Anonymity and Trust in Large-Scale Distributed Storage Systems

Thèse présentée à la Faculté de Sciences

Institut d'Informatique

Université de Neuchâtel

Pour l'obtention du grade de docteur ès science par

JOSÉ VALERIO

Acceptée sur proposition du jury :

Prof. Pascal	FELBER	Université de Neuchâtel, directeur de thèse
Prof. Peter	KROPF	Université de Neuchâtel
Dr. Etienne	RIVIÈRE	Université de Neuchâtel
Prof. Vivien	QUÉMA	INP, Grenoble, France
Dr. Martin	RAJMAN	EPF Lausanne

Soutenu le 3 Juillet 2013

Université de Neuchâtel

2013

Abstract

Large-Scale Distributed Storage Systems (LS-DSSs) are at the core of several Cloud services. These externalized services may run atop multiple administrative domains. While a client may trust the organization that provides a given Web service, a single server may belong to another organization that the client does not trust. The design of a Distributed Storage System is itself a challenging task, in particular when scalability, availability and consistency are required.

This thesis explores three important aspects within this context: anonymity and trust in LS-DSSs, auditing for Large-Scale Distributed Aggregation Systems (LS-DASs), and the evaluation of Distributed File Storage Service (DFSS) prototypes. The three systems proposed are evaluated using prototype implementations.

We present SPADS, a system that provides *publisher anonymization* and *rate limitation* to any generic LS-DSS over a Distributed Hash Table (DHT), running on unreliable and untrustworthy peers.

Then, we propose CADA, a system that sits on top of a generic LS-DAS, and detects when one or several servers attempt to bias an aggregation. The system performs *probabilistic auditing*, and raises suspicions based on the statistical deviation from an expected behavior.

Finally, we study DFSSs. Performing a fair comparison of the File System Consistency Levels (FSCLs) as supported by existing DFSSs is difficult because these systems feature different base performance and optimization levels. We make an empirical comparison of these FSCLs by instantiating them into a novel DFSS testbed named FlexiFS.

Contents

List of Figures

List of Tables

Acronyms

AA	Authentication Authority.
AID	Anonymous Insertion Delegate.
AP	Aggregation Point.
API	Application Programming Interface.
CAS	Compare-and-Swap.
CM	Credential Manager.
CTO	Close-to-Open.
DFSS	Distributed File Storage Service.
DHT	Distributed Hash Table.
EP	Entry Point.
FP	Forwarding Point.
FSCL	File System Consistency Level.
FUSE	File System in User Space.
KBR	Key-Based Routing.
LS-DAS	Large-Scale Distributed Aggregation System.
LS-DSS	Large-Scale Distributed Storage System.
P2P	Peer-to-Peer.
PKI	Public Key Infrastructure.
PKS	Public Key Server.
RPC	Remote Procedure Call.

SLA	Service-Level Agreement.
SLOC	Source Lines of Code.

Chapter 1

Introduction

1.1 Context

Distributed Storage Systems can be present nowadays in any type of network service. With the ever-growing exchange of digital information (e-mail, photos, videos, etc.), the needs for storage capacity increase almost at an exponential rate.

According to a study done by Martin Hilbert and Priscila López [58], the world's storage capacity in 2007 was 295 optimally compressed exabytes (295 billion gigabytes). Moreover, globally stored information perceived a 23% annual increase, general-purpose computing capacity grew 58% and the capacity for bidirectional telecommunication grew 28%. Another survey by Gantz and Reinsel [49] estimates that by 2015, there will be 7.9 zettabytes (7.9 trillion gigabytes) stored, and, thanks to the increasing use of cloud services, nearly 20% of this data (1.4 zettabytes) will at least go through some cloud service, and around 10% (0.8 zettabytes) will be stored and maintained in the cloud.

Distributed Storage Systems have been a subject of research since the introduction of early systems like XDFS, Swallow, CMU-CFS and Cedar during the 70s [110]. These systems already explored issues such as naming, concurrency control and caching. LOCUS [102, 126] was the first system to propose location transparency to relieve the client of handling data location, and high availability through replication. These two concepts are still present in modern Distributed Storage System designs.

In general, a Distributed Storage System must address the following

challenges:

Permanent storage: availability of the data stored in such systems must not be impacted by temporary hardware failures, and persist until explicitly eliminated. A way to mitigate failures is through replication, which introduces challenges on its own.

Location transparency: the location of the data within the system should be completely hidden to the user; the user must not deal with data location.

Consistency: since data is usually replicated across the system, it is necessary to ensure that replicas are updated in a consistent way.

Availability: nowadays, many Web services rely on Distributed Storage Systems. These Web services require high availability of the data. Ideally, a Distributed Storage System should still provide the required data in spite of server crashes or hardware failures.

Performance: like other computing systems, Distributed Storage Systems aim for high performance. The performance of a Distributed Storage System can be measured using different metrics, such as latency, throughput, CPU and memory usage.

Security: a Distributed Storage System is typically accessed by many users, and it is often required that some data is accessible to some users, but not to others. Sometimes, a user requires that the system does not disclose her identity. Security is an important issue, and techniques as authentication, encryption, anonymization and auditing are commonly leveraged in such systems.

Scalability: scalability is a concept with no general accepted definition [59]. However, a common way to describe a scalable distributed system is a system that keeps acceptable service guarantees under large, increasing amount of load. This load can be a large amount of data, a large amount of concurrent users, or both. In order to achieve scalability, designers make a big emphasis on decentralization and parallelization.

This thesis focuses on the security, consistency and availability aspects of Distributed Storage Systems. We give a high-level definition of the three scenarios that are the focus of the thesis:

Large-Scale Distributed Storage System (LS-DSS) is a network of hundreds of servers or more, which serve thousands of clients that can store and retrieve data from the system. Data is stored in a distributed manner across the network.

Large-Scale Distributed Aggregation System (LS-DAS) is a specific case of an LS-DSS, where data is aggregated before storage.

Distributed File Storage Service (DFSS) is another specific case of an LS-DSS, where data on the cloud can be perceived as a collection or collections of files.

LS-DSS is the most generic scenario, and it is the focus of the first part of the thesis. Examples of large-scale storage can be found in the context of social networks (e.g., Facebook, Google+, Twitter), media sharing portals (e.g., Instagram, YouTube, Dailymotion, Vimeo, Mega), and other large-scale internet services, such as Wikipedia and Amazon.

A delicate topic that surrounds internet nowadays is the protection of privacy. Privacy is the interest of an individual in keeping control on who can have access to some information about her. This definition will be further discussed in Section 4.2.

Major internet companies like Facebook and Google have often faced concerns because of their use of private information as their core business [3, 27, 40, 60, 116, 117]. These concerns have led to the creation of alternative projects that take care more about privacy, e.g., the social networks Diaspora and Friendica [22, 47, 56, 68, 119].

One of the most basic ways to maintain privacy is through confidentiality, which can be guaranteed through the use of encryption and digital certificates, to avoid well-known attacks like man-in-the-middle and phishing. Sometimes, systems must go further and offer other guarantees such as internet anonymity. In an anonymous transaction, the data being exchanged cannot be associated with any particular individual. This concept will be explained in detail in Section 4.2. An approach to achieve anonymity is the use of pseudonyms (e.g., in email services, chat rooms, login to webpages, where they can be called also "alias", "login name", "nickname", etc.).

Achieving anonymity is a difficult task, because even when the data has been "cleaned" (so it cannot be associated with any person), the correlation of this data with other data set can provide traces of identity. This occurred to America Online (AOL) in 2006. AOL Research released in a website a

compressed text file containing several millions of search keywords from over 650,000 users. This dataset was offered for research purposes. Although they did not show the identity of the users in the report, identifiable information was present in many of the queries, and all queries were associated with AOL user accounts, identified numerically. The New York Times cross referenced the "anonymized" records and phonebook listings, and they managed to locate an individual. AOL acknowledged later that it was a mistake and removed the data [9, 55].

LS-DAS is the focus of the second part of the thesis. Several large-scale distributed applications rely on the aggregation of information as a central component. Examples of such applications include monitoring, feedback aggregation [42], search mechanisms [17,95], trust management [131], or popularity tracking and monitoring [18, 84]. We do not consider e-voting systems; in these systems, security requirements demand for dedicated infrastructures and strong security enforcement.

A simple example of the use of distributed aggregations is the tracking of the popularity of the content of a media streaming application (e.g., music streaming: Spotify [72], Pandora [99], Deezer [34], or video streaming: Netflix [97], XBOX Video [127]), which can be implemented in a decentralized Peer-to-Peer (P2P) manner. A similar mechanism could be used in the same streaming applications to support streaming limitation based on the time or number of items streamed. Other uses of distributed aggregations within this context are polls. Consider a *song of the year* poll proposed by such media streaming applications, or by a social networking site. In any of these cases, the client can automatically emit insertions to the distributed aggregation to increment the counter associated with a given entry, in order to push their favorite option, or to simply inform that they accessed one given item. Figure 1.1 presents the general mechanism associated with this aggregation.

DFSSs are covered in the third and last part of the thesis. The arrival of cloud storage services with file system front-ends for the end-user (e.g., Dropbox, iCloud, SkyDrive, Google Drive), and their deep integration with operating systems have made the consumer-oriented cloud storage a very important research field. Several aspects are essential when it comes to quality of service and user satisfaction for such services, notably latency and throughput of operations, and consistency of reads and writes.

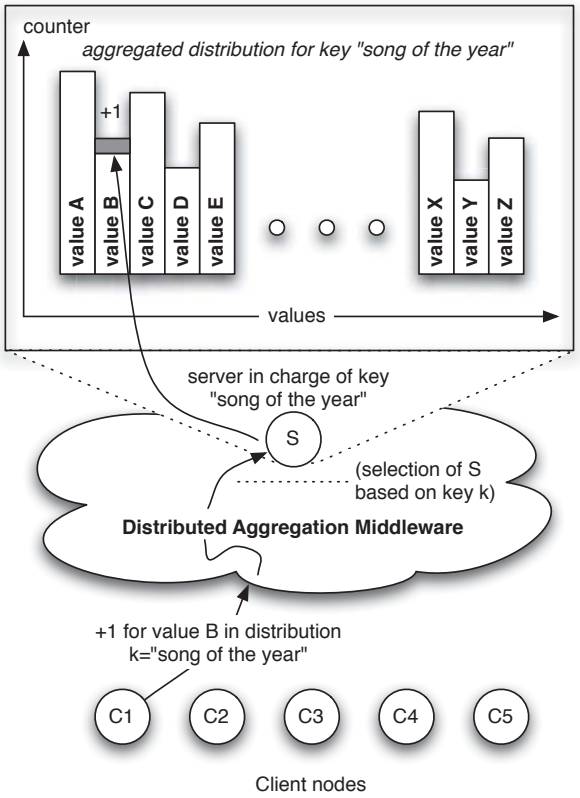


Figure 1.1: Principle of an aggregation of <value, counter> pairs at server S for key k. *(Client node C1 sends a request for increment of the value $k.B$, which is routed towards the node in charge of the distribution for key k).*

SPLAY We leveraged SPLAY [82] and its libraries for the implementation of the three systems we have developed (see Section 1.3). SPLAY is a platform that simplifies the prototyping and development of large-scale distributed applications and overlay networks. It covers the complete process of distributed system design, development and evaluation.

SPLAY allows developers to create distributed applications in a concise manner, using the Lua [65] scripting language. These applications are executed in a safe environment with restricted access to local resources (storage, memory, network) and can be instantiated on a large set of nodes.

1.2 Motivation

We introduced in Section 1.1 the current context of our research, and showed that there is a tendency to store more and more information on the cloud, while concerns about privacy regarding this information increment progressively. We have identified an ever-growing need for mechanisms that protect the identity of the end user, while preserving the functionality of the underlying system.

A specific motivating example of this thesis is the support of collective search and recommendation mechanisms. The systems created in Chapters 2 and 3 are motivated by the research done in the context of *Buzzaar*[1], a project done in collaboration with the EPFL (École Polytechnique Fédérale de Lausanne). The Buzzaar project proposes to exploit Web searches and access histories of users, collected from a simple browser extension, in order to support automatic browsing suggestions. Once collected, histories are sent to a collaboratively-operated network of aggregation servers. Thereafter, the information about previous accesses by other users is used to generate navigation suggestions. These previous accesses take the form of a distribution of elements for a given query and/or previously accessed element, and are refined based on the user interests derived from her own previous searches and accesses.

Buzzaar allows users to collectively create Internet maps, based on a variety of information derived from their navigation activities: co-occurrence of visits to Websites (*i.e.*, visitors of some Web resource A have a given probability of heading to resource B, for instance), frequencies of visits and co-visits, vector-based representations of the interests of the users based on their search and querying history, etc.

These maps are an expression of the emerging navigation patterns and implicit semantic links between Web resources for a given interest context. They are used by specialists of information retrieval and data representation to propose new Web navigation and representation tools.

Buzzaar is based on the collaboration of four kinds of entities. Figure 1.2 presents the relation between these entities and the flow of information between them.

First, clients install in their Web browser a lightweight plugin that deals with the gathering and pre-processing of browsing activity information. Sec-

[1] `http://www.buzzaar.net`

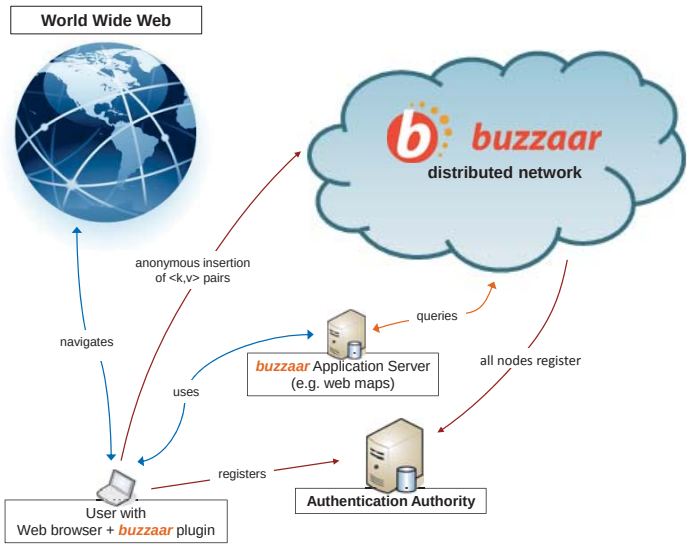


Figure 1.2: Buzzaar: general principle and flow of information.

ond, the Authentication Authority (AA) is responsible for registering users to the service, and for ensuring that these users are legitimate, that is, that they have a valid e-mail address and can answer a challenge-response test that is notably easy for a human and difficult for a machine (a "*captcha*"). This registration is a one-time operation, when the user installs the plugin. Third, a set of servers are collaborating to propose the service itself: the storage of user-generated Web activity information, and the computation of aggregates based on these values. The servers are running a fault-tolerant and self-organizing Distributed Hash Table (DHT) [87, 106, 115], with persistent-storage based on proactive replication. Finally, some external application servers are allowed by the AA to access the data stored in the DHT and the result of its aggregation, which are the inputs for the navigation and representation tools.

Such aggregation system can leverage cloud computing in any of its three service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) [89]. Thus, Buzzaar can be built upon the addition of rented resources from many different organizations and companies (See Figure 1.3).

The survey by Gantz and Reinsel [49] states that the presence of virtualization is increasing every year, and that in 2010 more virtual servers

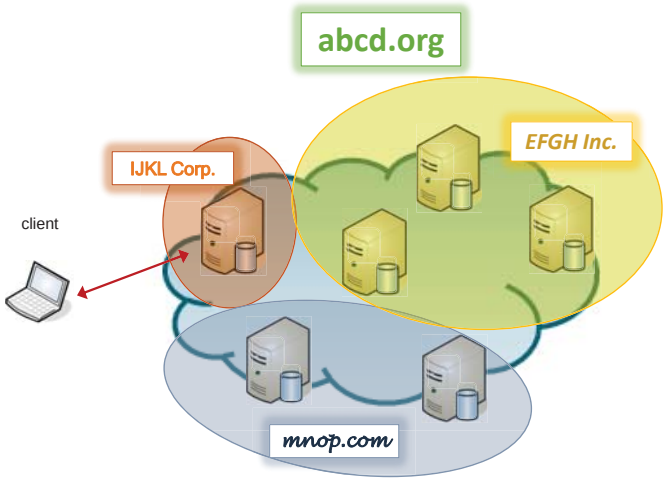


Figure 1.3: Multidomain data storage system.

were shipped than physical servers. This implies that while the client can fully trust the company or organization that provides the aggregation system (abcd.org in Figure 1.3), a single server may belong to another organization or company that is unknown by the client, or even worse, that the client does not trust.

In this context, only a small level of trust exists between clients and servers:

- Users trust the system authority for certifying their identity but not for processing data. Users cannot trust a single server for not disclosing private information, and should be able to rely on system-wide mechanisms to hide their identity to the servers that will effectively process their data.

- Servers cannot trust anonymous clients for using the system fairly; they must rely on certain mechanisms for limiting risks of information flood and guaranteeing that the information is posted by authenticated sources.

- A third risk is the possible presence of malevolent servers that may want to bias the aggregations.

To mitigate these three risks, we introduce the systems depicted in Chapters 2 and 3. While Buzzaar fits in the description of an LS-DAS (aggregation system), the solutions provided in Chapter 2 apply to any generic LS-DSS (storage system).

The third part of the thesis focuses on the comparison of consistency models on DFSS. Like any distributed storage service, the expected properties of a DFSS are consistency, availability, and tolerance to partitions. The CAP theorem [52] states that a distributed storage system can fully support at most two of these three properties simultaneously. This will be explained in detail in Section 4.2. Partition tolerance is usually considered essential for a DFSS, as data centers may be temporarily disconnected in a large-scale distributed setting and such events must be supported. As a result, developers of a DFSS usually decide on a trade-off between availability and consistency.

Historically, DFSS designs have focused on providing the POSIX strong model of consistency [64]. The need for planetary-scale and always available services, and thus the shift to geographically distributed platforms and cloud architectures, has led DFSS designs to focus more heavily on availability and weaker consistency levels. By introducing caching mechanisms and the *close-to-open* semantics, the Andrew File System (AFS) [61] was one of the first systems to offer a consistency level weaker than POSIX. Most operations in systems such as HDFS [45] or GoogleFS [51] are sequentially consistent. However, both systems are built around a central metadata server. Schvachko [112] recently pointed out that this approach is inherently not scalable because the metadata server cannot handle massive parallel writes and the physical limits of a central metadata server design hits the petabyte barrier, i.e., the system cannot address more than 10^{15} bytes. On the other hand, flat storage systems like Cassandra or Dynamo [33, 74] propose an even weaker consistency level: eventual consistency. This relieves designers from the need for a central metadata server.

Some systems [16, 31, 93] implement a file system interface on top of an eventually consistent storage system. However, to the best of our knowledge, none has gained wide acceptance. Enabling further research on DFSS to scale and break the petabyte barrier requires developers to understand and be able to compare systematically the multiple components of a design. These components include data distribution and replication (and associated consistency guarantees), request routing, data indexing and querying, or access control.

Performing a fair comparison of these aspects as supported by existing

DFSS implementations is difficult because of their inherent differences. Indeed, these systems propose not only different File System Consistency Levels (FSCLs), but they also feature different base performance and optimization levels, which largely depend on the programming language, environment, runtime, etc. This interesting topic is the motivation behind the research done in Chapter 4.

1.3 Contributions

The three contributions of this thesis are:

SPADS: Publisher Anonymization for DHT Storage

First, we present the rationales, design and implementation of SPADS [43], a system that provides *publisher anonymization* and *rate limitation* to any generic LS-DSS over a DHT, running on unreliable and untrustworthy peers.

Information stored in an LS-DSS can be sensitive and it might be necessary to protect the client's privacy. Furthermore, the risk of having the system polluted by malevolent clients flooding fake insertions is high. SPADS deals with both aspects. Publisher anonymity is guaranteed by the use of anonymizing paths between the clients and the aggregation layer. The influence of malevolent users is reduced by means of rate limitation, enforced during the anonymizing routing operation.

SPADS does not require any modification to the DHT but simply builds upon it. We evaluate the system using a real implementation to observe its scalability, cost, and effectiveness.

CADA: Collaborative Auditing for Distributed Aggregation

We focus then on the *detection* of malevolent behaviors from servers participating in an LS-DAS. We propose and evaluate CADA [120], which is composed by *oracles* that detect when one or several servers are attempting to bias the aggregation.

These two oracles are lightweight and operate in a decentralized and autonomous manner. They are based on the nature of the operation performed,

namely the aggregation of counters for a set of values, and consider the statistical deviation from an expected behavior.

The actual action(s) to be taken by the system once a server has been suspected are out of the scope of this thesis, and will often be application-specific. The CADA oracles are intended to provide the input for an external mechanism such as a blacklist management service, a trust management layer, or the triggering of a more generic—but also considerably more costly—strict protocol auditing mechanism such as PeerReview [53].

FlexiFS: Framework for the Evaluation of Distributed File Storage Service prototypes

Finally, we depict the construction of a file system on top of a regular key/value store with the simple addition of the *compare-and-swap* primitive, and present a clear typology of the different FSCLs and categorize existing implementations accordingly. Then, we make an empirical comparison of these FSCLs by instantiating them into a novel DFSS testbed named FlexiFS [121]. Our testbed is modular and implements a range of state-of-the-art techniques for the distribution, replication, routing, and indexing of data.

FlexiFS offers the choice of six flavors of FSCLs, plus, the possibility to tune several aspects about replication (total number of replicas, minimum number of replicas to write or read, etc.). We performed several benchmarks with FlexiFS and we discuss in this thesis the results obtained during these experiments.

1.4 Organization of the thesis

The rest of the thesis is organized as follows: In Chapter 2 we present SPADS, the first contribution of our thesis. CADA is presented in Chapter 3. In Chapter 4 we introduce FlexiFS and we discuss other topics, such as the integration of the aforementioned systems, improvements done to SPLAY during the realization of the thesis, and future work. Conclusions are presented in Chapter 5.

Chapter 2

Anonymity and Trust in Large-Scale Distributed Storage Systems

2.1 Introduction

In this chapter, we present the design and implementation of SPADS: Publisher Anonymization for DHT Storage. SPADS is a system designed in the context of an Large-Scale Distributed Storage System (LS-DSS) that is composed of servers, which are organized as a Distributed Hash Table (DHT) and store data, and clients, which send data to the servers. By using SPADS, an LS-DSS can provide *publisher anonymization* and *rate limitation* to the clients, while authenticating users in a reliable manner.

The chapter is organized as follows: In Section 2.2, we explain concepts that are needed for the understanding of our design. In Section 2.3, we present in more details the considered system model, the underlying Peer-to-Peer (P2P) layer characteristics, the problem definition, and SPADS' provided guarantees. Section 2.4 gives an high-level overview of the mechanisms underlying SPADS. These mechanisms are further described in Section 2.5. We present an evaluation using a prototype of SPADS deployed on a cluster in Section 2.6. Section 2.7 goes through similar work already done in this domain. Finally, we summarize in Section 2.8.

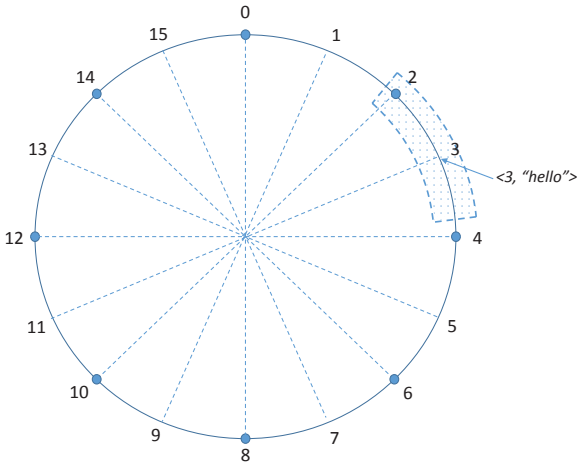


Figure 2.1: DHT ring example.

2.2 Background

Distributed Hash Tables

DHTs are a basic component of the designs depicted in this thesis. A hash table is a structure used to organize data inside an associative array. Associative arrays store items with the format $< key,\ value >$. In a hash table, the key is generated by means of a hash function. A hash function is a mathematical function that maps a variable length data set into a fixed length data set. The use of hashes gives a very high probability that the *key* is unique to each item (hash collisions may happen, but they are very seldom).

A DHT is functionally the same as a hash table, with the addition that, in a DHT, data is shared between several nodes interconnected through a network.

Several P2P systems gained popularity during the 90s, with the notable mention of Napster, Gnutella and Freenet. These systems used centralized or flooding mechanisms to index data in their networks, which as mentioned in Section 1.1, affects scalability. A solution to this was the use of DHT within the context of P2P systems.

The use of DHTs in P2P systems facilitates automatic indexing and load balancing of the data by leveraging consistent hashing [67]. A common abstraction for DHT design is to picture the overlay as a "ring" (see Figure 2.1).

