CS654 Project Report

Critical Examination of Service Discovery Techniques in Peer-to-Peer Systems

N.M. Mosharaf Kabir Chowdhury

(20262829)

nmmkchow@cs.uwaterloo.ca

July 30, 2007

1 Introduction

Web services are the new paradigm of distributed computing because of their promise toward interoperability of applications and integration of large scale distributed systems. On the other hand, with the ever increasing number of computers all around the globe, peer-to-peer(P2P) networks have gained large popularity capitalizing on their open, distributed, inter-operable, cost-effective and resource-friendly nature. As a consequence, the problem of finding a way to effectively combine web services with P2P technologies has gained much attention because of its potential to be an effective solution to business integration problems. Hence, different large-scale service infrastructures utilizing P2P systems have been proposed by many research groups. Service discovery, an integral part of any large-scale service infrastructure, ensures that all the services provided by different providers within the infrastructure are collected, organized and published in a way that the clients can efficiently search and find their desired services.

Due to the transitory nature of peer population and ever-changing information content in these peers, users almost always do not have the exact information about the available services/resources in the system. As a result, queries are based on partial information requiring a flexible search mechanism which, by the way, also have to be complete, scalable and efficient. Existing search mechanisms in P2P networks, in general, do not provide a satisfactory solution for achieving the conflicting goals of flexibility and efficiency. Unstructured search techniques sacrifice efficiency for flexibility, whereas structured ones (mostly

```
service-name=service:print
scope-list=staff,student
location=DC1234
color=true
double-sided=true
language=PS
paper-size=letter,a4
```

(a) Service Advertisement

```
service-name=service:print
scope-list=student
paper-size=letter
```

(b) Query

Figure 1: Example advertisement and query in Service Discovery Systems

Distributed Hash Table (DHT)-based) can route the queries efficiently but support exact-matching only. As all of the service discovery systems in P2P networks are, in one way or another, based on these search mechanisms they also suffer from same problems. A lot of efforts have been given in recent years by many research groups all over the world to minimize the conflict between the goals of flexibility and efficiency and to reach a viable solution; Secure Service Discovery System (SSDS) [7, 8], INS/Twine [6], Squid [15, 16], PWSD [11] to name a few.

The goal of this work is to review some of the most significant of these already existing service discovery solutions along with their underlying search technique Chord [17]. A new search technique in P2P networks called Distributed Pattern Matching System (DPMS) [2], that supports partial matching, is also explained and some ideas to build a service discovery system based on DPMS are proposed. We believe that the inherent capability of DPMS to partially match patterns can also be extended to service discovery problem.

1.1 Problem Definition and Motivation

The motivation behind solving partial matching problem for service discovery systems can be explained by formally defining the structure of advertisements and query patterns in these systems. An advertisement of a particular service, $\mathcal{A}_i$ can be expressed as a set of n_i attribute-value pairs, $\mathcal{A}_i = \{A_{i,1}, A_{i,2} \ldots A_{i,n_i}\}$, where each $A_{i,j} = (a_{i,j}, v_{i,j})$ and $1 \leq j \leq n_i$. If there are N advertisements from different providers in the system, the set of advertisements, $\mathbb{A}$ can be expressed as, $\mathbb{A} = \bigcup_{i=1}^{N} \mathcal{A}_i$. On the other hand, a query, $\mathcal{Q}$ is set of m attribute-value pairs, $\mathcal{Q} = \{Q_1, Q_2 \ldots Q_m\}$, where each $Q_k = (a_k, v_k)$ and $1 \leq k \leq m$.

So the problem of finding a service given a query $\mathcal{Q}$ is essentially a subset-matching problem, where the goal is to find all the services $\mathcal{A}_i$ such that $\mathcal{Q} \subseteq \mathcal{A}_i$ and $1 \leq i \leq N$. Figure 1 shows examples of an advertisement and a query that will match to the advertisement.

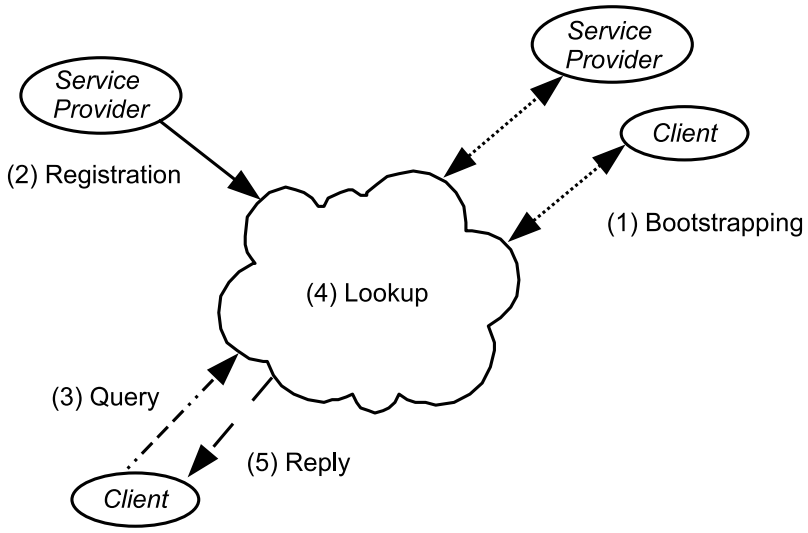


Figure 2: Generic architecture and steps of service discovery

The existing techniques (SSDS, INS/Twine, Squid etc.) require multiple DHT-lookup operations for solving a single query because standard operation is based on exact-match semantics. So, they create separate keys for each of the attribute-value pairs, $A_{i,j}$ in an advertisement $\mathcal{A}_i$ and while searching for query, $\mathcal{Q}$ they perform exact-matched DHT-lookup operation m times for m attribute-value pairs in $\mathcal{Q}$. This process is extremely inefficient. For the example query shown in Figure 1(b), a DHT-based solution will make three queries to find the advertisements that have `service-name=service:print`, `scope-list=student` and `paper-size=letter`. Then it will intersect the separate results from each query to find out the result, if any.

On the other hand, Distributed Pattern Matching System (DPMS) inherently supports partial matching and it is experimentally proved in [2] that it outperforms DHT-based lookup protocols such as Chord in the context of content sharing networks. We believe that same idea can also be applied here.

1.2 Generic Architecture

Before examining different systems in detail, an overview of a service discovery system is necessary. Service discovery systems generally consist of three types of entities - *servers*, *clients* and *middle agents* [9]. This *'middle agent'* can be *matchmakers*, *directory* service, *blackboard* agents or *brokers*. This process of discovery has following general steps (Figure 2) [5]: 1) *Bootstrapping*, where clients and service providers first attach themselves to the system, 2) *Service advertisement*, where service provider publishes advertisement(collection of attribute-value pairs), 3) *Querying*, where a client looks for a desired service (usually a

partial service description), 4) *Lookup*, where searching for the given query happens and 5) *Service handle retrieval*, the final step, where a client retrieves the means to access that service.

