Zghaibeh and Ul Hassan *EURASIP Journal on Wireless Communications and Networking* (2017) 2017:161
DOI 10.1186/s13638-017-0942-z

EURASIP Journal on Wireless Communications and Networking

RESEARCH Open Access

SHAM: Scalable Homogeneous Addressing Mechanism for structured P2P networks

Manaf Zghaibeh* and Najam Ul Hassan

Abstract

In designing structured P2P networks, scalability, resilience, and load balancing are features that are needed to be handled meticulously. The P2P overlay has to handle large scale of nodes while maintaining minimized path lengths in performing lookups. It has also to be resilient to nodes' failure and be able to distribute the load uniformly over its participant. In this paper, we introduce SHAM: a *S*calable, *H*omogenous, *A*ddressing *M*echanism for structured P2P networks. SHAM is a multi-dimensional overlay that places nodes in the network based on geometric addressing and maps keys onto values using consistent hashing.
Our simulation results show that SHAM locates keys in the network efficiently, is highly resilient to major nodes' failure, and has an effective load balancing property. Furthermore, unlike other DHTs and due to its distinguished naming scheme, SHAM deploys homogenous addressing which drastically reduces latency in the underlying network.

Keywords: P2P, Structured, Performance, DHT, Overlay, Latency

1 Introduction

Locating keys efficiently in structured P2P networks, commonly referred to as *distributed hash tables* (DHTs), is always associated with the variant of maintaining limited routing tables at nodes. Some research adopted and worked on the principle that in order to lower the cost of search, the node has to acquire the most information available from the overlay [1, 2]. Thus, the more routing information the node gathers, the less the lookup will take. From another angle to this argument, other research insisted on keeping the size of the routing tables as minimum as possible while sacrificing some of the lookup performance [3–6]. The point here is that maintaining large routing tables is not viable and, thus, giving up few additional hops to reach the destination is more satisfying. Proposed DHTs differ in this regard and are classified based on the number of hops they require in order to land a lookup at its destination. We categorize the classes in this respect into single-hop overlays, constant degree overlays, multi-dimension overlays, and logarithmic overlays.

*Correspondence: mzghaibeh@du.edu.om
Department of Electrical and Computer Engineering, Dhofar University, P. O. Box 2509, 211, Salalah, Oman

In single-hop overlays, a destination can be reached within one hop from the source node while maintaining a global knowledge about other nodes in the overlay [1, 2, 7–11]. However, although with the assertion that space requirements are cheap and bandwidth is abundant, still, keeping up with the continuous change in churn-intensive overlays is unfeasible.

Constant degree overlays on the other hand aim to reduce the global knowledge to a certain limit while increasing the latency from one single hop to a *fixed* number of hops regardless of the number of nodes in the system. Nevertheless, such DHTs are still not suitable for large networks where maintaining the routing table remains a challenge [12–14].

In multi-dimension overlays [4] and logarithmic overlays [3, 5, 6], neither the size of the routing table nor the latency is fixed. These systems work on balancing the two constraints based on the number of participating nodes: They tend to optimize the size of the routing table in order to reach a reduced latency. Such systems are known for their flexibility, scalability, and resilience to nodes' failure.

In similar direction, our motivation behind the work in this paper is to propose an addressing algorithm for

© The Author(s). 2017 **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

structured P2P networks that can balance between the latency and the size of a routing table. In addition to that, our goal for the mechanism is to be robust and resilient against nodes' failures, to be able to scale without sacrificing performance, and to have an effective load balancing property. We propose *SHAM*: a Scalable Homogeneous Addressing Mechanism for structured P2P networks. SHAM handles three preeminent procedures. First, it inserts and situates nodes into the overlay. Second, it maps keys onto available nodes in the overlay. And third, it retrieves the keys. Similar to other structured P2P systems [3, 4, 15], SHAM employs consistent hashing to generate a naming space for keys. The use of consistent hashing is known to balance the load since each node in the system will theoretically have the same number of keys [3]. However, unlike other systems in the same class, SHAM does not require hashing the identifier of the node (*IP*/*port*) in order to place it in the system. Yet, it deterministically places the node in the system and assigns it a unique identifier based on homogeneous geometric pattern.

We will show in this paper by matching SHAM up with the two prominent DHTs, CAN and Chord, the following:

1. SHAM has better routing capabilities than CAN in all dimensions. Furthermore, by adjusting its dimensions, SHAM has a shorter path length than Chord up to a certain network sizes with minimized routing tables.
2. Unlike Chord, load balancing is always maintained in SHAM without additional cost. This is an issue that has always been neglected in discussing Chord: Although Chord maintains entries for $O(\log N)$ nodes, however, the use of its load balancing property is associated with each node holding $\log^2$ N entries.
3. The addressing scheme adopted in SHAM enables the overlay to be used to reduce latency in the underlying network. This is not applicable in Chord since placing nodes in the overlay is the responsibility of the consistent hashing. In SHAM, however, nodes can be placed in the overlay based on their actual geographical location. Thus, similar to load balancing, latency in underlying network is another limitation in Chord that SHAM overcomes.

Lookups in SHAM are bounded to $O(N^{1/d})$ where d is the number of dimensions in the system. While node insertion affects $\left(3^d - 1\right)$ other nodes, each node in SHAM maintains $2\left(3^d - 1\right)$ routing entries.

The paper is organized in the following manner: Section 2 presents the design of SHAM. Results and discussion are presented in Section 3. The review of related work is in Section 4. And we conclude our work in Section 5.

2 Design

SHAM is created and maintained using three procedures. The node joining, the key inserting, and the key locating procedures. In this section, we describe its design and clarify the above mentioned procedures in detail.

In its core, SHAM is a self-managing structured P2P network built on a d-dimensional *hexadecimal* coordinate space to host nodes; see Figure 1. It consists of nodes that organize themselves into an overlay. Each node in SHAM is able to (1) receive and forward routing requests, (2) host and locate keys, (3) situate newcomers in the network.

Nodes in a structured P2P overlay must be assigned unique identifiers to indicate their locations in the network: *nodeIds*. This is achieved in many systems by hashing the IP address of the node using a consistent hashing function to generate its identifier. Nodes are then ordered in the network based on their nodeIds. For example, in Chord, the nodes are placed orderly on an identifier circle modulo 2^m, where m is a system parameter [3]. In

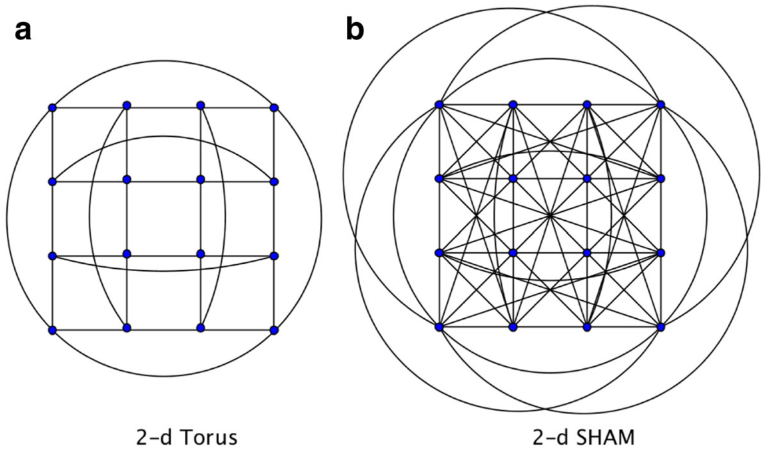

Fig. 1 a 2-d torus. **b** 2-d SHAM

like manner, each node in SHAM is assigned a unique identifier. However, a node's identifier in SHAM is completely independent of its IP address and is deterministically selected from a 2^m space. The design eliminates the consistent hashing phase by directly assigning an identifier to each node joining the system based on the *pre-constructed* geometry of the overlay. Thus, SHAM situates the node first in the overlay based on geometric addressing, then it concludes its nodeId from the address of its position. To accomplish that, we devise that the location of a node in our system is a d-tuple $\left(X^1, .., X^d\right)$, such that $X^d = \left(x^d_{r_d-1}..x^d_1x^d_0\right)$ is the d coordinate in the system and r_d is a system distribution parameter concerned with the topology of the overlay, where for a system with 2^m nodes, we have $m = \sum_{i=1}^{d} r_i$. Moreover, for any given node, $X^1, .., X^d$ bear hexadecimal representation of the location of the node in the system. Accordingly, the nodeId for node u in SHAM is simply the concatenation of its coordinates from X^1 to X^d to form the string $\left(X^1X^2..X^d\right)$. Throughout this paper, we shall use the term *nodeId* to refer to both the node's location and identifier.

Since addressing in SHAM is sequential, nodes that share the same coordinate are labeled in sequence in the other coordinates. The identifier space suggests addressing wraps around on each coordinate. Thus, we perceive the addressing as rings on regular coordinates $X^1, X^2, ..X^d$ and diagonal coordinates $X^1X^2, ..X^{d-1}X^d$.

Figure 2 illustrates the hexadecimal addressing in a 16-node 2-d system using two rules. The first rule is when $r_1 = r_2 = 2$, $X^1 = \left(x^1_1x^1_0\right)$, and $X^2 = \left(x^2_1x^2_0\right)$, where $x^1_0 = 0$, x^1_1 range is $\{1-4\}$, $x^2_1 = 1$, and the range of x^2_0 is $\{A-D\}$. The second rule is when $r_1 = 2$, $r_2 = 3$, $X^1 = \left(x^1_1x^1_0\right)$, and $X^2 = \left(x^2_2x^2_1x^2_0\right)$, where $x^1_1 = 0$, the range of x^1_0 is $\{1-4\}$, $x^2_2 = 1$, $x^2_1 = 1$, and the range of x^2_0 is $\{A-D\}$.