Ideally, nodes are equidistant from each other on the ring, and keys are asigned to nodes according to the "region" where they fall. This design presents the advantage that, when a node enters or leaves the network, only a small range of keys are redistributed. In Figure 2.1 one can see that the key/value pair < 3, "*hello*" $>$ falls into the region that is managed by the node with ID equals 2.

A common Application Programming Interface (API) for a DHT is detailed in the work of Dabek et al. [32]:

- `put`(k, v) inserts an element of value v and key k.
- `get`(k) returns the value v of the element associated with key k.
- `remove`(k) deletes the element associated with key k.

In 2001, four different approaches to DHTs appeared in academic research: Pastry [106], Chord [115], CAN [104] and Tapestry [129, 130]. All of these approaches present multi-hop routing; they assume large-scale networks, where maintaining a full list of all the nodes is not efficient. In contrast, each node only knows a sample of the network, and when a node needs to route a message toward a key, it forwards it to the node that is closest to it, among the nodes that are in its sample. This process goes on, until the message reaches the node in charge of the region that contains the key.

We explain briefly these DHT designs.

Pastry Introduced by Antony Rowstron and Peter Druschel, Pastry [106] is a DHT where nodes are organized as a ring, composed by 128-bit keys. Node IDs are chosen randomly and uniformly, so nodes who are adjacent in the ring can be physically distant.

Pastry uses concepts of the Plaxton routing schema [100]. The main idea is to route a key to the known node with the longest common prefix (i.e., all the bits that are equal, starting by the most-significant bit). A Pastry node holds two tables of possible routes:

- Routing table: Long distance links to other prefix realms.
- Leaf table: Numerically close nodes.

A third table of close nodes based on some metric (e.g., latency) called "Neighbor set" is also maintained, but it is not relevant to the routing.

When a key must be routed, the Pastry node checks first on the leaf table if the most suitable node is there. If the best match is not found there, it checks on the routing table the node with the longest common prefix.

Several systems have been based on Pastry, amongst them: a P2P storage system (PAST [38]), a P2P file system (Pastis [16]), and a publish/subscribe system (SCRIBE [107]).

Chord Ion Stoica et al., from the Massachusetts Institute of Technology presented Chord [115] in 2001. Keys and node IDs can have values between 0 and $2^m - 1$, where m is the number of bits of the key space. To "close" the ring, $k = 2^m - 1$ is followed by $k = 0$. Consistent hashing is done with the use of the SHA-1 algorithm.

Each node has a successor and a predecessor. The successor to a node (or key) is the next node in the DHT ring in clockwise direction. The predecessor is the next node in counter-clockwise direction. Each node also knows a list of m special nodes called "fingers". The i_{th} finger has ID equal to $n + 2^{i-1}$, where n is the address of the node that keeps the table. This is illustrated in Figure 2.2.

Chord is the base of the CFS file system [31].

CAN Content-Addressable Network (CAN) [104] was proposed by Silvia Ratnasamy et al., from the University of California at Berkeley and AT&T Center for Internet Research.

Instead of creating a ring (one-dimensional space), routing in CAN is based on a multi-dimensional space (torus) of virtual logical addresses of the nodes of the network. According to its virtual address and the one of its neighbors, each node gets assigned a region in the multi-dimensional space. A two-dimensional example of this arrangement can be seen in Figure 2.3. In this figure, we can see that the key is mapped to the coordinates (0.6, 0.3), which fall into the region managed by node 5.

When there is a key to be routed, it traverses the zones until it finds the one that corresponds to the key's mapping into the multi-dimensional space. Then, the node that maintains the zone handles the key.

CAN is the base of an application level multicast mechanism [105] among other systems.

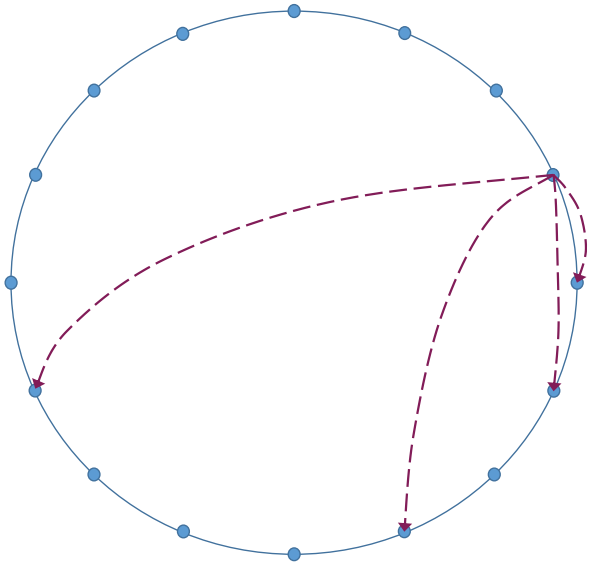


Figure 2.2: Chord fingers.

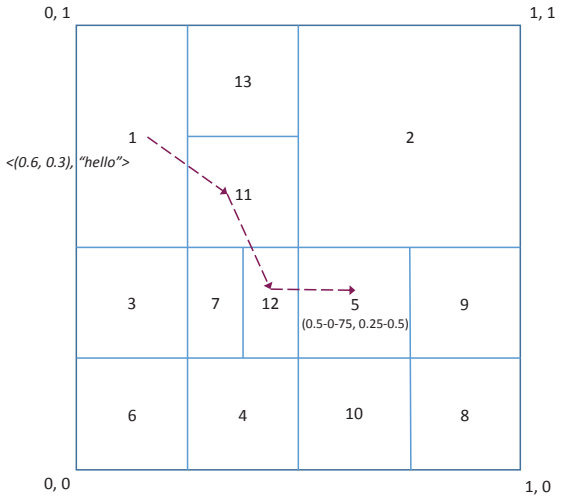


Figure 2.3: CAN two-dimensional example.

Tapestry Ben Y. Zhao et al., from the Computer Science Division of the University of California at Berkeley created Tapestry [129, 130]. The system uses a keyspace of 160 bits, where keys are created with the use of the SHA-1 hash function.

Like Pastry, Tapestry leverages the Plaxton algorithm to route messages. Objects (key/value pairs) in Tapestry have always a *root node OR*, which is the responsible node for the object O. The OR is the node with the numerically closest ID to O. There are four operations provided by the system:

- PublishObject(O_g, A_{id})
- UnpublishObject(O_g, A_{id})
- RouteToObject(O_g, A_{id})
- RouteToNode(N, A_{id}, *Exact*)

For all the operations, O_g is the GUID (Globally Unique ID) of object O, and A_{id} is the Application ID. The ability to keep different A_{id} per node is similar to having different TCP ports per machine. A particular feature about this API set is that they are not meant to transfer the object to its root OR; instead, they only serve to create a pointer in OR that indicates the node that actually holds the object.

When a node S wants to publish that it holds an object O, it sends a PublishObject message to OR, to announce it. For each of the nodes that forward this message up to OR, a pointer $O \rightarrow S$ is stored, indicating that node S holds the object.

When any node wants to retrieve O, it will send a RouteToObject message to OR (it is sure that at least OR has a mapping for object O), and as soon as this message goes through a node that has a pointer to node S, the message is redirected to S.

UnpublishObject announces that node S no longer holds object O, and thus, deletes the pointers along the path to OR. RouteToNode is used to send a message specifically to node N. The flag *Exact* is to indicate that the message must be delivered to the node with the exact ID specified on the message, and not to the closest one.

The distributed storage system OceanStore [73] uses Tapestry.

Onion routing

In Section 1.1 we discussed briefly the concepts of privacy and anonymity. Privacy is a general concept, not only related to computer science and internet. Clarke [26] defines privacy from the philosophical, psychological, sociological, economical and political point of view. He also introduces the term "information privacy", which is the composition of:

Privacy of personal communications is the interest of an individual in being able to communicate with other individuals, using various media, without monitoring of their communications by other persons or organizations.

Privacy of personal data is the interest of an individual that data about her is not automatically available to other individuals and organizations, and that, even when data is possessed by another party, she is able to exercise control over that data and its use.

There are several levels of information privacy, that are covered by different guarantees (e.g., confidentiality, anonymity). Some systems might want to enforce anonymity to provide privacy. According to Clarke [25], an identified record or transaction is "one in which the data can be readily related to a particular individual. This may be because it carries a direct identifier of the person concerned, or because it contains data which, in combination with other available data, links the data to a particular person.". On the other hand, an anonymous record or transaction is "one whose data cannot be associated with a particular individual, either from the data itself, or by combining the transaction with other data".

One way to achieve anonymity is through pseudonyms. As explained in Section 1.1 with the example of the AOL data leak, sometimes, the use of pseudonyms is not enough, because several records linked to the same pseudonym can be correlated and disclose private information.

A large body of distributed applications process or depend on data that contain user-centric information. This kind of data is usually considered, with reason, of high sensitivity by users, as it often allows linking to their identity or contact information and could be used for malevolent purposes, such as unsolicited targeted advertising. The majority of these applications are based on the client-server paradigm. The agreement on the conditions of processing

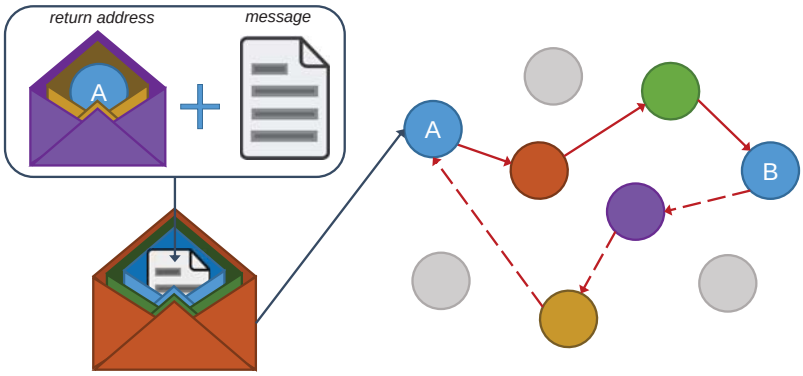


Figure 2.4: Example of onion routing with 3 layers and return address.

and storage of the sensitive data is a matter of trust given by the user to the application server.

Systems such as PeerReview [53] can detect modifications to the distributed protocol used by a node, based on its interactions (sent/received messages) with other nodes of the system. Nonetheless, it is impossible to detect that a node is using the data, and the identity of its publisher, in a non-allowed manner either internally or as part of another protocol.

As a result, when designing a distributed system that has to process user-sensitive information, it is unrealistic to ask each user to consider that no single server will use her sensitive information for other purposes. Instead, it is necessary to provide the user with a mechanism that allows her to send the sensitive data that is useful for the system and for other nodes, but at the same time hides her identity from the servers and other clients participating in the system. This process is called *publisher anonymization*. It ensures that the nodes that will be able to use and process data (and thus, see it in its *clear form*) have no mean to derive the identity of the original clients that produced it (e.g., their IP addresses, usernames).

The use of public key cryptography along a chain of forwarding nodes, in order to achieve anonymity, was introduced in the early 80s by Chaum [23] in the context of email anonymization. This technique, called *mixes*, was later popularized as part of the *onion routing* concept.

The scenario is shown in Figure 2.4. Node A wants to send a message to node B and receive an answer from it, but neither node A nor B wants other nodes to know with whom their are exchanging messages. The main idea is

to force the messages to travel through a path of nodes before arriving to the destination, making it untraceable. To achieve that, the message is encrypted iteratively with the public keys of the servers along the path. For that matter, we assume that the network leverages a Public Key Infrastructure (PKI) [2], thus, every server has a well-known unique public/private key pair. This is represented in the figure by the lower envelopes. Then, the message must go through the whole path (full-line arrows), in order to reach node B. Each server along the path decrypts the message with its own key and passes the result to the next step (like removing layers from an onion, hence the name). It is only the last server of the path, node B, who can read the original message.

If A wants to receive an answer from B, it can set another path that goes from B to A, encrypt the return address with the keys of the servers along this path (dashed-line arrows), and attach this to the message before its encryption. This is represented in Figure 2.4 by the upper envelopes and the addition symbol.

The system provides anonymity by decoupling the plaintext message to any identifiable information of the source, until it reaches the end of the path. If the paths are composed by $f + 1$ servers, one can say that the system guarantees anonymity of the source even in the case up to f servers agree to pass identifiable information (there will always be at least one node along the path that will drop the identity of the source).

We use in SPADS a variant of this algorithm, which is explained in Section 2.5.

2.3 Problem definition: Anonymity vs. Trust

Decentralized data processing and storage systems typically face the problem of having clients that want to compromise the good operation of the system. Specifically, some clients may want to flood the system with a large amount of fake information in order to corrupt the service given to other clients. For instance, in a collaborative search mechanism, if the collected information is used to derive the popularity or appropriateness of some Web pages, some cheating user may try to forge a non-deserved rank for her Web site by sending a large set of fake entries linking to it. It is thus necessary to provide *rate limitation* of the data inserted by users into the system.

When the system ensures publisher anonymization, it is difficult, if not

impossible, to deal with the problem of cheating users after the data has been inserted into the system. Indeed, it is by construction not possible to blacklist some user, or remove the entries sent by a given user who has been identified as a cheater, because the information about the origin of the data is no longer available. As a result, it is absolutely necessary to propose a rate limitation mechanism that takes place *before* the information is actually inserted into the system. Obviously, this rate limitation mechanism shall not interfere with publisher anonymization.

Finally, in a distributed setting where users can join or leave the system at any time, rate limitation is ineffective if it only limits the publication rate of an individual user, but accepts new users without a prior check that they are legitimate (that is, that the user is a human and not an automatic robot performing multiple registrations using different, automatically generated, identities). It is thus necessary to ensure that users are properly *legitimated* and *authenticated* by the system prior to allowing them to publish.

We start by a description of the target system and the prerequisites on which SPADS builds. Then, we further elaborate on the considered threat model and on SPADS' guarantees.

System model

We consider a system composed of:

- **Clients**, that produce some data.
- **Servers**, that store and process the data from the clients.
- An **Authentication Authority (AA)**, by which clients register and are certified as legitimate human users–this authority also controls some system-wide parameters and authenticates the servers, but does not process any user data.
- **Applications** that can access the information stored in the servers.

DHT

The servers are organized in a DHT. SPADS does not require any modification to the DHT itself. The current prototype uses Pastry [106] but SPADS supports other DHTs like Chord [115] or Kademlia [87]. Clients, applications, and the authentication authority do not participate in the DHT but know *at least* one node that can serve as an entry point to the service, or have means to obtain a reference to one or several such nodes.

The DHT shall propose a conventional API, with `put`(k, v) and `get`(k) operations (see Section 2.2). The DHT is itself based on a Key-Based Routing (KBR) layer and a structured overlay.

Additional DHT mechanisms

First, we assume that the DHT is augmented by a best-effort proactive replication mechanism (as in CFS [31]) that tries to maintain at all time a given number of backup copies of all objects stored in the DHT. This allows to maintain the storage service in case of servers failures.

Second, we assume that servers in the DHT use node ID authentication (a mechanism also introduced by CFS [31]). A server that wishes to connect to the DHT must use as its identifier the hash of its IP address, and will not be integrated in the routing structure if this relation between the ID and the IP address is not validated. A server, when receiving a connection request from a new server, send a nonce to the claimed IP address and check that (1) it gets back an answer with the proper ID and (2) the ID corresponds to the hash of the IP address. As forging more than a limited number of IP addresses is a hard task, this mechanism allows to remove the threat of servers voluntarily inserting themselves in one or several strategic positions in the DHT as part of an attack.

Third, we assume that only servers have access to the raw `put`() and `get`() interface of the DHT, and not clients. Without loss of generality, we consider that the links in the DHT are directed.

Cryptographic infrastructure

Each server is associated with a pair of RSA private and public keys. All public keys are known, and signed, by the AA. The AA plays the roles of a certification authority for the servers and of a Public Key Server (PKS) for the clients.

A server can communicate its identity only by using a certificate. The server first needs to get this certificate signed by the AA when joining the system, or when its IP address (therefore, its ID) changes. The AA keeps track of all issued certificates and ensures that an IP address cannot be associated with more than one certificate. The content of a certificate is as follows:

Certificate for a server s	
s.addr	IP address (determines also its ID in the DHT)
s.pubk	public key of the server
$[[s.\text{pubk}, \text{addr}]]_{\text{AA.privk}}$	signature of the certificate

Clients do not have a public and private key pair. All clients and all servers know the public key of the AA. It is used for certification and authentication purposes at several stages of the protocol.

Each server knows a set of *neighbor* servers, and for each of those, it knows the associated public key. When a new link is created between two nodes, a handshake protocol takes place resulting in both parties knowing a small symmetric key on both sides of the link. This key is then used to encrypt the data using AES before sending it (a checksum is also included to ensure that the message is valid). Clients do not have an encryption key for the link they use with the entry point in the DHT, and are restricted to using the SPADS API extension only.

Threats considered

SPADS considers the following two misbehaviors from clients and servers. First, some clients are *cheaters*. These clients (properly authenticated in the system or not) want to send floods of fake information to influence the service given to other clients. This can happen, for instance, if the user installs a corrupted plugin, or is infected by an external virus. Second, some of the servers may be operated by organizations that have an interest in gathering personal data (e.g., Web site visits, interest representations, search histories) and associating it with personal identification (IP address, username, etc.). These misbehaving organizations want to use the collected data to generate unsolicited targeted advertisement, spy on the the Web sites visited by some particular user, etc. We consider up to f colluding peers, f being a parameter that can be decided by clients at each insertion. The authentication authority is considered correct.

Note that we do not consider servers to be potential cheaters, or if they are, we do not consider that dealing with them is the role of SPADS. This is covered later in Chapter 3. Indeed, servers are already part of the DHT and as such have access to the unprotected `put()` operation, which they can use to directly insert fake data. Such a server protocol misbehavior can be detected by other servers using external means such as PeerReview [53], where the

server's external actions (message logs) are reviewed by some other witness servers against the protocol description. As the generation *ex nihilo* of new `put`() requests is not linkable to some incoming request by a client (which would have been received by routing from another server), it will fail the review. Similarly, the node ID authentication mechanism ensures that a node cannot insert itself at a position in the DHT keyspace where it could take the responsibility of the element it wishes to promote (and reply with fake values to `get`() operations).

Guarantees

SPADS provides the following guarantees to clients and servers of the system, and, by extension, to the application servers that will access the anonymized data: First, it provides publisher anonymity. A client c sends some pair (k, v) in an encrypted form. SPADS ensures that (1) only one node is able to decrypt (k, v) and (2) this node is not able to learn or deduce any information about c's identity or identification in the system. Second, it provides decoupled authentication. While servers neither store nor use any authentication information from users, the protocol allows by construction only legitimate and registered users to put information in the DHT. Third, it provides rate limitation. Registered and legitimate users are not allowed to put more than n (k, v) pairs in the DHT per period of time Δ. n and Δ are system-wide parameters known by all servers. Furthermore, since rate limitation is a process that takes place in the long term, it is protected against corruption due to servers churn.

2.4 SPADS: Publisher Anonymization for DHT Storage

This section gives a high-level overview of how SPADS works. Unlike the AA server, which is a special, trusted entity, SPADS' servers are generic and reactive nodes, which may perform any of the roles presented in Table 2.1 on demand, depending on the message they receive.

We explain below the the main steps of SPADS' mechanisms. The explanations are based on Figure 2.5.

Any user must first register with the AA, which checks its legitimacy using external means, e.g., a "*captcha*" (noted ❶). After to this one-time

Role	Description
EP	The *Entry Point (EP)* is in direct communication with the client, and it is the first server the message goes through.
AID_n $1 \leq n \leq f$	The *Anonymous Insertion Delegates (AIDs)* are the nodes that remove each of the onion routing layers until the message has been completely decrypted.
CM_n $1 \leq n \leq r$	The *Credential Managers (CMs)* perform token-based rate limitation of the client's messages.

Table 2.1: Server roles in SPADS.

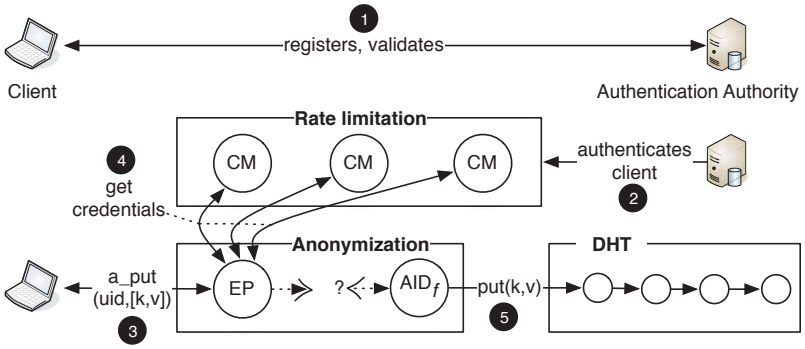


Figure 2.5: SPADS: general principle.

operation, the AA registers (❷) the user to a subset of r servers that will act as CMs (r=3 in Figure 2.5). The set of r CMs is different for each client. They are *collectively* responsible for the limitation of the rate at which this client can send messages. SPADS introduces a single DHT API extension: `a_put`$(uid, [(k, v)])$ (❸). The pair (k, v) is forwarded by a series of $f + 1$ intermediate servers, starting with the EP and followed by f AIDs. The EP and AIDs are selected among the list of servers known by the client. This list is periodically refreshed by contacting the AA.

The first phase is to enforce the rate limitation for the client. The EP knows the authentication information of the client, in particular, its IP address and *uid*. It uses the latter to perform the rate limitation through the use of credential (tokens). The EP obtains the number of credentials the client is currently allowed to use, by asking the r CM servers (❹). If there are no credentials available, the client is notified by the EP and the process stops.

Thereafter, the process of anonymization takes place by forwarding between the AIDs. The last AID is then responsible for sending the regular `put`(k, v) to the DHT, on behalf of the client (❺). The process ensures that any element that could relate to the identity of the client is lost on the way, and is never accessible to the last AID, even if at most f servers collude to spy on the user. The last AID is the only peer able to fully decode the (k, v) pair initially sent by the client.

2.5 Algorithms

This section presents the algorithmic details and properties of each of the phases mentioned in the previous section: client legitimation and authentication, management of insertion credentials for clients, and publisher anonymization.

Client legitimation

The first action is to legitimate the client, that is, to ensure that the client is operated by some human and not by a robot performing multiple automated registrations. This action is performed only once, upon installation of the client software. It relies on Web-based registration at the AA, or a similar method (Figure 2.6-❶). After the legitimation, the AA returns a client/user identifier (*uid*) to the client. This *uid* is randomly generated over a large

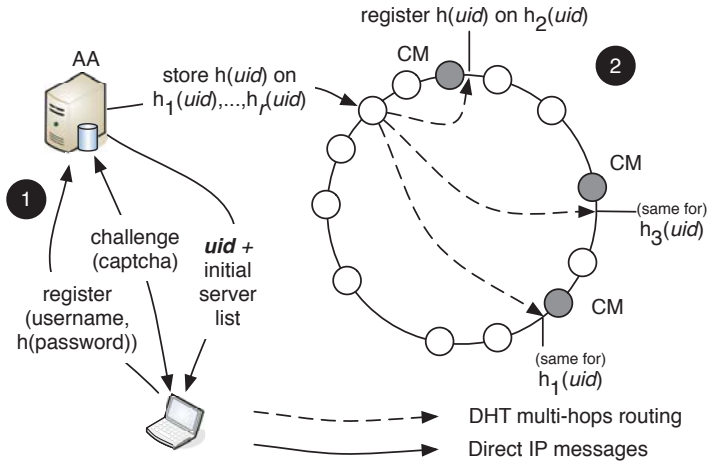


Figure 2.6: SPADS: one-time legitimation and authentication *(while SPADS can be built upon any kind of DHT, we pictured a ring-based DHT [106, 115] for the sake of graphical simplicity).*

identifier space. The AA records the username and the hash of the password of the client to allow for later authentication without the need to re-legitimate the user.

The *uid* is used by the client for authentication when sending data to the DHT. A *uid* cannot be linked to the identity of the client, as only the AA knows the relationship between the *uid* and the username of the client. Nonetheless, there is still the risk that the *uid* be stolen by some other malevolent client, which can then use this stolen identity to obtain illegitimate credentials. To reduce this risk, a timeout can be associated with each *uid*: the client-side software automatically fetches a new *uid* just before the timeout expires, using the username and the hash of the client's password. Note that the risk of having personal information identifying a user (e.g., username/password) stolen and misused is a widespread problem on the Internet and it is not specific to SPADS.

Client authentication

The *uid* generated for a client, either as a renewal or as a first-time authentication, is hashed, signed by the AA's private key, and sent to a set of r locations in the DHT (Figure 2.6-❷). The servers in charge of the corresponding keys,

the *credentials managers* (CM), play the important role of monitoring the usage of credentials by the client and enforcing rate-limitation policies. The r locations are obtained by hashing the *uid* of the client with r different hash functions: $h_1, \ldots, h_r$. These hash functions are known by all servers and by the AA. h_i is typically the SHA1 hash of *uid* using i as the seed of the hash function. Changing the *uid* of the client results in r different positions in the DHT and a new set of different CMs with high probability.

