Rendezvous Points-Based Scalable Content Discovery with Load Balancing

Jun Gao[1] Peter Steenkiste[1,2]
[1]School of Computer Science
[2]Department of Electrical and Computer Engineering
Carnegie Mellon University
jungao, prs@cs.cmu.edu

ABSTRACT

A Content Discovery System (CDS) enables content consumers to discover content published by content providers via a set of content resolvers. Existing solutions to CDS systems have difficulties in achieving both rich functionality and scalability. In this paper, we present a distributed and scalable approach based on Rendezvous Points (RPs) that efficiently supports flexible search of dynamic contents. Our system deploys load balancing matrices (LBMs) to dynamically balance registration and query load across resolver nodes to maximize system utilization. Our system utilizes existing hash-based algorithms for content and query distribution within the overlay resolver network. We use simulation to validate our design.

Keywords

Content discovery, Rendezvous Points, Load balancing

1. INTRODUCTION

A Content Discovery System (CDS) is a distributed system that enables the discovery of contents. Such a system typically consists of three types of logical entities: content providers (servers) that publish and provide contents, content consumers (clients) that issue queries to locate contents, and content resolvers that determine the set of contents that match content consumers' queries.

There exists a wide spectrum of distributed applications that either themselves are a CDS or use a CDS as one of their major components. We use the term "content" in a broad sense and its meaning may differ from application to application. As an example, a service discovery system is a CDS system in which users discover and utilize the abundance of devices and sensors running on the Internet. In such a system, "content" refers to the description of a service or a device, e.g., a printer, or a camera. As another example, in recent years, peer-to-peer (P2P) applications, such as on-line music sharing and file swapping, have gained great popularity among Internet users. The primary component of a P2P application is also a CDS system. In this case, peers can act as servers, clients, or resolvers in that they can provide contents to share with others, can discover contents on other peers, and can also help to resolve other peers' queries.

We use a nationwide highway traffic monitoring service as an example to illustrate the type of applications we are targeting. Devices such as cameras and sensors are installed along the road side of highways or mounted on patrol cars, to monitor traffic status, road and weather conditions, and measure speed. Cameras and sensors must frequently send updates to the system to accurately reflect the status of the highways. Users may send a wide range of queries to the system, such as, "What is the speed at Fort Pitt Tunnel?", "Find a camera on Mt. Washington that overlooks the city and can accept new connections for live images", "Identify the highway sections to the airport that are icy, so that a driver can avoid them".

This example represents a large category of applications that pose the following challenges for a CDS system:

- Contents stored in the CDS must be searchable: A client can locate content via the CDS without having to specify its canonical name. Instead, it can do so by specifying a combination of attributes and values that describe the content.

- Contents stored in the CDS may be dynamic and independent from each other: The property of a particular content may be changing over time and so will be the description or the "name" of the content. In addition, unlike web pages that are linked to each other, content descriptions are often independent from each other, and they may not necessarily display hierarchical properties, as do domain names.

- The CDS must scale with both the registration and query load: In the above example, the number of devices and users may be in the order of hundreds of thousands, and the devices may update their information as frequently as every few minutes. By scalability we mean that the performance of the CDS, such

This research was sponsored by the Defense Advanced Research Project Agency and monitored by AFRL/IFGA, Rome NY 13441-4505, under contract F30602-99-1-0518. Additional support was provided by Intel.

NGC '02, October 23-25, 2002, Boston, Massachusetts, USA.

as success rate and response time for registrations and queries, should not degrade significantly as the amount of available contents, the rate of content registration and the rate of queries increase.

In this paper, we present a distributed and scalable approach to the content discovery problem that meets the above challenges. Content names and queries are represented with attribute-value pairs, and clients can search contents using any combination or subset of attributes and values. We achieve scalability through the use of Rendezvous Points (RPs). The RP-based registration and query design avoids network flooding. We address the load concentration problem in our RP-based design by balancing the load across resolver nodes using Load Balancing Matrices (LBMs).

The rest of the paper is organized as follows. In Section 2, we present the basic RP-based CDS system design. In Section 3, we discuss how load balancing addresses the load concentration problem. We present simulation results that demonstrate the effectiveness of the system in Section 4. We present related work in Section 5 and conclude in Section 6.

2. BASIC CDS DESIGN

In this section, we present the basic Rendezvous Point-based CDS design.

2.1 Naming scheme

2.1.1 AV-pair representation

Contents in our system are represented using attribute-value pairs (AV-pairs), e.g., a device in a service discovery system may be described with attributes such as `Type`, `Location`, and `Model`, etc. Similarly, in a music sharing application, the representation of a song may have attributes such as `Title`, `Artist`, `Album`, and `Year` etc. We refer to the collection of the AV-pairs as the content's "name", or content "description". Content discovery is the discovery of the "content name", not the content itself. This type of naming scheme has been used in other work, e.g., INS [3], SDS [5], Siena [4].