Figure 3 presents an example of addressing in a 3-d SHAM system with $r_1 = r_2 = r_3 = 2$ with a node *13F9BC* having its complete set of direct neighbors. The address of the node will be translated onto coordinates: $X^1 = 13$, $X^2 = \text{F9}$, and $X^3 = \text{BC}$.

2.1 Routing tables

In a d-dimensional fully occupied SHAM system, every node has $(3^d - 1)$ direct neighbors it must be aware of. Strictly speaking, a direct neighbor of node u is sequential to u in addressing in a specific direction. Thus, the routing table of any node has $\left(3^d - 1\right)$ permanent entries for those neighbors. Each entry includes the nodeId and the IP address of the neighbor, the nodeId and IP address of the successor of that neighbor in the same direction, a *timeout* counter, and the weights of the paths of the node in every direction. The timeout counter is used to measure the connectivity and the availability of a neighbor. While the path weight of a node in a specific direction, or simply the *weight*, represents the number of consecutive adjacent successors of the node in that direction. The path weights are maintained by the node to realize the growth of the overlay on all direction. If a direct neighbor has not been placed yet, or it has left the system, a *Null* value is entered in its relative position in the routing table. Figure 4 shows an example of path weight measurement.

The weight counters are also used in gap detection. The *gap* notion is used in the remaining of this text to indicate an unfilled zone in the overlay caused by the departure of one or more nodes. This is the only way a gap is created in SHAM. Nodes monitor the change of the weights of their

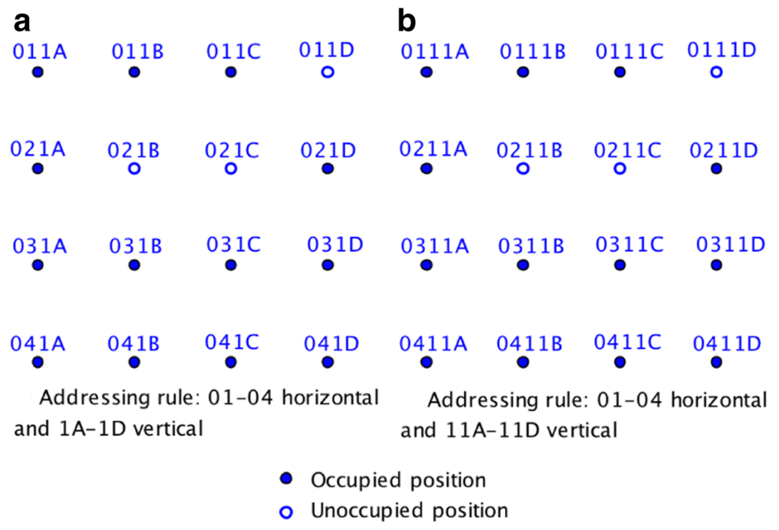

Fig. 2 A 16-node 2-d system with two different addressing schemes. **a** Addressing rule: 01-04 horizontal and 1A-1D vertical. **b** Addressing rule: 01-04 horizontal and 11A-11D vertical

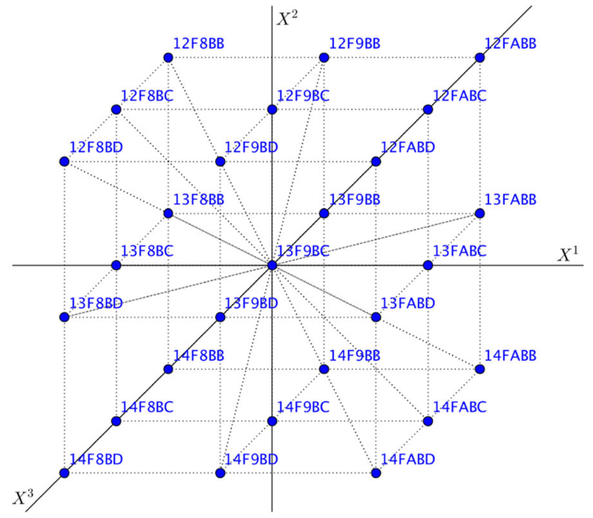

Fig. 3 Hexadecimal addressing in a 3-d SHAM system. Node 13F9BC has 26 direct neighbors organized in sequential order on each coordinate

neighbors continuously to detect the formation of gaps. For instance, say that node b observes that the weight of its direct neighbor c has decreased after sometime; b then deduces that one of the nodes down the direction of c has left the system.

To entrench routing in SHAM, the node's routing table includes temporary entries for *remote* neighbors. A node's remote neighbor in a direction d is not sequential in addressing to that node, yet it is the first available node in that direction. The sole purpose of remote neighbors is to facilitate routing. Their entries are created in case a direct neighbor does not exist. The essence here is that any node *must* have $\left(3^d - 1\right)$ routing entries in total, direct and remote. Table 1 shows routing entries for node 031C shown in Fig. 2a.

2.2 Node join

In P2P systems, peer dynamic refers to the ability of nodes to join, leave, or even fail without any delay or restriction imposed [16]. Thus, it is important for the DHT to have a simplified joining procedure in order to maintain its resilience and scalability when multiple nodes attempt to join. Here, we mention two methods of admitting newcomers to the overlay.

In Chord, after hashing the newcomer's IP address and generating its nodeId, the system relies on some external mechanism to introduce the newcomer to an existing node in the overlay. Afterwards, the existing node will generate a lookup message to find the successor of the newcomer based on its generated nodeId. The newcomer will then connect to its successor and join the overlay. CAN on the other hand assumes the existence of a domain name server that maintains the addresses of bootstrapping nodes which hold partial lists of available nodes in the overlay. In order to join the overlay, the newcomer has first to obtain the address of one of the bootstrapping nodes. Later, the newcomer contacts this node in order to retrieve a list of randomly selected available nodes. Finally, the newcomer sends a join request to one of those nodes which accordingly splits its zone in half and assigns it to the newcomer.

The joining procedure in SHAM is distinctively simple as well. Adding a nascent node to the system entails the following steps.

2.2.1 Bootstrap

Similar to Chord, to join SHAM, node u contacts a bootstrapping server that holds the IP addresses of nodes already situated in the system. The bootstrapping server responds with a list containing the IP addresses of nodes that had recently joined the system. Discovering and contacting a node in the system engages procedures outside of the overlay, i.e., contacting a web-server that is known to provide the addresses of nodes in SHAM. This server is simply a *Rendezvous* host that is not attached in anyway to overlay [17]. Hence, it does not impinge on the functionality of the mechanism such as node and key handling since its role is limited to providing addresses of available nodes.

2.2.2 Establish a connection to an existing node

After receiving a list from the bootstrapping server, the nascent node u selects a node p from the list and sends

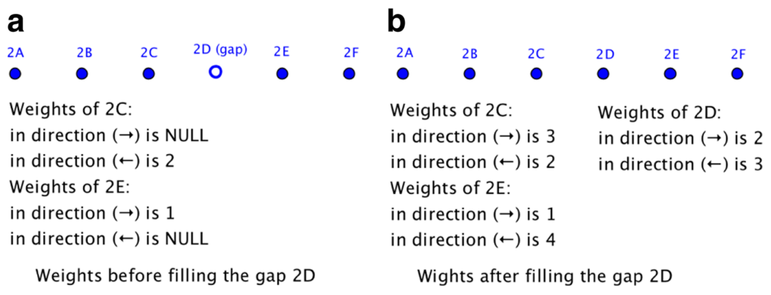

Fig. 4 Weight measurement in a 2-d SHAM. **a** Weights before filling the gap 2D. **b** Weights after filling the gap 2D

Table 1 Routing table entries for node 031C in Fig. 2a

Direction	Neighbor				Successor		Remote	
	NodeId	IP	Timer	Weight	NodeId	IP	NodeId	IP
→	031D	92.110.XXX.XXX	20	4	031A	67.209.XXX.XXX	Null	Null
↘	041D	87.89.XXX.XXX	611	2	011A	167.80.XXX.XXX	Null	Null
↓	041C	212.23.XXX.XXX	221	2	011C	67.115.XXX.XXX	Null	Null
↙	041B	89.213.XXX.XXX	342	4	011A	167.80.XXX.XXX	Null	Null
←	031B	55.10.XXX.XXX	512	4	031A	67.209.XXX.XXX	Null	Null
↖	Null	Null	0	0	Null	Null	011A	167.80.XXX.XXX
↑	Null	Null	0	0	Null	Null	011C	67.115.XXX.XXX
↗	021D	114.23.XXX.XXX	343	4	011A	167.80.XXX.XXX	Null	Null

it a join request which contains its IP address. If no reply is received, u chooses another node and sends a new join request.

2.2.3 *The existing node positions the nascent in the network*

Once the join request is received, p refers to its routing table to find a position for the newcomer. If p has a more than one direct spot[1], it will situate n in one of them directly such that the newcomer will get a maximum number of neighbors upon joining; see Fig. 5a.

