Peer-to-Peer Netw. Appl. (2015) 8:512–531
DOI 10.1007/s12083-014-0277-z

A flexible DHT-based directory service for information management

Tarciana Silva · Carlos Kamienski · Stenio Fernandes · Djamel Sadok

Received: 28 August 2012 / Accepted: 13 April 2014 / Published online: 26 April 2014
© Springer Science+Business Media New York 2014

Abstracts Information management is a key feature for the successful deployment of service architectures that involve highly distributed, dynamic, collaborative, and heterogeneous networks. Current solutions fail to meet important requirements of those networks since they have limited support for dynamicity of networks, nodes and information, or flexible information retrieval mechanisms for satisfying user's needs. In this paper, we propose a Global Directory Service based on Distributed Hash Tables (DHT) and Hilbert Space Filling Curves (HSFC) that provides distribution and flexibility for information retrieval. Performance analyses reveal that the proposed mechanisms are scalable with the number of networks, nodes, and amount of information.

Keywords Information management · Directory service · Information service infrastructure · Hilbert space filling curves · Distributed hash tables

1 Introduction

In the past decades the main communication infrastructure evolved from homogeneous interconnection of networks of stationary computers, running well-behaved applications, to a highly complex and dynamic multitude of access networks technologies. Such evolution was mainly due to the proliferation of wireless devices and new services on the top of them, running a good deal of applications with strict Quality of Service (QoS) requirements. Providing seamless mobility of sessions and services to users requires a huge amount of effort to coordinate information sharing between different types of networks (heterogeneous network access technologies) and operators. Such stringent requirements coming from different players are the driving force behind the need of services that can operate efficiently when dealing with network information gathering, aggregation, distribution, and searching.

Cooperation between heterogeneous networks has a stimulating attractiveness to the networking research community at both industry and academia, since it can leverage new business developments in the current highly competitive environments for fixed and wireless Internet Service Providers (ISP). For successful deployment of management architectures of future services that require network interoperability, a mandatory element is an Information Service Infrastructure (ISI) [15], which can provide ubiquitous and context-related information access to other specific system management components. With the emerging architecture for facilitating cross-layer information sharing between heterogeneous networks, namely the IEEE 802.21 Media Independent Handover (MIH) [6], we can envisage that even more information is expected to be available, delivered, correlated, and integrated into network interoperability management systems. Important performance factors for an ISI are resilience and scalability. Existing approaches to information management of heterogeneous networks [23] are typically centralized. For typical mobility management systems, a centralized network element periodically gathers available information, evaluates it by a given criterion set, and makes decision for the whole network or for a subset of users. It is clear that the most conspicuous disadvantage regards the central point of failure. Second, a

T. Silva · S. Fernandes · D. Sadok
Universidade Federal de Pernambuco, Recife, PE, Brazil

T. Silva
e-mail: tds@cin.ufpe.br

S. Fernandes
e-mail: sflf@cin.ufpe.br

D. Sadok
e-mail: jamel@cin.ufpe.br

C. Kamienski (✉)
Universidade Federal do ABC, Santo André, SP, Brazil
e-mail: cak@ufabc.edu.br

centralized approach limits scalability and can induce bottlenecks in highly dynamic environments where frequent exchange of information occurs. In the heterogeneous networks scenarios, an ISI that can provide access to subscriber-related data, network features (e.g., current capacity, available resources, and QoS) and offered services has not been addressed in a global or multi-domain context. In summary, in the context of network interoperability, requirements of different players, such as network operators, subscribers, applications, and service providers, must be met through a comprehensive use of a scalable network information infrastructure, through which essential components can enable management functions to higher-level decision-making elements.

This work provides a novel global distributed directory service to heterogeneous networks, where major players can register global information so that users can have access to such directory service to retrieve information related to the available resources through flexible and efficient queries. This information will be represented by means of a global information model, which can be easily extended as new global resources emerge and need to be represented. In P2P architectures, Distributed Hash Table (DHT) solutions [22] can be considered the most scalable one. However, flexible queries in pure DHT are not allowed (e.g., queries with non-exact keys or range query type). Therefore, in order to overcome this limitation, we use a different mechanism to store and query data, called Hilbert Space Filling Curves (HSFC), a mechanism that is transparent to the end user and to the directory service [12] [11]. We provide a complete description of the indexing algorithm used to insert the elements in the DHT as well as additional algorithms used to search for elements in the DHT. Our work addresses several functional requirements that a global directory service must fulfill, as follows: a) support for frequent information updates; b) support for information dynamics due to extensive collaboration among networks, in order to avoid replication of information and inconsistency; c) enabling flexible and efficient queries; d) providing scalability, fault tolerance, and robustness in a distributed environment; e) providing a suitable way to make it possible for multi-access networks to access and use the directory service.

The contributions of this paper are manifold. First, the design rationale adopted for the global directory recognizes the tradeoff between two opposite approaches: adapting an existing directory service or an existing highly distributed infrastructure. We chose the latter, adding flexible query features to a DHT using the HSFC concept. Second, we changed the key generation for DHT, providing locality for range queries and preserving scalability. Third, we built a new simulator on top of the OMNeT++ framework, implementing the main features of the directory service. Last, we conducted a performance analysis that revealed that the new mechanisms are scalable with the number of networks, DHT nodes, and amount of information. Particularly, results show that query time and number of messages are not influenced by the number of indexes and participating networks. Also, our approach is able to find all answers matching a flexible query in the DHT substrate.

The rest of the paper is structured as follows. Section 2 presents technical background and related work. Section 3 describes our novel global directory service for information management in heterogeneous networks. Section 4 provides details on the implementation of the architecture. Section 5 presents the evaluation methodology followed by performance analysis results, presented in Section 6. Section 7 discusses the main findings and insights and section 8 draws some conclusions and presents suggestions for future work.

2 Technical background and related work

Dynamic and heterogeneous networks environments require interoperability, mobility, and collaboration as the main driving forces behind their development. Mobility requirements in such environments imply that constant updating of information as well as information dynamics must be supported. Pentikousis e. al proposed an Information Service Infrastructure (ISI) capable of collecting, filtering and correlating events, and provisioning context to applications [15]. Although their proposal was in the context of mobility management, their goal was to provide services to support network information gathering and decision-making engines.

We argue that directory services such as X.500 [7], LDAP [21] are not suitable for dynamic heterogeneous environments for a number of reasons. First, they are not scalable with the number of records due to their centralized architecture. Second, they are highly optimized for reading operations, whereas updating and inserting new information are common operations in dynamic environments. Third, their containers are organized on a hierarchically static namespace, so that mobility and constant updates are not directly supported. Fourth, standard replication mechanism, an essential requirement for dealing with requirements of fault tolerant and robustness in dynamic environments, is usually not available.

Another alternative for directory services is UDDI [13], together with the Web Services technology [3]. UDDI was designed to operate in centralized environments, thus compromising the scalability, fault-tolerance and robustness requirement. A potential scalable solution with an efficient mechanism may be a Federated Web Services architecture based on a P2P solution to manage scalability [5]. However, since the original environment was designed to be centralized, new challenges arise. Replication of registries weakens the main benefits of a centralized Web services registry, since the management of those replicated registries, at different locations, adds considerable administrative overhead.

On the other hand, Peer-to-Peer (P2P) technologies, particularly DHT [22], are proven to be (self) scalable and efficient.

Lookups are performed in O (log N) hops, where N is the number of nodes in the overlay network built by DHT technology. However, it is well known that the use of DHT brings new challenges, such as the semantic organization of stored data. The way traditional DHT algorithm works (for instance, by the using of hash function) causes severe limitations in the query structure since results are only successfully returned when one has the exact key to be searched. Although some services do perform adequately in non-dynamic and/or centralized environments performing various types of queries, putting them under heavy load conditions still causes scalability issues. Indeed, network services based on DHT work in a massively distributed environment, yet they only take exact-key queries. Therefore, an important challenge is to investigate whether it is better to redesign a network directory service to add support for massive distribution and dynamicity or to adapt a highly distributed environment to support a query engine layer without changing how the network information infrastructure works. In this work, the second approach was chosen and the concept of HSFC was used to adapt this infrastructure. HSFC are used for multi-dimensional indexing in many different areas such as traditional databases, image compression, geographic information systems and the like [12] [11]. Their main feature is to perform a mapping from an n-dimensional space to a one-dimensional, while preserving locality (Fig. 1). Therefore, when one-dimensional indexes are close to each other, so they are in a multi-dimensional space. But the opposite is not true and this is called digital-causality property.

Two concepts have a major importance related to Hilbert Curve, which are derived-keys and n-points. A derived key is the one-dimensional result obtained from a set of dimensions. For instance, if there is a third order two-dimensional Hilbert space, point D in Fig. 1 represents the derived-key whose two corresponding dimensions are (000, 011). This pair (000,011) is called the n-point of the aforementioned derived-key and we call third order because there are three bits to represent each coordinate of each dimension. As long as we have more bits

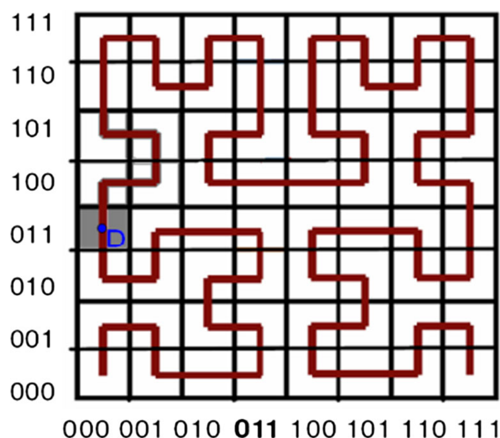


Fig. 1 Mapping between a two-dimensional space into an one-dimensional space (or derived-key)

representing the dimensions, i.e., the higher the order is, the more precise will be the representation of a point in the space. The Hilbert curve is formed in a recursive way and the number of derived-keys contained in each curve depends on its order. For instance, Fig. 2a shows a first order Hilbert Curve in a two-dimensional space with only four derived-keys (00, 01, 10, 11), whereas Fig. 2b is a Hilbert curve in the second order with sixteen derived-keys (0000 to 1111). Fig. 2c is a third-order curve containing 64 derived-keys (000000 to 111111).

A Hilbert curve in a specific order is the concatenation of its curves in the immediately previous order (or its rotation – this will depend on the state machine of the curve). This step can be observed in Fig. 2, representing the evolution from a first order curve, Fig. 2a, to a second order one, Fig. 2b, and further to a third order one in Fig. 2c. We used HSFC to add features to a DHT infrastructure in order to be make it fit to a global distributed directory for network information management.

Although there are many proposals for providing improved and flexible search techniques for P2P systems, in general they are not directly comparable to our work, since they are either based on different approaches or assume different premises. A 2011 survey [28] on P2P-based multidimensional indexing (MI) methods classify the solutions into two main broad categories: i) those that come from the MI area for centralized applications and add distribution features; and ii) those that come from the P2P area and therefore must add MI features for allowing flexible queries for similarity search. As mentioned before, our directory service for information management fits into the second category, because we first choose DHT due to its inherent distribution and scalability and then added features for flexible queries.