An AV-pair is written in the form of $\{a_i = v_i\}$, or $\{a_i v_i\}$ for short, where a_i is an attribute, and v_i is its value. We only consider exact matches in our current system. A content name that consists of n AV-pairs is represented as $CN : \{a_1v_1, a_2v_2, ..., a_nv_n\}$. In an actual implementation, a more powerful language like XML [1] may be used to represent names. Figure 1 is an example name for a highway monitoring camera.

```
Camera ID = 5562
Camera Type = Q-cam
Highway Number = I-279
    Exit Number = 4
City = Pittsburgh
Speed Measured = 45MPH
Road Condition = dry
Connection availability = yes
```

Figure 1: An example content name.

In this paper, we only consider names that are comprised of *orthogonal* attributes, which means they exist independently of each other. Attributes that *depend* on the presence of other attributes in a name, such as `Exit Number` in the above example, can be dealt with similarly. Dynamic attributes, such as `Speed Measured`, may take on different values at different times. When the value changes, the content name effectively changes. Dynamic content makes caching unsuitable, and requires continuous active registering.

2.1.2 Searchability and subset matching

A query is also comprised of a set of AV-pairs, e.g., $Q : \{a_1v_1, a_2v_2, ..., a_mv_m\}$ contains m AV-pairs. The matched content name must simultaneously satisfy *all* the AV-pairs present in the query.

Our CDS supports *subset matching*, i.e., a content name matches a query if and only if the set of AV-pairs in the query is a subset of the set of AV-pairs in the content name. In other words, there may be AV-pairs in that query that are not in the content name. Subset matching enables searchability in that to find a content name, the query does not have to have the exact same set of AV-pairs as the content name. The number of subsets (excluding the empty set) of a content name that consists of n AV-pairs is $2^n - 1$, which means it can match $2^n - 1$ different queries.

2.2 CDS overlay network

Nodes participating in the CDS connect to each other in a peer-to-peer fashion to form a CDS overlay network. Figure 2 shows the software architecture on a node. The CDS layer is designed as a common communication layer on which higher level applications, such as service discovery and file sharing, can be built. The primary responsibility of the CDS layer is to decide on the set of nodes that a content name or query should be sent to. For the actual forwarding of messages, the CDS system utilizes existing hash-based overlay network mechanisms ([16], [13], [14], [18]).

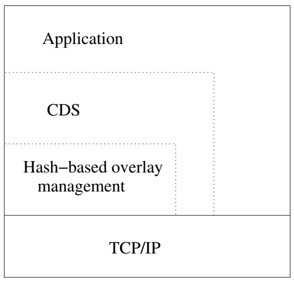

Figure 2: CDS node architecture.

Before a node can join the CDS system, it must first join the overlay network. It locally computes its overlay network address (node ID) by using a system-wide hash function $\mathcal{H}$. The overlay network is structured in such a way that node IDs encode network topological information: The node ID determines the set of nodes that this node will be neighboring with, and which node to use when forwarding a message in the overlay network. To preserve scalability, the number of neighbors is kept relatively small in these schemes, e.g., $O(\log N_c)$ in Chord[16], where N_c is the number of nodes in the network. The path length between any two nodes in the overlay is also kept short, e.g., $O(\log N_c)$ in terms of overlay hops in Chord.

2.3 Rendezvous Points-based content distribution and discovery

The fundamental problem in designing the CDS is where to store the content names. In a centralized system, names and queries are sent to one central location, which forms the single point-of-failure of the system and may become the bottleneck as load increases. Approaches based on flooding the CDS network with registrations or queries eliminate the single point-of-failure, but the amount of duplicated contents or queries makes them unscalable.

To address the scalability problem, we introduce an approach based on *Rendezvous Points*(RPs). In this approach, a content name is registered only with a small set of nodes in the system, known as the Rendezvous Points; thus full duplication of contents at all nodes is avoided. Queries are directly sent to the proper RPs for resolution, and no network-wide searching is needed. The term "rendezvous" is used because the RPs are where queries and the matched contents meet. This approach is in part motivated by the PIM-SM [6] multicast protocol, which reduces the amount of group information that must be stored on each router, and thus improves the scalability of IP multicast.

2.3.1 Registration with the RP set

To register a content name, the provider node must first determine the set of resolver nodes that should receive this name. It does this by applying the system-wide hash function, $\mathcal{H}$, to each AV-pair in the content name. For example, given content name $CN : \{a_1v_1, a_2v_2, ..., a_nv_n\}$, which has n AV-pairs, the provider computes the following:

$$\mathcal{H}(a_iv_i) = N_i$$

for $i = 1, n$. For N_i, if there exists a node whose ID is equal to N_i, then it will become one RP node; otherwise, the node whose ID is numerically closest to N_i will become the RP node. These nodes (n of them assuming no hash collision) form the RP set for this content name. The content name is then sent to each of the n nodes (Figure 3), which results in n replications of the name. From an RP node's point of view, it becomes a specialized node for the AV-pairs that are mapped onto it, e.g., N_1 contains all the names in the system that have a_1v_1 in their names.

