even 5 times less traffic. This clearly demonstrates the ability of SecSPeer to scale to larger numbers than other unstructured P2P systems, since its operation imposes less bandwidth requirements to its servents.

Chapter 5

Security Considerations

In this Chapter we analyze issues in regards to the Security considerations of the SecSPeer system. We first give a short introduction to Security issues, since we have mention them in detail in previous deliverables, we proceed and analyze the architecture of the new system and conclude with its evaluation.

5.1 Introduction

Unstructured P2P systems are very vulnerable to security exploitation. The main reason for the latter observation is the lack of a central component to verify the information injected by every node in the system or even to verify the identity of every node entering the system [6]. That is, adversaries may inject malicious nodes in the system, which in turn they may inject fake information that will drive the system in collapse or even redirect the system to a third party computer. Redirecting a major portion of the P2P system in a third party computer is considered as a pure Denial of Service attack, which includes properties of a Flash Crowd event.

The nature of the Denial of Service attack and the machanisms employeed in order to achieve this kind of trick are analyzed in previous Deliverables.

It is crucial to implement an architecture, which will prevent the use of the system as a Denial of Service weapon against third party computers.

5.2 Architecture

The base architecture of our system, as far as the security considerations are concerned, is build in the SEALING (Short Term Safe Listing) algorithm. Although the SEALING algorithm has been mentioned in previous deliverables, we provide the algorithm in this document, in Fig. 5.1, for easier reading.

Denote, that the algorithm is based on the SEALING heuristic, which tests nodes to verify that they operate the SecSPeer protocol. This kind of verification, which is performed in a distributed way for every node, is used in order to isolate the system from anaware third party computers, which can not serve SecSPeer requests. That is, every node of the system verifies that it connects to a compatible with the protocol node, before sending a SecSPeer request.

The SEALING Validation criterion is depicted in 5.2.

```
SafeListLifeTime := 30 mins;
if (GnutellaPacket(pkt) == QueryHitPacket) {
    GnutellaExtractNode(pkt, &GnutellaNode);
     if (SafeListContains(GnutellaNode)) {
        if (CurrentTime() -
            SafeListGetTimeOfNode(GnutellaNode) <
            SafeListLifeTime)
            GnutellaDropPacket(pkt);
    }
    else
          GnutellaParseHits(pkt);
}
...
onDownloadAttempt(node, file) {
    if (GnutellaHandShake(node))
        GnutellaDownload(node, file);
    else
        SafeListAdd(node);
}
...
```

Figure 5.1: SEALING Algorithm.

5.3 Evaluation

The evaluation of the SEALING algorithm must be done in the server (victim) side. Denote, that our security considerations transform SecSPeer to a safe P2P protocol in the Internet environment, in the sense that it may not interfere with other services, such as the World Wide Web. That is, although the SEALING algorithm is completely distributed and it is based on local heuristics, it is useless to evaluete it in per node basis. We are interested in the request attempts absorbed by the SEALING algorithm - requests that they never issued, since the server (victim) kept in a safe list. Thus, it is easier to measure this request rate in a sensor co-located with an attacked Web Server.

The evaluation testbed is as follows. A Web Server stands as a victim and it logs all incoming requests. The incoming requests in our controlled Web Server are side effect of the malicious behaviour of SecSPeer clients. That is, clients that advertize the Web Server as a SecSPeer participant in the system.

Legitimate SecSPeer nodes that they are fooled by malicious nodes and send requests to the victim Web Server represent the attack mangnituted. SecSPeer nodes, which utilize the SEALING algorithm have a reduced attack magnitude, compared to the attack issued by non-SEALING compatible SecSPeer nodes.

The evaluation graph is depicted in 5.3

SEALING Validation Criterion: *Any host advertised in a SecSPeer response packet which can not respond correctly to a SecSPeer Handshake process is considered as a non-SecSPeer participant and a potential victim for a Denial of Service attack.*

Figure 5.2: The SEALING criterion

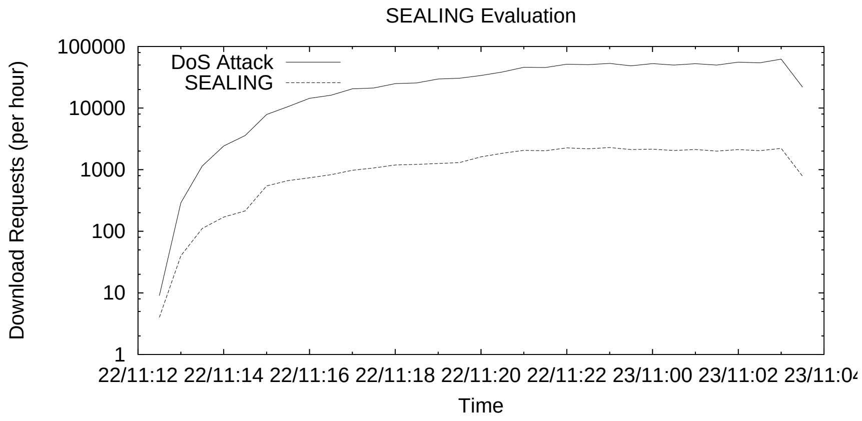


Figure 5.3: The evaluation graph of the SEALING algorithm. The solid curve represents the amount of requests during a DoS attack using the P2P system. The dashed curve represents the amount of requests that will be eventually exposed to a third party server (victim) when the system utilizes the SEALING algorithm. The difference of the two curves represents the savings - the reduction of the attack magnitude - intoduced by the utilization of the SEALING algorithm in the system.

Chapter 6

Conclusions

The information presented in this document illustrates the proven efficiency of SecSPeer. SecSPeer servents, by employing the Partitions scheme, were shown to face much less traffic load and thus exhibit a much larger degree of scalability, surpassing that of traditional unstructured P2P systems by even an order of magnitude. In addition, by using the Sealing algorithm, it was shown to efficiently detect and avoid efforts of malicious peers to exploit the system to perform DDoS attacks, thus providing better security than its counterparts. Finally, it includes an efficient broadcast mechanism, which can be used to broadcast system-wide information, such as an approximation of the size of the network.

References

[1] Gnutella 0.6 protocol specification.

[2] Kazaa inc. http://www.kazaa.com.

[3] Limewire inc. http://www.limewire.com.

[4] Gkantsidis C. Mihail M. Saberi A. Hybrid search schemes for unstructured peer-to-peer networks. In *Proceedings of INFOCOM'05*, 2005.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] J. Douceur. The sybil attack. In *John R. Douceur. The sybil attack. In Proc. of the IPTPS02 Workshop, Cambridge, MA (USA), March 2002.*, 2002.

[7] Elias Athanasopoulos Harris Papadakis. Secspeer deliverable 2.1: System design. 2005.

[8] Elias Athanasopoulos Harris Papadakis. Secspeer deliverable 3.1: System implementation. 2005.

[9] Shanyu Zhao R. Rejaie and D. Stutzbach. Characterizing files in the modern gnutella network: A measurement study. In *Proc. SPIE/ACM Multimedia Computing and Networking*, 2005.