Credential management

Once a CM has registered the hash of the *uid* of some client, it starts provisioning credentials based on time and the rate-limitation policy. Each of the r servers is accumulating credentials for the client separately. The r servers only know $h(uid)$ and have no mean to recompute *uid* itself. Each CM has no possibility to calculate the keys for the $r - 1$ other CMs. This means that the CM cannot cooperate to abuse the system. The management of credentials is a long-term operation. In case one of the CM fails and is not immediately replaced by a neighbor, or if the credentials count is not properly updated because of the loss of a message, the credentials allowance returned to the EP by the various CMs may differ. Using $r \geq 3$ servers allows us to proceed to a majority-vote between the returned values, and to report to the AA the server that consistently responds with an erroneous value compared to the majority. We allow a slight difference in the results returned by the various CMs to take into account the absence of clock synchronization, and delays between servers, e.g., a difference of minus/plus one credential reported by some CMs is not deemed erroneous.

The maximal number of credentials that can be used during a period of time Δ is noted n. On each CM, and for each $h(uid)$ entry, the list of credentials that were granted during the last Δ units of time is kept. The credentials left for some client is simply n minus the size of this list. Nonetheless, when a $h(uid)$ has been registered at some CM, the maximal number of credentials the client can get in the first Δ units of time is proportional to the fraction of Δ spent since registration. This measure is necessary to discourage clients from regularly asking the AA to register again with the system using a different *uid*: the expected number of credentials that the client can get with such re-registration is always less than what the client would have had without re-registering.

There is one exception for regular re-authentications allowed by the pro-

tocol to prevent the stealing of *uid*s: no more than once every s time units (re-authentication period, known by the client and the AA), the AA allows a client to get a new *uid* but copies the value of its credential count to the CMs in charge of this new *uid*.

Server selection

The AA plays the role of a public key server for clients. The certificates are used by clients to select the servers that will participate to the anonymization process. It is necessary that the client knows the public key of these servers beforehand. Asking the servers for their key would expose the client to the risk of having her identity inferred by one of the server during subsequent routing.

Each client obtains an initial list of certificates from the AA after its legitimation and authentication. Certificates allow clients to verify that the public key of some server they would like to use is certified by the AA, and thus that this server's information (IP address) is unique and correct. The client simply verifies by decrypting the signature of the certificate with AA's public key and comparing it to the IP address and public key stored in the certificate.

In the list maintained by the client, each certificate is associated with the time of its reception from the AA. Periodically, the list is cut down from half of its items, by removing the oldest certificates. These removed certificates are replaced by new ones gathered from the AA. Clients keep track of the servers they use to bias the selection process and ensure some load balancing on the servers.

Failed servers are reported by other servers to the AA, which will stop handing their certificate to clients.

Anonymization

SPADS uses a variant of Chaum mixes [23] to achieve publisher-anonymization. The mechanism ensures anonymization even if up to f servers are colluding to get the data in clear form, along with the identity of the client that produced it. A set of $f+1$ servers are selected by the client among the list of certificates it knows. The first server is the EP. The next servers are AIDs. The (k, v) pair, in encrypted form, will follow this path: client $\rightarrow$ EP $\rightarrow$ $\text{AID}_1 \rightarrow \cdots \rightarrow \text{AID}_f$. The objective of the routing and associated encryption is to ensure that the

message ends in clear form at the last AID, but that the identity of the publisher has been lost in the process. Thus, no intermediate node is able to get the content of the message along with the identity of the publisher, even if all nodes in the path until the last node are colluding.

Algorithm 1: Encoding a (k, v) pair at the client.

Notations:
- $[m]_k$: AES encryption of m using key k
- $[[m]]_{\{pub|priv\}k}$: RSA encryption of m

1 **Encode** $((k, v), \{\text{AID}_1, \ldots, \text{AID}_f\})$
2 $\quad$ $\mathtt{s} \leftarrow$ random number (256 bits)
3 $\quad$ $\mathtt{c} \leftarrow [(k, v)]_s$
4 $\quad$ $\mathtt{h} \leftarrow \text{SHA1}(\mathtt{c})$
$\quad$ `// Encode for` AID_f `(without next hop)`
5 $\quad$ $\mathtt{p} \leftarrow [[\perp, \mathtt{h}, \mathtt{s}]]_{AID_f.pubk}$
$\quad$ `// Encode for` $AID_{1\ldots f-1}$ `(with next hop)`
6 $\quad$ $i \leftarrow f - 1$
7 $\quad$ **while** $i \geq 1$ **do**
8 $\quad\quad$ $\mathtt{p} \leftarrow [[AID_{i+1}.\text{addr}, h, \mathtt{p}]]_{AID_i.pubk}$
9 $\quad\quad$ $i \leftarrow i - 1$
10 $\quad$ return $(\mathtt{c}, \mathtt{p})$

The principle of the anonymizing routing is to encode the message with a randomly generated AES key `s` at the client, and to encode this key several times with all the public keys of the AIDs on the path. Note that we do not encode the content itself using the RSA public keys as the size of the (k, v) pair can be of any size: RSA generates encrypted versions that grow larger and larger as the content is re-encoded, and the bandwidth cost increase would be dramatic. This multiple encoding process is described by Algorithm 1.

The client starts by encoding for AID_f (end of the path). It encodes in `c` the (k, v) pair to be sent to the DHT using some randomly generated AES key `s`, and computes the hash of `c` in a variable `h`. Then, the key `s` is encoded together with the IP address of the next AID (which is by convention the null value $\perp$ for AID_f) and `h`. This process is repeated for all AIDs along the path. Note that the field `p`, which contains the information to be transmitted to the next AID, is encoded using that next step's public key and its size will grow

as it contains the information for more and more steps. Conversely, the two first fields (the IP address of the next AID and the hash of the AES-encoded content) are of fixed size (4 and 16 bytes, respectively), which allows each AID to easily separate the information that it needs to transmit.

The multiple encryption of the AES key, using the public keys of all the AIDs on the path, implies that the message has to be decoded by all AIDs to be AES-decoded by the last of them. The client sends the final pair $(\mathtt{c}, \mathtt{p})$ to the EP along with the IP address of the first AID. The EP checks then the credentials for the client and sends the pair to the first AID, which is able to decode the next AID's IP address, the hash of the AES-encoded content, and the RSA-encoded `p` for the next AID. Each AID checks that the hash of the AES-encoded content `c` matches the hash that it got by decoding `p` with its own private key. If this content integrity verification fails, the message is simply dropped. The forwarding process repeats until IP address of the next step equals $\perp$ (this happens at the last AID). Then, the decoded content of `p` gives the AES key `s` for decoding `c`, and the (k, v) is sent to the DHT on behalf of the client.

A correct (non colluding) node only sends to the next AID a pair $(\mathtt{c}, \mathtt{p})$ if it has been able to verify that `c` is valid using the hash `h` embedded in `p`. It is not possible for some AID on the path to corrupt either the AES-encoded message `c` or the RSA-encoded `p`, so as to transmit information for another colluding AID, while *passing through* a non-colluding AID. It is not possible to corrupt `c` (e.g., to append the IP address of the user at the beginning of `c`), as the hash of `c` is checked by each AID on the way to the destination, and the message will be dropped if the check fails at some correct AID. It is not possible either for some AID to corrupt `p` or include more information in it as it would require knowing the private key of the next AIDs.

Correctness

We consider f colluding peers that share all their state and information, including their private keys. The client selects $f + 1$ peers for the EP and the AIDs. There are two cases. The first case is when all the f AIDs are colluding. In this case they all get the content in clear by sharing their private key. The only node that knows the IP address of the client is the EP, but since there are already f nodes colluding, it cannot be involved and therefore, does not transmit the IP address of the client. In the second case, $f - 1$ AIDs are colluding with the EP to get the content along with the IP address

of the client. To get the plain content, its encrypted version needs to go through some f^{th} AID. The IP address of the client is not kept by this f^{th} AID, regardless of its position in the path, and thus not transmitted to the next AID.

One could note that the second case can be prone to an attack by colluding peers based on traffic correlation over time. Indeed, two colluding peers could bypass an intermediate AID and communicate by different means to correlate the fact that two messages they received within a short time window are related, and then virtually isolate the correct AID from the chain. In order to get rid of this risk, each AID may wait for a random amount of time before sending the data further. Note that this has absolutely no impact on the client, since the client is not expecting a reply or acknowledgment for her insertion.

Bulk anonymous put operations

In the current description, only one (k, v) pair is sent with one `a_put`(k, v) operation and using one credential. It is desirable to also allow users to send several (k, v) pairs within the same message using only one credential. The last AID is then in charge of 1. splitting and sending the content as several (k, v) pairs and 2. enforcing the maximal number of allowed pairs per credential used, by dropping remaining pairs after the limit has been reached. It can be noted that this extension modifies the fairness property of the system. Instead of having each inserted pair equally likely to be ignored regardless of its origin, with bulk `a_put`() operations the aggregate sets of pairs are equally likely to be dropped (but their size may differ hence breaking the fairness property).

2.6 Evaluation

In this section, we evaluate the effectiveness of SPADS, the cost it imposes on the servers, and the delays for anonymously sending information to the network. All experiments are based on a prototype implementation running on a cluster. We first explore the average cost breakdown for a single anonymous put operation made by a client. Then, we evaluate the variation in delay imposed by varying f (how many colluding servers are supported at most) and varying the number of credential managers. We do not evaluate the

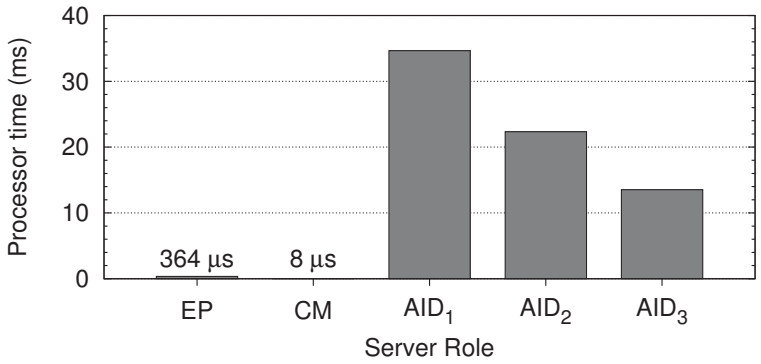


Figure 2.7: Breakdown of the computational cost associated with an `a_put`() operation, for the client and the different server roles (EP, AID_1 to AID_3).

credential management itself, as it does not constitute a performance-critical operation.

Experimental settings

SPADS is implemented in the Lua [65] language, with some performance-critical functions written as C libraries (hashing, encryption and decryption). For the server side, we leverage SPLAY [82] its libraries. The client is implemented in JavaScript. We use an implementation of the Pastry [106] DHT written entirely in Lua. Our Pastry implementation uses the parameters from the original paper. All experiments were run on a cluster of 12 machines, each equipped with a 2.4 GHz Pentium IV processor and 2 GB of main memory. The machines are linked through a 1 GBps switched Ethernet LAN.

Unless explicitly noted, we use the following parameters. All AES keys are 256 bits long, RSA public and private keys use 1024 bits. All hashes use the SHA1 function, which yields 128 bits secure hashes. The size of the DHT is 1,000 nodes, i.e., each machine of the cluster runs several Pastry nodes.

Computational cost at the servers

First, we evaluate the baseline computational cost of using publisher anonymization, with $f = 3$ (i.e., using a chain of 3 AIDs) and a set of 3 CMs. The inserted (k, v) pair is of minimal size: v is a random string of 16 bytes, k a key of 16 bytes (32 bytes total). We measure the processor time spent at

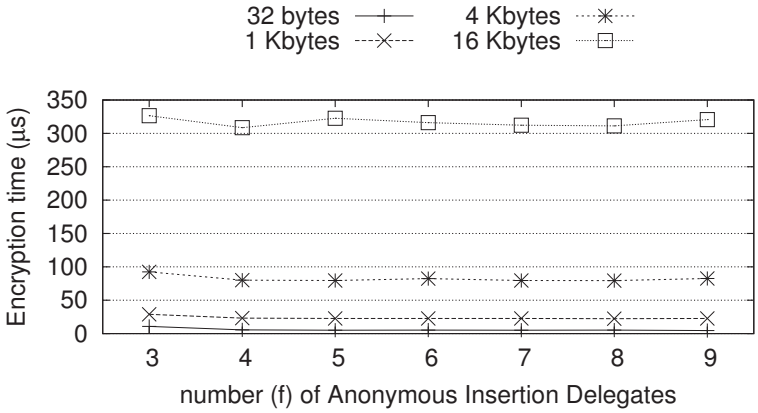
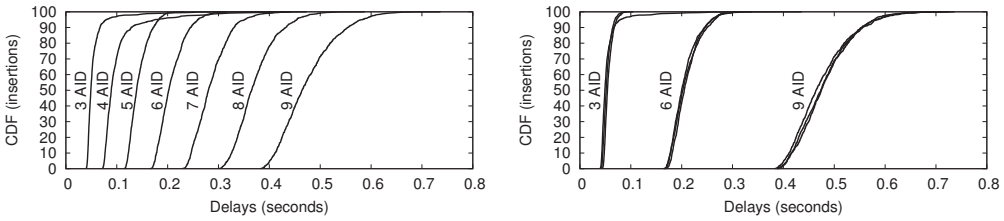


Figure 2.8: Computational cost of the encoding phase at the client side, for varying data sizes and number of AIDs.

each of the servers on the chain, starting from the EP to the last AID, and present the breakdown of these costs in Figure 2.7, averaged over a set of 1,000 anonymous insertions done in sequence. This figure does not take into account the time required for transmitting the message between servers; it corresponds to the processing time at each server between message reception and response generation.

We observe that the rate limitation phase (done by the EP, by asking the CMs) only costs 364 μs for the EP and 8 μs for each of the 3 CMs (a CM only needs to look up a table and reply with the value, while the EP needs to compute 3 hash functions to determine the keys of the 3 CMs. The anonymization phase itself involves some RSA cryptography, and the costs at each AID depends on the size of the message that has to be decrypted using the AID's private key (`p` in Algorithm 1). As the message progresses on the anonymization path, the size of `p` becomes smaller. The computational cost of decrypting the message for some AID is proportional to the number of AID that remain after it in the path, i.e., the number of times the `p` part of the message is encrypted. Note however that these costs are baseline values and are totally independent of the size of the message: only the AES decryption of the message at the last AID depends on the size of the initial message. SPADS exploits an efficient C-backed AES implementation. The average cost for decrypting a message using AES at the last AID is 288 ns, 34 μs and 24 ms for messages of size 1KB, 128KB, and 1MB respectively.



(a) Versus the number of AIDs.

(b) Versus the number of CMs.

Figure 2.9: CDF of the total time spent in the anonymizing path.

Computational cost at the client

We evaluate the time required at the client for encoding a message (a (k, v) pair), as a function of (1) the size of the message and (2) the length of the anonymization path (given by f). Figure 2.8 presents the evolution of the time required for encrypting a message as a function of its size and f.

The cost of encryption at the client is composed of two parts: the encryption of the content using the randomly generated AES key, and the encryption using RSA of that key for the anonymizing chain. The cost of the AES encryption depends on the size of the data, while the cost of the nested RSA encryptions depend on the number of AIDs that are used (f). The minimal size message of 32 bytes gives the baseline for RSA encryption. As expected, the cost of AES encryption increases linearly with the size of the data being encrypted. For the typical size of data items for which SPADS is designed for, this cost is perfectly acceptable.

Anonymization delays

Our next observations are based on the delay between the client's call to `a_put`(), and the moment the message is sent by the last AID to the DHT, on behalf of the client. The delay includes the communication with the EP, and the chain of decryption and forwarding between the AIDs. Moreover, this delay does not reflect a quality of service measure for the client, as the latter does not expect a reply for its insertion. We did not add random delays at each forwarding step. Instead, it allows us to evaluate the scalability of the approach. As network delays are stable and very low in our cluster, a high delay means that more work is done at the servers involved in the process.

Figure 2.9(a) presents the cumulative distribution of delays for a set of 1,000 requests done by 100 separate clients (for each client, insertions are done in sequence) for various values of f, hence various path lengths for anonymization. Increasing the number of AIDs, combined with the linear increase of the processing cost for decoding the RSA-encoded content as the AID is closer to the source, implies that the overall cost is evolving empirically in $O(\sum_{i=1}^{f} i) = O(f^2)$.

Figure 2.9(b) presents the delay variation when both the number of AIDs and CMs vary between 3, 6 and 9. The curves present almost identical behavior for the cases where the number of AIDs is the same; they are therefore grouped by the labels "3 AID", "6 AID" and "9 AID". This means that the impact of varying only the number of CMs is negligible. This behavoir is expected, as the requests from the EP to the CMs are done in parallel.

2.7 Related work

SPADS shares rationales, algorithmic techniques and design choices with a consequent body of previous work. We present related work along the lines of anonymizing systems, P2P overlays for anonymization, and rate limitation mechanisms in decentralized settings.

Anonymizing path

We reviewed onion routing in Section 2.2, among other well-know anonymization techniques. The most widely known implementation of the onion routing is the Tor anonymization layer [36]. Tor allows to tunnel a TCP connection through one or several relay server(s), among the ones that are volunteering to the system. The onion routers are not able to see content of a communication (and the destination of the TCP connection) if they are in a position in the path where they can know the IP address of the client. This concept is similarly used in SPADS.

Torsk [88] proposes to replace the relay selection of Tor, which is based on servers acting as PKS, with a selection based on the Kademlia DHT [87]. Other systems that use Chaum mixes for anonymization are AP3 [91], Bifrost [71], and Tarzan [46].

Publius [124] allows users to publish anonymously Web content but also prevents the data that has been sent by some user from being modified by

another user, by leveraging fountain codes and multiple uses of onion routing.

Finally, Cashmere [132] applies the principle of Chaum mixes for communication anonymization but relies on groups of servers rather than individual ones for retransmitting the data. This allows for a better resilience to failures, as any node in each group of servers can act as a mix and the probability of failure of all nodes in one given group is lowered.

The principle of using groups of servers rather than single servers can be straightforwardly applied to SPADS. It is worth mentioning, however, that using groups of nodes imply that these groups will need to share a common private key, making the power of colluding peers much greater, and thus trading anonymization efficiency for system stability.

We note that, unlike SPADS, none of these systems provides support for limiting the rate of information or the usage of the system by users.

P2P overlays for anonymization

FreeHaven [35] and FreeNet [24] are P2P unstructured overlays that also provide guarantees on the traceability of the data that is sent to the system. Nonetheless, due to their unstructured nature, they do not allow to apply rate limitation to the usage of the system by clients, nor do they allow a complete recall for the requests on the content that has been sent by users.

Rate limitation and spam limitation

Klonowski et al. [70] propose an admission control protocol combined with an anonymous channel protocol. Compared to this system, SPADS does not require that the servers sign each and every incoming and outgoing message. We are not aware of any work that mixes the two goals of *anonymizing publishers* in a distributed setting, and *limiting the influence of cheating* users.

As mentioned in Section 2.3, the problem of verifying that the servers themselves are acting correctly is orthogonal to the problem of validating the usage of the system by the clients. This verification can be done using systems such as the PeerReview accountability system [53], which we described in Section 2.3.

A common technique for reducing the impact of spammers is to blacklist the users that have been detected as corrupted (e.g., by DNS blacklisting, or as part of the DHT protocol, for instance by using the NeighborhoodWatch

DHT [10]). With these approaches, cheaters need to be detected after they have sent their content, which is obviously in contradiction with the objective of publisher anonymization.

SPADS' rate limitation, based on limited credentials allowance for each user that needs to send data to a distributed system, is shared by DQE [125]. DQE's objective is to fight spam in email communications by limiting the rate at which any user can send email. This eliminates the interest of sending spam, which comes from the ease of flooding offered by the email infrastructure. Similarly to SPADS, credentials are managed in a decentralized manner by *quota allocators* nodes, but communications are not anonymized in any manner.

2.8 Summary

In this chapter, we have presented SPADS, a publisher-anonymizing, rate-limiting publication interface for untrustworthy clients willing to send privacy-sensitive data to a DHT, itself based on untrustworthy servers. SPADS uses a combination of cryptography techniques and random path selection (Chaum mixes) to ensure that the identity of the client that publishes some information is lost on its way to the server that can process it, even if at most f servers are colluding to spy on the user. Rate-limitation is achieved by authenticating users with the DHT and using a robust credential management protocol. Experimental evaluation using a prototype deployed on a cluster assesses the applicability of the approach, in terms of computational load and delays.

Chapter 3

Auditing for Large-Scale Distributed Aggregation Systems

3.1 Introduction

In this chapter we present the design, implementation and evaluation of CADA (Collaborative Auditing for Distributed Aggregation). CADA operates in the context of an aggregation network, composed by: *(i)* clients, which send contributions to the aggregations, and *(ii)* servers, that aggregate these contributions. The system model will be explained later with more detail.

CADA provides probabilistic auditing mechanisms to limit the impact of a server that maliciously tries to bias aggregations. The mechanisms provided by CADA can be combined with the functionalities provided by SPADS (publisher anonymization and rate limitation), to build distributed aggregation systems where:

- anonymity is assured, and
- the impact of cheating clients and cheating servers is reduced to negligible levels.

The rest of the chapter is organized as follows. We introduce the system model and the problem definition in Section 3.2. Section 3.3 presents a high-level overview of the oracles. The auditing mechanisms are described in detail in Sections 3.4 and 3.5 for insertion and aggregation biasing, respectively. The oracles are evaluated in Section 3.6. Related work in the field is presented in Section 3.7. Finally, we summarize in Section 3.8.

3.2 System model and Problem definition

In this section, we present the model considered for aggregation middleware and the hypothesis that will later support collaborative auditing.

More specifically, the systems we consider rely on the *aggregation* of discrete *distributions* of data over a set of possible values. A distribution is composed of a set of $< value, counter >$ pairs, where each possible value in the set is associated with a counter of its occurrences as aggregated from client insertions. Note that the number of options is not necessarily bounded *a priori*. Each distribution is associated with a unique *key*, itself associated to a node (or group of coordinated nodes) acting as an aggregation point. Distributed aggregation middleware helps locating this node by employing an indexing mechanism such as a multi-hop distributed hash table [106, 115] or a single-hop routing layer [33]. The node receives and aggregates insertions from clients, and answers to requests for part or all of the distribution values and counters.

Throughout the description, we will refer to the notations summarized in Table 3.1.

We consider an aggregation middleware infrastructure formed by N servers. In the context of CADA, there are two important assumptions that support the design of the bias detection oracles: the *entry points uniform distribution* hypothesis, and the *authenticated servers* hypothesis.

Like in SPADS, servers are polyvalent and can perform any role, depending on the message they receive. The FP is the contact point for clients wishing to send or receive information to the system. Any client can contact any of the servers that will act as an FP for this transaction. This contact server is not necessarily the server performing the aggregation but is in charge of propagating the request in the system to the appropriate aggregation server, and contacting back the client.

The *entry points uniform distribution* hypothesis is the assumption of a random and uniform distribution of the number of requests that are received by each FP and thus propagated to the appropriate AP. This can be enforced by the use of one or several proxy servers or redirect servers (e.g., DNS servers), placed as a layer between the clients and CADA, that distribute the load evenly among all the FPs.

The entry points uniform distribution hypothesis is also enforced when using the SPADS layer to support anonymization, as it exploits multi-hop anonymizing routes following the principle of Chaum mixes and onion rout-

Element	Description
N	Number of servers participating to the aggregation service.
FP_n $1 \leq n \leq N$	The *Forwarding Point (FP)* is the role performed by a server when it is contacted by a client for routing contributions into the aggregation network.
AP_n $(1 \leq n \leq N)$	The *Aggregation Point (AP)* is the role performed by a server when it processes and aggregates contributions propagated from the FPs on behalf of a client, for a distribution it maintains. AP(k) denotes that the AP in charge of the aggregation for the distribution $D(k)$ associated with key k.
$a_k^v \equiv a$ $a_k^v.c \equiv a.c$ $a_k^v.i \equiv a.i$	An *accumulator* is a component of a distribution $D(k)$ for a value v: $D(k) = \{a_k^{v_1}, \ldots, a_k^{v_m}, \ldots\}$. When considering a single accumulator for some oracle, we simply denote it as a and omit indices. $a.c$ (for *counter*) denotes the value of the accumulator, and $a.i$ (insertions) denotes the number of insertions received for computing $a.c$ (i.e., $\frac{a.c}{a.i}$ gives the average insertion value).

Table 3.1: Notations for the base aggregation system without auditing support.

ing [23, 36]. In this context, the last server of the onion chain acts as the FP to the aggregation layer. This last element of the path is selected *uniformly at random* like the other elements from the full set of servers in the system, hence fulfilling the hypothesis. CADA and SPADS are independent and the presence of the latter is not required to support the operation of the former as long as the entry point uniform distribution hypothesis holds.

