International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2016): 79.57 | Impact Factor (2015): 6.391

# A Secure Data Management in Distributing Cloud Computing

**T. Arunambika[1], P. S. Vijayalakshmi[2]**

[1,2]Assistant Professor, Department of Computer Science, Rathinam College of Arts and Science, Echanari, Coimbatore

**Abstract:** ***Penny uses a distributed reputation management system based on EigenTrust to securely manage data labels without the introduction of a central authority. The data labels empower requester peers to avoid downloads of low-integrity data, and allow sender peers to deny low privilege peers to access high confidentiality data. In addition, sender peers may publish and serve their data anonymously, frustrating attacks that seek to single out and target owners of security-relevant data. We have applied Penny to construct a secure, fully decentralized, data management system for traditional data les as well as Resource Description Framework (RDF) data. In this paper, present penny network architecture and their result and analysis.***

**Keywords:** Penny, Distributing Cloud Computing, RFD Data, EigenTrust, P2P.

## 1. Introduction

We have centralized master nodes in clouds that are trusted for integrity, in order to reuse the existing cloud infrastructure. We adopt a structured peer to peer (P2P) topology that eliminates centralized trust. All of the master nodes can act as peers and they distribute jobs and data between them. However in order to obtain that level of decentralization, we must abandon the existing cloud structure and develop a whole new protocol. Peer to peer (P2P) networking is a distributed, load-balancing computing paradigm de-signed to scalable share workloads between peers. Unlike traditional client-server models, each peer in a P2P network is an equally privileged, equipotent participant in the distributed computation or service. This has the advantage of avoiding centralized points of failure that, when successfully attacked, suffice to dismantle the entire network. P2P was first popularized as a vehicle for music sharing but has since expanded to general purpose file and data sharing applications and is increasingly important as a basis for fault tolerant cloud computing. Since its inception, it has been tremendously popular and ubiquitous because of its collective computation power, natural load balancing, and low cost deploy ability. For example, it has been estimated that Bit Torrent traffic accounts for roughly 27-55% of all Internet traffic (depending on geographical location) as of February 2009.

Penny is overlay and resource sharing protocol as a preliminary study without any implementation or experiments. We here extend that theoretical work with improvements to the architectural design, new formulas for computing data integrity and confidentiality labels, empirically determined optimal neighborhood sizes, new publish and request protocols adapted for RDF data, and other new empirically tested algorithms necessary for the system.

## 2. Penny Network Architecture

Penny is designed using a standard Chord ring, but with an extended form of reputation tracking: For each peer and data object, Penny allots k score manager peers and k key holder peers (respectively) to compute and track the peer or object's trust label(s). Parameter k is fixed at network start and controls the degree of replication; greater k means greater security, since attackers must compromise more peers to successfully corrupt data. Penny strategically positions responsibility sharing score managers and key managers at adjacent ring positions, forming a neighborhood. This greatly improves lookup efficiency over standard Chord, since only one overlay message (instead of k) suffices to contact all k replicas. The result is high replication (and therefore high security) with low overhead.

To protect data ownership privacy, data lookups in Penny employ a cryptographically protected extra level of indirection. Data serving peers first encrypt their requests with the public key of the data item's key holder, and then ask an arbitrary score manger to forward the server's key (not its real identifier) and encrypted information to the key holder. As a result, the key holder does not know who the real owner of the data item is, and so when someone later requests that data item, the key holder forwards the request back through the score manger(s). Meanwhile, the score mangers do not know which data items are owned by which peers, and thus learn no peer object associations as they forward the requests. As a result, the ownership information is concealed from all other parties.

### 2.1 Definitions

**Agents:** We refer to the peers in a P2P network as agents. Each agent a is assigned an identifier $id_a$ by applying a one way, deterministic hash function to its IP address and port number. We assume that identifiers are unique and that agents cannot influence which identifiers they are assigned. An agent's identifier determines its position in the network's ring structure. When agents are arranged in a ring, each agent has a predecessor pred(a) and a successor succ(a). We refer to the interval ($id_{pred(a)}$, $id_a$)as the identifier range of agent a.

**Objects and keys:** An object o is an atomic item of data (e.g., a file) shared over a P2P network. Each object also has a unique identifier $id_o$ obtained by applying a one way, deterministic hash function to its name. Objects can be owned by multiple agents. A single key is associated with

each object and each agent. The keys for object o and agent a are defined by $key_o = h(id_o)$ and $key_a = h(id_a)$ respectively, where h is a one way, deterministic hash function over the domain of identifiers.

**Key holders and Score managers:** Each agent $a_1$ is assigned a (not necessarily unique) key range, denoted $kr(a_1)$. Agent $a_1$ is charged with tracking the global integrity and confidentiality labels assigned to all objects o that satisfy $key_o \varepsilon kr(a_1)$. In addition, agent $a_1$ tracks the global trust values assigned to all agents $a_2$ satisfying $keya_2 \varepsilon kr(a_1)$. Whenever $key_o \varepsilon kr(a_1)$ holds, we refer to agent $a_1$ as a key holder for object o, and we refer to object o as a daughter object of agent $a_1$. Likewise, whenever $key_{a2} \varepsilon kr(a_1)$ holds we refer to $a_1$ as a score manager for agent $a_2$, and we refer to agent $a_2$ as a daughter agent of agent $a_1$. Every peer in a Penny network acts as both a key holder for some objects and a score-manager for some peers.

**Local confidentiality and integrity labels:** Each object o is labeled with a measure of its integrity and confidentiality levels. We denote the integrity and confidentiality labels assigned to object o by agent a as $i_a(o)$ and $c_a(o)$, respectively. Similarly, there are local integrity and confidentiality labels for agents with whom other agents had transactions. Integrity labels measure data quality; confidentiality labels measure that should be permitted to own the data. In Penny, confidentiality and integrity labels are modeled as real numbers from 0 to 1 inclusive, with 0 denoting lowest confidentiality and integrity and 1 denoting highest confidentiality and integrity.

**Local trust values:** Trust measures the belief that one agent has that another agent or object will behave as expected or promised. Each ordered pair of agents ($a_1$, $a_2$) has a local trust value denoted $t_{a1}$ ($a_2$) that measures the degree to which agent a1 trusts agent $a_2$. Likewise, each ordered pair of agent and object (a, o) has a local trust value denoted $t_a(o)$ that measures the degree to which agent a trusts object o. Like confidentiality and integrity labels, trust values range from 0 to 1 inclusive. Local integrity and confidentiality labels are computed and assigned based on local trust values.

