Design and Evaluation of a Distributed Scalable Content Discovery System

Jun Gao, *Student Member, IEEE* and Peter Steenkiste, *Senior Member, IEEE*

***Abstract*— A Content Discovery System (CDS) allows nodes in the system to discover contents published by some other nodes in the system. Existing CDS systems have difficulties in achieving both scalability and rich functionality. In this paper, we present the design and evaluation of a distributed and scalable CDS. Our system uses Rendezvous Points (RPs) for content registration and query resolution, and can accommodate frequent updates from dynamic contents. Contents stored in our system can be searched via subset matching. We propose a novel mechanism that uses load balancing matrices (LBMs) to dynamically balance both registration and query load across nodes in the system to maintain high system throughput even under skewed load. Our system utilizes existing Distributed Hash Table (DHT) mechanisms for CDS overlay network management and routing. We validate our system's scalability and load balancing properties using extensive simulation.**

***Index Terms*— Content Discovery System, Rendezvous Points, Load Balancing.**

I. Introduction

A Content Discovery System (CDS) is a distributed system that enables the discovery of contents. Nodes in such a system form an overlay network, the CDS network. A node in the system can publish and provide contents, issue queries looking for contents, store contents or contents' metadata published by other nodes, and resolve other nodes' queries. There exists a wide spectrum of distributed applications that either themselves are CDS systems or use a CDS as one of their major components. Some examples are service discovery services, peer-to-peer object sharing systems, sensor networks and publication-subscription (pub/sub) systems.

We illustrate the type of applications we are targeting with the following example. Consider a nationwide highway traffic monitoring service, where devices such as cameras and sensors are installed along the roadside of highways or mounted on patrol cars, to monitor traffic status, road and weather conditions. These devices must frequently send updates to the system to accurately reflect the current status of the highways. In this example, "content" refers to the description of a device, and the CDS must be able to answer a large range of user queries, for instance, "*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 when designing a CDS system:

- Contents stored in the CDS must be searchable. A node can locate contents without having to use their canonical names. Instead, it should be able to do so by specifying a combination of attributes and values that describe the contents.
- The CDS must be able to handle frequent updates of dynamic contents. The description, or "name", of a piece of content may change over time. For example, when a camera observes a different speed, it must change its description and announce it to the CDS system.
- The CDS must scale with both the registration and query load. By scalability we mean that as the load (e.g., the registration and query rate) to the system increases, the performance of the CDS, such as throughput and response time, must not degrade significantly before the system as a whole reaches its capacity.

The primary task of a CDS is to efficiently locate the set of contents that matches a client's query. Existing CDS systems have difficulties in achieving both rich functionality and scalability. At one end, they may be able to scale to the Internet level but offer limited functionality, e.g., they support exact content name lookup ([1], [2], [3], [4]) only, or the search of strictly hierarchical content names [5], or they consider static contents only, e.g., search engines [6]. At the other end, they may offer powerful functionality such as the searching of general content names, but are not scalable [7].

In this paper, we present the design, implementation and evaluation of a distributed content discovery system that meets the above challenges. Content names in our system are represented by attribute-value pairs for searchability. We achieve scalability through the use of *Rendezvous Points (RPs)*. The RP-based scheme avoids network-wide message flooding at both registration and query time. We design a novel mechanism that uses *Load Balancing Matrices (LBMs)* to dynamically balance both registration and query load in the system to improve the system's throughput under skewed load.

The rest of the paper is organized as follows. In Section II, we present the CDS system architecture. In Section III, we present the basic RP-based CDS design. We present our distributed load balancing mechanism in Section IV. Section V describes the evaluation methodology, and we present simulation results in Section VI. We discuss related work in Section VII and conclude in Section VIII.

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.

Jun Gao is with the Computer Science Department, Carnegie Mellon University.

Peter Steenkiste is with the School of Computer Science and the Department of Electrical and Computer Engineering, Carnegie Mellon University.

II. System Architecture

Nodes participating in the CDS connect to each other in a peer-to-peer fashion to form a CDS overlay network. Figure 1 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 CDS layer is in turn built on top of a scalable distributed hash table (DHT), such as Chord[1], CAN [2], Pastry [3], and Tapestry [4].