Furthermore, our approach could be extended to any kind of stochastic distribution of the number of requests per FP as long as the distribution can be given as an input to the oracles or gathered during runtime. These extensions can be seen as future work as it is described in our conclusive remarks (Section 3.8).

The basis of our second hypothesis for the design of CADA oracles, the *authenticated servers* hypothesis, relies on the existence of a trusted Authentication Authority (AA), where all servers register and authenticate, and on the fact that all servers know the public key of every other server. This is needed to support the signing of insertions by the FP (See Algorithm 2).

Algorithm 2: Signature for authentication and uniqueness at the FP level.

Notations:
- $[[m]]_{\{pub|priv\}k}$: RSA encryption of m
- EP_{id}: entry point's identifier
- EP_{pubk}: entry point's RSA public key
- $m + n$: concatenation of the byte chains m and n
- s: sequence number, starts at 0

1 **Sign** (k, v)
2 $\quad \mathtt{h} \leftarrow \text{SHA1}(\mathtt{k} + \mathtt{s} + \mathtt{v})$
3 $\quad \mathtt{p} \leftarrow [[\mathtt{h}]]_{EP_{pubk}}$
4 $\quad \mathtt{s} \leftarrow \mathtt{s} + 1$
5 $\quad$ return $(\mathtt{k}, \mathtt{v}, EP_{\mathtt{id}}, \mathtt{s}, \mathtt{p})$

A server acting as an FP signs the requests received by clients according to Algorithm 2. The FP calculates a hash of the concatenation of the destination key (to certify the routing destination), the request itself (to support an integrity check), and a sequence number (to ensure that a message cannot be taken into account twice at the AP level). The hash is then signed with the

FP private key in order to support authentication, and sent along with the request.

The aggregation layer uses an indexing mechanism such as a multi-hop Distributed Hash Table (DHT) [106, 115] or a single-hop routing layer [33]. The server responsible for aggregating the distribution $D(k)$ associated to key k, and denoted as $\mathrm{AP}(k)$, is located by this indexing mechanism. When the request arrives, the AP first checks the validity of its signature and the authentication of the sender FP, and then processes the request. A request can be a query for part or the entire distribution or an insertion. Insertions increase the total associated with a value v in the distribution and stored in an accumulator a_k^v. The two components $a_k^v.c$ and $a_k^v.i$ contain the total of the *counter* and the number of *insertions* so far, respectively. The *counter* entries $a_k^v.c$ of the accumulators for all values v forms the distribution $D(k)$, while the contributions allow computing the average increment $a_k^v.c/a_k^v.i$.

The case for collaborative auditing

In our large-scale context, the aggregation operation can be performed by any server in the system, and these servers are typically located in several different administrative domains that do not necessarily trust each other. Nodes thus consider that some of the other nodes may wish to influence the system operation in order to bias the distribution aggregates. The bias attempts can take two forms, illustrated in Figure 3.1.

Insertion bias. First, some servers in the distributed aggregation middleware may be under control of an attacker willing to influence the aggregation for some key k managed by another server. In this case, the servers under the control of the attacker may generate fake insertions, e.g., to favor a given value over the others regardless of the inputs of the clients. As an example, in an aggregation system collecting feedback on website accesses and search queries [42], an aggregation server under the control of an attacker may wish to promote a given website regardless of the interest as perceived by users, making the recommendations obtained from user feedback useless.

Aggregation bias. Second, a server proceeding to the aggregation for a key k may wish to return a counterfeit distribution that favors or disfavors one of the values in contradiction with the increment operations received.

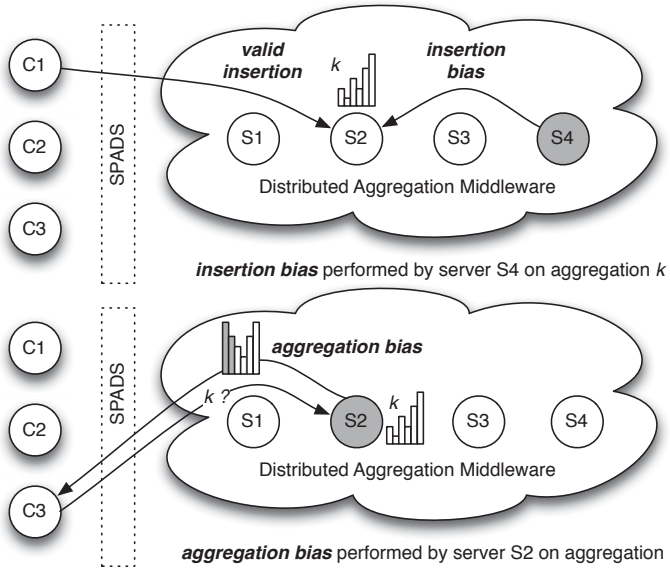


Figure 3.1: Servers in aggregation middleware can bias the results in two ways: by performing fake insertions (insertion bias) or by returning incorrect aggregation results for the keys they are in charge of (aggregation bias).

A typical example is for the aforementioned media popularity application, where the server responsible for the aggregation may be under the control of an institution with financial interests in one of the aggregated values.

3.3 Collaborative Auditing for Distributed Aggregation

In Collaborative Auditing for Distributed Aggregation (CADA), we are interested in detecting the two forms of bias introduced by servers participating to the aggregation layer (see Section 3.1). In this section, we give a high-level overview of the oracles that we use for detecting attempts by the servers to introduce bias.

Adversary model

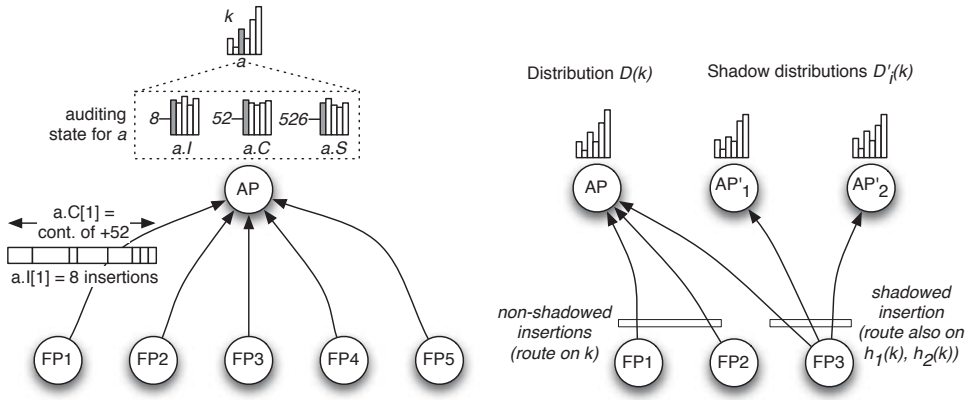
First, servers may exceed their role as FP and influence distributions they are not in charge of, by forging and sending fake insertions to some APs, or modifying insertions received from the clients. A variant of this bias is when nodes acting as FP for clients selectively drop insertions. We treat these two variants of *insertion biasing* conjointly. Second, servers may exceed their role as AP and bias one of the distribution they are in charge of, by returning fake counts for the requests they get from FPs on behalf of clients.

Detecting bias attempts

Our objective in CADA is to propose a set of *oracles* that trigger alarms, or *suspicions*, when detecting that a server is attempting to bias the distribution aggregation operation. This notion of suspicion is probabilistic, and is based on a statistical test over the observed behavior of the server(s) and their expected behavior. As such, it may yield false positives (unjustified suspicions) but a node that is effectively biasing will get a much larger number of suspicions than one that is not.

Our oracles, being probabilistic, are associated to confidence levels that allow expressing a trade-off between sensitivity and number of false positives. The confidence level represents the degree of certainty that one has about a given assumption. For example, if a croupier takes an ace of spades out of a card deck (which has a probability of happening equal to 1/52th, or about 2%), we can say that he is cheating with confidence level of about 98%. That is because after doing a substantial number of tries, if the croupier is not cheating we should get such result only 2% of the time.

Even high confidence levels like the one from the example can easily return a false positive after one try. This is why the output of CADA is not meant to be taken into account to directly ban or punish a server after one suspicion, but to serve as an input for trust or reputation systems that will accumulate suspicions to lower the reputation of a server. As a result, servers can sporadically cheat with a certain probability of not being detected, but a server that intends to cheat frequently, in order to alter the aggregation in a considerable way, will be detected by the accumulation of suspicions.



(a) Insertion bias oracle.

(b) Aggregation bias oracle.

Figure 3.2: General description of the oracles.

Element	Description
$a.I$	Vector of the number of insertions received from each of the FPs. For instance, $a.I[n]$ denotes the number of insertions received from EP_n; and $\sum_{n=1}^{N} a.I[n] = a.i$.
$a.C$	Vector of the sum of the contributions to the accumulator counter received from each of the FPs: $a.C[n]$ denotes the contribution to the total counter $a.c$ received from EP_n. $\sum_{n=1}^{N} a.C[n] = a.c$.
$a.S$	Vector of the sum of the squares of each of the contributions to the accumulator counter received from each of the FPs. If $a.inc_1^n, \ldots, a.inc_m^n$ are the individual contributions received for the accumulator a_k^v from EP_n, then $a.S[n] = \sum_{x=1}^{m} (a.inc_x^n)^2$.

Table 3.2: Additions to the AP state for the *insertion bias oracle*.

Insertion bias oracle

Our first oracle aims at detecting servers exceeding their roles as FP by sending illegitimate contributions that were not initially sent by users, or dropping or altering user contributions. This oracle is illustrated in Figure 3.2(a). It is run by each server as part of its AP role, in a periodic manner, or after a given number of contributions have been received since the last audit. This oracle monitors *independently* the counter for each value in the distribution. We denote the accumulator being audited as a. Based on the *entry points uniform distribution* hypothesis, we can derive two important properties over the set of contributions received for an accumulator: with a large number of inputs to a, the number of contributions received from each FP should be nearly equal, and the distribution of contributions to the total from each FP should also be similar, as the set of input received from each FP is simply a random uniform sample of all the client insertions. The authentication of the original FP is achieved thanks to the *authenticated entry points* hypothesis.

The oracle works by comparing the statistical properties of the set of insertions, and their number, received from each FP for an accumulator a. Note that we do not need to keep the full set of these insertions, but only aggregated values, of size $O(N)$. The additions to the state of the AP necessary to support the oracle are defined in Table 3.2 and illustrated in Figure 3.2(a). We replace the component $a.i$ (number of insertions) by a vector $a.I$ that keeps the number of insertions received from each FP independently. Similarly, the component $a.c$ (total of the accumulator) is replaced by $a.C$, a vector that distinguishes between each FP contribution to the counter $a.c$. Finally, to support the detection of statistically significant variations in the variance of elements sent by each FP, an additional vector $a.S$ collects the sum of the squares of all contributions received by each FP.

The insertion bias oracle operates in two steps. It first checks that the number of insertions for each FP follows a uniform multinomial distribution using a Pearson's goodness-of-fit test. If that is not the case, it removes iteratively the most deviating FP from the test, and checks again the similarity with a multinomial distribution, until a tolerable level of confidence is reached. In the second step it performs again the Pearson test with the remaining nodes, but taking into account the contribution to the total for each FP. This step requires the use of $a.S$, as we detail in Section 3.4. All the FPs that are removed during the test are reported as suspicious. We describe formally in the next section how these elements are used for assessing statistically

Element	Description
$D'_j(k)$ $(1 \leq j \leq R)$	$D'_j(k)$ is a *shadow aggregation* of $D(k)$; it holds a sample of the insertions sent to AP(k).
$AP'_j(k)$ $(1 \leq j \leq R)$	Each $D'_j(k)$ is maintained by a *shadow aggregation point* AP'$_j(k)$.
φ	Is the shadowing ratio; it represents the probability that an insertion being aggregated into $D(k)$ is also sent to the shadow aggregations.

Table 3.3: Additions to the AP state for the *aggregation bias oracle*.

significant deviance, which results in a suspicion for the FP considered.

Aggregation bias oracle

Our second oracle aims at detecting servers that exceed their roles as AP, that is, that do not aggregate properly the insertions they receive but rather favor or penalize some accumulator a over the others in the distribution $D(k)$. The principle of the aggregation bias oracle is illustrated in Figure 3.2(b). The extension required to the aggregation layer is given in Table 3.3.

The main idea of the aggregation bias oracle is to maintain R *shadow aggregations* of the distribution. These shadow aggregations are maintained by R different *shadow* AP's. A system wide parameter, the *shadowing ratio* φ, determines the probability for an FP to send an increment request not only to AP(k) but also to the R shadow AP'(k). The operation done by the shadow AP'(k) is exactly the same as the normal AP(k), except that they operate on a *sample* of all increments received. The identity of the shadow AP'(k) is determined by hashing k with multiple hash functions $h_1(k), h_2(k), \ldots, h_R(k)$ defined system-wide.

As we describe in Section 3.5, the oracle seeks to compare the distribution of values on the main distribution $D(k)$ and the shadow distributions.

Periodic auditing is carried out in two ways. First, whenever receiving a query request for a distribution k, the associated FP performs an audit of AP(k) with probability α. Second, FPs periodically audit random APs. A suspicion is triggered if there is a significant enough probabilistic difference between the main and the shadow aggregations.

3.4 Auditing mechanisms for insertion bias

The insertion bias oracle is called by a server as part of its AP role. The oracle checks all the accumulators a_k^v for all the distributions that this node aggregates. Each accumulator is checked individually. We denote the accumulator under test as a for simplicity. The oracle performs in two steps:

1. First, it detects if insertions were made by some EPs that are suspected of being insertion biasing attempts by forging or dropping insertions. The AP uses an iterative algorithm that checks if the vector of the number of *insertions* $a.I$ represents a uniform multinomial distribution;

2. In a second step, the AP detects FPs that are suspected of performing insertion biasing by modifying the content of the insertions they relay. We use a similar algorithm that operates on the *contributions* $a.C$.

The Pearson's goodness-of-fit test

Based on the *entry points uniform distribution* hypothesis, the vector $a.I = (a.I[1], \ldots, a.I[n], \ldots, a.I[N])$ should be a random vector following a multinomial distribution with uniform parameters $p_1 = \cdots = p_n = \cdots = p_N = 1/N$ in the regular case with no bias. The deviation from this regular case can be tested using a standard statistical *goodness-of-fit* test. We use the Pearson's goodness-of-fit test. It requires to compute the following statistic $T(a.I)$:

$$T(a.I) = \sum_{n=1}^{N} \frac{\left(a.I[n] - \frac{\sum_{n=1}^{N} a.I[n]}{N}\right)^2}{\frac{\sum_{n=1}^{N} a.I[n]}{N}} = N \cdot \frac{\sigma^2(a.I)}{\mu(a.I)},$$

where

$$\mu(a.I) = \frac{(\sum_{n=1}^{N} a.I[n])}{N}$$

is the mean of the components of the vector $a.I$, and

$$\sigma^2(a.I) = \frac{(\sum_{n=1}^{N} (a.I[n] - \mu(a.I))^2)}{N}$$

is the variance of its components. The *no bias* hypothesis can be rejected with a level of confidence c ($0 \leq c \leq 1$) defined as $c = cdf[\chi^2_{N-1}](T(a.I))$,

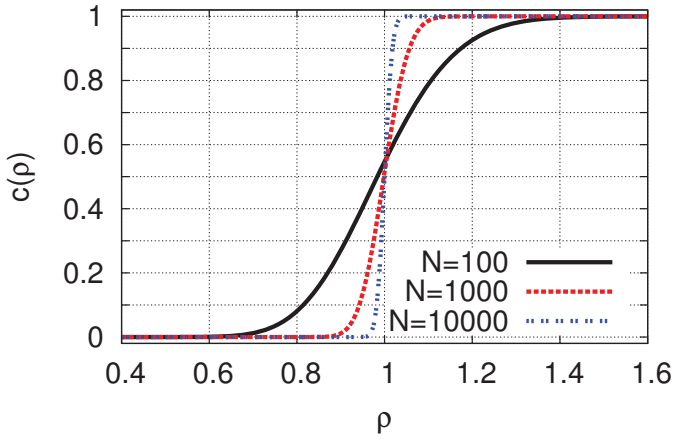


Figure 3.3: Confidence level vs. variance-to-mean ratio.

where $cdf[\chi^2_F](x)$ is the value at x of the cumulative distribution function of the chi-square probability distribution with F degrees of freedom χ^2_F [13]. It results:

$$c = cdf[\chi^2_{N-1}]\Big(N \cdot \frac{\sigma^2(a.I)}{\mu(a.I)}\Big) \tag{3.1}$$

where we denote $\rho = \frac{\sigma^2(a.I)}{\mu(a.I)}$. We observe (see Figure 3.3) that for N sufficiently large, the level of confidence c as a function of ρ, evolves according to a strongly S-shaped curve around the break-even value $\rho = 1$. This indicates that the bias detection will operate in an almost binary fashion, i.e., either $c \approx 0$ (when $\sigma^2(a.I) < \mu(a.I)$) or $c \approx 1$ (if $\sigma^2(a.I) > \mu(a.I)$).

First step: bias by insertion forging or dropping

Both steps of the oracle are based on Algorithm 3. At the beginning of the first step, the server (as an AP) creates a full copy of the set of FPs, named *left*. The corresponding part of the aggregation a that is related to all FPs in *left* is called a_{left}. It creates an empty set of *suspicious* FPs. Thereafter, the AP performs the Pearson's test in an iterative manner: during each iteration, the AP removes the most deviating FP from *left* and places it into the set *Suspicious*. This process is repeated until the number of remaining FPs is less than 2, or the confidence level falls below a confidence threshold. We define

Algorithm 3: Oracle for insertion bias.

Notations:
- c: confidence level for step 1
- c_2: confidence level for step 2
- c_t: confidence threshold
- a_{left}: aggregation for the set of remaining, not-yet-detected FPs
- $\delta_i[n] = (a.I[n] - \mu(a_{left}.I))^2$: deviation of the number of insertions of FP_n from the mean of $left.I$
- $\delta_c[n] = (a.C[n] - \mu(a_{left}.C))^2$: deviation of the number of contributions of FP_n from the mean of $left.C$

```
1  detectInsertionBias(k, v)
2      left ← {FP_1, ..., FP_n, ..., FP_N}
3      Suspicious ← ∅
4      while sizeof(left) ≥ 2 do // Step 1:
5          c ← performPearson(a_left) // See Equation 3.1
6          if c ≥ c_t then
               // picks the most deviating FP:
7              detected ← FP_n : δ_i[n] ← max(δ_i)
8              Suspicious ← Suspicious ∪ {detected}
9              left ← left \ {detected}
10         else
11             break
12     while sizeof(left) ≥ 2 do // Step 2:
13         c_2 ← performPearson2(a_left) // See Equation 3.2
14         if c_2 ≥ c_t then
15             detected ← FP_n : δ_c[n] ← max(δ_c) // most deviating FP
16             Suspicious ← Suspicious ∪ {detected}
17             left ← left \ {detected}
18         else
19             break
20     return Suspicious
```

the confidence threshold c_t as the minimum confidence level that an iteration can yield in order to continue the auditing process. When the process yields less than the confidence threshold, it means that we are less sure than c_t that the distribution is not uniform, and that there has been insertion bias, thus we stop.

Based on this detection process, we can calculate the probability for an FP that is not performing any insertion biasing, to be victim of a false positive as $P[e] = \frac{E(f)}{N}$, where $P[e]$ is the probability of an erroneous detection and $E(f)$ is the expected value of the number of false positives. The probability of having r false positives is $(1 - c_t)^r c_t$ (the probability of making r mistakes and correctly stopping at iteration $r + 1$). It follows:

$$\begin{aligned} P[e] &= \frac{1}{N} \cdot \Big(1(1-c_t)c_t + 2(1-c_t)^2 c_t + \cdots \\ &\quad + (N-1)(1-c_t)^{N-1} c_t + N(1-c_t)^N \Big) \\ &= \frac{1}{N} \cdot \Big[\sum_{n=1}^{N-1} \Big(n(1-c_t)^n c_t \Big) + N(1-c_t)^N \Big] \\ &= \frac{c_t(1-c_t)}{N} \sum_{n=1}^{N-1} \Big(n(1-c_t)^{n-1} \Big) + (1-c_t)^N \end{aligned}$$

And since we can apply:

$$\sum_{n=1}^{m} n.\delta^{n-1} = \frac{1-\delta^{m+1}}{(1-\delta)^2} - \frac{(m+1)\delta^m}{1-\delta}$$

we obtain:

$$P[e] = \frac{c_t(1-c_t)}{N} \Big[\frac{1-(1-c_t)^N}{c_t^2} - \frac{N(1-c_t)^{N-1}}{c_t} \Big] + (1-c_t)^N$$

With N large, components of the form $(1 - c_t)^N$ can be neglected, yielding:

$$P[e] \approx \frac{c_t(1-c_t)}{N} \cdot \frac{1}{c_t^2} = \frac{1-c_t}{c_t N}$$

We present the evolution of $P[e]$ in Figure 3.4, for varying confidence thresholds c_t and several system size N. We can see in this figure that the values of $P[e]$ approach asymptotically to 0 as we increase N. For 50 and more nodes, we have less than 10% of probability of false positives when c_t is more than 20%.

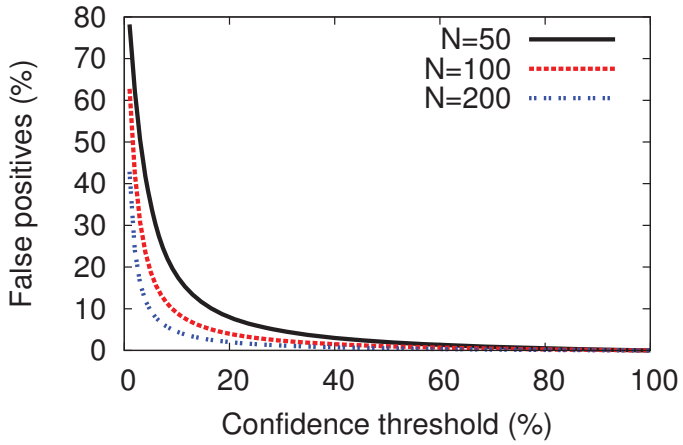


Figure 3.4: Probability of a false positive vs. the confidence threshold c_t.

Second step: bias by insertion tampering

Our second step is based on a variant of the Pearson's test. The AP performs this test iteratively until the confidence level is below the threshold c_t, as in step 1, and at each step it adds the most deviating FP to *Suspicious* and removes it from *left*. Here, the assumption on the distribution of the incoming contribution is not on a multinomial distribution anymore. Indeed, the $a.C$ vector is the result of the combination of the distribution $a.I$ (which is multinomial) and the probability distribution of the value of a contribution on each insertion. This latter distribution is not known *a priori*. In our evaluation of the oracle, we used fixed, Gaussian and Zipf (power law where the i^{th} most popular element out of 100 has a probability $P[n] \propto i^{-1}$) distributions. We present the results for the Zipf distribution as these constitute a worst-case scenario for the oracle, and is more representative of the target application context.

Let $a.C$ be a combination of the multinomial distribution $a.I$ with a given distribution $F(x)$. The resulting mean $\mu(a.C)$ and variance $\sigma^2(a.C)$ are the following:

$$\mu(a.C) = \mu(F) \cdot \mu(a.I), \text{ and}$$

$$\sigma^2(a.C) = \frac{a.i}{N}\Big(\mu(F^2) + \frac{1}{N}(\mu(F))^2\Big)$$

For large N, $\sigma^2(a.I) \approx \frac{a.i}{N}$ and $\sigma^2(a.C) \approx \frac{a.i}{N}\mu(F^2)$, where $\mu(F) = \frac{a.c}{a.i}$ is the mean of contributions per insertion and $\mu(F^2) = \frac{a.s}{a.i}$ is the mean of the squares of contributions per insertion.

As for the previous step, the ratio ρ between variance and mean affects the confidence level. Let ρ_2 be the ratio between the variance and the mean, used in the calculation of c, the confidence factor, for this second step of the test. Noteworthy, ρ_2 can be computed from the ρ used in the first step and observations on F. Since $\mu(a.C)$ and $\sigma^2(a.C)$ change by a factor of $\mu(F)$ and $\mu(F^2)$ respectively, we need to correct T in order to position again the break-even point of ρ at 1.

$$\rho_2 = \frac{\sigma^2(a.C)}{\mu(a.C)} = \frac{\mu(F^2) \cdot \frac{a.i}{N}}{\mu(F) \cdot \frac{a.i}{N}} = \frac{\mu(F^2)}{\mu(F)} \cdot \rho$$

We thus need to use the following correction factor to compute the confidence level in the test of this second step [13]: $\frac{\rho}{\rho_2} = \frac{\mu(F)}{\mu(F^2)}$, which yields the final confidence level c_2:

$$\begin{aligned} c_2 &= cdf[\chi^2_{N-1}]\Big(N \cdot \frac{\sigma^2(a.C)}{\mu(a.C)} \cdot \frac{\mu(F)}{\mu(F^2)}\Big) \\ c_2 &= cdf[\chi^2_{N-1}]\Big(N \cdot \frac{\sigma^2(a.C)}{\mu(a.C)} \cdot \frac{a.c}{a.s}\Big) \end{aligned} \tag{3.2}$$

3.5 Auditing mechanisms for aggregation bias

The aggregation bias oracle also performs in two steps, and it is depicted in Algorithm 4. The first step checks the expected distribution with respect to the shadow distribution based on the insertions, while the second looks at the distribution of contributions.

