J Supercomput
DOI 10.1007/s11227-012-0842-2

# Fully distributed authentication with locality exploitation for the CoDiP2P peer-to-peer computing platform

**J.A.M. Naranjo · F. Cores · L.G. Casado · F. Guirado**



**Abstract** CodiP2P is a distributed platform for computation based on the peer-to-peer paradigm. This article presents a novel distributed authentication method that suits the platform and adapts to its characteristics. The developed method is based on the Web of Trust paradigm, i.e., not depending on a traditional PKI infrastructure, and focuses on efficiency both in the number of messages transmitted and digital signatures processed by exploiting the inherent locality found in the platform. As part of the method, a reliable and efficient distributed public key repository is developed taking CoDiP2P's de Bruijn topology as a cornerstone.

**Keywords** Peer-to-peer · Distributed computing · Authentication

## 1 Introduction

The spreading demand of intensive computational resources from different emerging research areas (life-science, fluid dynamics, molecular dynamic, weather simulation,

J.A.M. Naranjo (✉) · L.G. Casado
Department of Computer Architecture and Electronics, University of Almería, Agrifood Campus of International Excellence (ceiA3), Almería, Spain
e-mail: jmn843@ual.es

L.G. Casado
e-mail: leo@ual.es

F. Cores · F. Guirado
Department of Computer Science, University of Lleida, Lleida, Spain

F. Cores
e-mail: fcores@diei.udl.cat

F. Guirado
e-mail: f.guirado@diei.udl.cat

Published online: 05 December 2012

etc.) has required high investments in HPC (High-Performance Computing) infrastructures during the last decades. These infrastructures pay higher cost of ownership, not only for acquiring the hardware, but also for maintenance and administration costs via staffing, power infrastructure, cooling, and networking.

In spite of such efforts and technology advances, there are many applications that do not fit well in traditional HPC infrastructures due to their special features. For example, some massively parallel applications (like cryptography challenges or protein folding) require such huge volume of resources that can not be executed on a supercomputer (they would require the exclusive use of the infrastructure for months or even years). In these HTC (High-Throughput Computing) applications, the execution time is not so critical. Thus, the main factor is to aggregate enough computational resources for large periods of time to achieve the application's objectives.

Also, other types of applications (bioinformatics services, health care, etc.) will not require so much performance but instead a good availability and accessibility. Applications with response-time requirements cannot wait so much time in the job queue in order to gain access to the computer for their execution.

These specific requirements—high maintenance costs, large high throughput performance, and fast response—have motivated the development of new paradigms and architectures of distributed computing capable of integrating and sharing multiple computational resources from personal computers and providing them to the final users through Internet or by an existing grid platform.

The CoDiP2P [1] is a new computing platform designed following these premises. It is a decentralized architecture to distribute both the computation tasks and resource management among all participants (peers or nodes) using the P2P paradigm. CoDiP2P is based in the concept of harvesting idle resources (cycles and storage) from personal computers connected to the Internet. However, contrary to other voluntary computing systems (BOINC [2] for example), the gathered resources are shared among all the users of the system and everybody can collaborate and take profit from the system.

CoDiP2P is capable of providing high throughput computing to a wide range of applications. It provides great advantages: scalability, fault-tolerance, low-complexity applications, low-cost, and low footprint. However, all these benefits can be not enough if users are not guaranteed on the protection of their resources and applications. Thus, one of the main challenges on CodiP2P is to achieve the safety of users' resources. This involves authorization and authentication, tracking malicious peers behavior, keeping the confidentiality on both code and data and finally guaranteeing the invulnerability of the system. For all this, the goal of the present work is to describe privacy-related mechanisms that provide the aforementioned features. First, we introduce a novel distributed repository for public keys that exploits CoDiP2P's topology in order to obtain availability and efficiency. Second, we use an efficient distributed authentication method based on digital signature chains that eliminates the need for a verified trusted third party. As a consequence of the use of public key cryptography, we also achieve privacy in communications, message integrity, and precise peer tracking.

The article is organized as follows. Section 2 reviews the literature. Sections 3 introduces CoDiP2P and a review on peer-to-peer authentication. In Sect. 4, our pro-

posed authentication method is presented. Security is discussed in Sect. 5, and a comparison against a previous solution from the literature is shown in Sect. 6. Finally, Sect. 7 shows the conclusions and future lines of work.

