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**)**

D3.1 "System Implementation"

Abstract: This document describes the implementation of the design of a scalable and secure unstructured Peer-to-Peer system, as described in Deliverable 2.1: "System Design".

Contractual Date of Delivery	29 December 2005
Actual Date of Delivery	19 June 2006
Deliverable Security Class	Public
Editor	Charalambos Papadakis, Elias Athanasopoulos
Contributors	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

Chapter 1

Introduction

This document comprises the report for the Deliverable 3.1: Implementation, of the SecSPeer project. The purpose of this document is to describe the implementation of the system described in Deliverable 2.1: Design, a P2P system designed to be scalable and secure. The structure of this document is as follows:
Chapter 1 provides an overview of the implementation.
Chapter 2 provides details on the implementation of the algorithm for reduction of duplicate messages, as described in Deliverable 2.1: Design, Chapter 3.
Chapter 3 provides details on the Coloring algorithm described in Chapter 4 of the Deliverable 2.1.
Chapter 4 describes the implementation of a policing mechanism to protect the system against spam traffic and exploiting it to stage indirect DDoS attacks. The design of this mechanism was described in Chapter 5 of Deliverable 2.1.
Chapter 5 provides some conclusional remarks. The document ends with the Bibliography.

Chapter 2

Overview

In this Chapter, we present an overview of the implementation of the algorithms proposed in Deliverable 2.1: System Design [8]. Each of the following chapters describes the implementation of each proposed idea, each meant to address one of the most important problems P2P systems face today. Each chapter begins with some introduction and continues with a summary of the problem and the proposed solution. Then follows the description of the implementation along with details on the implementation choices and the code itself. All snippets of code presented in the chapters are in C++-like pseudocode and can be easily integrated into most unstructured P2P clients today. Each chapter ends with a short discussion, to summarize its contents.

Great care has been taken in both the design and especially the implementation phases, to ensure that the proposed modifications are easily integrated in the code of the unstructured peer-to-peer (P2P) systems used today, such as Gnutella [12] clients, transforming them into SecSPeer compliant clients. Not only that, but also we have tried to ensure that components of our system can inter-operate with components without the modifications we have proposed and implemented, albeit of course with some reduction to the efficiency of our algorithms, compared to a system where all components use our modifications. The fact that our clients can operate in legacy networks and other clients can easily be modified to operate in our network will ease the deployment of SecSPeer. The implementation changes do change the Gnutella communication protocol, in such a way however that clients with legacy code can operate in our system and vice versa.

Recall that an unstructured P2P system is comprised of a number of clients, called servents. Each servent is usually aware of (and connected to) a small number of other servents. As was done in Deliverable 2.1, we model an unstructured P2P system as an undirected graph, where each node is a servent and each edge is a direct connection between neighbours. Throughout the rest of the document, we shall use the terms node, servent and client interchangebly, all three of them having the same meaning.

It should be noted that in this document, we only describe the implementation

of the modifications in the already existant P2P systems. For that reason, there is no code presented here of the algorithms used in today's unstructured P2P systems that we also use. For instance, we have not inlcuded the code that a client uses to connect to the network, which is the same for our system, as is for every Gnutella client.

Chapter 3

Implementation of Duplicate Elimination Algorithm

3.1 Introduction

This chapter contains the description of the implementation of the duplicate elimination algorithm described in Chapter 3 of Deliverable 2.1, entitled ”A feedback-based approach to reduce duplicate messages”. One can recall from Deliverable 2.1 that the lookup mechanism used in unstructured P2P systems today, called flooding, can generate a large number of redundant messages. The broadcast of a message in the network can result to each node receiving the same message multiple times, due to cycles in the network graph. In order to discern which messages will be redundant and which won't, the proposed solution dictates that each node distinguishes each incoming message based on the direction it is travelling (expressed as the neighbour the node received the message from) and the distance the message has already travelled (expressed as the number of times this message has already been propagated through the network, i.e.: the hop count). Each different neighbour-hop count pair defines a different message category. Each node stops forwarding to its neighbours those kind of messages (categories) that have a high ratio of redundant messages (duplicates).

3.2 Implementation

What follows is a detailed explation of the implementation of the algorithm. The implementation described here assumes a horizon value of one, since in Deliverable 2.1 we showed this horizon value to be the most efficient. The code we present here can be easily modified to support higher horizon values (the orgn variable used in the code presented below can be modified to represent the list of nodes in the messages path, instead of a single node).

```
Msg
{
  UID
  type
  TTL
  hops
  frm
  orgn
  payload
}
```

Figure 3.1: Message. This collection of data represents the information contained in a message.

3.2.1 Message format

Figure 3.1 shows the format of the messages exchanged between nodes. Information needed to be exchanged as part of our algorithm, is embedded is the normal flood messages. We only define a new message type *Stop* message) which can be formated as a normal message, with specific values in its fields. Thus, the same format is used for all messages. As is the case in most unstructured systems today, each message is assigned some globally unique id (*UID*). However, messages that belong to the same flood, are all assigned the same UID. This UID is used, among other things, to recognize duplicate messages. The *type* of the message defines whether the message is a query, a "*stop*" message, as described below, or any other type defined by the individual protocol of each unstructured P2P system. The *TTL* field defines the number of times this message can be forwarded. Each time it is forwarded, it is forwarded with its TTL value reduced by one. The *hops* field contains the number of times the message has already been forwarded. In contrast to the TTL field, the hops field in incremented each time the message is forwarded. This of course means that the sum of those two fields is always constant. The *frm* field contains some unique tag to identify the node that sent the message to the node that received it, whereas the *orgn* field contains the identity of the node that sent the message to the frm node. Finally, the *payload* field is added for completeness and contains the actuall information of the message (for instance, the query strings). Apart from the orgn field, all other fields are already part of the Gnutella messages. The reason for the orgn field is explained below. In short, the data for each node's categories (figure 3.2) are not maintained by the node itself but rather by the edge corresponding to each category. This is so that there is no need for each node to send a duplicate notification back, each time it receives a duplicate message but rather a single (*stop*) message.

```
Category
{
  orgn (*)
  hops (*)
  edge (*)
  good
  dups
}
```

Figure 3.2: Category data. This collection of data defines a category along with the number of non-duplicates (”good”) and duplicate (”dups”) messages of this category. The starred fields comprise the triplet that uniquely identifies the category in that node.