Directory architecture adopted by different service discovery approaches can broadly be classified as *centralized* and *decentralized* [5] depending on how the directory/registry is stored and managed. Centralized UDDI is an example of such architecture. The problem with this approach is that the centralized server becomes a performance bottleneck and forms a single point of failure. On the hand, there are several decentralized variants like *replicated* (e.g. INS), *distributed or partitioned* (e.g. SLP, Jini, UPnP) and *hybrid* (e.g. TWINE) [2]. The obvious advantage is the removal of the bottleneck.

1.3 Paper Organization

Section 2 discusses two lookup protocols for P2P systems, namely Chord and DPMS, which perform the lookup operation as mentioned in the fourth step of the generic architecture. Section 3 is concerned with existing service discovery systems. In this section, three popular systems, namely SSDS, INS/Twine and Squid, are examined and the way they handle multi-criteria search is discussed. Section 4 gives some ideas regarding a DPMS-based service discovery system that might be able to alleviate the problems of the existing systems as explained in Section 3.

2 Peer-to-peer Lookup Protocols

Most P2P service discovery systems are based on one or another existing lookup protocols such as Chord [17], CAN [12], Pastry [14] etc, which can broadly be categorized as structured DHT-based lookup protocols. There are other lookup protocols which are unstructured or semi-structured. Flooding is an example of an unstructured lookup protocol and DPMS is an example of a semi-structured one. Here we elaborate on Chord, the most widely used lookup protocol in P2P service discovery systems. We also discuss the basic architecture of DPMS which will be used as the underlying lookup protocol for our proposed system.

2.1 Chord

Chord [17] is a DHT-based distributed lookup protocol that provides only one operation: given a key, it maps the key onto a node. To achieve this functionality it uses consistent hashing to assign keys to Chord nodes. Because of its simplicity, there are a lot of P2P applications that are built on top of Chord. One can easily use it to store and lookup data

by assigning a key with each data item and storing the key/data pair at the node to which the key maps. Chord adapts well as nodes join and leave the system and it is theoretically and experimentally known to be scalable. And its communication cost, measured in number of hops to find a node given a key, is logarithmical with the number of Chord nodes.

Features Chord makes it easier to build P2P applications on top of it by providing some excellent features like, (details related to these features can be found in [17])

- Load balance
- Decentralization
- Scalability
- Availability
- Flexible Naming

2.1.1 Structure and Protocol

Chord uses fast distributed hashing to map keys to nodes but it improves the scalability of *consistent hashing* by removing the constraint that every node needs to know about all other nodes in the system. Instead, each node knows about $O(\log N)$ other nodes, and lookup requires $O(\log N)$ messages only, where N is the total number of nodes in the system. This also reduces the amount of 'routing' information saved in each node.

Chord Ring The consistent hash function assigns each node and key with an m-bit *identifier* using SHA-1 [10]. The identifier length must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible. Identifiers are ordered on an *identifier circle* modulo 2^m. Any key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space. This node is called the *successor node* of key k and the identifier circle is known as the *Chord ring*. Figure 3 shows a Chord ring (m=5) which has ten nodes and stores four keys. The successor of K7 is N8 so it resides in there. Similarly, the successor of both K21 and K29 is N31. So both of them are stored in N31 and K53 is located at N56 due to similar reasoning.

2.1.2 Lookup Operation

In its simplest form lookup on a Chord ring can be implemented with little per node memory. It starts from a node and visits all the nodes clockwise until the desired key k is found and

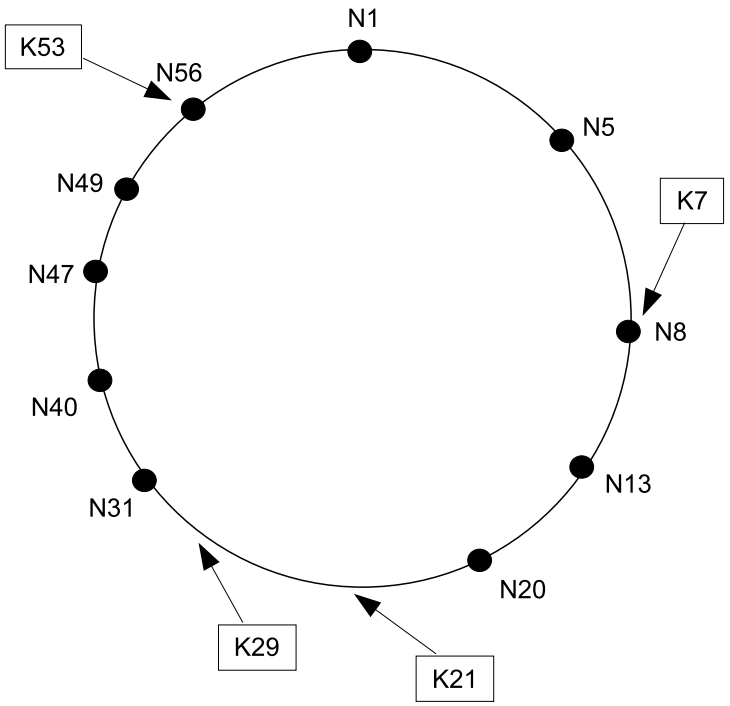


Figure 3: Chord ring consisting of ten nodes storing four keys

then returns back to the calling node with result. For example, in Figure 3 if a *lookup*(53) call is executed in N5, messages will pass through all the nodes up to N56, where K53 is located, and then return back with the result following the reverse path. As a result, number of messages exchanged is $O(N)$ in the worst case scenario.

To reduce the number of messages, Chord implements an $O(\log N)$ solution at the expense of some memory in each of the participating nodes. In this solution each node stores a *finger table* containing m entries, where ith entry in the table at node n contains the identity of the *first* node s that succeeds n by at least $2^{(i-1)}$ on the Chord ring, i.e., $s = successor(n + 2^{i-1})$, where $1 \leq i \leq m$. This node s is called the ith *finger* of node n. The example in Figure 4 shows the fingers of node N5. The first finger of N5 points to N8 as it is the first available node in the ring succeeding $(5 + 2^0) \bmod 2^5 = 6$.

Now the lookup operation speeds up using the fingers stored in each node. If the same *lookup*(53) operation is executed now, from N5 a message will be passed to N31 as it is the maximum valued identifier on the ring accessible from N5 which is smaller than N56 that holds the key K53. N31 will then use its own finger table to reach N49, which will eventually find N56 to complete the query. It is proved in [17] that number of hops required to perform a lookup operation is $O(\log N)$.

