# Secure File Allocation and Caching in Large-Scale Distributed Systems

Alessio Di Mauro[1], Alessandro Mei[2] and Sushil Jajodia[3]
[1]*DTU Informatics, Technical University of Denmark, Denmark*
[2]*Department of Computer Science, Sapienza University of Rome, Italy*
[3]*Center for Secure Information Systems, George Mason University, Fairfax*
*adma@imm.dtu.dk, mei@di.uniroma1.it, jajodia@gmu.edu*

Keywords: Load Balancing, Distributed Systems, Secure File Allocation

Abstract: In this paper, we present a file allocation and caching scheme that guarantees high assurance, availability, and load balancing in a large-scale distributed file system that can support dynamic updates of authorization policies. The scheme uses fragmentation and replication to store files with high security requirements in a system composed of a majority of low-security servers. We develop mechanisms to fragment files, to allocate them into multiple servers, and to cache them as close as possible to their readers while preserving the security requirement of the files, providing load-balancing, and reducing delay of read operations. The system offers a trade-off between performance and security that is dynamically tunable according to the current level of threat. We validate our mechanisms with extensive simulations in an Internet-like network.

## 1 INTRODUCTION

The ever-increasing resources and performances of modern technological systems have massively contributed to the development of applications that allow for resources to be easily and widely distributed. Scenarios where contents are pervasively shared, ubiquitously accessible and always available have become more and more common. Grid computing and cloud computing solutions make intensive use of these concepts in order to provide high-performance and scalable services.

Security aspects have therefore become even more important, confidentiality issues can make or break commercial applications. Resources such as files should be readily and constantly available from anywhere to whoever has the right to access them, and safely kept from those who do not have such rights. Distributed file systems such as OceanStore (Kubiatowicz et al., 2000), PASIS (Wylie et al., 2000) or Tahoe (Wilcox-O'Hearn and Warner, 2008) try to satisfy these properties. Different approaches and techniques are used to address different security aspects. Confidentiality is usually achieved through encryption, files are enciphered before being stored. Availability is generally obtained through replication of encrypted replicas of the file. However, file encryption makes it difficult to get high security and support for updates to the authorization policy at the same time. Indeed, if the encryption is performed with a single key only, either every user has a complete view of the files stored in the system or costly re-encryptions are needed at every change of the authorization policy. Moreover, if a legitimate user leaves the system, rekeying and (again) re-encryption are needed to provide forward-security. In order to avoid such problems and maintain good performance, dynamic policies are often not allowed, in this way operations like re-encryption are not needed. The price of this is paid in terms of flexibility. For example not being able to dynamically redefine the access policy for a file requires a more accurate planning when the access lists are created and, in any case, makes it impossible to cope with emergencies in critical applications.

In this paper we present a model for file allocation in large-scale distributed systems that addresses these problems. We provide provable assurance in a system that guarantees flexibility of the authorization policy. In our system, files are fragmented and stored in the clear at several nodes (or, similarly to what is done in (di Vimercati et al., 2007), stored with a base encryption layer whose key is known to all the possible legitimate users of the system). We characterize the system as highly heterogeneous, composed by elements whose characteristics and performance can vary significantly.

We then present a fragmentation and allocation scheme for storing files within the system. This scheme is compliant to our provable assurance model and produces overall load balance. The allocation procedure is *efficient* in the sense that very few and locally obtainable information are needed to perform the task. Furthermore, we allow the existence of files with assurance requirements much greater than the resilience guaranteed by each single node in the system. Lastly, we introduce a read-based, threat-adaptive, caching system that guarantees good load-balancing and high availability (i.e. low delay) for the readers.

## 2 RELATED WORK

Distributed file allocation has been addressed in many different ways. The Cooperative File System (CFS) (Dabek et al., 2001) uses DHT to provide provable efficiency, robustness and load-balance in a read-only context. FarSite (Adya et al., 2002) achieves Byzantine fault tolerance, secrecy, and scalability by adding an encryption layer and local file caching. Similarly OceanStore (Kubiatowicz et al., 2000) allows the existence of untrusted servers and yields continuous access thanks to encryption and redundancy. It also allows to improve performances and avoid DOS attacks by monitoring the usage patterns. Again, PASIS (Wylie et al., 2000) is a Survivable Storage System that builds on decentralized storage infrastructures to offer confidentiality, integrity, and availability. In (Tu et al., 2010) the authors use partitioning and replication in a two-level topology context. They propose an algorithm to find a subset of servers to allocate replicas to that would minimize the number of exchanged messages. The work in (Ye et al., 2010) instead focuses on a scenario that does not make use of a pure data partitioning approach in order to minimize the propagation latency and consistency verification messages. They provide a two-level storage system where servers are divided into groups according to their location and use secret sharing schemes to spread the shares within a group and lazy update to propagate the information within different groups. The authors in (Lakshmanan et al., 2003) present a distributed data store system that makes use of secret sharing and replication. This allows for the existence of a partial number of compromised nodes (Byzantine failures). Nevertheless the shares are not allowed to be moved and optimal allocation is not considered.

## 3 SYSTEM MODEL

We consider a distributed system $\mathcal{D}$. The system is hierarchically organized in a set $\mathcal{S}$ of subnets, each subnet $s \in \mathcal{S}$ consisting of a number of nodes. The set of nodes of a subnet $s$ is denoted by $\mathcal{N}_s$, and the set of all the nodes of the system is denoted by $\mathcal{N}$. The system is Internet-like for both size and typical usage. The concept of node can be assimilated to a generic host, while we can picture a subnet as a collection of hosts under the same administration domain and with local resources (e.g. firewalls). At a much greater scale, we can use the same model to represent a system in which subnets are autonomous systems with their own border routers and access policies.