However, if p does not have a direct spot, it forwards the join request to another node in the overlay. This forwarding is not arbitrarily done. p consults its routing table to determine on which direction the join request has to traverse. The node elects the direction that reflects the minimum weight entry in the routing table assuming that either a gap exists or that the network lacks growth in that direction. This also helps limit forwarding the request through the overlay. Figure 5b illustrates an example of forwarding the join request based on the weight entry. In this 36-node 2-d system, the host 2C does not have a direct empty spot. Thus, based on its routing table entries, the weight of its link is 1 in the direction of 3B, while the weight in directions 2B and 3C is 2. As a result of that, the join request will be sent to 3B.

After deciding on the position of u, p assigns the newcomer a nodeId tuple and embraces it in the system's routing information:

1. p and u update their routing tables with each other's information.
2. p provides u with its successor information on the direction u was inserted.
3. p sends u's information to their common direct neighbors. Accordingly, each one of them repeats the previous two steps with u.

All of the preceding steps engage adjusting the weight entries of related nodes.

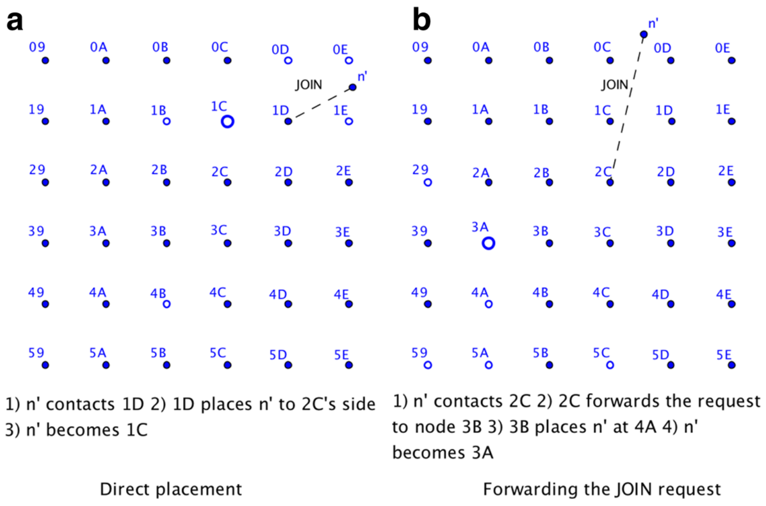
a
b
JOIN
n'
09
0A
0B
0C
0D
0E
19
1A
1B
1C
1D
1E
29
2A
2B
2C
2D
2E
39
3A
3B
3C
3D
3E
49
4A
4B
4C
4D
4E
59
5A
5B
5C
5D
5E
1) n' contacts 1D 2) 1D places n' to 2C's side 3) n' becomes 1C
1) n' contacts 2C 2) 2C forwards the request to node 3B 3) 3B places n' at 4A 4) n' becomes 3A
Direct placement
Forwarding the JOIN request

Fig. 5 Two cases of node n′ joining the system. **a** Direct placement: n′ will be placed by 1D based on a greedy manner at position 1C. **b** Forwarding the join request: 2C does not have an empty spot. The join request will be sent to node 3B which will place n′ at 3A

2.2.4 Graceful departure

Before a node leaves the system gracefully, it chooses a direct neighbor to become the *heir* to its keys. As a consequence, the heir receives the keys from the departing node and assumes their ownership. Furthermore, the new owner duplicates the keys on its direct neighbors. The selection of the heir is based on the weights of the direct neighbors. The departing node aims to choose a neighbor which reflects the maximum value of weight. This is a safeguard to store the keys at nodes which have as many neighbors as possible, thus, duplicating them more efficiently. Finally, the departing node informs the neighbors which are "uncommon" with the heir about its departure to delete its keys.

2.2.5 Ungraceful departure

In the case of a node's departure, the system needs to be informed and updated as well. However, due to the dynamism of P2P networks, nodes tend to leave ungracefully without informing the system. Nodes in SHAM rely on system maintenance messages to keep their routing tables updated. However, in the absence of such messages, SHAM nodes depend on the timeout counters and self-initiated *heartbeat* messages to check the availability of their neighbors. Each neighbor receives the heartbeat message responds with an *alive* message and resets the timeout counter of the neighbor that sent it.

The heartbeat messages used in SHAM are the same as alive messages that are communicated to detect node's failure in other systems, i.e., PASTRY [15]. That is, a heartbeat or alive message is a simple ping command from one peer to another peer in the overlay. Thus, such "soft" messages are not considered as an overhead for the traffic of the overlay as they might traverse through nodes that are not in the overlay.

2.3 Maintenance and recovery

Chord uses periodic stabilization protocol at each node to learn about newly joined nodes, update successor and predecessor, and fix finger tables. Nodes in SHAM announce the changes in their routing tables by piggybacking these changes on traffic messages such as node and key placements and key search [8]. Moreover, in addition to the heartbeats that are invoked when timeout counters expire, SHAM relies on two mechanisms to keep the routing tables up to date: the Join and the Routing Restoration mechanisms.

The join mechanism is the first recovery tool in the system. Whenever a new node joins the system, any other node that either handles its placement or forwards its join request in the overlay will update its routing table accordingly. In the routing restoration mechanism, the system recovers stale entries that are caused by the ungraceful departure of nodes. In this procedure, if a node's direct neighbor and its successor go offline ungracefully, it will send a *hunt* message looking for a node that succeeds the departed successor on the same direction. The hunt message will be forwarded until it reaches a node that is "currently" a neighbor of the first available successor of the departed node. Consequently, the neighbor responds by sending the routing information of the successor directly to the requester. The requesting node in turn updates its routing information to reflect the successor as a link in that direction.

Algorithm 1 The *update* procedure between nodes *p* and *u*

```
    $p.update(u)$
    $R_p^{d^u} \leftarrow u$    {$R_p^{d^u}$: Add $u$ to $p$'s routing table in direction of $u$}
    $R_u^{d^p} \leftarrow p$    {$R_u^{d^p}$: Add $p$ to $u$'s routing table in direction of $p$}
    if $(p.remote)^{d^u} == u^{d^u} + 1$ then
        $R_u^{d^{u+1}} \leftarrow u^{d^u} + 1$
        $R_{u+1}^{d^u} \leftarrow u$
        $p.successor(u^{d^u}) == u^{d^u} + 1$
        if $(p-1) \neq NULL$ then
            $(p-1).successor(p^{d^u}) = u$
            $(u).successor(p^{d^p}) = p - 1$
        end if
    else
        $(u.remote)^{d^u} = (p.remote)^{d^u}$
        $p.successor(u^{d^u}) = (p.remote)^{d^u}$
    end if
    $(p.remote)^{d^u} = NULL$
    $u.publish()$
    $u.weight()$
```

Algorithm 2 The *publish* procedure between nodes *p* and *u*

```
    $u.publish()$
Require: $\psi_u$: set of available direct neighbors of $u$
    for all $j \in \psi_u$ do
        $u \leftarrow \xi_j = \{\Re_u \cap \Re_j\}$    {$\xi_j$: common available neighbors between $u$ and $j$}
        for all $v \in \xi_j$ do
            if $v \notin \psi_u$ then
                $v.update(u)$
            end if
        end for
    end for
```

Successors also enforce the stability of the system. Fundamentally, each node should be in a relation with $\left(3^d - 1\right)$ other nodes. However, in a partially occupied system and due to nodes' departure, there exist gaps and

Algorithm 3 The *weight* procedure for node u

```
$u.weight()$
$\xi_k = \{\Re_u \cap \Re_p\}$
**for all** members of $\xi_k$ **do**
    **if** $u^{d^j} + 1 \neq NULL$ **then**
        **while** $u^{d^j}.link <$ maximum diameter **do**
            $u^{d^j}.link = (u^{d^j} + 1).link + 1$
        **end while**
    **else**
        $u^{d^j}.link = NULL$
    **end if**
**end for**
```

unoccupied positions in the system. SHAM requires each node to hold entries for the successors of its direct neighbors as a contingency plan. As a result of that, each node maintains $2(3^d - 1)$ entries of peers' addresses which fortifies the system and reduces the average path length as we will discuss later. Also, this is important to enhance connectivity and routing in the system. In case a direct neighbor fails or leaves the system, the node will move the successor nodeId and IP address from the routing table to the relative remote neighbor entry. Therefore, the node in SHAM has to maintain between $(3^d - 1)$ and $2(3^d - 1)$ entries in its routing table.

2.4 Key handling

Values are data items needed to be uniformly distributed over the P2P network [18–20]. In structured P2P networks this is usually carried out by using DHTs. Each value in the system will be paired with a key that is generated by a known hashing function. Each node identifier (usually the IP address of the node accompanied with a port number) will be hashed using the same hashing function to generate the nodeId. As a result of that, both of the generated nodeId and the key will fall within the same identifier space. Having created the namespace for nodes and values, every (*key, value*) pair will be stored at a node that has an identifier matches the generated key. If a match is not found, the key will be stored at a node with the closest identifier. This eventually makes every node accountable for a group of values. Our system as mentioned before does not require hashing the node's IP address to generate its nodeId, yet, it is derived directly from the node's position in the overlay. Choosing a hexadecimal representation in SHAM congregates the nodeIds and keys within the same namespace. We relax our guard when choosing the address space since no two nodes will have the same nodeIds. The criterion in choosing the address space becomes related to the load on each node in particular: The more nodes in the system, the less load on each node. On the other hand, the size of the namespace must be large enough to neglect the probability of having two values hash to the same identifier [3].