Ratti et al. [17] proposed a non-uniform Hilbert curve specially designed for mapping of object locations in Massively Multi-user Virtual Environments that are non-uniformly distributed. The resulting curve is non-symmetrical, whereas our directory service requires a uniform and symmetrical Hilbert curve. SiMPSON [26] is a P2P system that supports similarity search using a 3-phase process where each peer publishes a summary of its content into the network. Peers first apply a clustering algorithm to summarize their data and then they map data clusters to one-dimensional values using a modified version of a well-known indexing method. Finally, these one-dimensional values are indexed into a non-DHT structured P2P network. Unlike our proposal, SiMPSON neither supports DHT nor allows range queries. DHR-Trees [27] is a solution that comes from centralized MI for providing flexible queries in P2P systems based on Hilbert curves. However, it inherits concepts from the old B*-trees adapted to a distributed P2P network and therefore does not support DHT. Andreolini and Lancellotti proposed Fuzzy-DHT [1], which supports multiple keyword searches on a DHT substrate, but it does not support range or wildcard queries. Although queries based on multiple keywords

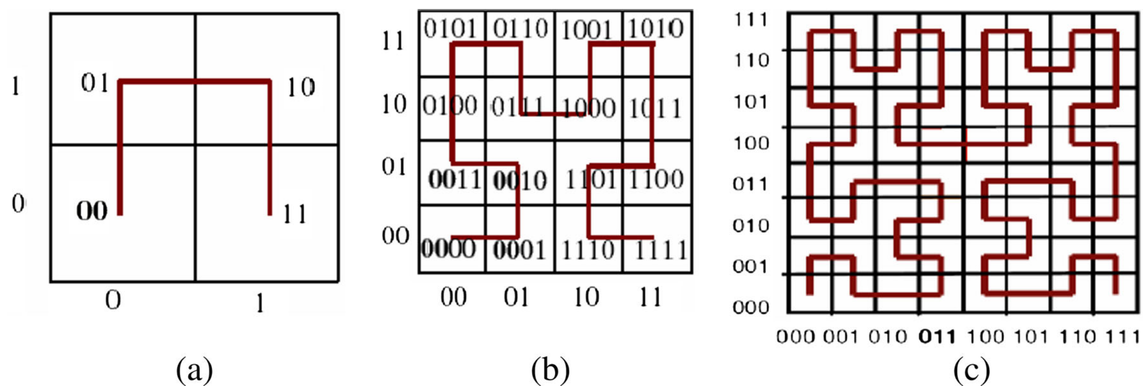


Fig. 2 Two-dimensional Hilbert curves with different orders

provide an additional level of flexibility compared to exact queries, they are clearly less flexible than range and wildcard based queries. This is because Fuzzy-DHT is aimed at providing some level of flexibility with minimum interference in the underlying DHT mechanism.

Schmidt and Parashar proposed Squid [20] that shares some similarities with our proposal, which is an evolution of an earlier work targeted for Web Services discovery [19], based on Chord [22] and strongly influenced by the mechanisms introduced by Lawder and King [11]. However, they do not present a detailed view of how indexes are generated and how data is retrieved. It is not clear, by their mapping indices mechanisms, how indexes (in the HSFC curve) are generated from the set of keywords that identifies a resource. Their scheme of publishing data is briefly described, without any further details. In this paper we focus on dynamic networks and provide further optimizations to the indexing and query mechanisms, which are particularly well described. There are various approaches for semantic searches in DHT, such as the one proposed by Zhua and Hug [29], which provides only approximated answers to queries. On the other hand, our approach is able to find all answers matching a query in a DHT.

There are some recent proposals for enabling flexible queries in peer-to-peer systems, yet they present significant differences to our proposal. Villaca et al. [25] introduced a new technique called Hamming DHT where Hilbert Filling Curves are not used. Instead it uses Gray codes and Hamming distance as a metric for similarity search. It is still based on hash functions and changes the Chord finger tables to new contents using its approach. It is highly based on Chord while our approach is independent of a particular DHT. Joung and Yang propose a complex scheme for flexible queries in a hypercube-based DHT network, based on n-gram search [8]. Their mechanism, although powerful, requires n-grams to be spread over the network and the percentage of nodes to be visited can be extremely high when n is small. Semantic Overlay Networks (SON) can also be used to retrieve information in peer-to-peer networks, but they are based on globally known structures rather than on DHT [10]. Pitoura et al. proposed Saturn [16], a multi-ring DHT aimed at concurrently dealing with efficient range query processing, load balancing and fault tolerance. Obviously, Saturn incurs in tradeoffs and additional costs to fulfill those goals. For example, there is overhead for the maintenance of multiple rings, although the paper claims Saturn incurs in no extra costs, in addition to the DHT cost, for building and maintaining routing information. Performance evaluation results do not show evidence of scalability related to the number of peers and information size. Rather, the number of messages grows linearly according to query range size. Also, the paper claims Saturn is independent of DHT mechanism, but there is no evidence of that feature.

3 A global distributed directory for network information management

Our architecture, called Global Distributed Directory (GDD), is described in this section.

3.1 Scenarios and requirements

Requirements for a GDD service have been identified through scenarios, where heterogeneous networks need to collaborate through composition before users can themselves collaborate with each other. Here we present the game service scenario as an example (Fig. 3), inspired in dynamic networks collaborating on the fly. In this scenario, Tom (connected to Net1) has already gained access to a game service hosted by Net3 (Net1 and Net3 has started a collaboration) and he wants to play with Mary. Since our focus is the directory service, we highlight here that three queries are sent to the directory. The first query is to find Mary and the answer is that she is connected to Net2. The second query aims at learning whether Net2 provides the game service, which is not true. So, a third query is needed, for finding out whether is there around any available network which provides the game service and finally Net3 is found out. Once the agreements are made between Net2 and Net3, Tom and Mary can interact and play the game.

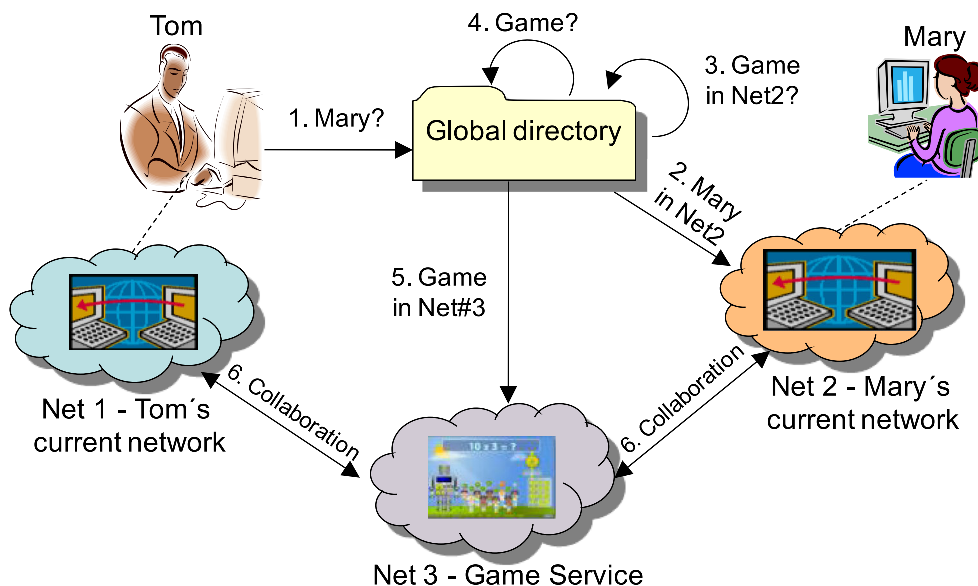


Fig. 3 Game service scenario

Based on this game service and other two scenarios not shown here due to space limitations, we derived some functional and mandatory requirements that a global directory service must fulfill, as follows:

1) Support for frequent information update;
2) Support for the dynamism of information due to extensive collaboration among networks, avoiding replication of information and inconsistency;
3) Enabling flexible and efficient queries;
4) Providing scalability, fault tolerance and robustness in a distributed environment;
5) Providing a suitable way to make it possible for networks, with different access technologies, to access and use the directory service.

3.2 Information model

Figure 4 presents the information model we use for current version of GDD, which is simple enough for being easily understood and just fits our needs with no added complexity. The information model contains four entities, service, user, network and node. It is worth stressing that a service can contain a set of other services (e.g., the self-relationship presented in Fig. 4). A service entity can represent a voice application, a security service, and so on. A User is the entity using the service provided by Network through a Node. Network, in turn is a collection of Nodes, which represent machines, for instance a smart phone or a tablet.

There is no standardization for a global information model to represent heterogeneous networks and this model is considered a case study. Therefore, it might be extended or customized as long as new network information or resources have been being discovered. Optionally, this model might be also described in DMTF CIM [4] if a formal and standard format is needed for other purposes. We assume that each service is identified through a well-known service definition, recognizable by all existing networks. Although there are many efforts to standardize services [9], further analysis of such research topic is out of scope of this work.

3.3 Architecture

GDD was designed in such a way to have a distributed structure, composed of nodes from each network, which can have more

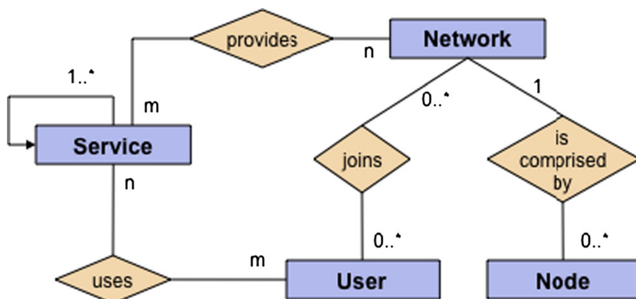


Fig. 4 Information model

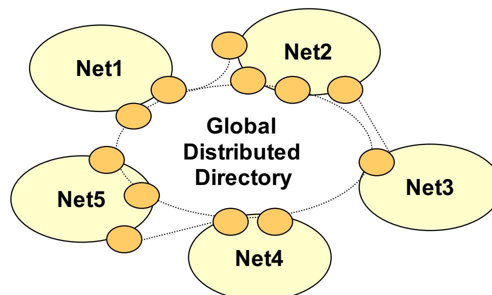


Fig. 5 GDD general architecture

than one node participating in the global directory. These nodes form an overlay structure as shown in Fig. 5. Each network must publish its global information according to the information model. Also, a network can access GDD through another participating network, which may be necessary due its own resource limitation. However, this may be accomplished by a previous collaboration agreement between these networks.

3.4 Insertion structures (indexes)

We derived indexes to represent the information model, and in particular, the entities and relationships between them, as shown in Fig. 6. NetID represents the network identifier, whereas the USER index is to be used for storing the network a user is currently connected to. All information stored in the directory that follows an index structure presented in Fig. 6 is called an index instance. For example, Mary and her network is an index instance of index USER.