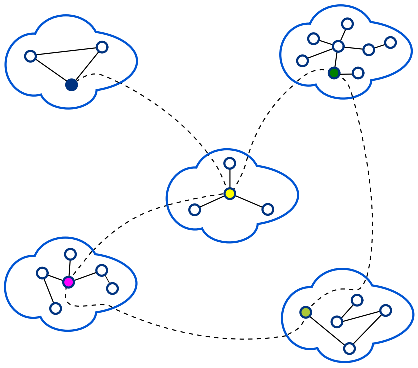

Figure 1: System structure.

The nodes in each subnet are internally interconnected. We assume that every two nodes of the same subnet can communicate directly by exchanging messages within the subnet itself. Subnets are also interconnected according to a *graph of subnets*. Two nodes of different subnets can communicate as well, but messages have to traverse the links of the graph of subnets and get into the subnet of the destination by going through its filters and protections (like, for example, a firewall). Both nodes and subnets have their own unique id, just like their Internet counterparts have a unique IP address. Figure 1 summarizes the structure of the distributed system.

Nodes and subnets are entities able to *store* and *protect* resources. In order to formalize this concept, we introduce the function $\mathcal{R}$ which defines the *resilience* provided by a node or by a subnet. For each node $u$, $\mathcal{R}(u)$ is a numeric value that represents the quantity of (ideal) effort needed by an attacker to crack node $u$. We assume that the system is heterogeneous, consisting in nodes with different resilience. A subnet is not just an hypothetical component created only for convenience, it is a real part of the system. If we recall the comparison between subnets and AS, we can see that a subnet could for example deny unauthorized incoming connection or spot identity theft attempts. Shortly we can say that subnets have a rather active role in the whole system dynamics. For this reason we add a resilience value to every subnet. For ease of work we chose this value $\mathcal{R}(s)$ to be the same for every subnet. Furthermore, we do not provide a real implementation of $\mathcal{R}$ as it would be too much system specific, anyway we think that such function could be defined by taking into account all the parameters of a given system such as the hardware equipment of a node, the procedural security of the site where the nodes are, the IT security aspects (e.g. firewalls, network organization, ...), etc.

Another main component of our model are the files, defined by the set $\mathcal{F}$. We assume that the files are *read-only* and that different files have different requirements in terms of confidentiality. To express this idea we introduce the function $\mathcal{H}$. For each file $f \in \mathcal{F}$, the value $\mathcal{H}(f)$ is defined as the *assurance requirement* of file $f$. In other words, we want the system to guarantee that the adversary cannot compromise file $f$ unless it spends at least effort $\mathcal{H}(f)$. Files are created by the *users* of the systems. Users can be servers of a cloud, mobile users that connect to the system to read shared files, or any similar entity depending on the application scenario. We assume that users log into one of the nodes of the system, typically one node in the closest subnet. When a user creates a file, it assigns the file an assurance requirement $\mathcal{H}(f)$, an access policy that depends on the application, and stores the file in the system.

Nodes and users have access to a trusted public key cryptography infrastructure. Therefore, we assume that every node and every user have a certified public key and the corresponding private key.

### 3.1 The Adversary

The adversary has the goal of compromising one or more files in the system. To achieve this goal, the adversary can break into subnets and compromise nodes. However, the adversary has to pay effort $\mathcal{R}(s)$ to break into subnet $s$, and effort $\mathcal{R}(u)$ to compromise node $u$. When a subnet or a node are compromised, they are so forever—there is no recovery procedure. We also assume that the adversary has a complete knowledge of file allocation—if it has the goal of compromising file $f$, it knows which nodes are used to store $f$. The adversary compromises all the fragments that are stored, even temporarily, in nodes that are currently under its control. A file is compromised when at least one copy of every fragment if has been divided into, is compromised.

### 3.2 The Problem

The problem that we consider in this paper is to design a file allocation and caching scheme with the following goals:

1. We should be able to securely store files whose resilience requirement is larger than the resilience of any subnet and of any node in the system;
2. performance of read operations should be maximized;
3. the system load should be balanced;
4. the adversary cannot compromise file $f$ by making an effort smaller than $\mathcal{R}(f)$.

To get Property 2, files can be dynamically replicated in more than one node in the system or moved from one node to another according to the read activities of the users. The challenge is to preserve security requirements and, at the same time, to perform efficient caching and load-balancing.

## 4 ALLOCATION AND FRAGMENTATION

One of the ideas in this study is to design a file allocation and caching mechanism that can obtain constantly high standards of security from a system with many weak nodes, a few stronger ones, and a very small number of high resilience nodes. To do so we push the limits of the system by accepting situations where the assurance required for a file is much greater than the resilience offered by a single node ($\mathcal{H}(f) \gg \mathcal{R}(u)$ for most files $f$ and nodes $u$) as described by Property 1. Doing so has the consequence that it is not possible to directly allocate a whole file over a single node. Every file $f$ will have to withstand a fragmentation process and each fragment $f_i$ will have to be rated with an assurance value $\mathcal{H}(f_i)$.

### 4.1 Allocation Security

Let us assume that file $f$ has been fragmented into $g$ fragments $f_1, \ldots, f_g$, and that all fragments are necessary to reconstruct the file (we will see that there are many efficient mechanisms to achieve this goal when we describe the fragmentation process). Each fragment $f_i$ is assigned a required assurance $\mathcal{H}(f_i)$. Then, we allocate the fragments into nodes with appropriate resilience and guarantee the assurance of the file by means of providing assurance to the fragments. To get this results, we impose a few constraints on the allocation process. With these constraints, it is easier to reason about the assurance provided by the allocation.