This scheme of hashing each AV-pair individually has the following properties. First, it yields an RP set size of n for a name that has n AV-pairs, and it requires the content provider to send $O(n)$ messages per registration. In practice, n is typically a small number on the order of 20 to 30, and thus registration is done efficiently. Second, it guarantees system correctness, in that, any query that is a subset of a content name, e.g., the query $Q : \{a_1v_1\}$, which contains only one of CN_1's AV-pair, can discover CN_1 by going to node N_1. As a comparison, registering with all nodes corresponding to all the $2^n - 1$ subsets of the content name would also ensure correctness, but requires an exponential number of registration messages. Third, from the system's point of view, hashing attribute and value together to determine the set of RP nodes rather than hashing attribute only provides a natural way of spreading registrations to more nodes in the system.

RP nodes store names in a soft state fashion, i.e., registrations automatically expire after a certain time interval. Therefore names must be periodically refreshed to avoid expirations. This provides protection against certain types of failures. For example, when an RP node leaves or crashes, the refresh messages will allow a lost content name to be registered at an active node. Also, when a name contains dynamic attributes, the RP nodes for the dynamic attributes may be different every time the value changes.

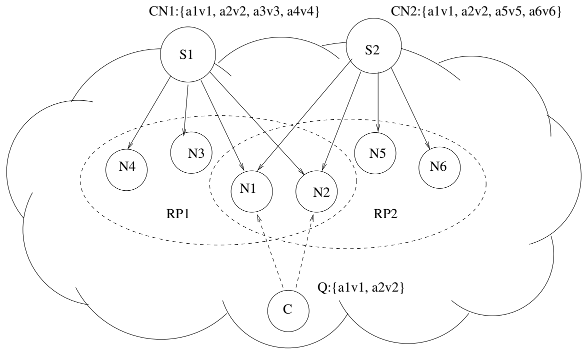

Figure 3: Example registration and query processing with RP set.

2.3.2 Query resolution

To resolve queries, clients must determine the set of RPs that may contain matching content names. As described in the previous section, all content names that contain a_iv_i are stored in the node $N_i(=\mathcal{H}(a_iv_i))$, therefore query $Q : \{a_1v_1, a_2v_2, ..., a_mv_m\}$ can be sent to any one of the m RP nodes, $N_1, ..., N_m$ (Figure 3). Since an RP node maintains the full list of the AV-pairs in a name, when a query arrives at this node, it can fully determine whether a name matches this query by comparing the name's AV-pair list with that of the query's.

Given these m candidate RP nodes, the client may pick one randomly, or send one query message to each of the m nodes. In addition, the client may use the following two-pass *query optimization* algorithm to select an RP node. It first sends a probe message to all the m nodes and then selects one based on their responses. The client has two choices: (1) it can send the query to the node that has the smallest response time, or (2) it can send to the node that has the smallest database. Choice (2) is often more beneficial from a system's point of view. As we will discuss in the next section, a node may have to replicate its database on other nodes when it observes high query load. For efficiency, the replication of a smaller database is preferred. Both with and without query optimization, the number of messages required to resolve a query is $O(m)$, which preserves system scalability.

An alternative to having queries fully resolved at the RP node is to have clients resolve the query based on information provided by the RP nodes. In this approach, the client has to send its query to multiple resolver nodes, each of which resolves the query partially and returns any matches. The client then performs a "join" operation to determine the final set of matched names. While this approach reduces the computation load on resolver nodes, it adds potentially significant communication overhead due to large sets of partial matches to the network and client. Given that exact matching for AV-pairs is a relatively light-weight operation, it is more efficient to do the matching on the RP nodes.

3. SYSTEM WITH LOAD BALANCING

We extend our basic system to deal with load imbalances.

3.1 Load concentration problem

The basic system tries to spread registrations to different nodes in the network by using a different set of RP nodes for different content names. However these sets may still overlap if the names share some common AV-pairs, e.g., node N_1 and N_2 in the scenario shown in Figure 3. Similarly, queries that share an AV-pair may also be sent to the node that corresponds to the common AV-pair.