## 2 Related work

The authentication problem in fully distributed networks (without a centralized trust model) is far from being solved. A full PKI (Public Key Infrastructure) approach is hard to implement due to the lack of a certification authority and, therefore, several research directions exist. Typically, distributed authentication solutions can be divided into partially distributed and fully distributed. The former category relies on one or more trusted third parties that provide authentication services to the nodes in the network. The latter does not assume the presence of those entities, hence the nodes themselves carry out the authentication task. Since we are addressing a fully distributed scenario, we will only review the latter category. We refer the reader to [3–5] for semidistributed authentication solutions.

Regarding fully distributed authentication solutions, and according to the nice survey in [6], one can find different approaches. Most of them use public key cryptography. Probably the most popular is the Web of Trust approach, in which a node A explicits its trust in another, B, by signing B's public key. Nodes trusting A can use A's public key to verify the signature. Based on this mechanism, signature chains can be built in order to express trust transitivity (if A trusts B and B trusts C then A trusts C). Some works that use this idea as a cornerstone are [7–9] and this very work. The work in [10] applies the idea in a hierarchical way for a MANET scenario. A different approach is followed by [11–13]: a poll is carried out regarding the authenticity of a peer's identity, and only if enough peers give an affirmative reply then the authentication is successfull. Finally, [3] combines the use of secret-sharing and trust graphs into a reliable and smart solution.

## 3 CoDiP2P architecture

CoDiP2P architecture is organized in logical areas, each one formed by a set of peers administered by a super-peer, namely a *manager*. Each manager controls and assigns the area's resources by applying the system policies.

The system is structured in a two-levels overlay, namely the *primary overlay*, to assure the system scalability and locality. All peers inside an area are connected with the manager, making up the first level of the overlay. The second level is formed by interconnecting managers by means of a tree topology (as shown in Fig. 1), however, other topologies can be used. By this, each peer has a number of $K$ neighbors (links), where $K \ll N$, with $N$ being the total number of online peers in the system.

Following this structure, all policies in the system are implemented hierarchically. They start with a request for peers that will be redirected to the corresponding manager as a first resort. If the manager cannot fulfill the request at all, then it is spread to neighboring managers and so on, until the request is completed. This hierarchical and

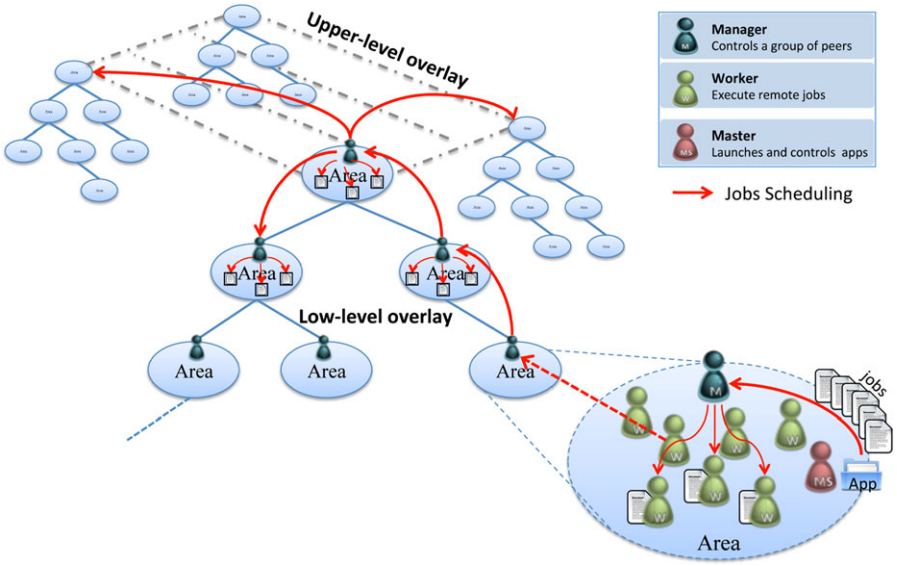


**Fig. 1** CoDiP2P Architecture and job scheduling

distributed design is applied to all system policies (like scheduling or authentication) in order to distribute over all the peers the management process and then avoiding bottlenecks.

Peers can have three different roles: *manager*, *worker,* and *master*. An additional responsibility of a manager ($M$) is to find workers for task execution. A worker ($W$) is responsible for executing assigned tasks. Finally, a master ($MS$) is any peer which submits the application for its execution by the system.

CoDiP2P is based on JAVA$^{TM}$ on both system and application development. Applications follow the master-worker paradigm: the master peer creates multiple *jobs* that are distributed among worker peers and executed in parallel. The master executes the main program, meanwhile workers execute jobs.