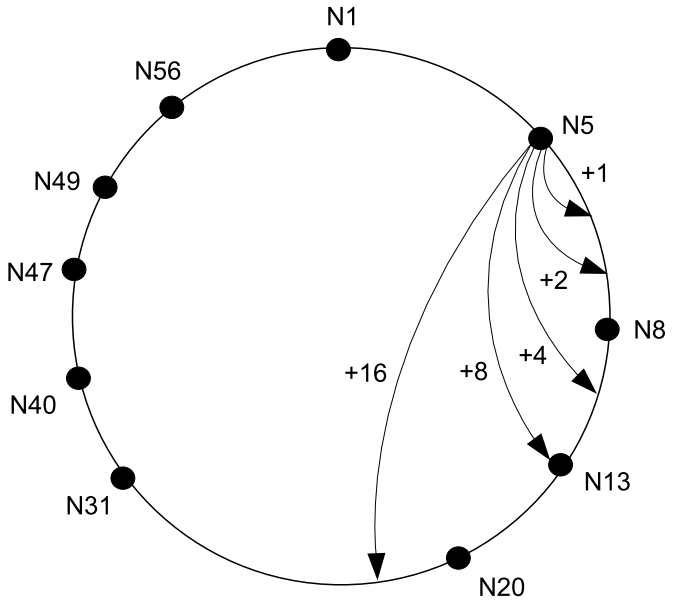


Figure 4: Fingers of node N5

2.1.3 Node Joins and Failures

At every point Chord needs to make sure that a node's successor pointers and the entries in its finger table are up to date. Inconsistencies can occur whenever a new node joins into the system or some one leaves or fails. To manage this problem Chord runs a *stabilization* protocol in the background for every join operation. Every time a node joins, this *stabilize* protocol updates the successors as well as the fingers using the information from the nodes preceding and succeeding the new node on the ring. Even though there might be a certain amount of time when all the fingers are not updated, it is guaranteed that the system will successfully perform lookup operations using the successor list albeit a little slowly [17].

The correctness of the Chord protocol basically relies on the fact that each node knows its successor. But it can be compromised if a successor fails or leaves. To prevent this each node keeps track of r successors instead of just one. As a result, the system will perform perfectly fine until all r successors fails, which is highly unlikely. Assuming each node fails independently with probability r, the probability that all the nodes in the successor list will simultaneously fail is only p^r. It can also be shown that $r = \Omega(N)$, which is pretty small in practice i.e. memory requirement per node does not increase that much due to this extra book-keeping.

2.2 DPMS

Distributed Pattern Matching System or DPMS [2, 3, 4] is a semi-structured search technique for P2P systems that enables flexible and efficient search with partial matching. DPMS uses a hierarchy of indexing peers for disseminating advertised patterns. Patterns are replicated and aggregated using Bloom filters at each level along the hierarchy to ensure robustness and fault-tolerance while keeping the storage requirement in each node as low as possible. An advertised pattern can be discovered using any subset of its 1-bits, which allows inexact matching and queries in conjunctive normal form. The search complexity of DPMS is $O(\log N + \xi \log \frac{N}{\log N})$, where N is the total number of nodes/peers in the system and ξ is proportional to the number of matches for a given query.

Features Features that make DPMS a better alternative in certain scenarios than Chord or other structured DHT-based P2P systems include,

- Partial matching capability
- Search completeness
- Load balance
- Decentralization
- Scalability

2.2.1 Architectural Overview

In DPMS a peer can act as a leaf peer or an indexing peer. A leaf peer is at the bottom layer of the indexing hierarchy and advertises its indices (created from the objects it is willing to share) to other peers in the system. An indexing peer, on the other hand, stores indices from other peers (leaf peers or indexing peers) that are located in lower level of the hierarchy.

Hierarchical indexing technique is also very common in many unstructured P2P systems. But DPMS improvises it by using replication at each level at the cost of some extra storage. DPMS uses replication trees (Figure 5(a)) for disseminating patterns from a leaf peer to a large number of indexing peers (darker edges in the figure shows the path of propagation). But this can generate large volume of advertisement traffic. To overcome this, DPMS combines replication with lossy-aggregation. As shown in Figure 5(b), advertisements from different peers are aggregated and propagated to peers in the next level along the aggregation tree. The amount of replication and aggregation is controlled by two system-wide parameters, namely replication factor R and branching factor B. Patterns advertised by a

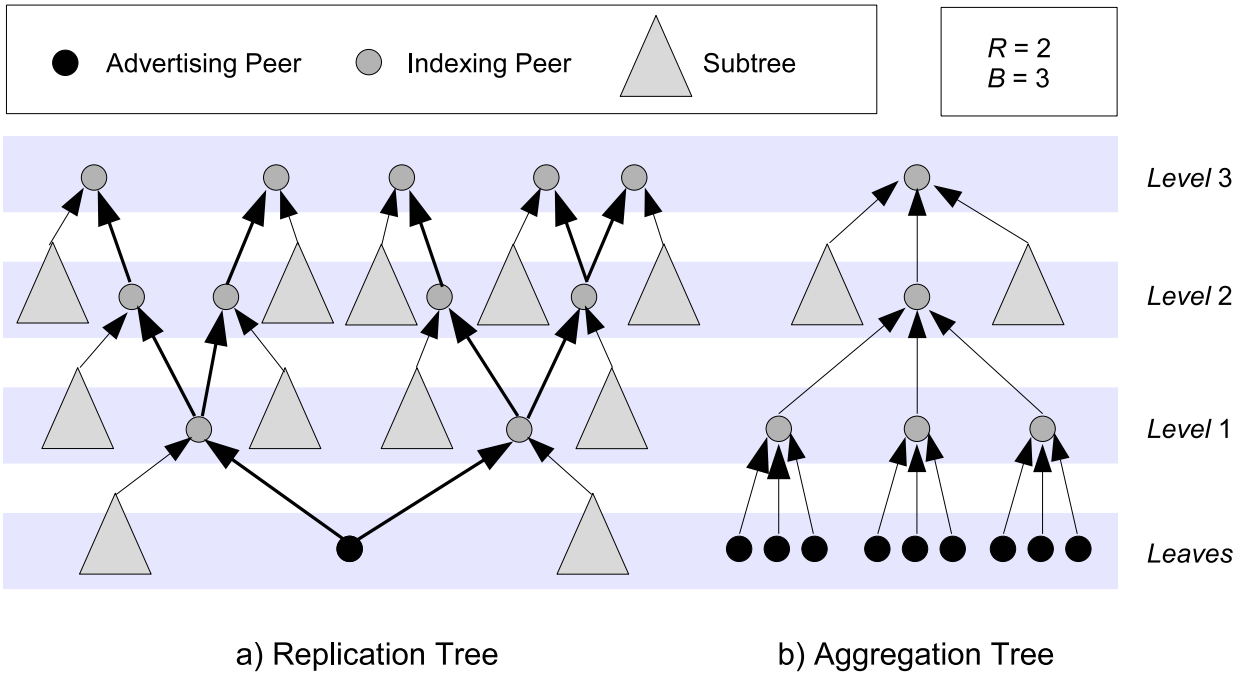


Figure 5: Formation of Replication and Aggregation trees in DPMS

leaf peer are propagated to R^l indexing peers at level l. On the other hand, an indexing peer at level l contains patterns from B^l leaf peers. Hence DPMS actually forms a lattice-like structure. The optimum values for R and B are experimentally derived by trial and error.

