SPADS: Publisher Anonymization for DHT Storage

Pascal Felber*, Martin Rajman§, Etienne Rivière*, Valerio Schiavoni* and José Valerio*
*University of Neuchâtel and §EPFL, Switzerland
first.last@unine.ch, martin.rajman@epfl.ch

Abstract—Many distributed applications, such as collaborative Web mapping, collaborative feedback and ranking, or bug reporting systems, rely on the aggregation of privacy-sensitive information gathered from human users. This information is typically aggregated at servers and later used as the basis for some collaborative service. Expecting that clients trust that the user-centric information will not be used for malevolent purposes is not realistic in a fully distributed setting where nodes are not under the control of a single administrative domain. Moreover, most of the time the origin of the data is of small importance when computing the aggregation onto which these services are based. Trust problems can be evinced by ensuring that the identity of the user is dropped before the data can actually be used, a process called publisher anonymization. Such a property shall be guaranteed even if a set of servers is colluding to spy on some user. This also requires that malevolent users cannot harm the service by sending any number of items without being traceable due to publisher anonymization. Rate limitation and decoupled authentication are the two mechanisms that ensure that these cheating users have a limited impact on the system. This paper presents SPADS, a system that interfaces to any DHT and supports the three objectives of publisher anonymization, rate limitation and decoupled authentication. The evaluation of a deployed prototype on a cluster assesses its performance and small footprint.

I. INTRODUCTION

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

A typical example is the crash report function implemented in many applications (e.g., the FireFox Web browser [1]). When the application experiences some failure, it sends back a report to a bug reporter server, which aggregates and stores these reports in order to help developers to track malfunctions in the software. In the case of FireFox, this crash report typically includes the set of URLs the user was visiting at the time of the crash. Obviously, while a recognized organization such as Mozilla is typically trusted by the user to properly anonymize the data received, and not use this information for other purposes, one cannot expect that users will grant such a level of trust to less-known service providers. The situation is even more complex for distributed applications that are implemented in a totally decentralized manner, i.e., on top of some peer-to-peer middleware. In such systems, many servers from different administrative domains are collaborating to propose the service. There is typically no centralized control or administration on the nodes forming the infrastructure, and it is often hard to verify that the software running on some node has not been modified for malevolent purposes.

Another example is the support infrastructure for collaborative ranking and profiling systems, where searches based on interest profiles can be submitted by users to complement an existing search engine. The results returned by such a system based on navigation and querying history of users with similar profiles [2]. A last example is the discovery mechanism used in content dissemination systems such as BitTorrent: an internal DHT is used to announce the availability of new files. The disclosure of the publisher identity to third party servers by a compromised or enforced-by-law server might eventually compromise the availability of the target file and of the service in general, and more generally impact its attractiveness.

Systems such as PeerReview [3] 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, etc.).

Another problem that is typically faced in decentralized data processing and storage is that of clients willing 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 a 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 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.

The main contribution of this paper is a system called SPADS, that allows to support the aforementioned goals of publisher anonymization and rate limitation, while authenticating users in a reliable manner. Before stating the contribution of the present paper, we briefly present the motivating example that led to the design of SPADS.

A. Motivating Example

The proposal of SPADS is motivated by research done by the authors in the context of the Buzzaar project. The goal of this project is to allow 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 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. Second, the authentication authority (implemented as a Web application) 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) [4]–[6], with persistent-storage based on proactive replication. Finally, some external application servers are allowed by the authentication authority to access the data stored in the DHT and the result of its aggregation, which are the inputs for the navigation and representation tools.

In this context, only a small level of trust exists between clients and servers. Users trust the project authority for certifying their identity but not for processing data. Users do not trust the servers for not disclosing private information, but rely on SPADS mechanisms to hide their identity to the servers that will effectively process their data. Servers do not trust clients for using the system fairly; they rely on SPADS for limiting risks of information flood and guaranteeing that the information is posted by authenticated sources.

An important aspect of such a system is its tolerance to message loss. The data is aggregated, and the general statistical meaning of the aggregation is what matters more than the individual contribution to this aggregation that were sent by the users. As a result, a system such as Buzzaar and the others mentioned in this introduction, while being strict on the absence of privacy impairment, can tolerate the loss of a part of the information sent to the system (e.g., due to servers failures). Nonetheless, a necessary condition is that the aggregation remains statistically meaningful, that is, that the statistical navigation trend of users is still captured adequately. For the aggregation to retain its meaningfulness, the tolerated message loss probability has to be the same for any message sent to the system. Obviously, an adversary that would systematically drop messages based on their destination, their origin or their content may bias the aggregation. Using a random selection of the servers that will participate to the

Fig. 1: Buzzaar: general principle and flow of information.
insertion of some items solves this problem.

B. Contributions

The contributions of this paper are as follows. We present the rationales, design and implementation of SPADS, a system that allows to support rate-limited, publisher-anonymizing insertions in a generic peer-to-peer distributed hash table (DHT) running on unreliable and untrustworthy peers. 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.