The SERVICE_USER index stores information derived from the relationship between the entities Service and User. This index is useful, for example, in order to find out whether a user called Anne is using a TV display service in her network. The PROTOCOLS index aims at storing the current protocols available in a specific network. This index is derived from the relationship between the network and node entities, where node entity in this particular case is represented by the protocol the node uses. This index can be used when a requester wants to find out which networks support TORA [14] or OSPF protocols. Finally the SERVICE index accepts queries about which networks provide a specific service of a specific subtype and the FULL_MODEL index stores all available information associated with the network identifier.

GDD indexes are classified according to their number of dimensions or metadata (underlined and bold in Fig. 6). These dimensions are associated with a value, which is, except for the FULL_MODEL index, the network identifier (represented by the field NetID). When multiple networks present index instances in common, the field NetID consists of a set of network identifiers. As an example, lets us consider two networks, identified by NetID 54 and 76, and a user service called "fifa", whose subtype is entertainment. If this service is only provided by these networks, the content of the NetID field may be "54, 76".

3.5 Query examples

Figure 7 depicts some examples of queries related to the SERVICE index. GDD accepts a variety of requests that sometimes do not contain all metadata filled in, because requesters do not know or need all the keywords associated to the involved indexes, but only part of them. For example, a network can be dependent on the service name to collaborate through composition with another network, with other parts of the information (e.g. the subtype for this service) being unnecessary. Hence, we can have partial queries, involving only a subset of metadata that make up the index.

In the first query, the directory user sends a request for finding out networks (any network, since the NetID field content is "???") with the exact information specified in the fields service and subtype. In other words, this query is for a network that offers a service called "fifa game", in which subtype is entertainment. In the second query, the user requests a service which begins with "fifa", and whose subtype begins with "enter". The third query specifies any service whose subtype is "entertainment". In this case, only one dimension was specified in a two-dimensional index. We can also specify ranges for dimensions in an index directory, as shown by the fourth query, where the request is for a network which owns a user service whose name begins with a word between "fifa" and "fife" and whose subtype is entertainment.

4 Implementation of the GDD architecture

This section describes the implementation of GDD using a DHT infrastructure. Since queries may not have exact keys, the first step to enable such global directory in a DHT infrastructure is to enhance the DHT mechanism to work with flexible queries. We consider a query to be flexible when one can discover a network with specific services without the exact values that were used to register it in the directory. In order to enable flexible queries, the GDD architecture implementation is based on HSFC and uses its own mechanism to prepare the data to be inserted, through an indexing algorithm. Data is retrieved from the DHT by a search algorithm also presented in this section.

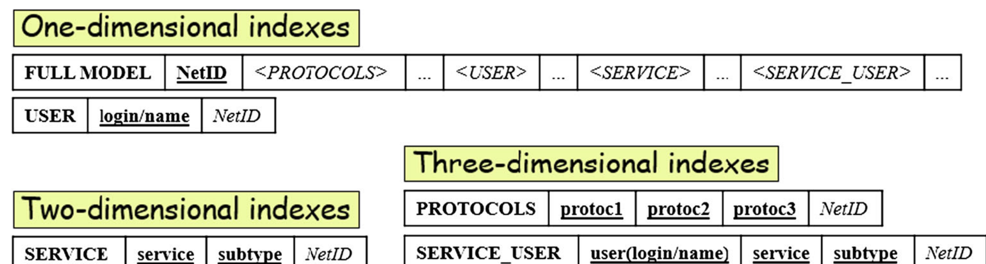


Fig. 6 Examples of indexes in GDD

1

SERVICE	service	subtype	*NetID*
SERVICE	fifa game	entertainment	???

2

SERVICE	service	subtype	*NetID*
SERVICE	fifa*	enter*	???

3

SERVICE	service	subtype	*NetID*
SERVICE	*	entertainment	???

4

SERVICE	service	subtype	*NetID*
SERVICE	fifa*-fife*	entertainment	???

Fig. 7 Examples of queries

4.1 Indexing algorithm

In GDD one can think of each directory index as a multi-dimensional space as depicted by Fig. 8. Our indexing algorithm translates these dimensions into a Hilbert curve derived-key, which will be in turn used in DHT structure. Except for the FULL MODEL index, where the NetID field is itself the key of this index, the NetID field in the other indexes (see Fig. 6) represents the value associated to each key inside the DHT, and this value is (are) the network (s) owning the specific resources or relationships specified by the instance.

Before running the Indexing Algorithm we first transform the dimension values represented by names into dimension values represented by bits. By doing this we have the n-points, which are the input for this indexing algorithm (Fig. 9). We work with a 32nd order Hilbert curve for a two-dimensional space, which justifies the use of the state machine represented on Fig. 10. There is a specific state machine for each dimensional space. We also limited each dimension to a 32 bit number and hence we have 64-bit derived-keys. Index fields are limited to accommodate at most six characters (letters) where each character will be represented by five bits, summing up 30 bits. The remaining two bits are left for future use. Letter ‘a’ represents 00001 (1), letter ‘b’ 00010 (2) and so forth until letter ‘z’, corresponding to 11010 (26). The correspondence between the letters and the bits is done from the

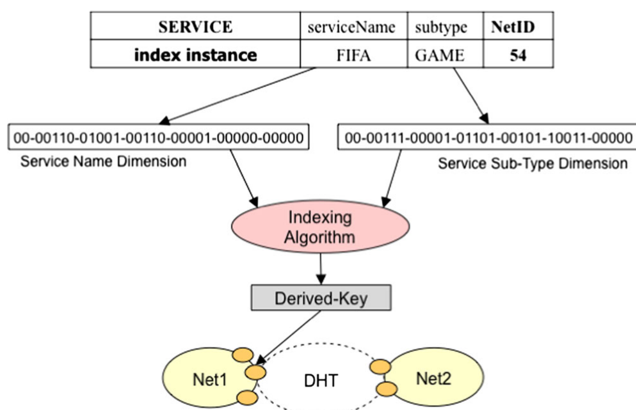


Fig. 8 Insertion of an index instance in GDD

Indexing Algorithm

INPUT: Dimension values in bits

OUTPUT: Derived-key

```
1 select the state machine to be used
2 derived-key <- blank
3 while (∃ unread bits in each dimension) {
4    check current state of state machine
5    key <- get derived-key for n-point
6    derived-key <- derived-key << 2
7    concatenate key into derived-key
8 }
```

Fig. 9 Indexing algorithm

third most significant bit as shown in Fig. 8. It is important to emphasize that the indexing algorithm can be straightforwardly extended to larger Hilbert curve orders with any arbitrary dimension values.

After having the dimension values represented by bits, the indexing algorithm from an n-point to a derived-key is performed step by step as the bits in each n-point dimension (in this case, only x and y) are read. As an example, let us consider the n-point <110,100>. The initial state is always the state 0, corresponding to the root of Hilbert tree as shown by Fig. 10. The current bits read from the n-point are one for the x dimension and one for the y dimension [<110, 100>]. Then, according to state 0 of the state machine represented by Fig. 10, this current n-point corresponds to the derived-key 10, which is in turn left-shifted by two positions (line 6 in Fig. 9) and the next bits are read: one for the x dimension and 0 for the y dimension [<110, 100>].

According to the state machine for a two-dimensional space (Fig. 10), the next state from the n-point 11 (this is the first n-point read) of the state 0 is the state 0 itself. In the state 0, we may observe that the derived-key corresponding to n-point 10 is 11. Hence, the current derived-key is 1011. As we proceed, the next state for n-point 10 (this is the second n-point read) in the state 0 is state 2. The next n-point 00 from [<110, 100>] corresponds to the derived-key 10 in the state 2.

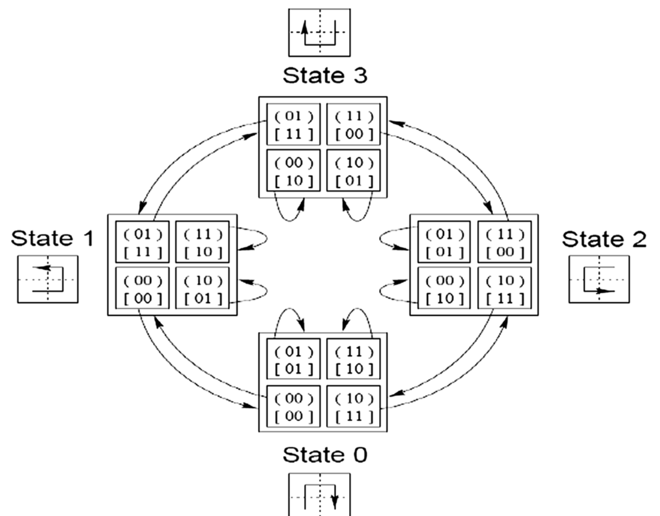


Fig. 10 State machine of a Hilbert curve in a two-dimensional space

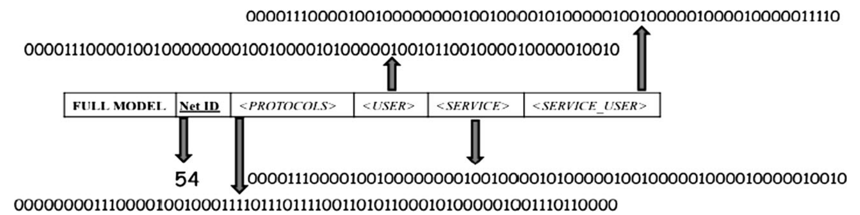


Fig. 11 Example of a FULL_MODEL index with its key and value

Therefore, the final derived-key corresponding to the n-point [<110, 100>] is 101110.

The indexing algorithm just described is used to calculate the keys of all indexes except the one-dimensional ones like FULL MODEL and USER as described in the beginning of this section, since in these cases the 'key' to be inserted in the DHT is, respectively, the network identifier itself and the login of the user. Figure 11 shows an example of the contents of FULL MODEL index, where NetID (key54) is associated to a value that corresponds to the concatenation of each index instance corresponding to PROTOCOLS, USER, SERVICE, SERVICE_USER indexes. Each of these index instances values (the numbers in binary) corresponds to the derived key calculated from the dimensions of these indexes, as described at section 4.1, Fig. 8. In this way, FULL MODEL index, or the network identifier corresponding to its key, could be used to retrieve the complete set of network information.

Thus, with regard to the information insertion in the overlay, our DHT mechanism is basically the same as traditional one with the difference that keys are not generated by a hash function anymore but by the indexing algorithm (explained in section 4.1). The exception is for unidimensional indexes as already explained. The reason is that, using the hash function, information is consistently spread over the existing nodes and we want to preserve locality. Similarly, in our search process mechanism, instead of simply applying a hash function in the name that identifies a resource in order to get the key and search it in the overlay, as is done by the traditional DHT mechanism, we apply the search algorithm (explained in further sections) in the index dimensions (given as entry) to find all possible derived-keys in the Hilbert space that are also stored in the DHT. Thus, there is an interaction between our search algorithm and the search algorithm of a traditional DHT mechanism, where this last one is used only to verify if a specific derived-key exists in the overlay. As further explained in section 4.3, our search algorithm may request the traditional one many times.