Index/Pattern Construction Unlike DHT-based solutions where hash functions are used to obtain an index for a node, DPMS uses Bloom filters for the same purpose. To provide inexact matching it fragments all the keywords into η-grams (usually tri-grams). It then passes all these η-grams through a Bloom filter to get a bit-pattern for the advertisement. This procedure is followed for all the advertising peers and then DPMS hierarchy is created, as described earlier, using repeated replication and aggregation. For example, if any peer want to advertise '$lord\ of\ the\ rings$', it will create a bit pattern by passing the set of tri-grams $P_1 = \{lor, ord, the, rin, ing, ngs\}$ through a Bloom filter. Similarly '$lord\ of\ war$' will pass $P_2 = \{lor, ord, war\}$ through the Bloom filter.

Query keywords are also fragmented into η-grams and encoded into a Bloom Filter. The 1-bits on a query should be a subset of any pattern that it should match against. This kind of encoding allows DPMS to retrieve information based on partial matching. For example, if someone has a query '$*lord*$', the resulting tri-grams' set would be $Q = \{lor, ord\}$ and it will match with both P_1 and P_2 as $Q \subset P_1$ and $Q \subset P_2$, respectively.

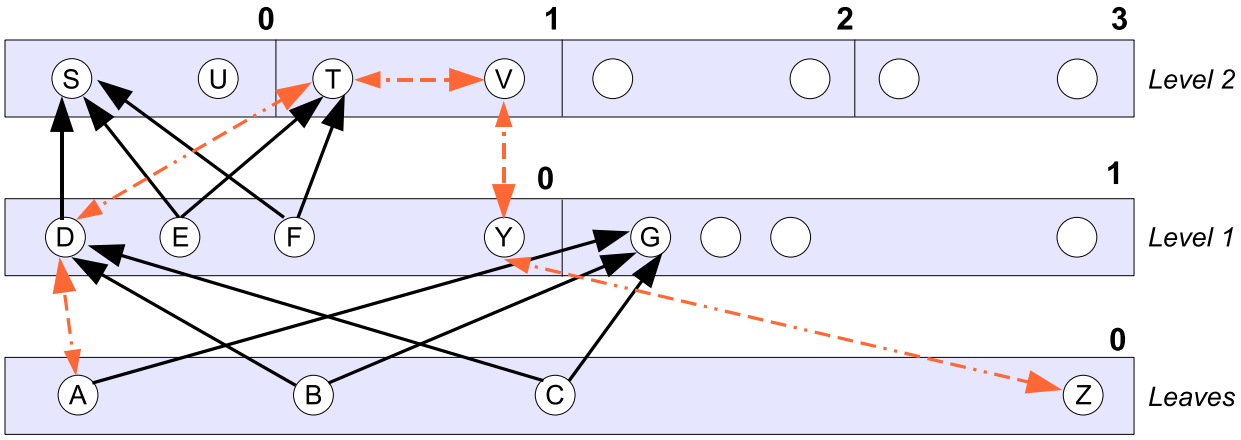


Figure 6: Index Hierarchy and Query Routing in DPMS

Aggregation Scheme As DPMS provides pluggable interface for aggregation, choice of an appropriate aggregation scheme is very important to reduce the large volume of traffic it creates by using replication at each level. A trivial way of pattern aggregation is bitwise OR function, which is already used by several other systems. But due to its high loss of information, DPMS proposes a *don't care* based aggregation scheme where each bit of the aggregated pattern is replaced by a don't care, X whenever corresponding bits of the two participating patterns mismatch. It ensures maximum matching but doubles the size of each pattern as there are three states for each bit now. Another observation is this aggregation scheme may result in more matches than there actually are as X will match with anything whether be it 0, 1 or X. Details of this aggregation process can be found in [2, Section 3.6].

Index Distribution and Topology Maintenance Index peers are distributed along the levels of DPMS hierarchy. Each of them belong to two sets, vertical(i.e. level) and horizontal(i.e. replicated groups). For example, peer T in Figure 6 belongs to level 2 and group 1.

Peers interact with each other in different roles. So each indexing peer keeps **Replica-List**, **Parent-List**, **Child-List**, **Sibling-List** and **Neighbor-List**. In Figure 6, peer D has replica G, parents $\{S, T\}$, children $\{A, B, C\}$, siblings $\{E, F\}$ and neighbor Y.

2.2.2 Lookup Operation

The query life-cycle can be divided into three phases: *ascending phase*, *blind search phase* and *descending phase*. In the *ascending phase*, an initiating peer, I searches in its local information and if there is no match it sends the query to any of its parents. This continues until a peer E at the top-most level is reached, i.e. E has no parents. At this point,

blind search phase starts as E has no option but to blindly forward the message to some other peer(its neighbors and siblings) in its group. As the query traverses a peer at level l, aggregates from R^l leaf peers are being checked. If any match is found the query is forwarded to associated child, otherwise if no peer in the topmost level can find a match the query results in a failure.

A query enters into the *descending phase* when it hits a peer that has some aggregate matching the query. The query is then forwarded to the child peer advertising the matching aggregate. This process recurs until the query reaches the leaf peer advertising the pattern. Then same path is followed in reverse to send back the result.

In Figure 6, a query is initiated by peer A and the result of the query is in node Z. The path of the query resolution is shown by dashed lines. The path $A \rightarrow D \rightarrow T$ forms the ascending phase, whereas $T \rightarrow V$ and $V \rightarrow Y \rightarrow Z$ are blind search phase and descending phase respectively.

2.2.3 Node Joins and Failures

A peer can join the system as a leaf peer or an indexing peer or both. To join as a leaf, a peer say C, has to find a level 1 indexing peer, say P, with an empty slot in its child-list. C joins the indexing hierarchy as a child of P. On the other hand, to join as an indexing peer, C has to choose a level and group (described earlier) with empty space and then construct required lists e.g. parent-list, child-list etc. This is a non-trivial process and details of this algorithm is explained in [2, Section 3.10]

Peer departures and failures are handled in almost similar way. As there are multiple replicas, a query will almost always find its way. The probability that all the replicas of a peer will fail is very low. One problem is that whenever a peer leaves or fails, there is a cascading effect on all its parents and children, as all of them were keeping track of it. Everyone has to update there lists and there might be a certain latency until the system reaches a stable state.