First step: bias by insertion forging or dropping

We are interested in determining if the number of insertions reported by the main AP and the shadow AP's corresponds to a correct sampling. For any insertion received by the AP, there is a probability φ that it was also received

Algorithm 4: Oracle for aggregation bias.

Notations:
- c: confidence level for step 1
- c_2: confidence level for step 2
- c_t: confidence threshold

DetectAggregationBias(k, v)
- `// First step:`
- c = performNormal(a_k^v) `// See Equation 3.3`
- **if** $c \geq c_t$ **then**
  - return $susp_1$
- `// Second step:`
- c_2 = performNormal2(a_k^v) `// See Equation 3.4`
- **if** $c \geq c_t$ **then**
  - return $susp_2$
- return no_susp

by AP'. The probability distribution of the value $a'.i$ is thus a binomial, with its center at $a.i \cdot \varphi$. For $a.i$ sufficiently large, the binomial distribution $B(a.i, \varphi)$ can be approximated by a normal distribution with mean $a.i \cdot \varphi$ and variance $a.i \cdot \varphi(1-\varphi)$.

For $a.i > 5N$, the normal approximation is considered as valid if [13]:

$$\frac{|\ \sqrt{\varphi(1-\varphi)} - \sqrt{\frac{1-\varphi}{\varphi}}\ |}{\sqrt{a.i}} < 0.3$$

and, if we normalize $a'.i$ by subtracting the mean $a.i \cdot \varphi$ and dividing by the square root of the variance $\sqrt{a.i \cdot \varphi(1-\varphi)}$, we obtain: $z = \dfrac{a'.i - a.i \cdot \varphi}{\sqrt{a.i \cdot \varphi(1-\varphi)}}$ that approximately follows a standard normal distribution $\mathcal{N}(0,1)$.

Thus, the value:

$$c = 2cdf[\mathcal{N}_{0,1}]\Big(\frac{|\ a'.i - a.i \cdot \varphi\ |}{\sqrt{a.i \cdot \varphi(1-\varphi)}}\Big) - 1 \tag{3.3}$$

represents the level of confidence that $a'.i$ does not follow a normal distribution [11] $\mathcal{N}(a.i \cdot \varphi, a.i \cdot \varphi(1-\varphi))$ and thus AP has not reported a trustworthy

$a.i$ value.

Second step: bias by insertion tampering

For any population of values X with size $a.i$, and any size $a'.i$ random samples x drawn from X without replacement, we have:

$$\sigma^2(\mu(x)) = \frac{\sigma^2(x)}{a'.i} \cdot \Big(1 - \frac{a'.i}{a.i}\Big)$$

where

$$\sigma^2(x) = \Big(\sum_{j=1}^{a'.i}(\frac{x[j]^2}{a'.i} - \mu(x)^2)\Big)\frac{a'.i}{a'.i - 1}, \text{ and}$$

$$\mu(x) = \frac{\sum_{j=1}^{a'.i} x[j]}{a'.i}$$

is an unbiased estimator of $\sigma^2(\mu(x))$. Therefore, for $a'.i$ large enough, $z = \dfrac{\mu(x) - \mu(X)}{\sqrt{\sigma^2(\mu(x))}}$ approximately follows a standard normal distribution $\mathcal{N}(0,1)$ [11]. With this additional result, a shadow AP' can compute the confidence level c that AP has not reported trustworthy $a.i$ and $a.c$ values:

$$c_2 = 2cdf[\mathcal{N}_{0,1}]\left(\frac{\mid \mu(a'.C) - \mu(a.C) \mid}{\sqrt{\dfrac{\sigma'^2(a'.C)}{a'.i} \cdot \Big(1 - \dfrac{a'.i}{a.i}\Big)}}\right) - 1 \tag{3.4}$$

where $\sigma'^2(a'.C) = \Big(\sigma^2(a'.C)\Big)\dfrac{a'.i}{a'.i - 1}$.

Of course, if $\sigma^2(x)$ is equal to 0, this estimator does not work. In this case, since the variance is 0, it means that all insertions of the sample contain an equal number of contributions. We can use the same model as in the first step, thus:

$$c = 2cdf[\mathcal{N}_{0,1}]\Big(\frac{\mid a'.c - a.c \cdot \varphi \mid}{\sqrt{a.s \cdot \varphi(1-\varphi)}}\Big) - 1 \tag{3.5}$$

If several shadow AP's are used, the suspicion will be based on a majority voting amongst their respective auditing results, with the leverage of a quorum-based technique. The use of several shadow AP's avoids that a single AP' can trigger fake suspicions to incriminate an AP.

3.6 Evaluation

We present in this section the evaluation of CADA oracles.

Experimental settings

These results were obtained by simulating 200 servers performing the role of FPs, one server AP(k) (henceforth called simply AP) responsible for the accumulator a_k^v, and one shadow AP'$_1$(k) (henceforth called AP') for the aggregation bias oracle. We use only one AP' in our experiments to focus on the auditing process and leave the complexity of the agreement protocol to quorum-based techniques. For the experiments that evaluate the performance of the insertion bias oracle, a subset of 40 of the 200 nodes attempt to bias the aggregation, all in the same manner.

As the communication cost is not important in these experiments, processes were run in our local cluster. Yet, the implementation is full-featured and can readily be deployed in large networks. CADA was developed in Lua [65] and C language, and it uses SPLAY [82] libraries.

Insertion bias oracle

In our first experiment, we observe how the confidence threshold affects the accuracy of the Pearson's goodness-of-fit test. Figure 3.5 presents averaged results from 5,000 experiments with a network of 200 nodes. We consider a scenario where no server attempts to bias the insertions. Before doing the tests, we aggregate 5,000 contributions per FP (for a total of 1,000,000 contributions). One curve shows the result of performing only one iteration of the first step of the insertion bias oracle (i.e., one Pearson test), while the other curve shows the result of performing one Pearson test from the second step of the same oracle. Contributions on each of the insertions follow a 1-100 Zipf distribution (in the 1 to 100 range, and value 1 is 100 times more popular than 100). On the vertical axis we present the percentage of cases where the chi-square test yields a false positive. On the horizontal axis we measure the confidence threshold (on percentage levels) required to trigger a suspicion. As the chi-square test establishes, the expected behavior of both curves is to be linearly dependent on the confidence threshold: if the oracle requires 80% of confidence that the distribution is not multinomial and uniform, it is expected that the test will yield false positives 20% of the times in practice.

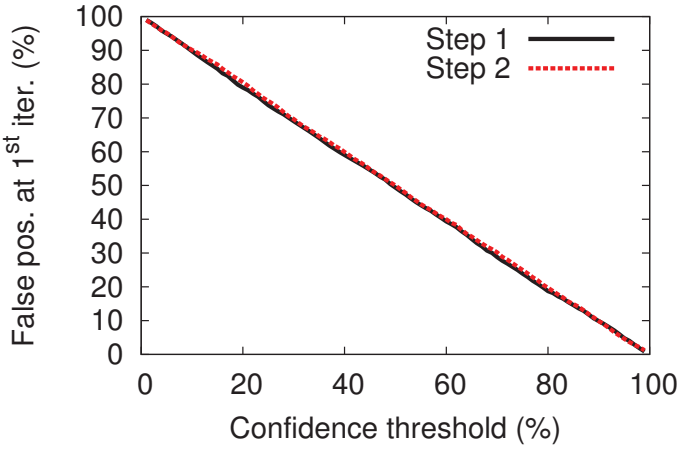
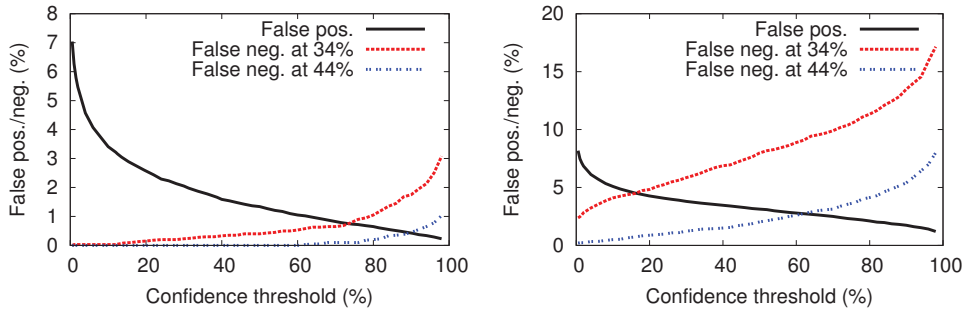


Figure 3.5: Performance of the insertion bias oracle when no FP introduces bias. The graph shows the evaluation of the two steps.

As expected, the curves evolve in a linear way, starting from almost 100% at 0% confidence threshold and decreasing to 0% at 100% confidence threshold.

Figures 3.6(a) and 3.6(b) show the probability for a non-biasing FP to be victim of a false positive and the probability for a biasing FP to not be detected (false negative), depending on the confidence threshold. In our network of 200 nodes, 40 (20% of the population) introduce a bias in insertions while the remaining 160 (80%) have no bias. In order to test the first step of the oracle, the bias in Figure 3.6(a) is performed only through duplication of insertions. For testing the second step, the bias in Figure 3.6(b) is performed only through duplication of contributions on the insertions. Contributions for each insertion follow a 1-100 Zipf distribution. All FPs introduce bias with the same probability. Results from experiments where FPs insert a negative bias (dropping insertions instead of duplicating them, or setting the contributions of insertions to 0) reflected similar results and are not shown due to space constraints.

Both graphs show averaged results from 100 experiments. For each experiment, the insertion bias oracle is activated after collecting 500 insertions for each FP, which corresponds to 100,000 insertions in a situation where no FP performs insertion bias. The percentages that accompany the legends represent the global impact of the insertion bias on the totals (on $a.i$ in the case of Figure 3.6(a) and on $a.c$ in the case of Figure 3.6(b)). This means,



(a) Bias by duplicating insertions.

(b) Bias by duplicating contributions.

Figure 3.6: Performance of the insertion bias oracle when 20% of the FPs introduce bias. Plots show false positive and negative rates vs. confidence threshold. Percentages on the legend represent the global impact of the bias on the aggregation.

for example in Figure 3.6(a), that for a 34% bias level, the AP held about 134,000 insertions in total, about 34,000 of them coming from the 40 biasing FPs.

The probability of a false positive is expected to decrease in a non-linear way, since it is not the result of one chi-square test, but the result of an iteration of tests (see Figure 3.4). The probability of a false negative is expected to grow at a steeper pace in the case of Figure 3.6(b), because we apply a correction of the variance on the formulas, since the distribution to which the vector $a.C$ is compared is no longer a multinomial but the result of a multinomial combined with the distribution of the contributions (Zipf in our case). Since the evolution of false positives does not depend on how much the biasing FPs alter the total, it should be the same for all cases. We observe that this assumption does indeed apply in practice, since the false positive evolution is kept similar no matter how high the bias level. We thus show only one curve for false positives, in the scenario where the global impact is equivalent to 20%.

The graphs clearly show that the probability for a biasing FP of not being detected grows with larger confidence threshold. As expected, the higher the bias level, the higher the probability of being detected for a biasing server, and thus, the lower the probability of false negatives. We can conclude that the choice of the confidence threshold represents a compromise between correctly detecting a biasing FP and not falsely accusing non-biasing FPs in the process.

Interestingly, the curves cross at low percentage values in both graphs (below 5%).

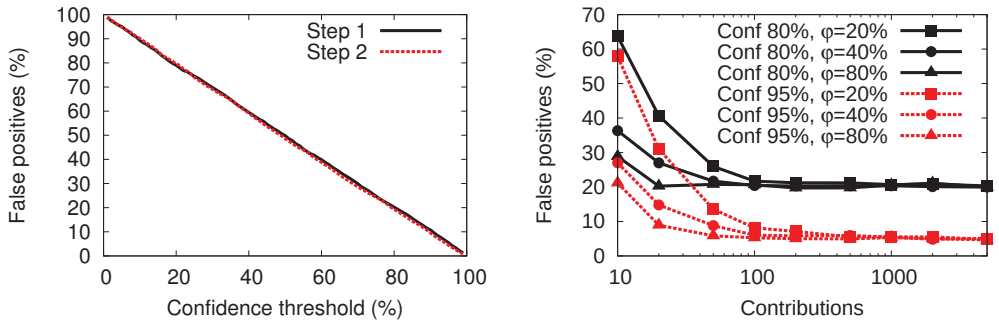
Aggregation bias oracle

Figure 3.7(a) shows the influence of the confidence threshold on the performance of the oracle. On these experiments, averaged from 5,000 runs, neither the AP nor the shadow AP' introduce aggregation bias. The oracle acts after the aggregation of 500,000 contributions on the AP. All FPs send a copy of an insertion to the shadow AP' with a probability of φ =20%. For the evaluation of the second step, contributions on each of the insertions follow a 1-100 Zipf distribution. The figure presents the percentage of cases where the test yields a false positive as a function of the confidence threshold (in terms of percentage) required to trigger a suspicion. Since we can never tell with 100% of certainty that a server is biasing the aggregations (because of the probabilistic nature of the distributions), what we can provide is a confidence level: a level of certainty that such assumption is correct. Figures 3.5 and 3.7(a) confirm in fact that the theoretical confidence levels are correctly reflected in the number of false positives.

For instance, with 80% confidence that the totals in AP do not correspond to the sample in the shadow AP', the test yields false positives approximately 20% of the times in practice.

In Figure 3.7(b) we show the impact of the number of insertions aggregated before performing the tests, for the second step of the aggregation bias oracle. In this scenario the servers do not introduce any bias. We measure the probability of a false positive when increasing the number of insertions. Since both the expected value and the variance of the evaluated aggregation reach a stable value when the number of samples tends to infinity, we expect the results of the test to stabilize after a given number of aggregated insertions. The graph clearly shows that, after 10,000 insertions, the probability reaches a lower bound whose value depends on the confidence threshold (see Figure 3.7(a)). The incidence of false positives in the cases where the confidence threshold is set to 80% and 95% respectively stabilizes at 20% and 5%. The graph also shows the impact of changing the shadowing ratio φ. As expected, the figure confirms that larger shadowing ratios produce more accurate tests. Indeed, the probability of false positives stabilizes faster for a shadowing ratio of 40% than for 20%, and convergence is even faster for 80%.

In Figures 3.8(a) and 3.8(b), we show the probability that the oracle



(a) Evaluation of the two steps. $\varphi = 20\%$.

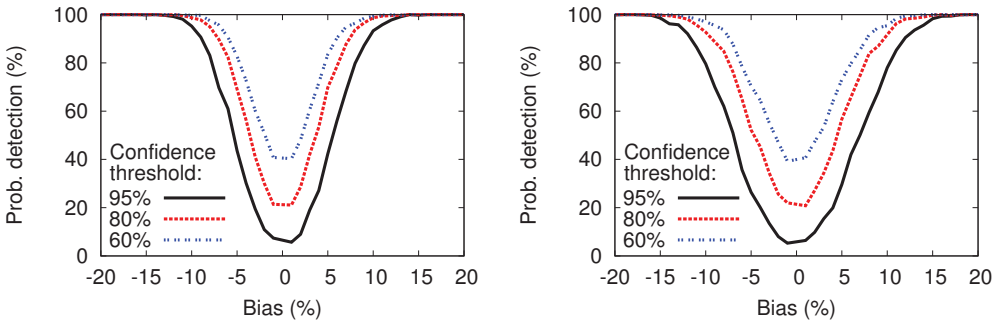
(b) Evaluation of the second step. Contributions on each insertion follow a 1-100 Zipf distribution.

Figure 3.7: Performance of the aggregation bias oracle when AP does not introduce bias.

detects a biasing AP. The probability of detecting the AP is expected to grow and approach 100% as AP introduces a higher bias. In these experiments, we use two models of bias: the AP can either manipulate insertions or manipulate their contributions, which allows us to test the first (see Figure 3.8(a)) and second (see Figure 3.8(b)) steps, respectively.

When the AP drops an insertion or sets its contribution to 0, it produces a negative impact on the component $a.c$ (and also on $a.i$ in the case of dropping the insertion). This effect can be observed in the left side of the figures. When the AP duplicates an insertion or doubles its contribution, it introduces a positive bias by increasing $a.c$. This appears on the right side. The graphs present results from experiments on 5,000 insertions aggregated on the AP. Each value is an average over 1,000 runs. A sample equivalent to 20% of the insertions is sent the shadow AP', and the AP introduces biases from -20% to 20%.

Both graphs show a higher likelihood for a server to be detected when increasing the bias level, on either the negative or the positive side. The curve is symmetric, which results from the detection method that effectively measures the absolute deviation from the expected amount of insertions. The graphs shows three curves for confidence thresholds of 95%, 80%, and 60%. One can observe that, in practice, there is a bias threshold from which a biasing AP will be detected with probability of almost 100%, which meets



(a) Evaluation of the first step.

(b) Evaluation of the second step.

Figure 3.8: Performance of the aggregation bias oracle when AP introduces bias.

our objective. It is important to notice that the minimum values of the curves coincide with the detection probability obtained when the AP does not perform any bias, as shown in Figure 3.7.

3.7 Related work

CADA provides an application-specific auditing framework that ensures a form of low-overhead fraud detection. The suspicions of misbehavior can be leveraged to ensure some confidence on the aggregated data. It operates by means of statistical tests between the expected behavior of servers and the observed one. Other aggregation layers and protocols for large-scale systems include probabilistic techniques to deal with the loss of data elements [62], an orthogonal problem to the accountability of the aggregation servers, or top-k aggregation mechanisms [90].

CADA can be seen as a form of probabilistic failure detector [20] that is specialized for the aggregation operation. Byzantine failure detectors [69] in the context of agreement protocols also produce suspicions of detectable misbehavior but under a more general model of Byzantine behavior. Their generality does however come at a much higher price than the application-specific detection of CADA. In [5], the authors propose to detect byzantine server failures in the context of a replicated database service. The system gathers statistics about the number of faulty servers and their impact on client requests, which can be seen as a common feature to CADA's oracles

mechanisms.

CADA, when coupled with trust or blacklist management, also constitutes an incentive mechanism for preventing servers from deviating from the protocol specification. Such incentive mechanisms are typically used to prevent free-riding [30, 94]. Similarly to CADA, they are application-specific and tailored to a particular kind of Byzantine behavior. Incentive mechanisms however typically only consider the lack of service from free-riders, whereas CADA also considers the case where nodes wish to serve compromised data.

CADA finally relates to accountability mechanisms for distributed systems. The PeerReview [53] accountability framework checks the conformance of a distributed protocol to its specification as a deterministic state machine, by logging the message sent and received by each node and having them replayed by an external (witness) node against the state machine. PeerReview targets *general accountability* and, as opposed to CADA, does not take into account the specificities of the operation implemented by the protocol. This leads to a high overhead both in terms of memory and computation, and puts strong requirements on the protocol. The focus of CADA is different in that it tolerates some limited deviant behavior from a server in the aggregation layer before issuing a suspicion. Note also that, unlike PeerReview, CADA does not log the network activity for each node and as such cannot support verifiable evidence of misbehavior.

CATS [128] is another accountability mechanism that, similarly to CADA is application specific (it targets a storage service). Similarly to the AP auditing mechanism of CADA, CATS checks the legitimacy of insertions and modifications to the stored elements, but does so in a comprehensive manner, checking all incoming messages as in PeerReview. This results in a larger overhead than with the node-local operations performed by CADA oracles.

Since CADA is a heuristic auditing system (it does not check every message sent or received like PeerReview or CATS), it is lightweight and highly scalable. In fact, tests in Section 3.6 show that increasing the rate of insertions or contributions improves the accuracy of the system, without any compromise on computing load, because the statistics that the system use are performed on the aggregation, not on individual insertions.

3.8 Summary

The aggregation of distributions is a fundamental component of many large-scale distributed applications. However, in order to enforce the liability of a service based on aggregation, such as a monitoring infrastructure, a feedback aggregation and recommendation layer, or a trust management system, some guarantees on the good behavior of the servers collaboratively providing the service must be enforced. In particular, it is necessary to discourage biasing behaviors where a node in the aggregation layer may want to compromise either the distributions it maintains or the distribution maintained by others. In this chapter, we presented the CADA oracles, which consider two misbehaviors that servers may implement: insertion and aggregation biasing. The oracles assess the probability of correct behavior of the servers according to statistical tests over the distributions of information at aggregation peers and as received from clients. Based on these tests, CADA issues suspicions for servers that deviate from the expected behavior observed in the rest of the system.

Chapter 4

Framework for the evaluation of Distributed File Storage Service prototypes

4.1 Introduction

At present, the research community lacks tools that facilitate the development, testing, and benchmarking of Distributed File Storage Services (DFSSs), and that can compare the advantages and disadvantages of tweaking properties of the underlying algorithms, e.g., consistency model, number of replicas, or block size.

As it was mentioned in Section 1.2, practical DFSSs propose not only different File System Consistency Levels (FSCLs), but they also have different base performance and optimization levels. This makes difficult to perform a fair benchmark comparing different FSCLs.

In this chapter we introduce FlexiFS, a flexible distributed file system framework for the fast prototyping and benchmarking of distributed file systems, based on the open-source SPLAY [82] platform.

We make a step toward allowing the systematic comparison of DFSS designs. We instantiate our approach by isolating and evaluating the impact of the FSCL on performance.

The remainder of this chapter is organized as follows: In Section 4.2, we explain concepts that are needed for the understanding of our design. We introduce and explain the design of FlexiFS in Section 4.3. In Section 4.4, we

present the six different FSCLs that FlexiFS offers. Section 4.5 presents the algorithms that correspond to each of the six FSCLs. Section 4.6 describes the implementation and deployment of FlexiFS as part of the SPLAY platform. We present in Section 4.7 several benchmark results that evaluate each FSCL. Section 4.8 leads the discussion of other work related to the topic. We finally summarize in Section 4.9.

4.2 Background

The CAP Theorem

As seen in Section 1.1, replication is used in Large-Scale Distributed Storage Systems (LS-DSSs) as a mean to address challenges such as permanent storage and high availability. The use of replication raises new challenges, such as how to maintain consistency between replicas.

An ideal LS-DSS would have full consistency and full availability, i.e., when an update is made, all observers can see immediately that update. It will be explained below that this is impossible to achieve. Database systems in the late 70s experienced this difficulty; people working on such systems knew empirically that there is a trade-off between these two features, and leaned towards offering consistency in detriment of availability. This lead to the conception of the ACID (Atomicity, Consistency, Isolation, Durability) properties in databases, defined by Jim Gray during the 70s, and more formally presented in 1983 by Theo Haerder and Andreas Reuter [54].

In the 90s large Internet services arose, and for these systems, availability was the biggest priority. This change in needs derived in the conception of an alternative model called BASE (Basic Availability, Soft State, Eventual consistency), which is oriented to systems that aim for high availability.

The first time that the trade-off between availability and consistency was presented in an explicit way was during the keynote address to the Principles of Distributed Computing conference (PODC) in 2000. Professor Eric Brewer from the University of California at Berkeley presented in this keynote the *CAP Theorem*. The theorem was later presented in a more formal way in an article by Seth Gilbert and Nancy Lynch [52].

There are three properties a distributed system can provide:

Consistency: Full consistency is achieved if, for all operations, there is a total order such that each operation looks as if it were completed at a single

instant. This is the same as to say that for a distributed system, operations must look as if they are executed in a single node. Full consistency is not achieved if, for example, a read operation that was executed by the system after a write operation retrieves a value that is prior to this write.

Availability: Full availability means that every request that is handled by a non-faulty node must have a response. It also means that every algorithm that is used by the system must eventually terminate.

Partition tolerance: A network is partitioned when all communication between a set of nodes and another set of nodes of the network is completely interrupted. To be partition tolerant, a network must still function even if all messages sent from one partition (set of nodes) to another are lost.

Brewer stated during his keynote that a distributed system can achieve at most two out of these three properties. Gilbert and Lynch presented in their article a formal proof of this statement. The theorem has served as base for distributed systems design since its conception and up to this day.

Systems that forfeit Partition tolerance are called CA, because they offer Consistency and Availability. Examples of such systems are traditional cluster and single-site databases, non-distributed file systems, and distributed systems such as Greenplum [123], Vertica [75].

Systems that offer Consistency and Partition tolerance are for example Google's BigTable [21] and Spanner [29], Apache HBase [8] and Oracle Berkeley DB [98].