4.2 Search algorithm

The main problem we have to deal with here is implementing flexible queries. In a traditional DHT, information that satisfy a query such as "fif*" may, in the worst-case scenario, be spread over the entire network, due to the hash function used in DHT. In addition, there will not be semantic relationship between information registered in the DHT, making flexible queries unfeasible due to the lack of efficiency in the search mechanism. Therefore, our search algorithm supports flexible queries, while keeping the scalability of a DHT.

Figure 12 shows a macro view of the search algorithm. Suppose a range query, shown in Fig. 12, composed of dimensions 'fif*' and 'gam*'. We can deduce its bounds as 'fifaaa…fifzzz' and 'gamaaa…gamzzz' respectively, where fifaaa and gamaaa are the lower query bounds whereas fifzzz and gamzzz are the upper. Afterwards, the query bounds specified with names are passed into query bounds specified with bits. In this way, once we have the complete bit values corresponding to each query bounds, we can delimit the space in the Hilbert space that satisfies our query. For instance, Fig. 14a shows the gray space delimited by the bounds {(000,100), (111,100)}.[1] In this case, all the points in the Hilbert curve inside the gray region might be candidates to the result of the query specified by the bounds. But it is necessary to check if any of these points are also stored by the nodes in the overlay. This is what is done by our search mechanism. Firstly, the next-match algorithm (section 4.3) is applied. This one is used to discover the first point in the Hilbert curve belonging to the query region. Whenever a point in the Hilbert curve belonging to the query region is found by the algorithm, we will call this point as the current match. The next step is to query the overlay to discover which DHT node is responsible for the newly found match.

Once the node responsible for the point is found, the next step is to query the data of this node in order to check if this match is a real valid data (exists) or a key (data element) that belong to the query region. The first point found can help on this. Therefore, one can say that the search algorithm mainly involves two procedures, as mentioned earlier in this section: the searching of a DHT node responsible for a point in the Hilbert curve that is inside a specific query region, and once this node is found, the finding of all matches satisfying the query (described in Fig. 13) and stored by the overlay.

Data-elements (keys) are stored in ascending order inside the node data structure (that we will call the node data storage) in the overlay and they may or may not belong to the given query. The current match is then compared to the key stored in DHT node storage – as shown at lines 3, 6 and 19 in Fig. 13. If it is identical, we find an element that belongs to the query region and it is stored in the DHT. Otherwise, if the current

[1] A similar space is delimited by the bounds formed by the range query shown in Fig. 12 but this one is more difficult to see as it is necessary a 32nd order Hilbert curve for a two-dimensional space.

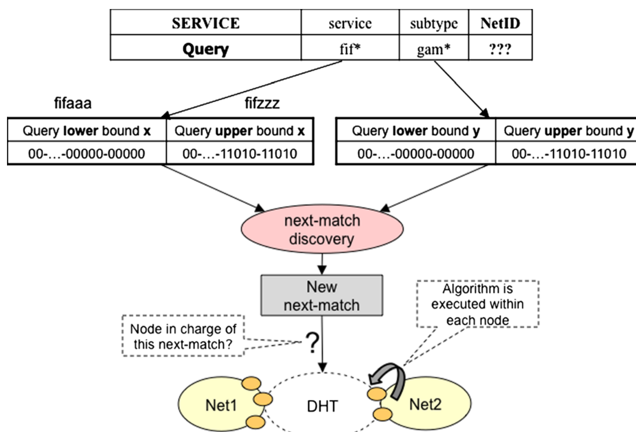
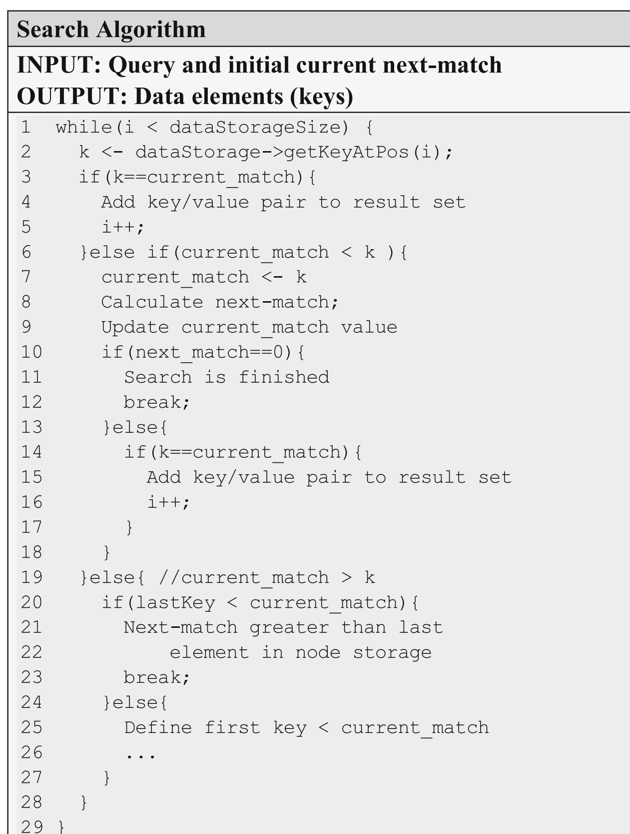


Fig. 12 Search algorithm (*macro view*)

match is less than the element stored, we have to calculate another match, passing the element stored as the current match (parameter) to the next-match discovery algorithm (cf. lines 7 and 8, Fig. 13). In the case that next-match calculated is zero (cf. line 10, Fig. 13), this means that the node and the network do not have any data element that satisfies the query, and the possible elements satisfying the query would be between the previous found match and the element stored in the node data storage indicated by i index. Otherwise, if the new match found is not zero (cf. line 13 of Fig. 13), this means that this match is minimally greater than the previous match and is also



Search Algorithm

INPUT: Query and initial current next-match
OUTPUT: Data elements (keys)

```
1  while(i < dataStorageSize) {
2    k <- dataStorage->getKeyAtPos(i);
3    if(k==current_match){
4      Add key/value pair to result set
5      i++;
6    }else if(current_match < k ){
7      current_match <- k
8      Calculate next-match;
9      Update current_match value
10     if(next_match==0){
11       Search is finished
12       break;
13     }else{
14       if(k==current_match){
15         Add key/value pair to result set
16         i++;
17       }
18     }
19   }else{ //current_match > k
20     if(lastKey < current_match){
21       Next-match greater than last
22           element in node storage
23       break;
24     }else{
25       Define first key < current_match
26       ...
27     }
28   }
29 }
```

Fig. 13 Search algorithm inside a DHT node

in the query region. Then we compare this new current match with the same element in the data storage (line 14, Fig. 13). If it is identical, this element is in the query region. Otherwise we continue and repeat the same procedure with the next element in the node data storage.

If the key in the DHT node is smaller than the current match (cf. line 19, Fig. 13), we compare whether the current match is greater than the last element stored (line 20), for processing time optimization purposes. If so, the search in this node is ended and we have to find the next node in DHT that could have more matches satisfying the query. If not, we can apply an optimization mechanism to find the most immediately key smaller than the current match (cf. line 25). This is necessary since performing a linear search in the node data storage for finding a key element that satisfies the current match is costly in terms of processing time.

Depending on how many data elements a node stores, the comparison process between the current match and each element stored in node data storage could be also too costly. In order to minimize the time of the queries, not querying the data storage of the node linearly, and to decrease the processing time, we introduce some optimizations. The first optimization is to determine where to begin the search (or the best point to begin when we have a current match) in the data node storage. This can be very useful in the case that the current match is greater than the first element in the data storage and there are too many data elements between them. We can optimize by dividing the length of the node data storage into two equal size parts and comparing the derived-key in the middle (of the data storage) with the current match. If they are equal, this means that we find the first element in the query region that also exists in the network. On the other hand, if the key of the element is still less than the current match, we continue the same process, now dividing into two the second half part. If, during the process, the key of the element becomes greater than the current match, we stop at the point immediately before, where the key of the element was smaller. In both cases, we do the process until smaller element immediately before the current match is found. This is the appropriate point to start applying the next-match discovery algorithm (see section 4.3) in order to verify if a specific stored element is or is not in the query region. After finding the element where to begin the search, comparison between the key of this element in the node data storage and the current match is started, according to Fig. 13.

The procedure described in Fig. 13 is performed with all the keys stored inside the DHT node (variation of index i). In summary, there are three possibilities during the searching: (a) One finds a key lesser than the current match and thus the search continues with the next key in the node data storage being compared with current match. (b) one finds a key in the data storage equal than the current match. This represents a match, key inside the DHT node that is a derived-key

contained in the query region. (c) One finds a key greater than the current match. In this case, one needs to recalculate a new match where this key is the current match parameter of the next-match discovery algorithm. (c.1) In the case of a zero result for the new match, the search can be considered finished as explained earlier.

After ending the search in the node data storage and match zero be reached, we return all the keys found to the node that requests the query. In the case that match=0 was not reached inside the node data storage, this means that the search needs to continue since there may be more data elements belonging to the query region in other nodes of the overlay network. In an overlay DHT network, all nodes are organized in ascending order according to their identifiers. A node is responsible for the storage of all the keys in the range between the identifier of its predecessor and its identifier. Hence, when a query with a non-zero reached match returns from specific node data storage, the query needs to continue, and it is necessary to calculate a new current match, passing its identifier plus 1 as a parameter as well as the query itself for the next-match discovery algorithm. After doing this, if the new current match just found is zero, then no more data elements can exist in the overlay and we can collect the metrics for the query. Otherwise, in the case that a non-zero current match is found, this means that this match is minimally greater than the previous match and is in the query region also; hence the node responsible for this current match has to be queried because it can store data elements satisfying the query. Another lookup is sent to the overlay, now for the node responsible for this new next-match, with the new current-match found and the query, and then repeat the algorithm inside this new node data storage. The algorithm running into the node that is requesting the query is known as macro search.