3 Existing Service Discovery Systems

3.1 Secure Service Discovery Service (SSDS)

Secure SDS or SSDS [7, 8] is an authenticated Java RMI based scalable, fault-tolerant, and secure information repository providing clients with directory-style access to all available services. It supports both push-based and pull-based access; the former allows passive discovery, while the latter permits the use of a query-based model. Service descriptions and

queries are specified in XML, which gives flexibility in expressing advertisements and queries. On the other hand, it enforces authentication to make sure that no one misuses its open environment. The SDS architecture handles network partitions and component failures to make it robust against typical problems in P2P networks. One of the achievements of SSDS is that it supports a form of multi-criteria search using a novel query filtering approach. A brief summary of its architecture and operation is presented below.

3.1.1 Design Concepts

The SDS system is composed of three main components: clients, services, and SSDS servers. Clients want to discover the services that are running in the network. SDS servers enable this by gathering information from the services and then using it to fulfill client queries. To enable this SSDS uses the following concepts,

Announcement-based Information Dissemination SSDS uses *periodic multicast announcements* as its primary information propagation technique, and uses information caching to manage the states of the system. The caches are updated by the periodic announcements or purged based on the lack of them. This mechanism allows SSDS to make sure that component failures are tolerated in normal mode of operation rather than using a special recovery mechanism.

XML Service Description Instead of using flat name-value pairs, SSDS uses XML to describe both service descriptions and client queries. XML allows the encoding of arbitrary structures of hierarchical named values and enables subtyping via nested tags.

Privacy and Authentication To save itself from malicious attacks SSDS uses encryption and authentication. Associated with every component in the SSDS system is a principal name and public-key certificate that can be used to prove the component's identity to all other components in the system.

Hierarchical Organization As a scalability mechanism, SSDS servers organize into multiple shared hierarchies. From an architectural point of view, SSDS is the closest match to DPMS. Like DPMS, SSDS uses Bloom filters and aggregation, and index distribution is achieved through a hierarchy of indexing nodes. In contrast to the lattice-like hierarchy used by DPMS, SSDS uses a tree-like hierarchy for index distribution. Also it does not use replication and uses bitwise OR as its aggregation function. Figure 7 shows the main components of SDS hierarchy and interactions between them. Apart from clients, SDS servers

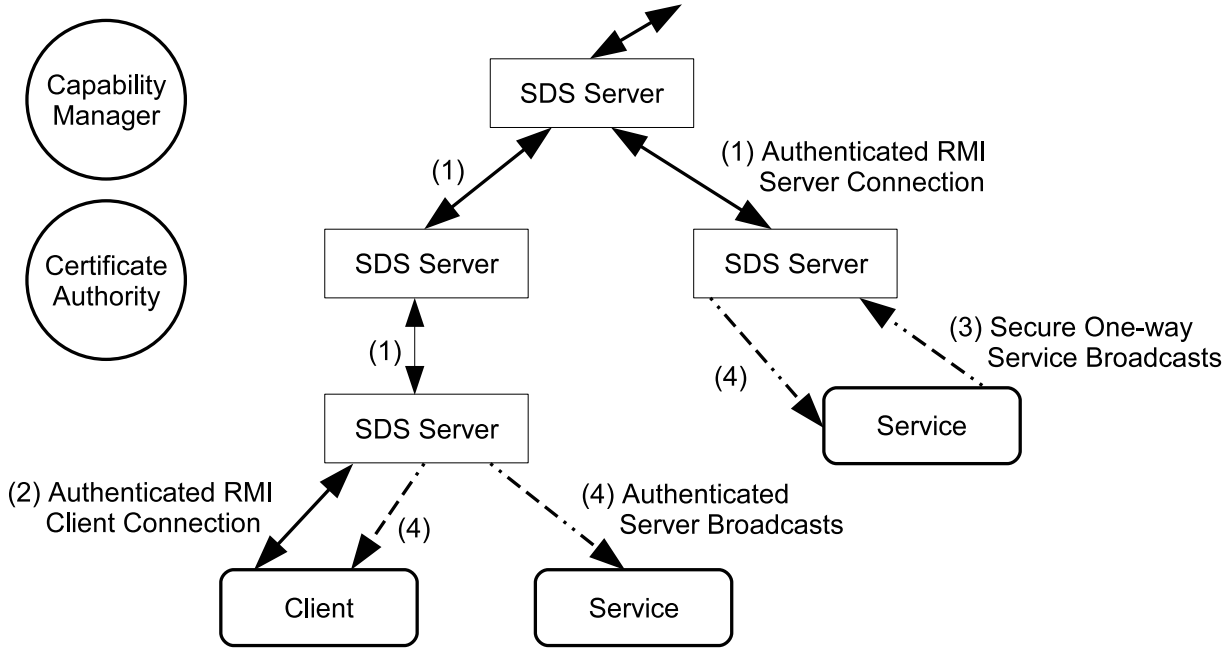


Figure 7: SDS Architecture

and services there are two more components, namely Capability Manager and Certificate Authority, who take part in routing and authentication procedures of SSDS respectively. Detailed explanation can be found in [8, Section 3].

3.1.2 Multi-Criteria Search

SSDS hybridizes flooding, mapping and centralization to approach *multi-criteria search* problem. It uses two key ideas. First, it uses push model along with pull based protocols to *update* the system proactively. Second, instead of proactively filling nodes with cached query responses from the information in updates, it instead propagates *summaries* of node contents, which are used as *filters* that are applied to queries.

To implement query filtering it uses dynamic construction and adaptation of the neighbor relationships between SDS servers. It also decomposes the information propagation problem into two sub-problems: providing lossy aggregation (it uses bitwise OR) of service descriptions as they travel farther from their source and dynamically flooding client queries through the filters to the appropriate servers based on the local aggregate data (a query routing mechanism). In order to support multi-criteria search it cannot use DHT lookup so it uses Bloom filters as in DPMS.

Query Routing and Lookup Operation Upon receipt of a service announcement, an SDS server S1 applies multiple hash functions to various subsets of tags in the service description and uses the results to set bits in a bit vector. The resulting bit vector (i.e. the filter) summarizes its collection of descriptions. This filter is given to neighbor S2. When S2 receives a query that it cannot resolve locally, it checks to see if the query should be forwarded to S1 by similarly multiply hashing it and checking that all the matching bits are set in S1s filter. If any bit is not set, then the service is definitely not there. If all bits are set, then either it is a hit, or a *false positive* has occurred, which is a common problem of using Bloom filters. The latter forces unneeded additional forwarding of the query, but does not sacrifice correctness.

If SDS Server has children it aggregates the bit vectors from all of its children with its own vector using bitwise OR function and passes the resultant vector to its parent, who then associates it with corresponding branch of the tree.