**Global labels and trust values:** Each object o in the system is associated with global integrity and confidentiality labels, denoted $i_o$ and $c_o$, respectively, and measured by global trust values To. Likewise, each agent a is associated with global integrity and confidentiality labels, denoted $i_a$ and $c_a$, respectively, and measured by global trust values $T_a$. Key holders with a common key-range compute to and score managers with a common key range calculate $T_a$ using secure EigenTrust. Thus, the global labels and global trust values for any object o and for any agent a can be acquired by any agent in the network by contacting all key holders $a_{kh}$ for object o, and all score managers $a_{sm}$ for agent a.

### 2.2 Network Architecture

**Identifier Space and Neighborhood**

A Penny ring is like a Chord ring, with Penny's identifier ranges being equal to Chord's key ranges. However, a Penny agent's key range strictly subsumes its identifier range, and agent key ranges are not unique. Key ranges are assigned in a Penny ring so that for every agent a, there are between min(k, n) and c agents in the ring whose key ranges are equal to $k_r(a)$, where n is the total number of agents and c is a fixed bound on neighborhood size. Bounding neighborhood size from below by k limits the influence of malicious agents, because each contributes at most 1/k of the responses to a secure query. Bounding it from above by c ensures that lookup is not too costly, and it bounds the storage overhead for finger tables.

**Message Routing**

An agent can contact all score managers for a particular agent a, or all key-holders for a particular object o, using O(log N + k) messages. The first O(log N) messages propagate the message using the Chord algorithm to an agent whose identifier range includes $key_a$ or $key_o$, who then forwards it directly to the other O(k) agents in its neighborhood. Penny therefore reduces the overhead of all network operations that involve contacting key holders, score-managers, and RDF data owners by a factor of k over EigenTrust. This permits higher replication rates (e.g., k = 16) that are often infeasible with past approaches.

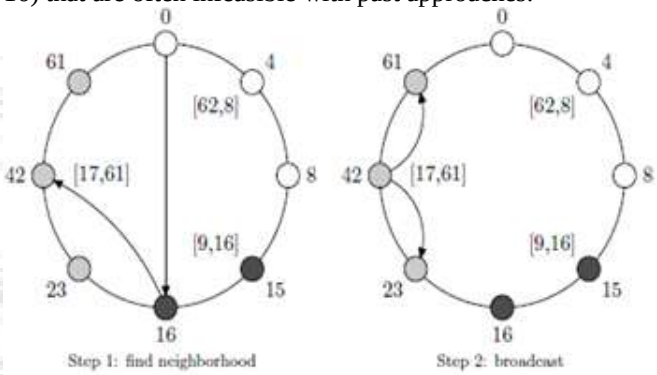


**Figure 1.1:** Penny message propagation

As in Chord, each agent a in a Penny ring maintains a finger table that is used to route messages efficiently. For each i ε (0, m), agent a's finger table includes the agent whose identifier range includes $(id_a + 2^i) \bmod 2^m$ (where $2^m$ is the size of the identifier space). In addition, agent a's finger table also includes an entry for each agent in its neighborhood. The size of each finger table is therefore O(m + k), where k is a constant dictating the number of redundant key holders assigned to each key.

Figure 1.1 shows an example of the propagation of a Penny message through the resulting ring. In this example, m = 6. Agent 0 wishes to send a message to all agents whose key range includes identifier 28. First, the message is propagated along the ring according to the Chord algorithm to the agent whose identifier range includes 28 (agent 42). This involves first sending the message to the agent whose identifier range includes $0 + 2^4 = 16$ (owner is agent 16), and next to the agent whose identifier range includes $16 + 2^3 = 24$ (owner is agent 42). Once the message reaches an agent whose key range includes 28, that agent forwards the message directly to all other agents in its neighborhood. These are all agents in the ring whose key-ranges include 28.

**Network Dynamics**

To maintain the invariant that the number of score managers for each key range stays between k and c, a Penny network

must occasionally split or merge neighborhoods as agents join and leave the network. If a peer join causes a neighborhood's population to rise above c, it splits into two smaller neighborhoods. Dually, if peer leave reduces a neighborhood's population below k, some or all peers from an adjacent neighborhood migrate in. When an agent anew joins a Penny ring, it is by default assigned a key range identical to its successor's. Its successor informs all agents in its neighborhood that they should update their finger tables to include $a_{new}$. However, if this would result in a neighborhood size b that is greater than c, a split occurs. The first (b/2) agents and the last b-(b/2) agents in the neighborhood each become their own neighborhoods. The key ranges of the new neighborhoods are the unions of the identifier ranges of the agents within each. Figure 1.2 illustrates a join operation with a split. Identifiers are labeled next to each agent outside the ring, and agent key ranges are labeled inside the ring. In this example, k = 2, c = 4, and m = 6, so when the agent with identifier 8 joins, key range [5, 42] has more than c agents and must be split.

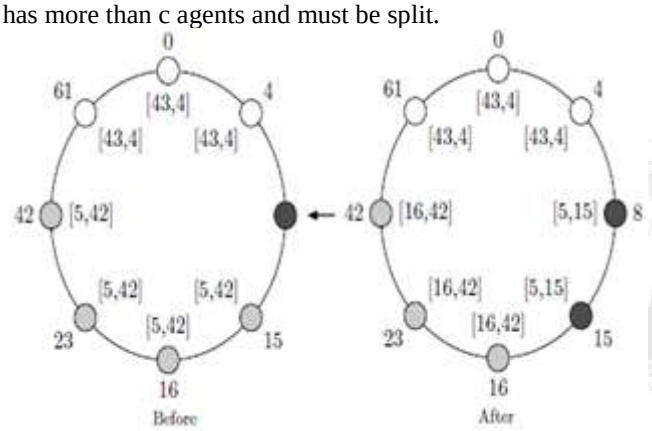


**Figure 1.2:** Agent joins operation

When an agent $a_{old}$ leaves a Penny ring, it informs its successor $a_{succ}$ and the other agents in $a_{old}$'s neighborhood. If $a_{succ}$ is in a different (adjacent) neighborhood, $a_{succ}$ informs the other agents in that neighborhood that the neighborhood's key range has grown to include identifiers up to and including $id_{pred}$ + 1 (where $a_{pred}$ is $a_{old}$'s predecessor). Likewise, agents in aold 's neighborhood must shrink their key ranges so that they end with $id_{pred}$. If the departure of $a_{old}$ causes $a_{old}$ 's neighborhood to have fewer than k members, two adjacent neighborhoods must be merged. Let Hold be $a_{old}$ 's neighborhood and $H_{pred}$ be the preceding neighborhood. If $|H_{old}| < k$ then the agent in $H_{old}$ whose predecessor is in $H_{pred}$ sends a merge request to its predecessor. That merge request is then forwarded to all agents in $H_{pred}$. If $|H_{pred}| < k+1$ then both neighborhoods merged to form a single neighborhood. Otherwise, the rightmost $|H_{pred}| - |H_{old}| / 2$ agents of neighborhood $H_{pred}$ join neighborhood $H_{old}$. The key ranges of the new neighborhoods are the unions of the identifier ranges of the agents in the new neighborhoods.