AP (Availability and Partition tolerance) systems are for example Amazon Dynamo [33], LinkedIn Voldemort [103], Yahoo! PNUTS [28], and Apache Cassandra [74].

Nowadays, large-scale systems are often composed of hundreds or thousands of nodes (which are prone to failures), and spread across several data centers, which can be geographically separated, even by large distances. Therefore, network partition is a scenario that designers of such systems often consider. Thus, most distributed systems fit either into the CP or the AP model.

The CAP theorem was revised in 2012 by Brewer himself [14]. In this article, Brewer states that the "two out of three" condition is often used in a too stric way, and that is a vast range of flexibility for handling partitions and recovering from them.

Nowadays, several algorithms are used to provide different guarantees along the CP to AP spectrum. We explain below Vector Clocks and Paxos, which are the ones implemented in FlexiFS.

Vector Clocks

Vector clocks is an algorithm for generating partial ordering of events and detecting causality violations. The algorithm was developed by Colin Fidge [44] and Friedemann Mattern [86] in an independent manner, in 1988. The scenario for vector clocks is a set of processes that can exchange information only through messages. When a process sends a message, it sends the state of its own logical clock. A vector clock for a set of N processes is an array of N logical clocks.

The logical clocks are actually counters that start at value 0 and behave according to the following rules:

1. When a process experiences an event, it increments its logical clock by one.

2. Each time a process sends a message, it increments its logical clock by one and then sends the entire vector with the message.

3. Upon receiving a message, a process increments its logical clock by one and updates its vector by taking the maximums between the values in its own vector clock and the vector in the message.

By following these rules, processes can keep track of events, and thus, can compare, through the events timestamps, if two events are causally related, and if so, which one occurred first.

Vector clocks is used in several distributed systems (e.g., Amazon Dynamo [33], Apache Cassandra [74]) to achieve causal consistency (a variety of eventual consistency) among replicas.

The Paxos algorithm

Leslie Lamport proposed Paxos as a family of protocols for solving consensus between unreliable processes. Although similar work on the topic was already done [12, 39, 113], Lamport was the first to present a resilient consensus protocol for asynchronous networks, with a proof of correctness.

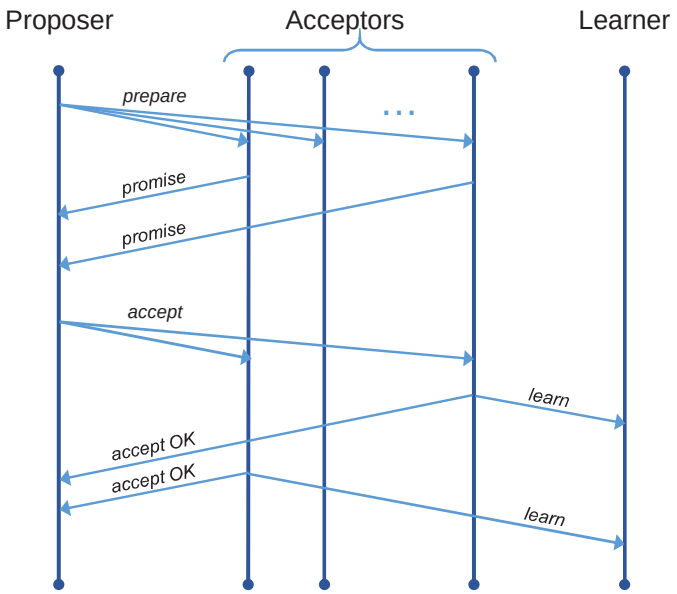


Figure 4.1: Failure-free execution of the Basic Paxos protocol.

Paxos was formally introduced in 1998 [77]. However, Lamport developed it and used it since the late 80s. Lamport described the algorithm through a fictional parliament established in the greek Paxos islands. He wrote a more concise description in 2001 [78].

The scenario for the Paxos algorithm is a network of processes that are prompt to crash-recover failures, and need to agree upon a given value. These processes were depicted by Lamport as Greek legislators that worked part-time, and thus could leave the parliament at any time, and for an unbounded amount of time.

In order to describe the Paxos algorithm, we define the following roles:

- **Proposer:** proposes the request to other nodes, and coordinates the progression of the protocol.
- **Acceptor:** receives proposals from the Proposer and determines whether to accept or not the aforementioned proposals.
- **Learner:** receives the instruction from an Acceptor to store values.

The failure-free execution of the Basic Paxos protocol is depicted in Figure 4.1. The Proposer sends a *prepare* message to all other processes, which act as Acceptors. The Acceptor will check if the Proposal ID that comes with the message is higher than any proposal that was already agreed upon. If this is the case, the Acceptor replies *promise* with the values of the proposals it has accepted. If not, the Acceptor replies *reject prepare* and the highest proposal ID that it is aware of.

If the Proposer receives a sufficient quorum of *promise* messages (typically, half of the nodes plus 1), it will proceed with the propagation of an *accept* message, that is sent to each of the Acceptors that responded positively. Upon reception of an *accept*, an Acceptor checks again if the proposal ID is still higher than any other accepted proposal. If so, the Acceptor answers *accept OK* to the Proposer and also sends a *learn* message to the Learners.

The transaction is finished and successful, when a sufficient quorum of processes answers *accept OK* to the Proposer.

The Paxos algorithm guarantees correctness over liveness. A well-known scenario where liveness is not guaranteed is the dueling Proposers. In this case, two processes are trying to propose values, and each process alternatively interrupts the Paxos algorithm that the other process is executing. This scenario can go eternally, unless one of the processes decides to wait some time before sending a new proposal, in order to let the other to finish.

There are several variants of the Paxos algorithm, which adapt to specific scenarios and requirements. Multi-Paxos is an optimization for the case when the Proposer is usually the same node; in this case, the first phase can be avoided. Cheap Paxos [81] is designed to tolerate f failures; it requires the presence of $f + 1$ main and f auxiliary processes. Other variants are Fast Paxos [80], Byzantine Paxos [19] and Generalized Paxos [79].

The basic operation of Paxos is simple from the point of view of the application that leverages it. Nodes can *propose* (put) values, and query for a value. When a value is returned (get), Paxos guarantees that this is a agreed-upon value between the nodes participating to the run. Paxos is widely used in fault-tolerant distributed systems, state machine replication, and synchronization services (e.g., Google Chubby [15], Apache Zookeeper [63]), to decide the order of transitions to apply to a deterministic state.

POSIX File Systems and FUSE

In a general way, a file system is a process or set of processes that allows to access a storage device (or a group of storage devices), through items called files.

Basically, a file is an entry in a directory. This entry points to an *inode*, which holds the file metadata. The standard attributes that a POSIX file has are the following:

- **mode**: the file's 3-digit access rights (for *user*, *group* and *others*).
- **ino**: the inode number.
- **dev**: the device number.
- **nlink:** number of links (number of files associated to this inode).
- **uid, gid:** the inodes user and group IDs.
- **size:** file size in bytes.
- **atime, mtime, ctime:** date and time of last *access*, *modification* of the file contents, and *change* in metadata, respectively.

They inode does not keep the file data itself, but pointers to blocks that contain the data. However, an inode can be associated to several files; these files are to be called "hard linked".

Directories contain lists of *files*: a character string (the file name), linked to a pointer to the corresponding inode. POSIX file systems always have a root directory, which starts the directory hierarchy and is represented by the symbol "/".

The manipulation of a file in POSIX file systems is session oriented; `read` and `write` operations are always enclosed with an `open` and a `release` operation. On the other hand, between and `open` and a `release` operation, there is the concept of a "session", where many reads and writes can occur.

POSIX file systems normally reside in the kernel space, as they have the goal to achieve optimal performance through direct access to the machine's resources. However, developing in user space has some advantages (e.g., the file system is easier to develop and to debug, no kernel recompilation is needed, bugs do not take down the entire system). Because of these reasons, a need for user space file systems emerged.

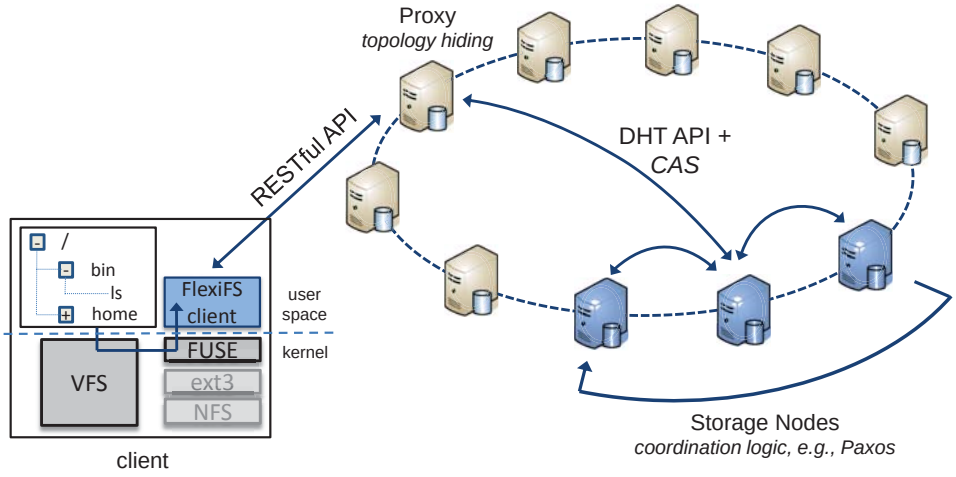


Figure 4.2: General architecture of FlexiFS.

File System in User Space (FUSE) is a framework that allows to create a file system with code written entirely in user space. It was originally developed to support AVFS [1], but it is currently a separate project. FUSE is currently available for several operating systems, such as Linux, FreeBSD, NetBSD, Minix 3, Android and MacOS X.

The framework is composed of a kernel module, a user space library and a mount utility. The kernel module is loaded in privileged mode, and it comes with the Linux kernel since version 2.6.14. The user space library offers a set of functions that are called when the user accesses files in the file system. We present in Table 4.1 an excerpt of the most important FUSE operations and their expected behavior.

Our use of FUSE will be described with more detail in Section 4.3.

4.3 FlexiFS design and architecture

FlexiFS is a DFSS offering a transparent file system interface. Figure 4.2 illustrates its general architecture. FlexiFS has been built in a modular way to allow evaluation of different choices of DFSS designs. A typical deployment

[1]Since several processes can open concurrently the same file, it is convenient to keep a counter instead of a true/false flag.

Operation	Expected behavior
`create`	Receives a file name as argument. If a file with such name does not exist, creates an inode for an empty file and leaves it open (a special *open_sessions* counter[1] is initialized to 1).
`mkdir`	(Make Directory). Creates a directory.
`getattr`	(Get Attributes). Retrieves all the standard file metadata used by the operating system. One can always store extra metadata inside an inode if wanted, but only standard information will be retrieved by `getattr`.
`open`	Retrieves the inode and increments the number of open sessions (for a closed file, $open\_sessions = 0$).
`read`	Reads bytes from an open file. In fact, the operation is provided already an inode structure (which is retrieved during an `open` or `create` operation), so, it is not necessary to retrieve the inode again from the storage facility.
`write`	Writes a sequence of bytes to an open file. An offset is specified. If the length of the chain plus the offset is bigger than the original file size, the file size is increased to fit the sequence.
`truncate`	Deletes all bytes of an open file, starting from a specified byte index. If the byte index is bigger than the file size, then no byte is deleted, and 0s are appended in order to fit the new size.
`flush`	Forces changes in memory to be written to the physical storage.
`release`	Decrements the number of open sessions. If it reaches 0, writes on physical storage all the metadata changes (dates, block list, etc.) from the current session, in order to close the file.
`rename`	Changes the name of a file. Equivalent to a "move" operation.
`link`	Creates a file which points to the inode specified in the arguments. It does not create a new inode.
`unlink`	Deletes a file; if the inode that this file is pointing to does not have any other file associated to it, it deletes the inode too, and optionally the blocks associated to the inode.
`symlink`	Creates a file and its corresponding inode; inside the inode, instead of creating a list of blocks, it places a pointer to another file.

Table 4.1: Important FUSE operations.

contains two sets of nodes: *Storage Nodes* implement a distributed flat storage layer, while *client nodes* present a file system abstraction to the users, and store file and directory hierarchies on the Storage Nodes.

The user has access to the files through a file system based on the FUSE module (see Section 4.2).

Proxy

In FlexiFS, each access to the file system is transformed into a HTTP/REST instruction (GET, PUT, DELETE methods over HTTP), and routed toward a *Proxy* node that acts as an entry point to the distributed storage system. The Proxy redirects requests to the adequate Storage Node(s), which store or return data blocks.

Thanks to the use of a RESTful API, the Proxy can be implemented with any lightweight HTTP server. Other well-known services that offer RESTful interfaces are CouchDB [7], Amazon S3 [6] and MongoDB [92], for example.

The role of the Proxy is to hide the topology of the distributed storage from the client. In FlexiFS, *any Storage Node can act as a Proxy.* Therefore, for clients to have access to FlexiFS, they must only know the IP address of one of the Storage Nodes. This information can be easily provided by a DNS server, for example. When a client executes an operation and contacts a Proxy via its Web service interface, the Proxy accesses the underlying storage system, executes the operation, and returns the result to the client.

Distributed Storage Layer

FlexiFS is modular and decouples the file system logic from the actual storage.

The storage layer is essentially a key/value store extended with a Compare-and-Swap (CAS) primitive. Devising a file system on top of this interface is a contribution of our work. The operations of the interface are as follows:

- `put`(k, v): writes the value v for key k,
- `get`(k): returns the data stored for key k,
- `cas`(k, u, v): checks whether the stored value is still u, and if so, replaces u by v; in any case the old value is returned,
- `delete`(k): removes the element with key k.

Depending on the file system consistency level in use in FlexiFS, the semantics of the above interface may change. For instance, CAS is not atomic under eventual consistency. We detail how FlexiFS implements this interface in Section 4.5.

The storage layer supports data indexing. Due to its modular design, FlexiFS is able to use different indexing and storage layers, such as a multi-hop Distributed Hash Table (DHT) or a central server. We detail below a common design in existing flat storage layers [33, 74], that we also use here.

FlexiFS' indexing and storage layer is a simple yet efficient one-hop DHT structured as a ring that relies on consistent hashing [67] to store and locate data. Figure 4.2 presents its general architecture. It supports the following features:

Routing. For performance reasons and in order to reduce noise in our experiments, we have chosen a one-hop routing design, i.e., every node knows all other nodes in the ring.

Elasticity. Upon joining, a node chooses a random identifier along the ring and fetches the ring structure from some other DHT node. It then informs its two direct neighbors that it is joining.

Storage. FlexiFS uses consistent hashing to assign blocks to nodes with replication factor r: a block with a key k is stored at the r nodes whose identifiers follow k on the ring.

Failure detection. Each node periodically checks the availability of its closest successor on the ring, and repair mechanisms are triggered upon a lack of response within a timeout.

A gossip mechanism spreads topological changes throughout the ring. Each node notifies its closest neighbor whenever it learns about a leave/join event. If the time to spread a message along the ring is shorter than the time between two leave/join events, this mechanism is guaranteed to maintain the ring topology. In our experience, such an assumption is reasonable for a deployment size of a few hundred Storage Nodes or less (our typical testbed size). For larger network sizes, this simple gossiping mechanism can be replaced with more complex mechanisms [48, 66, 122].

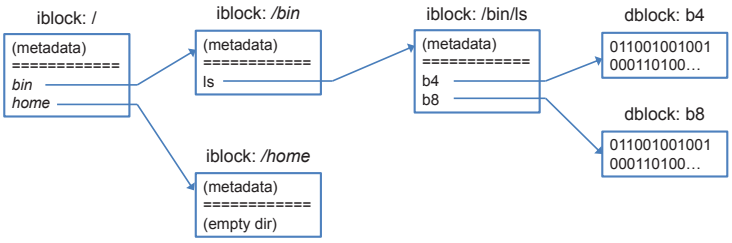


Figure 4.3: Example of a file system structure stored in FlexiFS.

File System

Like most contemporary DFSSs, FlexiFS decouples metadata from data storage. For each file, an inode block (`iblock` hereafter) contains the metadata information about the file, e.g., size and user/group ownership. One or more data blocks (`dblocks`) hold the content of the file. FlexiFS provides several hooks to tune how files are stored.

Figure 4.3 illustrates our current design: `dblocks` are of constant and configurable size. The default size of `dblocks` is 128 kB, matching the maximal payload size for FUSE read and write operations. If an `iblock` represents a regular file, it contains, along with metadata, the list of `dblocks` that are associated to that file. An `iblock` representing a directory contains the list of files that reside in the directory. Compared to the typical redirection-based architecture of Unix, the above mechanisms help reducing the network overhead [51].

Both `iblocks` and `dblocks` are represented as elements of the same key/-value store, where they get replicated according to the different consistency models. Only `iblocks` are mutable. The key of a `dblock` is equal to the hash of its content. This ensures good balancing of the data across Storage Nodes in order to deliver aggregate performance and increased fault tolerance. In case of an `iblock`, the client generates a unique key at creation time, based on a hash of the concatenation of the session number and an incremental sequence number. The session number is system-wide unique, and it is obtained when the file system is mounted. Session numbers are issued and managed by a central authority.

FlexiFS has several built-in sharing semantics and their corresponding implementations, which we describe in the next section. Additional semantics can be easily added thanks to FlexiFS' modular design.

Amazon S3 API

As mentioned before, the Proxy presents a RESTful API to the client. This API is compatible with the Amazon S3 [6] API. Amazon S3 stands for Simple Storage Service, and it is a distributed storage service that arranges items by *buckets* and *keys*. A bucket is a repository of objects, identified with a unique name. A key is the identification for an object. A key must be unique only within a bucket. Amazon S3 is nowadays one of the most popular online storage services.

4.4 File System Consistency Levels

An important design aspect for a DFSS is defining the semantics of sharing, i.e., how clients accessing simultaneously the same file observe modifications by other clients.

In Section 4.2, we have made an overview of the CAP theorem and several mechanisms to keep consistency between replicas. In this section, we describe several consistency levels and their relation with DFSS.

Overview

In Section 1.1, we defined a DFSS as a distributed storage system where clients perceive the data as a local file system. This definition leads us to separate DFSS functionally in two parts: (i) the innards of the DFSS, i.e., a key/value store; and (ii) the front-end to the client, i.e., a file system.

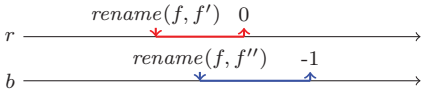
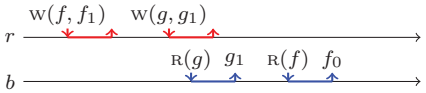
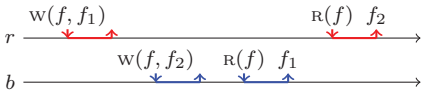
According to this distinction, we classify the semantics of sharing with

- the consistency level from the key/value store perspective, and
- the use (or not) of the close-to-open semantics (point of view of the file system).

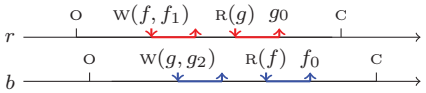
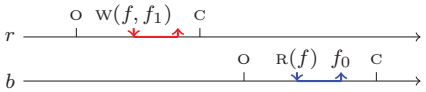
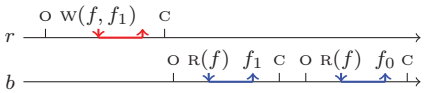
The combination of these two parameters defines a FSCL.

Consistency levels at the key/value store

The three consistency levels at the key/value store are the following:

Consistency of File Operations	Example	Implementations
Linearizability		POSIX [64]
Sequential Consistency		Sprite [96], GoogleFS [51], HDFS [45]
Eventual Consistency		Pastis [16]

(a) *Without* close-to-open semantics.

Consistency of File Operations	Example	Implementations
Linearizability		AFS [61], NFS [118]
Sequential Consistency		SinfoniaFS [4]
Eventual Consistency		Coda [111], Ivy [93]

(b) *With* close-to-open semantics.

Figure 4.4: File System Consistency Levels *(operation* $\text{W}(f, v)$ *means a write to file* f *with value* v*;* $\text{R}(f)$ *is a read on* f*;* O *and* C *respectively open and close all files accessed during the session).*

Linearizability. The most powerful synchronization level for processes in a distributed environment is obtained through the use of atomic, or *linearizable*, objects [57]. A linearizable object is a shared object that provides the illusion of being accessed locally. More precisely, this consistency level states that each operation takes effect instantaneously at some point between its invocation and response.

Figure 4.4(a) presents an execution of linearizable operations. The blue client (represented also with the letter b) renames file f to f'. Concurrently, the red client (r) renames file f to f''. Since operations are linearizable, one of the two accesses must fail.

A common way to implement linearizability is to use the Paxos [77] algorithm (see Section 4.2, which provides consensus among all replicas of any key/value record, for each read and write operation.

Sequential Consistency. Under sequential consistency, "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [76]. Sequential consistency is weaker than linearizability. In particular, this consistency level is not composable [57]: even if each file is sequentially consistent, the file system as a whole is not sequentially consistent (this is also called the *hidden channel* problem). We illustrate this issue in Figure 4.4(b) (middle). In this figure, accesses with respect to file f are sequentially consistency, and similarly this property holds for g. However, the execution (i.e., when we consider both f and g as a whole) is not sequentially consistent.

A way to implement sequential consistency is using primary replication. For each record, a primary replica is elected. Upon a $\texttt{put}(k, v)$ call, the primary for key k sends to all replicas the value v and then waits until a majority of replicas acknowledges the reception before returning to the proxy. To execute a $\texttt{get}(k)$ call, the proxy accesses any replica of k that contains the version it previously read, or a newer version.

Eventual Consistency. Under eventual consistency [41], there must exist a monotonically growing prefix of updates on which correct replicas agree. Since there is no assumption on the time of convergence, eventual consistency does not offer any guarantee on the return value of non-stable operations (that do not belong to the common prefix).

Figure 4.4(b) (bottom) depicts a run under eventual consistency. In this figure, both b and r clients write then read file f. Because the r client reads version f_2 while the b client reads version f_1, no linearization of the four operations can satisfy the returned values.

Close-to-Open Semantics

Under POSIX semantics, almost all file operations shall be linearizable [64, page 58]. In particular, a read shall see the effects of all previous writes performed on the same file. The CAP impossibility result [52] tells us that such constraint hinders the scalability of a DFSS, because it will limit either the Availability or the Partition tolerance of the system.

By introducing the notion of *file session*, close-to-open semantics [109] aim at reducing the amount of synchrony required to access shared files. A file session is a sequence of read and/or write operations enclosed between an *open* and a *close*[2] invocation. It has the following properties [83]:

(1) Writes to an open file are visible to the client but are invisible to remote clients having the same file opened concurrently.

(2) Once the file is closed, changes are immediately visible to sessions that are starting afterwards.

Since operations execute in isolation and either all writes or none execute, these sharing semantics are close to the familiar notion of transaction. Notice, however, that close-to-open semantics apply the last writer wins rule: two concurrent updates do not abort, one of them is simply overwritten.

The definition of close-to-open consistency above has been formulated with atomicity in mind. One can actually combine close-to-open semantics with sequential consistency and eventual consistency as well. For sequential consistency, rule (2) is replaced by:

(2a) There exists a sequential ordering of the sessions such that (i) for every read in a session, there exists a matching write prior to it, and (ii) reads are causally ordered on the same client.

For eventual consistency, rule (2) above is replaced by:

[2]In FUSE, the equivalent of a *close* operation is the *release* operation.

(2b) If at some point in time no more changes occurs, then eventually all sessions observe the same state of the file.

Figure 4.4(b) illustrates the combination of close-to-open semantics with linearizability, sequential, and eventual consistency. Obviously, if a single read or write operation is executed per session, the consistency level per operation defines how sessions behave. In other words, close-to-open semantics have no effect (e.g., middle row in Figure 4.4(b)). Now, when a client executes multiple operations or opens multiple files at the same time, the file system is neither linearizable nor sequentially consistent. For instance, execution depicted at the top row in Figure 4.4(b) is admissible. Under close-to-open semantics, linearizability is stricter than sequential consistency, which is itself stricter than eventual consistency (last two rows in Figure 4.4(b)).

Consistency of the File System

A FSCL is obtained by the combination of a consistency level governing file operations and the use or not of the close-to-open semantics. This leads to six *different* FSCLs. In Figure 4.4, we list in the last column one or more matching implementations for each level.

POSIX semantics is obtained when file operations are atomic and close-to-open is not supported. To implement sequential consistency, Sprite [96] relies on a cache consistency manager while GoogleFS [51] and HDFS [45] make use of a leasing mechanism. NFS [118] implements the consistency level offered in Andrew File System [61]: the close-to-open semantics is respected and metadata operations are atomic. Sinfonia [4] supports mini-transactions, a generalized form of compare-and-swap operation. To advocate for this paradigm, the authors of Sinfonia built a file system: SinfoniaFS. This file system implements sequential consistency with close-to-open semantics. Ivy [93], and Pastis [16] implement an eventually consistent DFSS, respectively, with and without close-to-open semantics. Busca et al. also present in [16] a version of Pastis that supports read-your-write semantics.