The first constraint is very intuitive. If fragments $f_i$ and $f_j$ are allowed to be hosted on the same node $u$, we would be in a situation in which the adversary can compromise node $u$ and instantaneously gain access to both $f_i$ and $f_j$ with effort $\mathcal{R}(u)$. This can reduce the assurance that the system is providing to file $f$. Therefore, we introduce the *single fragment constraint*.

**Constraint 1** (Single fragment)**.** *No two fragments of a same file can be hosted on the same node at any time.*

A second constraint deals with the number of nodes allowed into every subnet. We define the system parameter $\kappa$ as the maximum number of fragments *of the same file* that any subnet can hold. This is called the $\kappa$-constraint and is formally described as follows.

**Constraint 2** ($\kappa$-constraint)**.** *No more than $\kappa$ fragments of the same file can be kept by the same subnet at any time.*

This constraint is not crucial for the correct functioning of the system, but allows for a trade-off between security and performance. Intuitively, smaller values of $\kappa$ force the allocator to use a greater number of subnets for the fragments of the same file. Therefore, assurance is higher since the adversary has to break into more subnets to compromise the file, while performance of read operations is poorer, since the file is more spread and getting all the fragments takes longer. High values of $\kappa$, instead, produce the exact opposite results. To complicate the analysis, a smaller value of $\kappa$ allows the system to generate fewer fragments to get the same file assurance, therefore, the trade-off is not obvious.

The third constraint is the *eligibility constraint* and it has to be satisfied for a fragment to be allocated to a node.

**Constraint 3** (Eligibility)**.** *A fragment $f_i$ can be allocated to a node $u_j$ if and only if $\mathcal{R}(u_j) \geq \mathcal{H}(f_i)$.*

An *allocation* $\mathcal{A}_f$ for fragments $f_1, \ldots, f_g$ of file $f$ is a function $\mathcal{A}_f : \{1, \ldots, g\} \to \mathcal{P}(\mathcal{N})$, where $\mathcal{P}(\mathcal{N})$ is the set of all subsets of $\mathcal{N}$. Subset $\mathcal{A}_f(i)$ is the subset of nodes that store a replica of fragment $f_i$. An allocation is *valid* if it satisfies Constraints 1, 2, and 3.

If the adversary is willing to compromise file $f$ allocated with valid allocation $\mathcal{A}_f$, the effort to be paid is composed by two factors: One to break into enough nodes that store fragments of the file, and one to break into the subnets that host those nodes. Thanks to Constraint 1, the first factor is at least equal to the effort needed to compromise the weakest node in $\mathcal{A}_f(1)$, the weakest node in $\mathcal{A}_f(2)$, and so on, up to the weakest node in $\mathcal{A}_f(g)$. Moreover, thanks to Constraint 2, the adversary has to break into at least $\lceil g/\kappa \rceil$ subnets. To summarize, we can introduce function $\mathcal{J}(\mathcal{A}_f)$ as the assurance *achieved* by $f$ through $\mathcal{A}_f$.

$$\mathcal{J}(\mathcal{A}_f) \geq \left\lceil \frac{g}{\kappa} \right\rceil \cdot \mathcal{R}(s) + \sum_i \min_{u \in \mathcal{A}_f(i)} \{\mathcal{R}(u)\}. \qquad (1)$$

If the amount of assurance required for $f$ is grater than the resilience offered by any valid allocation, we say that the allocation is *unsatisfiable*.

## 4.2 The Fragmentation Process

One of the most important point to address is finding an appropriate number of fragments to use for each file in accordance to its assurance requirement. The fragmentation process for a file $f$ can be implemented as a $(g, g)$ Shamir's secret sharing scheme (Shamir, 1979), where again $g$ is the number of fragments that $f$ is divided into, or with faster erasure codes (whose security is not information-theoretic) such as Tornado codes (Byers et al., 1998), Turbo codes (Berrou et al., 1993), or Reed-Solomon codes (Reed and Solomon, 1960) among many others.

The challenge here is to perform the fragmentation process in such a way that the assurance of the file is guaranteed and that fragments can later be allocated to nodes in such a way that the load is balanced. Since the system is very large, we just assume to know what is the *distribution* of the resilience of the nodes. This information does not depend on the size of the network.

Let $f \in \mathcal{F}$ be a generic file and, as usual, let us assume that it has already been divided into $g$ fragments. Let $\mathcal{A}_f$ be a valid allocation and let us assume that every fragment is stored in one single replica. That is, for all $i$ $|\mathcal{A}_f(i)| = 1$. In this case, we will refer to such an allocation as *initial allocation* and function $\mathcal{J}(\mathcal{A}_f)$ is simply

$$\mathcal{J}(\mathcal{A}_f) = \left\lceil \frac{g}{\kappa} \right\rceil \mathcal{R}(s) + \sum_{i=1}^{g} \mathcal{R}(u_i) \qquad (2)$$

where $u_i$ is the only node in $\mathcal{A}_f(i)$. Let $\mathcal{Y}$ be the random variable that models the resilience provided by the nodes. We assume that this distribution has a finite expectation. It is then possible to say that, on average, each node will contribute with a resilience of $\mathrm{E}(\mathcal{Y})$. Hence, we can compute the average number of fragments needed to make the achieved assurance at least $\mathcal{H}(f)$ as follows:

$$\mathcal{J}(\mathcal{A}_f) = \left\lceil \frac{g}{\kappa} \right\rceil \mathcal{R}(s) + g\mathrm{E}(\mathcal{Y}) \geq \mathcal{H}(f) \qquad (3)$$

Let $\overset{*}{g}$ be the smallest value for which the 3 is true. Then, we can expect that the average number of fragments of file $f$ needed in our system is $\overset{*}{g}$.