C. Outline

The paper is organized as follows. In Section II, we present in more details the considered system model, the underlying peer-to-peer layer characteristics, the problem definition, and SPADS’s provided guarantees. Section III gives an high-level overview of the mechanisms underlying SPADS. These mechanisms are further described in Section IV. We present an evaluation using a prototype of SPADS deployed on a cluster in Section V. Finally, we present related work in Section VI and conclude in Section VII.

II. Prerequisites and Problem Definition

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.

A. System model

We consider a system composed of 1) clients, that produce some data; 2) 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; 3) a set of servers that aggregate and process the data from the clients; 4) applications that access the information stored by the servers.

B. DHT

The servers are organized in a distributed hash table (DHT). SPADS does not require any modification to the DHT itself. The current prototype uses Pastry [4] but SPADS supports other DTHs like Chord [5] or Kademlia [6]. 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.

C. Additional DHT mechanisms

First, we assume that the DHT is augmented by a best-effort proactive replication mechanism (as in CFS [7]) 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 [7]). 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 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. 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. 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). Since clients are not inserted in the DHT, they 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.

D. 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, etc.) 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. 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 [3], 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 key space where it could take
the responsibility of the element it wishes to promote (and reply with fake values to get(\(\cdot\)) operations).

E. 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 \(\Delta\). \(n\) and \(\Delta\) 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.

The publisher-anonymizing put(\(\cdot\)) operations does obviously not allow SPADS to acknowledge to the user the reception of her \((k, v)\) pair by the corresponding server in the DHT. The absence of such a notification would typically be used in presence of message loss or node failures to reinsert the data after a timeout. As such, SPADS does not provide a strong guarantee to the user about the aggregation of her data to the pool. Indeed, even if the various steps of the protocol are using acks to ensure the proper diffusion of protocol messages, the sudden failure of a node at some step of the protocol can still result in the \((k, v)\) pair being lost.

As previously mentioned, this limitation is not much of a problem in the context of the targeted system. Only the overall statistical aggregation of all values sent by the client has a meaning, and it can accommodate the loss of some values, provided that these loss happen equally probably for any message sent to the system. SPADS provides clients with this fairness property: the probability of one message being lost is the same for any message, regardless of its source. All the servers that participate in the anonymization process are chosen at random, and the rate limitation process is resilient to failures.

III. SPADS AT A GLANCE

This section gives a high-level overview of SPADS. The explanations are based upon Figure 2, which describes the main steps of SPADS’s mechanisms.

The user first registers with an authentication authority (AA), which checks its legitimacy using external means, e.g., a “captcha” (noted \(\circ\)). After to this one-time operation, the AA registers (\(\Theta\)) the user to a subset of \(r\) servers that will act as credential managers, or CMs (\(r=3\) in Figure 2). The set of \(r\) CMs is different for each client. They are collectively responsible for the rate-limitation of messages sent by this client. SPADS introduces a single DHT API extension: a\_put(\((\text{uid}, [(k, v)])\)) (\(\Theta\)). The pair \((k, v)\) is forwarded by a series of intermediate servers, starting from some entry point server (EP) and ending at the last anonymous insertion delegate server (AID). The EP and AID 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 and uid. It uses the latter to obtain the number of credentials the client is currently allowed to use, by asking the \(r\) CM servers (\(\Theta\)). 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 (\(\Theta\)). 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, 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.

IV. ALGORITHMS

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

A. RSA infrastructure

Each server is associated with a pair of private and public keys. All public keys are known, and signed, by the authentication authority (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 (and thus, its ID) changes. The AA keeps track of all issued certificates and ensures that an IP cannot be associated with more than one certificate. The content of a certificate is as follows:

<table>
<thead>
<tr>
<th>Certificate for a server s</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s.addr</td>
<td>IP address (also gives its ID in the DHT)</td>
</tr>
<tr>
<td>s.pubk</td>
<td>public key of the server</td>
</tr>
<tr>
<td>[s.pubk,IP]AA.privk</td>
<td>signature of the certificate</td>
</tr>
</tbody>
</table>

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.

B. 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 server, or a similar method (Figure 3-➊). After the legitimation, the authentication authority (AA) returns a client/user identifier (uid) to the client. This uid is randomly generated over a large 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 to authenticate itself by the servers 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.

C. 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 3-➋). 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: h1, ..., hr. These hash functions are known by all servers and by the AA. hi 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.

D. Credentials management

Each CM server, once it has registered the hash of the uid of some client, 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 server has no possibility to calculate the keys for the r − 1 other CM servers. This means that the CM servers 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 CM may differ. Using r ≥ 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.2

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 de-incentive clients from regularly asking the AA to register again with the system using a different uid: the expected number of credentials that the client

2Note that we allow a slight difference in the results returned by the various CM 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 CM is not deemed erroneous.
can get with such a re-registration is always less than what the client would have had without re-registering.\(^3\)

\[E. \text{ Servers selection}\]

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) 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 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.

\[F. \text{ Anonymization}\]

SPADS uses a variant of Chaum Mixes [8] 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 entry point to the DHT (EP), the next servers are anonymous insertion delegates (AIDs). The \((k, v)\) pair, in encrypted form, will follow this path: client \(\rightarrow\) EP \(\rightarrow\) AID\(_1\) \(\rightarrow\) \(\cdots\) \(\rightarrow\) 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.

