Peer-to-Peer Networking and Applications manuscript No.
(will be inserted by the editor)

An identity-based approach to secure P2P applications with Likir

Luca Maria Aiello · Marco Milanesio · Giancarlo Ruffo · Rossano Schifanella

***** PREPRINT *****

Accepted: 21 December 2010

Abstract Structured overlay networks are highly susceptible to attacks aimed at subverting their structure or functionalities. Although many secure architectural design proposals have been presented in the past, a widely accepted and comprehensive solution is lacking. Likir (Layered Identity-based Kademlia-like Infrastructure) is our solution for implementing a secure Peer-to-Peer network based on a Distributed Hash Table. Our purpose is to focus on three main goals: (1) providing security services and a secure overlay infrastructure against the vast majority of security threats on P2P systems, (2) dynamically creating a bridge between randomly generated peer identifiers and user identities, and (3) supplying the developer with a middleware API that can easily deal with peers' identities. Placing the emphasis on user identity results in a highly secure distributed framework which is very fitting for privacy-aware and efficient implementation of identity-based applications like social networking applications. Detailed security analysis and performance evaluation are provided. Moreover, an implementation of Likir is introduced and a case study is presented in order to show its practical use in a real-life example.

Keywords DHT · Routing Poisoning · Sybil Attack · Storage Attacks · Distributed Social Networking Systems

Luca Maria Aiello · Marco Milanesio · Giancarlo Ruffo · Rossano Schifanella
Università degli Studi di Torino, Computer Science Department
C.So Svizzera, 185 - 10149 Torino, Italy
Tel.: +039-0116706711
Fax: +039-011751603
E-mail: aiello@di.unito.it

milane@di.unito.it · ruffo@di.unito.it · schifane@di.unito.it

1 Introduction

Recent research on reliability of Peer-to-Peer (P2P) overlay networks has focused mainly on three aspects: scalability of fully decentralized architectures [13], incentive mechanisms against free-riding [20] and Distributed Hash Tables (DHTs) security problems. Of course, security is very relevant, especially when proposed architectures are addressed to the implementation of applications that are more critical than file-sharing. Nevertheless, despite the efforts spent in securing such systems, many security threats are still largely feasible on overlay networks, since Sybil-based attacks, attacks on storage, as well as attacks on routing and DDoS remains without widely accepted countermeasures. Furthermore, the new horizons that are emerging for DHTs' usage provides the deployment of increasingly high-level applications, like Social Networking Applications. In this context, a very comprehensive and strong security countermeasure has even a greater practical importance, because sensitive users' information is exchanged and possibly stored in the network.

This paper describes *Likir*, a secure extension of the Kademlia protocol. When using a P2P overlay network, one of the most problematic issues arises with the uncontrolled assignment of node identifiers, that are used to set the responsibility of content storage and to route messages. The idea behind Likir is to complement such identifiers with a *strong* node's identity notion which allows to build a secure, authenticated communication protocol that provides an effective defense against well known attacks. By exploiting a Certification Service, we give peers verifiable and certificated node identifiers, which are tightly coupled with the users' identities. Doing so, most of the security issues are overwhelmed, or at least strongly mitigated, under a very general adversary model. Likir security services are transparent, enabling developers to consider decentralized implementations of distributed applications, without concerning to many of the security bonds that emerge from the adoption of such solutions.

In addition to improving the DHT security features, the embedding of a strong identity management into the overlay allows to provide *services* to the application level. First, reputation and trust management algorithms can be deployed on such systems, because a proof of certain actions from a specific peer can be easily provided to other peers if our protocol is adopted. Second, a safe identity-based index side filtering, which allows the aggregation of different services on a user identity basis, can be built; this feature allows the composition of many application modules, facilitating their interoperability .

The advantage of including identity in the overlay is then twofold. First of all, reaching this at the application level will worsen the load on the application itself, thus increasing its complexity. Conversely, if this is achieved at a routing level, the same identity-based services can be reused by several applications. Furthermore, integration between (possibly) many different applications is made easier because of the sharing of the same identity. This has a significant impact on the design of new applications, because it allows mash-ups based on the explicit link that resources have with their owner.

A simple Application Level Interface that can be used by the developer in a very straightforward way is described along with its Java implementation. A practical case study based on a social networking setting is described in detail.

Lastly, an evaluation of the impact of the security layer on a given P2P system is needed. It is well known that authentication and verification services add a significant overhead to the performance of a given architecture. For this reason, we performed an extensive test on a real network environment in order to assess the feasibility of our approach.

1.1 Contributions and roadmap

This paper is rooted in another work by the same authors, originally presented in [2], where we focused mainly on the definition of the peer interaction protocol. Here, we extend the former work both in width and depth. We define the Likir architecture more in detail and introduce a deeper discussion about attacks. We perform an accurate security analysis, a performance analysis performed through PlanetLab experiments, a discussion on the Reputation Service together with a simulative experiment on its effectiveness and an API definition. Furthermore, the recent attention of the P2P community toward Online Social Networks (see Section 7) made the time ripe to highlight the contribution that the Likir framework can produce in a distributed Social Network System setting. For this reason, we describe a wider architectural specification that includes a web registration service and we present a case study focused on two simple identity-based applications; such contributions are aimed to show Likir's inbred adaptability to a distributed Social Network System setting.

The paper is organized as follows. Section 2 gives the state-of-the-art in Distributed Hash Tables research, focusing particularly on Kademlia and in previous attempts in securing DHTs. In Section 3 the adversary model that we are taking into account and a description of known attacks and ad-hoc countermeasures are presented. Section 4 presents the architectural model and the Likir protocol, explaining the involved primitives for the nodes' interactions. The following sections present the performance and the security analysis (Section 5) and the performance evaluation computed through emulative tests (Section 6) carried on PlanetLab. Section 7 presents Likir API together with a case study. Conclusions are given in Section 8

2 Background

Next, we briefly expound the features common to all Distributed Hash Tables (DHT) in order to establish a useful terminology for the following; then the specific case of Kademlia DHT, on which Likir is based, is analyzed with more detail.

2.1 Structured P2P systems

Distributed Hash Tables (e.g.,[46,37]) are a class of fully distributed systems which provide an exact-match lookup functionality: given a certain *key* from a flat identifier space, they retrieve the value associated with such key. DHTs can be exploited to develop a wide range of applications (e.g. [38,32,24]).

From an application point of view, at a high level of abstraction, a generic DHT system could be defined with a 6-tuple:

$$\boxed{\begin{array}{c} DHT = \langle K, N, C, \kappa_N, \kappa_C, \lambda \rangle \\ \kappa_N : N \to K \\ \kappa_C : C \to K \\ \lambda : K \to \{N\}^* \end{array}}$$

K is the DHT *keyspace*, a large (usually 2^{128} or 2^{160}) set of numeric keys, N is the set of online nodes and C is the set of all the resources owned by the users (we call them *contents*). An identifier chosen from the keyspace is assigned to every node (function κ_N) and content (function κ_C). Usually, nodes generate randomly their ID (the *NodeId*) while the content ID is calculated from the cryptographic hash (for which a collision is unlikely, for large values of $|K|$) of the content payload or from its metadata; however the definition of the function κ_C could be delegated to each specific application, depending on the structure of the resources. λ is a function of *responsibility* that associates the task of storing all the content marked with the same key to a set of replica nodes; these nodes are called *indexes* for the key. A node interaction protocol can perform a lookup procedure that, in $O(log|N|)$ steps, is able to locate the index nodes for any key.

Various DHT specifications differ in the routing table structure and updating procedures and in the nature of the lookup procedure (which can be either iterative or recursive). A real DHT implementation must also provide several other features like techniques to maintain contents over time even with a high node *churn rate*, or caching strategies that avoid *hot spots* for popular keys.

We talk about *structural components* of a DHT referring to four elements:

– The mechanism for *identifiers assignment*
– The *routing table*, containing the contacts of the known nodes
– The *storage* of contents
– The *interaction protocol* between nodes which determines the lookup procedure and the *bootstrap*, that is the join process of a new node to an existing network.

2.2 Kademlia

Kademlia has a 160 bit keyspace and assigns random $NodeId$s at the beginning of the bootstrap phase; contents are marked with keys determined by the application above (e.g. with SHA-1 hash). The routing table is organized into *k-buckets*, lists of at most k contacts structured as follows:

$\langle IPaddress, UDPport, NodeId\rangle$

Such triples are kept ordered with a Last Recently Seen (LRS) policy. Buckets are arranged as a binary tree and contacts get assigned to buckets according to the shortest unique prefix of their $NodeId$s. There are no specific routing update messages; the routing table is updated just when a generic message is received. If the sender's contact is already present in the corresponding k-bucket, the contact list is rearranged in accordance with the LRS policy, or added otherwise; if the k-bucket is full, the contact at the bottom of the list is probed and replaced if it fails to answer. A *splitting* procedure is used to extend the routing table: when a contact is added to the k-bucket corresponding to the local node's $NodeId$, if that k-bucket is full it is split into two new k-bucket that become children of the previous one in the binary tree.