The actual fragmentation process consists in iteratively sampling from random variable $\mathcal{Y}$ the assurance requirement for each fragment of file $f$. Every $\kappa$ fragments, we know that one more subnet will be used when allocating the file and that one more subnet has to be broken by the adversary to compromise the file. As a consequence, additional subnet resilience $\mathcal{R}(s)$ is taken into account. The process is stopped when the number of fragments is enough to guarantee the required assurance of file $f$. Of course, the actual number of fragments for the file can be different from the average computed in the previous section. Finally, we can state the following simple result.

**Proposition 1.** *The fragmentation process guarantees that: 1. The distribution of the assurance requirement of the fragments is equal to the distribution of the resilience of the nodes; and 2. if $\mathcal{A}_f$ is a valid allocation, then $\mathcal{I}(\mathcal{A}_f) \geq \mathcal{H}(f)$.*

Of course, this result holds only if the allocation is static. With caching, allocation changes dynamically and therefore we have to take particular care to guarantee that the adversary cannot compromise the file with smaller effort.

## 4.3 Distribution of Node Resilience

The random distribution we use for node resilience is the *Bounded Pareto distribution* described as $BPa(min, max, \alpha)$ where $[min, max]$ is the interval of values and $\alpha$ is the shape. We use two different random variables, both following the aforementioned distribution, albeit with different parameters. The variable $\mathcal{Y}$ has already been introduced, the second one is $\mathcal{X}$ and we will use it in our experiments to generate the assurance required by the files generated in the system.

Thanks to this it is possible to perform various experiments. For instance in Figure 2 we can see an example of the fragmentation of 100 files. On the $x$ axis there is the id of the files (we have sorted the files ids according to increasing assurance requirement), while on the $y$ axis there is the number of fragments the file has been divided into. In blue we have the expected number of fragments as computed above and in red there is the actual number of fragments obtained via the fragmentation process. For this experiment we used the distribution $BPa(100, 200, 2.0)$. The average difference between expected and real is 0.01 fragments.

## 4.4 File to Subnet Allocation

After a file has been created and fragmented, every fragment must be assigned a subnet to create an initial allocation. We call this the *file to subnet allocation*. The goal of this phase is to create a list of eligible subnets for each fragment and then to select the "best" element out of this list. It is in this phase that the $\kappa$-constraint is enforced.

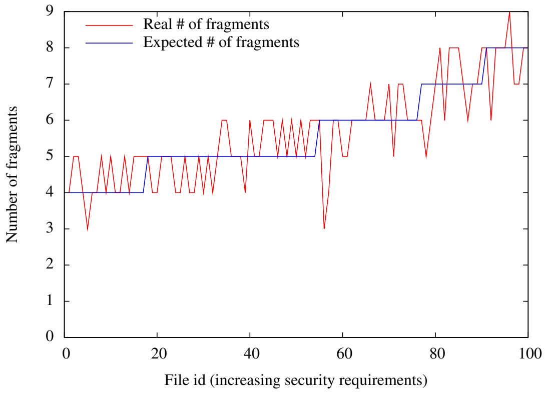

Figure 2: Expected number of fragments (blue) and Real number of fragments (red).

Our design choice is to enforce a static matching between the fragments and the eligible subnetworks. Formally, given a fragment $f_i$, we say that

$$s \text{ is eligible for } f_i \Longleftrightarrow \mathrm{H}_1(s|f) \equiv f_i \mod \left\lceil \frac{g}{\kappa} \right\rceil \quad (4)$$

where $\mathrm{H}_1 : \mathbb{N} \to \mathbb{N}$ is an arbitrarily chosen *hash function*[1] and $|$ is the juxtaposition operator. The hash function is not used for security purposes, but rather as a randomizer (any quasi-random function could be used). This means that properties like collision resistance, pre-image resistance and second pre-image resistance are not crucial.

With this mechanism, no compromised subnet can attract more fragments then it is supposed to store. This limits the possible damage that the adversary can perform by compromising a particular subnet.

By using this technique we are actually defining $\left\lceil \frac{g}{\kappa} \right\rceil = t$ colors $\{0, \ldots, t-1\}$ for all the subnets. We are then assigning one color $c$ to every fragment $f_i$ and saying that all and only the $c$-colored subnets are eligible for $f_i$. Furthermore we are also assuring that the $\kappa$-constraint is satisfied because for any given subnet, at most $\kappa$ fragments of the same file will be eligible. In Equation 4 the hash function is applied to $s$ juxtaposed to the file id $f$. We use this expedient to even out the unbalance due to the fact that $g$ is not a multiple of $\kappa$. Furthermore, this also has the effect that every file induces a different coloring on the system.

This procedure does not guarantee that every color is used, but we can show that with high probability

[1] Not to be confused with function $\mathcal{H}$, the assurance requirement.

this holds true if $|\mathcal{S}| \gg t$. If we assume that the function $\mathrm{H}_1$ is uniformly distributed over its codomain, we are actually assigning $|\mathcal{S}|$ fragments to $t$ colors and, if $|\mathcal{S}| = \Omega(t \log t)$ (which is the case in a large network since it is reasonable that $\lceil \frac{g}{\kappa} \rceil$ is much smaller than $|\mathcal{S}|$), we can also show that colors are well balanced. This is done by using well-known results on a famous problem referred to as *balls and bins* (Mitzenmacher, 2001).

