Dependable and Secure Storage in Pervasive Peer-to-Peer Systems

Rudi Ball, Vicki Spurrett
and Rogério de Lemos

Technical Report No. 11-06
September 2006

Dependable and Secure Storage in Pervasive Peer-to-Peer Systems

Rudi Ball, Vicki Spurrett, Rogério de Lemos
Computing Laboratory
University of Kent
Canterbury, Kent CT2 7NF, UK

ABSTRACT

This paper describes an approach to dependable and secure storage for pervasive systems based on the fragmentation-redundancy-scattering (FRS) technique. FRS is able to tolerate both accidental and intentional faults by fragmenting confidential information into insignificant fragments, and scattering these fragments in a redundant fashion across a system. The goal is that the original information that has been fragmented and scattered can only be reassembled by a trusted agent. The system architecture is based on the peer-to-peer architectural style in which each peer (agent) is able to request services for the storage of information, store information fragments, and forward to other agents those fragments that are to be stored elsewhere. The feasibility of the approach is demonstrated in the context of a multi-agent prototype implemented using IBM's Aglet System.

Keywords

Agents, Dependability, Security, Fragmentation-Replication-Scattering, Encryption, P2P architectural style.

1. INTRODUCTION

Current static file storage systems store a file in a single location. This presents the problem of availability, as when that single storage location fails all the data stored on it is inaccessible until the failure is corrected. If a file on the storage becomes corrupted, then that file is lost. The benefit of peer-to-peer filing systems is their ability to store files in many different locations. However, simple file storage in peer-to-peer systems does not enforce the replication of data throughout the system. Data stored in a single location is lost when the peer within the system is lost.

Users increasingly have many pervasive computing devices at their disposal with some of these devices containing reasonably large unused collections of storage capacity. Not only is there a proliferation of these devices, but also the mechanisms of communications and middleware between these devices have improved radically over the past decade. The increased connectivity of varying devices could provide increased and more available shared or pooled storage. Coupled with this, the usage and popularity of some decentralised data storage systems has grown with file sharing networks like Gnutella [9], information anonymity systems like Freenet [4] and Freehaven [7] and persistent storage systems like Oceanstore [11] being some of the examples. However the storage in some of these systems still requires the active knowledge of a storage location and data's attributes (such as capacity and location). When storing data in these types of systems, data needs to be actively directed to the storage location and when retrieving information, that data location needs to be actively found. Replicated information is more easily found.

Work on secure computing systems has focused mainly on intrusion prevention, that is, the means for preventing the occurrence of intrusions [15], and which is based on forecasting and preventing, as far as possible, the different intrusions that could damage overall system security. Such approaches become unfeasible in the context of open and decentralized systems containing a large number of components. Instead of attempting to prevent any type of intrusion, which in the context of ever changing environments may be very costly to achieve, some contributions have already been made in tolerating them. The basis for such approach is that, in case intrusions are successful, the whole system's security will not be compromised since the intrusions will be handled in the same way faults are tolerated. Although the term "intrusion tolerance" has been introduced a while ago [8], only recently there has been a rising interest in this area [16], which can be confirmed from several major projects, such as, MAFTIA, OASIS and ITUA, that have been doing groundwork into concepts, mechanisms and architectures.

Previous work has been done using fragmentation, redundancy and scattering (FRS) for file storage systems operated under the assumption of a static or unchanging system of nodes. This paper's work has sought to investigate FRS in the scope of a peer-to-peer system. Utilising a peer-to-peer architecture takes advantage of storage and computational resources available within a potentially large network. The system is assumed to change with nodes entering and exiting the system without warning. Connections are made between peers, without the expressed or complete knowledge of all peers within the system.

In this paper, we present an architectural solution for a dependable and secure storage system based on intrusion tolerance, which is application independent, and so it can be scalable to large pervasive systems. For such systems, some application dependent solutions, such as exception handling, are not amenable because of the costs involved in dealing with the combinatorial explosions of undesirable situations. The approach presented in this paper is based on the fragmentation-replication-scattering technique [6][8], which has been one of the approaches used as a basis for building intrusion-tolerant computing systems.
The rest of the paper is structured as follows. In Section 2, we present some background concepts in dependability and security, and describe the basis of the fragmentation-replication-scattering technique. Section 3 presents the overall architectural design of a dependable and secure storage system for pervasive systems. In Section 4, we describe an implementation of the proposed approach, and present some preliminary results of its evaluation. Related work is presented in Section 5. The last section presents some concluding remarks, and identifies future directions for research.

2. BACKGROUND

Before proceeding to discuss the details of a dependable and secure storage system, in this section we contextualize dependability and security, and provide some basic concepts associated with fragmentation-replication-scattering.

2.1 Dependability and Security

The dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable [1]. There is a service failure when the delivered service deviates from fulfilling the intended function. A failure can be characterized as a security failure when there is the violation of a security policy. A security policy is normally expressed through properties, which are related to the absence of unauthorized access to, or handling of, system state. System failures are caused by faults, and an intrusion is a malicious external fault that might lead to a security failure. There are two complementary means for the provision of dependability and security: fault prevention and fault tolerance. The former prevents the occurrence or introduction of faults, while the latter avoids service failures in the presence of faults. Thus intrusion tolerance is regarded as the process for tolerating the violation of a security policy. The fragmentation-redundancy-scattering technique, to be presented in more detail in the following section, is an example of intrusion tolerance, which uses fault tolerant techniques as a means to avoid system failures and the violation of security policies.

In the context of this work, a storage system is *dependable* if there is continuity (reliability) and readiness (availability) in the services it provides, and *secure* if it prevents the unauthorized disclosure of information (confidentiality), amendment or deletion of information (integrity), and withholding of information (availability).

2.2 Fragmentation-Replication-Scattering

One of the existing approaches for intrusion tolerance is fragmentation-redundancy-scattering (FRS) technique [6]. The aim of FRS is to tolerate both accidental and intentional faults/intrusions by fragmenting confidential information into insignificant fragments, and randomly scattering these fragments in a redundant fashion across nodes of a network. Fragments contain no significant information, so any intrusion into some part of the system only gives access to unrelated fragments, thus maintaining the confidentiality of the information (providing intrusion tolerance). By increasing the number of fragments a file is broken into, we can reduce the usefulness of a fragment, thereby improving the security of the system.