4.3 Next-match discovery algorithm

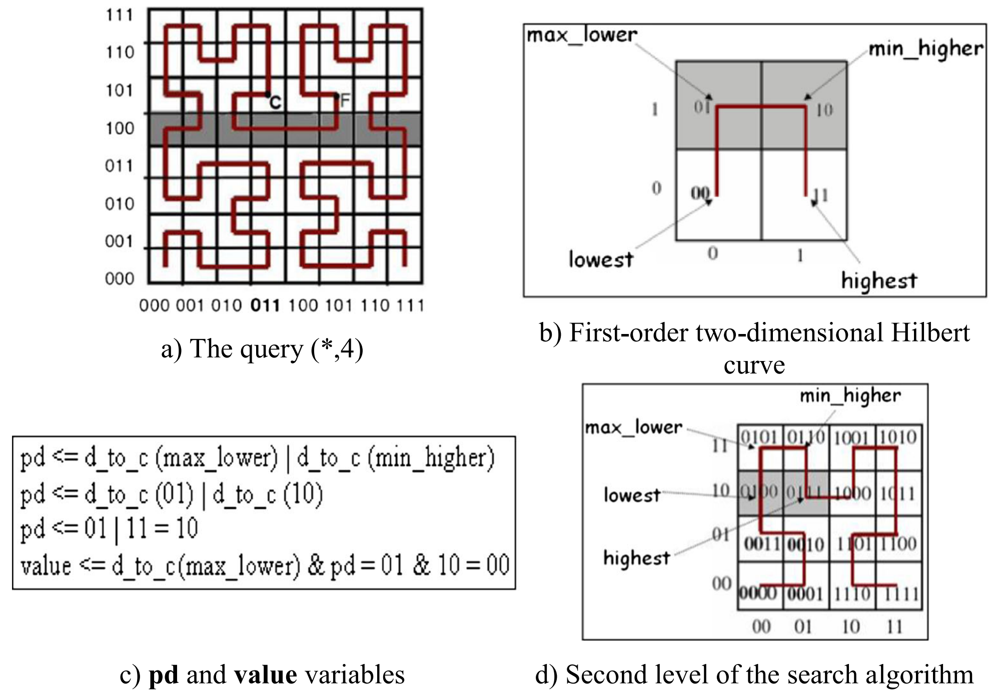
The next-match discovery algorithm aims at finding a derived-key that belongs to the query region and that is minimally greater than the value passed as a parameter. Although this parameter is not necessarily a value inside the query region, we call it the current match. As for the derived-key resulting from the execution of the algorithm, it is always an element inside the query region. We rather explain how this algorithm works through an example. Consider the query (*, 4), assuming a third-order two-dimensional Hilbert space as depicted in Fig. 14a. Firstly, we translate (*, 4) into (*, 100)=(000,100)–(111,100).

Initially, the current search space corresponds to the root of the Hilbert tree represented by the first order of the Hilbert curve shown in Fig. 14b. Therefore, the current-level is 1; and the query region in this level (that is, the current query region) corresponds to (01) - (11), since we retrieved from the initial query [(000,100)–(111,100)] the most significant bits in each dimension. Point C in Fig. 14a represents the derived-key 28 (011100), in the third order, and it is the current match. The range between lowest and max_lower is known as the lower bound, while the range between min_higher and highest, the upper bound [11]. In the case of a two-dimensional space, there are only two values in both the lower and upper bounds. However, if we consider a three-dimensional space, there are more values between lower and upper bounds.

Afterwards, the pd (partitioned dimension) variable is calculated and shown in Fig. 14c, where '|' is the exclusive or operation and d_to_c is a function that maps the corresponding derived-key, into a n-point. The result found for pd is '10' which means that both lower and upper bound have a common value in the x axis. In order to know the lower bound we calculate the variable value as shown in Fig. 14c, resulting in '0' which means that lower bound has $x=0$, which indeed is observed in Fig. 14b. As min_higher and highest also have a common value in the x axis, one may conclude that $x=1$ for the upper bound values.

In order to know if the current query region (or cqr) intercepts the lower or upper bound, we pick up the current query region in the first order Hilbert curve (01) - (11), where 01 and 11 are the lower and upper bound of the query region respectively. Then, it is possible to see that the lowest current query region has $x=0$ and thus is intercepted by the lowest – max_lower bounds. On the other hand, the upper query region has $x=1$ and therefore is intercepted by the min_highest-higher bounds. But we need to know which exact values in these bounds are intercepted by the query region. First of all, we have to compare if the lower and upper cqr are equal to 00 and 11, respectively. If this is the case, it means that all bounds (lowest, max_lower, min_higher, highest) intercept the query region and then the search should proceed in each of these corresponding quadrants. For the example, this result is negative and hence we can conclude that max_lower (n-point 01) intercepts the lower cqr and min_higher (n-point 11) intercepts the upper cqr.

However, we also have to analyze the current match, which is 011100 (point C in Fig. 14a). The derived-key in the first order Hilbert curve for this current match is '01'. This derived-key corresponds to the quadrant 01 in the first order Hilbert curve. This means that the search algorithm has to proceed in the quadrant space 01, which intercepts the query region and is minimally greater than the current match. The rule is to firstly proceed in the minimum quadrant satisfying the query to, in the case of a not found match, backtrack to the other remaining greater quadrants. Then the search will proceed in the quadrant of the max_lower bound (01) which gives the first incomplete next_match: 01????. The search in level 1 is finished and then we proceed to the second level as shown in Fig. 14d. In the case that the backtrack occurs and the algorithm is executed in the other remaining quadrants (satisfying the query) and no data element meets the requirements as the



a) The query (*,4)

b) First-order two-dimensional Hilbert curve

c) **pd** and **value** variables

d) Second level of the search algorithm

Fig. 14 The next-match discovery algorithm in action

next-match, the algorithm returns zero, meaning there is no element satisfying the query minimally greater than the current next match.

From the second level on, one has to calculate the query region from the current query region and the current search space (css), which represents the space in the current Hilbert quadrant where the search is occurring. In the second level, there is as current query region the bounds (00, 10) {lower cqr bound} and (11, 10) {upper cqr bound}; the current search space corresponds to quadrant 01 in the first level, which is, in the second level, (00, 10) in the lower css bound and (01, 11) in the upper css bound. Note that the css region was formed by putting the corresponding x from the quadrant in the first order (01) in the most significant bit of x coordinates of both css lower and upper bounds, whilst the corresponding y from this quadrant (01) in the most significant bit of y coordinates (of both css lower and upper bounds). Then we complete x and y coordinates of lower css bounds with ‘0’ since it is the ‘lower’ and two bits are necessary to be in the second-order Hilbert curve. On the other hand, x and y coordinates of the upper css bounds are completed with ‘1’.

After correctly calculating the current search space, one needs to check if cqr is equal to css. This comparison is always required when dealing with the next to last level (in this case this is true because the current level is 2 and the next to last level, 3). If so, one must consider all the quadrants in the cqr (or css) as possible valid quadrants to perform the query. But it is not the case for query (*, 4) so that the intersection described in Fig. 15a is required.

In Fig. 15a, xcss_l corresponds to the x coordinate, in the lower bound, of the current search space, while ycss_l corresponds to the y coordinate, also in the lower bound, of the current search space. The values for xcss_u and ycss_u follow the same logic but for current search space of upper bounds. These coordinates represent the current query region and in the current level (2) css={(00,10),(01,11)} and cqr={(00,10),(11,10)}, so we have xcss_l=00, ycss_l=10, xcss_u=01 and ycss_u=11. In addition, xcqr_l=00, ycqr_l=10, xcqr_u=11 and ycqr_u=10. The resulting query region for the intersection between css and cqr is (00, 10) for the lower bounds and (01, 10) for the upper bounds. Then we have cqr resulting as {(00,10), (01,10)}.

As are result, the values found for pd and value variables are ‘10’ and ‘00’, respectively. Again, it means that bounds

```
If xcss_l > xcqr_l  → xcss_l else xcqr_l
If ycss_l > ycqr_l  → ycss_l else ycqr_l
If xcss_u > xcqr_u → xcqr_u else xcss_u
If ycss_u > ycqr_u  → ycqr_u else ycss_u
```

a) Intersection between **css** and **cqr**

```
pd <= d_to_c (max_lower) | d_to_c (min_higher)
pd <= d_to_c (01) | d_to_c (10)
pd <= 01 | 11 = 10
value <= d_to_c(max_lower) & pd = 01 & 10 = 00
```

b) Calculating **pd** and **value** variables

Fig. 15 The next-match discovery algorithm (part 2)

have the x value in common, which is 0 for the lower bounds, as depicted by Fig. 15b. We can conclude from the current query region {(00, 10), (01, 10)} that only lowest and highest Hilbert quadrants intercept the cqr. Since the value of derived-key (011100) in the second level is '11', one may choose the highest quadrant in order to continue the search because the corresponding derived-key of the highest quadrant is also 11, minimally greater than the current derived-key in the second level. The search continues upwards from the highest quadrant in the second level. And the next_match is now 0111??. Finally, the algorithm proceeds to the third and last level and the same steps are performed repeatedly.

5 Evaluation methodology

5.1 Simulation environment

We performed a simulation-based performance analysis to validate GDD using the Over Sim simulator [2]. Over Sim is a flexible discrete event simulation framework for the exchange and processing of network messages based on OMNeT++ [24] supporting several structured and unstructured peer-to-peer protocols such as Chord [22] and Pastry [18]. A simplified view of the Over Sim architecture is shown in Fig. 16. According to their terminology, the DHT tier implements put and get functions as well as the storage of the keys a specific node owns (and the replication of them), whereas the Overlay tier is responsible for providing functions such as the organization of keys in finger tables of the nodes when a node leaves or joins in the network.

The Application in Tier2 is a user of the DHT module and GDD is an example. The Underlay provides the basic underlying network model that sends data packets directly from one overlay node to another by using a global routing table [2]. The Global Observer is a generic module that maintains a global view of the overlay network during the entire simulation.

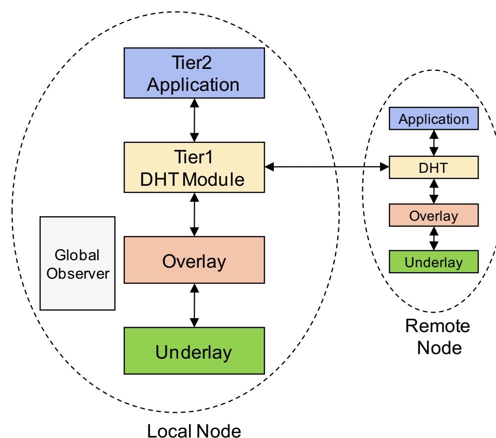


Fig. 16 Simplified oversaw architecture

5.2 Implementation of GDD in over sim

The development of GDD required a new application in the same layer of *Tier2* as well as a specially modified version of the DHT module, using Chord (Fig. 17). GDD mechanisms such as Indexing, Searching and Discovery Next-Match algorithms, explained in Section 4, were included in the DHT module to be used by the new *Tier2* application: the Indexing Algorithm, instead of the traditional DHT *put* function and likewise the Searching Algorithm, which in turn uses the Discovery Next-Match algorithm, instead of the traditional DHT *get* function. This new *Tier2* application is the one used in our experiments presented in Sections 5.3 and 6.

In GDD a network can be represented by an overlay node or by a set of overlay nodes. We can also have multiple networks accessing GDD through a unique overlay node. Each network is seen from the point of view of its information model, and thus its topology and other characteristics are not taken into consideration. GDD supports information storage by network nodes as well as queries (including the flexible ones). Since GDD is all about answering to queries, we will provide an explanation of the query event as an example of the changes we made in Over Sim to develop our simulation environment.

Queries are generated in the following way: for each index inserted, one exact query is generated, two invalid queries (substrings of the dimensions of the index, appended by an invalid alphabetic character such as '$'), and four range queries. In order to generate a range query, a random number is generated between 0 and the length - 1 of each index dimension (number of characters in each index dimension). After that, a substring for each dimension is created with a "*" character attached, where its length is a random number. The set of generated queries is stored in a data structure within the Global Observer (not presented in Fig. 17) and at the same time index values are inserted into the DHT. At simulation time, queries are randomly chosen by picking one of them in the data structure using a uniform distribution (ranging from 1 to the length of the data structure where the queries are stored) that will return the element to be searched. When there is a query event (Fig. 17), a query is picked up from the Global Observer and the execution begins.