The second concept that we need to introduce has to do with the resilience associated to a subnet. We define the function $\mathcal{Q} : \mathcal{S} \times \mathcal{F}' \to \{0,1\}$ as follows

$$\mathcal{Q}(s, f_i) := \begin{cases} 1 & \text{if } \exists\, u \in \mathcal{N}_s \mid \mathcal{R}(u) \geq \mathcal{H}(f_i), \\ 0 & \text{otherwise.} \end{cases} \quad (5)$$

Basically this function assumes value 1 when the subnet $s$ contains at least one node that can provide a resilience value equal to or greater than the assurance value required by the fragment $f_i$ and assumes value 0 otherwise.

In general, multiple subnets are eligible for a given fragment, introducing the need for a tie-breaker. The subnet chosen among all the eligible is the one *closest* to the subnet where the file has been created, where we measure distance in terms of hops. In the case that two or more subnets have this property one is chosen uniformly at random. We call this method *min_path*.

So if we call $G(\mathcal{S}, E)$ the graph associated to the underlying network infrastructure of subnets, we can say that a fragment $f_i$ is allocated to the subnet $s \in \mathcal{S}$ if: $s$ is eligible for $f_i$, $\mathcal{Q}(s, f_i) = 1$, $s$ has the minimal shortest path in $G$ from the subnet of creation of file $f$.

## 4.5 Fragment to Node Allocation

Once the subnet for fragment $f_i$ has been selected, we must choose the node where to actually store the fragment. Similarly to what we did before, we introduce a *node-fragment* eligibility notion: Given a fragment $f_i$ and a node $u \in \mathcal{N}$ we say that

$$u \text{ is eligible for } f_i \Longleftrightarrow \mathrm{H}_2(u|f) \equiv f_i \mod \kappa \quad (6)$$

By doing this we have a practical way of enforcing the single fragment constraint. Briefly, we are again coloring fragments and nodes and saying that it is legal to allocate a fragment within a node only if the colors match. Since we are using the mod operator we will have $\kappa$ contiguous different colors in $\{0, \ldots, \kappa - 1\}$. Thanks to the fact that we can not have more than $\kappa$ fragments of the same file in a subnet, there cannot be two fragments of the same color.

To perform the real allocation we take advantage of the fact that the distribution used for the nodes resilience and the fragments assurance is the same. For this reason we decided to use a *best-fit* type of allocation. The nodes of a subnet are partitioned into a set of equivalence classes $C$. All the nodes with the same resilience belong to the same class. After that, we choose the class $c \in C$ that has the smallest amount of resilience and that can provide the assurance required by $f_i$. After that a specific class has been picked up, one *eligible* node is taken uniformly at random in the class.

# 5 FILE HANDLING

In this section we describe the procedure needed to create a file, to retrieve it, and to improve the performance of read operations by caching.

## 5.1 File Creation

File creation is performed by a user logged in the system at a node. The user chooses the assurance requirement of the file, performs the fragmentation process, computes the allocation, and sends the fragments to the nodes by encrypting each fragment with the public key of the proper node (after checking that the resilience of the node on its certificate). Note that computing the file allocation is a local process—the client has to know only the nodes in the subnet and in the close neighborhood (just enough to have at least one subnet per color). The nodes store the fragment in the clear, or with a base encryption known to all the potential users of the system.

## 5.2 File Retrieval

File retrieval requests are issued from the users to access a file in the system. Whenever a read request is issued, the first thing needed to perform the retrieval is the list of nodes storing a replica of the fragment, one for each fragment. This part is performed by a dedicated underlying protocol (e.g. Chord, Pastry). Once the list has been obtained a *fragment request* is sent to the closest node storing the fragment, for every fragment. Upon receiving a fragment request, the node performs the authorization checks (that is, the node checks that the user has the right to read the file according to the policy associated to the file itself) and, in case, sends the fragment back encrypted with the client public key. Note that this mechanism supports dynamic changing of access policies, as, for example, the mechanisms described in (di Vimercati et al., 2007). When the user has obtained all the fragments, it can reconstruct the file. A file request is an exchange of *control messages* (like the request message) and *payload messages*, which are also used as unit of measurement to determine the request's cost. A payload message is a message exchanged between two subnets and containing a file fragment. Retransmissions of payload messages are of course payload messages. The cost of a file request is the total number of payload messages needed to complete it. Thanks to this we have that the cost of a single file request is the sum of the shortest paths from the requiring subnet to every fragment. With regards to the cost of a request, we also introduce the *delay* as the length of the longest path from a reader to one of the fragments.

## 5.3 Caching

In order to reduce the cost of a file request, we introduce *caching*. In our system, caching is designed in such a way to preserve the assurance of the file. Upon receiving a fragment as a response of a fragment request, if it is a first time read, the user also issues a *cache request*. Cache requests can be considered as brand new allocations originated at the subnet where the user is logged in. This means they undergo the same procedures described for a regular allocation. This technique produces the migration of a file toward its readers and will improve subsequent reads issued by the same reader or by one of its neighbor. Moreover, since caching replicates the fragments following the same rules of the initial allocation, we still guarantee that the adversary needs $\mathcal{H}(f)$ effort to compromise file $f$. Indeed, when a node is compromised, the adversary can get at most one fragment of a given file, even over time and during dynamic re-allocations.