The CodiP2P architecture includes a logical structure based on a de Bruijn graph [14]. This allows to reduce the resources search time throughout the network. Thus, resources are grouped by locality [15], indicating the closeness of peers with similar resources, mapped into a multidimensional space by using the distances provided by the *de Bruijn overlay*.

The de Bruijn directed graph dimension is defined by $N = e^D$, where $e$ is the number of peer edges and $D$ is the diameter (maximum distance between any two peers). In a $d$-dimensional, $k$-base de Bruijn graph, each peer $i$ is connected to $k$ out-neighbors, being those peers whose de Bruijn indexes are $(ki + j) \bmod kd$, $j = 0, \ldots, k-1$. Thus, the topology is able to reach any peer in at most $\log_k N$ hops.

The main achievements of the CoDiP2P are: *scalability* by ensuring the massive entry and exit of peers; *distributed management* to harness the peers computing resources; *robustness* to deal with the high probability of a peer failure without affecting the working of the global system; *self-organization* to allow peers to change their role

dynamically according to the needs of the system; *heterogeneous resource* management in order to perform efficiently the scheduling and load balancing of jobs among peers.

## 4 The CoDiP2P authentification proposal

This section introduces the proposed distributed public keys repository and the routines associated. In this CA-less model, users express their trust in others by signing the others' public keys, and thus done by transitive trust. The use of this model in a peer-to-peer network deals with two problems: (i) distributed public keys storage and (ii) efficiency for a large number of peers.

The notation used for the rest of the paper is the following. $ID_X$ is the identifier for peer $X$. $(+K_X, -K_X)$ denote the public and private keys of peer $X$ respectively. SIGN($message, -K_X$) denotes the signature of *message* with the private key of $X$. The concatenation of data $a$ and $b$ is expressed by $a||b$. The number of online peers in the network is $N$, and $P$ the number of offline peers at a given time. Peer $Y$ is an *in-neighbor* of $X$ if there is a directed link from $Y$ to $X$. $X$ is an *out-neighbor* of $Y$ for the same reason.

### 4.1 Authentication of public keys: a distributed repository and locality exploitation

We take advantage of the de Bruijn topology to design the distributed repository of public keys in a DHT-like manner [18]. The peer index value $i$ can be obtained from the hash function $i = H(ID_X)$ that produces a binary value of $n$ bits with negligible collision probability for any peer identifier.

To adapt our distributed repository to the de Bruijn topology, each peer $i$ will store the public keys of her online de Bruijn out-neighbors, i.e., $(ki + j) \bmod kd$, $j = 0, \ldots, k-1$, plus the public keys of the offline nodes whose identifiers are closest. This makes the peer store $k + \frac{P}{N}$ public keys plus her own and allows to find any public key in the network in at most $(\log_k N) - 1$ hops. We also define function dist$(i, j)$, as the distance between peers with indexes $i$ and $j$. Any peer looking for a given public key can obtain it by following Algorithm 1.

Every peer signs the public keys of her out-neighbours providing transitive trust (see Sect. 4.2), which assures redundancy of a given key. Also, different instances of the same key are signed by different peers, allowing to authenticate any legal node $ID_X$ by recovering her public key from any of her in-neighbors.

#### *4.1.1 Exploiting locality*

The construction explained above allows for public key authentication in a logarithmic number of hops. However, CoDiP2P's primary overlay shows a strong sense of inherent locality that we have not exploited so far. For example, managers keep pools of workers formed by their neighbors. What is more, the manager will store the workers public keys she controls so masters can obtain them all from the manager rather than starting their own logarithmic searches. In a similar way, managers keep

**Algorithm 1** Public key search

*get_public_key*($ID_{current}$, $ID_{searched}$)

**INPUT:**
$ID_{current}$: current peer's identifier
$ID_{searched}$: searched peer's identifier

**OUTPUT:**
$+K_{searched}$: searched peer's public key

1. if $ID_{searched} \in out\_neighbours(ID_{current})$ return $+K_{searched}$
2. else
3. find $ID_{closest} \in out\_neighbours(ID_{current})$ \ with min(dist($H(ID_{closest})$, $H(ID_{searched})$))
4. return *get_public_key*($ID_{closest}$, $ID_{searched}$)

links among them. As a consequence, peers can share the public key information they handle to minimize the number of searches, while the latter remains as a backup repository in case the required public key is not found in the neighbourhood.