2.4.1 Key mapping

Key mapping in SHAM is similar to that in other DHT mechanisms. In SHAM, CAN, and Chord, the key will be cached at a node that has a nodeId that is closest to the key. Each time a new node joins with closer nodeId to that of they key, the key will be forwarded to it. Specifically, in SHAM, keys are distributed to nodes based on the coordinate system $(X^1, .., X^d)$. The address of the key will be resolved to get its X^1, X^2, .., and X^d coordinates. For an address space with a tuple $(X^1, .., X^d)$, any key in the system will be regarded as a concatenation of $\left(x^1_{r_1-1}..x^1_1x^1_0, .., x^d_{r_d-1}..x^d_1x^d_0, z\right)$ bytes, where r_d is the number of hexadecimal bytes along the X^d coordinate and z is a suffix of hexadecimal bytes in which $z = 0$ if the address space is the same size of the key space. After extracting the coordinates of the key, it will be forwarded through the system until it reaches a node that matches its X^1, X^2, .., and X^d coordinates. If no node has been placed in the system with that address yet, the key will be stored at a node that has the closest identifier to that of the key. Later on, whenever a node whose identifier is closer to the key joins the system, the key will be reallocated to that newcomer. The reallocation of the key may continue until it is stored at node whose nodeId matches the key[2].

Figure 6 shows an example of key mapping in a 2-d SHAM system, (X^1, X^2). In the figure, node 3E receives key 1B23F which is resolved to $X^1 = 1$, $X^2 = B$, and $z = 23F$. Thus, the node deduces that the key should be stored at node 1B. 3E forwards the key to another node that has a closer identifier to that of the key. In this particular case, 3E forwards the key to node 1C. After receiving the key, 1C applies the same procedure to decide on which direction the key should be forwarded. Since 1B is a direct neighbor to 1C, and since it has not been placed in the system yet, 1C caches the key locally. After 1B joins the system, the key 1B23F will forwarded to 1B.

2.4.2 Key duplicate

Replication techniques play a major role in reducing latency, improving load balancing, and enhancing availability [21–24]. Some known techniques are synchronous, asynchronous, dynamic, full, and neighborhood replication [25–27]. Such schemes differ based on their complexity. For example, synchronous, asynchronous, and dynamic replication techniques require frequent messaging in order to keep the system updated. Whereas the full replication technique is much simpler in principal, yet it is associated with high cost in maintaining the replicas of all keys in the system. In SHAM, we employ the neighborhood replication: Once the key is stored at a node, it will be directly replicated at the node's neighbors only as a

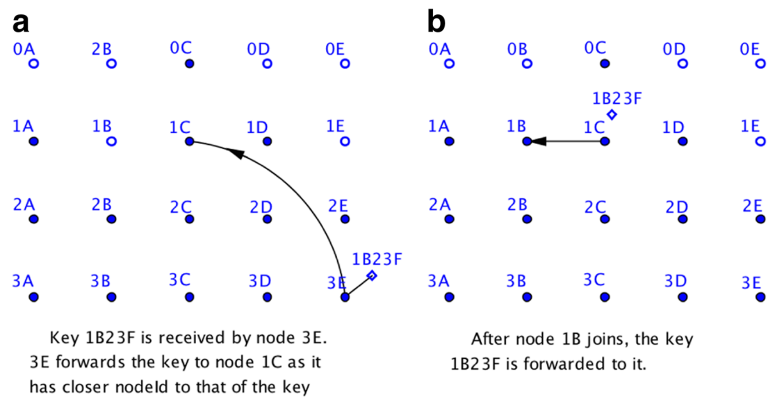
a
0A
2B
0C
0D
0E
1A
1B
1C
1D
1E
2A
2B
2C
2D
2E
3A
3B
3C
3D
3E
1B23F
Key 1B23F is received by node 3E. 3E forwards the key to node 1C as it has closer nodeId to that of the key
b
0A
0B
0C
0D
0E
1B23F
1A
1B
1C
1D
1E
2A
2B
2C
2D
2E
3A
3B
3C
3D
3E
After node 1B joins, the key 1B23F is forwarded to it.

Fig. 6 Key mapping in a 2-d SHAM. **a** Key 1B23F is received by node 3E. 3E forwards the key to node 1C as it has closer nodeId to that of the key. **b** As node 1B joins, the key 1B23F is forwarded to it

safeguard that the key will still be an accessible contingent upon the failure of its host. In this scheme, no messaging is required to update the system, i.e., there is no need to know who has what. Moreover, the cost of maintaining the direct neighbors' keys is considerably a small compromise for enhancing the availability of keys.

We require the node to discriminate between its own keys and its neighbors' keys. The discrimination is crucial to prevent the neighbors' keys from being replicated over and over again in the network. Strictly speaking, the node will only replicate its original keys. This rule holds until the neighbor which replicated its keys departs. In that case, as an heir, the node assumes the neighbor's keys to be original and replicate them at its neighbors except those which are common with the departed.

2.4.3 *Key search*

The presence of the weight entries in the routing tables fortifies the routing efficiency of the system. Those entries give the node sufficient knowledge of the approximate depth of the network from all $\left(3^d - 1\right)$ directions. Searching for a key in our system comprises two steps. First, resolve the direction of which the search will traverse. Then, estimate the distance (*depth*) of the key from the requesting node. Using simple mathematical interpretation, the node can resolve the direction on which the query should traverse. Consequently, the node will approximate its distance from the requested key based on the weight of the neighbor which is in that direction. The mathematical interpretation is an inherited characteristic from the topology of the system. The property of sequential addressing places the nodes within specific intervals from each other. Table 2 shows the distance between a node and its direct neighbors in a 2-dimensional system X^1X^2.

Figure 7 illustrates an example on key search in a 2-d system that uses an addressing rule with $r_1 = r_2 = 1$ and $X^1 = \left(x_0^1\right)$, $X^2 = \left(x_0^2\right)$. In the figure, when node 5C searches for key $1731A$, it first determines the location of node 17 in the system, which is $5 - 1 = 4$ steps away on the upward direction of X^2 coordinate and $C - 7 = 5$ steps away on the leftward direction of X^1 coordinate. Accordingly, 5C forwards the request to node 3A which is in the upward and leftward direction in its routing table. The forwarding continues by 3A following the same procedures.

3 Discussion

We compare SHAM with two major DHT systems: CAN and Chord. Each one of them represents a family of DHT routing protocols. There are more recent DHT routing protocols that have been recently developed [28–30]. However, the majority of these systems adopt the paradigms of CAN or Chord or they do not fall into the class of systems that balance between the routing performance and the size of the routing table.

We choose three areas of comparison in this paper: routing performance, robustness against failures, and load balancing. In addition to that, we explore a distinctive

Table 2 Address resolution in 2-d SHAM

	With direct neighbors	
$X^1 - 1X^2 - 1$	$X^1 - 1X^2$	$X^1 - 1X^2 + 1$
$X^1X^2 - 1$	X^1X^2	$X^1X^2 + 1$
$X^1 + 1X^2 - 1$	$X^1 + 1X^2$	$X^1 + 1X^2 + 1$
	With other neighbors	
$X^1 - jX^2 - j$	$X^1 - jX^2$	$X^1 - jX^2 + j$
$X^1X^2 - j$	X^1X^2	$X^1X^2 + j$
$X^1 + jX^2 - j$	$X^1 + jX^2$	$X^1 + jX^2 + j$

Note: $j = 2, 3, .., E, F$
$1 + F = 0$ and $0 - 1 = F$

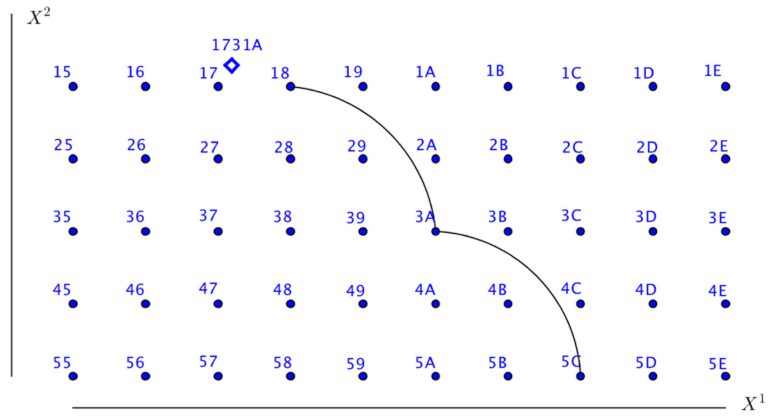
X^2
1731A
15 16 17 18 19 1A 1B 1C 1D 1E
25 26 27 28 29 2A 2B 2C 2D 2E
35 36 37 38 39 3A 3B 3C 3D 3E
45 46 47 48 49 4A 4B 4C 4D 4E
55 56 57 58 59 5A 5B 5C 5D 5E
X^1

Fig. 7 Key search in a 2-d SHAM system. Following the address resolution procedure, node 5C forwards the query to node 3A from its routing table. Subsequently, 3A forwards the query to node 18. Based on key replication technique used in SHAM, node 18 will answer the query

characteristic of SHAM, homogeneous addressing, and show its effectiveness in reducing latency in the underlying network.