3.2.2 Categories

Figure 3.2 shows the data structure used to define a category of messages. Each category is defined by a triplet. This triplet consists of the *originator(orgn)*, the *hop count(hops)* and the *edge(edge)*. As explained above, each node distinguishes each message it receives, based on the previous node and the hop count of the message. Since, however, messages that belong to the same category, have different duplicate ratios for each neighbour the node forwards it to, each node subdivides each category based on the edge. The *good* and *dups* fields represent the non-duplicate and duplicate count of all messages of this category, respectively. In the code presented below, we assume the existance of a dictionary of Category items, called *”Categories”*. This dictionary stores Category items, based on the key formed by the $< orgn, hops, edge >$ triplet. In effect, each category tells the node that, forwarding a message, received from node *orgn* and coming from *hops* nodes away, to neighbour *edge*, has a *good*/*dups* chance of being a redundant message. Notice that, although this information is relative to some node X, it is actually stored in the node *edge*. As mentioned before, this is because, in order for node X to be aware of the number of redundant messages (*dups*) it has sent for that category, node *edge* should notify node X for each duplicate message node X sends it. Instead, to avoid this steady stream of messages, the category is stored in node *edge*. However, since that node is not aware of the aforementioned *orgn* for each message node X receives and forwards, node X includes this information in each message propagated (hence the *orgn* field in the message format).

3.2.3 Banned Categories

Figure 3.3 shows the data structure that defines a banned category. This structured is similar to the Category structure of figure 3.2. However, since this data structure defines an already ”eliminated” category, which means that no messages of this kind will be propagated by this node, there is not need for the ”good” and ”dups”

```
BannedCategory
{
  orgn (*)
  hops (*)
  edge (*)
}
```

Figure 3.3: Banned Category data. This collection of data define a category of messages that this node will not forward any more.

```
01  function onReceiveMessage(Msg m)
02  {
03    if (m.type == "Query")
04      onReceiveQueryMessage(m);
05    if (m.type == "Stop")
06      onReceiveStopMessage(Msg m);
07  }
```

Figure 3.4: Function onReceiveMessage. This function is called to process each incoming message.

fields. As is the case of the Categories dictionary, the code presented also assumes the existence of a BannedCategories dictionary. As is shown in the code below, each node will first check this dictionary. If the category of the message it has just received exists in the dictionary, for some edge-neighbour, the node does not forward the message over that particular edge. In addition, and in contrast to the normal categories, the BannedCategories information is stored on the actual node to whom they belong.

3.2.4 Function onReceiveMessage

Function *onReceiveMessage* (figure 3.4) is used to distinguish between Query messages and Stop messages and to call the appropriate function. We discern between those two cases, in order to make the code snipets presented here smaller and easier to understand.

3.2.5 Function onReceiveQueryMessage

Function *onReceiveQueryMessage* (figure 3.5) is used to process a query message. It first checks to see whether the message received is a duplicate or not (Another dictionary to hold the UIDs of messages already received is assumed. Such a dictionary, in some form or another, already exists in all unstructured P2P systems in use today, that we know of). If it is a duplicate, the appropriate category is located

and its *dups* field incremented. On the other case, if the message is not a duplicate, its UID is added to the corresponding dictionary of UIDs, and the *good* field of the appropriate category is incremented. In both cases, if such a category does not exist in the dictionary, the code of the dictionary creates one at that time. Again in both cases, the banCategories function is called to check whether the measurements the node has collected so far are sufficient for it to decide which categories generate most duplicates, while at the same time having minimal contribution to the total number of the non-duplicates. When this process is completed, the node starts forwarding the message to its neighbours. However, as mentioned before, it will not forward the message to some neighbour, if the category of the message, for that neighbour can be found in the BannedCategories dictionary. Finally, the processQuery method implements the node's processing the new query locally (for instance to see if some of the data it holds match the query's criteria).

3.2.6 Function onReceiveStopMessage

Figure 3.6 shows the code of the function used whenever a *"Stop"* message is received. The transmission of such a message is explained further on in this chapter. The node may receive a *"Stop"* message from any of its neighbours. The message notifies the node that most of the messages received from its neighour defined in the orgn field and which originated a number of hops away from the node equal to the hops field of the message, are mostly duplicates when forwarded to the neighbour that sends the *"Stop"* message. This means that the node should stop forwarding those kind of messages to that neighbour, and thus adds that category, for that edge-neighbour in the BannedCategories dictionary. Since the node defined in the orgn field is a neighbour of the node that receives the *"Stop"* message (and not of the node that sends it), this explains the need for an orgn field in the Msg format (figure 3.1), where the node that forwards a message includes the previous node in the message.