Peers keep a cache to store the authenticated public keys they know at a given time. Let us assume the size of that cache is $c$. A number of slots $k + \frac{P}{N}$ are used to store those keys allocated by the de Bruijn repository (recall $k$ is the number of out-neighbors in that topology). Also, as a manager, the peer will keep the public keys of her neighboring managers in the primary overlay. This set of keys, of size $mg$, will be frequently accessed during an online session and probably in consecutive sessions, hence it is not expected to change frequently. On the other hand, when the peer plays the role of master she will be given a number $w$ of workers that will carry out jobs for her. Due to the CoDiP2P worker search model the set of workers executing jobs for a given master is likely to change between different jobs, so this set of keys should be rotatory: these keys should be ready to be replaced in case a given maximum cache size is reached. Finally, when the peer acts as a worker she must request jobs from one or more masters at a given time. Therefore, she must store $ms$ keys from them: These keys are also good candidates to be replaced if new keys are obtained. The total number of keys is then $c = (k + \frac{P}{N}) + mg + w + ms$, from which those under $w$ and $ms$ should be the first keys to replace in case the storage limit is reached. The keys in $mg$ might vary between sessions, and the remaining $k$ should only change if the de Bruijn topology does. An efficient replacement policy for those under $w$ and $ms$ will minimize the number of cache misses (a simple approach would be to base on both age and number of reads). We leave the latter out of the scope of this article. Note that we do not take in consideration the space required for the distributed mirroring for the off-line peers public keys.

### 4.2 Bootstrapping, obtaining trust, and initial signatures

CoDiP2P uses a tracker host to allow new peers enter the system. Thus, the newcomer receives a list with her de Bruijn in-neighbors and out-neighbors, then each peer knows $k$ public keys while $k$ other peers store her public key. Additionally, a fraction of currently offline peers' public keys ($\frac{P}{N}$) is stored by every peer.

Before submitting the first job, any peer must obtain a minimum trust threshold from others that can be achieved by means of *tickets*: We define a *ticket* as a proof of having solved a previous job, and it is issued by a master including the signature of the (i) worker's public key, (ii) worker's identifier, (iii) master's identifier, (iv) absolute issue date and (v) a score mark from the worker's performance (the previous job difficulty and execution time, for example). Equation (1) shows a ticket $T_{Y\to X}$ issued by master $Y$ for worker $X$.

$$\begin{aligned} &T_{Y\to X} : ID_X, ID_Y, +K_X, +K_Y, date, score, \\ &SIGN\big(+K_X||ID_X||ID_Y||date||score, -K_Y\big) \end{aligned} \tag{1}$$

Masters should give workers correct tickets after completion of assigned jobs that the peers will handle to unknown managers or masters to prove her reliability. Tickets allow managers to evaluate the quality of any peer in order to decide whether to recommend her for a job. Clearly, a worker will always decide to send the best scored tickets but recall that an issue date has also been included, so it is up to the manager to decide on questions like the balance between a ticket's age and score, the maximum number of tickets accepted, or whether more than one ticket coming from the same issuer is accepted when evaluating the trust on the worker. We leave those questions open to future work.

### 4.3 Seeking workers

To launch any job, the master asks its manager for a given number of workers. If there are not enough available workers, then the manager redirects the request to another manager, and so on, until enough workers are found. There are three interesting facts: (i) the master only needs to know the first manager in the chain, (ii) the master will be contacted by workers previously unknown to her, and (iii) those workers will not possibly know the master either. Note therefore that the need for authentication can be found in both directions of the master $\leftrightarrow$ manager $\leftrightarrow$ worker relationship: not only the master needs to verify the workers, but also workers need to validate the source of the code they will be running. Moreover, the whole trust chain should be built in an efficient way, minimizing the messages sent and signatures to process.

The locality property can be exploited thanks to transitive trust. First, let us analyze how the master trusts workers. The first manager, say $M(1)$, gives each worker from her pool, say $W(1,i)$, the signature of the worker's public key with $M(1)$'s private key, i.e., $SIGN(+K_{W(1,i)}, -K_{M(1)})$. This signature can be handled later to the master (namely *MS*) by the worker in order to prove her relationship with $M(1)$. It is enough to do it just once, at bootstrapping. $M(1)$ can use the same approach when passing the job over to other managers: $M(1)$ gives $SIGN(+K_{M(2)}, -K_{M(1)})$ to $M(2)$, who will give it to worker $W(2,i)$ along with $SIGN(+K_{W(2,i)}, -K_{M(2)})$. This way workers provide the master with enough material to reconstruct the trust chain, which is extended in a similar manner until enough available workers are found.