Kademlia lookup is an iterative procedure that aims to identify the k nodes whose $NodeId$s are the closest to a given key; the distance between two generic elements of the keyspace is their exclusive or (XOR), interpreted as an integer. The lookup starts by selecting the α contacts closest to the target id; at each lookup step, α nodes are queried with a UDP RPC called FIND_NODE for the k contact they know nearest to the target key. Again, the α contacts nearest to the target are chosen from the returned sets and the procedure is iterated until no returned contact is closer to the target key than previously known nodes. The result of the lookup is the set of the k probed nodes nearest to the target key.

Contents retrieval is made by replacing the FIND_NODE with a FIND_VALUE RPC, which has the same semantic but returns a set of contents if the queried node has in its storage a resource marked with the lookup key. Resources storing is made by invoking a STORE RPC on the nodes found with a lookup. Bootstrap is simply made by performing a lookup procedure for the local $NodeId$, starting to query a bootstrap node, whose contact is assumed to be known. The protocol provides also a PING RPC for signaling purposes.

Nothing is assumed on the structure of node's storage but it has the semantic of a map from keys to contents, provided with appropriate *put* and *get* primitives.

For further details on Kademlia specifications we refer to [31].

3 Security threats and countermeasures

Opponents that we take into account are users that aim to break off or degrade the DHT service or to exploit the potential of the network to attack another peer or a target service outside the DHT. We suppose that an attacker is able to perform the following operations with minimum computational effort:

– run a large number of node instances on the same computer
– spoof its nodes' $NodeId$s and network addresses

– intercept and alter the communication flow between any two nodes
– conspire together with other malicious peers in order to accomplish coordinate attacks

This broad freedom of action allows an attacker to effectively put off a large spectrum of attacks against every network's structural element of any DHT. Next, we classify and inspect the attacks categories that such adversary can put off. Several classifications of DHT attacks can be found in literature (e.g. [42,12,47]), and many of them focus on the exploitation of arbitrary $NodeId$ assignment and on the routing procedure compromise. Our purpose is to consider a wider range of attacks, including those against DHT storage functionality, and also Man In The Middle attack.

3.1 Attacks on DHTs

Sybil attack

In a structured P2P network, $NodeId$s are generated locally from each node instance, arbitrarily. As we stated before, we suppose that a user can generate many node instances on the same machine, with as many different $NodeId$s. Multiple identities belonging to a single user are called *Sybils* [17]. Such behavior is in itself harmful because it undermines the redundancy property of the DHT system. However, Sybils can be used by an attacker to put off massive and organized attacks [30]. Assigning identifiers near to a target key to a sufficient number of Sybil nodes, an attacker could be able to intercept and discard most of the lookup requests for that key, thus censoring the contents stored in the DHT for that key.

Honest nodes can be distinguished from Sybils using validation mechanisms based on cryptographic puzzles [36], or exploiting assumptions on the underlying physical network (e.g. [48]) in order to identify nodes which are instantiated at the same physical position. Several ad-hoc protocols have been proposed in more recent years; for example, in [4], each node is dynamically associated to a monitor node that moderates transactions involving its twin node, thus making ineffective any Sybil attack attempt. Another recent approach leverages the acquaintance between human users to detect the untrusted contacts established by Sybils [50]. A centralized access control service (e.g., [44]) could be a very effective solution, but such architectures should avoid single point of failures and grant high scalability to be adopted.

Routing attacks

The **routing table poisoning** is the most commonly documented routing attack. It consists in injecting *ad-hoc* entries in the victim's routing table to alter the correct message routing procedure. Doing so, an attacker can disrupt the correct message flow or cut off groups of nodes from the network (i.e. **eclipse attack** [41]). Routing table poisoning comes out easily in a DHT environment, because of the *push-based* approach in routing information updating: since the routing table is built and updated on the basis of unsolicited messages received from other peers (like neighbor nodes' routing table publishing), an attacker could easily replace most of the entry victim's routing table entries with fake information. In particular, a node is prone to poisoning attacks during its bootstrap phase, when a very little information on the overlay is available. Subverting the routing procedure by uncorrectly answering to routing queries is a simpler, but sometimes effective, attack called **lookup misdirection**).

The most common countermeasures to routing poisoning is putting constraints on $NodeId$s assignment and routing procedure [12]: if a malicious node cannot choose arbitrarily its identifier and he can insert only its reference into a specific slot of the victim's routing table, the index poisoning would be unfeasible in practice. Ad-hoc distributed protocols for routing table verification (e.g., [41]) or periodic resets of routing tables aimed at flushing out poisoned entries [15] have also been proposed.

Storage and DDoS attacks

A node is free to insert into the DHT any content bound to arbitrary lookup keys, which are chosen at application level. Attackers can disseminate contents reporting fake or harmful information. We talk about **index poisoning attack** when bogus contents are deliberately spread to the nodes responsible for those contents lookup keys (the *index* nodes); this attack is particularly effective and notorious in P2P file sharing systems. If index poisoning is massively carried into effect, the ratio between the number of fake and true contents can soar, hiding the original contents from the lookup process [35]. When contents are references to other resources that are intentionally corrupted or fake we talk about **pollution attack** [26], a index poisoning closely related attack. Index poisoning is the main mean to perform DDoS attacks [33]. Indeed, in content sharing applications, for example, nodes publish in the DHT the network addresses of content providers. If an attacker spreads references to a very popular item, specifying a target service as the source of that item, he/she will cause the redirection of all the item requests to the victim, easily realizing a TCP flooding. So, solving the index poisoning problem in DHTs provides also a robust shelter agains DDoS attacks.

Rating systems, based on resources or users [27], or the exploitation of trust bonds established between users in an external social network [18] can be used to evaluate the quality of a content in order to detect attempts of index poisoning. Ideally, one would like to have a distributed reputation scheme that responds to attempts to evade detection by changing identity. The use of public-key schemes to mark contents has been explored in file sharing context [28].

Man In The Middle attack

In our adversary model an attacker is able to overhear and modify the content of messages flowing between any two endpoints. This is a tangible occurrence in real overlay networks, because of the use of *buddies* to manage the communication to nodes who reside behind a firewall or a NAT service. A detailed description of how buddy system works in the Kad DHT is given in [9], while in [45] a crawling analysis of Kad network shows that the portion of nodes behind a NAT or a firewall is very significant.

To avoid Man In The Middle (MITM), a *mutually authenticated* channel between the two communication endpoints must be established and the integrity of exchanged data must be assured. MITM is related not only to P2P systems but to every distributed service; for this reason, many studies about MITM resistant two-ways authentication protocols are available in literature. A detailed systematic design of families of MITM resistant authentication protocols is available in [7]; we refer to this document for further detail.

3.2 Applying countermeasures

The foregoing overview highlights the main instruments that can be taken to develop a comprehensive defense against cited attack categories. Specifically:

1. *NodeId*s must be generated randomly. The possibility of arbitrary *NodeId* selection should not be left to any node
2. The possibility for a user to generate many nodes on a single machine must be severely restricted or made as expensive as possible
3. The procedure for routing tables updating should provide appropriate constraints. A peer should be able to insert into a second peer's routing table only its own contact. The routing table in which its contact is added should not be determined by the message sender
4. During the bootstrap, the node must acquire routing information from trusted sources
5. A unique, strong user identity must be associated to each node. This identity must be certified and verifiable by other peers so that a system for the evaluation of user behavior can be realized
6. The communication protocol between nodes must be two-way authenticated and must ensure the integrity of messages

As we stated before, our attack analysis is quite general because it does not take in consideration any detail of a specific DHT. So, Kademlia protocol is included in the previous security considerations. The following Section describes the model of a DHT-based system that is able to fulfill all the mentioned points.

3.3 Secure DHTs

In the following, an overview of some the most significant approaches in making DHTs more secure and robust against adversarial behavior is carried.

Myrmic [49] is a enhancement of Chord [46] designed to resist attacks against the routing procedure. Myrmic adopts a “root verification protocol” that allows to check that the responsibility function λ is correctly applied. This is accomplished by the combined action of a trusted authority, which issues certificates specifying the responsibility keyspace area for a node, with a set of designed witness nodes, that checks that the responsibility area is respected. The main drawbacks of this approach can be seen in mainly three issues. First, there are too many assumptions: an external Certification Authority is used, loose synchronization between nodes is needed and the impossibility of carrying Man In The Middle attacks is stated. Second, if the Certification Service fails it will be not possible to join again the system. Instead of this constraint, our approach allows nodes to bootstrap even if the Certification Service fails, except for the very first join. Third, Myrmic does not face attacks against storage.

An approach very similar to Myrmic is adopted also in NeighborhoodWatch DHT [6], where a third party authority issues signed tokens to certify the responsibility of a node on a keyspace subset; here loose peers' clock synchronization is required.