3.2.7 Function banCategories

The warm-up period of the algorithm is defined as the time the node spends observing the traffic patterns, in order to make the necessary decisions later on. The *banCategories* function of figure 3.7 is used to first determine whether the measurements collected so far are sufficient to decide which categories are more harmful to the system than useful. If that is the case, the *cut* function described below is used to do the actual elimination of the harmful categories, by deciding which is which. The *banCategories* function declares the warm-up period over when the ratio of all the duplicates over all the messages received by that node more or less stabilizes to some value. The value is assumed to have stabilized when the total change over the last (thirty in this case) changes in its value are less than 0.01 and the average of the absolute of each change is less than 0.02. Those thresholds ensure that not only the changes in the ratio value are small, but also that the value is not slowly increasing

or decreasing constantly over time. If the value in question has stabilized, the *cut* function is called. The *oldPerc* variable is used so that the ratio value computed by one call of the *banCategories* function is retained in the next call of the same function, in order to compute the difference. Finally the value of the *cgood* variable is checked. If it is positive, this means that not only the warm-up phase was over and the *cut* finction was called, but the *cut* function also banned some of the categories, as explained below. If this is the case, then all other measurements are purged and the process starts anew, in the warm-up phase (The *oldPerc* variable is re-initialized, as are the rest of the variables that control the stabilization process). This is done because the traffic patterns will change, due to the fact that some categories have been banned. If the value of the *cgood* variable remains zero, either the warm-up period is not over, or no categories were eliminated. In either case, the measurements continue until we are able to ban some categories.

3.2.8 Function cut

The last function presented in this chapter is used whenever the banCategories function declares the warm-up phase over. This means that the knowledge of the traffic patterns of the network is sufficient to allow us to decide which messages should not be forwarded. The *cut* function (figure 3.8) iterates through all categories. As is shown, for each category, the function computes the current ratio of all duplicate messages over all messages (line 18). It then computes the same ratio with the duplicate messages quantity reduced by the duplicate messages of the category considered. However, if that category is banned, the non-duplicate messages as well as the duplicate ones are eliminated. For this reason, line 27 computes the ratio of the remaining non-duplicate messages over the total non-duplicate messages. The same ratio, when the category was not banned is one (line 19). Having two D and C pairs, the algorithm has to decide which one is more efficient. The efficiency function used is defined as the product of each C and D pair. This efficiency function can easily be modified to adapt to cases where increased coverage of the flood is required, even at the cost of more duplicate messages, by using, for instance, $(C^2 * D)^{1/3}$. Should the efficiency rating value of the C and D pair that corresponds to the case where the considered category is banned be higher, the node sends a *"Stop"* message to the appropriate neighbour, to notify it not to forward certain messages to itself. Finally the *cgood* and *cdups* variables are used to count the total number of non-duplicate and duplicate messages of all the categories that were banned.

3.3 Conclusions

In this chapter, we presented the implementation of the duplicate elimination algorithm, designed in Deliverable 2.1. The code we presented were written in C++ with only some small parts of pseudo-code to facilitate its understanding and fo-

cus on the important parts of the implementation. The extreme similarity of the code to C++ enables the code to be easily imported to any unstructured P2P system client, especially any Gnutella client, such as LimeWire [5], BearShare [3] and Gtk-Gnutella [4], to make them compatible with SecSPeer. In addition, since during the initialization of a connection between two servents, information regarding the version of the software of each client is exchanged, it is easy for any client incorporating our code, to behave as a legacy client over connections with other legacy clients. This means that not only legacy clients can easily be enhanced to be able to work in our system, but also clients of the SecSPeer system can easily be integrated in the existing unstructured P2P networks. The more clients participating in the network use our code, the better the efficiency. Notice that inside the implementation, a number of dictionaries are used. The dictionary is a well-known data structure and different and efficient implementations of this data structure are abudant. In our case, for instance, we used red-black trees [13].

```
01  function onReceiveQueryMessage(Msg m)
02  {
03
04    if (m is duplicate)
05    {
06      Category c = Categories.find(m.orgn, m.hops-1, m.frm);
07      c.dups++;
08      banCategories();
09    }
10    else
11    {
12      Category c = Categories.find(m.orgn, m.hops-1, m.frm);
13      c.good++;
14
15      processQuery(m);
16      banCategories();
17
18      foreach (Neighbour n in Neighbours)
19      {
20          if (BannedCategories.exists(frm, m.hops, n))
21          {
22            Msg nm = new Msg(m.UID, m.type,
23                                m.TTL-1, m.hops+1, m.frm, m.paylo
24            sendMessage(nm, n);
25          }
26      }
27    }
28  }
```

Figure 3.5: Function onReceiveQueryMessage. This function processes a query message

```
01  function onReceiveStopMessage(Msg m)
02  {
03    BannedCategories.add(new BannedCategory(m.orgn, m.hops, m.frm))
04  }
```

Figure 3.6: Function onReceiveStopMessage. This function processes a stop message

```
01  function banCategories()
02  {
03    int msgs, good, dups, cgood = 0, cdups = 0;
04    static float oldPerc = -1;
05    static int change = 0;
06
07    foreach(Category c in Categories)
08    {
09      good += c.good;
10      dups += c.dups;
11    }
12
13    msgs = good+dups;
14
15    if (!msgs) return;
16
17    if (oldPerc != -1)
18    {
19      float newPerc = (dups*100.0)/msgs;
20
21      if (change < 30)
22      {
23        change++;
24        percdiff += oldPerc - newPerc;
25        percabs += fabs(oldPerc - newPerc);
26      }
27      else
28      {
29        percdiff -= percdiff /30;
30        percdiff += oldPerc - newPerc;
31        percabs -= percabs /30;
32        percabs += fabs(oldPerc - newPerc);
33
34        if (percdiff < 1 && (percabs/30) < 2)
35          cut();
36      }
37    }
38    if (cgood > 0)
39    {
40      Categories.purgeAll();
41      oldPerc = -1;
42      percdiff = percabs = change = 0;
43    }
44    else
45      oldPerc = ((dups-cdups)*100.0)/(good+dups-cdups-cgood);
46  }
```