A similar process might be applied to allow workers authenticate the master by adding $(+K_{MS}, SIGN(+K_{MS}, -K_{M(1)}), +K_{M(1)}, SIGN(+K_{M(1)}, -K_{M(2)}), \ldots)$ to the chain. However, the chain can be shortened if each manager directly signs the

**Algorithm 2** Authenticated worker search

*get_workers*($j$, *workers_needed*, *trust_chain*)

**INPUT:**
$j$: manager ordinal in the managers' trust chain: $1, 2, 3, \ldots$
*workers_needed*: number of workers to find
*trust_chain*: chain of public keys and signatures

**OUTPUT:**
Void.

```
1.  workers_hired = 0
2.  if authenticate_master(trust_chain) is not successful then return
3.  trust_chain = trust_chain − SIGN(+K_MS, −K_M(j−1))
4.  trust_chain = trust_chain ∪ SIGN(+K_MS, −K_M(j)) ∪ SIGN(+K_M(j+1), −K_M(j))
5.  while (workers_hired < workers_available ∥ workers_hired < workers_needed)
6.     W(j, i) = get_next_available_worker()
7.     send trust_chain to W(j, i)
8.     if W(j, i) accepts
9.        workers_hired + +
10.       workers_needed − −
11. if workers_needed == 0 then return
12. else
13.    get_workers(j + 1, workers_needed − workers_hired, trust_chain)
```

master's public key after authenticating her: $M(j)$ can give $(+K_{MS}, SIGN(+K_{MS}, -K_{M(j)}))$ to its pool workers and to the next manager, thus the length of the trust chain and verifications needed are reduced.

Algorithm 2 shows how managers build the chain. We assume that $M(j)$ already trusts $M(j-1)$ (or $MS$ if $M(1)$) and $M(j+1)$. We assume that $W(j,i)$ obtained $SIGN(+K_{W(j,i)}, -K_{M(j)})$ at bootstrap. The length of *trust_chain* grows linearly by the number of managers. In any case, the composition of the chain is efficient if $M(j)$ keeps $SIGN(+K_{M(\alpha)}, -K_{M(j)})$ stored for every neighboring manager $\alpha$.

We analyze the number of processed signatures needed for peer authentication in a workers search. Assume that $I$ is the average number of available peers in a manager's pool of workers, $J$ is the total number of managers involved in the search, $1 \leq i \leq I$ and $1 \leq j \leq J$.

The best case implies zero signatures processed since the peer might trust all other involved peers already. If managers keep their neighboring managers' public keys signed, then they can build the trust chain at zero cost. This is a real possibility if a worker executes more than one job for the same master. Let us now analyze the worst case in each role, which corresponds to an hypothetical situation in which peers know no public key other than theirs.

- Manager $j$ ($M(j)$) must verify one signature: $SIGN(+K_{MS}, -K_{M(j-1)})$. This is deleted from the trust chain when passed to the next manager. Additionally, she must generate $2+I$ signatures: one of the form $SIGN(+K_{MS}, -K_{M(j)})$ for authentication in the $W \rightarrow MS$ direction plus one of the form $SIGN(+K_{M(j+1)}, -K_{M(j)})$ and $I$ of the form $SIGN(+K_{W(i,j)}, -K_{M(j)})$ for the $MS \rightarrow W$ direction.
- Worker $j, i$ ($W(j,i)$) needs only to verify one signature: $SIGN(+K_{MS}, -K_{M(j)})$.

- The heaviest burden is taken by the master MS, who must verify $JI + J$ signatures: the whole trust chain ($J$ signatures) in the $MS \rightarrow W$ direction and every worker's signature by the corresponding manager ($JI$ signatures).

Worst case situations are not expected to occur since managers are trusted by their workers and neighbouring managers from bootstrap on. In any case, an additional signature can be added to assure the authenticity and integrity of every whole message.

## 5 Security related considerations

Two kinds of adversaries are commonly assumed in the security and privacy field: *semi-honest* and *malicious* [16]. The former follows the protocol correctly but exploits any available data to obtain knowledge about others. The latter deviates from the correct protocol and takes arbitrary actions in order to obtain private information or to carry out other kinds of attacks.

In a semihonest adversary scenario, our proposal achieves authentication, privacy, and data integrity thanks to the use of public-key cryptography, and allows additional features like peer tracing and statistical measures.