4.5 Algorithms

In this section we present the different algorithms used by FlexiFS. FUSE operations are performed on the client, while the distributed storage layer API and its three consistency models are handled by the key/value store.

FUSE API with Close-to-Open (CTO) semantics

We present in this and the following subsection the algorithms followed by the FlexiFS client when the system calls a FUSE operation (see Section 4.2). Only the relevant operations and arguments are presented. For some operations, the client performs a different algorithm depending on whether it follows or not CTO semantics. This subsection presents the case when using CTO. The next subsection describes the differences in the algorithms when considering atomic read/write operations instead of CTO semantics.

`create`(*filename*, *mode*)**:** Upon the creation of a file, the client builds a corresponding `iblock` and sends it to the distributed storage. The argument *mode* is copied into the corresponding field in the metadata. The client executes then a CAS on the parent directory to add the file. Performing a CAS operation ensures that two clients cannot create the same file concurrently. If the file was concurrently created, the operation returns -1. If the CAS operation is successfully executed, the file is left open for subsequent operations (i.e., the `iblock` is kept in memory, with an *open_sessions* counter set to 1) and the function returns.

`mkdir`(*filename*)**:** A "make directory" call is similar to a `create` operation, but in this case, an `iblock` for directory is created. The directory is not left open.

`getattr`(*filename*)**:** The client requests the `iblock` that corresponds to the file name given in the arguments. Then, it extracts a set of standard attributes from the `iblock` and returns them. This attributes are described in Section 4.2.

`open`(*filename*)**:** To open an existing file, the client follows the structure of the file system and invokes the `get`() operation to retrieve the corresponding `iblock` from the storage interface. Once the `iblock` is fetched, the client checks that permissions are correct. If so, the file is left open for further operations, the *open_sessions* counter is incremented by 1, and the function returns.

`read`(*filename*, *iblock*, *size*, *offset*)**:** A `read` operation is always preceded by a `open` or `create` operation. Since the `iblock` is kept in memory and provided as an argument, the client also knows all the `dblocks` attached to it. To retrieve the content of the file, the client fetches the required

`dblocks` from the storage system by invoking `get`() operations in parallel. The arguments *size* and *offset* help to calculate which `iblocks` are going to be retrieved. The client then concatenates all these streams and returns a single string of bytes.

`write`(*filename*, *iblock*, *buffer*, *offset*)**:** The client first produces the new `dblocks`. With the help of *offset*, it calculates which blocks will be overwritten. The client leverages the `put`() operation to insert (in parallel) the new `dblocks` in the distributed storage. Notice that because `dblocks` are content-addressed and immutable, every modification that produces an existing `dblock` leads to the creation of a new `dblock` with a different key. Then, the client updates the copy of the `iblock` that resides in memory and returns the size of the written *buffer*.

`flush`(*filename*, *iblock*)**:** The client sends the updated `iblock` to the distributed storage system.

`release`(*filename*, *iblock*)**:** The client decrements the number of open sessions in the `iblock` structure. If this counter reaches 0, the client proceeds to close the file; it sends an updated version of the `iblock` to the distributed storage.

`rename`(*from*, *to*)**:** The client first retrieves the `iblocks` of the *from* and *to* parent directories. If the directories are the same, the client attempts to update the `iblock` of the parent directory. In case they are different, the client first tries adding the file to the target directory, then it attempts removing the file from the source directory. To make atomic modifications, the client retrieves an `iblock` with a `get`(), modifies it locally, and attempts to update it with a CAS.

If any of the CAS operations fails, the operation returns an error. Even when CAS is atomic, the whole operation is not atomic; the renamed file might end up in both source and target directories. Renaming shall be strictly atomic in POSIX semantics. We note however that such behavior is admissible in certain systems (e.g., Win32).

`link`(*from*, *to*)**:** The client retrieves and modifies the `iblock` representing the parent directory of the new file (*to*), in order to add this entry. I also retrieves the `iblock` that is pointed by *from*, and increments the field

`nlinks` (number of links) in it. Then it updates the `iblock` with a CAS. Just like `rename`, this operation is not strictly atomic.

`unlink`(*filename*)**:** The client retrieves the `iblocks` that correspond to *filename* and to its parent directory. It removes the entry from the parent directory and updates it with a CAS. Then, it decrements the number of links on the `iblock`; if this number reaches 0, it removes the `iblock` too (and optionally also the `dblocks`).

`symlink`(*from*, *to*)**:** The client creates an `iblock` and writes the string *from* as the only element of the `iblock`, apart from the standard metadata. The `iblock` has a special mode that indicates that it is a *soft link*. The client puts the `iblock` on the distributed storage, and then, it attempts then to update the parent directory with a CAS.

When the FlexiFS client implements the CTO semantics, it keeps track of the open files. In other words, upon a successful call to `open`(f) the client records the `iblock` of f. This `iblock` is used for all the operations during a file session: a `read`() operation accesses the `dblocks` indexed by the `iblock`, and a write operation changes only the cached version. When the client closes file f, it stores the `iblock` of f using `put`(). It can then remove f from memory.

FUSE API without CTO semantics

CTO semantics help to improve the efficiency of a file system by reducing the number of accesses to the storage device(s) (synchronization with storage occurs only when flushing or releasing files). In the specific case of a DFSS, it reduces the the number of `put`() and `get`() requests sent to the distributed storage system. However, multi-user file systems that support concurrent read-/write access from different client nodes, will likely require atomic operations, in order to provide POSIX-like guarantees (see Section 4.4).

The algorithms described in the previous subsection change when atomic reads and writes are assumed. We depict here theses changes:

- The operations `read` and `write` discard the `iblock` that comes in the arguments (which was retrieved during a previous `open` instruction). These operations retrieve the `iblock` from the distributed storage system with a `get`().

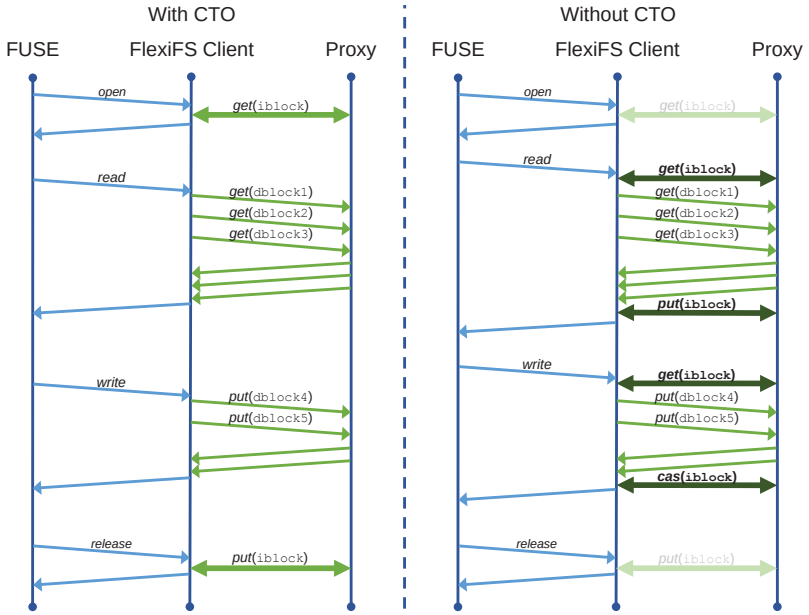


Figure 4.5: CTO semantics vs. atomic operations *(The dark arrows represent added messages, while the light arrows represent messages that are no longer needed).*

- `read` and `write` operations update the corresponding `iblock` at the end of their execution with the use of a `put`() and a `cas`(), respectively.
- The operations `open` and `release` are not necessary anymore, and become empty functions.
- The *open_sessions* counter is not necessary anymore, since there is no longer the concept of sessions.

Another way of seeing these differences is picturing that every `read` and `write` operation are implicitly "enclosed" by an `open` and `release` operation. These differences are shown in Figure 4.5. The dark arrows represent the messages that are added to atomic read/write operations, w.r.t. CTO semantics. The light arrows are the messages that are not present when CTO semantics are not used, because they are no longer needed.

When writing the `iblock` at the end of a `write` operation, the client uses CAS to update the `iblock` corresponding to the file. If the `iblock` changed meanwhile, the client has to recompute (if necessary) the `dblocks`, as well as an updated version of the `iblock`; then it re-executes CAS. This last sequence

of operations is executed until the CAS operation succeeds. Since a `write` operation may access any offset of the file, the above mechanism is necessary to avoid a lost update phenomena when two clients concurrently write the file.

A `read` operation also changes the `iblock`, because it modifies the field `atime`. However, at the end of a `read` operation, there is a `put`() instead of a `cas`(), because `atime` is always overwritten without taking into account the previous value.

FSCLs and the distributed storage layer API

In Section 4.4 we discussed the 6 different FSCLs that are supported by our design, which are the result of 3 different consistency levels in the distributed storage layer, combined with the use or not of CTO semantics. We discussed in the two previous subsections the technical details regarding the use of CTO. Since CTO semantics are inherent to file handling, the differences between the two approaches (*with* or *without* CTO) are present only on the client's algorithms.

On the other hand, the differences between the 3 consistency levels supported by FlexiFS are present on the Storage Node's algorithms. We explain below the algorithms for each of the consistency levels:

Linearizability: As explained in Section 4.4, a common way to achieve linearizability is through consensus, with the use of Paxos. On top of such consensus, we implement for FlexiFS a replicated state machine executing the four operations listed in Section 4.3.

Sequential consistency: For this consistency level, we use primary replication. All `put`(k, v) calls are processed by the primary replica. To execute `cas`(k, u, v), the primary replica tests locally if the old value equals u. If it is the case, it executes a `put`(k, v) and returns the old value to the Proxy. To execute a `get`(k) call, the Proxy accesses any replica of k that contains the version it previously read, or a newer version.

Eventual consistency: We implement eventual consistency in FlexiFS using vector clocks (see Section 4.2) and the "last writer wins" approach [108]. This optimistic replication schema works as follows: Each version of a $< k, v >$ record is timestamped with a vector clock. Upon updating the value of a record (via `put`() or `cas`()), the Proxy contacts one of the

replicas responsible for this record. This replica atomically increments its local vector clock, timestamps the record with it, and returns to the Proxy. Replicas then converge using an anti-entropy protocol. If two versions of some record are concurrent, we apply the "last writer wins" approach. Concurrent operations are totally ordered according to the identifier of the replicas that emitted them. Upon a `get`() operation, the Proxy simply returns a version stored at any of the replicas.

Also, these algorithms apply only when accessing `iblocks`. All `dblocks` are immutable, and thus, they are trivially linearizable. To improve performance, access to `dblocks`, in any of the 3 consistency levels, follows a simpler algorithm: `put`() and `get`() operations access a majority of replicas, respectively storing and fetching the content from it.

4.6 Implementation and Deployment

FlexiFS' implementation is built upon SPLAY [82], and it constitutes at the same time an enhancement to the platform's features. The implementation is separated in two parts: the Storage Node's code, and the client's code.

The Storage Nodes

The Storage Node's code is implemented as two SPLAY libraries, that can be loaded by any job running on the platform. The libraries are `paxos.lua`, which spans around 200 Source Lines of Code (SLOC), and `distdb.lua` (around 1000 SLOC).

The `paxos.lua` library contains the functions `paxos_read` and `paxos_write`. These functions allow operating a Paxos run between several nodes.

The function `paxos_read` receives the following arguments:

- An array of nodes, each node described as pair $< IP\_address, port >$.
- A unique and consistently increasing proposal ID.
- Number of retries. When this number of retries is depleted, the function stops and returns on error.

The function `paxos_write` receives the same arguments as `paxos_read`, in addition to the `value` to be proposed.

Both functions perform Remote Procedure Calls (RPCs) over UDP to all the nodes in the array and follow the Basic Paxos algorithm. The `read` function returns the latest stored value (agreed on consensus). For the `write` function, the proposed value is stored if consensus is reached (i.e., if the majority of nodes commit to store the new value). If the proposal does not get enough quorum before a given timeout, or if any node indicates that it has already committed to a bigger proposal ID, the process is repeated. Timeouts are attributes of the library calls, and can be tuned to fit the developer's needs.

The library can be used independently of FlexiFS, e.g., as base for an agreement protocol [15, 63].

The `distdb.lua` library contains the complete functionality of a Storage Node, including the functionality related to the Proxy role: a lightweight HTTP server to communicate with clients through a RESTful API, and the forwarding functions for the DHT API.

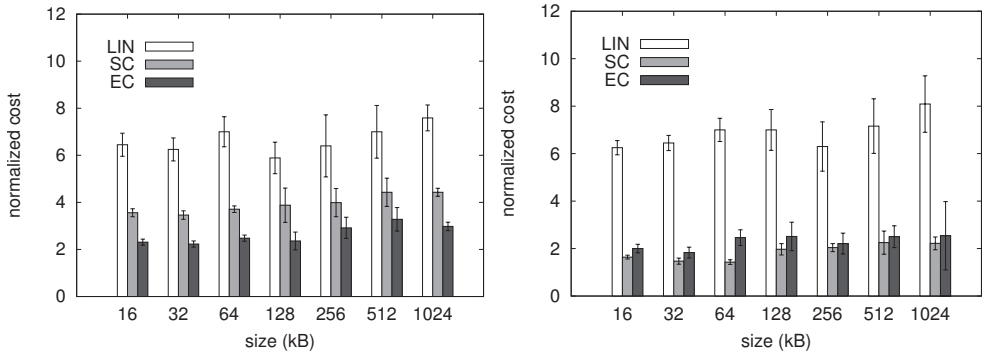
The client

The client's code is a set of Lua scripts that are loaded in the client premises, independently of the execution of SPLAY. It does not use any of the SPLAY libraries. It leverages instead the FUSE C library and a Lua binding to this library from the *luafuse* project [85].

The client side is composed of two files: distdb-client.lua (163 SLOC) and flexifs-client.lua (621 SLOC). The former contains functions that communicate with the Proxy's RESTful API, and the latter contains all the FUSE logic.

4.7 Evaluation

In this section, we present experimental results obtained using FlexiFS, where we observe empirically and in isolation the trade-offs between sharing semantics and performance in DFSS designs.



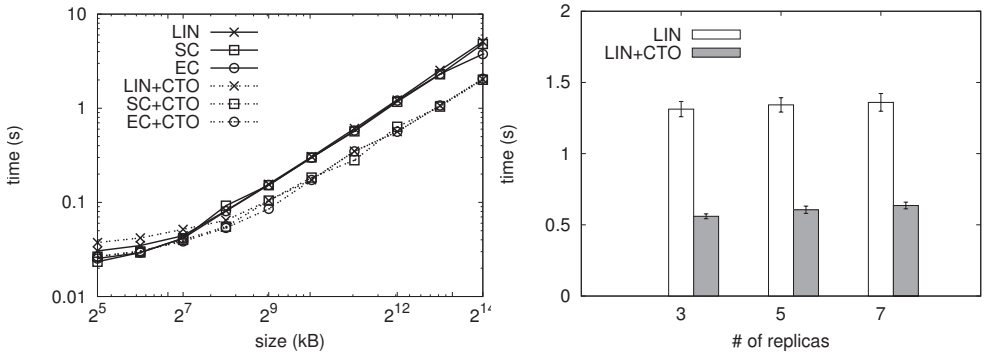
(a) `iblock` creation. (b) `iblock` retrieval.

Figure 4.6: Evaluating consistency at the key/value store level *(LIN, SC and EC stand for Linearizable, Strong Consistency and Eventual Consistency models, respectively).*

Experimental Settings

All tests were performed on a cluster of 8-core virtualized Xeon 2.5 Ghz servers running Ubuntu 12.04 GNU/Linux and connected by a 1 Gbps switched network. We use 3 to 7 servers for the storage layer and one client. Our implementation uses the Lua [65] language and leverages the SPLAY [82] framework and libraries. Bindings to the FUSE C API employ the `luafuse` library (`http://code.google.com/p/luafuse/`). The FlexiFS implementation is modular and easy to modify. The conciseness of Lua and the use of SPLAY allow the whole implementation to be less then 2,000 lines of code (LOC). In particular, the code to support each FSCL is very concise and easy to extend, e.g., 62 LOC for sequential consistency, and 160 LOC for eventual consistency.

In what follows, we explore the impact of each FSCL on the cost of `iblock` operations, and file operations. We also investigate the impact of the replication factor on performance. All experimental results are averaged over 10^3 operations, and we present standard deviations when appropriate. During our experiments, the size of a `dblock` was set to 128 kB.



(a) Varying the FSCL.

(b) Varying the Replication Factor.

Figure 4.7: Evaluating FSCLs and the Replication Factor by writing a file *(CTO stands for close-to-open semantics).*

Benchmarks

Metadata Operations. In Figure 4.6(a), we experiment the insertion of a novel `iblock` in the storage system when both the FSCL and the size of the inserted block vary. For the sake of comparison, results are normalized by the time required for executing a dummy RPC carrying a payload that equals the size of the block.

We observe in Figure 4.6(a) that eventual consistency is the cheapest FSCL as it costs around 2 times more than the baseline RPC call. This is expected since a call to CAS under eventual consistency requires 2 roundtrips: one to go from the client to the Proxy, then one to go from the Proxy to a replica. Sequential consistency costs 4 roundtrips: Once the primary is reached, the update must reach a quorum of replicas. For linearizability, 2 more roundtrips for the "propose" phase of the Paxos algorithm are executed, leading to 6 times the baseline cost.

Our second experiment evaluates the cost of fetching an `iblock` from the file storage. We report the results in Figure 4.6(b). Under linearizability, a `get`() operation has an identical cost to a CAS operation, since both operations go through the replicated state machine. On the other hand, the cost of sequential consistency is reduced because the Proxy can access any replica to fetch the `iblock` content. Therefore, performance is in that case identical to eventual consistency.

File Operations. Figure 4.7(a) depicts the time required to write a complete file at the client side using the FUSE interface. Both scales in this figure are logarithmic. Eventual consistency is the fastest FSCL when either *(i)* close-to-open semantics is not used, or *(ii)* there is a single `dblock` to write, i.e., less than 128 kB in our settings. The POSIX semantics of sharing (i.e., linearizability without close-to-open semantics) performs the worst in this respect. When the size of the file reaches 128 kB, the use of close-to-open semantics leads to better performance (between 2 and 3 times faster). Below 128 kB, close-to-open semantics pays the cost of the necessary *open*() and *close*() operations. All FSCLs offering close-to-open semantics reach similar performance when more than 4 MB of data are written. Read operations over a file (not reported here) follow a similar pattern.

Impact of the Replication Factor. Our last experiment measures the impact of the replication factor on performance. In this experiment, a client writes a file of 4 MB under linearizability. We vary both the replication factor of FlexiFS, and the use or not of the close-to-open semantics. Figure 4.7(b) depicts our results. In this figure, we observe that increasing the replication factor has a small impact on performance: below 3% without close-to-open semantics, and 7% with. Paxos is the most demanding consistency control algorithm we have implemented. Thus, this result shows that the FSCL is contributing more than the replication factor to the DFSS performance.

4.8 Related work

Several papers discuss the performance, consistency, and semantics trade-offs in DFSS designs. The Andrew file system (AFS) [61] introduced caching mechanisms and the close-to-open semantics for both files and directories. This was inspired by earlier designs such as LOCUS [126], which relied on a strict—but costly and inefficient—POSIX semantics. Since its second version, the Network File System (NFS) [118] also implements the close-to-open semantics; its fourth version distinguishes data from metadata management.

OceanStore [73] is a flat data storage system that provides both eventually consistent and atomic operations. OceanStore follows a design close to the eventually serializable data storage [41]. In OceanStore, an application may emit two types of file operations: a *weak* operation is tentative and executes on any replica; a *strong* operation waits until replicas agree on some total

ordering of the operations.

GoogleFS [51] uses a central server for storing metadata. CFS [31] builds a single-user file system by storing content-addressable blocks in the Chord DHT [115]. Ivy [93] extends this design by allowing a predefined group of users to access a shared file system. Content blocks are stored in the Chord DHT and each writer maintains its own modification log, implementing read-your-write semantics and eventual consistency.

Stamatakis et al. [114] propose to build centralized metadata storage services for DFSS, providing linearizability guarantees using the Paxos [77] consensus algorithm while maintaining high availability.

The authors of Pastis [16] compare the close-to-open against the read-your-write semantics. Levy [83] surveys DFS designs and four different types of file sharing semantics: POSIX, close-to-open, immutable files, fully transactional semantics, and survey corresponding implementations. The use of a modular framework for evaluating design choices and establishing performance/trade-offs in systems software design has been successfully used in various domains. Examples include virtual machines construction [50] or CORBA-based Middleware [101].

4.9 Summary

This chapter depicted a study of the impact of the FSCL on the performance of a DFSS. While the FSCL offered by a DFSS has fundamental impact on performance, it is difficult to systematically evaluate this impact in isolation from other design aspects, due to the design and implementation diversity of existing systems. In this chapter, we have presented FlexiFS, a framework for the systematic evaluation of DFSS aspects. In more details, we have depicted a file system interface to users and leverage a set of servers implementing a fully distributed storage layer for both data and metadata. We implemented three forms of consistency: linearizability, sequential consistency and eventual consistency, together with and without close-to-open semantics. Remarkably, a DFSS providing all these FSCL can be supported with the simple addition of a *compare-and-swap* primitive to a regular key/value store.

Our experimental results establish that linearizability under the close-to-open semantics is a sound design choice and a good compromise between operational semantics and performance, while illustrating the trade-offs offered by the other design options.

Chapter 5

Conclusions

We have shown in the introduction that computing is nowadays advancing towards new trends:

- There is an increasing tendency to virtualization and resource sharing (CPU power, memory, storage).
- Large-scale internet services tend to scale out instead of scaling up.
- More and more small devices (e.g., sensors) are connected to the internet and provide big amounts of new data.

These trends lead many concerns about privacy, specifically anonymity, and trust, which become important topics more than ever. A big part of this thesis is dedicated to mitigate issues about anonymity and trust. The other important aspect covered by this thesis is the creation of tools for the evaluation of new paradigms in large-scale distributed storage.

To conclude, we sum our contributions in Section 5.1, followed by Section 5.2, where we present scenarios that can combine the three systems. Finally, we present in Section 5.3 several perspectives that our work opens to the research community.

5.1 Contributions

During the realization of this thesis, we have accomplished to create three systems, which solve critical problems encountered on our study of Large-Scale Distributed Storage Systems (LS-DSSs).

First we focused on LS-DSSs and how they can guarantee *publisher anonymization* and *rate limitation* at the same time. We designed and implemented SPADS: Publisher Anonymization for DHT Storage [43], which deals with both aspects. Publisher anonymity is guaranteed by the use of Chaum mixes, while rate limitation is enforced by a credential management protocol. We evaluated the system using a real implementation, and we observed that it is applicable.

Then, we focused on Large-Scale Distributed Aggregation Systems (LS-DASs) and how they can *detect* suspicious behavior from malevolent nodes. We designed and implemented Collaborative Auditing for Distributed Aggregation (CADA) [120], which consists in two decentralized and lightweight *oracles*, which detect servers that attempt to bias the aggregation. The oracles perform a probabilistic auditing, that raises suspicions based on the statistical deviation from an expected behavior. CADA is intended to provide an input for an external mechanism (e.g., blacklist management service, trust management layer).

Finally, we focused on Distributed File Storage Services (DFSSs) and the lack of a fair benchmarking platform for important settings like the consistency model or the replication factor. We designed and implemented FlexiFS [121], a testbed DFSS that allows a clean comparison of consistency models and their impact on the system's performance.

FlexiFS allows an end-user to mount a Linux file system with the help of File System in User Space (FUSE), on top of a key/value store that offers a traditional Application Programming Interface (API) with the addition of the *compare-and-swap* primitive. FlexiFS is modular, and leverages several state-of-the-art mechanisms for data distribution, replication, routing, and indexing. We benchmarked the six different File System Consistency Levels (FSCLs) offered by the system and concluded that linearizability and the use of close-to-open semantics is a good combination and a gives a good compromise between operational semantics and performance. We also exposed the trade-offs offered by the other design options.

5.2 Integration of the three systems

In this section, we show scenarios were we can combine the three systems developed during this thesis (SPADS, CADA and FlexiFS).