Figure 3.7: Function banCategories. This function checks for the conclusion of the warm-up phase.

```
01  function cut()
02  {
03    foreach(Category c in Categories)
04    {
05      int oldg, oldd, newg, newd;
06      double oldRate, newRate, D, C;
07
08      oldg = good - cgood;
09      oldd = dups - cdups;
10      newg = oldg - c.good;
11      newd = oldd - c.dups;
12
13      if (oldg+oldd == 0)
14        oldRate = 0;
15      else
16      {
18          D = ((oldg*1.0)/(oldg+oldd)) ;
19          C = ((oldg*1.0)/oldg);
20          oldRate = D*C;
21      }
22      if (newg+newd == 0)
23        newRate = 0;
24      else
25      {
26        D = ((newg*1.0)/(newg+newd));
27        C = ((newg*1.0)/oldg);
28        newRate =  D*C ;
29      }
30
31      if (newRate > oldRate)
32      {
33        Msg nm = new Msg(m.UID, STOP, 0, c.hops, c.orgn, "");
34        sendMessage(nm, c,frm);
35        Categories.remove(c.orgn, c.hops, c.edge);
36        cgood += (c.msgs - c.dups);
37        cdups += c.dups;
38      }
39    }
47  }
```

Figure 3.8: Function cut. This function performs the actual filtering on the categories.

Chapter 4

Implementation of the Coloring algorithm

4.1 Introduction

This chapter contains the description of the implementation of the algorithm described in Chapter 4 of Deliverable 2.1, entitled ”Flood Driving algorithms - Divide and Conquer”. The purpose of the algorithms implemented in the previous chapter was to reduce the cost (in messages) of flooding a system of N nodes with d average degree, from (d-1)*N to N, thus making it independant of the average degree. The purpose of the algorithms implemented in this chapter is to further reduce that cost to less than N. If possible, the cost should be in the order of the number of results requested in each query. As one may recall from Deliverable 2.1, the main idea behind this scheme is to create subnetworks of the original network. Each subnetwork corresponds to some type of content and some node may join in any subnetwork only if it contains data of that type. Each query is also assigned a type in the same manner and is directed only to the corresponding subnetwork. This way, the flood is directed only to nodes that have a chance of containing some result. If the node that initiates the query is not aware of any node participating in the subnetwork to be flooded, a flood on all its neighbours (on all subnetworks) can be conducted to locate such a node. Since, by definition, the number of the nodes participating in some subnetwork is larger than the data being looked up, the search for a node will be less costly. The criterion we use to categorize the data and the queries is the character set used in the names of the data and the strings of the query. All possible character sets are sorted according to how widespread their use is. In cases where a query contains more than one character set, the one used less often is used to direct the query. The implementation of the scheme summarily outlined above, which we call *”Coloring”*, is described below.

4.2 Implementation

This section presents, in detail, the implementation of the Coloring algorithm. The algorithm (and its implementation) is divided into two subparts, the bootstrapping procedure and the actual searching. The bootstrapping procedure is used to connect the node in the system, by connecting it to the appropriate subnetworks, depending on the types of data it shares. Each node should not connect to many subnetworks. Not only the opposite would require a lot of connections, but connecting to many subnetworks means that this node will receive many queries, which beats the reason for using the Coloring scheme. Thus, the criterion used to categorize the content of each node should define a few categories which describe most of the content of the node. The character set criterion we have used exhibits this property, since it is quite rare that some user's client will contain data that use all character sets (in a sense, it is quite rare that a user speaks all languages). One may note that almost every user may posses data that use the english character set, however, on the other hand, not all users contain data that use other character sets, such as chinese or cyrillic. Using this criterion, it is also easy to characterize a query even without the help of the requesting user. We use the Unicode Character Code for many reasons. Not only is it the current standard, containing symbols for all languages (even ancient or fantastical ones) but also the code for the latin letters is the same as in ASCII code, making it possible to categorize data whose names are in ASCII code.

4.2.1 Unicode-based Categories

Figure 4.1 shows the categories we have defined in the implementation, using the available Unicode character scripts. The subnetworks that will be formed in the system correspont to the middle column of the figure. Some categories may also function as a super-category for other categories. This hierarchy, shown in figure 4.2, reflects the fact that some languages share many characters (usually because they evolved from the same base language).This hierarchical information, however, is only used when querying the system and is not reflected in the structure of the subnetworks. Since characters of a parent category are also used by each of its children, as shown in figure 4.2, one needs a prioritization scheme to define unambiguously the category each data belongs to. It is the case that all characters of some piece of data can only be categorized to one category on each level of the tree shown in figure 4.2, since the character sets that distinguish each category on the same level, are distict. For instance, in the level of the "*German-based*" category, the other two categories on the same level ("*Spanish*" and "*Portuguese*") do not use umlauted characters. However, one needs to disambiguate between categories of different levels. In the *Latin* super-category for instance, should the word "försäljningsframgång" be assigned to the *Latin*, the *German-based* or the *Scandinavian* category? The rule used to solve this problem is that some piece of data is assigned to the category lowest in hierarchy tree. Thus, in the example above

the word will be assigned to the *Scandinavian*. Any node containing this word will only connect to that category and not any ”parent” categories. Using the same scheme, each query assigned to some category, should be sent to that category and all its ”children” in the tree. For instance, any query that is assigned to the *German-based* category should be also sent to the *Scandinavian*, *Dannish - Norwegian* and *Finnish* categories.In the code presented in the rest of this chapter, we assume the existence of a function which returns a data structure (*List*) containing all the categories defined by the contents of a node. (Figure 4.3).