S/Kademlia [5] is a secure Kademlia-based routing protocol, robust against common attacks. It limits free *NodeId* generation by using crypto puzzles in combination with public key cryptography. It extends the Kademlia routing table by a sibling list and it reduces the complexity of the bucket splitting procedure. Its lookup algorithm uses multiple disjoint paths to increase the lookup success ratio. Finally, it allows the DHT to store data in a safe replicated way to reduce impact of attacks against the storage.

In [39] authors leverages on node identity assignment procedure to reduce the impact of some dangerous attacks. An ID assignment protocol based on identity-based cryptography is presented, showing that the id-based cryptography is a suitable and affordable technique that preserves scalability by introducing a slight overhead. The described bootstrap procedure is accomplished through a weak authentication method (i.e., based on a callback to the presented IP address) that has to be executed at each join.

For reasons of space, we cannot present a full overview of DHT security solutions, but a wider dissertation on DHT security techniques can be found in [47]. In the next Section, in parallel to the analysis of each attack, we inspect some of the most relevant specific countermeasures proposed in the past.

4 Architectural model

Likir, Layered Identity-based Kademlia-like InfRastructure, is the architectural model of a new DHT system that offers both a very high protection level from all the most common attacks against structured P2P networks and a simple framework supporting identity-based services. The means by which Likir reaches these goals is the delegation of user's identity management to the overlay network layer. Likir architecture is structured in three main modules, shown in Figure 1.

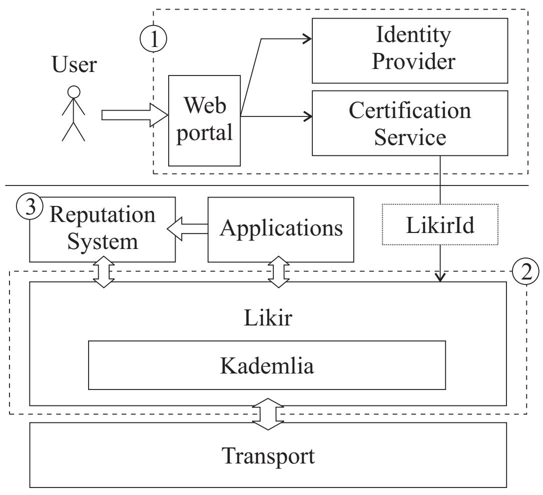

Fig. 1 Likir architectural model

The first is a user registration service, accessible via Web, which returns to the user the certified identity that will be used to mark its node on the overlay network. The service is composed of a web portal, which interacts with an Identity Provider for user's identity verification purposes and with a Certification Service for the creation of the certified identity. The second is a DHT protocol which extends the Kademlia protocol, encapsulating it. This layer provides an essential set of simple and general purpose API for developing any kind of application. The third consists of a reputation service at the same level of the applications layered on Likir, which interacts both with the underlying DHT node and with the various applications.

Next, we inspect all three modules, defining in detail purposes, functions and interactions between them. We made only a pair of assumptions, useful for the following discussion; we suppose that each user has a pair of RSA keys and an *OpenId* account. Finally, we adopt the following notation.

A, B	:	*Likir* users
$NodeId_A$	:	node A's DHT identifier
$UserId_A$	:	A's OpenId
K_A^+, K_A^-	:	A's public and private key
$Sig(msg, K_A^-)$	:	message signed with key K_A^-
$H(o)$	:	hash code of the object o
ts	:	timestamp
$a\|\|b$	:	concatenation of strings a and b

4.1 User registration service

In order to use Likir services, you must fulfill a user registration procedure; Likir architecture provides a web portal for this purpose.

A generic user A is authenticated by the registration website using the OpenId protocol. This implies that A sends its OpenId (the $UserId$) to the registration service, which in turn contact a third-party OpenId provider, where A has a valid account, to validate the user identity (a detailed description of the OpenId 2.0 framework is given in [34]). Once the *UserId* is validated, A sends its public key to the registration portal through a simple submission form supplied on the website. The OpenId and the public key are then forwarded to a trusted entity, the *Certification Service* (CS). We pass over the specific structure of the CS and we handle it as a black box service, which peculiar function is creating signed identifiers for newly joining users; we only suppose that it owns an RSA key pair $\langle K_{CS}^+, K_{CS}^- \rangle$.

Upon A's request, the CS binds A's *UserId* to A's public key and to a random 160bit string that will represent the DHT identifier of the Likir node. The binding is made through the production of a cryptographic token which is then sent to A through a secure channel:

$$LikirId_A = E_{K^-_{CS}}(NodeId_A||UserId_A||K^+_A||ts_{exp})$$

CS keeps track of the association between $UserId_A$ and $LikirId_A$, to prevent subsequent requests from causing the production of unnecessary signatures. Only when $LikirId$'s validity is near to its expiration (determined by ts_{exp}) the CS must create a new $LikirId$.

When user registration procedure is successfully terminated, A can instantiate its own Likir node just supplying the $LikirId_A$, its key pair and the CS's public key, that we suppose to be publicly available on the registration portal.

Observation: It is very important to notice that, once a user has obtained his $LikirId$, he does not need to contact CS until his signed ID validity expires. If the CS fails, the user registration service becomes unavailable but the network activities are not affected, because the users that previously obtained their $LikirId$ can join the overlay without querying any central service. Since the ts_{exp} can be chosen to last even many years, we can state that *the CS is **not** a real single point of failure of the system.*

Of course, the CS infrastructure and the maintenance of the registration portal have a cost that have to be sustained by some project promoter. Such infrastructural cost could be low enough to be set up also by no-profit organizations like universities (realizing a single-server PKI and a OpenId-compliant web service is relatively cheap). However, also commercial promoters could be interested into supporting the project because of the potential revenues from advertisements. Since Likir can be exploited also as a platform for the creation of a decentralized, privacy-aware online social network (see Section 7), we believe that its potential attractiveness to consumers could be high. An alternative way to finance the registration service cost could be to ask for a micropayment for each new $LikirId$ issued.

4.2 Node interaction protocol

To join the Likir network, a node must just perform the Kademlia bootstrap procedure, namely performing a lookup for its own $NodeId$ starting querying a live bootstrap node. If the node is not aware of any alive contact (e.g. it is performing its first bootstrap), he can send a proper request directly to the CS, which responds with a signed *bootstrap list*. The CS controls at least one alive Likir node which executes a periodical probing task (simply performing lookups for random $NodeId$s) in order to learn of fresh contacts. However,

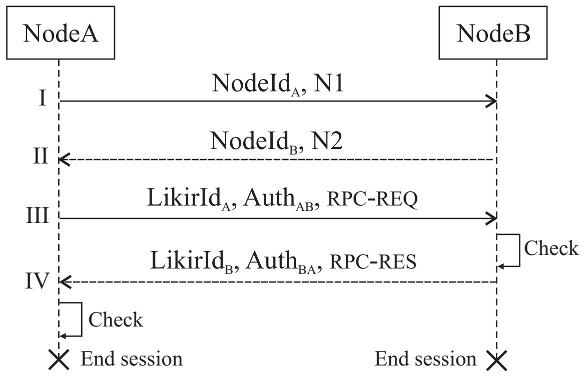

Fig. 2 Likir node session

it is important that every node keeps track of a substantial number of previously known contacts to use as bootstrap nodes, to avoid the CS to be flooded by bootstrap list requests.

A node A can successfully send a Kademlia RPC to a node B only if both A and B follow this four way *session.*

I $A \rightarrow B : NodeId_A, N1$
II $B \rightarrow A : NodeId_B, N2$
III $A \rightarrow B : LikirId_A, Auth_{AB}, \text{RPC-REQ}$
IV $B \rightarrow A : LikirId_B, Auth_{BA}, \text{RPC-RES}$

$N1$ and $N2$ are randomly generated nonces; RPC-REQ and RPC-RES fields are respectively the request and response RPC defined in Kademlia. Messages sent at steps I and II must be somehow marked differently (e.g. distinct opcodes), to differentiate the request from the response.

$Auth_{AB}$ and $Auth_{BA}$ are two authentication tokens structured as follows:

$$Auth_{AB} = Sig(NodeId_B||N2||H(\text{RPC-REQ}), K^-_A)$$
$$Auth_{BA} = Sig(NodeId_A||N1||H(\text{RPC-RES}), K^-_B)$$

Figure 2 shows the session message exchange. Steps I and II just accomplish a preliminary nonce exchange. Messages III and IV are thoroughly symmetric; the Kademlia RPC is sent with the $LikirId$ of the sender and with a signed authentication token. The $Auth$ contains the addressee $NodeId$ (to avoid replay attacks), the previously received nonce (to assure the freshness of the token) and the RPC hash (to protect RPC message from modifications). We should observe that freshness of authenticators can be granted also replacing the nonces with timestamps, avoiding the preliminary message exchange; however this would require at least to assume a loosely synchronization of nodes' clocks. Later, in Section 5, we show how this protocol assures authenticity of messages.