Before fragmenting, the original information is encoded and signed. Incorporating fragment digests may also protect the integrity of the information. Redundancy is added to tolerate accidental or deliberate destruction, or alteration of fragments. Moreover, in case some nodes suffer denial of service attacks, information fragments can always be retrieved from other nodes, depending on the existing failure assumptions. The complete information that has been fragmented can only be reassembled by an authenticated user in a trusted computing base [6]. The motivation behind the FRS technique is that an intruder attacking an individual node has no access to all fragments. Even if an intruder gets access to all *n* fragments, *n*!/2 cryptanalysis have to be performed to re-constitute the whole information.

3. DEPENDABLE AND SECURE STORAGE

The fragmentation-redundancy-scattering (FRS) technique has shown to be useful in the implementation of dependable and secure storage for decentralized systems. These implementations are based on the client-server model, which is used primarily to manage small scale collaborations. However, the challenge is how to apply FRS in the context of pervasive systems in which devices may have both client and server roles.

This paper considers objects or devices which exist within a pervasive system as agents. The interaction or co-operation of the multitude of agents is considered a multi-agent system.

In this section, firstly, the problem domain is scoped by providing some intuitive concepts concerning the application of FRS to multi-agent systems, and then the architectural description of an FRS agent will be described in detail.

3.1 FRS in a Pervasive Peer-to-Peer System

Figure 1 provides an example how the FRS technique could be applied to a multi-agent system, where not all agents are connected to other agents, and where agents are expected to provide services and require services from other agents. As depicted in the diagram of Figure 1, we have an agent assuming the role of client in node #1, and the rest of the agents, for illustration purposes, are considered to be storage agents, though they could assume the dual role of client and storage.

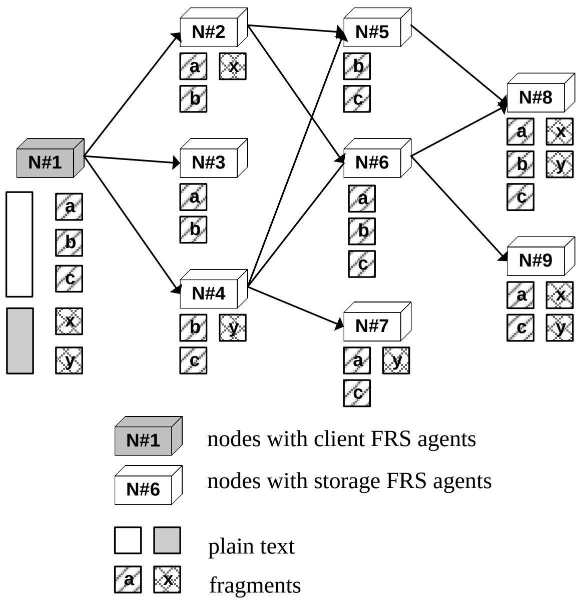

Figure 1. FRS applied to a multi-node system.

In this scenario, the client agent produces from the plain text information the fragments that are to be scattered within the system. These fragments will be multicast among the neighboring agents that will store and scatter them amongst their neighbors. This scattering process, based on multicast, relies on an agent sending a fragment to a randomly selected group of neighboring agents. Each fragments lifetime counter determines the scattering depth. Whether an agent stores a fragment depends on its storage policy. In the diagram of Figure 1, the fragments associated with two plain text files are scattered amongst several agents. It should not be a problem if all the fragments associated with a plain text file are stored in a single agent. The agent should not be able to distinguish which fragment is related to which plain text, and it should not be able to identify the order of the fragments.

Cycles are avoided by each storage node recording the occurrence of named fragments. Recording information concerning the scattering or retrieval of fragments could be seen as a failure point in the security of the system. When a sending peer seeks to retrieve a fragment, a requesting peer's fragment name is recorded by a receiving peer so as to combat the likelihood of cycles occurring within the system. Any requested fragment name previously recorded is not forwarded on to other peers. During the scattering of fragments, an agent should know which agents have forwarded fragments for avoiding the fragments to be sent back, thus creating a loop. This information should be stored in the agent, and it should not be associated with the fragment. The process of scattering fragments will terminate when there are enough fragment replicas in the system. In the context of dynamic environments, there are no assurances whether this can be achieved. However, by exploiting the redundancies available in the system, it is assumed that all the fragments can eventually be retrieved. In a static system where the availability of resources is known, agents before scattering the fragments might enquire about their environment to find what resources are available. Concerning dynamic environments, it is assumed that both storage and communication are cheap. In mobile ad hoc networks this is not a reasonable assumption to be made, but for peer-to-peer networks the above assumption is perfectly acceptable. However, in order to deal with those systems that have limited resources several optimizations can be envisaged for reducing the processing and storage costs.

Although it is not represented in the diagram of Figure 1, the process for retrieving the fragments is very similar to that of scattering the fragments. When a client wants to retrieve a fragment, it sends a request to its neighboring nodes. An agent that receives that request will store its source. This allows an agent to send the fragments only to those agents that have requested it, thus forming a virtual path between the client requesting the fragment and the client storing it, and avoiding unnecessary loops to be formed when forwarding a request. By sending the fragment back to the agent that has requested it, the fragment eventually reaches the client. In case the client receives several copies of the same fragment, the client discards the additional copies. Once the client receives all fragments related to a particular plain text, it joins all the fragments. This is decrypted to obtain the original plain text.

In order to scope the problem, we consider in this paper only the operations for storing and retrieving fragments. A more complete approach would have considered other operations, such as, remove/delete fragments, update fragments, and query the environment of an agent for checking the availability of resources.

Authentication and authorization will not be considered in this paper, although they provide key support for FRS technique [6]. Authentication, in particular, is important for retrieving the information scattered in the system, and it would prevent an intruder to have access to the fragments and their order.

3.2 Architectural Representation

The architecture of a multi-agent system can be represented in terms of the peer-to-peer architectural style. In this style, any component can interact with other components for providing services to them or requesting their services [5]. Peers can play the role of both client and server by directly interacting among themselves. The main type of connectors in the peer-to-peer style is the invoke-procedure connector, which can encapsulate complex interaction protocols reflecting the communication that may exist between two or more collaborating peers. The services provided and required by the peers are described in terms of interfaces.