Distributed computing protocols commonly suffer from some well-known problems in a malicious adversary scenario. The first one, the *Sybil attack*, is always possible in a fully distributed environment as stated in [17], but can be partially alleviated with the use of tickets (recall Sect. 4.2). The second one is a denial-of-service attack: The adversary may refuse to provide a public key she stores in the de Bruijn topology, preventing its owner from being authenticated. Our proposal, however, largely mitigates this problem thanks to (1) key replication at the de Bruijn topology by providing different instances of the same public key from different sources, and (2) locality exploitation, hence authentication can also be carried out with the aid of already trusted neighbours. Third, malicious peers may provide deliberately wrong results, easily prevented if masters randomly issue test jobs (whose results are already known to the master) to not trusted peers. Fourth, *man-in-the-middle attacks* are prevented by the use of digital signature: Peers cannot be impersonated as long as their private keys remain safe and no collusion amongst peers exists. In order to successfully carry out such an attack the impersonating peer's de Bruijn in-neighbors and the impersonated peer's manager should collude.

## 6 Schemes comparison

Next, we compare our de Bruijn-based distributed repository of public keys with HDAM [7]. HDAM uses a chord-like [20] topology, which assures logarithmic performance both on searches and storage. HDAM does not explicitly use locality for performance enhancement, therefore, we are not considering it in the comparison for the sake of fairness. Another difference relies on the fact that HDAM stores public keys from online peers only, therefore, needing an external process for initial authentication. Instead, our proposal stores public keys from both online and offline peers.

**Table 1** Comparison between HDAM and the CoDiP2P distributed repository

| | HDAM [7] | de Bruijn |
|---|---|---|
| Storage/node, online keys only (in public keys) | $\log_2 N$ | $k$ |
| Storage/node, online + offline keys (in public keys) | – | $k + \frac{P}{N}$ |
| Public key search (in hops) | $(\log_2 N) - 1$ | $(\log_k N) - 1$ |
| Public key replication | $\log_2 N$ | $[k,\ k+1]$ |
| Nodes affected per join/leave | $\log_2 N$ | $k$ |

This allows for peer authentication without using external measures since only the legal peer can be the owner of the public key in the repository (assuming no theft of the private key).

Table 1 shows the comparison between both proposals ($P$ and $N$ are the number of offline and online peers, respectively). We show the number of keys stored per node in case that online keys only were considered: Our proposal would achieve a smaller storage requirement since a typical $k$ in CoDiP2P is lower than $\log_2 N$. If we consider online and offline keys stored, our proposal needs to store $k + \frac{P}{N}$ keys, which increases storage needs at the cost of avoiding external authentication mechanisms.

Regarding efficiency in searches, the diameter of a HDAM topology is $O(\log_2 N)$ while ours is $O(\log_k N)$, meaning that in a typical deployment of our topology the higher degree of our peers shortens the worst case search. On the other hand, key replication is also $O(\log_2 N)$, which is more (and better) than our $O(k)$. For the same reason, the number of peers affected by a join or leave event is lower in our case.

Next, we compare the performance of (1) HDAM, (2) our de Bruijn repository without locality exploitation shown in Table 1, and (3) our de Bruijn repository with cache for locality exploitation, namely CoDiP2P. To do so, we analytically model the number of peers/accesses that must be consulted in order to resolve a public key request.

As it was explained previously, the HDAM (Chord DHT) can be modeled as a binary search, being the number of accesses required to resolve a request $\log_2 N$ with $N$ the number of peers. On the other hand, the de Bruijn scheme depends on the peers connectivity overlay. Thus, if the maximum number of neighbors of any peer is $k$, the average number of accesses is $\log_k N$.

Our cache scheme corresponds to a $M$ level hierarchy, being the first level the own requesting peer, the second the peers in its own local area, and following levels, $2 < l < M$ those from remote areas located at distance $l$. The areas are connected by a tree topology, thus in a cache level the number of areas depends on the level and the nodal degree of the tree, $O$.

The model we apply to the CoDiP2P distributed cache behavior is based on the work present in [21]. Thus, the number of accesses to a distributed cache depends on the cache level that contains the requested public key $pk_i$.

$$\text{CoDiP2P} = \sum_{l=0}^{M} (P(L_i = l)(l+1)) + P(L_i = M+1) \times Bruijn_{cost} \tag{2}$$

In this equation, $L_i$ is the number of cache levels that a request for $pk_i$ has to query in the caching hierarchy before being satisfied. $P(L_i = l)$ corresponds to the probability that the request is solved at level $l$. Level $M+1$ determines the maximum level of the hierarchical cache. In the CoDiP2P architecture, if the request reaches this level it will be redirected to the de Bruijn DHT, adding the cost defined as the last term of (2).