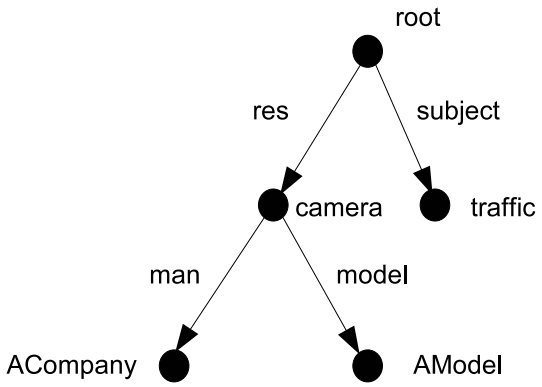
Whenever a query arrives, it can be coming in either upward direction or downward direction. If a query is coming up the hierarchy, the receiving SDS server checks to see if it hits locally or in any of its children; if not, it passes it upward. On the other hand, If a query is coming down the hierarchy, it is checked locally and against the childrens tables. If there is a hit locally then it is resolved locally. If there is a hit in any of the childrens tables, the query is routed down to the matching children. If neither of these occur, it is known as a miss.

3.2 INS/Twine

INS/Twine [6] is an approach to scalable intentional resource discovery, where resolvers collaborate as peers to distribute resource information and to resolve queries. It uses Intentional Naming System (INS) [1] as its backbone and relies on Chord for routing. It maps resources to resolvers by transforming descriptions into numeric keys in a manner that preserves their expressiveness, facilitates data distribution and enables efficient query resolution. Twine achieves scalability via hash-based partitioning of resource descriptions and can evenly distribute resource information among partitions. It works with arbitrary attribute sets and can handle queries with wildcards at the end of the query string. Twine uses XML based resource description and from a resource description, it extracts each unique subsequence of attributes and values. Each such subsequence is called a *strand*. It then computes numeric values of these strands using hash function provided by Chord to maintain its efficient lookup system. The architecture and operation of Twine is summarized below.

```
<res>camera
    <man>ACompany</man>
    <model>AModel</model>
</res>
```

(a) Resource description



(b) AVTree

Figure 8: Example of a very simple resource description and its corresponding AVTree

3.2.1 System Architecture

One of the most important aspects of Twine architecture is the way it represents resource descriptions. It deserves special attention because of its uniqueness.

Resource Description Resources in INS/Twine are described with hierarchies consisting of attribute-value pairs as in XML. It then converts any such description into a canonical form : an attribute-value tree (AVTree). Figure 8 shows an example of a very simple resource description and its AVTree. Each resource description points to a name-record, which contains information about the current network location of the advertised resource, including its IP address, transport/application protocol, and transport port number. In INS/Twine, a resource matches a query if the AVTree formed by the query is the same as the AVTree of the original description, with zero or more truncated values. For example, the resource from Figure 8 would match the query: `<res>camera<man>ACompany</man></res>` or the query: `<res>camera</res>`.

Architecture Overview INS/Twine uses a set of resolvers that organize themselves into an overlay network to route resource descriptions to each other for storage, and to collaboratively resolve client queries. Each resolver knows about a subset of other resolvers on the network. Communication between resolvers takes place in the network core. These resolvers are known as *Intentional Name Resolver*s (INRs).

The architecture of INS/Twine is such that there are three layers in each INR, *Resolver*, *StrandMapper* layer and *KeyRouter* layer. Resolver interfaces with a local AVTree storage and query engine. When the Resolver receives an advertisement from its client appli-

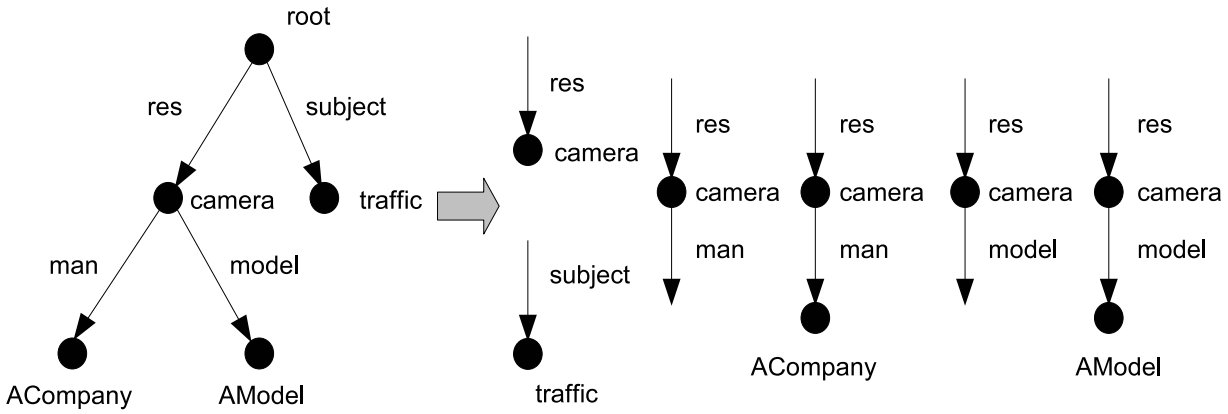


Figure 9: Splitting a resource description into strands

cation, it stores it locally using that engine. The Resolver then splits the advertised resource description into strands and passes each one to the StrandMapper layer. The StrandMapper maps the strand onto one or more numeric m-bit keys using a hash function. It then passes each key, together with the complete advertisement, to the KeyRouter layer. Finally, given a key, the KeyRouter determines which resolver in the network should receive the corresponding value and forwards the information to the selected peer. The KeyRouter may be thought of as a distributed hash table, where each node on the network keeps key-value bindings within a dynamically determined key range. It is built on top of Chord, which efficiently rebuilds its overlay network in the presence of failures. As a result, Chord actually handles the problem of finding the location of a specific key, which it can do in $O(\log N)$ complexity.

3.2.2 Partial Query Support

At the core of an INS/Twine resolver there is strand-splitting algorithm that extracts strands from a description. The goal of the algorithm is to break descriptions into meaningful pieces so resolvers can specialize around subsets of descriptions. At the same time, the splitting must preserve the description structure and support partial queries. Instead of independently mapping attribute-value pairs into keys, Twine algorithm for strand-splitting preserves the description structure and supports partial queries. It extracts each unique *prefix* subsequence of attributes and values from a description (either advertisement or query) as illustrated in Figure 9. As a result, Twine supports partial matching of *common_prefix** form only. Each such subsequence is called a strand. Each strand is then used to produce a separate key using 128-bit MD5 [13] hash function in the StrandMapper layer. Finally, as described earlier Chord handles the lookup operation for Twine.