A fragmentation-replication-scattering (FRS) agent can be represented as component containing four interfaces, as represented in Figure 2(a). The provision of a dependable and secure storage by an FRSAgent is captured by the provided interface ds_storage. The provided and required interfaces f_storage capture the services associated with the storage and retrieval of fragments. Although authentication will not be discussed in the paper, we have nevertheless left the representation of the interface as a reference. In order to facilitate the description of the different services associated with an FRSAgent, this component has been specialized into two different components, as depicted in Figure 2 (b) and (c). In the following, we proceed to describe in more detail each of these components.

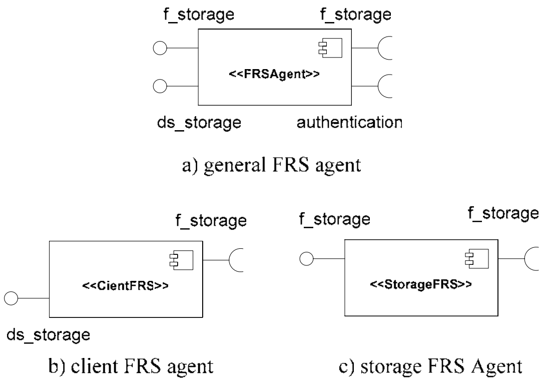

Figure 2. Peer architectural component.

3.2.1 Client Agent

The client agent is responsible for the fragmentation and scattering of the information, and the retrieval of information that has been previously scattered. Before fragmentation, the information is encrypted, and once the fragments are obtained a message digest is obtained for purpose of naming the fragment and checking the integrity of the fragment. The named fragments are subsequently sent to neighbouring peers in pseudo random order. The process of retrieving the information involves collecting the fragments, checking their integrity, assembling all the fragments in their initial order, and decrypting the collection. The design of the ClientFRS is shown in Figure 3.

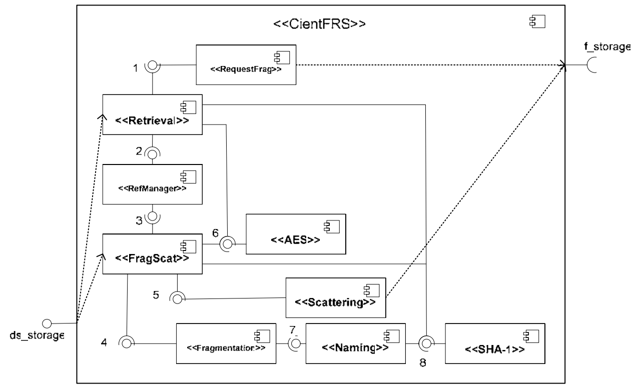

Figure 3. Client FRS Agent.

The main components of ClientFRS are the following. The FragScat is responsible for the encryption and scattering of the information to be stored. First the information is encrypted by the AES component, and then fragmented by Fragmentation. The name of each fragment is obtained from the digest of the fragment plus the information name and fragment number, done by SHA-1, which uses the keyed Secure Hash Algorithm-1 (SHA-1) – the probability of obtaining two distinct fragments with the same name is possible but unlikely. The Hash function used for naming and integrity is limited to the client or owner of the information. The integrity of fragments is not processed elsewhere in the storage system and hence the system relies on communications protocols to provide integrity checks on data during communication between storing peers. For the purposes of this implementation SHA-1 was used, due to improved provision of uniqueness and security. The hashing algorithm serves the purpose of providing a "unique" naming quality for fragments and is also useful for checking the integrity of the fragment when it is retrieved. Figure 4 shows the different data representations of plain text information - as it can be observed, no other information is appended to the fragments except for their name otherwise, vulnerabilities might be introduced. Before scattering the fragments, the sequence and the names of the fragments are stored in the RefManager. It is assumed that this data, which provides the basis for retrieving the original information from its fragments, is stored in a trusted computing base (TCB). The component Scattering is responsible for multicasting all the fragments amongst the neighbors of the client agent. The scattering of the fragments is performed randomly and on individual basis to prevent an intruder knowing the precise order and location of the fragments. In a situation in which an intruder might possess the secret key, the intruder does not know where the different fragments reside, or their order.

A notion of coverage was introduced to describe the subset or full-set of nodes to which data was multicast. Taking an example, coverage of 0.5 with 10 available storage nodes would require that a single fragment would be redundantly scattered to 5 (half) of the available storage nodes.

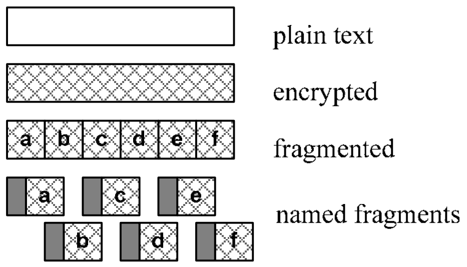

Figure 4. Data representation.

For retrieving the information stored in the system, the client through Retrieval has to access the RefManager to obtain the fragments' sequence and their names. The request for fragments is done through RequestFrag. Once Retrieval has obtained all the fragments, and after checking their integrity using MD5, all the fragments are put together. The original information is obtained by decrypting the collection of fragments. If additional assurances are necessary concerning the integrity of the original information, its digest could have been obtained before its fragmentation, and used at this point to check the integrity. If the integrity of the whole cannot be confirmed, all the fragments have to be discarded, and new fragments have to be requested. It is expected that the client might receive several copies of the same fragment, but once the original information is assembled and the integrity confirmed, these copies can be discarded.

```
Get Data_Source
Encrypt Data_Source
Divide Data_Source into Fragments of
constant length

FOR-EACH Fragment
     Associate Digest with Fragment
     record Digest in sequential order
END-FOR

FOR-EACH Fragment
     pseudo-randomly scatter fragments to
     subset of neighbouring peers according
     to policy of coverage
END-FOR
```

Figure 5. Pseudo-code describing the Client Scatter Process.

```
Generate random ordered number list of
recorded Digest size
FOR-EACH number in list
       Send Request for recorded numbered
       Digest
END-FOR

FOR-EACH MessageEvent
       IF (Message == ReturnType)
              Check Fragment Digest
              Record Fragment in order

              IF (HaveAllFragments == true)
                     Assemble Encrypted Data
                     Decrypt Encrypted Data
              END-IF

       ELSE-IF …

       END-IF
END-FOR
```

Figure 6. Pseudo-code describing the Client Retrieval Process.