The RPCs exchanged during the session follows exactly the Kademlia RPCs specification, except for the

STORE RPC. In Likir, Kademlia STORE request message is enhanced to prevent the disownment of the insertion operation; it is structured as follows.

$$\text{STORE RPC} = k||content$$
$$content = Obj||Cred$$
$$Cred = Sig(UserId_A||k||H(Obj)||ts||ts_{exp}, K_A^-)$$

The STORE RPC embeds a *content* and its lookup key. The content is composed by the actual object, which is application-specific, and by a signed credential *Cred*. *Cred* token includes all the useful information on the published object: its ownership ($UserId_A$), its lookup key (k), its SHA-1 hash to grant its unalterability, the publish time (ts) and its validity period (ts_{exp}).

Content store and retrieve operations on the DHT can easily built upon the defined primitives, according to the Kademlia lookup protocol. We call PUT and GET such operations, respectively.

4.3 Reputation system

The last Likir module to be introduced is a *Reputation System* (*RS*), placed at the application level, which interacts both with the other applications and with the underlying DHT node.

We do not define a specific Reputation System for Likir, because different application suites could adopt different systems, depending on their needs. Many *RS* models could be suitable for Likir architecture even if fully decentralized reputation systems like [43] or [23] could better match with the pure P2P Likir design. A wide overview of existing and proposed reputation and trust systems models can be found in [22].

We make just a few loose assumptions on the *RS* behavior. We presume that the *RS* computes a reputation score for each known peer on the basis of information received by the applications or retrieved from the DHT; we suppose that the *RS* exhibits a simple API that allows the applications to evaluate other user's behavior in order to single out the misbehaving peers.

When an application retrieves a resource from the DHT, it evaluates the genuineness of that content on the basis of application-dependent rules. For example, a method to assess the validity of a resource could be the evaluation of the relationship between the resource's content and its lookup key; usually, every application defines specific rules to bind lookup keys to objects (e.g. key calculated with the hash of object's metadata), so if a resource was inserted with a lookup key that is not related with its content, respect to the lookup key production rules of that application, the resource can be marked as invalid, and its publisher as a polluter.

When a polluted content inserted by a malicious node X is retrieved, the application passes to the *RS* a evidence of X's misbehavior:

$$evidence_X = UserId_X||content||applicationID$$

The evidence contains the polluter's $UserId$, the polluted *content* and a string which identifies the application that retrieved the content. Depending on its particular policies, the *RS* takes care of spreading the evidences to other remote *RS* instances. When receiving a new evidence from a remote peer, the *RS* pass it to the application corresponding to *applicationID* (if installed) in order to evaluate if the received *item* is actually a polluted content.

Based on the number of the acquired evidences on a user, the *RS* determines its reputation score; when a user's reputation score falls below a predetermined threshold, the *RS* instructs the underlying Likir node to add his $UserId$ in a local *blacklist*. Note that, since the *content* inside the $evidence_X$ contains a *Cred* token signed by X, the ownership of the polluted item is verifiable by any peer. This implies that a node can always prove that a misbehaving peer have inserted a polluted content and, symmetrically, a malicious user cannot forge fake evidences for honest peers.

During a communication session, when the identity of the other endpoint is learned (step III for server node, step IV for client node), the blacklist is looked up; if the identity appears in the blacklist the session is aborted, thus preventing a considered malicious node to use network services. If the contact of a node that is recognized as a polluter is in the routing table of an honest node, it cannot be erased immediately, because the routing table retains only the information about the $NodeId$s and not about the $UserId$s; besides, it will be wiped out when the first session with the polluter is established, when the binding between his $UserId$ and $NodeId$ is verified.

If the *RS* design is effective and the reputation information is quickly and efficiently spread across the network, a polluter peer is quickly excluded from Likir service, because his $UserId$ is added to many honest nodes' blacklist. An example of a simple *RS* scheme and an emulative experiment on its effectiveness in banishing malicious peers is given in Section 5.2.

It is worth noting that managing the blacklist at middleware level allows to apply a strict exclusion policy towards misbehaving users; if a user takes a malicious behavior within a specific application and its bad reputation spreads across the network, he will no longer be able to use any other application service, because the blacklist is shared between all Likir-layered applications. This feature is advisable because a severe punishment for polluters can be an effective deterrent for intentional bad behaviors.

4.4 Replacing RSA with IBS

Likir protocol exploits conventional RSA cryptography for token signatures, however, in agreement with the identity-centered Likir design, an Identity Based Signature (IBS) approach could be adopted as well.

IBS is a cryptographic technique that allows to compute a key pair whose public counterpart could be obtained from an ASCII string. This cryptographic paradigm allows a user to verify a signature of another user just knowing his user ID. The original IBS idea, based on the RSA function, was presented by Shamir [40], but it was subsequently revisited by Boneh and Franklyn [8] and Cocks [14], who used bilinear pairings [29] for efficient Identity Based Cryptosystems (IBCs) design. The presentation of IBS mathematical background goes beyond the goals of this work, however we give a brief overview on how such paradigm works.

In IBS, when a user A wants to send a signed a message to a user B, the following steps must be executed.

1. **Setup**: a trusted third party, the Private Key Generator (PKG), creates a pair of *master keys*: a public key MK^+ and a private counterpart MK^-.
2. **Extraction**: A presents his identity (Id_A) to the PKG, who produces a private key K_A^- from MK^- and Id_A; the new key is then sent to A through a secure channel
3. **Generation**: using its private key K_A^-, A creates a signature s on message m and sends (m, s) to B.
4. **Verification**: B checks whether s is a genuine signature on m using Id_A and MK^+.

Likir can profitably take advantage of an IBS scheme. Since the $UserId$ must be sent in every communication session, the recipient of a request or response RPC always knows the sender's user name, so, using IBS, the Likir protocol overhead could be noticeably streamlined because the information of the user's public key in the *LikirId* could be omitted, and the $UserId$ only could be used to verify tokens signatures.

Nevertheless, IBS has two main drawbacks. The first concerns efficiency: the known IBS algorithms [19] in current implementations (e.g. Stanford PBC Library[1]) are slower than RSA, both in signature and in verification phases. The second (the most severe) is the *key escrow* property: the PKG is a genuine single point of failure, because if an attacker takes possession of the private master key, he could generate the private keys related to every $UserId$, thus violating the whole cryptosystem.

For these reasons, the current Likir implementation adopts a traditional public key scheme. However, the scientific community is still very active in IBC research, therefore the possibility of replacing RSA with IBS should be taken into account.

5 Security analysis

Likir contrasts the security menaces analyzed in Section 3 with its three main architectural elements: the enhanced node interaction protocol, the *CS LikirId*s issue service, and the Reputation System. The first two strongly mitigate the impact of the Sybil attack and avoid the occurrence of routing poisoning or MITM attacks, while the third reduces the effectiveness of the attacks based on storage pollution. So, our security analysis is divided in two parts. First (Section 5.1), we show the security properties of Likir protocol, then (Section 5.2) we present an emulative experiment that highlights how the adoption of a RS (even a very simple one) can quickly wipe out the polluter users from the network.

5.1 Secure communication channel

To show the Likir protocol effectiveness against poisoning, Sybil and MITM attacks we must proof two properties.

Property 1 ***Node authentication****. A node can communicate with others only providing its own LikirId*

Proof Suppose an attacker node X who's pretending to be A during a session with node B. Clearly, since the node session protocol requires to provide a valid proof of the *LikirId* ownership (the *Auth* token), X cannot simply reuse the intercepted A's *LikirId* to spoof its identity, but a valid couple of *LikirId* and *Auth* is needed.

X cannot produce a valid $Auth_{AB}$ itself, because the $Auth_{AB}$ must be signed with the private counterpart of the RSA key included in the $LikirId_A$, that is assumed unforgeable because it is signed by the CS; so, the entity that is able to produce a valid $Auth_{AB}$ is A only. So, X has only two ways to counterfeit its identity: to intercept and reply a valid $Auth_{AB}$ or to trick A into produce a valid $Auth_{AB}$.

In the first scenario, X must intercept an $Auth_{AB}$ that contains the same Nonce received by B (at session step I or II, depending if X is the session initiator or not); but if Nonce's size is big enough and a good pseudo-random generator is used by Likir clients, the probability that X can find such $Auth_{AB}$ is negligible.