$P(L_i = l)$ can be expressed as

$$P(L_i = l) = P(L_i \geq l) - P(L_i \geq l+1) \tag{3}$$

Note that $P(L_i \geq l)$ is the probability that the number of cache levels accessed to find $pk_i$ is equal to $l$ or higher. To calculate $P(L_i \geq l)$, let us denote $\tau$ the time within the interval $[0, \Delta]$ at which the request occurs. The random variable is uniformly distributed over the interval, thus we have

$$P(L_i \geq l) = \frac{1}{\Delta} \int_0^{\Delta} P\big(L_i \geq l \mid \tau\big)\, d\tau \tag{4}$$

where $P(L_i \geq l \mid \tau)$ is the probability that there is no request for $[0, \tau]$

$$P\big(L_i \geq l \mid \tau\big) = e^{-O^{l-1}\lambda_i \cdot \tau} \tag{5}$$

The value $\lambda_i$ is the average request rate for the $pk_i$ following a Poisson distribution. We assume that the $pk_i$ access probability follows a Zipf distribution. Finally, by combining (4) and (5), we get

$$P(L_i \geq l) = \frac{1}{O^{l-1}\lambda_i \cdot \Delta}\big(1 - e^{-O^{l-1}\lambda_i \cdot \Delta}\big) \tag{6}$$

In order to perform the experimentation CoDiP2P was set with a binary tree topology, $O \approx 3$, with $N = 10000$ peers, being the de Bruijn connectivity $k \approx 4$. The Zipf distribution used to model the access probability for each public key is parameterized by the skew factor $\alpha = 0.64$.

Figure 2 is focused on comparing the three different schemes by varying the number of public key requested per hour in the Zipfz distribution, $1 \leq \beta \leq 200$. The time period evaluated is $\Delta = 24$ hours, and we assume $M = 5$ as the maximum level on the hierarchical cache. As we can observe, the HDAM scheme requires $\log_2 N = 13.28$ accesses to peers in the DHT in order to get the public key. The number of accesses is reduced to $\log_k N = 6.64$ for the de Bruijn scheme, while CoDiP2P clearly performs better. We can appreciate two different behaviors of the latter. For a low request rate, $\leq$14 req/h, the majority of public keys miss the cache and forces to access the de Bruijn repository. This is because every next $pk_i$ request arrives after its cache replacement. On the other hand, when the $pk_i$ request rate increases, meaning a higher popularity, the number of cache hits grows, and thus the number of required accesses diminishes as can be observed in the figure.

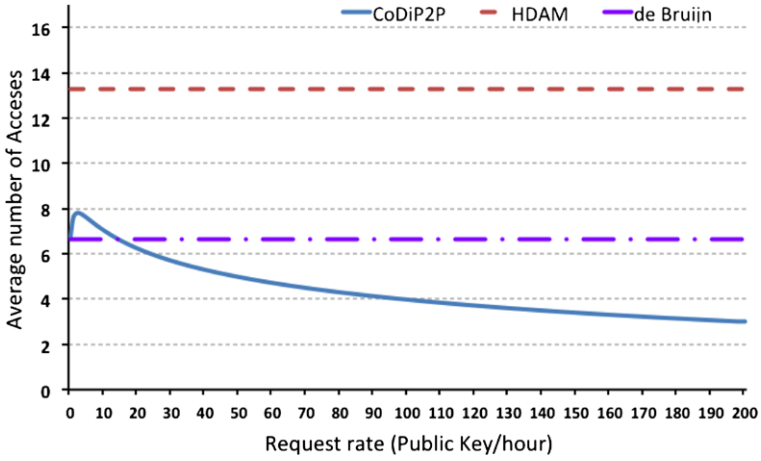


**Fig. 2** CoDiP2P architecture and job scheduling

## 7 Conclusions and future work

The present article introduces a fully distributed authentication system for CoDiP2P, a peer-to-peer computing platform. The proposed system is based on the Web of Trust paradigm and implements efficient mechanisms for transitive trust usage and public key search. First, the inherent locality found in CoDiP2P's topology is exploited in order to minimize the number of required public key searches per peer authentication. Second, the same locality property reduces the number of needed authentications (thus reducing the number of processed signatures too) since peers are likely to interact with already trusted neighbours. Third, a reliable and efficient distributed public key repository is built on top of CoDiP2P's de Bruijn topology as a backbone for public key search and authentication. The novel repository is compared against a previous solution from the literature.