3.2.2 Storage Agent

The storage agent is responsible for storing fragments, scattering fragments amongst other agents, forwarding requests for fragments, and returning fragments to the source of the request. The design of the StorageFRS is shown in figure 5.

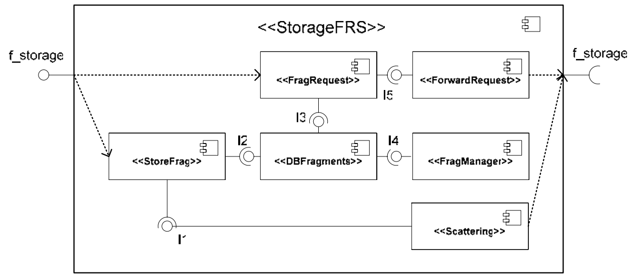

Figure 7. Storage FRS Agent.

The role of the main components of StorageFRS is the following. The StoreFrag when receiveing a request for storing a fragment, first checks whether a fragment with a same name already exist in the data base (DBFragments), if the fragment does not exist it stores the fragment in DBFragments. StoreFrag then sends the fragment to Scattering, for the fragment to be multicast to neighboring agents. FragManager manages the storage of fragments since they cannot be stored indefinitely in the data base. A policy is associated with the storae of fragments. Similarly, when there is a change in the storage policy, for example, reducing the amount of space allocated to fragment storage, the FragManager is responsible for identifying which fragments to remove.

The process of retrieving the fragments amongst agents is more complex, but very similar to routing messages in mobile ad hoc networks. For the purpose of this paper, we have adopted a very simple algorithm for retrieving and forwarding the fragments to the client FRS agent. When FragRequest receives a request for a fragment, FragRequest checks whether there is a fragment with that name in DBFragments. If there is one, it sends back the fragment to the agent that has requested it. FragRequest also sends the request to ForwardRequest for the request to be multicast among a set of random neighboring agents. (A potential optimization to this process would be for an agent not to forward the request if it stores the requested fragment. However, if the client receives a corrupted fragment, then the client would send a request that should be forwarded by all the agents that receives it.) This request is stored so the agent knows which downstream agents have made the request. The fragment should be sent only to those agents that have requested the fragment. By keeping a record of the requests made for fragments, it is a way for tracking which agents have actually requested the fragments. With this approach only those agents that have forwarded a request for a fragment will receive a copy of that fragment. In case the security of an agent is breached only partial information about the fragments is revealed i.e. the previous requestor of a fragment could be determined. For an intruder to obtain the source of the request, it has to breach the security of several agents – the further away the intrusion occurs, the harder it becomes to find the source of the request. If this is achieved, the intruder might attempt to destroy fragments that seem to be correlated by the same request. Likewise, this is the reason for multicasting fragments' requests, otherwise if they were to be broadcast, some agent would contain all the requests for fragments related to a particular data collection, thus introducing a major vulnerability into the system. Considering that no additional information can be appended to a fragment, the only way for retrieving a set of fragments is to establish a virtual path by recording the immediate source of a particular request, but not the original source of the request. In this way a collection of requestors is created to the location of a fragment (storage node), and the fragment traverses these links to return to the original source of the request.

```
IF (Message == ScatterType)
       Check Policy

       IF (Store == true)
              Store Fragment
       END-IF

       IF (Forward == true)
              Forward Fragment
       END-IF
END-IF
```

Figure 8. Pseudo-code describing the Storage Node Scatter Process.

```
IF Message == Request THEN
    IF Fragment Name Found Locally THEN
        Return Fragment To Requestor
    ELSE
        Record Request

        Associate Requestor with
        Fragment Name

        Modify Message Requestor as
        this Peer

        Forward Message to Neighbouring
        Peers
    END-IF
END-IF
```

Figure 9. Pseudo-code describing the Storage Node Retrieval Process.

4. IMPLEMENTATION AND EVALUATION

4.1 Implementation

A prototyping methodology was used to incrementally build upon the static system described by Deswarte et al. [6] to produce this P2P system.

For the encryption of information, we have employed the Advanced Encryption Standard (AES), which is a symmetric block cipher that uses a secret key encryption. Its combination of robustness, performance, efficiency, low memory requirements, ease of implementation and flexibility, make it desirable to use. AES supports key lengths of 128, 192 and 256-bit. There are two variants for 128-bit blocks: 128-bit key (likely to become the commercial norm), and 128-bit with 256-bit key. AES uses substitution, permutations and multiple rounds, the number of which depends of the key and block sizes (10 for 128/128 up to 14 for largest key or largest block). Its also uses repeat cycles, 9/11/13 for keys of 128/192/256-bit. The algorithm has been designed for achieving great security and speed, and is easily implemented on simple processors.

For the provision of the fragment digest, which is used for naming the fragments and to check their integrity, we have adopted the keyed Secure Hash Algorithm-1 (SHA-1), which produces a 160-bit digest providing robustness and improved uniqueness.

It should be reiterated that while the components for encryption and integrity were chosen in this prototype to be AES and SHA-1, these components could be changed, with ease, depending on the desired improvement of security and integrity required by the users of such a system. The prototype implementation was more interested in the interactions within the multi-agent system and the investigation of how data was stored and retrieved by collaborating agents within the system.

A HashMap was used for the storage of fragments on the storage agents providing complexity of O(1) access to stored fragments, while a Vector was used to store the reference list on the Client agent.

The prototype of the decentralized storage system based on the FRS technique was implemented using the Aglet System, developed by IBM Research Japan [10]. This is a simulation environment that facilitates the implementation of multi-agent systems. In this environment, aglets are the software instantiation of an agent. Each aglet could be considered a persistent object, such that they live and interact within the Aglet System until actively disposed of. In the simulation of the FRS-based storage system, agents were implemented as aglets. After an agent is created, whether a client or a storage agent, it checks its environment to identify neighboring agents. Since the environment is assumed to be dynamic, these checks are performed periodically for the agents to have an accurate view of their environment.