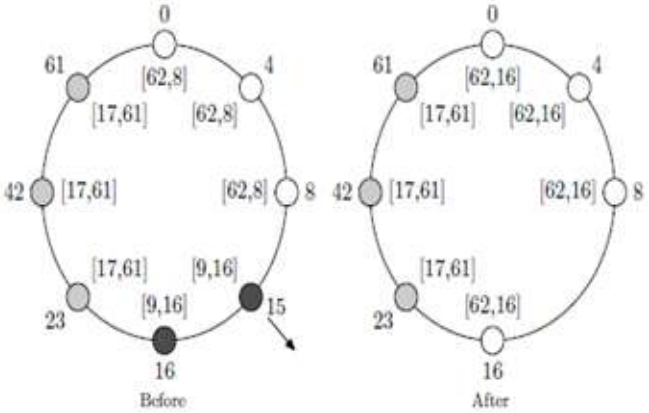


**Figure 1.3:** Agent leaves operation

Figure 1.3 illustrates an agent leave operation that requires a key range merge. Here, the departure of agent 15 from the ring leaves fewer than k = 2 agents in its neighborhood. Agent 16 therefore merges with its predecessor neighborhood; agents in both neighborhoods extend their key ranges to include the identifier ranges of all agents in the new neighborhood. Whenever an agent's key range shrinks due to any of the above operations, it must transfer any state associated with keys not in its new range to the appropriate key holders. Similarly, whenever its key-range grows, it receives state associated with new keys from the agents who previously occupied that range. An average net population change of 1/2 (c - k) agents per neighborhood is required before that neighborhood will need to be split or merged. Thus, by initializing c to be large relative to k, the frequency of these state transfer operations can be reduced.

**Agent's Local State**

In addition to routing messages, each agent a in a Penny network plays three different roles. It acts as a server when sharing objects, as a score manager for agents whose keys fall within its key-range, and as a key holder for objects whose keys fall within its key range. For each of these roles, it maintains some internal state:

- To act as server, it maintains a list of the identifiers $id_o$ of each object o that it owns.
- To act as score manager, it maintains a list of daughter agents ad that satisfy $key_{ad} \varepsilon k_r(a)$. These are the agents for whom agent a is a score manager. For each daughter agent $a_d$, it also maintains a vector of global trust values $T_{ad}$ with global integrity and confidentiality labels $i_{ad}$ and $c_{ad}$ respectively.
- To act as key holder, it maintains a list of daughter objects $o_d$ that satisfy $key_{od} \varepsilon k_r(a)$. These are the objects for which agent a is a key holder. For each daughter object $o_d$, it maintains a vector of global trust values $T_{od}$ with global integrity and confidentiality labels $i_{od}$ and $c_{od}$ respectively.
- For encrypted communication, it chooses a public key, private key pair ($K_a$, $K_a$).
- It maintains a list of the keys $key_{svr}$ and public keys $K_{svr}$ of the agents that serve object o. Thus key holders do not learn the actual identifiers of agents who serve object o, only their keys.
- It maintains local trust values $t_a(a_1)$ and $t_a(o)$ for agents $a_1$ and objects o with whom it had experience. These local trust values give rise to local integrity and confidentiality labels that agent a associates with $a_1$ and o.

**Publishing and Downloading Protocols for Traditional File Objects**

Once a Penny network has been initialized, agents interact according to the protocols detailed below. The protocol diagrams that follow use solid arrows to denote messages that are sent directly from agent to agent without using the P2P overlay, and dashed arrows for messages that use the P2P overlay to find the message target based on its ring identifier. Dashed arrows therefore actually involve sending O(log N + k) total messages. Arrows with double heads may optionally be sent via anonymizing tunnels for privacy. Notation $K_a$ denotes agent a's public key, and $< \ldots >_K$ denotes a message encrypted with key K.

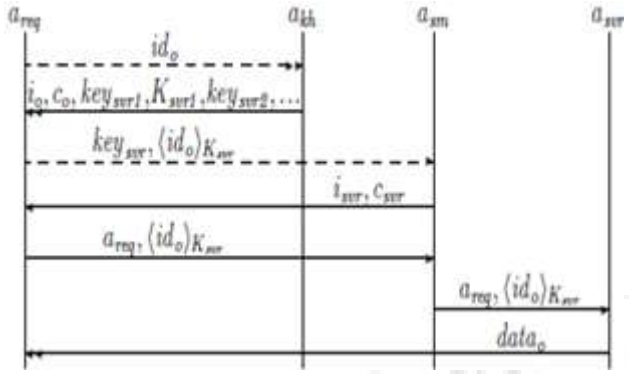


**Figure1.4:** Request protocol for traditional file object

When an agent $a_{svr}$ wishes to share an object o, it must first publish that object according to the protocol depicted. Agent $a_{svr}$ first obtains (possibly anonymously) the public keys of all key holders $a_{kh}$ for object o. Agent $a_{svr}$ next encrypts the object identifier and its own public key with each of the key holders public keys. It asks one of its score-managers $a_{sm}$ to forward the encrypted messages to the key holder's $a_{kh}$. Agent $a_{sm}$ conceals agent $a_{svr}$'s identity by sending only its key (which later can be used to get the global trust values and labels from the server's score-manger) to the key holder rather than its identifier, along with the encrypted message.

To request an object shown in Figure 1.4, requester $a_{req}$ first sends the requested object's identifier to all key holders $a_{kh}$ for the object. Each key-holder responds with the object's global integrity and confidentiality labels, and a list of the keys and public keys of servers who offer the object. Agent $a_{req}$ can then obtain the object from any server $a_{svr}$ by sending a request to all score-managers for agent $a_{svr}$. Score mangers reply to $a_{req}$ with the server's global trust labels. Based on a selection procedure $a_{req}$ then sends a download request message. In the message, the requested object's identifier is encrypted with the server's public key to avoid disclosing it to the selected server's score manager. The score managers forward the request to the server. The server can then anonymously send the data directly to the requester.