The first step is finding the first next-match for this query (or the first point in the Hilbert curve satisfying the query) using the current match equal to zero. The next-match is obtained from the Hilbert library, also used by the macro search. After that, message #1 is sent to Tier1 to find data-elements satisfying the query. It is received by Tier1, which sends message #2 to the overlay for finding the node responsible for the first next-match. The overlay answers with message #3 and sends message #4 to the node that answers the request in order to execute the search inside the DHT. The DHT node receives it and executes the search. Afterwards,

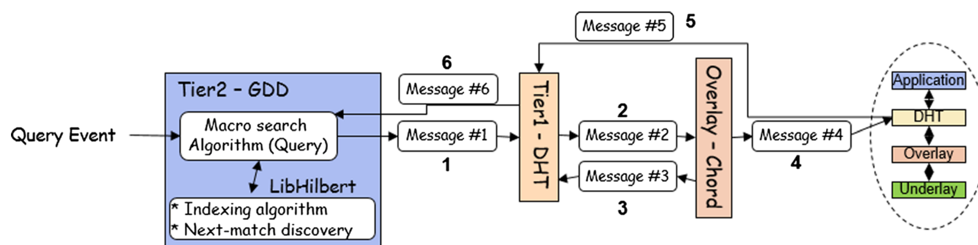


Fig. 17 GDD mechanism mapped into oversaw for query processing

message #5 is sent with the results of the search and received by the source node (the requester), which sends message #6 to Tier2 with the same results of the previous message. The message is received and processed by the macro search algorithm, which can decide whether the search process is finished, or not. In the latter case, message #1 is sent again and the process goes on.

5.3 Experimental design

The following metrics were adopted to evaluate the performance of the GDD mechanism:

a) Number of data elements: Total number of data elements (or keys) found for each type of query. In the experiments we have exact queries (for specific information), invalid queries (for specific non-existing information) and range queries.
b) Number of data nodes: Number of nodes where data elements to a specific query are found. It is better when the number of data nodes is around the same number of processing nodes. A best result is when the result of the rate number of processing nodes/number of data nodes is around 1, which means that whenever the algorithm inside a node is executed (i.e., it is a processing node), data elements are found (i.e., it is also a data node).
c) Number of processing nodes: Number of nodes whose data storage is being queried during a specific query, i.e. the number of nodes that effectively take part in the query, by processing the query and searching for matches. In other words, processing nodes are all nodes that take part in the search for data elements in a query, whereas data nodes are the ones that effectively store queried data.
d) Number of messages in overlay: Number of messages sent to the overlay for each query that search for the node responsible for a specific key during the processing of the mechanism. In exact or invalid queries, there is only one message sent directly to the overlay.
e) Number of messages per query: Total number of required messages to process a specific query. The exchange of six messages is required for a lookup for an exact key. A range query may require an unknown number of exact queries to be finished and therefore the nodes may have to exchange a high number of messages.
f) Query time: the time for each query to be finished.

Table 1 shows factors and levels used in the performance analysis. In different experiments we varied the number of index instances, the query inter arrival time and the number of participating networks. The choices factors and their values (levels) were based on previous experiments and the observation of their results. For example, it was observed that the use of a number of networks higher than 1,000 does not make any difference to the results (this can be observed in section 6). The query inter arrival time follows an Exponential distribution, i.e. a Poisson arrival rate.

6 Results

The performance analysis of GDD is performed by observing how metrics presented in section 5.3 behave as we vary each factor at a time. In this section, graphs depict a metric by its final mean values according to the variation of the respective factors. The 99 % asymptotic confidence intervals were calculated but the vertical bars are not shown since they are not visually significant.

Each simulation generates an amount of exact (Q0), invalid (Q1) and range (Q2) queries, which are executed during the simulation. Invalid queries are important because they represent a possible action of the user, i.e. the user requests information not available in the directory. Thus, the directory service must return a negative answer for invalid queries.

In the sequence we present results showing the effect of varying the number of index instances and the number of networks. We also varied the query load, with different query inter arrival times, but they are intentionally omitted since the results were similar to those for the number of instances.

Table 1 Factors and levels

Factor	Levels
Number of index instances	10, 20, 40, 70, 100
Query inter arrival time (milliseconds)	20, 30, 50
Number of networks	100, 250, 500, 750, 1,000

6.1 Effect of the number of index instances

In the first experiments we vary the amount of information stored in the directory. The number of index instances assumes five different levels (10, 20, 40, 70 and 100), the query inter arrival time is 30 milliseconds and the number of networks is 100 (i.e., also 100 DHT nodes). The duration of each simulation experiment is 1,000 sec and the results are presented in Fig. 18.

As we increase the number of index instances, there is also an increase in the number of data elements found for Q2 query (Fig. 18a), which is an expected result because we have more information stored. Q0 and Q1 queries remain with the same number of data elements (keys), respectively 1 and 0, since we always find one data element in an exact query (considering that there is no replication in the DHT) and there is no data element in an invalid query. The same happens to the number of data nodes, where there is only one for exact queries and none for invalid query. In addition, the other metrics are the same for both exact and invalid queries, except for the query time. We have the same number of processing nodes (1), the same number of messages (6) and messages sent to the overlay (1). The metric values for exact and invalid queries are the same for all the experiments – that is an expected behavior –, except for the query time metric. Hence all the comments made are relative to range queries (Q2). However, we are showing Q0 and Q1 in all figures in order to make it easier to compare them with the range queries (Q2).

In Fig. 18b, we observe a slight increase in the number of data nodes when the number of index instances in each index varies from 10 to 40, the metric remains almost the same between 40 and 70 and increases again from 70 to 100 index instances. This is a positive result showing that data tends to spread more in the GDD system (with more data nodes found in each query) as more information exists, whereas the mean value of data nodes and processing nodes found during the processing of queries does not represent more than 15 % (seeing y axis and considering a range of 100 DHT nodes) of the total of data nodes available for storing information. This is due to the use of HSFC concept, since they concentrate the semantically related data in a part of the space. This fact does not imply that only 15 % of the nodes store valid information. Figure 19f depicts the distribution of data elements among all nodes. The x axis represents the nodes (100) and y axis indicates the number of data elements that node owns. In this case, almost all nodes store information, although the distribution is asymmetric.

Hence, we can conclude by the results in Fig. 18b that it is only necessary to query, on average, less than 15 % of the total nodes in GDD in order to find data elements satisfying the queries, which is a very important result because few nodes have to be queried in the whole network in order to return the results for the query. This result is even better when we look the histograms presented in Fig. 19d and e, where most of the queries only use 5 processing or data nodes, which represent only 5 % of the total 100 DHT nodes.

In Fig. 18c, we see that the number of processing nodes remains the same when the number of index instances is increased and this is expected since there is no change in the number of nodes in the network structure and therefore the same nodes are responsible for the same ranges. The number of total messages exchanged in range queries is more than 8 times higher than the ones exchanged in exact or invalid queries - Fig. 18d -, whereas the number of messages sent to the overlay is 14 times higher - Fig. 18e. Although this may seem to be very high, if we observe the histogram presented in Fig. 19b, we see that the number of messages sent to overlay between 0 and 5 for most queries. This number depends intrinsically on the type of query and can be really very high. For instance, a query for "fif*, game*" tends to be faster than a query for "f*, g*". The processing time for range queries is also higher compared to the time for exact or invalid queries - Fig. 18f. However the same explanation given to the number of messages is valid for the query time. For almost 4000 queries the query time is around 5 sec whereas for exact or invalid queries is around 1.3 sec. We can see this in histogram of the Fig. 19e. In Fig. 19a we see that between 0 and 500 data elements were found in most queries. However, for some general range queries, such as "a*,b*", more than 9,500 data elements were found. This is also linked with the increase of data elements in Fig. 18a.

An interesting fact is the decrease of the query time as long as we have more information in GDD - Fig. 18f. This is due the fact that with a higher concentration of data in the nodes we have more chances to reach a zero next-match more quickly and therefore finishing the query. When the query is processed inside a node, the next-match algorithm passes the current data element stored in the data storage as a parameter for the algorithm. Hence, the more this algorithm is executed (due to more data elements) the higher is the probability to find a zero next-match inside the node. This avoids an additional request for a new current next-match in Tier2, when the result of the processing in the node returns to the requester node. When this processing is finished and returns, the next-match is already zero, which means that there is no more data elements in GDD satisfying the query. Then, no more queries to the overlay are necessary and the time decreases. This fact can be also linked with the smooth decrease of the other metrics such as processing nodes, number of total messages and number of messages sent to the overlay.

Although HSFC is known as a mechanism that fills in only a small part of the space, forming clusters, in our case data is well distributed over the node space. This happens because we used a simple load balancing mechanism. As our stored data elements were transformed to 64-bit keys, we used this threshold (64) to limit the identifier of the DHT nodes. In this way,