A limitation of the simulator was that the results viewed the system at a specific moment or state. The system was not autonomous in its action of repeatedly generating information, storing and retrieving that information in the system. The limitation was necessary to explore the sequential behaviour of the system with regards to the tests required.

Aglets communicate with one another using message objects. Messages are managed by a MessageManager built into the Aglet System. Each Message object in the Aglet System contains parameters for the message “Kind”, message “Arguments” and message “priority”. The Kind parameter was modified, acting like an address, to specify the targeted aglet a message was sent to.

For the implementation of the FRS-based storage system in terms of aglets, a Client, Storage and Tester were inherited from the Aglet class with overloaded methods. For the basic activities of encrypting, fragmenting and scattering fragments, the Client uses, respectively, the methods applyEncryption(String datasource), fragmentData(String encryptedString) and ReplicateAndScatterFragments(Fragment[] data), where the Fragment class was created as a temporary data structure to contain a fragment’s name and data payload. The Client retrieves fragments from the Storage by invoking retrieveFragments(). Identical to the scattering of fragments, their retrieval relies on the message handling provided by the Aglet class. In a similar manner, the Storage aglet makes full use of the multicasting behavior of the Aglet System. An additional class (Tester) was implemented to generate the required number of instances of Storage and Client aglets for setting up or administering the simulation of a multi-agent system.

4.2 Evaluation

To evaluate the decentralized peer-to-peer storage system based on fragmentation-replication-scattering (FRS) technique several hypotheses were made and tested. The hypotheses were as follows:

1. Increasing the number of fragments generated by the client from the initial data increases the number of messages generated in the system. This also increases the amount of time taken to scatter and retrieve the fragments.
2. Restricting the node storage increases the retrieval time and the number of messages generated in the system during the scattering and retrieval phases.

3. If there are n replicas of a fragment, n-1 copies can be corrupted without affecting the ability of the client to retrieve the data from the system.
4. Increasing the value of the fragment lifetime counter will increase the amount of time taken to complete the scattering process.

Scalability was measured as the efficiency of the system with increased numbers of components added; hence we would expect a similar performance for multiple collection of resources provided to the system. The primary variables which affected the behaviour of the system included: a) the number of nodes in the system, b) the number of fragments required to be stored in the system, c) the variation of coverage (broadcasting or multicasting), d) message limitation methods (message lifetime or circular path limit), e) fragment integrity and f) node failure. A secondary affect was noted by the Aglet System on the performance of the P2P system.

The systems' ability to cope with the loss of all but one copy of a fragment, demonstrated the dependability of the system.

From the perspective of security, the attributes of confidentiality and integrity are established by the intrinsic robustness of AES and SHA-1, respectively. One important security attribute that could have been tested was the availability from the perspective of denial-of-service attacks. This type of analysis, however, would be more related to a qualitative evaluation – through, for example, threat analysis, rather than the quantitative evaluation performed in this work. All the simulations using the Aglet System were performed on a Sun Microsystems Enterprise E450-CA Server with 900MHz Quad Processors. The experiments were executed between five and ten times to obtain significant statistical data. In the following, we present some of the results obtained. For each experiment performed, the duration of the scattering process was timed from when the client sent the first fragment out to be stored to the latest time that a storage node stated that is had received a storage request. Once the storage nodes stopped reporting that they have received storage requests, it was assumed that there were no more messages traveling within the system. When an agent receives a fragment, whether that fragment is stored depends on the agent's storage policy, but the agent has to forward that fragment to a subset or full set of neighbouring agents, depending on the lifetime counter of the fragment. The number of replicas produced depends on the resources available in the system, and this is not known when a client starts scattering a fragment – it is assumed that we are dealing with very dynamic environments. The lifetime counter removed the chance of flooding the system with replicas of a specific message. Figure 10 shows the number of fragments generated in the system, which is the total number of fragments that were forwarded by each agent.

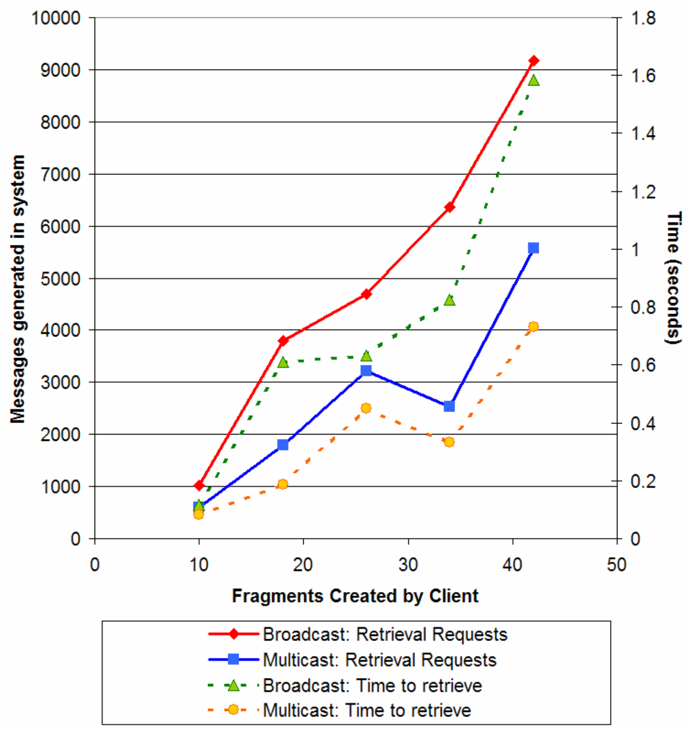

Figure 10. Number of fragments generated in the system.