**Publishing and Downloading Protocols for RDF Datasets**

Besides traditional file lookup, Penny also supports RDF dataset queries. RDF triples are stored within file objects, but it would be prohibitively inefficient and insecure to store all triples within a single file owned by a single peer. Triples are therefore distributed over many smaller files distributed across many peers, with a protocol for locating each triple's containing file. Neighborhoods therefore collaborate to manage a subset of triples. Instead of keys for files, we associate triples with identifiers directly, and all neighborhood agents reply with list of servers who own the identified triples. This publishing procedure is detailed in Publish protocol for RDF data Algorithm. To distribute the load of serving particularly popular triples, each agent maintains a usage count for each triple it serves. When this count exceeds an agent imposed popularity threshold, it defers storage of future instances of that triple component to its successors in the ring. This implements a form of coalesced chaining in the distributed hash table.

RDF queries have syntax $(|?|s, |?|p, |?|o)$, where s is a subject, p is a predicate, and o is an object, and where each optional ? indicates an unknown in the query. For example, query $(s, p, ?o)$ requests all RDF triples satisfying subject s and predicate p. For downloading or querying RDF datasets over Penny network, agents implement Download protocol for RDF data Algorithm.

**Reputation based Trust Management**

Penny incorporates a reputation based trust management system based on EigenTrust. EigenTrust is a secure, distributed trust management system that maintains a globalized trust value for each agent. These globalized trust values are obtained by an iterative computation that approximates the left eigenvector v of the matrix T of all local trust values in the network. That is, if we define element $T_{ij}$ to be the degree to which agent $a_i$ trusts agent $a_j$, then the left eigenvector v of matrix T measures each agent a's global trust based on how much each agent trusts a, how much each agent trusts the agents who trust a, etc. If an agent $a_i$ downloads a file or RDF data from an agent $a_j$, it rates the transaction as positive or negative based on the experience. We may define local trust value $s(a_i, a_j)$ as the sum of these ratings of agent $a_j$ by agent $a_i$. Then, in order to aggregate the local trust values, they are normalized. We may define normalized local trust value, $c(a_i, a_j)$, as follows: Equation 1.1

$$c(a_i, a_j) = \frac{\max(s(a_i, a_j), 0)}{\sum_x \max(s(a_i, a_x), 0)}$$

This ensures that all values are between 0 and 1. These normalized local trust values are then aggregated. To keep the algorithm scalable and robust, eigenvector v is computed in a distributed and redundant fashion, where k different agents (score managers) are responsible for computing each element of v. This conforms to Secure EigenTrust except with global trust labels extended to objects as well as agents, and score manager replicas grouped into Penny neighborhoods for better performance rather than disbursed throughout the ring.

**Data Selection Procedure**

Every object request (whether a traditional file downloads or RDF query) delivers to requesting agent $a_{req}$ a set S of agents who can supply the object. If some respondents are malicious, some of these responses may differ. Agent $a_{req}$ must choose among them based on their reputations. To do so, it partitions S by response. Let R denote the resulting equivalence relation, so that quotient set S/R is the set of

agent groups, each of which returned a common response. For each partition P ε S/R we compute the following evaluation function.
Equation 1.2

$$f(P) = w_1 \frac{|P|}{|S|} + w_2 \frac{\sum_{a \in P} t(a)}{\sum_{a \in S} t(a)} + (1-w_1-w_2)\frac{1}{|S/R|}$$

**Publish Protocol for RDF Data Algorithm**
**Step-1:** For each RDF triple r do
**Step-2:** For each part (subject/predicate/object) $r_p$ of r do
**Step-3:** attempt ← 0
**Step-4:** while true do
**Step-5:** id succ(h($r_p$ + attempt))
**Step-6:** ask $a_{id}$ to store the triple r
**Step-7:** if $a_{id}$ already at popularity threshold then
$a_{id}$ refuses to store r
attempt ← attempt + 1
else
$a_{id}$ stores r
break
end if
end while
end for
end for

(Algorithm-1)

**Download Protocol for RDF Data Algorithm**
**Step-1:** For each sub query q in query Q do
**Step-2:** For each part (subject/predicate/object) $q_p$ in q do
**Step-3:** attempt ← 0
**Step-4:** while true do
**Step-5:** id ← succ(h($q_p$ + attempt))
**Step-6:** Request triples D from aid satisfying qp
**Step-7:** if $|D|$ < agent $a_{id}$'s popularity threshold then
break
else
attempt ← attempt + 1
end if
end while
end for
end for

(Algorithm-2)
where $w_1$ and $w_2$ are weights in [0, 1] that prioritize each partition's relative size and reputation, respectively, in the evaluation.

**Data Server Selection Procedure Algorithm**
**Step-1:** if all members of S have trust 0 then
Select one server from S randomly
else
if transaction is a police transaction then
$w_1$ ← 0
$w_2$ ← 0
**Step-2**: For each partition P ε S/R do
choose partition P with probability f(P )
end for
else
**Step-3:** $w_1$ ← 0.2
$w_2$ ←0.8
**Step-4:** B ← $arg\ max_{p \in S/R} f(P)$
**Step-5:** B ← $arg\ max_{p \in B} |P|$
**Step-6:** choose a partition randomly from set B
end if

(Algorithm-3)

Equation 1.2 is used by Data Server Selection Procedure Algorithm to resolve the selection choice. In the algorithm, police transactions are non-user transactions submitted by the security system during idle times in order to improve convergence.

## 3. Results and Analysis

We focus on four classes of attacks:

- A malicious agent or collective might spread corrupt or incorrect data. For example, the malicious agent or collective might spread malicious code or circulate false facts.
- A malicious agent or collective might attach incorrect security labels to data. In particular, low integrity data might be ascribed a high integrity label, or high confidentiality data might be described a low confidentiality label.
- A malicious agent or collective might attempt to learn which agents own certain data, perhaps as a prelude to staging additional attacks against those agents.
- A malicious agent or collective might attempt to generate a list of all data served by a particular agent, violating that agent's privacy.

We do not consider attacks upon the network overlay itself, such as message misrouting, message tampering, or denial of service attacks. These attacks are beyond the scope of this work, but could be addressed with various techniques, such as digital signatures, delivery receipts, and non-deterministic routing. Data Server Selection Procedure Algorithm makes the natural choice of preferring high over low reputation agents for user submitted requests. We discovered that this tends to cause Eigen Trust (and other reputation based trust management systems) to converge slowly because low reputation agents are so rarely exercised. We introduced a new form of transaction, called a police transaction that is designed to harmlessly exercise the system during idle periods rather than yield a correct result. Such transactions utilize low reputation agents, providing higher reputation agents additional opportunities to evaluate their answers. We used 50% police transactions.