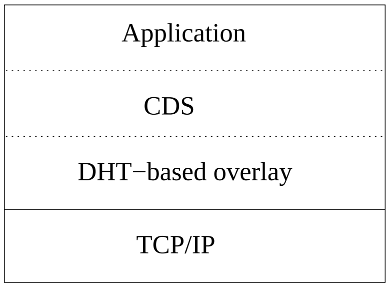


Fig. 1. CDS node architecture.

A. AV-pair Based Content Naming Scheme

To provide content searchability, applications built on top of the CDS layer use a flexible attribute-value based naming scheme, similar to what is used in [8], [9]. Contents are represented using attribute-value pairs (AV-pairs). For example, in a service discovery system, a device may be described with attributes such as `Type`, `Location`, and `Model`, etc. In multimedia applications, such as the P2P music file sharing system described in [10], to enable content-based search, attributes include not only manually configured ones such as `Artist` and `Song Name`, but also features extracted from the audio signals, such as `Tempo` and `Strength`.

We refer to the collection of the AV-pairs as the "content name", or "content description". In our terminology, "content discovery" means the discovery of the "content name", not the actual content. We consider mechanisms such as contacting the device or retrieving the actual file after the "content discovery" step as a separate function. An AV-pair takes 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. A content name that consists of n AV-pairs is represented as $CN : \{a_1v_1, a_2v_2, ..., a_nv_n\}$. Languages such as XML may be used to describe content names. Figure 2 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
```

Fig. 2. An example content name.

Content names may consist of both *orthogonal* attributes and *dependent* attributes. Orthogonal attributes exist independently of each other, whereas a dependent attribute relies on the presence of some other attribute. For instance, the `Exit Number` attribute is meaningful only when the name also contains the `Highway Number` attribute. An attribute may be dynamic, e.g., the `Speed Measured` attribute, in that it may take on different values at different times. When the value changes, the content name changes.

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. In our current system, we consider equality matching only. Content names registered in the CDS are searched via *subset matching*. More specifically, a content name matches a query as long as the set of AV-pairs in the query is a subset of the set of AV-pairs in the content name. The AV-pairs in the query that are not in the content name are treated as "don't care". The number of non-empty subsets of a content name that consists of n AV-pairs is $2^n - 1$, which means it can match $2^n - 1$ different queries.

B. DHT-based Overlay Substrate

The CDS system uses the DHT layer [11] for two purposes: (1) constructing and managing the overlay network, and (2) delivering messages within the overlay network.

In a DHT, each node is assigned a node ID as its overlay network address, and it is responsible for a contiguous region in an m-bit address space. The nodeID may be obtained locally, e.g., by applying a system-wide hash function to some local information such as the node's IP address. Overlay networks built using DHT are structured in that node IDs encode overlay network topological information: The node ID determines the set of nodes that this node will be neighboring with, and which next hop node to use when forwarding a message in the overlay network. DHT-based systems are scalable by keeping both the number of routing table entries on a node and the number of overlay hops between any two nodes small, e.g., both are $O(\log N_c)$ in Chord[1], where N_c is the number of nodes in the network.

The CDS system uses the DHT layer to forward its messages within the overlay network. Communication is based on node IDs. When the DHT layer receives a tuple {`nodeID, message`} from the CDS layer, it will subsequently forward `message` to the node that corresponds to `nodeID`. The DHT layer does not dictate how CDS chooses the `nodeID` for `message`.

C. CDS Functionality

The API that the CDS layer provides to the application layer must include at least the following two methods: `register(content_name)` and `locate_contents(query)`. Once it receives a data item, a content name or a query, from the application layer, the CDS must determine the set of nodes it should send the data item to. In our architecture, this translates to computing a set of node IDs.