4.2.2 Bootstrapping

As mentioned before, each subnetwork corresponds to one category and vice versa. Bootstrapping in the appropriate character sets can be done in the same way that a Gnutella client connects to the network today. This is done using webcaches to learn of some other nodes already in connected in the network and use those nodes to connect. Since there is a known number of character sets, one webcache can be used for each category/subnetwork. Assuming a function that Gnutella uses today to bootstrap in the system, given a webcache and the number of connections to initiate, we can use it to implement our bootstrapping function as is shown in Figure 4.4. Parameter *webcaches[]* is assumed to be an array containing the webcaches for all categories, indexed on a per-category basis. The *size* function returns the number of categories in the *List*. Thus, we divide all available connections equally among the subnetworks the node should join.

Ultrapeers

Notice that the *GnutellaBootstrap* function differs when a node is in *”Leaf”* mode from when it is in ”*Ultrapeer*” node. We assume the reader is familiar with the ”*Ultrapeer* architecture [2], which was also described in Deliverable 2.1. This means that in the case of Leaves, the result of this function will be that for each category it is invoked, at most three connections will be made to Ultrapeers belonging to that category/subnetwork. This is the standard behaviour of this function in Gnutella-compliant clients and it has almost the same behaviour in our implementation. It only differs in one aspect. This difference explains the existence of array *mode*. This array contains information regarding the status of the node in each category it belongs to. Notice that the same node can be a Leaf in some category and an Ultrapeer in another. In standard GnutellaBootstrap function, each node sends a Bloom filter [9] of its content to all neighbours. If the node is in Ultra-peer node, that bloom filter includes the contents of its Leaves. In the case of SecSPeer, a Leaf sends to each Ultrapeer it connects to, only the content that belongs to that subnetwork. Similarly, an Ultrapeer of category A (for instance), will send to its Ultrapeer neighbours in subnetwork A a Bloom filter of its category A content, including the content of its category A Leaves. What it more, during the handshaking of a new connection, the two participating nodes should exchange information

regarding all the subnetworks each is part of, regardless of but including if they are Leaves or Ultrapeers in each one of them. Since the new neighbour Y of some node X may be also part of some other subnetwork that node X is also part of, this means that this connection can be used as a connection for both subnetworks, reducing the total number of connections required. For instance, nodes than belong to the *Scandinavian* category will more likely also belong to the *German-based* category, since they will most likely contain words with umlauts but without the åcharacter. The *pruneConnections* function computes the available connections for each subnetwork and disconnects any of them which are in excess of the number of connections required for each subnetwork, in order to try to reduce the number of connections to D, if the bootstrapping procedure resulted to more than D connections. Another approach that can be used in the Ultrapeer-Leaf connectivity scheme can be used to even more reduce the number of connections (and workload) of Leaves, at the same time of course, placing more stress on Ultrapeers. Each Leaf can use only one Ultrapeer to connect to the network, instead of one per category it belongs to. This means however that all the content (of any type) of that Leaf should be sent to the Ultrapeer. In turn, the Ultrapeers ”adopts” that content and thus connects to any appropriate catefories. This scheme has the drawback that the content of each Ultrapeer becomes more diverse, thus forcing the Ultrapeer to connect to more subnetworks. As we mentioned before, the larger the conjuctions between any two subnetworks, the less efficient the Coloring scheme is. One way to still use this Ultrapeer-Leaf connectivity scheme and yet not jeopardize the efficiency of the algorithm is to make the Ultrapeer reject any Leaves that will expand its range of categories over some limit. For instance, if that limit is two categories, then the Ultrapeer will reject any Leaves that differ in more than two categories. If the Ultrapeer already has its categories expanded by two, it will not accept any Leaves with even one category not already part of the Ultrapeer’s epanded list of categories. According to this scheme, the bootstrapping of both Leaves and Ultrapeers is the same as in Gnutella. The difference with the SecSPeer bootstrap is that now, the categories of the Ultrapeer can change every time a new Leaf is connected, while in the previous scheme, each Ultrapeer received from its Leaves only appropriate content. Thus, we now need an ”*onLeafConnect()*” function, illustrated in figure 4.5. The *nodeList* dictionary corresponds to the Leaf’s categories, whereas the *compare* function used returns a dictionary of the categories of *nodeList* that are not already in *catList*, in this case. The *expLimit* variable contains the maximum number of categories this Ultrapeer can accept.

4.2.3 Searching

Searching in SecSPeer is divided into two, distinct phases, explained below, in detail. Even before those two phases, however, the query to be sent is assigned to some category (using the *getCategories()* function, which in this case will return one result/category). The purpose of this is that that query should only be propagated across the nodes of the subnetwork that correspond to that category. Notice

that if the category of the query is not a leaf node in the tree of figure 4.2, then the query will have to be propagated to more than one categories/subnetworks. To be able to do that, the node has to be (or become) aware of at least one node in each of these subnetworks. At this point, the *Searching for the subnetwork* process is invoked, for each subnetwork we need to contact.

Searching for the subnetwork

The pseudo-code for locating a node of a subnetwork is illustrated in figure 4.6. The node initiating the subnetwork search first checks whether it is itself a part of that subnetwork. If that is the case, the first phase of the lookup (*Searching for the subnetwork*) is already complete, and the node moves to the second phase. Otherwise, it checks to see if at least one of its neighbours is part of that subnetwork. This process requires no communication, since during the handshake procedure, when a node first connects to a neighbour, information about the nodes, including the subnetworks they belong to, is exchanged. Finally, if all has failed, the node asks its neighbours if they are aware of any node of that subnetwork. The neighbours will check whether some of their neighbours are part of the sub-network and may or may not also propagate the request, according to the TTL value. If the node receives some positive replies, it caches them for so long, as half the average lifetime of a node it has observed so far. This information can be used to answer queries of other nodes, themselves looking for nodes of that subnetwork. This means that the reply of the neighbours we mentioned just before, can either be because at least one of their own neighbours belongs to that category, or their cache contained some entry for that subnetwork. Given the number of distinct categories, the fact that each node may belong to more than one of them and the average degree of the connections (around thirty) it is highly unprobable that this algorithm will fail to locate the necessary nodes, even with a TTL of one. In the pseudo-code of figure 4.6, *catList* is assumed to contain all the categories the node belongs to and its exists function is assumed to return a boolean value, depending on whether the category-argument exists in the *catList* or not.