Registration and query load observed on each resolver node are determined by the AV-pair distributions in content names and queries. These distributions are likely to be skewed as some AV-pairs are common or significantly more popular than others. For instance, it has been observed that the popularity of keyword search[1] strings in Gnutella follows a Zipf-like distribution [15]. In the CDS system, consider the case where the number of names that contain a_iv_i, $N_{a_iv_i}$, follows a Zipf distribution:

$$N_{a_iv_i} = N_s \cdot k \cdot \frac{1}{i^\alpha},$$

for $i = 1...N_d$, where N_d is the number of different AV-pairs in the system. k and α are two parameters with α close to 1. N_s is the total number of names in the system and i is the rank of AV-pair a_iv_i in terms of its frequency of occurring in names; $i = 1$ corresponds to the AV-pair that is contained in the most number of names. Suppose in a particular application, there are $N_s = 10^5$ names, and $k = 0.5, \alpha = 1$. In this example, half of the 10^5 names would contain the most popular AV-pair, which would be mapped onto one node in the basic system. On the other hand, for nodes that correspond to AV-pairs ranked from 10^3 to 10^4, each would receive fewer than 50 names. Clearly, a few nodes will be swamped by registrations, while the majority of nodes in the system are only rarely used.

In this section we introduce a distributed load balancing solution that allows the CDS to utilize lightly loaded nodes to share the registration and query load of heavily loaded nodes.

3.2 Load balancing matrix (LBM)

For a popular AV-pair, the CDS system uses a set of nodes instead of one to share the registration and query load. This set of nodes is organized into what we call the Load Balancing Matrix (LBM). Figure 4 shows the layout of the matrix for AV-pair a_iv_i. A node in the matrix is denoted by $N_i^{(p,r)}$, where p is its column index and r is the row index. The nodes' IDs are determined by applying the hash function, $\mathcal{H}$ to the AV-pair, the column and row indices:

$$N_i^{(p,r)} = \mathcal{H}(a_iv_i, p, r).$$

As an optmization technique, for each matrix, a node $N_i^{(0,0)} = \mathcal{H}(a_iv_i, 0, 0)$, called the "head node", is used to store the size of the matrix.

Each column contains a subset, or *partition*, of the content names that contains a_iv_i. Nodes in the same column are *replicas* of each other, i.e., they all host the same set of names. The matrix dynamically grows or shrinks depending on the load along its two dimensions. New partitions are added when the registration load of the pair a_iv_i increases, and new replicas are added when the query load increases. The matrix may consist of only one node when the load is low, as was the case in the basic system. It may also be in the shape of one row, when only the registration load is high, or one column, when only the query load is high.

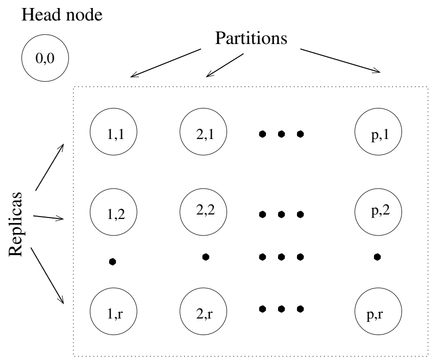

Figure 4: Load balancing matrix for a_iv_i.

To grow and shrink the matrix, each node maintains the following statistics: the number of names it receives, the registration and query rates. Corresponding to these statistics, each node also keeps three thresholds: T_{CN}, the maximum number of content names a node can hold; T_{reg}, the maximum rate of registration it can sustain; and T_q, the maximum query rate the node can sustain. It is worth noting that in our design, the decisions on when to grow or shrink a matrix are made distributedly by individual nodes based purely on local information.

We now describe the registration and query operations with a matrix and the matrix management mechanisms. For simplicity, we assume all nodes are homogeneous in that they have the same processing power and network connectivity.

3.3 Registration and partition management

Unlike in the basic system, where a content provider registers its content name with each RP node that corresponds to an AV-pair, with LBM, the provider must register its content name with a column of nodes in each matrix that corresponds to an AV-pair (Figure 5).

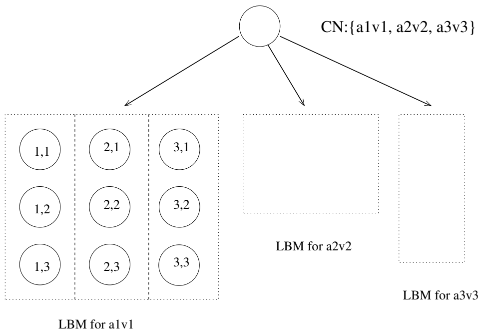

Figure 5: Registration with load balancing matrices.

The registration pseudo code is shown in Figure 6. To register with the matrix LBM_i corresponding to AV-pair a_iv_i, the content provider must first discover its size: the

[1]Note that AV-pair based search is more general than keyword based search.