In choosing the set of nodes, the primary goal is to meet the scalability and content searchability requirements. In a centralized system, names and queries are sent to one central location, which constitutes the system's single point-of-failure and bottleneck. Approaches based on flooding the CDS network with registrations or queries are not scalable due to the prohibitive number of duplicated registration or query messages. In our system, we introduce an approach based on *Rendezvous Points (RPs)*. In this scheme, a content name is registered only with a small set of nodes in the system, the RPs; thus the full duplication of content names 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 names meet.

III. Basic CDS Design

We now present the basic RP-based CDS design.

A. Registration with RP Set

To register a content name, the provider node must first determine the set of nodes that should receive this name. It does this by applying a system-wide hash function, $\mathcal{H}$, to each AV-pair in the content name. For example, given content name $CN_1 : \{a_1v_1, a_2v_2, ..., a_nv_n\}$, which has n AV-pairs, the provider computes the following:

$$\mathcal{H}(a_iv_i) = N_i, i = 1..n.$$

The node whose ID is either equal to or numerically closest to N_i will become the ith RP node. These nodes (n of them assuming no hash collision) form the RP set for this content name. The *complete* 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 them. For a dependent AV-pair, we apply the hash function to it and all of its parent AV-pairs together. In this paper, we focus on orthogonal AV-pairs; the same mechanisms can be directly applied to dependent AV-pairs.

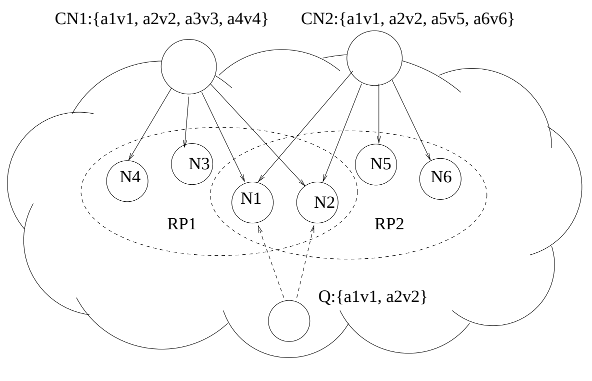


Fig. 3. Example registration and query processing with RP set.

Hashing each AV-pair individually has the following properties. First, it yields an RP set of size n for a name that has n AV-pairs, thus requiring $O(n)$ messages per registration. In real-world applications, n is typically a small number (e.g., < 50), and registration can be 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 alone provides a natural way of spreading registrations to more nodes in the system.

An RP node stores the names it received in a local database, and maintains them in a soft state fashion. As such, names automatically expire after a certain time period, and must be periodically refreshed. This provides protection against certain types of failures. For example, when an RP node leaves or crashes, the refresh messages will automatically restore a lost content name at a node that is alive. Also, when a name contains dynamic attributes, the refresh messages may register the name at a different set of RPs when their values change.

B. Query Resolution

To resolve queries, clients must determine the set of RPs that may contain matching content names. Since all content names that contain the pair $\{a_iv_i\}$ are stored in the node $N_i(= \mathcal{H}(a_iv_i))$, 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). Given these m candidate RP nodes, the client may pick one node randomly and send one query message to that node. Once an RP node receives a query, it simply goes through its name database, and determines the set of names that match the query by comparing each name's AV-pair list with that of the query's. No communication between nodes is needed, and query resolution is done efficiently.

An alternative to having queries fully resolved at one RP node is to have a client send its query to multiple 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 lightweight operation, it is more efficient to do the complete matching on the selected RP node.

C. Load Balancing Property

In the basic RP-based design, AV-pairs are used as the argument by the hash function and are mapped onto nodes. However, the registration and query loads observed on each node are determined by the AV-pair distributions in content names and queries. The basic design performs well when the distributions are even, in which case the distribution of load in the system should be even.