Searching for data

After all appropriate nodes have been located, the querying node sends a standard Gnutella request to each node. The query is sent by first establishing a TCP connection with that node, which remains active until the requesting client receives enough results, or up to some timeout. If GUESS [1] is used, then the query can be sent via UDP.

Routing

Each node that receives a query, routes it in the standard Gnutella fashion. However, it does so only after it has itself categorized the query and then forwards it

only to those neighbours that belong to the category corresponding to the query. Notice that, as mentioned before, each connection can serve more than one categories, if both nodes at the end of the connection share more than one common categories-subnetworks.

4.3 Conclusions

In this chapter, we presented the implementation of a scheme that aims to reduce the cost of flooding in messages, by better directing the flood to nodes with appropriate content and avoiding to contact nodes without the type of information we want. One should notice that the low level functions used are the standard Gnutella protocol functions, making this scheme very easy to implement and integrate to the Gnutella network. In addition, the categorization scheme we used can easily be modified to include more or less categories, to improve the efficiency of the system. However, this task is part of the Evaluation phase of this project.

4.4 Introduction

This chapter contains the description of the implementation of two different solutions to prevent malicious action in an unstructured P2P system. As has been described in Chapter 5 of Deliverable 2.1, entitled ”System Security”, there is a number of issues regarding Security in unstructured P2P systems. The most important among them is the ability to insert malicious nodes, which can instrument the whole system to perform Denial of Service attacks to third parties.

In this chapter we describe the implementation of the algorithm for the detection of malicious nodes of a P2P system, which we analyzed in section 5.6 of Deliverable 2.1, as well as the implementation of SEALING. SEALING is a validation algorithm, which focuses on preventing Denial of Service attacks to third parties.

4.5 Implementation

4.5.1 Detection of Malicious Nodes

A malicious node of a P2P system is considered as the node that is able to provide service for every incoming request. For example, in a P2P system dedicated to file sharing, a malicious node will respond to every incoming Search Query and will redirect queriers to a victim node. This is the fundamental property, which is used in our algorithm in order to detect malicious nodes and isolate them from the system. Our strategy forces a legal peer to issue a Query with random search criteria and 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.

The algorithm is depicted in Fig. 4.7. We have implemented the function MaliciousDetector() (line 15), which is called during random time intervals. The largest time interval is 255 seconds (defined in line 1). When the MaliciousDetector() function is called, all neighbors are queried by generating Query packets with random search criteria (SendRndQuery(), line 3). Each random Query is stored in a dictionary (FakeQueries). When a QueryHit is received it is first checked in order to investigate if it is a QueryHit generated by one of our random Queries. If this is the case, we drop the connection (line 29) of the node which issued the QueryHit, since it is considered as a malicious one.

4.5.2 Short Term Safe Listing: the SEALING algorithm

This section describes the implementation of the SEALING algorithm, which has been evaluated in a real P2P System (Gnutella).

Our SEALING algorithm mainly focuses on protecting innocent victims such as non-Gnutella participants from DoS attacks originated from Gnutella. We consider a non-Gnutella participant as any host advertised to Gnutella (i.e. with an IP address and port number delivered in a Gnutella QueryHit) that does not support the Gnutella protocol. That is, the following Validation Criterion is used to distinguish between third parties that are potential victims of a DoS attack from normal Gnutella peers:

SEALING Validation Criterion:*Any host advertised in a Gnutella QueryHit packet which can not respond correctly to a Gnutella Handshake process is considered as a non-Gnutella participant and a potential victim for a Gnutella-based DoS attack.*

The SEALING algorithm is shown in Figure 4.8. The goal of the algorithm is to place potential DoS victims in a Safe List based on the SEALING Validation Criterion. This Safe List keeps track of machines that should not be contacted for downloads. Each Gnutella node keeps a Safe List and periodically updates its records. Each record has a lifetime of a fixed time interval. For the purposes of our evaluation, we used a fixed time interval of 30 minutes.

4.5.3 SEALING Evaluation

We attempt to evaluate the SEALING algorithm using the trace collected from the Middle DoS attack. We group the download requests by the IP address recorded by the Web Server during the attack. We consider the first download request as a download attempt that, according to SEALING, will fail since the Web Server will not respond correctly to the Gnutella Handshake. Based on SEALING, all the download requests following the first download and for the next 30 minutes will be filtered out by the Gnutella peer and eventually will not make it to the Web Server. That is, we assume that the Gnutella peer that received the QueryHit, will add the Web Server to its Safe List after failing to Handshake with it.

For every download request we find in the trace we compare its timestamp with the first one encountered in the trace, which serves as the time offset of the SEALING algorithm. If the timestamp of a download request is found to be over 30 minutes after the time offset, then we consider that the download request serves as a new Handshake, which will also eventually fail. Again, we filter out the next download requests we encounter in the trace that have relative time difference less than 30 minutes with the new time offset. The results of the evaluation, as shown in Figure 4.9, indicate that SEALING reduces the effectiveness of the DoS by roughly two orders of magnitude in terms of the number of download requests to the victim site. We believe that this is sufficient to downgrade the threat of Gnutella-based DoS attacks to the level of mere nuisance for the majority of potential victims.