3.1 Average path length in CAN vs. SHAM

In CAN, the average path length for a d-dimension system is $\frac{d}{4}N^{\frac{1}{d}}$, where each dimension has an average path length of $\frac{1}{4}N^{\frac{1}{d}}$. In SHAM, however, the average path length for each dimension is $\frac{1}{4}N^{\frac{1}{d}}$, whereas for a d-dimension system the average path length becomes $\frac{1}{8}N^{\frac{1}{d}}$. This is because the search in SHAM takes diagonal paths between coordinates while in CAN the search traverses on the coordinates as Fig. 8 illustrates [4, 31]. Furthermore, each hop in SHAM is being performed from a node to its direct neighbor's successor which reduces the path length by half.

3.2 Performance against CAN and Chord

In Fig. 9, we present the results of the first experiment in matching SHAM up with CAN and Chord. The basis in this part of the simulation is to saturate the overlays with nodes by setting Poisson arrival rate $\lambda = 1$ and the departure rate $\mu = 0$. The overlay size varied from 2^{10} nodes with a query rate of 0.1 and an update rate of 0.001 to 2^{23} nodes with a query rate 0.001 and an update rate of 0.0001. Table 3 summarizes these parameters.

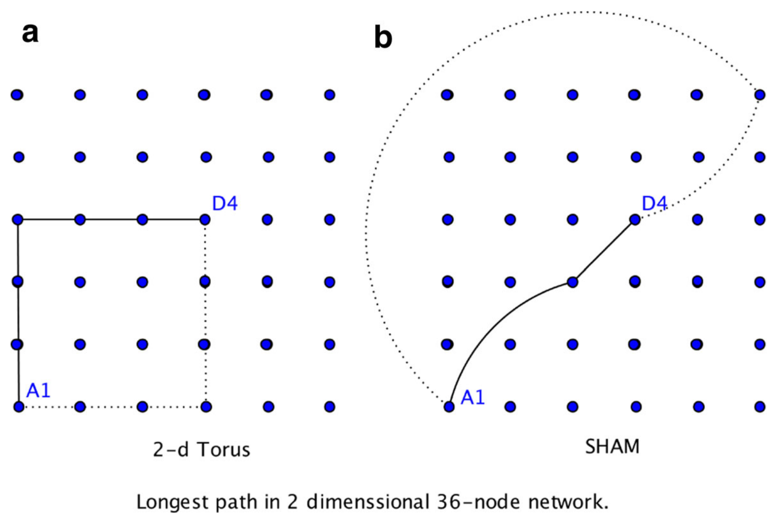

Fig. 8 Depiction of routing in SHAM vs. Torus. **a** In 2-d torus: the longest path is between nodes A1 and D4. It requires the lookup six hops to reach D4 from A1 (solid line). Another possible same length route is the dashed line. **b** In case of SHAM: the longest path is also between A1 and D4. It requires the lookup two hops to reach D4 (solid line). Another same length route is the dashed line

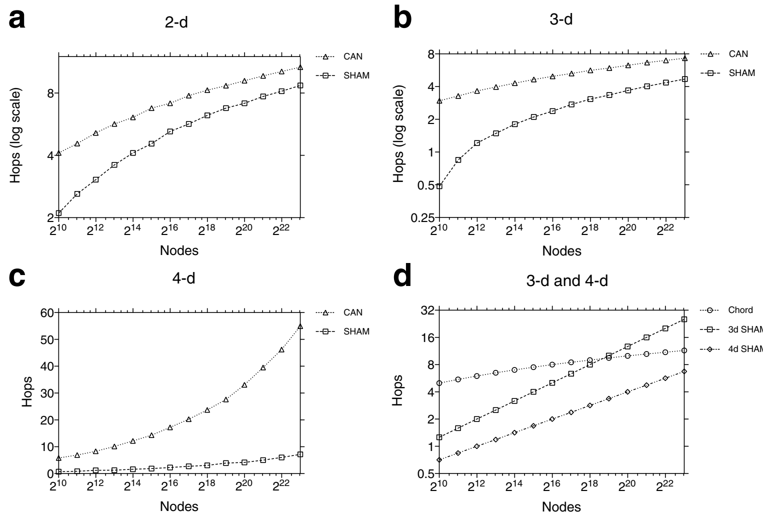

Fig. 9 a–d Average path length in SHAM, CAN, and Chord

This experiment is comprised of two steps: First, we compare the performance of SHAM with CAN in 2-d, 3-d, and 4-d systems. Second, we compare the performance of SHAM in 3-d and 4-d systems with Chord.

Regarding CAN, the results seen in the figure support our analysis that SHAM performs better than CAN and achieves a shorter path length. The advantage in performance is clearly seen in Fig. 9a–c. Distinctly, the performance of SHAM in 3-d is even better than CAN in 4-d. Part of this improvement in performance is credited to the use of diagonal paths between coordinates which minimizes the average path length by a factor of d. The other part is ascribed to the double hops that are being performed from the node to the successor of its direct neighbor which further reduces the average path length by a factor of 2.

On the other hand, the results presented in Fig. 9d show that up to a certain network size, SHAM performs better than Chord in 3-d and 4-d. This is consistent with expectation: in 3-d and 4-d, SHAM has shorter path length than Chord in networks of sizes up to 2^{19} and 2^{27} nodes, respectively. Thus, increasing the number of dimensions plays a significant role in reducing the average path length in SHAM comparing to Chord.[3]

Table 3 Simulation parameters 1

Operating system	Red hat
Simulator	C++
Processor	Xeon E7 @ 3.20–3.50 GHz
Cache	45 MB
RAM	16 GB
Network size	$2^{10} - 2^{23}$
Number of keys	$2^{10} - 2^{23}$
Arrival rate λ	1
Departure rate μ	0
Query rate per node	{0.001–0.1} per time interval
Update rate per node	{0.0001–0.01} per time interval

3.3 Failed lookups

Another part of the analysis is failed lookups, i.e., queries that do not result in keys. Although Chord has a better performance after specific network sizes in 4-d, however, when it comes to failed lookups, SHAM noticeably outperforms Chord.

To examine this venue, we run the simulation on fully occupied 2^{16}-node 3-d SHAM and Chord overlays and measure the number of failed queries in each system. The scenario of this second experiment is to fail nodes at different capacities in both systems by increasing the departure rate, then monitor the percentage of queries that encounter failed nodes through their paths to destination; see Table 4. From Fig. 10a, we notice that in SHAM and Chord, as the percentage of failed nodes increases, the probability that a query will face at least a failed node also increases. However, Fig. 10b, which represents the percentage of failed lookups, signifies the difference in performance between the two systems. In SHAM, when the percentage of failed nodes is 20% for instance, the probability that a query will encounter at least a failed node is around 65%, yet, the percentage of failed queries is less than 10%. This is not the case for Chord: For the same percentage of failed nodes, the probability of facing at least a failed node is around 80%, while the percentage of failed lookups reaches more than 35%.

Table 4 Simulation parameters 2

Network size	65,536
Number of keys	262,144
Graceful departure %	30%
Departure rate μ	{0–0.2}
Query rate	{0.2–0.5} per time interval
Update rate	{0.1–1} per time interval

This reduction in failed lookups in SHAM comparing to Chord is related to the following. First, it is related to the connectivity of the system in SHAM where each node maintains entries to $2(3^d - 1)$ nodes in total, thus, having different possible routes to the destination. Second, each key in SHAM is being replicated at the direct neighbors of the host. As a result of that, even if the host failed, once the query reaches one of its direct neighbors, it would be considered as a successful hit. In Chord, however, a failed lookup is attributed to one of these two reasons: either the node that is hosting the key has failed which means a failed lookup or the finger table of some of the predecessors are inconsistent which hinders forwarding the query [3].

The last part of the comparison is to examine the additional number of hops visited when lookups face failed nodes. In this experiment, we use the previous network configuration in SHAM and Chord with two failure settings: 5 and 20%. Figure 11 illustrates that SHAM outperforms Chord in this part of the performance as well. For a 5% failure, in Fig. 11a, around 84% of lookups encountering at most two additional hops is observed in SHAM, while in Chord, 86% of lookups encounter at most three additional hops. On the other hand, Fig. 11b shows the results of the same experiment for a 20% failure rate. The figure shows that both systems start sluggishly with 11 and 6% of at most two additional hops visited for SHAM and Chord respectively. However, as seen in the figure, 61 and 56% of lookups encountered at most five and seven additional hops in SHAM and Chord respectively.

3.4 Load balancing

Load balancing is an important aspect of structured P2P networks [18, 32–35]. A robust and resilient system should be able to fairly distribute the keys in the network over the participating nodes [32, 34, 35]. This will avert inundating nodes with keys and hence improving resources accessibility in the network. Consistent hashing uniformly distribute keys over the namespace as discussed earlier. However, in structured P2P system where both of the nodeId and the key are generated using the consistent hashing, there exist situations where the node identifiers do not uniformly cover the entire namespace system [3]. Thus, some nodes will be overloaded with keys while other nodes do not cache any keys. Chord maintains uniformity in this regard by requiring each node to cache its keys at an additional $O(\log N)$ *virtual* nodes.

SHAM emulates the property of consistent hashing by deterministically assigning nodeIds based on the predefined addressing scheme. Moreover, if a node that is supposed to store a key has not been placed yet, the key will be cached at a node that has a nodeId which is closest to the key's identifier temporarily. Whenever a node with closer nodeId is placed in the system, the key will be forwarded to it. This ensures that each node will roughly receive the expected load. However, most importantly, SHAM benefits from its key replicating scheme at direct neighbors in enhancing its load balancing. This is nearly similar to Chord with two differences. First, in Chord, the node will need to discover its virtual nodes and saves their routing information whereas in SHAM, those nodes are already in the node's routing table. Second, the cost of adding the virtual nodes in Chord is increased [3]. For example, in an overlay of one million nodes, a Chord's node has to maintain entries for $\log^2 N \leq 400$ nodes. In 4-d SHAM, however, for the same network, each node maintains entries for $2\left(3^4 - 1\right) = 160$ other nodes in total.