Future lines of work include the development of a quantitative trust evaluation based on rewards per accomplished job, the study of replacement policies for the local public key cache of peers, and experimental evaluation.

**Acknowledgements** This work is supported by the CAPAP-H3 network (TIN2010-12011-E), the Spanish Ministry of Science and Innovation (TIN2008-01117 and TIN2011-28689-C02-02), and Junta de Andalucía (P11-TIC-7176), in part financed by the European Regional Development Fund (ERDF).

## References

1. Castell D, Barri I, Rius J, Giné F, Solsona F, Guirado F (2008) CoDiP2P: a peer-to-peer architecture for sharing computing resources. In: Int symp on distr computing and artificial intelligence 2008 (DCAI 2008). Advances in soft computing, vol 50, pp 293–303
2. Anderson DP (2004) BOINC: a system for public-resource computing and storage. In: Fifth IEEE/ACM international workshop on grid computing, pp 4–10
3. Omar M, Challal Y, Bouabdallah A (2009) Reliable and fully distributed trust model for mobile ad hoc networks. Comput Secur 28(3–4):199–214

4. Oh B, Lee S, Park H (2008) A peer mutual authentication method on super peer based peer-to-peer network. In: IEEE int symp on consumer electronics, pp 1–4
5. Josephson W, Sirer E, Schneider F (2005) Peer-to-peer authentication with a distributed single sign-on service. In: Peer-to-peer systems III. LNCS, vol 3259, pp 250–258
6. Li Z, Xu X, Shi L, Liu J, Liang C (2009) Authentication in peer-to-peer network: survey and research directions. In: International conference on network and system security, pp 115–122
7. Takeda A, Chakraborty D, Kitagata G, Hashimoto K, Shiratori N (2009) Proposal and performance evaluation of hash-based authentication for P2P network. Inf Media Technol 4(2):594–606
8. Takeda A, Hashimoto K, Kitagata G, Zabir SMS, Kinoshita T, Shiratori N (2008) A new authentication method with distributed hash table for p2p network. In: Proceedings of the 22nd international conference on advanced information networking and applications—workshops, AINAW'08, pp 483–488
9. Liu H, Luo P, Wang D (2008) A scalable authentication model based on public keys. J Netw Comput Appl 31(4):375–386
10. Ngai adn ECH, Lyu MR (2004) Trust- and clustering-based authentication services in mobile ad hoc networks. In: Proc 2nd international workshop on mobile distributed computing (MDC'04), pp 582–587
11. Zhang Y, Li X, Huai J, Liu Y (2005) Access control in peer-to-peer collaborative systems. In: 25th international conference on distributed computing systems workshop on mobility of peer-to-peer systems
12. Zhang Y, Li X, Huai J, Liu Y (2005) Access control in peer-to-peer collaborative systems. In: 25th IEEE int conf on distr computing systems workshops, pp 835–840
13. Aberer K, Datta A, Hauswirth M (2004) Efficient, self-contained handling of identity in peer-to-peer systems. IEEE Trans Knowl Data Eng 17:858–869
14. de Bruijn N (1946) A combinatorial problem. Proc K Ned Akad Wet 49:758–764
15. il Jeong K, Yoon U, Han J, Ahn J, Song J, Kim S (2008) Rnet: a hierarchical P2P overlay network for improving locality in a mobile environment. In: Int conf on networked computing and advanced information management, vol 1, pp 623–630
16. Canetti R (2000) Security and composition of multiparty cryptographic protocols. J Cryptol 13(1):143–202
17. Douceur J (2002) The sybil attack. In: First int workshop on p2p systems, pp 251–260
18. Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D (1997) Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In: Proc of the 29th ACM on theory of computing, pp 654–663
19. Dinger J, Waldhorst O (2009) Decentralized bootstrapping of p2p systems: a practical view. In: NETWORKING 2009. LNCS, vol 5550, pp 703–715
20. Stoica I, Morris R, Liben-Nowell D, Karger DR, Kaashoek MF, Dabek F, Balakrishnan H (2003) Chord: a scalable peer-to-peer lookup protocol for Internet applications. IEEE/ACM Trans Netw 11(1):17–32
21. Rodriguez P, Spanner C, Biersack EW (2001) Analysis of web caching architectures: hierarchical and distributed caching. IEEE/ACM Trans Netw 9(4):404–418