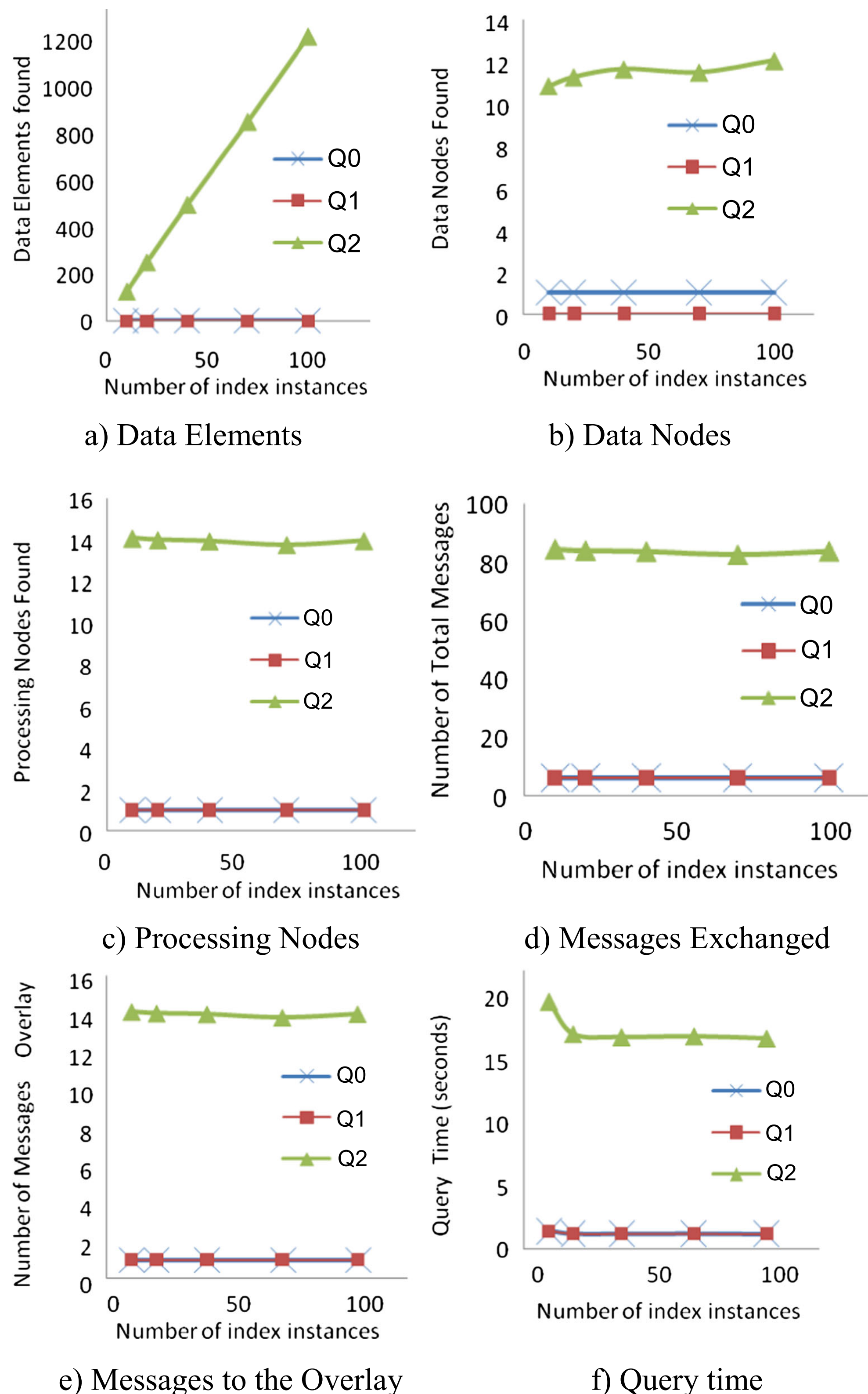


Fig. 18 Effect of varying the number of instances per index

the range of keys which nodes are responsible for is limited to 64-bit keys, which helps considerably in the natural distribution of data. If we were to apply a more elaborate balancing mechanism or take more advantage of the physical space available, we would have to measure the other metrics and observe their behavior, to conclude if it is worthwhile.

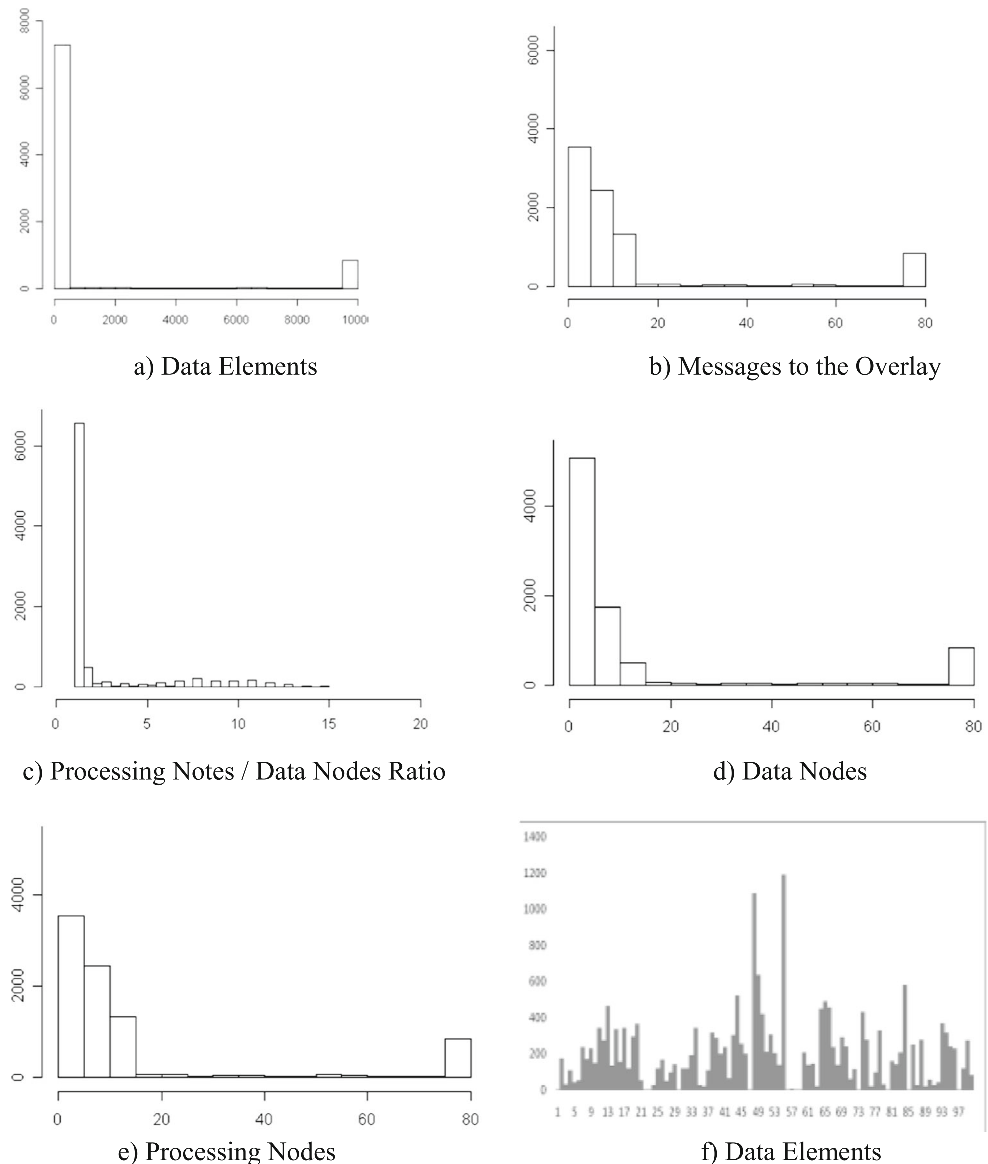


Fig. 19 Histograms for 100 instances per network index; **a)** number of data elements; **b)** number of messages in the overlay; **c)** ratio between processing and data nodes; **d)** data nodes; **e)** processing nodes; f) distribution of data elements into nodes

We also measure the relation between processing nodes and data nodes found during the queries. A perfect situation would be if the result of this division was always 1, which means that every time the algorithm inside a node is executed (i.e. this node is a processing node), data elements are found (i.e. this node is a data node). However, this fact does not always occur and sometimes we can pass through a node without finding data elements belonging to the query. GDD presented adequate results (Fig. 19c), since the ratio between processing nodes and data nodes is around 1 in most of the queries.

6.2 Effect of the number of networks

Figure 20 presents the results of the selected metrics for 100, 250, 500, 750 and 1,000 networks, 10 index instances for each network and 100 DHT nodes. In Fig. 20a we can observe that the number of data elements found per query increases as the number of networks increase. This is an expected result because the information used by GDD grows with the number of networks.

Figure 20b shows a steady increase in the number of data nodes as the number of networks is increased, which means that at a first moment the data tend to spread in the network, as observed in Fig. 18b. However, the number of data nodes for range queries (Q2) has a peak for 500 networks, whereas we can see a steep decrease over 750 and 1,000 networks. This is associated with the Hilbert Space-Filling curves concept, which tends to concentrate the data. Hence, fewer data nodes are returned per query when there is more information. In Fig. 20c, the number of processing nodes

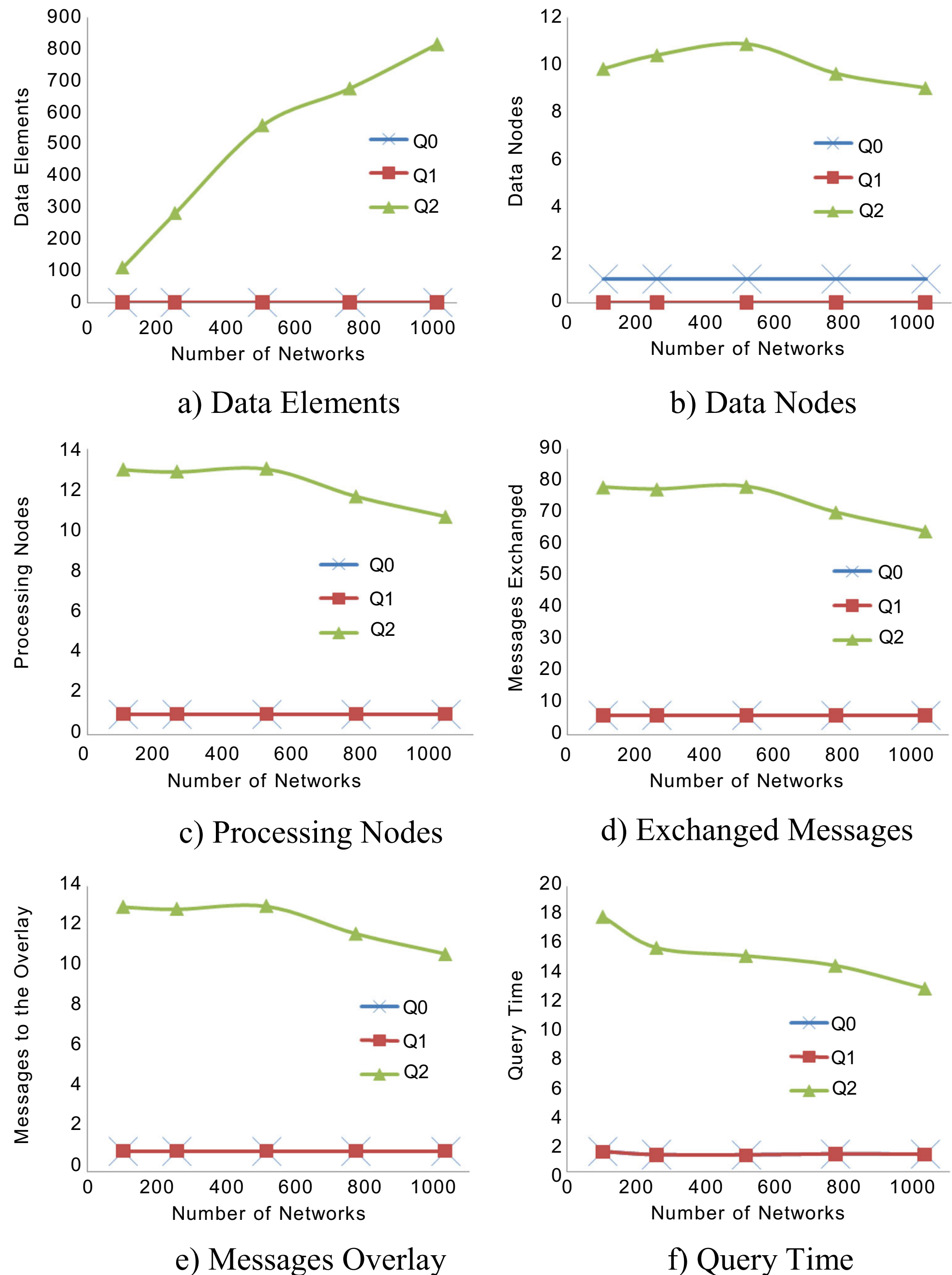


Fig. 20 Effect of varying the number of networks

remains almost constant over 100, 250 and 500 networks, but for 750 and 1,000 networks there is a steady decreasing trend. The explanation given for Fig. 18c can be also applied to the initial constant values of the number of processing nodes in Fig. 20c. However, the decrease observed for 500 and 1,000 networks follows the same trend in data nodes showed in Fig. 20b. This happens because the more concentrated the data, the faster a zero next-match is returned so that fewer nodes have to be queried and processed in order to complete a query. The other metrics in Fig. 20d, e and f also follow this trend for more than 500 networks.