Figure 12 signifies the load balancing capability in SHAM. The figure presents the results obtained by simulating a 2-d SHAM overlay with a capacity of 2^{12} nodes and 2^{14} keys. Intuitively, if the system is fully occupied, we expect theoretically each node to hold $2^{14}/2^{12} = 4$ keys. However, the deficiency in load balancing in DHTs arises when the overlay is not fully occupied or the namespace is not entirely covered. Therefore, to show the efficiency of SHAM in load balancing, we run the experiment on

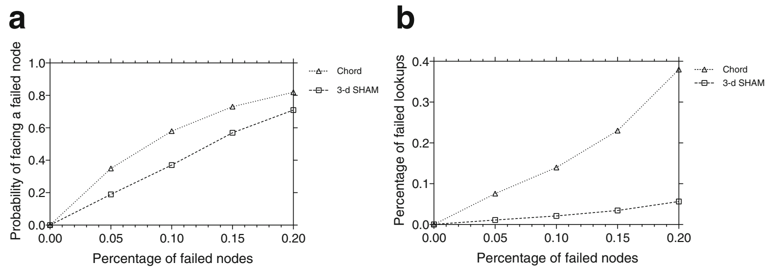

Fig. 10 Performance comparison between SHAM and Chord. **a** Probability that a query will encounter a failed node. **b** Percentage of failed lookups

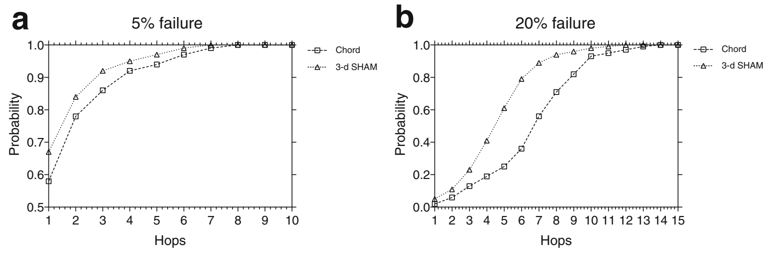

Fig. 11 Performance comparison between SHAM and Chord. CDF of number of additional hops visited in case a lookup encounters failed nodes in a network of 65,000 nodes. **a** Percentage of failed nodes is 5%. **b** Percentage of failed nodes is 20%

a partially occupied overlay. The first figure, Fig. 12a, shows how SHAM distributes 2^{14} keys over nodes without invoking key replication procedure. The results indicate that when 2^{11} nodes are present, load balancing is maintained in the overlay with around 60% of nodes holding between 10 to 16 keys each. Additionally, Fig. 12b presents the results of running the same experiment when the key replicating procedure is invoked. The results indicate that replicating keys at neighbor nodes adds more balance in distributing keys: around 23% of nodes are storing 160 keys each. The tradeoff here is in the increase in the number of keys the node holds; however, we believe that this increase is viable and practical since not only it enhances load balancing but also it increases availability in the network.

3.5 Homogeneous addressing

In addition to load balancing, nodes dynamics, and self-organizing capabilities, reduced latency is a crucial property in P2P networks [23, 24, 36, 37]. Latency in P2P network is related to the distributed nature of nodes. Nodes with adjacent addresses in the overlay might be located in different geographical areas. This mismatch between the overlay and the IP network is the major factor for increasing latency. For example, in SHAM, a direct neighbor is within one hop, yet, this same neighbor could be many hops away in the IP network. A remedy to this problem is to enable nodes to connect to physically nearby neighbors. Other method is utilizing clustering to impose physical proximity of overlay neighbors or even using geographical routing, i.e., geographic awareness [38, 39].

In SHAM, we propose positioning nodes with adjacent physical addresses at close spots in the overlay as a solution to this problem. SHAM can function as to arrange nodes in classes, where each class holds a range of nodes with close physical locations. Thus, once a newcomer arrives to the bootstrap server, a traceroute command is sufficient to indicate to which class the newcomer belongs. In that case, the bootstrap server constructs a list of nodes from that class to handle the positioning of the newcomer. Figure 13 shows the results of testing the latency in SHAM under two conditions. First, when nodes are placed with no regard to their geographical positions; see Fig. 13a. And second, when we apply homogenous IP/location addressing scheme; see Fig. 13b.

In this experiment, we simulate a 2-d SHAM system having 2^{11} nodes distributed over five IP classes with each class representing a geographical area. The latency between nodes within the same class is drawn uniformly at random from $[5, 15]$ intervals. Similarly, the interclass latency is selected from the space $[16, 45]$. The scenario here is to send queries from source nodes to their most distant nodes in the overlay and measure the latency these queries accumulate in a heterogeneous and homogeneous addressing schemes. Noticeably, the fluctuation in delay in first condition is due to the heterogeneity of nodes locations in the overlay. On the other hand, when we apply homogenous addressing in SHAM, fluctuation has been

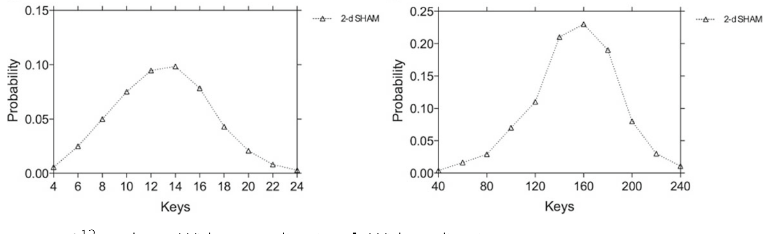

Fig. 12 Distribution of 2^{14} keys over 2^{12} nodes. **a** Without replication. **b** With replication

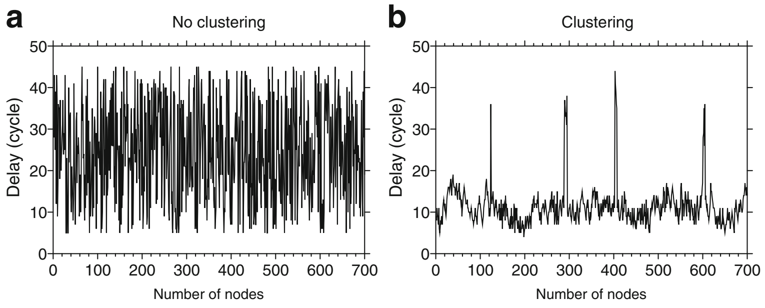

Fig. 13 Latency when no clustering is imposed. **a** No clustering. **b** Clustering

reduced drastically with occasional overshoots. The overshoots are affiliated with forwarding the queries from one class of addresses to another.

4 Related work

In classifying structured P2P overlays based on their routing performance, we put them in four major groups: single-hop overlays, constant degree overlays, multi-dimension overlays, and logarithmic degree overlays. In single-hop overlays and as the name implies, routing can be resolved within one hop. Some of the known $O(1)$ overlays are D1HT [1], OneHop [2], Kelips [7], and EpiChord [8]. As mentioned before, in order to resolve lookups within one step from source to destination in such overlays, nodes must maintain global knowledge about the status of the overlay. Thus, a major drawback to the $O(1)$s is the high cost associated with maintaining the global routing tables and their inability to handle churn-intensive workloads.

Constant degree overlays take fixed number of hops in resolving lookups regardless of the number of nodes in the system. One of the well-known systems in this class is Cycloid [12]. The system is a d-dimensional cube-connected cycles. Every vertex in the cube is replaced by a *cycle* of 3° d nodes each maintaining a routing table of seven entries. Although routing is achieved within $O(d)$, however, stabilizing and maintaining valid routes degrade the system's performance.

The third category in DHTs is the multi-dimension overlays. Our system falls within this category. The most prominent system in this category and one of the earliest DHT systems that have been proposed is CAN [4, 31]. Nodes self-organize themselves in CAN in a virtual d-dimensional Cartesian coordinate space built on a d-torus. To join the system, a newcomer must acquire the information of an available node in the overlay. This information is obtained usually from a bootstrapping server. A join request is then sent by the newcomer destined to a point P in the overlay. The message is routed using CAN routing algorithm through the available nodes until it reaches a node which is the owner of the zone where P exists. The node then divides its own zone by half and assigns one half to the newcomer. The split process is performed in such a way that the zones will be re-merged upon the departure of nodes. Each node in CAN holds entries for a number of neighbors based on the dimension of the coordinate system. Thus, insertion process in CAN affects $O(d)$ existing nodes. Average path length in CAN is bounded to $O\left(\mathrm{N}^{1/d}\right)$. If d is chosen such that $d = (\log N)/2$, routing performance converges to $O(\log N)$. Although our mechanism seems similar to CAN, however, in addition to lowering the cost of search, there are also major points that distinguish SHAM. First, unlike CAN, the overlay in SHAM is already organized into fixed addresses, or spots, that will be occupied when nodes join, whereas in CAN, the *spot* will be created for the node upon its arrival, which may include zone splitting. Second, situating a newcomer in SHAM is much simpler and requires minimum effort comparing to CAN, which involves forwarding the join request to the selected point P as mentioned earlier. Third, SHAM employs homogeneous addressing which is not possible in the CAN structure.