The number of DHT-lookups increases with the number of property-value pairs in the advertisement (or query) and consequently the amount of generated traffic becomes high. Load-balancing also becomes a problem in this system. Peers responsible for small or popular strands become overloaded, and the overall performance degrades. To handle this problem, specific thresholds are used to cut down traffic into a peer.

3.3 Squid

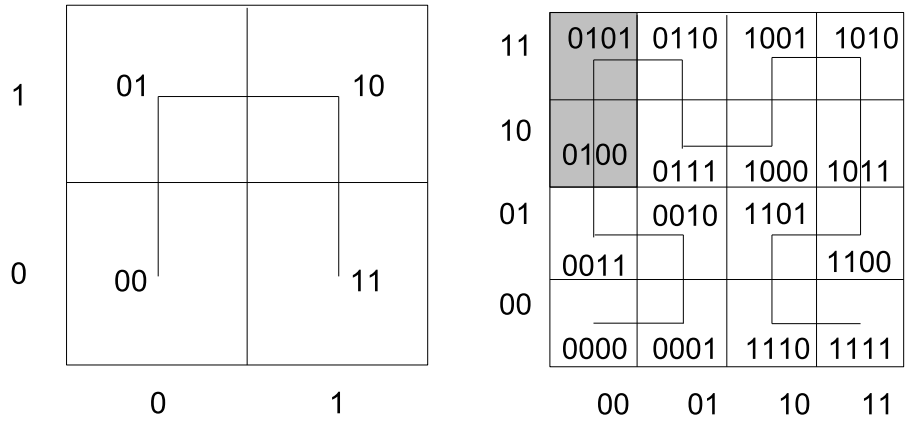
The Squid [15, 16] is a peer-to-peer information discovery system that supports complex queries containing partial keywords, wildcards, and ranges. Squid is built on a structured overlay (Chord) using its lookup protocol and guarantees that all existing data elements matching a query will be found with bounded costs in terms of the number of messages and nodes involved. The 'key' difference of Squid to other DHT based systems is the way it maps data elements to the DHT space. While other systems perform this mapping by using a hashing function that uniformly distributes data elements to nodes enabling them to perform a search only if the exact identifier is known, Squid uses a dimension-reducing mapping called a *space-filling curve* (SFC). SFC works as a recursive, self-similar, and locality preserving mapping function that enables Squid to support more complex queries than others.

3.3.1 Architectural Overview

The Squid architecture consists of a locality preserving mapping that deterministically maps data elements to indices, a mapping from indices to nodes in the overlay network and a query engine for routing and efficiently resolving keyword queries using successive refinements and pruning. As mentioned earlier Squid uses Chord as its overlay network and for routing. So the part of the architecture that stands out from all other service discovery systems is actually the locality preserving mapping of data elements to indices and then mapping of indices to nodes on the overlay network.

The idea *locality preserving* means that for some defined measure of locality, data elements are physically placed close together in the network. For example, if we consider lexicographical distance as a locality measurement then the words, *computer* and *computing* should be placed in nodes close together, if not the same one; whereas *newspaper* need not necessarily be close to them. To achieve this property of locality preservation Squid uses Hilbert space filling curve.

Hilbert Space Filling Curve An SFC is a *continuous* mapping from a d-dimensional space to a 1D space. If there is a d-dimensional cube then an SFC enters the cube in one



(a) 1^{st} order approximation (b) 2^{nd} order approximation

Figure 10: Hilbert SFC approximations for d = 2, n = 2

point, passes through all the points in the cubes volume exactly once and then comes out through another point. So using this mapping, it is possible to describe a point in the cube by its spatial coordinates.

The construction of SFCs is recursive. The d-dimensional cube is first partitioned into n^d equal sub-cubes. An approximation to a space-filling curve is obtained by joining the centers of these sub-cubes with line segments such that each cell is joined with two adjacent cells. Figure 10(a) gives an example of such a curve. The same algorithm is recursively used to fill each sub-cube, rotating and reflecting as necessary in order to keep the connecting line continuous. The line that connects n^{kd} cells is called the k^{th} approximation of the SFC. Figure 10(b) shows the 2^{nd} order approximation of Figure 10(a).

A unit length curve constructed at the k^{th} approximation has n^{k*d} equal segments, which are equally distributed in each sub-hypercube. If distances across the line are expressed as base-n numbers, then the numbers that refer to all the points that are in a sub-cube and belong to a line segment are identical in their first $(k * d)$ digits (Figure 10(a),10(b)).

The most important property to notice is the way it preserves locality. Points that are close together in the 1-dimensional space (the curve) are mapped from points that are close together in the d-dimensional space. For $k \geq 1$ and $d \geq 2$, the k^{th} order approximation of a d-dimensional Hilbert space filling curve maps the sub-cube $[0, 2^k - 1]^d$ to $[0, 2^{kd} - 1]$. The reverse is not true though; not all adjacent sub-cubes in the d-dimensional space are adjacent. A group of contiguous sub-cubes in d-dimensional space will typically be mapped to a collection of segments on the SFC known as clusters. The grayed portion in Figure 10(b) shows such a cluster.

In Squid, SFCs are used to generate the 1-d index space from the multi-dimensional

keyword space.

Mapping Indices to Peers Each data element is mapped to the first node on the Chord ring whose identifier is equal to or follows the key in the identifier space. The complete steps are, attaching keywords that describe the data elements content, using the SFC mapping to construct the data elements index, and finally, using this index, store the element at the appropriate node in the overlay.

3.3.2 Query Processing

As described earlier, data elements in the system are associated with a sequence of one or more keywords (up to d keywords, where d is the dimensionality of the keyword space). The queries can consist of a combination of keywords, partial keywords, or wildcards. For example, for a 2 dimensional keyword space (flight, schedule), (flight, info*), (flight, *) are all valid queries.

Processing a query is a 2-step task: translate the keyword query to relevant clusters of the SFC-based index space, and query the appropriate nodes in the overlay network for data-elements. If the query has no wildcard than it will map to at most one point in the index space, and node containing that index is found using Chord protocol. Otherwise, in presence of wildcards, the query identifies a set of indices in the index space. In Figure 10(b), the grayed portion corresponds to query (00, 1*). Depending on its size, a cluster may be mapped to one or more adjacent nodes in the overlay network. Once the clusters associated with a query are identified, straightforward query processing consists of sending a query message for each cluster.

This technique can be further refined by recursively processing the queries using the properties of Hilbert SFC. Details can be found in [16, Section 3.4].

4 Proposed Solution

We present a possible implementation of a service discovery system based on the Distributed Patter Matching System or DPMS. Using the inherent capability of DPMS to partially match any text, we believe that this newly proposed system will successfully be able to handle the problems that different existing service discovery technologies, e.g. SSDS, INS/Twine, Squid etc., face regarding subset matching. If the service advertisements as well as the queries can be expressed in a common form that can be used as input to Bloom Filter then this problem can easily be reduced to the searching problem addressed in DPMS.