[1] http://crypto.stanford.edu/pbc

In the second scenario, X must solicit A to build an *Auth* containing the Nonce received by B. This can be easily achieved by establishing a proper session with A, but the token thus obtained would be an $Auth_{AX}$, and not an $Auth_{AB}$. To get a proper $Auth_{AB}$, X must pretend to be B during the session with A, but this creates a cyclic dependency: to pretend to be A in a session with B you must pretend to be B in a session with A. □

Property 2 ***Message integrity.*** *Every message flow alteration attempt causes the session to abort*

Proof If an attacker modifies the data at session steps I or II, the *Auth* tokens sent in the following steps will be no more valid, because they include all the fields of the preliminary Nonce exchange messages. *LikirId*s and *AuthId*s are unalterable by assumption because they are signed; the same is for the RPC message, because its cryptographic hash is included into the *Auth*. So, the message flow between two nodes is unalterable by a third party. □

From Property 1 follows straight that $NodeId$ cannot be generated arbitrarily and cannot be spoofed. In Kademlia a new routing table contact is added at the end of a communication session, if there is enough room in the proper bucket. Since the overlay communication is authenticated, the $NodeId$ are randomly chosen and cannot be spoofed, and the position of the new contact in the routing table is determined locally on the basis of the sender's $NodeId$, an attacker cannot insert an arbitrary contact in an arbitrary position of the routing table. So, any routing table poisoning based attack is practically unfeasible.

Property 1, combined with Kademlia design, prevents also lookup misdirections. The nodes considered during the lookup process must be directly probed; since the node responsibility function depends only on the $NodeId$ and since the $NodeId$s cannot be spoofed, the initiator knows for sure what key responsibility area he's addressing to. Of course, denial of service (e.g. a node replies to a lookup query with valid contacts whose $NodeId$s are not close to the target) is always possible, but this is an inherent problem of the distributed lookup mechanism; anyway, the Kademlia lookup procedure is pretty resistant to such attack because favors the retrieved contacts that are nearest to the target. As we stated in Section 2.2, at every lookup step, the α nodes whose $NodeId$s are the closest to the target are probed. If, during the process, a honest node is queried, the k contacts returned will be likely closer to the target than the contacts returned by other non-collaborative nodes, so the next α nodes to be queried will be chosen from this set (since, usually, $\alpha < k$).

Protocol authentication and $NodeId$ randomness limit also the sybil attack impact. To run many different nodes, as many $LikirId$s are needed, but if the user subscription service provides valid techniques to avoid the registration phase automation, the effort needed to produce many sybils can be arbitrarily increased. For example, during user subscription phase, a credit card number can be required. Different identities could be issued for the same user, but the CS can limit the number of different $LikirId$s issued for the same credit card number, thus making very expensive the hoard of a huge number of identities. Anyway, even if an attacker has many nodes under his control, he cannot position them in specific key space areas, because the $NodeId$s randomness.

Finally, proof of Property 1, together with Property 2, shows the protocol resistance to MITM; furthermore, the bootstrap list provided by the CS for the first bootstrap is signed, thus partition attacks during the join phase are avoided.

5.2 Banishment of polluters

We show the effectiveness of the Likir blacklisting feature, combined with the RS action, through an emulative experiment. Each peer runs a Likir-layered test application with a very simple behavior: broadly, the peer periodically stores and retrieves contents from the DHT on random lookup keys; we suppose that the application is able to verify if a resource is polluted in a fully automated way, without interacting with a human user, so when a fake resource is retrieved, its publisher's $UserId$ is immediately inserted into the blacklist.

Each application interacts with a Reputation Client (RC), just notifying it when a new $UserId$ is blacklisted and providing the related *evidence*[2]. The RC stores a *evidence list* in a local database.

The RC behavior follows a simple, zero-tolerance gossip-based approach to spread local reputation information. Periodically, the *evidence* list is published in the DHT using a lookup key which is easily obtainable from the publisher's $NodeId$. Before the publishing phase, the RC retrieves the lists of other users in order to learn of new polluters' $UserId$s and, possibly, to increase its own list. To do so, the lists of the k known $NodeId$s nearest to the local $NodeId$ (the closest overlay neighbors) are retrieved from the DHT and the local list is possibly updated with new $UserId$s.

To make our experiment easier, and to get timescale independent results, we organize the emulation

[2] The RC design is just functional to our experiment, it is not an element of the Likir architecture

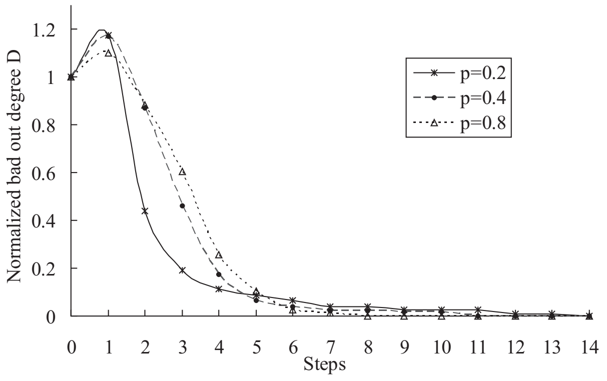

Fig. 3 Reputation test results

into time *steps*. At each step, every peer on the network performs a variable number of DHT operations; we refer to N_{put} and N_{get} respectively as the random variables of the number of store and retrieve operations executed and we suppose that these variables are normally distributed.The lookup keys specified as PUT and GET parameters are selected from a set of 10^5 randomly generated 160-bit keys; accordingly to previous studies on P2P contents popularity distribution on file sharing networks [21], we suppose that the frequencies of such keys are distributed accordingly to a Zipf's law; we choose an exponent equal to 1. When a step is over, the *RC* gossip service is activated and new blacklisted contacts are possibly learned.

The network is, of course, partitioned into two subsets: the *Good* and the *Bad* peers. The bad nodes are distinguished from others because they publish only fake contents and they do not take part in the *RS* activities. At each step, the **bad out degree** (D) of good nodes is measured. D is simply defined as follows:

$$D = |\,\{(x,y)\,|x \in \mathit{Good} \land y \in \mathit{Bad}\}\,|$$

A link between a good and a bad node (the (x, y) edge) is determined by the presence of the bad node's contact into the good node's routing table; the percentage of bad nodes is given by the p parameter.

We instantiate a 150 nodes network and then we run our emulation for different values of p. We set Kademlia routing parameters k and α to tiny values (respectively to 4 and 2) because the network size is relatively small. Our experiment is aimed to model a coordinated pollution attack, so we suppose that attacker nodes start polluting the DHT at step 0, and no new node is instantiated during the emulation period. Of course, during the whole overlay network life, several attacks like this can happen, however we focus only on a snapshot of a single attack which anyway easily allows to understand the effectiveness of Likir blacklist and RS.

The results are shown in Figure 3; the plotted values of D are normalized on the initial D value. In an initial phase, before fake contents are widely spread across the network, the number of bad contacts in the good peer's routing tables increases, because new contacts are learned due to the lookup procedures executed by the nodes of each partition. But the diagram shows that this trend is reversed after the first step; the value of D is reduced to about one fifth in just three or four synchronization steps, for every value of p, and then decreases asymptotically to zero, thus cutting off the cluster of bad nodes from the healthy part of the network.

The blacklisting method results effective also for very high values of p (e.g. $p = 0.8$) because, even there is a slight probability that a non-polluter node has the contact of another honest node in its closest overlay neighbors set, a great portion of contents on the DHT results corrupt, so many evil nodes are discovered at each step.

6 Performance evaluation

Compared to Kademlia, the Likir protocol introduces an overhead that affects both the number and the size of messages exchanged between nodes; besides, the cryptographic effort spent during a node session due to the signature operations increases the computational load on every single peer.

In order to quantitatively evaluate the performance decay due to additional messages, enlarged message size and cryptographic overhead, we opted for a test in a real, large-scale distributed environment. We run small Likir and Kademlia overlay nets on PlanetLab network, in order to compare the time effort needed for PUT and GET primitives in both protocols and to measure the average impact of cryptographic operations on the whole Likir session time.

The reader should note that a *scalability* test is not needed here, for three reasons mainly. First, we do not modify the Kademlia routing protocol neither its routing table management policy; thus, the number of hops for a lookup operation in Likir is exactly the same as in Kademlia. Second, the number of messages sent during a Likir session is incremented by a *constant* number, compared to a Kademlia session; this implies that the number of messages per lookup still grows logarithmically with the network size, like in Kademlia. Last, the cryptographic operations impact on the nodes that perform RSA checks and signatures but clearly do not burden the network with any additional traffic.

For these reasons, Likir has *by design* the same scalability properties that have been shown for Kademlia.

Element	Size	LIKIRID	AUTH	CRED
NodeId	20	●	●	
DHTkey	20			●
UserId	128	●		●
K^+	128	●		
Signature	128	●	●	●
Nonce	16		●	
Hash	20		●	●
ts	8	●		●●
Total size:		412	184	312

Table 1 Crypto token size (bytes)

RPC	Request	Response
PING FIND-NODE	596	596
FIND-VALUE	596	596 + 312 · n
STORE	908	596

Table 2 RPC spatial overhead (bytes)

Of course, the time needed for a lookup operation is greater in Likir if compared to Kademlia, so we want to quantify this gap in a real network environment. Prior to this, we present also a static analysis on the message size overhead and on the cryptographic primitives cost.

6.1 Spatial and cryptographic overhead

The size of a Likir message is greater than the size of ordinary Kademlia RPC due to the addition of *LikirId*, *Auth* and *Cred* tokens. In Table 1 the whole set of elements that composes these tokens is shown, together with their size; furthermore, the specific composition of each signed tokens is given, together with their total size. We suppose that 1024 RSA keys are used.