# 6 LOAD BALANCE

Load balance is a fundamental property to make a caching system efficient. Given our system model, there are two independent load balance problems, one at the node level and one at the subnet level. The first problem, at the node level, is easy to understand—we do not want to overload strong nodes compared with weaker nodes. We have designed the fragmentation process exactly with this goal in mind. The second problem, at the subnet level, is less intuitive. Caching tends to allocate fragments close to the readers. However, even if the readers are uniformly distributed in the network, fragments tend to be allocated to a few "hub" subnetworks. This is intimately related to the scale-free property of the Internet. Let us discuss the first problem first.

We know that the nodes' resilience and fragments' assurance are distributed with a Pareto's law denoted by the variable $\mathcal{Y}$. Using a best fit policy, the allocator partitions the nodes of $s$ into equivalence classes based on their resilience. The best fitting class is then selected and one node is chosen from such class uniformly at random. Since both nodes' resilience and fragments' assurance are identically distributed, we can expect that fragments are well balanced over the nodes. In Figure 3 there is an example of an actual allocation. As we can see the load is evenly balanced among the nodes. Note that, according to our design, the best balance we can get at the node level is the same balance achieved by a uniform distribution. The load shown in Figure 3 has standard deviation 2.3, compared with the standard deviation of the same random process according to the uniform distribution which is 2.0. Therefore, the balance is very close to optimality.

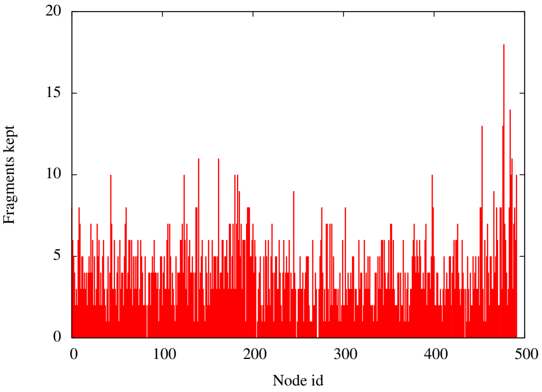

Figure 3: Fragments kept by a node. Example with 500 nodes per subnet and $10,000$ files. Load is balanced.

Let us now approach load balancing at the subnet level. As said before, the topology of the system is modeled to be Internet-like. This means that there are typically few *hub* subnets with a very high number of connections, and lots of subnets with very few links. The method used to establish the eligibility of a subnet for a given fragment is unaware of such a structure (just like many allocation mechanisms in the literature). This, combined to the fact that the best eligible subnet chosen by *min_path* is the one connected to the subnet where the file was created, through the shortest available path, makes the allocation highly biased toward the hubs as shown in Figure 4.

In order to avoid this phenomenon, we developed a method called *maxmin_path+1*. Let $f$ be a file to be allocated, $D$ the set of distances from the subnet of creation of $f$ and the subnet used to allocate the fragments according to the initial allocation, and $d_{max}$ the greatest element in $D$. This method consists of allocating every fragment by uniformly choosing an eligible subnet whose distance from the subnet of cre-

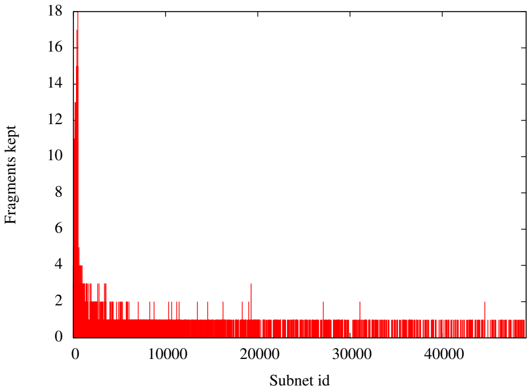

Figure 4: The load among different subnets is not balanced, hub subnets hold more fragments.

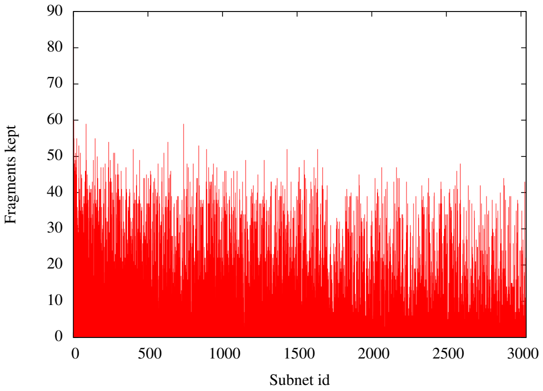

Figure 5: Method *maxmin_path+1*. Load is balanced, but delay is increased approximately by one.

ation is in the interval $[0 : d_{max} + 1]$. Since the diameter of the whole subnet graph is not high (being a scale-free network), a plus one increment allows many more subnets to be reached. Furthermore, by choosing a subnet at random from a larger set, the hub problem is greatly mitigated. As show in Figure 5 *maxmin_path+1* yields a very good load balance. On the other hand this method has the disadvantage of increasing the average delay. The number of eligible subnets after the plus one increment is much larger. Since the final subnet is chosen uniformly at random, with high probability at least one fragment will be allocated at distance $d_{max} + 1$. This will turn into a plus one increment of the average longest path which, by definition, is the delay.