4.5.4 Conclusions

In this chapter we presented the implementation of two different algorithms, which aim to prevent a P2P system to perform DoS attacks to computer machines, part of the P2P system infrastructure, or to computer machines that are not even aware of the existance of the P2P system.

The first algorithm tries to identify malicious nodes inside the P2P system, by forcing a node to periodically send random Queries, that should not generate any QueryHits, to its immedient neighbors. A node that replies to a random Query with a QueryHit is considered as malicious and it connection to the system is dropped. That is, our algorithm aims on preventing malicious nodes to have strong connectivity with the P2P system.

Our second algorithm focuses on preventing a P2P system to perform a DoS to a third party. This is acomplished by forcing every peer to validate that a node which advertizes a Service in the P2P system (for example it advertizes that it has files in a P2P file sharing system) is actually part of the P2P system. We evaluated our algorithm, SEALING, in a real world P2P system, Gnutella, and the results, as we see in **??**, are quite impressive and promissing.

Super – Category	Categories	Unicode Character Scripts
Armenian	Armenian	Armenian
		Armenian Ligatures
Cyrillic	Cyrillic	Cyrillic
		Cyrillic Supplement
Gregorian	Gregorian	Georgian
		Georgian Supplement
Greek	Greek	Greek
		Greek Extended
Latin	Latin	Basic Latin
		Latin - 1
		Latin Extended A
		Latin Extended B
	German – based	Ä, Ö
	Scandinavian	Å
	Danish – Norwegian	Æ, Ø
	Finnish	Š, Ž
	Spanish	Ñ
	Portuguese	Ã, Ẽ, Â, Ê, Ô
Arabic	Arabic	Arabic
		Arabic Supplement
		Arabic Presentation Forms A
		Arabic Presentation Forms B
Hebrew	Hebrew	Hebrew
		Hebrew Presentation Forms
Eastern Asian	Eastern Asian	Unified CJK Ideographs
		CJK Ideographs Ext. A
		CJK Ideographs Ext. B
	Chinese	Bopomofo Extended
	Japanese	Hiragana
		Katakana
		Katakana Phonetic Ext.
		Halfwidth Katakana
	Korean	Hangul Syllabes
		Hangul Jamo
		Hangul Compatibility Jamo
		Halfwidth Jamo

Figure 4.1: Unicode-based categories and the corresponding Unicode alphabets

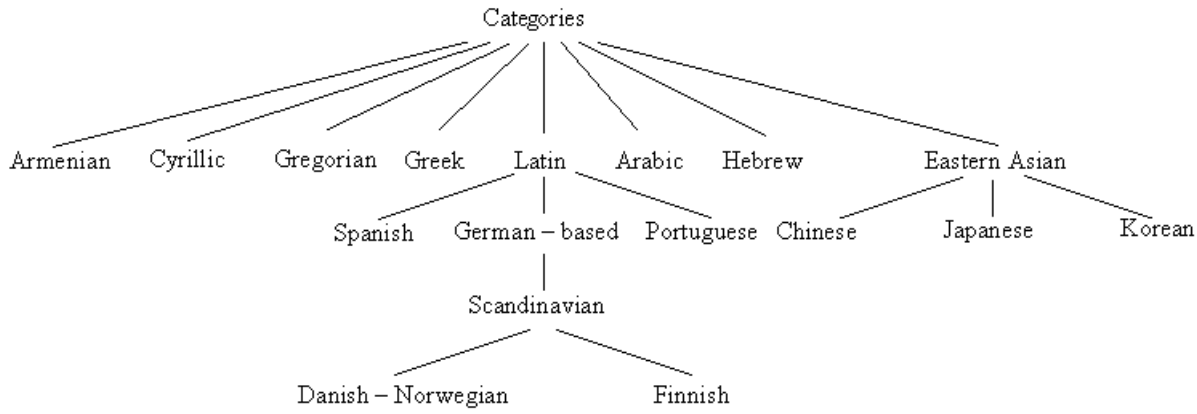

Figure 4.2: Hierarchies of the categories defined in figure 4.1

```
01  List getCategories(FileIndex)
```

Figure 4.3: Function getCategories. Returns the categories a node belongs to.

```
01  void SecSPeerBootstrap(webcaches[], int D)
02  {
03    List catList = getCategories(FileIndex);
04    foreach (Category c in catList)
05    {
06      int deg = D/catList.size();
07      if (mode[c] == "Leaf")
08        deg = max(deg, 3);
09      else
10        deg = min(deg, 3);
11      GnutellaBootstrap(webcache[c], deg);
12    }
13    pruneConnections();
14  }
```

Figure 4.4: Function SecSPeerBootstrap. Used to insert a node in the network.

```
01  void onLeafConnect(Node n)
02  {
03    List nodeList = getCategories(n.getFileIndex());
04    List catList = getCategories(this.getFileIndex());
05
06    List diff = catList.compare(nodeList);
07    if (catList.size() + diff.size() <= expLimit)
09      LeafConnect(n);
10    if (diff.size() > 0)
11      foreach (Category c in diff)
12      {
13        catList.add(c);
14        deg = min(D/catList.size(), 3);
15        GnutellaBootstrap(webcache[c], deg);
16      }
17    pruneConnections();
18  }
```

Figure 4.5: Function onLeafConnect. Used by an Ultrapeer whenever it receives a request for connection by a new Leaf node

```
01  Node findSubNetwork(Category c, int TTL)
02  {
03    if (this.catList.exists(c)) return this;
04    foreach (Node n in Neighbours)
05    {
06      if (n.catList.exists(c)) return n;
07    }
08    foreach (Node n in Neighbours)
09    {
10      Node result = n.findSubNetwork(c, TTL-1);
11      if (result != NULL) return result;
12    }
13    return NULL;
14  }
```