1: Register_Name(`name`) {
2: **foreach** AVpair a_iv_i **in** `name` {
3: **do** {
4: $N_i^{(0,0)} \leftarrow \mathcal{H}(a_iv_i, 0, 0)$;
5: (P, R) $\leftarrow$ retrieve_matrix_size($N_i^{(0,0)}, a_iv_i$);
6: p $\leftarrow$ generate_random_number(1, P);
7: r $\leftarrow$ 1;
8: $N_i^{(p,r)} \leftarrow \mathcal{H}(a_iv_i, p, r)$;
9: flag $\leftarrow$ send_to($N_i^{(p,r)}$, `name`);
10: } **until** (flag **equals to** SUCCESS);
11: }
12:}

Figure 6: The algorithm for content providers to register with LBM.

1: Receive_Name(`name`, a_iv_i) {
2: check_node_status();
3: **if** (reach T_{CN} **or** T_{reg}) {
4: reject(`name`);
5: **if** (partition_added_flag **equals to** FALSE) {
6: $N_i^{(0,0)} \leftarrow \mathcal{H}(a_iv_i, 0, 0)$;
7: update_matrix_size($N_i^{(0,0)}, a_iv_i$, `INC_P`);
8: }
9: } **else** {
10: insert_to_database(`name`);
11: }
12:}

Figure 7: The algorithm for adding new partitions to LBM_i used by RP nodes for AV-pair a_iv_i, which is contained in `name`.

number of partitions, P, and the number of replicas, R. It can do so in several ways. The most efficient way is to retrieve the size from the pair's corresponding head node (line 4, 5). In the case that the head node is down or becomes a bottleneck, the provider may find out the matrix size by directly sending probe messages to the nodes in the matrix. For example, to discover P, the provider may first estimate a maximum number P_0, and probe a node in the P_0th partition, e.g., $N_i^{(P_0,1)}$. Node $N_i^{(P_0,1)}$ can determine whether it belongs to LBM_i by examining its database: if it has seen a_iv_i before, then it does; otherwise, it does not. Since the partitions are ordered, the current number of partitions can be discovered in $O(\log P_0)$ steps via binary probing between partition 1 and P_0. Content providers may cache an AV-pair's matrix size and use it without rediscovering it. This will for example be useful when refreshing registrations.

Once the size of the matrix is found, the content provider selects a random partition between 1 and P (line 6) and registers with the nodes in that column (line 8, 9). The provider may send the registration to the first node in the column (line 7) and let it propagate the message to other nodes, or it can register with each node directly. In case the registration at this selected partition fails, the provider will have to repeat the registration process (line 10). In practice, an upper limit is set for the maximum number of partitions a matrix can have, and a registration will fail if its corresponding matrix has reached the maximum value.

New partitions are introduced to a matrix when the matrix receives more names than the current partitions can host or when the name registration rate observed by the matrix exceeds what the matrix can sustain. Figure 7 shows the pseudo code for adding new partitions to the matrix. New partitions are added only by nodes in the last partition. For example, when a node in the last partition, $N_i^{(P,r)}$, reaches threshold T_{CN} or T_{reg} (line 3), it will send an `INC_P` message to the head node (line 6, 7). When the head node receives such a message, it increases the P value corresponding to this AV-pair. In an additive increase approach, it increases P value by 1, whereas in a multiplicative approach, the P value is doubled every time. We are evaluating both approaches. Suppose only one new partition is added, then nodes in the new column have the following IDs: $\mathcal{H}(a_iv_i, P+1, r) = N_i^{((P+1),r)}$. Based on the registration algorithm, future registrations will be shared by the $P+1$ partitions.

The number of partitions of a matrix should be kept as small as possible, since more partitions would require clients to send more messages when resolving queries. Push and pull techniques are used to reduce the number of partitions, e.g., older partitions, when they have space available, can "pull" names from the last partition. When the last column is not holding any content, it will remove itself from the matrix. The head node will then decrease P by 1.

3.4 Query and replication management

Similar to the basic system, clients can use any matrix that corresponds to an AV-pair in the query for query resolution. To collect all the possible matched names, the client must send the query to all the columns of the selected matrix because each column stores only a fraction of the names that have that AV-pair. In selecting a matrix, the client may use the query optimization mechanism to probe the head node of each corresponding matrix to get their sizes and then select the one that has the fewest columns. Since the nodes in the same column are replicas of each other, the query needs only to be sent to one node in each column. To balance the load within each column, the client picks a random one. Thus, the query load observed on each node in a matrix is uniformly distributed.

Suppose matrix LBM_i currently has R rows, or replicas. New replicas will be added to the matrix as the query load of the matrix increases. In particular, when the query rate observed by a node $N_i^{(p,R)}$, in the last row, reaches T_q, it will send an `INC_R` message to the head node $N_i^{(0,0)}$. Suppose replicas are added additively, then the head node will increase R by 1 and send `DUPLICATE` messages to all nodes in the last row, asking them to replicate themselves. For example, node $N_i^{(1,R)}$ will send its names that contain a_iv_i to node $N_i^{(1,R+1)}$.