For non-police transactions, we placed greatest weight on reputations ($w_2$ = 0.8) and the remaining weight on consensus size ($w_1$ = 0.2). We consider each 20 downloads as one frame and thus show the frame position over time with 1000 downloads. After each frame, we run the EigenTrust algorithm and compute global trust values accordingly. We run it 5 times and take the average success rate and we pessimistically assume that all malicious agents know the identities of all the pre-trusted agents, and that they must display high trust for those agents in order to

avoid lowering their own reputations. Thus, malicious agents trust only other malicious agents and pre-trusted agents. In our negative feedback, malicious agents always serve malicious files, and non-malicious agents who download the files always submit negative feedback for the transaction. Figure 1.5 shows that under these conditions, malicious agents fail to accrue high trust. Figure 1.5 (a) is for a static network with no leaves or joins, and Figure 1.5(b) is for a dynamic network undergoing constant churn. As expected, convergence is slower in the presence of dynamic activity; the static network converges at about frame 10, whereas the dynamic doesn't until about frame 20. For both, we get a very high average success rate 95.58% for the static network and 92.22% for the dynamic one, even with 20% malicious agents. Figure 1.6 records the results of our half correct behavior in which malicious agents provide correct les 50% of the time. Non-malicious agents always provide positive feedback for correct les and negative feedback for corrupt ones. Both static and dynamic networks converge quickly at approximately frames 14 and 24, respectively. Average success rates were also still very high 96.72% for the static network and 94.50% for the dynamic one. We further observe that the success rates are higher than each corresponding negative feedback since malicious agents provide correct les 50% of the time. On the other hand, convergence is slower because non-malicious agents take longer to identify the malicious agents.

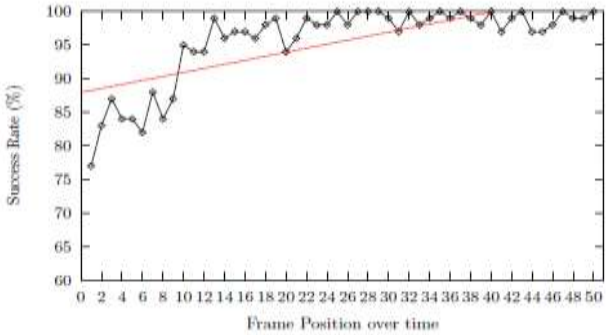


**Figure 1.5(a):** Static network

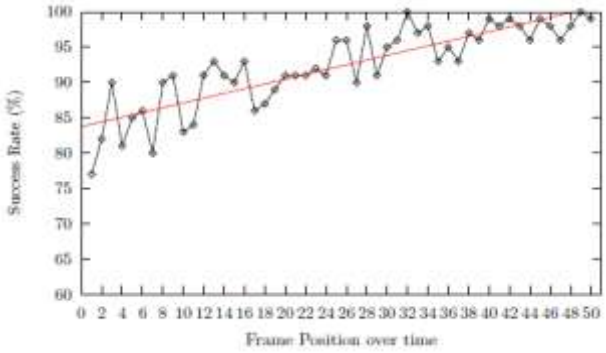


**Figure 1.5(b):** Dynamic network.
**Figure 1.5:** Negative feedback success rate.

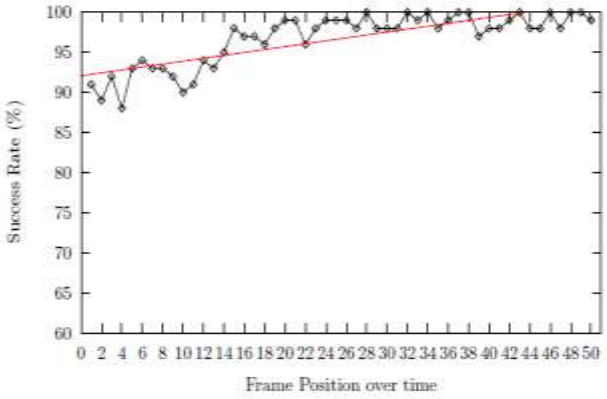


**Figure 1.6 (a):** Static network

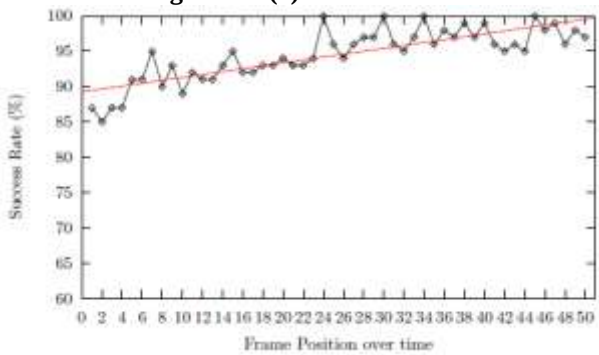


**Figure 1.6(b):** Dynamic network
**Figure 1.6:** Half correct behavior of success rates

Figure 1.6 records the results of our half correct behavior in which malicious agents provide correct files 50% of the time. Non-malicious agents always provide positive feedback for correct files and negative feedback for corrupt ones. Both static and dynamic networks converge quickly at approximately frames 14 and 24, respectively. Average success rates were also still very high: 96.72% for the static network and 94.50% for the dynamic one. We further observe that the success rates are higher than each corresponding negative feedback, since malicious agents provide correct files 50% of the time. On the other hand, convergence is slower because non-malicious agents take longer to identify the malicious agents.

Our malware propagation considers the pervasive problem of botnet malware infections of P2P file sharing networks. Non-malicious down loaders of malicious files have a 20% chance of becoming infected and exhibiting malicious behavior thereafter. Malicious agents behave the same as in the half-correct behavior experiment. In both static shown in Figure 1.7 (a) and dynamic shown in Figure 1.7(b) networks, success rates initially drop as previously high reputation agents suddenly attack the system. However, the reputation system adapts and around frame 16 the non-malicious agents manage to largely isolate the infection. The count of malicious agents continues to grow monotonically, as seen in Figures 1.8(a) and 1.8(b), because it includes no facility for disinfection. But the growth slows, and any new malicious agents are identified relatively quickly by the non-malicious majority. The average success rates were 93.04% for static networks and 90.44% for dynamic ones.