## 6.1 Load-degree Correlation

In order to address the delay increase we designed a third method. As said before the load is unbalanced toward the subnets with the highest degree. We designed an experiment to determine whether or not

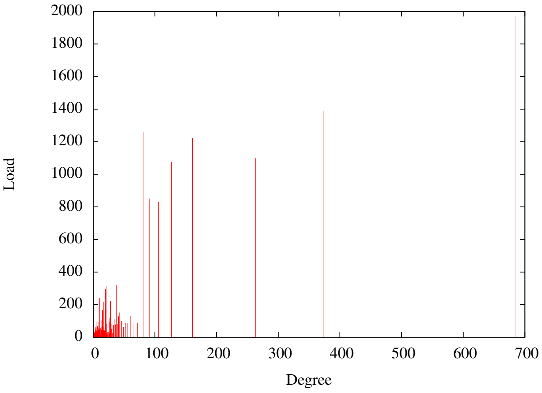

Figure 6: There is a direct correlation between load and degree.

there is a relation between the load of a subnet and its degree. Let $C$ be a fixed set of colors, $c$ a color in $C$, and $s$ a $c$-colored subnet. We define the degree of $s$ as the number of edges connected to it, while the load is the number of subnets the would choose $s$ as the designated subnet to allocate a $c$-colored fragment. As we can see from Figure 6 there is a direct correlation between load and degree.

After observing this property, we implemented a method called *maxmin_path*. Similarly to *maxmin_path+1* this new method uses the interval $[0 : d_{max}]$ for the list of eligible subnets (without increasing the delay, this time). The actual node used to store the fragment is then chosen from such list at random with a probability of $1/\deg(s)$, where $\deg(s)$ is the degree of subnet $s$. Taking advantage of the load-degree correlation, *maxmin_path*, can restrict the interval. This means that the average delay is left unchanged, only the average number of messages exchanged is slightly increased. We will present results on delay and traffic in the following sections. Figure 7 shows the load obtained through *maxmin_path*, as can be seen, it is still excellent.

# 7 SIMULATION RESULTS

In order to test our system, we used an ad-hoc simulator and run several experiments. The generator used to create the subnet graph is *Inet 3.0* (Jin et al., 2000) (Winick and Jamin, 2002). Inet can generate graphs that approximate the topology of the Internet at the autonomous system level and has already been used for similar works (Tu et al., 2010) (Kaune et al., 2009) (Xiao et al., 2010). The authors of Inet suggest to generate networks with a node number of no less than 3037 nodes in order to achieve a valid graph.

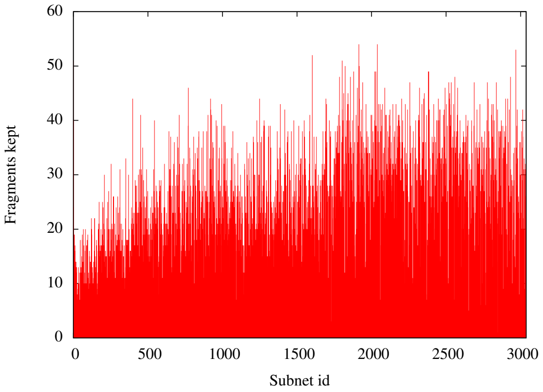

Figure 7: Method *maxmin_path*. Good load balance and delay is unchanged.

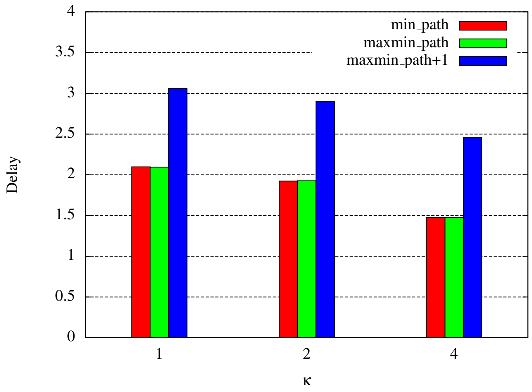

Figure 8: $\kappa$ vs Delay. Increasing values of $\kappa$ produce a decreasing delay.

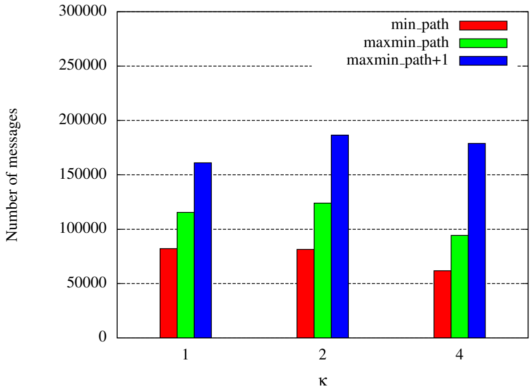

Figure 9: $\kappa$ vs number of messages.

## 7.1 Simulation Design

We designed a simple test environment. Multiple files are allocated by nodes chosen uniformly at random. Once the allocation is completed, the read phase starts. Reads are organized into two rounds, every file is assigned to a reader node and during a round each subnet sends out a read request for its designated file.

The first round is used to warm up the caching system, here all the caching requests are sent by every subnet, and the files spread over the entire system by getting closer to whoever has performed a read operation. During the second round the system is already running at full stretch and the readers can take full advantage of the vicinity of a file. Since cached fragments are kept indefinitely, adding more rounds would have been useless. The following experiments were run in a system with 3037 subnets, 500 nodes per subnet and $10,000$.

## 7.2 Simulation Results

The first experiment that we are going to describe is about delay. As we can see from Figure 8, by increasing the value of $\kappa$, the delay decreases. This happens because the subnets close to who issued the request can hold more fragments. As expected, the *maxmin_path+1* method approximately produces a plus one increase on the delay.