In Figure 10 the experiments, the number of storage nodes was kept constant (20) for each test. The lifetime counter of each fragment was set to a small value (4). Initially, coverage was set to 1.0 to make the system broadcast messages. The tests were run then the coverage was lowered to 0.5 and the tests were run again. Figure 10 illustrates the results for this experiment, comparing the results for coverage of 1.0 (broadcast) to coverage of 0.5 (multicast). It shows the number of messages generated in the system during the retrieval phase and also shows the time it takes to perform the retrieval. Figure 10 also shows that increasing the number of fragments increases the amount of time it takes for the client to retrieve those fragments. It also increases the number of messages generated in the system during the retrieval process. This was consistent with the hypothesis. The results for scattering produced a similar graph as that in Figure 10. Broadcasting requests resulted in both a longer retrieval process and the generation of more messages in the system than when requests were multicast. It was found that restricting a storage node's capacity increased the retrieval time and the number of messages generated in the system during the scattering and retrieval phases. To investigate this phenomenon the number of storage nodes used in each test was 10 and the counter of each storage message was set to 4. The number of unique fragments generated by the client was 10. The probability that any storage node would store a fragment received was varied from 0.1 to 1.0. A probability of 1.0 would signify that all the received fragments would be stored.

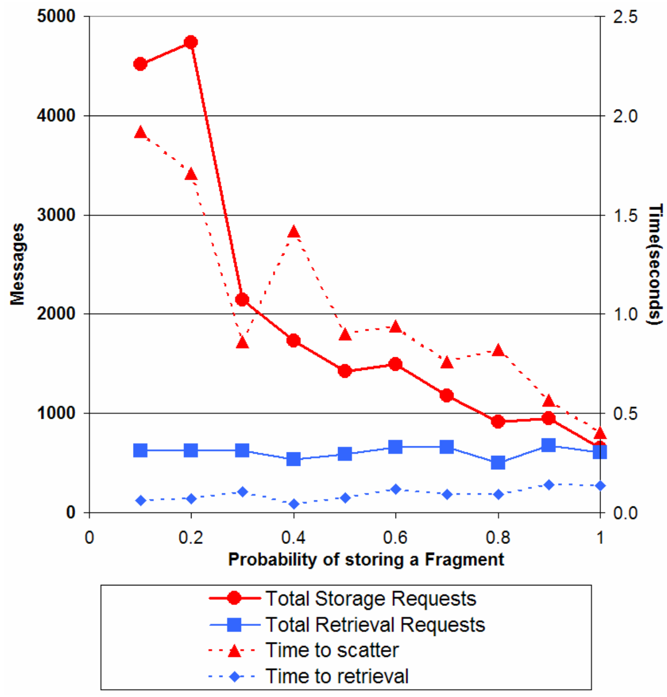

Figure 11. Increasing the probability of storing a fragment.

By increasing the probability of fragment storage the total number of storage requests generated in the system was reduced (Figure 11). There was no substantial effect on the number of retrieval requests, as retrieval requests were not linked to the storage probability in any way. The trend in the time to complete the scattering and retrieval processes is almost identical to that for the number of requests, although the time to scatter appeared to be quite erratic. To verify the dependability of the system all but a single replica of a fragment were deleted within the system at random locations within the system. The results illustrated that the system could retrieve the correct fragment as the data was successfully reconstructed (Figure 12). Had the final fragment not been found within the system, the retrieval time would have tended towards infinity.

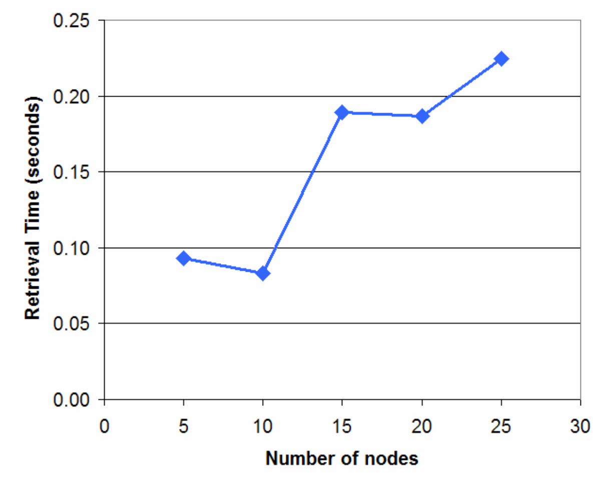

Figure 12. Corruption of all but a single copy of a fragment.

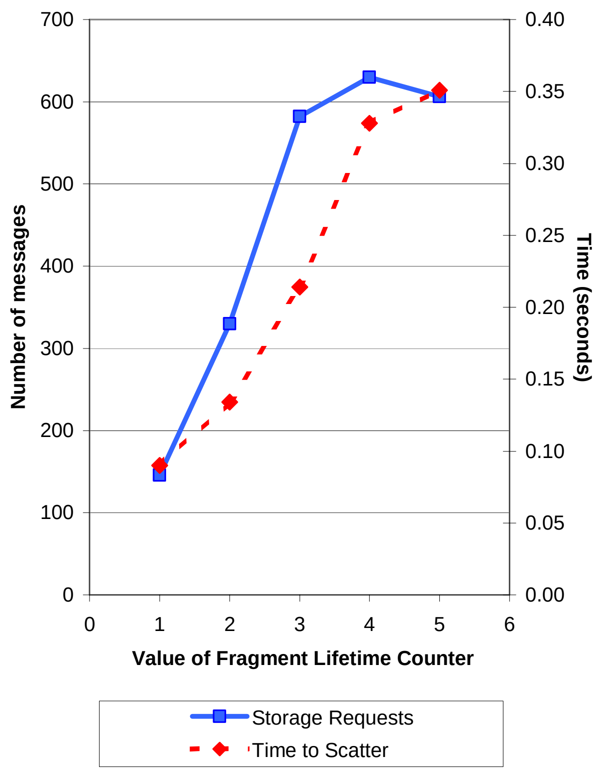

Figure 13. Varying the fragment lifetime counter.

It was found that the system could handle fault similarly as long as a reasonable number of nodes were still available and at least one replica of each fragment was still stored within the system, the system could find those replicas for data retrieval. The more failed nodes in the system, the less time retrieval took, due to the decreased multiplicity of messages. To investigate whether the increasing the value of the fragment lifetime counter would increase the amount of time taken to complete the scattering process, the number of storage nodes was kept constant, as was the number of unique fragments generated by the client. The lifetime fragment counter was increased from 1 to 5 (Figure 13). As the fragment lifetime counter was increased, the number of fragments generated in the system during the scattering process increased. The time taken for the system to return to a stable state (no further communication) also increased. With both broadcasting and multicasting, the intermediate performance improvement exhibited by both limiting broadcasting and multicasting, produced a trade-off of relative availability of fragments in the system. An advantage was found whereby limiting the lifetime to low values reduced load on the network resources (improving performance of the network); however the disadvantage was a loss of replication in the system and a limiting effect on the retrieval of data within the system, as the scattering and searchable depth of nodes was reduced (fragments could only lie a certain distance of hops from the client). In conclusion the system was found to be of poor performance even for small numbers of initial fragments stored in the system and non-scalable as the efficiency of the system was not constant for increasing sized systems simulated.