More replicas in the matrix means providers must register with more nodes. To reduce the number of replicas, when the last node in each column observes that its query rate drops below a threshold, it can remove itself from the matrix. Due to the uniform query load distribution in the matrix, in fact, the last row will be removed.

3.5 System properties with LBM

When LBMs are created, both the registration and query costs increase. Consider matrix LBM_i which corresponds to a_iv_i and has P partitions and R replicas. Registering a name with LBM_i takes $O(R)$ messages. A client that chooses LBM_i to resolve its query that contains a_iv_i must send $O(P)$ messages to LBM_i. The values of P and R increase linearly with the overall registration load and query load of a_iv_i respectively. Our design ensures that a matrix is not large in both dimensions: the query optimization mechanism makes sure that a matrix that has many partitions will not have many replicas. This is important because we expect popular AV-pairs to show up both in many registrations and many queries.

From a resolver node's point of view, its observed load is limited by the thresholds, and thus it can process registrations and queries efficiently. From the system's point of view, it does not reject registrations or queries prematurely, and tries to balance load across all nodes until every node reaches its capacity.

4. SIMULATION EVALUATION

In this section, we use simulation results to demonstrate the effectiveness of the designed system.

4.1 Simulator implementation

We developed an event-driven simulator to evaluate the CDS system. The simulator implements both the basic design and the load balancing mechanisms, excluding the matrix shrinking. Events in the simulator include registrations, queries, probe messages, and their replies. A node uses an FCFS queue to process registrations and queries with exponentially distributed service rates. Each node estimates the registration and query rates it observes based on a simple sliding window of recently received registrations and queries.

The simulator uses a 24-bit name space for node IDs and the hashed values of AV-pairs, and node IDs are assigned in such a way that each node covers an equal slot in the entire name space to ensure an even mapping of hashed values onto nodes. One requirement of the hash function used by the system is that it must be able to generate values uniformly distributed in the name space and insensitive to the input. The simulator uses Unix's random number generator to produce "hashed" values, and in practice, deterministic cryptographic functions such as SHA-1 should be used. The simulator assumes the existence of Chord-like underlying overlay management and routing mechanisms.

4.2 Experiment setup

In the following experiments, we consider a CDS network that has $N_c = 10^4$ nodes. There are 50 attributes in the system, and each of which can take on 200 values ($N_d = 10^4$ distinct AV-pairs).

As workload, we generate two sets of content names and one set of queries. Each name dataset contains 10^5 names and each name is comprised of $n = 20$ AV-pairs. The AV-pair distributions in names are shown in Figure 8. In the uniform dataset, each AV-pair is equally likely to appear in a name, and on average each AV-pair occurs in 200 names. In the skewed case, some AV-pairs are assigned higher weights, and the overall distribution of AV-pairs is Zipf-like, as it is close to a straight line in the log-log plot($\alpha \sim 0.88$), and the most popular AV-pairs are contained in over 23,000 names.

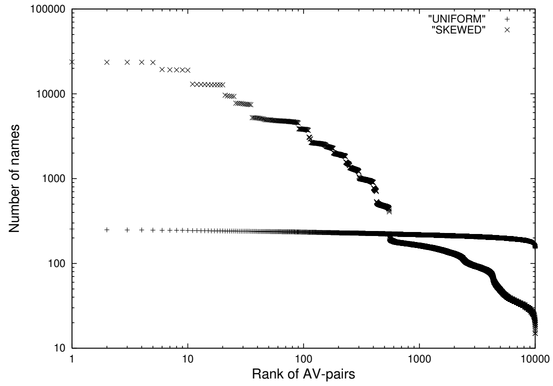

Figure 8: AV-pair distributions in two sets of content names.

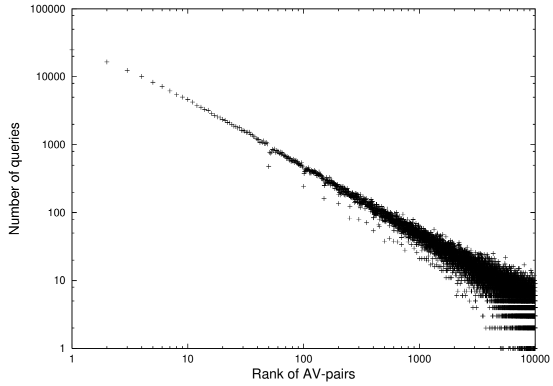

Figure 9: AV-pair distribution in queries.

The query dataset contains approximately 10^5 queries and is generated based on a Zipf distribution with $\alpha = 1$ (Figure 9). The maximum number of AV-pairs a query has is 10 and the average is 4. The most popular AV-pair in queries occurs in about 25,000 queries. Both the arrival times for names and queries are modeled with a Poisson distribution.

In our experiments, we limit the size of a matrix for any AV-pair to be at most 10×10, i.e., 10 partitions and 10 replicas. The size is a tunable parameter, whose value should be determined based on an application's requirement. The larger the size is, the more registration and query load the system can sustain, but with higher cost.