7 Discussion

Design rationale adopted here recognizes the tradeoff between two different and actually opposite approaches for providing such a distributed directory. The first approach analyzed is to adapt an existing directory

service adding support for massive network, node and information distribution and dynamics. The second approach is to adapt a highly distributed environment that supports dynamicity but only exact-key searches, and add a flexible query engine layer. In this paper, we chose the second approach and therefore GDD is based on DHT with extended support for flexible queries provided by our directory indexing and searching mechanisms.

Our new HSFC-based DHT mechanism do not use a hash function to generate the keys, since it spreads the content over the existing nodes. Rather, we use an indexing mechanism to generate the derived keys for each index instance, which are directly used to insert information into the DHT. In addition, a library that composes our new HSFC-based DHT mechanism was developed, containing the indexing, macro search and next-match discovery algorithms used to guarantee key insertion and search, which extend typical DHT mechanisms. The results showed that the use of DHT for allowing flexible queries together with HSFC for flexible queries yields an important synergy.

We also built a new simulator on top of the OMNeT++ framework, implementing the main features of GDD. Performance analysis revealed that GDD fulfills different requirements of distribution and scalability, typical for some present and probably future networks. The scalability was evaluated with the number of networks and information. Results show that the number of data and processing nodes, the number of messages and the query response time do not present significant variations as we increase the number of data elements and networks. Particularly, results show that query time and number of messages are not influenced by the number of indexes and participating networks. Our approach keeps the complexity of the underlying DHT structure, which can be observed by the query time and number of messages for range queries (Q2) that maintain a consistent linear relationship with exact queries (Q0). Also, unlike other proposals for flexible queries, our approach is able to find all answers matching a query in a given DHT substrate.

As for the requirements for a global directory service presented in Section 3.1 we think they have been fulfilled by GDD and its implementation and validated through the simulation study. Also, the use of DHT for distribution and dynamicity together with HSFC for flexible queries yields an important synergy. DHT is robust to node churn and highly tolerant to node failure. HSFC and the performance of the new mechanism are significantly invariant to the growth of the network size.

A different, and simpler, way to implement range queries in a DHT might be by generating multiple indexes, each one with a different possible search option. However, this approach suffers from a serious drawback, since there will be a tradeoff between the number of possible search options and the maximum number of indexes needed for storing a data item and the maximum number of queries needed for retrieving the information. Therefore, the usefulness of such a strategy might be low if we store just a few different options for each data item and the scalability may be an issue if we store many options.

8 Conclusion

Distributed directory services are not novel, but still today there is no definitive solution for information management in highly mobile, heterogeneous, dynamic, and collaborative networks. In this paper we proposed the Global Distributed Directory (GDD) to deal with such important challenge.

The design rationale of GDD follows the approach of adding features for providing flexible queries to a highly distributed and scalable environment, by using Hilbert curves in a peer-to-peer DHT system. This required the key generation process of the DHT to be changed for a mechanism based on Hilbert curves. Simulation results we obtained from a customized simulator based on the OMNeT++ framework. Results show that the new mechanisms are scalable with the number of networks, DHT nodes, and amount of information. Particularly, query time and number of messages exchanged are orthogonal to the number of indexes and participating networks.

As future work, we intend to extend the GDD architecture and its evaluation. We aim at proposing and evaluating load balancing mechanisms and indexes that go beyond the limit of three-dimensions, as well as security mechanisms to control the access to the information. In addition, we aim at making the GDD mechanism as generic as possible in terms of receiving an information model as parameter and automatically generating the corresponding indexes.

References

1. Andreolini M, Lancellotti R (2009) A flexible and robust lookup algorithm for P2P systems, In: IEEE international symposium on parallel. Distributed Processing, IPDPS 2009, Rome, pp 1–8

2. Baumgart, I., Heep, B., Krause, S., "OverSim: A Flexible Overlay Network Simulation Framework", 10th IEEE Global Internet Symposium (GI '07), May 2007.
3. Curbera F et al (2002) Unraveling the Web services Web: an introduction to SOAP, WSDL, and UDDI. Internet Computing 6(2):86–93
4. DMTF (2014) Common Information Model (CIM) Specification, Version 2.40, Distributed Management Task Force, Available at http://dmtf.org/standards/cim/cim_schema_v2400.
5. Dustdar S, Treiber M (2005) A view based analysis on Web service registries. Distributed and Parallel Databases 18(2):147–171
6. IEEE Std 802.21 (2009) IEEE Standard for Local and Metropolitan Area Networks - Part 21: Media Independent Handover Services. IEEE.
7. ITU-T (2005) Recommendation X.500 - The Directory: Overview of concepts, models and services. v. 08/2005.
8. Joung, Y.-J., Yang, L.-W. (2014), On character-based index schemes for complex wildcard search in peer-to-peer networks, Inform. Sci., http://dx.doi.org/10.1016/j.ins.2014.02.095.
9. Kamienski C, Sadok D (2004) The case for inter domain dynamic QoS-based service negotiation in the internet. Comput Commun 27(7):622–637
10. Klampanosa IA, Jose JM (2012) Searching in peer-to-peer networks. Comput Sci Rev 6(4):161–183
11. Lawder, J. K., King, P. J. H. (2000) Using Space-Filling Curves for Multi-dimensional Indexing. In: 17th British National Conference on Databases, BNCOD 2000, LNCS 1832:20–35 17.
12. Moon B, Jagadish H, Faloutsos C, Saultz J (1996) Analysis of the clustering properties of the Hilbert space-filling curve. IEEE Trans Knowl Data Eng 13(1):124–141
13. OASIS (2010). Universal Description Discovery and Integration. OASIS UDDI Version 3.0.2, July 2010.
14. Park, V., Corson, M. (1997) A highly adaptive distributed routing algorithm for mobile wireless networks. 17th IEEE International Conference on Computer Communications. INFOCOM 97. 7–12 April 1997, Kobe, Japan, pp 1405–1413.
15. Pentikousis K, Galis A, Agüero R (2009) Information management and sharing for ambient Multiaccess networks. Global Information Infrastructure Symposium, GIIS 2009, Hammamet, pp 23–26
16. Pitoura T, Ntarmos N, Triantafillou P (2012) Saturn: range queries, load balancing and fault tolerance in DHT data systems. IEEE Trans Knowl Data Eng 24(7):1313–1327
17. Ratti, S., Hariri, B., Shirmohammadi, S. (2008) NL-DHT: A Non-uniform Locality Sensitive DHT Architecture for Massively Multi-user Virtual Environment Applications. In: 14th IEEE International Conference on Parallel and Distributed Systems. ICPADS 2008. Victoria, Australia. 8–10 December 2008.
18. Rowstron, A., Druschel, P. (2001) Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, In: IFIP/ACM International Conference on Distributed Systems Platforms, Middleware '01, Heidelberg, Germany, 12–16 November 2001, pp 329–350.
19. Schmidt CE, Parashar M (2004) A peer-to-peer approach to Web service discovery. World Wide Web 7(2):211–229
20. Schmidt C, Parashar M (2008) Squid: enabling search in DHT-based systems. J Parallel Distrib Comput 68:962–975
21. Sermersheim, J. (2006.) Lightweight Directory Access Protocol (LDAP): The Protocol, RFC 4511.
22. Stoica I, Morris R, Karger D, Kaashoek M, Balakrishnan H (2001) "Chord: a scalable Peer-to-Peer, lookup protocol for internet applications. ACM SIGCOMM 2001, San Diego, pp 149–160
23. Taniuchi K (2009) IEEE 802.21: media independent handover: features, applicability, and realization. IEEE Commun Mag 47(1):112–120
24. Varga, A., Hornig, R. (2008) An overview of the OMNeT++ simulation environment. In: 1st international conference on Simulation tools and techniques for communications, Simutools '08, Marseille, France, 3–7 March 2008, art 60.
25. Villaca, R., de Paula, R., Pasquini, R., Magalhaes, M. (2013) Hamming DHT: Taming the Similarity Search, In: IEEE Consumer Communications and Networking Conference. CCNC 2013. Las Vegas, USA, 11–14 January 2013, pp 7–12.
26. Vu, Q.H., Lupu, M., Wu, S. (2009) SiMPSON: Efficient Similarity Search in Metric Spaces over P2P Structured Overlay Networks. In: 15th International Euro-Par Conference on Parallel Computing, Delft, The Netherlands, 25–28 August 2009, pp 498–510.
27. Wei, X., Sezaki, K. (2006) DHR-Trees: a distributed multidimensional indexing structure for P2P systems. In: International Symposium on Parallel and Distributed Computing, ISPDC 2006, Timisoara, Romania, 6–9 July 2006, pp 281–290.
28. Zhang C, Xiao W, Tang D, Tang J (2011) P2P-based multidimensional indexing methods: a survey. J Syst Softw 84(12):2348–2362
29. Zhua Y, Hub Y (2007) Efficient semantic search on DHT overlays. J Parallel Distrib Comput 67(5):604–616

Tarciana Silva received her Master Degree in computer science from the Federal University of Pernambuco (UFPE, Brazil) in 2008. She is currently a PhD Student at UFPE and an associate professor at the University of Pernambuco, Recife, PE, Brazil. Her current research interests include peer-to-peer networks and heterogeneous networks.

Carlos Kamienski received his Ph. D. in computer science from the Federal University of Pernambuco (Recife PE, Brazil) in 2003. He is currently an associate professor of computer networks at the Federal University of ABC in Santo André SP, Brazil. His current research interests include peer-to-peer networks, traffic measurement and analysis, service-oriented computing, cloud computing and future Internet.

Stênio F. L. Fernandes received his Ph. D. in Computer Science from the Federal University of Pernambuco (UFPE), in 2006, where he recently joined as Adjunct Professor. He was a Research Fellow at the School of Information Technology and Engineering at the University of Ottawa, Canada (2004/2005, 2008/2010). His current research interests include traffic measurement, modeling and analysis, mobility management in heterogeneous networks, and network virtualization.

Djamel Sadok received his Ph. D. degree from Kent University in 1990. From 1990 to 1992, he was a Research Fellow in the Computer Science Department, University College London. He is currently a Professor of computer networks at the Computer Science Department of the Federal University of Pernambuco, Recife PE, Brazil. His current research interests include traffic engineering of IP networks, wireless communications, broadband access, and network management. Dr. Sadok is a senior member of the IEEE Communications Society.