Then, we show how $\kappa$ influences the total number of messages exchanged in the system. Again, in this case we observe a general behavior depending on the method used. For *min_path*, by increasing $\kappa$ the number of messages goes down. As before this is caused by a more intensive use of the closest subnets. On the other hand we can see that *maxmin_path+1* presents almost the opposite pattern, it achieves worse performances due to the use of random choices. While whenever *min_path* can put more fragments into a close subnet it will always do so, *maxmin_path+1* will do that much more seldom because for fixed values of $\kappa$ it has many more available choices, choices that are made uniformly at random. Therefore, the influence of the fact that smaller $\kappa$ yields a smaller number of fragments is dominating the performance. Ultimately *maxmin_path* places in the middle, it will always have more eligible subnets than *min_path*, but not as many as *maxmin_path+1*, and its decisions are biased by the degree of the subnets.

Another interesting value is *assurance composition*. As the name suggests it is used to show how much of the assurance achieved by a file is issued by the actual nodes and how much is issued by the subnets. We already showed that the total amount of assurance achieved by an allocation is given by the 3. By using that we can calculate the *subnet assurance* and the *node assurance*:

$$\mathsf{subnet\_assurance} := \frac{\lceil \frac{g}{\kappa} \rceil \mathcal{R}(s)}{\lceil \frac{g}{\kappa} \rceil \mathcal{R}(s) + g\mathrm{E}(\mathcal{Y})} \tag{7}$$

$$\mathsf{node\_assurance} := \frac{g\mathrm{E}(\mathcal{Y})}{\lceil \frac{g}{\kappa} \rceil \mathcal{R}(s) + g\mathrm{E}(\mathcal{Y})} \tag{8}$$

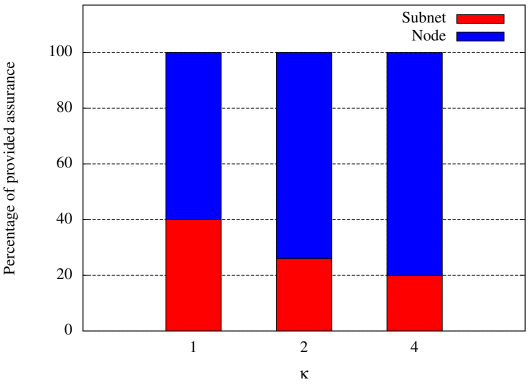

Figure 10: κ vs subnet assurance (red) and node assurance (blue). Subnet assurance decreases as κ increases and is compensated by node assurance.

As we can see from Figure 10, with $\kappa = 1$ more than 40% of the assurance is issued by the subnets. By increasing κ that percentage gets lower while the node security increases. This is a further demonstration of the importance of κ.

# 8 CONCLUSIONS

This paper introduced a model that provides guaranteed assurance, while achieving good load balancing both at node and subnet level and good traffic performance. The proposed model is dynamically adaptable through the parameter κ that can react to ongoing attacks and balance the system between higher security and better performance. We explained how our system can store files whose requirement exceed single nodes capabilities. Future research might focus on caching replacement policies, file-size driven allocators and reconfigurable networks.

# ACKNOWLEDGMENTS

Alessandro Mei is supported by a Marie Curie Outgoing International Fellowship funded by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n. 253461.

# REFERENCES

Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. (2002). Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.*, 36.

Berrou, C., Glavieux, A., and Thitimajshima, P. (1993). Near shannon limit error-correcting coding and decoding: Turbo-codes. In *ICC 93 IEEE international conference on Communications*, volume 2. IEEE.

Byers, J. W., Luby, M., Mitzenmacher, M., and Rege, A. (1998). A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM '98*.

Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. (2001). Wide-area cooperative storage with cfs. In *SOSP '01*.

di Vimercati, S. D. C., Foresti, S., Jajodia, S., Paraboschi, S., and Samarati, P. (2007). Over-encryption: management of access control evolution on outsourced data. In *VLDB '07*.

Jin, C., Chen, Q., and Jamin, S. (2000). Inet: Internet topology generator. Technical Report UM-CSE-TR-433-00, EECS, U. of Michigan.

Kaune, S., Pussep, K., Leng, C., Kovacevic, A., Tyson, G., and Steinmetz, R. (2009). Modelling the internet delay space based on geographical locations. *PDP*.

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. (2000). Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11).

Lakshmanan, S., Ahamad, M., and Venkateswaran, H. (2003). Responsive security for stored data. *IEEE Transactions on Parallel and Distributed Systems*, 14.

Mitzenmacher, M. (2001). The power of two choices in randomized load balancing. *IEEE Trans. PDS*, 12.

Reed, I. S. and Solomon, G. (1960). Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2).

Shamir, A. (1979). How to share a secret. *Communications of the ACM*, 22(11).

Tu, M., Li, P., Yen, I.-L., Thuraisingham, B. M., and Khan, L. (2010). Secure data objects replication in data grid. *IEEE Trans. Dependable Secur. Comput.*, 7(1).

Wilcox-O'Hearn, Z. and Warner, B. (2008). Tahoe: the least-authority filesystem. In *Proc. of the 4th ACM international workshop on Storage security and survivability*, StorageSS '08.

Winick, J. and Jamin, S. (2002). Inet-3.0: Internet topology generator. Technical Report UM-CSE-TR-456-02, EECS, U. of Michigan.

Wylie, J. J., Bigrigg, M. W., Strunk, J. D., Ganger, G. R., Kiliççöte, H., and Khosla, P. K. (2000). Survivable information storage systems. *Computer*, 33.

Xiao, L., Ye, Y., Yen, I.-L., and Bastani, F. (2010). Evaluation and comparisons of dependable distributed storage designs for clouds. *High-Assurance Systems Engineering, IEEE International Symposium on*, 0.

Ye, Y., Xiao, L., Yen, I.-L., and Bastani, F. (2010). Cloud storage design based on hybrid of replication and data partitionnig. *ICPADS*.