Once crypto token size is assessed, we can easily calculate the size overhead on each Kademlia RPC. Every RPC contains at least a *LikirId* and and *Auth*, which form the message header. In addition to this, the STORE RPC request contains also a *Cred* bound to the content to be stored, and the FIND-VALUE RPC response payload contains a *Cred* attached to every content returned to the querier. Of course, the number of contents per FIND-VALUE response is variable due to the availability of objects bound to the requested key. For this reason, the overhead for such RPC can change. To seize this variability, we define n as a variable representing the number of *Cred* per FIND-VALUE response. Table 2 summarizes the given considerations, showing the overhead for every RPC. It is worth noting that the additional header size is smaller than 1KB in the worst case, and the FIND-VALUE payload dimension overhead is linear with the number of retrieved contents.

The node interaction protocol (Section 4.2) requires also that both sender and receiver generate and verify signatures; for the sake of brevity, we refer to *gen* and *check* respectively for a signature generation and a signature verification. Table 3 summarizes the number of cryptographic primitives to be performed by a node during a whole session, for every RPC. We deliberately ignore the SHA-1 hashing operations due to the non-influential cost. The n additional *checks* reported to FIND-VALUE RPC client side represent the *Cred* verification of all retrieved contents. The impact of such primitives on the RPC session time is discussed in the next Section.

RPC	Sender	Receiver
PING FIND-NODE	$gen + 2 \cdot check$	$gen + 2 \cdot check$
FIND-VALUE	$gen + (n+2) \cdot check$	
STORE	$2 \cdot gen + 2 \cdot check$	

Table 3 Cryptographic primitives used in each RPC

6.2 Network emulation

We built a Kademlia implementation simply by replacing the Likir node interaction protocol with the classic Kademlia protocol on our Likir Java implementation; the Kademlia parameters k and α we chose are respectively equal to 4 and 2.

We bootstrapped 250 overlay nodes on as many PlanetLab nodes; we used the support of a centralized server for bootstrap lists distribution, as described in Section 4.2. Then, each Likir node executed 25 PUT and 25 GET, randomly interleaved, on random keys. The sequence of called primitives followed a Poisson process; the temporal distance between two events was determined by an exponential distributed random variable. The same experiment was made for the Kademlia configuration.

We measured the whole execution time of each PUT and GET. The cumulative distribution function of these times is depicted in Figure 4. We observe that, in both plots, the relationship between the two curves is different depending on the time range taken into account. In a first interval, from 0 to the value highlighted with the arrow, the Kademlia curve assumes values that are more than double than the Likir curve; in the second interval, up to infinity, the curves get asymptotically closer. This happens because in short lookup procedures the cost of Likir cryptographic operations assumes a non-neglegible weight respect to the overall PUT/GET time, while in the second range the network delay prevails on the time spent in signatures generations and checks.

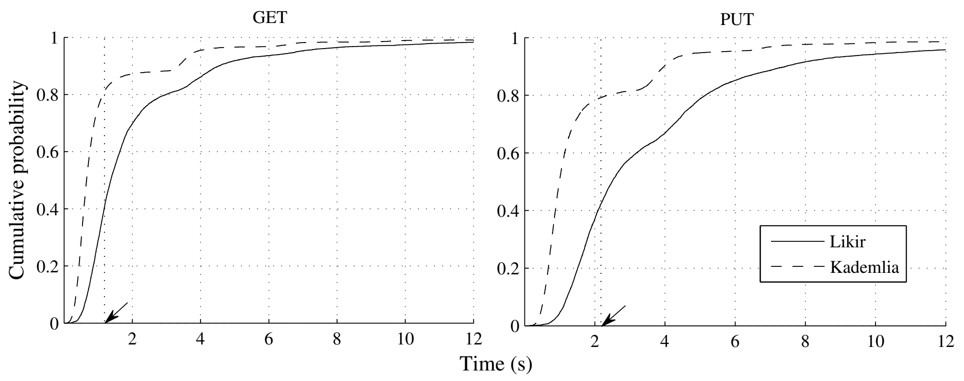

Fig. 4 CDF of GET and PUT times in the PlanetLab experiment

Operation	Likir			Kademlia		
	E	$\mu_{\frac{1}{2}}$	σ	**E**	$\mu_{\frac{1}{2}}$	σ
GET	2291	1366	2930	1276	659	2402
PUT	3877	2408	4123	1844	1091	2626

Table 4 Likir session duration (milliseconds) for both PUT and GET in the PlanetLab experiment

As expected, PUT requests are slower than GET because they require an additional hop to command the found index nodes to store the content. We measured that the mean lookup hop number is 2, so the number of step required is 2 for the GET and 3 for the PUT.

To give a more precise estimation on the overhead introduced by Likir we need some statistics (presented in Table 4). We notice that the PUT and GET mean and median time in Likir are roughly double respect to the same primitives executed with the Kademlia protocol. The standard deviation assumes always high values because the huge network latency variability and the different number of hops of the lookup processes. This is the result we expected, since in a Likir session four messages must be exchanged, compared with the two messages of a Kademlia session. This suggest that, on average case, the cryptographic overhead has a little impact on the overall time.

To give a more precise estimation of the impact of checks and signatures on the session time, we measured the mean time for *gen* and *check* operations on a PlanetLab node. Since we know the lookup hops mean, the average number n of retrieved content in a GET operation (we calculated $n = 4$) the number of primitives needed in PUT and GET for each hop (Table 3), and the mean time needed for cryptographic operations, an estimation of the mean time spent on the local node for cryptographic primitives can be easily done. We calculated an overhead of about 172 *ms* for GET and 347 *ms* for PUT. These values are less then one tenth of the total operation time, and, however, they even do not impact in full on the total PUT and GET time because of the parallel nature of the lookup process.

In conclusion, the network emulation results shows that the predominant Likir overhead is given by the additional message exchange, that necessarily doubles, on average, the basic DHT operations execution time.

7 Building P2P social applications

The great popularity of Online Social Networks (OSNs) has supplied Social Network Service (SNS) providers (e.g. Facebook, MySpace, Flickr) with a large amount of user data. Indeed, the centralized structure of such services allows providers to easily collect user's contents and to mine information about user social behavior. The possession and the consequent (and often unavoidable) exploitation of these data raises evident concerns on user privacy and on the right to use the data.

An interesting architectural solution to this problem comes from the peer-to-peer community. In fact, building a SNS over a pure P2P layer avoids the interference of a centralized control on information exchanged and possibly stored in the network [10]. Encryption can protect sensitive and private data from malicious crawling activities.

A recent research line have expanded this insights proposing decentralized frameworks suitable for SNSs [11,16,1], however, many design aspects should be explored further. In this context, Likir is an ideal platform for SNSs mainly for two reasons. First, the security level offered by its protocol grants a very high robustness to the modules above, which is very important for this kind of applications. Second, Likir's identity support better matches the SNSs' requirements rather than any other DHT, because it implicitly links data belonging to different applications with its embedded identity notion. For these reasons, in this Section we present Likir API showing how they can be profitably used in a identity-based OSN context. Anyway, Likir is a general-purpose framework and its security advantages are appreciable by any kind of application, even those without a strong identity notion (e.g. more “classical” file sharing or distributed storage applications). Further considerations on possible customizations of Likir for SNSs adaptation can be found in [3].

7.1 Likir API

The Likir’s interface to the application offers a very simple and essential set of primitives. We denote the node N of a user *userId* as N_{userId} and we suppose that *key* is an identifier of the keyspace.

1. BOOTSTRAP*()*: join the local node to an existing Likir network by contacting previously known peers as bootstrap nodes. If no peer is known, send a proper request to the CS to gain a fresh bootstrap list
2. PUT*(key, obj, type, ttl)*: lookup index nodes for *key* and ask them to save the binding between *key* and *obj* in their storage; *type* is an application-specific string while *ttl* is the time after which the binding might be eliminated from the storages. Returns the number of successfully queried nodes
3. GET*(key, type, userId, recent)*: lookup index nodes for *key* and query them for object binded to *key*. *type*, *userId* and *recent* are optional filtering parameters that can use to require only contents of a certain type, submitted by *userId*, or only the most recent versions of the content. Returns the set of contents found (possibly empty)
4. BLACKLIST*(userId)*: add *userId* to the local blacklist; every future session established with N_{userId} will be aborted

In the following, we refer to PUT and GET also to denote the messages (RPCs) originated by the corresponding API call.

Very sharp resource retrieval can be made through the index side filtering facility. If all GET parameters are set, at most one resource is returned (i.e. the last resource inserted by the specified user, under the specified key and type). Obviously, identity-based resource filtering could be achieved simply tagging the stored resource with a label that specifies the owner identifier. However, such method is vulnerable to storage poisoning attacks, therefore it cannot grant the resource ownership. On the contrary, Likir protocol assures verifiability through certificates.