5. RELATED WORK

The idea of having a file system based on a decentralized solution is not new. File availability, confidentiality, and integrity on large-scale distributed file systems usually rely on the fragmentation of a file, and the scattering of these fragments among different nodes of the system [8][13]. The fragmentation principle, as already mentioned above, is based on splitting an encrypted file into fragments, where each fragment is then distributed. There are two different schemes based on fragmentation and scattering: fragmentation-scattering and replication [8][6], and fragmentation-scattering and threshold [15]. In both schemes, a file read accesses a subset of the fragments, while a file write has to be performed on all fragments. In fragmentation-replication-scattering (FRS), fragments of information with little value to a potential intruder are replicated and scattered across a large number of nodes. The approach presented in this paper based on the peer-to-peer models was inspired by [6], although seminal work was presented in [8]. The major disadvantage of this approach is the communication overheads. Moreover, most of work related to this scheme has focused on the client-server model, which is used primarily to manage small scale collaborations. The fragmentation-threshold-scattering (FTS) scheme is based on the same principles of threshold cryptography [14]. Instead of replication, this scheme relies on the processing of information. Seminal work in this area is the Information Dispersal Algorithm (IDA) [13]. The algorithm reliably disperses the information in a file into *n* fragments, across *n* nodes. The file can be reconstructed from any *m* fragments. This approach can be viewed as belonging to the field of error correction codes, in which extra bits are added to a piece of information, so that if there are *k* errors within that information, the information can still be reconstructed. The FTS scheme contains some redundancy for tolerating both accidental and intentional faults with respect to integrity, but confidentiality is not clearly addressed. It is claimed that the scattering of fragments and the reconstruction of information can be made space efficient if $n/m \sim 1$. However, since threshold schema is based on polynomials, it may become computationally expensive. In the following, we summarize some recent approaches that appear in the literature that resemble the peer-to-peer architectural solution presented in this paper. In terms of fragments distribution, a dynamic allocation algorithm has been proposed in which fragments are moved between servers for achieving maximal assurance [10]. In order to avoid a single server to compromise the integrity of a file, a restriction was introduced in the movement of the fragments: no fragment can go through a server that either holds or has hold another fragment from the same file. In the context of the proposed approach, such a problem would not exist because there is no correlation among the fragments' names of a particular file. However, an intruder might attempt to destroy all fragments of the same name.

The Information Dispersal Algorithm (IDA) [13], proposed by Rabin, has been considered in the context of Redundant Residue Number System (RRNS) for encoding information [3], which provides uniform coverage of both erasures and errors. The objective of this work is to provide a dependable and secure data storage (DS^2) to mobile wireless networks. It has been shown that this approach and IDA have almost the same performance in terms of code efficiency and complexity [2], even though DS^2 provides richer security features than IDA by exploiting the RRNS codes. An apparent drawback of this approach is that the system has to be reasonably static for it to be feasible. It is difficult to envisage the application of DS^2 to extremely dynamic ad hoc mobile networks where large number of nodes can be joining or leaving the system at the same time. The solution presented for these possible scenarios was for client/user to adopt an appropriate level of redundancy during the creation of the file. Other work on decentralised storage peer-to-peer systems includes Gnutella [9], Freenet [4], Oceanstore [11] and Freehaven [7]. Gnutella uses the expensive flooding technique to find data in a system. A broadcasting approach using what is referred to as Ping-Pong and Query/Push routing is used for file searching, a costly approach to network resources. A hop counter is attached to every request where the hop counter is decremented at each location where a query is made, reducing the lifetime or search depth of a request. When the broadcast request find the file being search a direct peer-to-peer communication is made between the client and the holder of that file for download. Oceanstore seeks to provide access to persistent information stores from anywhere in the globe. With Oceanstore a fee is charged for access to the persistant storage. The data is then highly available from anywhere in the network, with automatic replication to allow for disaster recovery and strong security as a default. OceanStore is constructed from an untrusted infrastructure, i.e.: it consists of unreliable servers. The system uses cryptography and redundancy to protect data. The system monitors itself to improve its performance and guards against denial of service attacks and failure of nodes. Their method of naming also seeks unique names and information migrates to where it is needed in the form of a cache. Routing information is however transferred between nodes, replicas are managed and the updating of data stored in the system is achieved using multicasting. Freehaven seeks to provide an anonymous publication system, through information trading. The principles of adding and retrieving documents from the system are made anonymously. The system is considered dynamic where the frequent trading makes the failure of nodes "transparent". Freenet is a P2P system designed for the provision freedom of speech using anonymity. All users donate a portion of their hard drives for the storage of files provided by other freenet users. Users are encouraged to encrypt their information before inserting it into the freenet network, but it is not mandatory. The aim is that information not be censored using decentralisation for the improving of system availability, reliability and tolerance to failure. Data is encrypted for confidentiality and the network's routing is trained over a period of time, using spiders for finding information, but also allowing information to be published by individuals as indexed bookmarks.

All of these systems contain various components which make up the P2P system we employed, however they are all differing in their approaches to data scattering and retrieval and not all seek to define assurances for dependability and the security of data in the relevant systems.

6. CONCLUSIONS AND FUTURE WORK

This paper has proposed an approach to a decentralised dependable and secure P2P storage system. The system utilises the mechanisms of fragmentation, replication and scattering (FRS), encryption block ciphers (AES) and cryptographic hash functions (SHA-1) to achieve this aim. This technique relies on encoding and fragmenting a piece of information, whose fragments, subsequently, have to be scattered and replicated among the nodes of a system. The motivation for investigating the FRS technique in multi-agent systems was to evaluate the performance of FRS in very dynamic environments in which it is difficult to establish a stable system configuration. For dealing with this limitation, one of the potential solutions is to explore redundancies available in the system. But for that, it is required to flood the system with replicas, in order to be able to recover the original information even in the presence of a high number of malicious or accidental faults. There is a cost associated with such an approach, and purpose of this work was to have a preliminary insight of this cost.