4.3 Balanced name distribution

We first study how well the system balances content names across nodes by applying the two name datasets to the system under two schemes: the basic system without LBMs, and the system with LBMs. To deploy LBMs, a node can either set T_{CN} to limit the number of names it will receive, or set T_{reg} to limit the rate of registrations coming to it. In this set of experiments, T_{CN} is set and LBMs are created to hold extra registrations when a node reaches threshold. A smaller T_{CN} is used for the uniform set to ensure LBMs will be deployed.

Results are shown in Table 1. For the skewed dataset, when no thresholds are set, no LBMs are created, and as expected, there is a large variance in the name distribution, e.g., some nodes receive as many as 100 times more names than the average, and some nodes do not get any names. For the uniform dataset, names are distributed more evenly, but since the hashed values of two different AV-pairs may fall into the range of the same node, a node may be responsible for multiple AV-pairs, and receive more than the average number of names. With LBM, in both cases, the maximum number of names a node stores is reduced to the desired threshold, and names are distributed more evenly: standard deviations are reduced by 48% and 40% respectively.

Datasets	Schemes	Max	Min	Avg.	Std. dev.
Skewed	No LBM	23816	0	200	989
	$T_{CN} = 2500$	2500	0	200	512
Uniform	No LBM	1218	0	200	200
	$T_{CN} = 320$	320	0	200	120

Table 1: Statistics of name distribution on nodes.

4.4 Registration scalability

In the following experiments, we assume each node has approximately $500Kbps$ available link bandwidth (DSL level), which is the limiting factor of the node's performance. Corresponding to the bandwidth, each node sets a threshold of $T_{reg} = 50reg/sec$ as the maximum sustainable registration rate (assume the registration packet size is 1000 bytes). We examine how the system scales with registration load by showing the registration success rate.

Figure 10 plots the effect of arrival rate on registration success rate. For the uniform dataset, the system works well without using LBMs as the success rate remains over 95% until the overall rate observed by the system, t_{system}, reaches $2500reg/sec$, which translates to $t_{node} = 5reg/sec$, the average registration rate on a node.[2] The success rate starts dropping before each node reaches the threshold because: (1) names are not truly uniformly distributed on nodes, as seen in the previous section, and (2) T_{reg} is the peak rate threshold. When LBMs are used, partitions are added to matrices, and the system can maintain 100% success rate until the average registration rate on each node reaches 40% of the threshold.

For the skewed dataset, the system behaves similarly: with LBMs, the system can sustain a registration rate that is 6 times higher than without LBMs, while maintaining 100% success rate. Due to the extreme name concentration, in both schemes, with and without LBMs, the success rates drop below 100% at a lower incoming rate compared to the uniform dataset. Additional experiments, where more than 10 partitions are allowed for each LBM, show that success rates can be further increased for a given registration rate.

4.5 Query scalability

To test the system's scalability property with regard to queries, we first inject the skewed name dataset into the system and then issue the set of Zipf queries. The query rate threshold on each node is set to be $T_q = 200q/sec$ (assume the query packet size is 250 bytes). RP nodes for queries are selected using three different schemes: (1) as a base case, use the most popular pair in the query, which is the worst choice; (2) select a random pair; (3) use the query optimization mechanism, i.e., select the pair that has

[2] $t_{node} = t_{system} \cdot n/N_c$. For $n = 20$ and $N_c = 10^4$, $t_{node} = t_{system}/500$.

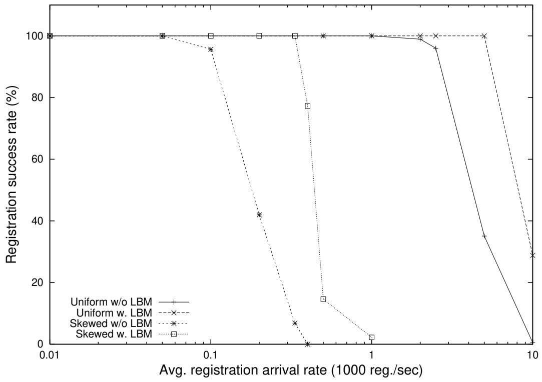

Figure 10: Registration success rate comparison.

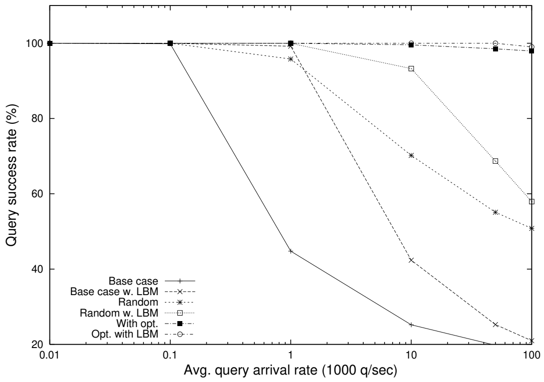

Figure 11: The success rate of queries under three schemes: (1) base case, (2) use random pair, (3) with query optimization. ("with LBM" means allowing replications in the matrices).

the smallest matrix in terms of the number of partitions. A query is declared successful if it is not rejected by the intended RP node regardless of whether there are matching names on the node. For a fair comparison, each query is sent to the system only once.