We realized a Java implementation of Likir[3]. It follows faithfully the Kademlia specification except for the node interaction protocol, for the addition of a nested-map data structure which implements the content storage and for the management of the blacklist.

[3] Likir library is available at http://likir.di.unito.it

Algorithm 1: LiCal events management

```
1 Node n = new Node ($UserId_A$)
2 n.bootstrap ()
3 Object Calendar$_A$ = createUsrEvents (...)
4 String weekID = getCurrentWeekID ()
5 Int replicas = put ($UserId_A$|| weekID, Calendar$_A$, “Cal”,
  defaultTS)
6 List <Object > res = get ($UserId_B$|| weekID, “Cal”,
  $UserId_B$, true)
```

7.2 Applicative case studies

Modern Online Social Networks are increasingly oriented toward cooperation and integration of different services. All the most popular OSNs are designed to host a customizable set of modules (e.g., Facebook applications) and are often adapted to interact with other OSNs (e.g. integration between Twitter and LinkedIn). A P2P storing and retrieval layer like a DHT is a very good decentralized framework to support this model, because, in principle, each application can access to any published resource, which implies a maximum integration potential.

However, in order that this potential could be profitably exploited, applications should be able to easily gather the correct information which constitutes the public profile of a user. To give a practical demonstration on how Likir API allows this task and to show how Likir primitives are easy to use, we consider two simple demonstrative applications.

First, we consider **LiCal** (**Li**kir **Cal**endar), a client that allows a user to publish her commitments and to consult the public events of her friends. Algorithm 1 shows that few code lines must be executed to startup a node (lines 1,2), publish user A’s weekly arranged events (lines 3-5) and to consult her friend B’s public events (line 6). We omit details about events’ structure and we suppose to deal with a time interval of one week, even if of course different time granularities can be chosen.

Quite differently to DHT-based file sharing clients (e.g. eMule), when the calendar client queries the DHT it is not interested in receiving a huge amount of results. If the weekly events of a known friend are looked up, only a specific entry, inserted by a definite identity is wanted; moreover, only the last calendar update is sought. Setting all the GET filtering parameters each index node return only one content (its most recent version), so the DHT data retrieval can be realized with

Algorithm 2: LiCha bootstrap

```
1 Object Options$_A$ = get ($UserId_A$||"options", "LiCha", $UserId_A$, true)
2 Int replicas = put ($UserId_A$||"contact", contact, "LiCha", defaultTS)
3 Object friend
4 for $contact_k$ in options.buddylist do
5     friend = get ($UserId_K$||"contact", "LiCha", $UserId_K$, true)
6     friendEvents = get ($UserId_K$|| weekID, "Cal", $UserId_K$, true)
```

an accuracy that is uncommon for classic DHT services and the application is relieved of *any* filtering task. This possibility is very handy to social network applications, which often need to access to data updates made by a single user (e.g. user status modification).

Even if quite embryonic, LiCal represents an example of a straightforward identity-based application. It shows how, in principle, synchronization problems afflicting ordinary calendar manager systems, can be solved using a reliable DHT approach. In fact, a LiCal client can be interfaced with a commonly used calendar client (e.g., Apple iCal, Microsoft Eudora), in order to access our data from different machines without a centralized provider (e.g., Google, Plaxo), and maintaining the control of events' confidentiality and privacy through a possible encryption.

As a second demonstrative example, we introduce **LiCha** (**Li**kir **Cha**t) that is a more mature social networking application we developed[4] using the Likir API. LiCha is a instant messaging client whose architecture is fully decentralized; the conventional central server that, in classic chat clients, retains all user's information is replaced with the DHT.

Two kind of contents are managed: the user *options*, containing the buddylist and other local user preferences, and the client *contact*, that is trivially a TCP socket address of the chat service and a status specification (offline/online). Algorithm 2 shows how (encrypted) local options are retrieved from the DHT (line 1) and how the client network contact is published (line 2). Friends' contacts are then retrieved (lines 3-5). Finally, the LiCha client pings each online friend to inform them of its status. When LiCha clients exchange contacts each others, they can simply start chat sessions without involving the Likir layer.

LiCal and LiCha can be profitably integrated. For example, the LiCha buddylist can be enriched displaying the daily events of those users that are also LiCal users. Such feature can be achieved with a single code line (line 6), supposing that the rules to build the correct LiCal lookup key are known. This basic example shows how any cross-application integration can be implemented in Likir; knowing the $UserId$ of a friend and the correct lookup keys production rules, a generic module can easily retrieve public information related to any other Likir application used by that friend.

[4] http://likir.di.unito.it/applications

8 Conclusions

Vulnerability to attacks suffered by overlay networks is a strong obstacle to the development of critical applications on DHTs. We designed an identity-aware version of Kademlia that offers an effective defense against a wide range of attacks, with a limited overhead. Even if a registration service is introduced, our architecture does not present a single point of failure, because the Certification Service is contacted only during the user subscription phase, through a simple web service. Furthermore, the presence of a web portal that every user is constrained to visit at least once can become a good point of aggregation for developers' products. If all the applications are listed in a website, the user can choose to install all the modules he likes in order to gain a customized application suite, like in popular services like iGoogle, whose applications are however managed in accordance with a client-server paradigm. If the presence of a centralized authority *must* be avoided, the CS could be easily replaced by efficient distributed PKI infrastructures like [25].

We showed how embedding identity at overlay level can be exploited also beyond security purposes. Developers can leverage the identity support to implement P2P reputation management systems and to build customizable applications suites that benefits identity sharing to collaborate each other. Furthermore, enhanced index side filtering functionalities allows to sharpen the data retrieval operation. Safe identity based data retrieval allows mash-ups between different applications through a very simple API.

Such features make Likir an ideal framework for P2P Social Networking Systems. In this context, Likir represents a point of contact between previous works on DHT security issues, that mainly focused on attacks, and the new trend in using pure P2P layers as privacy-aware frameworks for Online Social Networks, that often neglects security aspects laying at routing level.

The implementation of the Likir Java library, together with case studies and experimental results give credit to the feasibility of our proposal.

Acknowledgments

This work was produced in part within the “TeTraCo” project, with support of MIUR (“Progetti di ricerca e formazione ai sensi dell’art.13 del D.M. 593/00 - Distretto ICT Piemontese”).

We would like to thank the anonymous reviewers for their precious suggestions, that have been useful to improve the paper. A special thank to Mark Lillibridge, HP Senior Research Scientist, who gave us useful suggestions during the Eight International Conference on P2P Computing, Aachen 2008.