The first scenario is mentioned as a key motivation of the thesis (Sec-

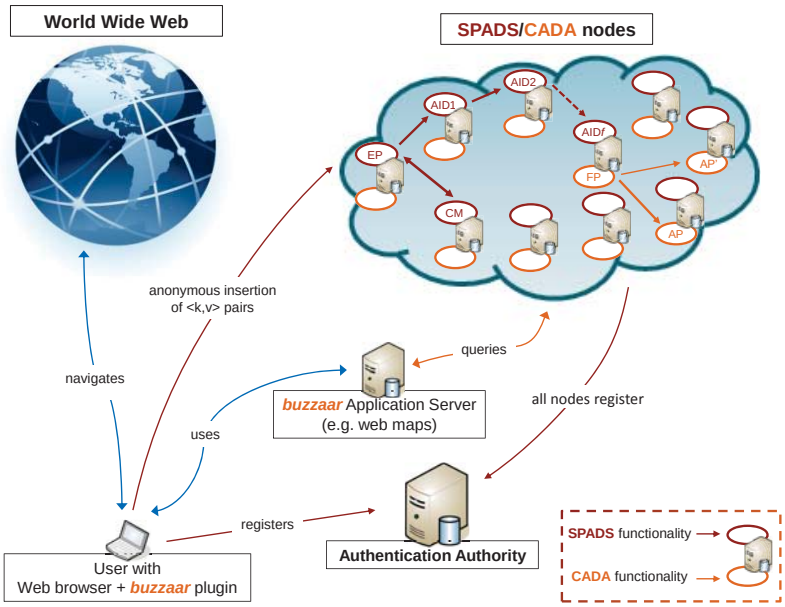


Figure 5.1: Buzzaar general architecture with SPADS for *anonymization and rate limitation* and CADA for *collaborative auditing*.

tion 1.2). The Buzzaar project is an LS-DAS built upon a multi-domain network, which is based on several smaller networks from different cloud computing providers.

The "multi-domain" condition puts constrains on trust, such as the impossibility of trusting a single server of not biasing aggregations or not wanting to disclose identifiable information of the clients.

The second constraint raises the need to provide system-wide guarantees that the information will remain anonymous, but this poses the threat that malicious clients will want to bias the aggregations and take advantage of their guaranteed anonymity.

Finally, in an LS-DAS, we want to avoid as much as possible the presence of centralized entities that diminish the scalability of the network.

So, we want for Buzzaar, a system or combination of systems that provide:

- publisher anonymization,
- rate-limitation, and
- collaborative auditing.

By mixing the mechanisms provided by SPADS and CADA, we can provide such features to Buzzaar and similar LS-DASs. Figure 5.1 shows a high level picture of Buzzaar, functioning with the two aforementioned systems.

An important use of such systems can be in privacy-preserving social networks, which might want to perform data mining on their clients contents, while guaranteeing their anonymity.

Another possible scenario is a DFSS that wants to provide several FSCLs according to different Service-Level Agreements (SLAs) established with its clients. This DFSS wants to provide system-wide anonymity guarantees, but at the same time, wants to limit the update rate and/or the storage quota of its clients. Data in `dblocks` can be encrypted or not by the client before storage, with a secret symmetric key that only the client knows.

In order to provide a full set of anonymous operations, we must provide also an *anonymous get*. Such operation can simply use the onion routing scheme described in Section 2.2, where a return path is encoded along the message.

For such system, we want to provide:

- publisher anonymization.
- rate-limitation.
- configurable FSCL.

A combination of SPADS and FlexiFS provides the desired features. Notice that the use of FlexiFS, with its simple algorithms for failure-handling and Key-Based Routing (KBR), is oriented rather to prototyping or the development of proof-of-concept systems. This is shown in Figure 5.2.

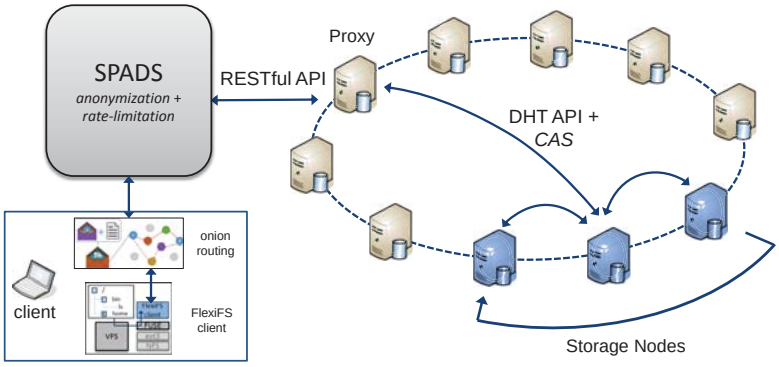


Figure 5.2: Architecture of a DFSS with SPADS for *anonymization and rate limitation* and FlexiFS for *tunable FSCL*.

Integration of the three systems Finally, we can picture the above scenario, and add the possibility of performing data mining tasks over the data that clients store. Examples of data mining can be the extraction of keywords out of file names, text on documents. Aggregations that result from data mining and gathering of statistics can be stored in an LS-DAS, that can be based on a second distributed storage layer, or leverage the same as the DFSS.

Data mining can be performed on the server side, over each block that is handled. For this case, data must be stored without encryption. If data mining is done rather on the client side, this restriction is no longer necessary.

Assuming that the LS-DAS requires protection of its aggregations from byzantine behaviors, and, as a large-scale system, it is not desirable to depend on a central entity that handles all the auditing, we can leverage CADA to provide collaborative auditing.

Figure 5.3 shows the integration of the three systems, with a common SPADS layer, and then FlexiFS and CADA, for storage of the client's data and statistics on the client's data, respectively.

5.3 Perspectives

Our work on the three systems presented in Section 5.1 leads to several perspectives, that we discuss in this section.

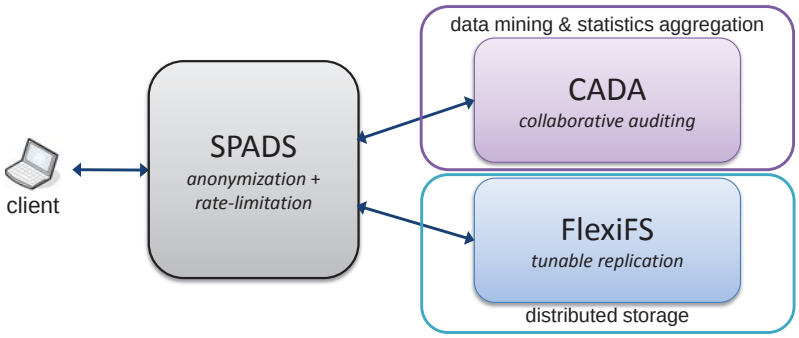


Figure 5.3: Integration of the three systems.

Many perspectives are opened by the development of SPADS. First, it would be interesting to study the possibility of replacing the centralized authentication authority with a decentralized implementation of the same service. Trust-based network could be a sound approach for this, but the mechanisms shall be made resilient to Sybil attacks [37], which is not an easy task. Second, even if rate limitation greatly reduces the impact of cheating peers, it does not totally prevent them from sending a small amount of fake data to the system. It would be interesting to investigate the possibility for servers to collaboratively blacklist cheaters without compromising anonymization. Note that, similarly to rate limitation *vs.* publisher anonymization, these two objectives appear to be contradictory and supporting both is a challenging problem.

Our work on CADA oracles opens interesting perspectives. First, the oracles can automatically purge the information originated by misbehaving servers (as identified by their number of suspicions). Since the distribution of elements coming from servers that are deemed safe follows the same trend, the distribution with no bias can be built by interpolating the distributions from safe servers (or from shadow servers in the case of aggregation biasing). This results in an additional disincentive for servers to bias the aggregation: their contributions may simply be ignored silently.

Another future work on the topic could be to focus on relaxing the *entry points uniform distribution* hypothesis, allowing the insertion bias oracle to support any load distribution at the Forwarding Point (FP) level, other than the uniform distribution. This approach is feasible only if the distribution of FPs contributions is known in advance; a way to achieve that is to collectively

build a model of the distribution, based on observations made by other servers in the aggregation layer, and provide it as an input to the insertion bias oracle. It would also be desirable to consider the pairing of the oracles with a reputation management mechanism, in which the signed suspicions reports by the servers are used to build the reputation of servers and enforce trust metrics.

Finally, our work on FlexiFS opens perspectives in the field of DFSS design. We plan on investigating further aspects of DFSS pertaining to indexing, client interaction, and semantics. FlexiFS offers currently six different flavors of FSCLs, but many more can be added, e.g., by implementing more replication schemes on the key/value store. Further experimentation regarding the benchmark of FSCLs can be done, by performing tests with concurrent clients and with servers leaving and entering the network (churn). We can, in fact, leverage SPLAY's churn manager to simulate complex churn schemes. We also plan to release FlexiFS as a part of the open-source SPLAY framework [82].

Bibliography

[1] A virtual file system (avfs) sourceforge page. http://sourceforge.net/projects/avf/.

[2] Carlisle Adams and Steve Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[3] Michael Agger. Google's evil eye. does the big g know too much about us? http://www.slate.com/articles/technology/the_browser/2007/10/googles_evil_eye.html, October 2007.

[4] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *21st ACM Symposium on Operating Systems Principles (SOSP'07)*, 2007.

[5] Lorenzo Alvisi, Dahlia Malkhi, Evelyn Pierce, and Michael K. Reiter. Fault detection for byzantine quorum systems. *IEEE Transactions on Parallel and Distributed Systems*, 12:996–1007, September 2001.

[6] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/.

[7] Apache couchDB. http://couchdb.apache.org/.

[8] Apache HBase. http://hbase.apache.org/.

[9] Micheal Barbaro and Tom Seller, Jr. A face is exposed for AOL searcher No. 4417749. http://select.nytimes.com/gst/abstract.html?res=F10612FC345B0C7A8CDDA10894DE404482, August 2006.

[10] Adam Bender, Rob Sherwood, Derek Monner, Nate Goergen, Neil Spring, and Bobby Bhattacharjee. Fighting spam with the neighborhoodwatch DHT. In *28th IEEE International Conference on Computer Communications (INFOCOM'09)*, pages 1755–1763, April 2009.

[11] Stephen Bernstein and Ruth Bernstein. *Elements of statistics 2: Inferential Statistics*. McGraw-Hill Professional, 1st edition, 1999.

[12] Kenneth P. Birman. Replication and fault-tolerance in the ISIS system. In *10th ACM Symposium on Operating Systems Principles (SOSP'85)*, pages 79–86, 1985.

[13] George E. P. Box, J. Stuart Hunter, and William G. Hunter. *Statistics for Experimenters: Design, Innovation, and Discovery*. Wiley, 2nd edition, 2005.

[14] Eric Brewer. CAP twelve years later: How the "rules" have changed. `http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed`, 2012.

[15] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 335–350. USENIX, 2006.

[16] Jean-Michel Busca, Fabio Picconi, and Pierre Sens. Pastis: a highly-scalable multi-user peer-to-peer file system. In *11th Euro-Par 2005 - Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science (LNCS)*, pages 1173–1182, August 2005.

[17] Min Cai and Martin Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *13th International Conference on World Wide Web (WWW'04)*, New York, NY, USA, 2004.

[18] Min Cai, Martin Frank, Jinbo Chen, and Pedro Szekely. MAAN: A multi-attribute addressable network for grid information services. In *4th International Workshop on Grid Computing (GRID'03)*, Washington, D.C., USA, 2003.

[19] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 173–186, 1999.

[20] Tushar Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43:225–267, March 1996.

[21] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 15–15. USENIX, 2006.

[22] Suw Charman-Anderson. Could Diaspora ever challenge Facebook and Google Plus? `http://www.firstpost.com/tech/could-diaspora-ever-challenge-facebook-and-google-plus-130802.html`, November 2011.

[23] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24:84–90, February 1981.

[24] Ian Clarke, Theodore W. Hong, Scott G. Miller, Oskar Sandberg, and Brandon Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, 6:40–49, January 2002.

[25] Roger Clarke. Identified, anonymous and pseudonymous transactions: The spectrum of choice. `http://www.rogerclarke.com/DV/UIPP99.html`, April 1999.

[26] Roger Clarke. What's 'privacy'? `http://www.rogerclarke.com/DV/Privacy.html`, July 2006.

[27] Adam Cohen. Will we ever get strong internet privacy rules? `http://ideas.time.com/2012/03/05/will-we-ever-get-strong-internet-privacy-rules/`, March 2012.

[28] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel

Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *VLDB Endowment*, 1:1277–1288, August 2008.

[29] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, pages 251–264, 2012.

[30] Landon P. Cox and Brian D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *19th ACM Symposium on Operating Systems Principles (SOSP'03)*, New York, NY, USA, 2003.

[31] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 202–215, New York, NY, USA, 2001. ACM.

[32] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common API for structured peer-to-peer overlays. In *2nd International workshop on Peer-to-Peer Systems (IPTPS'03)*, February 2003.

[33] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Operating Systems Review*, 41:205–220, December 2007.

[34] Deezer website. `http://www.deezer.com/`.

[35] Roger Dingledine, Michael J. Freedman, and David Molnar. The free haven project: distributed anonymous storage service. In *International workshop on Designing privacy enhancing technologies*, pages 67–95, 2001.

[36] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *13th USENIX Security Symposium (Security'04)*, San Diego, CA, USA, 2004.

[37] John R. Douceur. The sybil attack. In *Revised papers from the 1st International workshop on Peer-To-Peer Systems (IPTPS'01)*, pages 251–260, 2002.

[38] Peter Druschel and Antony Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *8th Workshop on Hot Topics in Operating Systems (HotOS'01)*, pages 75–80, May 2001.

[39] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35:288–323, April 1988.

[40] Richard Esguerra. A handy Facebook-to-English translator. `http://www.eff.org/deeplinks/2010/04/handy-facebook-english-translator`, April 2010.

[41] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220:113–156, June 1999.

[42] Pascal Felber, Peter Kropf, Lorenzo Leonini, Toan Luu, Martin Rajman, and Etienne Rivière. Collaborative ranking and profiling: Exploiting the wisdom of crowds in tailored web search. In *10th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'10)*, volume 6115 of *Lecture Notes in Computer Science (LNCS)*, pages 226–242, Amsterdam, The Netherlands, June 2010.

[43] Pascal Felber, Martin Rajman, Etienne Rivière, Valerio Schiavoni, and José Valerio. SPADS: Publisher anonymization for DHT storage. In *10th IEEE International Conference on Peer-to-Peer Computing (P2P'10)*, August 2010.

[44] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference (ACSC'88)*, pages 56–66, February 1988.

[45] The Apache Software Foundation. The hadoop distributed file system. `http://hadoop.apache.org`.

[46] Michael J. Freedman and Robert Morris. Tarzan: a peer-to-peer anonymizing network layer. In *9th ACM Conference on Computer and Communications Security (CCS'02)*, pages 193–206, 2002.

[47] Friendica website. `http://friendica.com/`, August 2006.

[48] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52:139–149, February 2003.

[49] John Gantz and David Reinsel. Extracting value from chaos. `http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf`, June 2011.

[50] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. VMKit: a substrate for managed runtime environments. In *VEE*, 2010.

[51] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003.

[52] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[53] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: practical accountability for distributed systems. In *21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Stevenson, WA, USA, 2007.

[54] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computer Survey*, 15:287–317, December 1983.

[55] Katie Hafner. Researchers yearn to use AOL logs, but they hesitate. `http://www.nytimes.com/2006/08/23/technology/23search.html`, August 2006.

[56] Terry Hancock. Why you should join Diaspora now, like your freedom depends on it. `http://www.freesoftwaremagazine.com/columms/why_you_should_join_diaspora_now_your_freedom_depends_it`, September 2011.

[57] Maurice Herlihy and Jeannette Wing. Linearizability: a correcteness condition for concurrent objects. *ACM Trans. Prog. Lang. and Sys.*, 12, July 1990.

[58] Martin Hilbert and Priscila López. The world's technological capacity to store, communicate, and compute information. *Science*, 332:60–65, April 2011.

[59] Mark D. Hill. What is scalability? *SIGARCH Computer Architecture News*, 18:18–21, December 1990.

[60] Harrison Hoffman. Facebook's source code goes public. `http://news.cnet.com/8301-10784_3-9758702-7.html`, August 2007.

[61] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6, February 1988.

[62] Ling Huang, Ben Zhao, Anthony D. Joseph, and John D. Kubiatowicz. Probabilistic data aggregation in distributed networks. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2006.

[63] P. Hunt, M. Konar, F.P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX 2010 Annual Technical Conference (ATC'10)*, pages 145–158. USENIX, 2010.

[64] IEEE and The Open Group. Standard for information technology-portable operating system interface (POSIX) system interfaces, 2004.

[65] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of lua 5.0. *Journal of Universal Computer Science*, 11:1159–1176, 2005.

[66] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25, August 2007.

[67] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *29th ACM Symposium on Theory of Computing (STOC'97)*, 1997.

[68] Martin Kaste. Who are you, really? activists fight for pseudonyms. `http://www.npr.org/2011/09/28/140879480/who-are-you-really-activists-fight-for-pseudonyms`, September 2011.

[69] Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46:16–35, January 2003.

[70] M. Klonowski, M. Kutylowski, A. Lauks, and F. Zagorski. Universal re-encryption of signatures and controlling anonymous information flow. In *4th Central European Conference in Cryptology (WARTACRYPT'04)*, 2004.

[71] Masaki Kondo, Shoichi Saito, Kiyohisa Ishiguro, Hiroyuki Tanaka, and Hiroshi Matsuo. Bifrost: A novel anonymous communication system with DHT. In *10th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'09)*, pages 324–329, 2009.

[72] Gunnar Kreitz and Fredrick Niemelä. Spotify - large scale, low latency, P2P music-on-demand streaming. In *10th IEEE International Conference on Peer-to-Peer Computing (P2P'10)*, Delft, The Netherlands, 2010.

[73] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*, 2000.

[74] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review*, 44:35–40, April 2010.

[75] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The Vertica analytic database: C-store 7 years later. *VLDB Endowment*, 5:1790–1801, August 2012.

[76] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, 1997.

[77] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16:133–169, 1998.

[78] Leslie Lamport. Paxos made simple. *SIGACT News*, 32:51–58, December 2001.

[79] Leslie Lamport. Generalized consensus and Paxos. Technical report, March 2005.

[80] Leslie Lamport. Fast paxos. *Distributed Computing*, 19:79–103, 2006.

[81] Leslie Lamport and Mike Massa. Cheap Paxos. In *2004 International Conference on Dependable Systems and Networks (DSN'04)*, pages 307–314, 2004.

[82] Lorenzo Leonini, Etienne Rivière, and Pascal Felber. SPLAY: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *6th Symposium on Networked Systems Design and Implementation (NSDI'09)*, pages 185–198. USENIX, April 2009.

[83] Eliezer Levy and Abraham Silberschatz. Distributed file systems: concepts and examples. *ACM Comput. Surv.*, 22(4), December 1990.

[84] Xin Li, Fang Bian, Hui Zhang, Christophe Diot, Ramesh Govindan, Wei Hong, and Gianluca Iannaccone. MIND: A distributed multi-dimensional indexing system for network diagnosis. In *25th IEEE International Conference on Computer Communications (INFOCOM'06)*, Barcelona, Spain, 2006.

[85] luafuse: A simple lua binding for linux FUSE file system. `http://code.google.com/p/luafuse/`.

[86] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 120–131, October 1988.

[87] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Revised papers from the 1st International workshop on Peer-To-Peer Systems (IPTPS'01)*, pages 53–65, 2002.

[88] J. McLachlan, A. Tran, N. Hopper, and Y. Kim. Scalable onion routing with Torsk. In *16th ACM Conference on Computer and Communications Security (CCS'09)*, pages 590–599, 2009.

[89] Peter Mell and Tim Grance. The NIST definition of cloud computing. Technical report, July 2009.

[90] Sebastian Michel. *Top-k Aggregation Queries in Large-Scale Distributed Systems*. PhD thesis, Max-Planck-Institut für Informatik, Saarbrüken, Germany, 2007.

[91] Alan Mislove, Gaurav Oberoi, Ansley Post, Charles Reis, Peter Druschel, and Dan S. Wallach. AP3: cooperative, decentralized anonymous communication. In *11th ACM SIGOPS European Workshop (EW11)*, page 30, 2004.

[92] MongoDB. `http://www.mongodb.org/`.

[93] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: a read/write peer-to-peer file system. In *OSDI*, 2002.

[94] Animesh Nandi, Tsuen-Wan Ngan, Atul Singh, Peter Druschel, and Dan S. Wallach. Scrivener: Providing incentives in cooperative content distribution systems. In *6th ACM/IFIP/USENIX International Middleware Conference (Middleware'05)*, Grenoble, France, 2005.

[95] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. EDUTELLA: a P2P networking infrastructure based on RDF. In *11th International Conference on World Wide Web (WWW'02)*, Honolulu, HI, USA, 2002.

[96] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the sprite network file system. *ACM Trans. Comput. Syst.*, 6(1), February 1988.

[97] Netflix website. `http://www.netflix.com/`.

[98] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *USENIX Annual Technical Conference (ATEC'99)*, 1999.

[99] Pandora website. `http://www.pandora.com/`.

[100] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *9th ACM Symposium on Parallel Algorithms and Architectures (SPAA'97)*, pages 311–320, 1997.

[101] PolyORB Middleware Technology. `http://libre.adacore.com/tools/polyorb`.

[102] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: a network transparent, high reliability distributed system. In *8th ACM Symposium on Operating Systems Principles (SOSP'81)*, pages 169–177, 1981.

[103] Project Voldemort website. `http://www.project-voldemort.com/`.

[104] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *ACM SIGCOMM 2001 Annual Conference (SIGCOMM'01)*, pages 161–172, 2001.

[105] Sylvia Ratnasamy, Mark Handley, Richard M. Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In *3rd International COST264 Workshop on Networked Group Communication (NGC'01)*, pages 14–29, 2001.

[106] Antony Rowstron and Peter Druschel. Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *3th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01)*, Heidelberg, Germany, 2001.

[107] Antony I. T. Rowstron, Anne M. Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *3rd International Workshop on Networked Group Communications (NGC'01)*, 2001.

[108] Yasushi Saito and Marc Shapiro. Optimistic replication. *Computing Surveys*, 37, 2005.

[109] Jerome H. Saltzer and M. Frans Kaashoek. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann Publishers Inc., 2009.

[110] M. Satyanarayanan. A survey of distributed file systems. In *Annual Review of Computer Science*, 1989.

[111] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39:447–459, April 1990.

[112] Konstantin V. Shvachko. HDFS scalability: The limits to growth. *USENIX login*, 35, April 2010.

[113] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, 9:219–228, May 1983.

[114] Dimokritos Stamatakis, Nikos Tsikoudis, Ourania Smyrnaki, and Kostas Magoutis. Scalability of replicated metadata services in distributed file systems. In *DAIS*, 2012.

[115] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11:17–32, February 2003.

[116] Louise Story. Coke is holding off on sipping Facebook's Beacon. `http://bits.blogs.nytimes.com/2007/11/30/coke-is-holding-off-on-sipping-facebooks-beacon/`, November 2007.

[117] Louise Story. The evolution of Facebook's Beacon. `http://bits.blogs.nytimes.com/2007/11/29/the-evolution-of-facebooks-beacon/`, November 2007.

[118] Inc. Sun Microsystems. NFS: Network file system protocol specification. RFC 1094, Network Information Center, SRI International, March 1989.

[119] Uproot Nigeria: A social network alternative for occupy Nigeria. `http://news.infoshop.org/article.php?story=20120116052851518`, January 2012.

[120] José Valerio, Pascal Felber, Martin Rajman, and Etienne Rivière. CADA: Collaborative auditing for distributed aggregation. In *9th European Dependable Computing Conference (EDCC'12)*, Sibiu, Romania, May 2012.

[121] José Valerio, Pierre Sutra, Etienne Rivière, and Pascal Felber. Evaluating the price of consistency in distributed file storage services. In *13th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'13)*, 2013.

[122] Spyros Voulgaris, Daniela Gavidia, and Maarten Steen. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13:197–217, 2005.

[123] Florian M. Waas. Beyond conventional data warehousing - massively parallel data processing with Greenplum database. In *2nd Workshop on Business Intelligence for the Real-Time Enterprise (BIRTE'08)*, volume 27 of *Lecture Notes in Business Information Processing*, pages 89–96, 2009.

[124] Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: a robust, tamper-evident, censorship-resistant web publishing system. In *9th USENIX Security Symposium (Security'00)*, 2000.

[125] Michael Walfish, J. D. Zamfirescu, Hari Balakrishnan, David Karger, and Scott Shenker. Distributed quota enforcement for spam control. In *3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, pages 21–21, 2006.

[126] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *SOSP*, 1983.

[127] Xbox video. `http://www.xbox.com/en-US/video`.

[128] Aydan R. Yumerefendi and Jeffrey S. Chase. Strong accountability for network storage. *ACM Transactions on Storage*, 3:205–220, October 2007.

[129] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, 2004.

[130] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2001.

[131] Runfang Zhou and Kai Hwang. Powertrust: A robust and scalable reputation system for trusted peer-to-peer computing. *IEEE Transactions on Parallel and Distributed Systems*, 18:460–473, April 2007.

[132] Li Zhuang, Feng Zhou, Ben Y. Zhao, and Antony Rowstron. Cashmere: resilient anonymous routing. In *2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*, pages 301–314. USENIX, 2005.