Figure 11 shows the success rates from six series of experiments. First we observe that the query optimization mechanism plays an important role in improving system's performance: choosing the wrong AV-pair will easily swamp a few RP nodes and cause large failure rates, and selecting a pair randomly does better, but since there are only a few AV-pairs in each query, there is still a good chance a popular AV-pair is chosen. With query optimization, the query load is spread to rarely used nodes, and thus the system can sustain much higher query load.

The effect of LBM, i.e., allowing nodes to adaptively replicating themselves when load is high, is demonstrated by the base scheme and the random scheme. For example, in the base case, the success rate increases from 44.76% to 99.21% when the query rate is $10^3 q/sec$, and in the random case, it increases from 70.19% to 93.26% when the query rate is $10^4 q/sec$. In the query optimization scheme, because load is distributed across nodes, even under the highest query rate in our experiments, the average rate on a node is far below T_q. LBMs are needed only occasionally, and when they do the success rate is further increased.

5. RELATED WORK

As with the wide range of applications that use CDS systems, there are a variety of solutions to the CDS system. However, these solutions have difficulties in achieving both scalability and rich functionality. Based on how the resolvers are organized, we classify the solutions into two categories: centralized and distributed.

In centralized solutions ([7],[12],[2]), the central resolver(s) is the bottleneck and single point-of-failure. Distributed solutions based on an unstructured general graph type of resolver network ([3],[4],[9],[8]) often need either registration flooding or query broadcasting, which does not scale with content names and queries. A hierarchical organization of resolvers ([17],[5],[10],[11]) scales better but works mostly for hierarchical content names, e.g., domain names. Recent hash-based peer-to-peer lookup systems ([16],[13],[14],[18]) improve scalability by organizing resolvers into a structured network according to hashed IDs, and provide efficient content name look up by associating names with resolvers. However these systems typically do not provide content searchability, and we use these mechanisms as the underlying substrate in our CDS system.

6. CONCLUSIONS

In this paper, we presented a distributed and scalable approach based on Rendezvous Points to the content discovery problem. The RP-based content distribution and discovery mechanism allows the CDS system to scale with the amount of content names and queries by avoiding network-wide flooding. Load balancing matrices are deployed to maximize system's utilization and eliminate hot-spots. Our approach is fully distributed in that entities in the system, resolvers, content providers and clients, can make decisions and compute destinations based on local information and by taking advantage of the hash-based system.

Our simulation results confirmed the system's scalability properties. We are now completing the simulator implementation and conducting a more comprehensive evaluation. We will also evaluate the system on the Internet with a real implementation.

7. ACKNOWLEDGMENTS

We thank Syed Umair Ahmed Shah for many valuable discussions.

8. REFERENCES

[1] Extensible Markup Language(XML). http://www.w3.org/XML.

[2] Google inc. http://www.google.com/.

[3] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The Design and Implementation of an Intentional Naming System. In *Proceedings of SOSP 1999*, Kiawah Island, SC, December 1999.

[4] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[5] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An Architecture for a Secure Service Discovery Service. In *Proceedings of Mobicom 99*, Seattle, WA, August 1999.

[6] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, and L. Wei. The PIM Architecture for Wide-area Multicast Routing. *ACM Transactions on Networks*, April 1996.

[7] Elvin. http://elvin.dstc.edu.au/.

[8] Freenet. http://freenet.sourceforge.net/.

[9] Gnutella. http://gnutella.wego.com/.

[10] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol. IETF, RFC 2165, November 1998.

[11] P. Mockapetris. Domain Names - Concepts and Facilities. IETF, RFC 1034, November 1987.

[12] Napster. http://www.napster.com/.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of SIGCOMM 2001*, pages 161–172, San diego, CA, August 2001.

[14] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. Submitted, May 2001.

[15] K. Sripanidkulchai. The Popularity of Gnutella Queries and Its Implications on Scalability. http://www.cs.cmu.edu/ kunwadee/research/p2p/gnutella.html.

[16] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of SIGCOMM 2001*, pages 149–160, San diego, CA, August 2001.

[17] H. Yu, D. Estrin, and R. Govindan. A Hierarchical Proxy Architecture for Internet-scale Event Services. In *Proceedings of WETICE 99*, Stanford, CA, June 1999.

[18] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, U. C. Berkeley, April 2001.