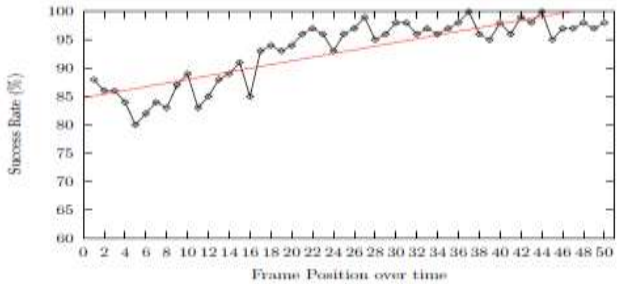


**Figure 1.7 (a):** Static network success rate

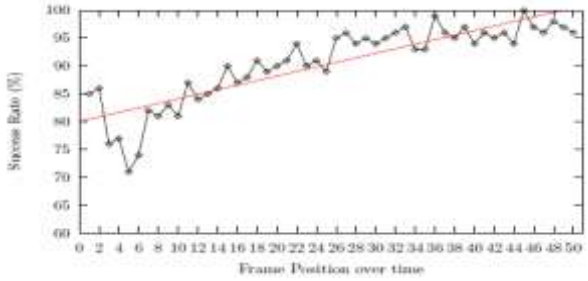


**Figure 1.7 (b):** Dynamic network, success rate
**Figure 1.7:** Malware propagation success rates

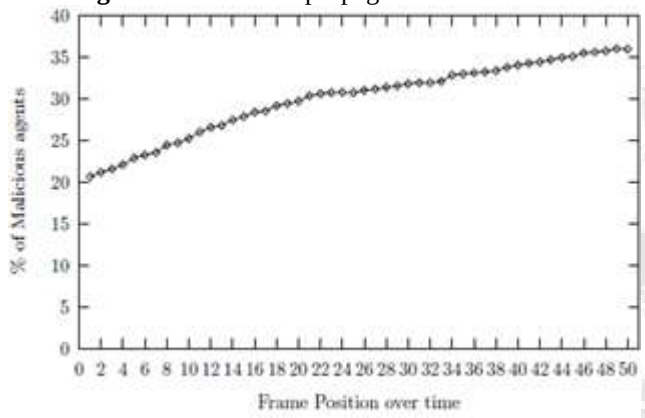


**Figure 1.8(a):** Static network propagation rate

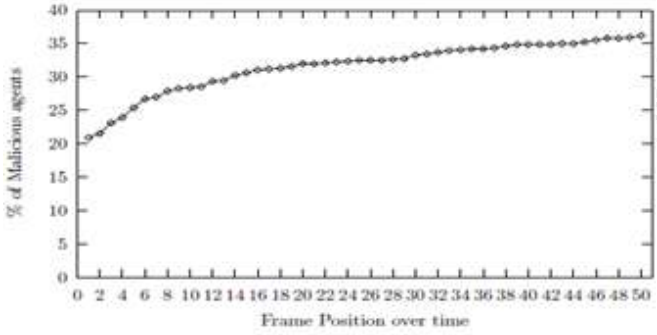


**Figure 1.8 (b):** Dynamic network propagation rates.
**Figure 1.8:** Malware propagation rates.

### 3.1 Results for RDF Datasets

We present the algorithmic results for RDF dataset download in the presence of malicious agents, and show the robustness of Penny networks. We publish protocol Algorithm-1 and download protocol Algorithm-2. We use the LUBM100 dataset for our algorithm which is broadly used by researchers for similar evaluations. The LUBM data generator yields datasets in RDF/XML format, which we converted to N-triples format. For download or query purposes, we use atomic triple queries and conjunctive multi predicate queries. We conduct the same three sets of algorithmic method for RDF datasets. For the negative feedback results shown in Figure 1.9 we see average success rates of 95.12% for static networks and 87.26% for dynamic ones. These are slightly lower than the corresponding rates for non-RDF file downloads because of the additional number of transactions required to successfully answer RDF queries. If any sub query fails, the entire query fails. In addition, the coalesced chaining implemented by Algorithm-2 requires additional transactions to retrieve popular triples. Convergence rates are slightly lower for the same reason. Despite this, both success rates and convergence rates remain quite high for a network with so much malicious population. The half-correct behavior experiment exhibits even faster convergence, as seen in Figure 1.10. The static network converges at about frame 15 and the dynamic at 25. Average success rates were similarly high at 96.46% and 92.78%, respectively. While malware is not possible in RDF data to our knowledge, for the sake of completeness we replicated the malware propagation experiment for the RDF publish and download protocol. Results are reported in Figures 1.11-1.12. Both static and dynamic networks exhibited fast convergence; about frame 19 for the static network and 29 for the dynamic one. Success rates were similarly promising, being 92.90% and 88.98% on average for the static and dynamic cases, respectively. Again, these are slightly lower than for file downloads because of the higher complexity of the RDF protocol. As before, both networks exhibit an initial drop in success but manage to adapt and recover fairly smoothly.