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 of the next AID (which is by convention the null value \(\bot\) 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 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 \((c, p)\) to the EP along with the IP of the first AID. The EP then checks the credentials for the client and sends the pair to the first AID, which is able to decode the next AID’s IP, 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 the field IP equals \(\bot\) 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 \((c, p)\) 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.

\[Algorithm 1: \text{Encoding a } (k, v) \text{ pair at the client.}\]

\begin{table}[h]
\begin{tabular}{|l|l|}
\hline
\textbf{Notations} & \\
\hline
\(m\) & AES encryption of \(m\) using key \(k\) \\
\(m\) & RSA encryption of \(m\) \\
\hline
\end{tabular}
\end{table}

\begin{algorithm}
\caption{Encoding a \((k, v)\) pair at the client.}
\begin{algorithmic}
\State \textbf{Encode} \((k, v, \{\text{AID}_1, \ldots, \text{AID}_f\})\)
\State \(s \leftarrow \text{random number (256 bits)}\)
\State \(c \leftarrow ((k, v), s)\)
\State \(h \leftarrow \text{SHA1}(c)\)
\State // Encode for \text{AID}_1 (no next hop)
\State \(p \leftarrow [[\bot, h], s]\)_{\text{AID}_1.pubk}\)
\State // Encode for \text{AID}_1 \ldots \text{AID}_{f-1} (w/ next hops)
\State \(i \leftarrow f - 1\)
\While {\(i \geq 1\)}
\State \(\hat{p} \leftarrow [[\text{AID}_{i+1}.IP, h, p]]_{\text{AID}_i.pubk}\)
\State \(i \leftarrow i - 1\)
\EndWhile
\State \text{return} \((c, \hat{p})\)
\end{algorithmic}
\end{algorithm}

\[\text{\(^3\)There is one exception for regular re-authentications allowed by the protocol to prevent the stealing of uids: no more than once every } s \text{ 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 CM in charge of this new uid.}\]
It is not possible to corrupt \( c \) (e.g., to append the IP 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.

G. 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 of the client is the EP but since there are already \( f \) nodes colluding it cannot be involved and did not transmit the IP. In the second case, \( f − 1 \) AIDs are colluding with the EP to get the content along with the IP. To get the plain content, its encrypted version needs to go through some \( f^{th} \) AID. The IP 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 waits 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.

H. 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).

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

A. Experimental settings

SPADS is implemented in the Lua language, with some performance-critical functions written as C libraries (hashing, encryption and decryption). We leverage the SPLAY platform [9] and its open-source implementation. The client is implemented in JavaScript. We use an implementation of the Pastry [4] 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.

B. 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 credential managers (CM). 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 each of the servers on the chain, starting from the EP to the last AID, and present the breakdown of these costs in Figure 4, 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 \( \mu s \) for the EP and 8 \( \mu s \) for each of the 3 CM (the 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. The current prototype of SPADS exploit 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.

C. 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 5 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.

D. 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 6(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. The number of CMs in Figure 6(a) is fixed to 3, but Figure 6(b) presents the delay variation with CM counts of 3, 6 and 9. The different curves for these CM values are nearly indistinguishable: as the requests from the EP to the CMs are done in parallel, using more CMs has no impact on delays. 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 \(O(\sum_{i=1}^{f} i) = O(f^2)\).

VI. RELATED WORK

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

A. Anonymizing routing

The use of public key cryptography along a chain of forwarding nodes was introduced in the early 90s by Chaum [8]...
Similarly to SPADS’s anonymous insertion delegates, the onion server(s), among the ones that are volunteering to the system, allows to tunnel a TCP connection through one or several relay nodes. The most widely known implementation of the concept of onion routing is the TOR anonymization layer [10]. 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.

The 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 [20]). 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’s rate-limitation, based on limited credentials allowance for each user that needs to send data to a distributed system, is shared by DQE [21]. 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.

**B. Peer-to-peer overlays for anonymization**

FreeHaven [17] and FreeNet [18] are peer-to-peer 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.

**C. Rate-limitation and spam-limitation**

In [19] it is proposed an admission control protocol combined with an anonymous channel protocol. Compared to this system, SPADS doesn’t need the CMs to sign each and every incoming and outgoing message. We are not aware of any work that mixes the two goals of anonymizing the publishers of information in a distributed setting, and the limitation of the influence of cheating users.

As mentioned in Section II, 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 [3], which we described in Section II.

**ACKNOWLEDGMENTS**

This work was supported in part by the Swiss National Foundation under agreement number 200021-127271/1, and by the Hasler foundation.
REFERENCES