References

1. Abbas, S., Pouwelse, J., Epema, D., Sips, H.: A gossip-based distributed social networking system. In: WETICE’09: 18th IEEE International Workshops on Enabling Technologies. Groningen, Netherlands, pp. 93–98. IEEE Computer Society (June 29 - July 1, 2009)
2. Aiello, L.M., Milanesio, M., Ruffo, G., Schifanella, R.: Tempering Kademlia with a robust identity based system. In: P2P ’08: Proceedings of the 2008 Eighth International Conference on Peer-to-Peer Computing, pp. 30–39. IEEE Computer Society, Washington, DC, USA (2008). DOI http://dx.doi.org/10.1109/P2P.2008.40
3. Aiello, L.M., Ruffo, G.: Secure and flexible framework for decentralized social network services. In: SESOC ’10: Security and Social Networking Workshop, pp. 594–599. IEEE Computer Society (2010)
4. Dharanipragada Janakiram, J.: SyMon: Defending large structured p2p systems against sybil attack. In: P2P ’09: Proceedings of the 2009 Ninth International Conference on Peer-to-Peer Computing. IEEE Computer Society, Seattle, WA, USA (2009)
5. Baumgart, I., Mies, S.: S/Kademlia: A Practicable Approach Towards Secure Key-Based Routing. In: Proc. of P2P-NVE 2007 in conjunction with ICPADS 2007, Hsinchu, Taiwan, vol. 2 (2007). DOI 10.1109/ICPADS.2007.4447808
6. Bender, A., Sherwood, R., Monner, D., Goergen, N., Spring, N., Bhattacharjee, B.: Fighting spam with the NeighborhoodWatch DHT. In: INFOCOM (2009)
7. Bird, R., Gopal, I., Herzberg, A., Janson, P., Kutten, S., Molva, R., Yung, M.: Systematic design of a family of attack-resistant authentication protocols. Tech. rep., IBM Raleigh, Watson and Zurich Laboratories (April 1992)
8. Boneh, D., Franklin, M.: Identity-Based Encryption from the Weil Pairing. SIAM J. Comput. **32**(3), 586–615 (2003). DOI http://dx.doi.org/10.1137/S0097539701398521
9. Brunner, R.: A performance evaluation of the kad protocol. Master’s thesis, Institut Eurecom (2006)
10. Buchegger, S., Datta, A.: A Case for P2P Infrastructure for Social Networks - Opportunities and Challenges. In: WONS’09: 6th International Conference on Wireless On-demand Network Systems and Services. Snowbird, Utah, USA (2009)
11. Buchegger, S., Schiöberg, D., Vu, L.H., Datta, A.: PeerSoN: P2P Social Networking - Early Experiences and Insights. In: SNS’09: 2nd ACM Workshop on Social Network Systems Social Network Systems. Nürnberg, Germany (2009)
12. Castro, M., Druschel, P., Ganesh, A., Rowstron, A., Wallach, D.S.: Secure routing for structured peer-to-peer overlay networks. In: OSDI ’02: Proceedings of the 5th symposium on Operating systems design and implementation, pp. 299–314. ACM, New York, NY, USA (2002). DOI http://doi.acm.org/10.1145/1060289.1060317
13. Cheng, B.N., Yuksel, M., Kalyanaraman, S.: Virtual direction routing for overlay networks. In: P2P ’09: Proceedings of the 2009 Ninth International Conference on Peer-to-Peer Computing. IEEE Computer Society, Seattle, WA, USA (2009)
14. Cocks, C.: An Identity Based Encryption Scheme Based on Quadratic Residues. In: Proc. of the 8th IMA Int. Conf. on Cryptography and Coding, pp. 360–363. Springer-Verlag, London, UK (2001)
15. Condie, T., Kacholia, V., Sankararaman, S., Hellerstein, J.M., Maniatis, P.: Induced churn as shelter from routing-table poisoning. In: Proc. of NDSS 2006, San Diego, California, USA (2006)
16. Cutillo, L.A., Molva, R., Strufe, T.: Leveraging social links for trust and privacy in networks. In: INet Sec 2009. Open Research Problems in Network Security. Zurich, Switzerland (2009)
17. Douceur, J.: The sybil attack. In: Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS) (2002)
18. Ennan, Z., Ruichuan, C., Zhuhua, C., Long, Z., Huiping, S., Eng, K.L., Sihan, Q., Liyong, T., Zhong, C.: Virtual direction routing for overlay networks. In: P2P ’09: Proceedings of the 2009 Ninth International Conference on Peer-to-Peer Computing. IEEE Computer Society, Seattle, WA, USA (2009)
19. Gangishetti, R., Gorantla, M.C., A.Saxena: A survey on ID-based cryptographic primitives. cryptology eprint archive, report2005/094 (2005)
20. Guerraoui, R., Huguenin, K., Kermarrec, A.M., Monod, M.: On Tracking Freeriders in Gossip Protocols. In: P2P ’09: Proceedings of the 2009 Ninth International Conference on Peer-to-Peer Computing. IEEE Computer Society, Seattle, WA, USA (2009)
21. Iamnitchi, A., Ripeanu, M., Foster, I.: Small world file sharing communities. In: InfoCom ’04: Proceedings of the 23rd Conference of the IEEE Communications Society (2004). URL http://citeseer.ist.psu.edu/iamnitchi04smallworld.html
22. Josang, A., Ismail, R., Boyd, C.: A survey of trust and reputation systems for online service provision. Decision Support Systems **43**(2), 618–644 (2007)
23. Kamvar, S.D., Schlosser, M.T., Garcia-Molina, H.: The eigentrust algorithm for reputation management in p2p networks. In: WWW ’03: Proceedings of the 12th international conference on World Wide Web, pp. 640–651. ACM, New York, NY, USA (2003)
24. Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B.: Oceanstore: An architecture for global-scale persistent storage. pp. 190–201 (2000)
25. Lesueur, F., Me, L., Viet Triem Tong, V.: An efficient distributed pki for structured p2p networks. In: P2P ’09: Proceedings of the 2009 Ninth International Conference on Peer-to-Peer Computing. IEEE Computer Society, Seattle, WA, USA (2009)
26. Liang, J., Kumar, R., Xi, Y., Ross, K.: Pollution in p2p file sharing systems. In: INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE, pp. 1174–1185 (2005)
27. Liang, J., Naoumov, N., Ross, K.W.: The index poisoning attack in p2p file sharing systems. In: INFOCOM (2006)

28. Lou, X., Hwang, K.: Prevention of index-poisoning DDoS attacks in peer-to-peer file-sharing networks (2006). Submitted to IEEE Trans. on Multimedia, Special Issue on Content Storage and Delivery in P2P Networks
29. Lynn, B.: On the implementation of pairing-based cryptosystems. Ph.D. thesis, Stanford University (2007)
30. Maccari, L., Rosi, M., Fantacci, R., Chisci, L., Milanesio, M., Aiello, L.M.: Avoiding eclipse attacks on Kad/Kademlia: an identity based approach. In: ICC 2009 Communication and Information Systems Security Symposium, to appear. Dresden, Germany (2009)
31. Maymounkov, P., Mazières, D.: Kademlia: A peer-to-peer information system based on the XOR metric. In: IPTPS 2002, pp. 53–65 (2002)
32. Mislove, A., Post, A., Reis, C., Willmann, P., Druschel, P., Wallach, D.S., Bonnaire, X., Sens, P., Busca, J.M., Arantes-Bezerra, L.: POST: a secure, resilient, cooperative messaging system. In: HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems, pp. 11–11. USENIX Association, Berkeley, CA, USA (2003)
33. Naoumov, N., Ross, K.: Exploiting p2p systems for DDoS attacks. In: InfoScale '06: Proceedings of the 1st international conference on scalable information systems, p. 47. ACM, New York, NY, USA (2006)
34. Recordon, D., Reed, D.: Openid 2.0: a platform for user-centric identity management. In: DIM '06: Proceedings of the second ACM workshop on Digital identity management, pp. 11–16. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1179529.1179532
35. Ross, K., Liang, J., Naoumov, N.: Efficient blacklisting and pollution-level estimation in p2p file-sharing systems. In: Proc. of Asian Internet Engineering Conference (2005)
36. Rowaihy, H., Enck, W., McDaniel, P., Porta, T.L.: Limiting sybil attacks in structured peer-to-peer networks. Tech. Rep. NAS-TR-0017-2005, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA (2005)
37. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. LNCS **2218**, 329–351 (2001)
38. Rowstron, A., Kermarrec, A.M., Castro, M., Druschel, P.: Scribe: The design of a large-scale event notification infrastructure. In: Proc. of the Third International Workshop on Networked Group Communication (NGC 2001), pp. 30–43 (2001)
39. Ryu, S., Butler, K., Traynor, P., McDaniel, P.: Leveraging identity-based cryptography for node id assignment in structured p2p systems. In: Proc. of AINAW '07, pp. 519–524. IEEE Computer Society, Washington, DC, USA (2007). DOI http://dx.doi.org/10.1109/AINAW.2007.221
40. Shamir, A.: Identity based cryptosystems and signature schemes. In: CRYPTO 84: Proceedings of Advances in cryptology, pp. 47–53. Springer-Verlag New York, New York, NY, USA (1985)
41. Singh, A., Ngan, T.W., Druschel, P., Wallach, D.: Eclipse attacks on overlays: Threats and defenses. In: Proc. of the 25th IEEE InfoCom 2006. IEEE Computer Society, Barcelona, Spanien (2006)
42. Sit, E., Morris, R.: Security considerations for peer-to-peer distributed hash tables. In: IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems, pp. 261–269. Springer-Verlag, London, UK (2002)
43. Srivatsa, M., Xiong, L., Liu, L.: TrustGuard: countering vulnerabilities in reputation management for decentralized overlay networks. In: WWW '05: 14th international conference on World Wide Web, pp. 422–431 (2005). DOI http://doi.acm.org/10.1145/1060745.1060808
44. Steiner, M., En-Najjary, T., Biersack, E.W.: Exploiting KAD: possible uses and misuses. SIGCOMM Computer Communications Review **37**(5), 65–70 (2007)
45. Steiner, M., En-Najjary, T., Biersack, E.W.: A global view of KAD. In: IMC '07: Proc. of the 7th ACM SIGCOMM, pp. 117–122. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1298306.1298323
46. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 149–160. ACM, New York, NY, USA (2001). DOI http://doi.acm.org/10.1145/383059.383071
47. Urdaneta, G., Pierre, G., Van Steen, M.: A survey of DHT security techniques. ACM Computing Surveys (2009). http://www.globule.org/publi/SDST_acmcs2009.html
48. Wang, H., Zhu, Y., Hu, Y.: An efficient and secure peer-to-peer overlay network. In: LCN '05: Proceedings of the The IEEE Conference on Local Computer Networks, pp. 764–771. IEEE Computer Society, Washington, DC, USA (2005). DOI http://dx.doi.org/10.1109/LCN.2005.27
49. Wang, P., Osipkov, I., Hopper, N., Kim, Y.: Myrmic: Secure and robust dht routing. Tech. rep., DTC Research (2006)
50. Yu, H., Gibbons, P.B., Kaminsky, M., Xiao, F.: Sybillimit: A near-optimal social network defense against sybil attacks. In: Security and Privacy, 2008. SP 2008. IEEE Symposium on, pp. 3–17 (2008)