In real-world applications, 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 strings[1] in both traditional web searches [12] and Gnutella peer-to-peer networks [13] follows a Zipf-like distribution. This type of skewed distribution implies that some nodes in the CDS system may be overloaded while others are underutilized. More specifically, consider the case where the number of names that contain $\{a_i v_i\}$, $N_{a_i v_i}$, follows a Zipf distribution:

$$N_{a_i v_i} = N_s \cdot k \cdot \frac{1}{i^\alpha}, \tag{1}$$

for $i = 1...N_d$, where N_d is the number of different AV-pairs in the system. k and α are two parameters, where α is close to 1. N_s is the total number of names in the system and i is the rank of AV-pair $\{a_i v_i\}$ 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. As an example, suppose an application has $N_s = 10^5$ names, and $k = 0.5, \alpha = 1$. Half of the 10^5 names would contain the most popular AV-pair, which would be sent to one node. In the meantime, 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 would be swamped by registrations, while the majority of the nodes in the system would be rarely used.

IV. System with Load Balancing

We next present a distributed load balancing solution that allows the CDS to dynamically discover and utilize lightly loaded nodes to share the registration and query load on heavily loaded nodes.

A. Load Balancing Matrix (LBM)

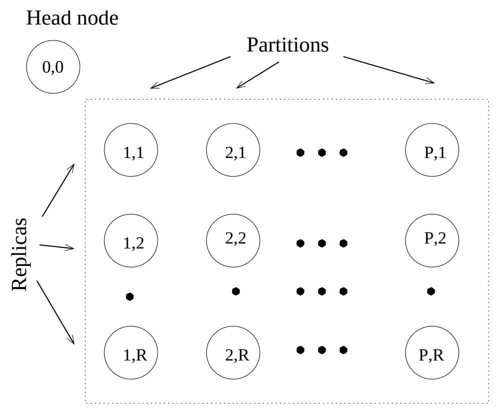


Fig. 4. Load balancing matrix for $\{a_i v_i\}$.

For a popular AV-pair, the CDS system uses a set of nodes instead of one node to share the registration and query load. This set of nodes is organized into a logical matrix, the Load Balancing Matrix (LBM). Figure 4 shows the layout of the matrix for AV-pair $\{a_i v_i\}$. Each node in the matrix has a column and row index, (p, r), and node IDs are determined by applying the hash function, $\mathcal{H}$, to the AV-pair, and the column and row indices together:

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

Each column in the matrix stores one subset, or *partition*, of the content names that contain $\{a_i v_i\}$. Nodes in the same column are *replicas* of each other: they host the same set of names.

The matrix dynamically expands or shrinks along its two dimensions depending on the load it receives. It starts with one node when the registration and query load are low; this corresponds to the basic system. New partitions are added to the matrix when the registration load of the pair $\{a_i v_i\}$ increases, and new replicas are added when the query load increases. Matrices may end up in different shapes. For example, a matrix may have only one row, when only the registration load is high, or one column, when only the query load is high. Each matrix uses a node, called the *head node*, with ID $N_i^{(0,0)} = \mathcal{H}(a_i v_i, 0, 0)$, to store its current size and to coordinate the expansion and shrinking of the matrix.

To expand matrices, each node in the system maintains 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. Three corresponding low thresholds are also set for shrinking purpose. Note that a node may belong to multiple matrices when multiple AV-pairs are mapped onto it, and the thresholds are used to regulate the aggregated load from all of these pairs. In the following discussions, for simplicity, we assume all nodes are homogeneous in that they have the same computation power and network connectivity.

B. Operations With LBM

We first describe the registration and query operations when LBMs are present in the system.

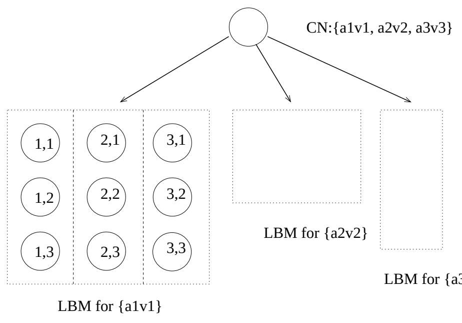


Fig. 5. Registration with load balancing matrices.

1) Registration: In the basic system, a content provider registers its content name with each RP node that corresponds to one of the AV-pair in the name. In contrast, with LBMs, the provider must register its content name with one column of nodes in each matrix that corresponds to an AV-pair (Figure 5).

The pseudo code for registration is listed in Figure 6. To register with matrix LBM_i, the content provider must first discover its size: the number of partitions, P, and the number

[1]Keyword-based search is a special case of AV-pair based search where attributes are omitted.