The last category in our discussion is logarithmic degree overlays. In this category of DHTs, routing is reduced by half in each step the query takes towards the target. Systems such as Chord, Ulysses, Pastry, Kademilia, Tpastry, and P-grid belong to this class [3, 5, 15, 40–43].

Chord is the most renowned DHT in this category. It utilizes consistent hashing to generate an $m - bit$ identifier for nodes and keys. Nodes are then organized in an identifier circle modulo 2^m. The node identifier will be generated from the node's IP address while the identifier of the key will be generated by hashing the key itself. This arrangement makes both of the nodes and keys have the same naming space. Chord assigns keys to nodes based on the naming space as well. A key k will be stored at its *successor* node whose nodeId matches the identifier of the key in the space. If no match is found in the system, i.e., if the node whose nodeId matches the key has not been placed in the system or has left the system, the key will be cached at the next node whose nodeId follows the identifier of they key in the space in a clockwise manner. Also, Chord recognizes a *predecessor* to a node (or a key) as a node whose nodeId precedes that node in the circle in a counterclockwise manner. Each node in Chord maintains a routing table of $O(\log N)$ entries in N-node system. Lookups are resolved in $O(\log N)$ messages to other nodes, while insertion or deletion of a nodes affect $O(\log^2 N)$. However, in order for Chord to be load balanced, each node has to store its keys at additional $O(\log N)$ virtual nodes. Thus, increasing the routing table for the purposes of routing and storing to $O(\log^2 N)$. For reader's reference, other DHTs that are derived from Chord are [28–30].

Ulysses is a P2P structured system that adopts the *static butterfly* topology and attempts to reduce the $O(\log N)$ latency by a factor of log log N [5]. Naming space in Ulysses is based on a row-level convention. In a network with l levels and N nodes, each DHT node is depicted by a tuple of row and level identifiers (P, l), where P is a binary string signifying the row to which the node belongs and l denotes the level in that row. The row identifier can be mapped into a concatenation of bits between 0 and $k - 1$, where k is the size of the dimension in the static butterfly topology. The space in Ulysses is divided into disjoint zones in which each DHT node is responsible for a specific zone. A node with an identifier (P, l) caches all keys (α, l) where P is a prefix to α. Routing in Ulysses traverses the network through levels. If a node (P, l) searches for a key (α, i), the query will be forwarded to the next level $(P', l + 1)$. α in this case will match a range of the binary string P' in level $l + 1$. The forwarding continues in Ulysses through levels where at each step, the search for α narrows down until the target level is reached. Ulysses is optimized such that its nodes maintain routing tables of an average size of size log N in order to achieve a minimum diameter of $\lceil \frac{\log N}{\log \log N} \rceil + 1$. Although Ulysses reduces the path length while maintaining log N routing entries, however, this comes at a cost of uniformity. Thus, two major drawback exist in Ulysses: node congestion and load balancing.

In addition to routing performance in DHTs, we also list some of the overlays that discuss the geographical awareness in structured P2P networks [39, 44, 45].

Jedda and Mouftah [39] proposes the Geographic-Aware Content Addressable Network (GCAN) to solve the issue of geographic awareness in object naming service architecture, ONS, by placing it on top of Chord-like systems. According to the paper, GCAN preserves the complexity of Chord to $O(\log N)$ and the routing table size to $O(\log N)$ as well. GCAN is built on a grid that could cover a certain geographical area. This area is divided into cell using vertical and horizontal chords. Later, using a discretization procedure, cells will be filled with nodes that represent nameservers.

SpatialP2P is a decentralized mechanism that provides node indexing and key storing for multi-dimensional data [44]. According to the paper, the proposed framework upholds the major requirements of DHTs such as indexing, retrieval, and load balancing. It is built on grid that is divided into *cells* that hold the spatial data and are identified by the grid's coordinates. Spatial data can be stored at one single cell or they can span over a group of cells. Moreover, nodes in the system are also identified by the same coordinates making cells and nodes sharing the same coordinates eventually map to each other. Nodes in SpatialP2P maintain lists of successors and indexed nodes as well. Successors are needed for connectivity and routing, while indexed nodes are essential for enhancing the lookups. The cost of search in SpatialP2P when mapped to a one-dimensional Chord-like system is bounded to 3log N. Although the paper discusses load balancing, however, the experiment tested the scenario of having fully occupied network whereas the issue of load balancing is when the network is not fully occupied.

Geodemlia is yet another P2P overlay that is based on Kademlia [40, 45]. It is a location-based search algorithm that allows nodes to locate keys around specific geographical area. Geodemlia is to be built on static nodes that provide storage and search functionalities for mobile devices. Nodes in Geodemlia are positioned on a sphere inspired by the shape of the earth. Accordingly, each node can be located using longitude and latitude angles using some location services, i.e., GPS or IP locators. Nodes in the system split the geographical area into predefined directions, where for each direction, the area will be divided into distance buckets that store fixed number of the system's nodes. Geodemlia, as a geographical awareness mechanism and similar to SpatialP2P, preserves the locality and directionality of data in the overlay. However, the assumptions of the static nature of nodes and load balancing are a matter of question.

5 Conclusions

In this paper, we presented SHAM, an addressing mechanism for structured P2P overlays. SHAM is robust, highly scalable, and decentralized. SHAM is simple, it uses a hashing function to generate keys' identifiers, and a primitive hexadecimal scheme to address nodes in the overlay. In performance, given a key, SHAM can route queries to that key in $1/8\left(N^{1/d}\right)$ steps, with each node maintaining $2\left(3^d - 1\right)$ routing entries, where d is the number of dimensions and N is the number of nodes in the steady state.

With a limited increase in the size of the routing tables and the use of diagonal paths, we have shown that SHAM significantly outperforms CAN. Moreover, SHAM even performs better than the dominant Chord system. Our results demonstrated that failed lookups were noticeably reduced in SHAM comparing to Chord. We attributed this reduction to the presence of multiple routes to the destination and to the duplication of keys in the system.

The rigidness of SHAM stems from the use of the gap filling mechanism in which the priority is to situate newcomers in vacant positions formed by departed nodes. Thus, entrenching routing as nodes perform double hopping to the successors of their direct neighbors.

Finally, with the use of homogeneous addressing scheme, nodes which have close geographical locations can be positioned adjacent to each others in the overlay. Thus, giving SHAM a major advantage in reducing latency in the network.

Endnotes

[1] The term *spot* in this paper refers to an empty position in the overlay.

[2] Key refers to the (key, value) pair.

[3] Choosing the number of dimensions in SHAM such that $d = \log(N)$ reduces SHAM to be from the same family as Chord. Thus, reducing the average path length to $\frac{1}{2}\log(N)$.

Authors' contributions

MZ proposed and designed the mechanism and the simulator. NUH assisted in the analysis of the output data. Both authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 8 February 2017 Accepted: 8 September 2017
Published online: 25 September 2017