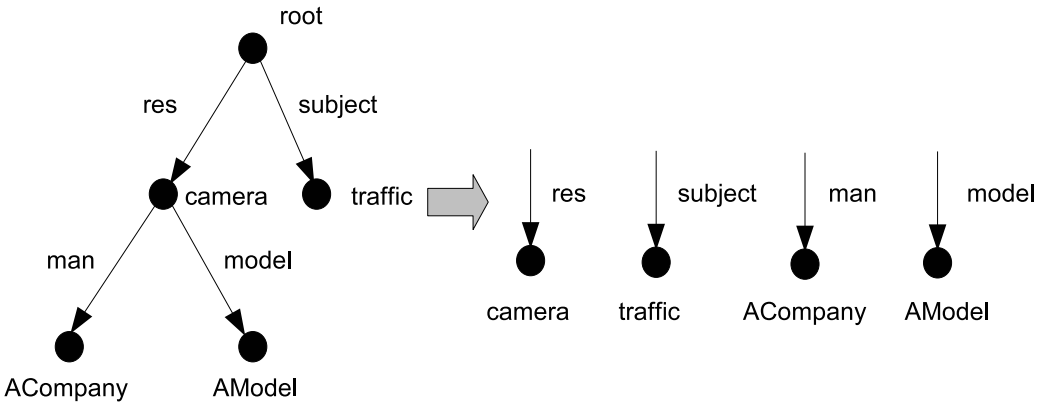


Figure 11: Splitting a resource description into edges

4.1 Architectural Overview

The basics are similar to DPMS (Section 2.2). As before, a peer can act as a leaf peer or an indexing peer. A leaf peer is at the bottom layer of the indexing hierarchy and advertises its indices (created from the advertisement) to other peers in the system. An indexing peer, on the other hand, stores indices from other peers (leaf peers or indexing peers) that are located in lower levels of the hierarchy.

4.1.1 Resource Description

As mentioned earlier, if we can convert an advertisement or a query into a form that can be used to create indices in DPMS, we can convert DPMS into an efficient service discovery system. Hence, a suitable representation of resources is of utmost importance in this proposed system. There are two choices to describe a resource. One is the trivial flat description of attribute-value pairs and the other is using XML for describing advertisements and queries. Even though XML has been extensively used in most of the current service discovery systems for its hierarchical representation of attributes, it may not be very useful when used with a DPMS based service discovery solution.

In general, DPMS uses the keywords used to describe an object to create indices using Bloom filters. So we can feed flat description of attribute-value pairs into Bloom filter to create index for an advertise or a query. On the other hand, we can also use XML and then break it up in 'strands' as in INS/Twine (Section 3.2) and then feed those strands into the Bloom filter to create the index. The problem with 'strands' have already been described before. In this representation, prefix information is embedded in each strand and as a result the main goal of using DPMS, wildcards at any position to perform a search, is hampered.

A better solution would be to use AVTree as in INS/Twine but to decompose it

differently. Instead of keeping the prefix information in each decomposed part by taking the paths from root to each node, we can use all the edges of an AVTree separately for indexing purpose. This would give succinct representation of advertisement and query along with flexibility to search for any portion independently. Figure 11 shows an example of this new splitting process. The output from the splitting process can then be fed into the Bloom filter to indices for advertisements as well as for performing queries.

Lookup operation then proceeds in a way similar to DPMS.

4.2 Security

DPMS has no security and privacy mechanisms in the system which can be improved by introducing secured connections and authentication procedures. As in SSDS, a global Certificate Authority (CA) can be introduced to ensure security. Each service provider should have a certificate signed by the CA, whose public key is well known. When a client tries to contact a service provider from the search result, it will contact the CA and in reply the CA will send back a matching certificate. That certificate will ensure that the client is contacting with a valid, authentic provider. Secured connections should also be enforced to make sure that valuable data transfered between client and service providers are protected from malicious attacks.

5 Conclusion

In this paper we have summarized some of the existing DHT-based service discovery systems along with their underlying lookup protocol as well as their capabilities and limitations in handling queries with wildcards i.e. multi-criteria search. We have also discussed a comparatively new lookup protocol, DPMS and proposed a service discovery system capable of wildcard matching based on DPMS. Performance of the proposed system should be similar to DPMS as presented in [2, Section 3.13]. But for actual results, which we intuitively believe to be satisfactory, we will have to conduct simulations on large data sets and compare the results with existing systems.

References

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Symposium on Operating Systems Principles*, pages 186–201, 1999.

[2] R. Ahmed. *Efficient and Flexible Search in Large Scale Distributed Systems*. PhD thesis, University of Waterloo, 2007.

[3] R. Ahmed and R. Boutaba. Distributed pattern matching for p2p systems. In *Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS)*, May 2006.

[4] R. Ahmed and R. Boutaba. Distributed pattern matching: A key to flexible and efficient p2p search. *IEEE Journal on Selected Areas in Communications (JSAC)*, 25(1):73–83, January 2007.

[5] R. Ahmed, R. Boutaba, F. Cuervo, Y. Iraqi, T. Li, N. Limam, J. Xiao, and J. Ziembicki. Service discovery protocols: A comparative study. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM) Application Sessions*, pages 22–37, Nice, France, 2005.

[6] M. Balazinska, H. Balakrishnan, and D. Karger. Ins/twine: A scalable peer-to-peer architecture for intentional resource discovery. In *Proceedings of International Conference on Pervasive Computing*, pages 195–210, 2002.

[7] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure service discovery service. In *Mobile Computing and Networking*, pages 24–35, 1999.

[8] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for secure wide-area service discovery service. *Wireless Networks*, 8:213–230, 2002.

[9] K. Decker, K. Sycara, and M. Williamson. Middle-agents for the internet. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 1997.

[10] P. Jones and D. Eastlake. RFC3174: US secure hash algorithm 1 (SHA1), 2001.

[11] Yin Li, Futai Zou, Zengde Wu, and Fanyuan Ma. Pwsd: A scalable web service discovery architecture based on peer-to-peer overlay network. *LECTURE NOTES IN COMPUTER SCIENCE*, 3007:291–300, 2004.

[12] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *In Proceedings of ACM SIGCOMM*, pages 161–172, 2001.

[13] R. Rivest. RFC1321: The MD5 message-digest algorithm, April 1992.

[14] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[15] C. Schmidt and M. Parashar. Enabling flexible queries with guarantees in p2p systems. *IEEE Internet Computing*, 8(3):19–26, June 2004.

[16] C. Schmidt and M. Parashar. Peer-to-peer approach to web service discovery. *WWW: Internet and Web Information Systems,*, 7:211–229, 2004.

[17] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003.