The paper, firstly describes an FRS-based decentralized storage system for large multi-agent systems in terms of the peer-to-peer architectural style, and then evaluates the feasibility of the proposed approach through a prototype implemented using Aglet System. It was claimed that the storage system should be both dependable and secure, so in the following we briefly analyze the compliance of the implemented approach towards key dependable and secure attributes. From the dependability perspective, reliability and availability were achieved by replicating fragments amongst several agents, which provides assurances that if at least one replica of each fragment is obtained, then original file or information can be retrieved. From the security perspective, confidentiality was achieved by ciphering the file or information to be stored, which provides assurances that only those which have the secret key are able to access the encoded file or information; integrity was achieved in two ways, first, by signing the file or information to be stored, which provides assurances that only the original file or information is retrieved, though different fragments may remotely have the same name; availability was achieved by replicating fragments, which provides assurances that in case a denial of service attack on some agents, for it to be successful all agents containing a particular fragment should be attacked, which should be unlikely because nowhere in the system such information exists.

Another benefit of this P2P storage approach is that fragments stored have the side effect of being anonymous to all nodes of a system, except the client, which owns the fragment. With increased usage of such a model it could be expected that the security of data could be improved through the sheer volume of fragments which reside from different clients within the system. Every storage node if attacked holds obscure and meager information on the data which it is storing. However, all the above good qualities come with a price, there is a very high communication overhead associated with the scattering and retrieval of fragments. Since the approach attempts to exploit all the available redundancies in the system, depending on the size of the system it might be the case that there is no end to the process of scattering fragments and requests, thus overloading the communication system with messages.

In terms of future work, a great deal remains to be done since the work presented in this paper has provided a preliminary insight on the effectiveness of the fragmentation-replication-scattering (FRS) technique when designing dependable and secure storage for large multi-agent systems. First and foremost, the other file operations, such as, file updating and removal, should also be implemented and evaluated. Also missing are proper schemes for optimizing the scattering of fragments and their request depending on the resources available in the system. A way in which the ReferenceManager should be dependably and securely stored it needs also to be investigated. Above all, the evaluations of the presented approach should be re-done, since the Aglet System has shown not to be scalable for the type of analyses that are necessary for properly evaluate the proposed approach.

Future work could explore optimisation techniques for the system. Scattering while expensive is extremely successful in enforcing the redundancy of fragments throughout the system (placing few performance requirements of the client). The retrieval of fragments could be improved such that searches for fragments are made not with the present exhaustive flooding search routine, but with a more controlled and intelligent probing methodology, such as a simple depth first search or a trained searching model. A more limited search routine would facilitate the better usage of resources and perhaps while being slower to recollect the original data, be less resource intensive.

REFERENCES

[1] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr. “Basic Concepts and Taxonomy of Dependable and Secure Computing”. *IEEE Transactions on Dependable and Secure Computing 1(1)*. January-March 2004. pp. 11-33.

[2] S. Chessa, R. Di Pietro, P. Maestrini. “Dependable and Secure Data Storage in Wireless Ad Hoc Networks: An Assessment of DS^2”. *Proceedings of First IFIP TC6 Working Conference Wireless On-Demand Network Systems (WONS 2004)*. Lecture Notes in Computer Science 2928. Springer. Berlin, Germany. 2004. pp. 184 - 198

[3] S. Chessa, P. Maestrini. “Dependable and Secure Data Storage and Retrieval in Mobile, Wireless Network”. *Proceedings of the International Conference on Dependable System and Networks (DSN 2003)*. San Francisco, CA, USA. 2003

[4] Clark I., Sandberg O., Wiley B., and Hong T.W., “Freenet: A Distributed Anonymous Information Storage and Retrieval System”, in Hannes Federrath (ed), *“Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability”*, LNCS 2009, Springer, 2001.

[5] P. Clements, et al. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley. 2003.

[6] Y. Deswarte, L. Blain, J.-C. Fabre. “Intrusion Tolerance in Distributed Computing Systems”. *Proceedings of the IEEE Symposium on Security and Privacy*. Oakland, California, USA. May 1991. pp. 110-121.

[7] Dingledine R., Freedman M.J., Molnar D., "The Free Haven Project: Distributed Anonymous Storage Service", in *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000 (LNCS 2009).

[8] J. Fraga, D. Powell. “A Fault- and Intrusion-Tolerant File System”. *Proceedings of the 3rd International Conference on Computer Security (IFIP/SEC'85)*. Dublin, Ireland. August 1985. pp. 203-218.

[9] Gnutella. "The Gnutella Protocol Sepcification v0.4". Online: http://rfc-

gnutella.sourceforge.net/Development/GnutellaProtocol0_4-rev1_2.pdf

[10] D. B. Lange, M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley. Reading, MA, USA. 1998.

[11] Kubiatowicz J., Bindel D., Chen Y., Czerwinski S., Eaton P., Geels D., Gummadi R., Rhea S., Weatherspoon H., Weimer W., Wells C., and Zhao B., "OceanStore: An Architecture for Global-Scale Persistent Storage", appears in *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000),* November 2000.

[12] A. Mei, L. V. Mancini, S. Jajodia. "Secure Dynamic Fragment and Replica Allocation in Large-Scale Distributed File Systems". *IEEE Transactions on Parallel and Distributed Systems 14(9)*. September 2003. pp. 885-896.

[13] M.O. Rabin. "Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance". *Journal of the ACM 36(2)*. 1989.

[14] A. Shamir. "How to Share a Secret". *Communications of the ACM 22(11)*. November 1979. pp. 612-613.

[15] G. Trouessin, Y. Deswarte, J.-C. Fabre, B. Randell. "Improvement of Data Processing Security by Means of Fault Tolerance". *Proceedings of the 14th National Computer Security Conference*. Washington, USA. 1991. pp. 295–304.

[16] P. Veríssimo, N. F. Neves, M.Correia. *Intrusion-Tolerant Architectures: Concepts and Design*. Technical Report DI/FCUL TR03-5. Department of Computer Science. University of Lisbon. 2003.