References

1. LR Monnerat, CL Amorim, in *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*. D1HT: a Distributed OneHop Hash Table, (Rhodes Island, 2006), p. 10. doi:10.1109/IPDPS.2006.1639278
2. A Gupta, B Liskov, R Rodrigues, in *Proceedings of 1st Symposium on Networked Systems Design and Implementation*. Efficient Routing for Peer-to-peer Overlays (USENIX Association Berkeley, San Francisco, 2004), pp. 113–116
3. I Stoica, R Morris, D Karger, MF Kaashoek, H Balakrishnan, in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01)*. Chord: A Scalable P2P Lookup Service for Internet Applications (ACM, New York, 2001), pp. 149–160. doi:http://dx.doi.org/10.1145/383059.383071
4. S Ratnasamy, P Francis, M Handley, R Karp, S Schenker, in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01)*. A Scalable Content Addressable Network (ACM, New York, 2001), pp. 161–172. doi:http://dx.doi.org/10.1145/383059.383072
5. A Kumar, S Merugu, J Xu, X Yu, in *Proceedings of the 11th IEEE International Conference on Network Protocols*. Ulysses: a Robust, Low-Diameter, Low-Latency Peer-to-peer Network, (2003), pp. 258–267. doi:10.1109/ICNP.2003.1249776
6. F Kaasheok, D Karger, in *Proceedings of the 2nd International Workshop, IPTPS*. Koorde: a Simple Degree-optimal Distributed Hash, (Berkeley, 2003). doi:10.1007/978-3-540-45172-3_9
7. I Gupta, K Birman, P Linga, A Demers, R van Renesse, in *Proceedings of the 2nd International Workshop, IPTPS*. Kelips: Building an Efficient and Stable P2P DHT Through Increased Memory and Background Overhead, (Berkeley, 2003). doi:10.1007/978-3-540-45172-3_15
8. B Leong, B Liskov, E Demaine, in *Proceedings of the 12th IEEE International Conference on Networks, ICON*. EpiChord: Parallelizing the Chord Lookup Algorithm with Reactive Routing State Management, vol. 1, (2004), pp. 270–276. doi:10.1109/ICON.2004.1409145
9. L Monnerat, CL Amorim, in *Proceedings of the 2009 IEEE Global Telecommunications Conference*. Peer-to-Peer Single Hop Distributed Hash Tables, (Honolulu, 2009), pp. 1–8. doi:10.1109/GLOCOM.2009.5425764
10. L Monnerat, CL Amorim, An Effective Single-Hop Distributed Hash Table with High Lookup Performance and Low Traffic Overhead. Concurr. Comput. Pract. Experience, 1767–1788 (2015). doi:http://dx.doi.org/10.1002/cpe.3342
11. J Risson, A Harwood, T Moors, in *IEEE Transactions on Parallel and Distributed Systems*. Topology Dissemination for Reliable One-Hop Distributed Hash Tables, (2009), pp. 680–694. doi:10.1109/TPDS.2008.145
12. H Shen, C-Z Xu, G Chen, in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. Cycloid: a Scalable Constant-degree Lookup-Efficient P2P Overlay Network, (Santa Fe, 2004), p. 26. doi:10.1109/IPDPS.2004.1302935
13. MI Yousuf, S Kim, in *Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP)*. Kistree: A Reliable Constant Degree DHT, (Goettingen, 2013), pp. 1–10. doi:10.1109/ICNP.2013.6733613
14. D Li, X Lu, J Wu, in *Proceedings of IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies*. FISSIONE: a Scalable Constant Degree and Low Congestion DHT Scheme Based on Kautz Graphs, (2005), pp. 1677–1688. doi:10.1109/INFCOM.2005.1498449
15. A Rowstron, P Druschel, in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems, (2001), pp. 329–350. doi:10.1007/3-540-45518-3_18
16. J Augustine, G Pandurangan, P Robinson, S Roche, E Upfal, in *Proceedings of IEEE 56th Annual Symposium on Foundations of Computer Science*. Enabling Robust and Efficient Distributed Computation in Dynamic Peer-to-Peer Networks (Foundations of Computer Science (FOCS), Berkeley, 2015), pp. 350–369. doi:10.1109/FOCS.2015.29
17. V Venkataraman, K Yoshida, P Francis, in *Proceedings of the 2006 IEEE International Conference on Network Protocols*. Chunkyspread: Heterogeneous Unstructured Tree-Based Peer-to-Peer Multicast, (Santa Barbara, 2006), pp. 2–11. doi:10.1109/ICNP.2006.320193
18. E Balaji, G Gunasekaran, in *2016 International Conference on Information Communication and Embedded Systems (ICICES)*. Efficient Range Query Processing, Load Balancing and Fault Tolerance with Popular Web Cache in DHT, (Chennai, 2016), pp. 1–4. doi:10.1109/ICICES.2016.7518891
19. D Thatmann, A Butyrtschik, A Kupper, in *Proceedings of the 9th International Conference on Signal Processing and Communication Systems (ICSPCS)*. A Secure DHT-Based Key Distribution System for Attribute-Based Encryption and Decryption, (Cairns, 2015), pp. 1–9. doi:10.1109/ICSPCS.2015.7391732
20. Y Zhang, L Liu, *Distributed Line Graphs: a Universal Technique for Designing DHTs Based on Arbitrary Regular Graphs*, vol. 24, (Beijing, 2008), pp. 152–159. doi:10.1109/ICDCS.2008.35
21. Z Qiu, J Pérez, P Harrison, in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, ICPE '16*. Tackling Latency via Replication in Distributed Systems (ACM, New York, 2016), pp. 197–208. doi:10.1145/2851553.2851562
22. D Wang, G Joshi, G Wornell, Using straggler replication to reduce latency in large-scale parallel computing. SIGMETRICS Perform. Eval. Rev. **43**(3), 7–11 (2015)
23. A Vulimiri, B Godfrey, R Mittal, J Sherry, S Ratnasamy, S Shenker, in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies. Low latency via redundancy*. Low latency via redundancy (ACM, New York, 2013), pp. 283–294. doi:10.1145/2535372.2535392
24. A Vulimiri, O Michel, PB Godfrey, S Shenker, in *11th ACM Workshop on Hot Topics in Networks (HotNets-XI)*. More Is Less: Reducing Latency via Redundancy (ACM, New York, 2012), pp. 13–18. doi:10.1145/2390231.2390234

25. E Cohen, S Shenker, in *Proceedings of ACM SIGCOMM'02*. Replication Strategies in Unstructured Peer-to-Peer Networks (ACM, New York, 2002), pp. 177–190. doi:10.1145/633025.633043
26. Q Lv, P Cao, E Cohen, K Li, S Shenker, in *Proceedings of ICS '02*. Search and Replication in Unstructured P2P Networks (ACM, New York, 2002), pp. 84–95. doi:10.1145/514191.514206
27. X Shen, et al., *Handbook of Peer-to-Peer Networking, 3*. (Springer Science+Business Media, LLC 2010. doi:10.1007/978-0-387-09751-0 1
28. W Xiong, DQ Xie, LX Peng, J Liu, in *Proceedings of 2011 International Conference on Electronic & Mechanical Engineering and Information Technology*. PrChord: A Probability Routing Structured P2P Protocol, Harbin, Heilongjiang, 2011), pp. 3142–3145. doi:10.1109/EMEIT.2011.6023753
29. S Wang, S Yang, L Guo, in *Proceedings of the Third International Conference on Communications and Mobile Computing*. LiChord: a Linear Code Based Structured P2P for Approximate Match, (Qingdao, 2011), pp. 118–121. doi:10.1109/CMC.2011.31
30. Y Wang, X Li, Q Jin, J Ma, in *Proceedings of the 9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing*. AB-Chord: an Efficient Approach for Resource Location in Structured P2P Networks, (Fukuoka, 2012), pp. 278–284. doi:10.1109/UIC-ATC.2012.158
31. S Ratnasamy, A Scalable Content-Addressable Network. Technical report, University of California at Berkley. www.icir.org/sylvia/thesis.ps. Accessed Sept 2017
32. A Takeda, T Oide, A Takahashi, T Suganuma, in *Proceedings of the 18th International Conference on Network-Based Information Systems*. Efficient Dynamic Load Balancing for Structured P2P Network, (Taipei, 2015), pp. 432–437. doi:10.1109/NBiS.2015.66
33. D Liu, Z Yu, in *Proceedings of the 5th International Conference on Electronics, Communications and Networks (CECNet 2015)*. Towards Load Balance and Maintenance in a Structured P2P Network for Locality Sensitive Hashing, (2015). doi:10.1007/978-981-10-0740-8_46
34. B Godfrey, K Lakshminarayanan, S Surana, R Karp, I Stoica, in *Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*. Load Balancing in Dynamic Structured P2P Systems, INFOCOM 2004, (2004), pp. 2253–2262. doi:10.1109/INFCOM.2004.1354648
35. Q Vu, B Ooi, M Rinard, K Tan, Histogram-Based Global Load Balancing in Structured Peer-to-Peer Systems. IEEE Trans. Knowl. Data Eng, 595–608 (2009). doi:10.1109/TKDE.2008.182
36. J Ghimire, M Mani, N Crespi, T Sanguankotchakorn, Delay and Capacity Analysis of Structured P2P Overlay for Lookup Service. Telecommun. Syst. **58**, 33–54 (2015). doi:10.1007/s11235-014-9872-9
37. N Varyani, S Nikhil, VS Shekhawat, in *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. Latency and Routing Efficiency Based Metric for Performance Comparison of DHT Overlay Networks, (Crans-Montana, 2016), pp. 337–342. doi:10.1109/WAINA.2016.16
38. S Ratnasamy, B Karp, S Shenker, D Estrin, R Govindan, L Yin, F Yu, Data-centric storage in sensor nets with GHT, a geographic hash table. Mob. Netw. Appl. **8**(4), 427–442 (2003). doi:10.1023/A:1024591915518
39. A Jedda, HT Mouftah, in *2015 6th International Conference on the Network of the Future (NOF)*. Enhancing DHT-based Object Naming Service Architectures with Geographic-awareness, (Montreal, 2015), pp. 1–6. doi:10.1109/NOF.2015.7333309
40. P Maymounkov, D Mazieres, in *IPTPS f01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*. Kademlia: a Peer-to-Peer Information System Based on the XOR Metric, (2002), pp. 53–65. doi:10.1007/3-540-45748-8_5
41. BY Zhao, L Huang, J Stribling, SC Rhea, AD Joseph, J Kubiatowicz, in *IEEE Journal on Selected Areas in Communications*. Tapestry: a Resilient Global-Scale Overlay for Service Deployment, (2004), pp. 41–53. doi:10.1109/JSAC.2003.818784
42. K Aberer, M Hauswirth, M Punceva, R Schmidt, in *IEEE Internet Computing*. Improving data access in P2P systems, (2002), pp. 58–67. doi:10.1109/4236.978370
43. K Aberer, A Datta, M Hauswirth, *Peer-to-Peer Systems and Applications P-Grid: Dynamics of Self Organization Processes in Structured P2P Systems*. (Springer Verlag, 2005), pp. 137–153. doi:10.1007/11530657_10
44. V Kantere, S Skiadopoulos, T Sellis, Storing and Indexing Spatial Data in P2P Systems. IEEE Trans. Knowl. Data Eng. **21**(2), 287–300 (2009). doi:10.1109/TKDE.2008.139
45. C Gross, B Richerzhagen, D Stingl, C Munker, D Hausheer, R Steinmetz, in *Proceedings of IEEE P2P 2013*. Geodemlia: Persistent Storage and Reliable Search for P2P Location-based Services, (Trento, 2013), pp. 1–2. doi:10.1109/P2P.2013.6688730

Submit your manuscript to a SpringerOpen journal and benefit from:

- ▶ Convenient online submission
- ▶ Rigorous peer review
- ▶ Open access: articles freely available online
- ▶ High visibility within the field
- ▶ Retaining the copyright to your article

Submit your next manuscript at ▶ springeropen.com