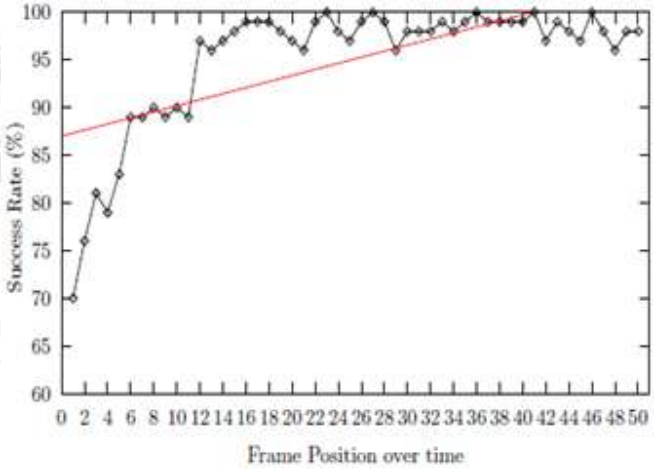


**Figure 1.9 (a):** Static Network

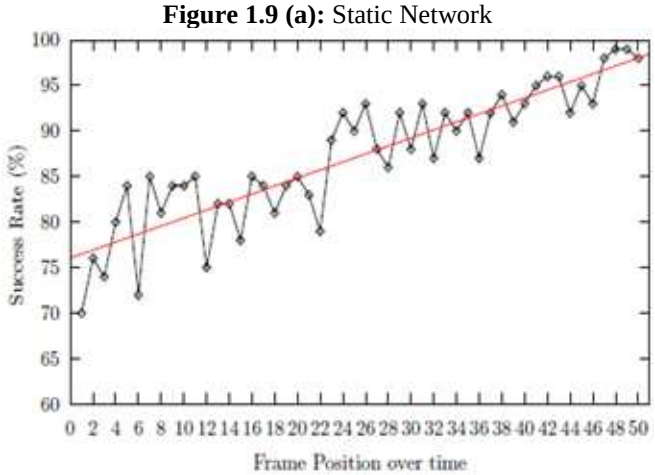


**Figure 1.9 (b):** Dynamic network
**Figure 1.9:** RDF negative feedback success rates

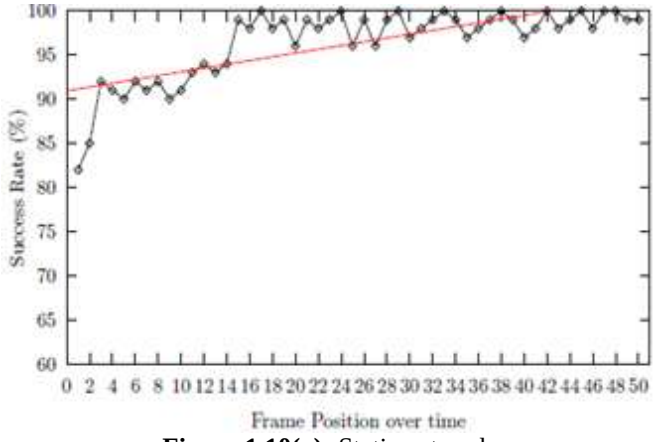


**Figure 1.10(a):** Static network

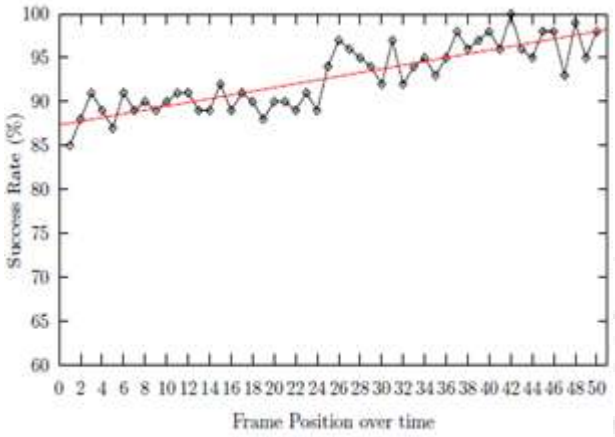


**Figure 1.10(b):** Dynamic network

Figure 1.10: RDF half correct behavior success rates

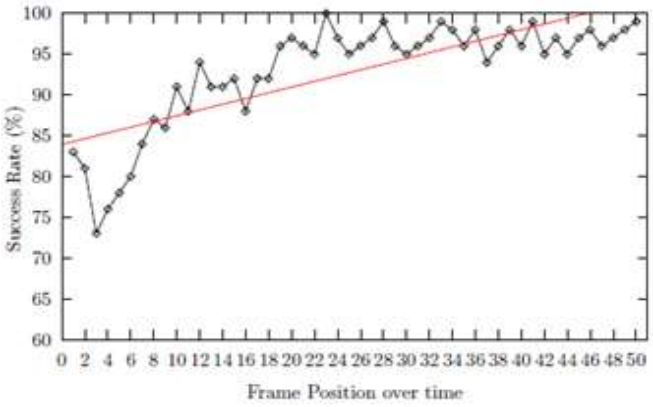


**Figure 1.11(a):** Static network, success rate

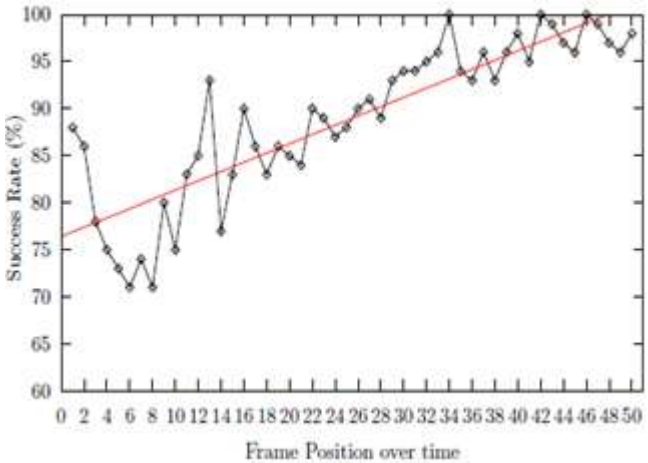


**Figure 1.11(b):** Dynamic network, success rate

**Figure 1.11:** RDF malware propagation success rates

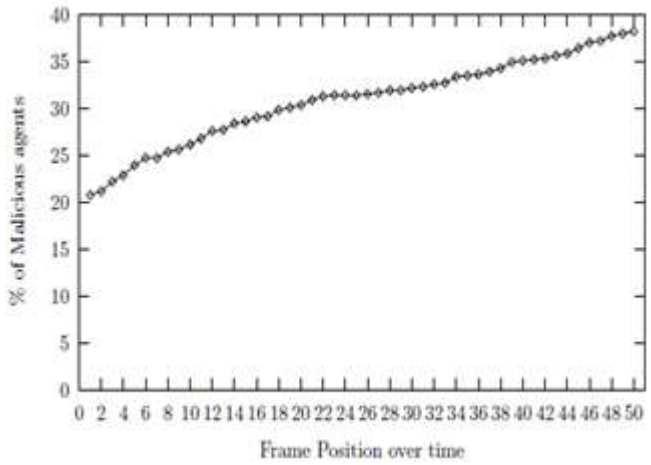


**Figure 1.12(a):** Static network, propagation rate

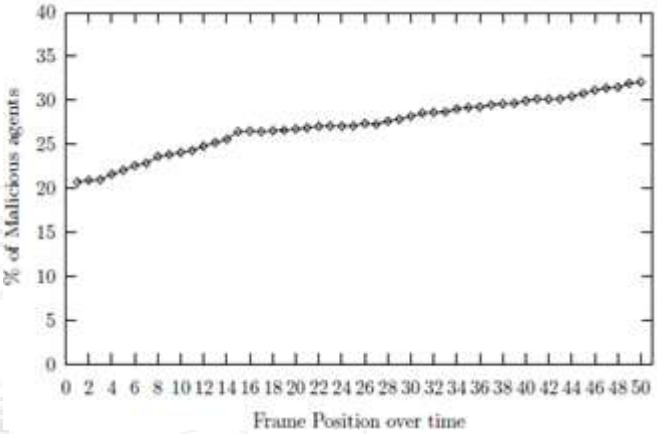


**Figure 1.12(b)** Dynamic network, propagation rate

Figure 1.12: RDF malware propagation rates

### 3.2 Analysis

The high success rates and strong convergence properties algorithmically observed and can be traced largely to Penny's support for exceptionally high data replication via its neighborhood topology. Label retrieval is efficient in Penny, requiring approximately the same number of messages as object lookup in a Chord network, but with k independent replicas of each label. An agent can retrieve any object's global integrity label by sending a single request message, which gets forwarded at most O(logN + k) times throughout the network. The request solicits O(k) response messages, from which one response is selected via Algorithm-3. Penny inhibits the spread of low-integrity data (e.g., malware) by maintaining a global integrity label for each object shared over the network. Agents wishing to avoid such data can therefore consult each object's global integrity label before downloading it. Thus, the problem of restraining the spread of malware over a Penny network reduces to the problem of efficiently maintaining and reporting accurate integrity labels. In addition to global integrity labels, Penny also maintains global confidentiality labels for objects. Agents can use these labels as a basis for selectively serving data to other agents possibly based on the requester's trust level, global confidentiality label, or other credentials.