Figure 4.6: Function findSubNetwork. Used to locate a node belonging to some subnetwork.

```
01 #define MALICIOUS_DETECTOR_INTERVAL 255 /* Seconds.  */
02
03 function SendRndQuery(Node n) {
04    Query q;
05    String str;
06
07    str = String.new(Rand(64));
08    str.shuffle();
09    q = new Query(str);
10    q.TTL = 1;
11    q.hops = Rand(3);
12    n.SendQuery(q);
13 }
14
15 function MaliciousDetector(void) {
16    foreach (Neighbour n in Neighbours) {
17        FakeQueries.push(SendRndQuery(n));
18    }
19
20   alarm(Rand(MALICIOUS_DETECTOR_INTERVAL);
21 }
22
23 function onQueryHitReceive(Node n, QueryHit qh) {
24    GUID g;
25
26    g = QueryHitExtractGUID(qh);
27    /* Check if it is a QueryHit for a random Query.  */
28    if (FakeQueries.exists(g)) {
29        ConnectionDrop(n);
30    } else {
31        QueryHitProcess(n, qh);
32    }
33 }
34
35 int main(void) {
36    MainApp.AddSignalHandler(MaliciousDetector, SIG_ALARM);
37    alarm(1);
38 }
```

Figure 4.7:

```
0   SafeListLifeTime = 30 mins;
1   function if (GnutellaPacket(pkt) == QueryHitPacket) {
2       GnutellaExtractNode(pkt, &GnutellaNode);
3        if (SafeListContains(GnutellaNode)) {
4           if (CurrentTime() -
5               SafeListGetTimeOfNode(GnutellaNode) <
6               SafeListLifeTime)
7               GnutellaDropPacket(pkt);
8       }
9       else
10            GnutellaParseHits(pkt);
11  }
12  ...
13  function onDownloadAttempt(node, file) {
14      if (GnutellaHandShake(node))
15          GnutellaDownload(node, file);
16      else
17          SafeListAdd(node);
18  }
19  ...
```

Figure 4.8: SEALING Algorithm.

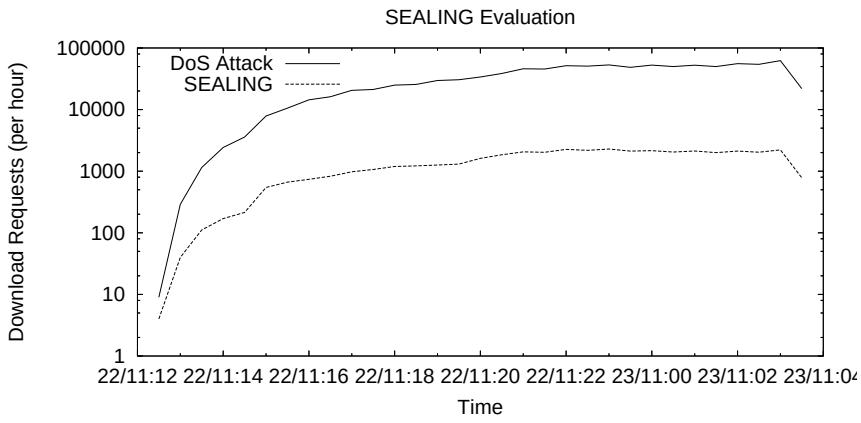

Figure 4.9: The evaluation graph of the SEALING algorithm. The solid curve represents the amount of download requests during a DoS attack using Gnutella. The dashed curve represents the amount of download requests that will be eventually exposed to a Web Server, if Gnutella nodes utilize the SEALING algorithm.

Chapter 5

Conclusions

This document presents the implementation of the SecSPeer system, a scalable and secure peer-to-peer system, which meets the requirements outlined in Deliverable 1.1: SYstem Requirements [6]. The work described in this and the previous deliverable (Deliverable 2.1) addresses both problems, namely the scalability and security of a global-scale P2P system. We have adopted an unstructured architecture, which makes the system robust in the face of arbitrary failures and more secure. This is because in unstructured P2P systems peers are trully ”expendable”, meaning that the arbitrary loss of any of them will not affect the functionality of the network in any way. In structured systems [10], [11], [7] however, although P2P systems themselves and as such, there is no peer with central responsibilities, still each peer depends on some other peers to store its information. This increased dependability among peers makes those systems less robust and more susceptible to attacks that can exploit the structure of the system. Additional security algorithms have been implemented to reduce the impact of the most important security threats in unstructured systems, namely spam and reflective DDos attacks. The increased robustness of an unstructured P2P system comes at the cost of increased number of messages required to perform a lookup in the system. This problem has also been addressed with the proposals outlined in chapters 3 and 4 of the Deliverable 2.1: System Design and implemented in the same chapters of Deliverable 3.1: System Implementation. Finally the system implemented in this document was made to easily inter-operate with existing unstructured P2P systems, thus facilitating its deployment and use.

References

[1] Guess specification. http://groups.yahoo.com/group/the gdf/files/proposals/guess/guess 01.txt. Technical report.

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

[3] BearShare. Bearshare, http://www.bearshare.com.

[4] Gtk-Gnutella. Gtk-gnutella, http://www.gtk-gnutella.com.

[5] LimeWire LLC. Limewire llc, http://www.limewire.com.

[6] E. P. Markatos. Secspeer deliverable 1.1: System requirements. 2005.

[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] H. Papadakis. Secspeer deliverable 2.1: System design. 2005.

[9] Christopher Rohrs. Query routing for the gnutella network. Technical report, Limewire LLC.

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

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

[13] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms, Second Edition*. The MIT Press, 2nd edition (September 1, 2001), 2001.