An object's global security labels are determined by the votes of other agents in the network via EigenTrust. Votes

are weighted by the reputation of each voter so that the votes of agents who are widely regarded as trustworthy are more influential than the votes of those who are not. This makes it difficult for a malicious agent to attach a high integrity label to low integrity data. In order for such an attack to succeed, malicious agents must collectively have such good reputations that they outweigh the votes of all other voters. Penny uses EigenTrust to track agent reputations and to prevent malicious agents from accruing good reputations. Secure hashing and replication are both employed to protect against malicious key holders and score-managers who might falsify an object's global integrity labels or an agent's global trust value. Use of a secure hash function for identifier assignment ensures that agents cannot dictate the set of objects and agents for which they serve as key holders and score managers. By ensuring that there exist at least k key holders and score managers for every key range, Penny prevents any one agent from subverting the reputation of any object or agent. At least [b/2] agents in a neighborhood must be malicious in order to subvert a reputation, where b >= k is the neighborhood size. Malicious peers cannot elevate their own reputations by switching IP addresses or creating false network accounts because all agent and object reputations start at zero in Penny. An agent or object acquires a positive reputation only by participating in positive transactions with other agents. Agents with established reputations then report positive feedback for those transactions, elevating the new agent's reputation. Unlike Penny, Chord requires each key-holder to maintain a list of the agents who own the key-holder's daughter objects. These lists are reported to any agent who requests the object, divulging the identities of all agents who own a particular object To address this privacy vulnerability, Penny conceals information associating agents with the objects they own by splitting that information amongst key holders and score managers. A malicious key holder and a malicious score manager must therefore collaborate to learn that a particular server owns a particular object. Opportunities for such collaboration are limited because key holders and score-managers cannot choose their key ranges. It is therefore unlikely that a malicious collective will occupy both a key range that includes a particular victim object's key and a key range that includes a particular victim agent's key (assuming the collective is small relative to the size of the network). Thus, Penny enforces a notion of object ownership privacy.

Key holders and score managers can, of course, learn ownership information through guessing attacks, but this is prohibitively expensive when the space of object and agent identifiers is large. For example, a malicious agent am can discover whether a particular object o is served by any agent for which $a_m$ serves as score manager by requesting $id_o$ and comparing the key holders' responses against its list of daughter agents. However, $a_m$ cannot easily produce a list of all objects served by any of its daughter agents because to do so it would have to search the entire space of object identifiers. Likewise, am can discover whether a particular server $a_{svr}$ owns any object for which am serves as key holder. To do so, $a_m$ computes $key_{svr}$ and searches for that key in its list of keys of servers that own $a_m$'s daughter objects. However, am cannot easily produce a list of all servers that own any given object because it would have to search the entire space of server identifiers. So a large identifier space provides natural resistance to guessing attacks.

## 4. Conclusion

Penny decentralizes trust by distributing clouds master nodes trust among many peers. It efficiently supports global trust labels, data integrity labels, and data confidentiality labels in a fully decentralized, structured, peer to peer network. Global labeling assures convergence for all security queries, while decentralization avoids centralized points of failure typically associated with centralized label servers. Its reputation management system applies and extends EigenTrust, distributed hash tabling based on Chord, and anonymizing tunnels based on Tarzan or Sure Path. The security labeling scheme preserves the efficiency of network operations; lookup cost including label retrieval is O(logN + k), where N is the network size and k is a constant replication factor. The results illustrate Penny's efficiency and reliability over realistic network operations, including high dynamic churn; object publications, lookups, and downloads; and regular reputation maintenance via the Secure Eigen Trust algorithm.

## References

[1] Kamvar, S., M. Schlosser, and H. Garcia-Molina (2003). The EigenTrust algorithm for reputation management in P2P networks. In Proceedings of the 12th International World Wide Web Conference (WWW), pp. 640-651..

[2] Dingledine, R., N. Mathewson, and P. Syverson (2004). Tor: The second-generation onion router. In Proceedings of the 13th USENIX Security Symposium, pp. 303-320.

[3] Kamvar, S., M. Schlosser, and H. Garcia-Molina (2003). The EigenTrust algorithm for reputation management in P2P networks. In Proceedings of the 12th International World Wide Web Conference (WWW), pp. 640-651.

[4] Manuel, P. D., S. Thamarai Selvi, and M.-E. Barr (2009). Trust management system for grid and cloud resources. In Proceedings of the International Conference on Advanced Computing (ADCONS), pp. 176-181.

[5] Chen, D. and H. Zhao (2012). Data security and privacy protection issues in cloud computing. In Proceedings of the International Conference on Computer Science and Electronics Engineering (ICCSEE), pp. 647-651.

[6] Chow, R., P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina (2009). Controlling data in the cloud: Outsourcing computation without outsourcing control. In Proceedings of the ACM Workshop on Cloud Computing Security (CCSW), pp. 85-90.

[7] Necula, G. C. and P. Lee (1998). Safe, untrusted agents using proof-carrying code. In Proceedings of Mobile Agents and Security, pp. 61-91.

[8] F. Azzedin and M. Maheswaran, “A Trust Brokering System and Its Application to Resource Management

in Public Resource Grids", in Proceedings of IPDPS 2004.

[9] Marcos Dias de Assuncao: Provisioning Techniques and Policies to Enable Inter-Grid Resource Sharing, PhD Thesis, Melbourne University, 2009.

[10] Jemal H. Abawajy, Andrzej M. Goscinski: A Reputation-Based Grid Information Service. International Conference on Computational Science (4) 2006: 1015-1022.

[11] A. Jøsang, R. Ismail, and C. Boyd. A survey of trust and reputation systems for online service provision. Decision Support Systems, 43(2):618–644, March 2007.

[12] C. Catlett, P. Beckman, D. Skow, and I. Foster, "Creating and operating national-scale cyberinfrastructure services," Cyberinfrastructure Technology Watch Quarterly, vol. 2, no. 2, pp. 2–10, May 2006.

[13] C. Dellarocas. Immunizing Online Reputation Reporting Systems Against Unfair Ratings and Discriminatory Behavior. In ACM Conf

[14] Jemal Abawajy, Determining Service Trustworthiness in Intercloud Computing Environments, Proceedings of the 10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN '09), pp.784~788, 2009.

[15] B. Yu, M. P. Singh, and K. Sycara, "Developing trust in large-scale peer-to-peer systems", 2004, pp. 1-10.4-260, 1999.