```
1: register(name) {
2:     foreach AVpair $a_i v_i$ in name {
3:         $N_i^{(0,0)} \leftarrow \mathcal{H}(a_i v_i, 0, 0)$;
4:         (P, R) $\leftarrow$ retrieve_matrix_size($N_i^{(0,0)}, a_i v_i$);
5:         p $\leftarrow$ generate_random_number(1, P);
6:         foreach r in [1, R] {
7:             $N_i^{(p,r)} \leftarrow \mathcal{H}(a_i v_i, p, r)$;
8:             send_to($N_i^{(p,r)}$, name);
9:         }
10:    }
11:}
```

Fig. 6. The algorithm for content providers to register with LBM.

of replicas, R. It can do so in several ways. First, it may be able to retrieve the size from the pair's corresponding head node (Line 4 in Figure 6). Second, in case the head node is down or becomes a bottleneck, the provider may find out the matrix size by directly sending probe messages to nodes that are potentially 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 checking its database to see if it has seen $\{a_i v_i\}$ before. Since partitions are indexed contiguously, the current number of partitions can be efficiently discovered in $O(\log P_0)$ steps via binary probing between partition 1 and P_0. Third, content providers may cache an AV-pair's matrix size and use it without re-discovering it. This is useful when refreshing a previously registered name.

Once the size of the matrix is found, the content provider selects a *random* partition between 1 and P and computes the node IDs in the partition. It then registers with each of them. Since the partition within the matrix is randomly selected, the registration load within the matrix is distributed evenly.

2) Query Resolution: Similar to the basic system, clients can issue a query to the matrix that corresponds to any AV-pair in the query. The cost of resolving a query is determined by the number of partitions in the selected matrix. If the query contains only one AV-pair, it would be sent to the matrix corresponding to that pair. When this matrix has a lot of partitions, the client can contact a small subset of the partitions to receive enough matches. The client may then refine its query by adding more AV-pairs. In fact, this is the behavior of Internet users when using a search service. A study conducted in [12] shows that 71.5% of the searches found in one large web cache contains more than two keywords.

In our system, when multiple AV-pairs are present, we use a two-pass *query optimization* algorithm to determine which pair a client should use for its query. First, the client probes the sizes of all the matrices corresponding to each AV-pair in the query using one of the mechanisms presented above, and then selects the one with the fewest partitions. In practice, since the matrix sizes can be cached, the cost of the probing phase is amortized when the client issues many queries.

Once a matrix is selected, the client must send the query to all the partitions in the matrix, if it needs to collect all possible matches. In reality, sending to a subset of the partitions may return the client sufficient number of results. Since nodes in the same column are replicas of each other, the query needs only to be sent to one node in each column, and the client chooses a *random* node to ensure the query load is distributed evenly within a matrix.

C. Matrix Management

In this section, we present the matrix expansion and shrinking mechanisms. When a matrix receives high load, it must expand itself quickly to accommodate the excessive load. When the load decreases, the matrix should shrink itself to reduce registration and query cost. We use a multiplicative approach to expand the matrix by doubling the number of partitions or replicas when the existing matrix is saturated by registration or query load. To ensure a stable system, matrix shrinking is done linearly, i.e., we decrease the number of partitions or replicas by one at a time.

We describe the detailed expansion and shrinking mechanisms using matrix LBM_i as an example. LBM_i corresponds to $\{a_i v_i\}$, and its head node is $N_i^{(0,0)}$. Suppose there are currently P_i partitions and R_i replicas in LBM_i.

1) Partition Expansion: New partitions are added to LBM_i when the existing partitions in the matrix receive high registration load. We define the *expansion region (ER)* as the set of partitions that are last added to the matrix. The expansion mechanism works as follows.

- When the registration load on a node in the matrix reaches the threshold T_{CN} or T_{reg}, it will send an `INC_P` request to the head node, $N_i^{(0,0)}$.
- The head node doubles the number of partitions to $2P_i$, upon receiving the first such request from a node in the current ER. It ignores requests from non-ER partitions and from other ER partitions that may arrive later.
- The head node then informs nodes with partition index from $P_i + 1$ to $2P_i$ that they are now in the matrix and will be responsible for $\{a_i v_i\}$. Partitions $P_i + 1$ to $2P_i$ become the new ER.

When a new content name that contains $\{a_i v_i\}$ comes up, the registering node will discover LBM_i has $2P_i$ partitions, and then select one to register this name. Hence, the registration load is shared by the expanded matrix.

The head node acts upon only one request from the ER and suppresses others to avoid unnecessary expansion. The reason is that the head node expands the matrix only when the following two conditions are met: (1) the load are distributed evenly among all partitions; and (2) the load on any partition reaches threshold. In the algorithm above, we use a request from an ER partition as the signal of when the above conditions are satisfied, since the load observed on the non-ER partitions do not reflect the average when the new partitions are being added.

The multiplicative increase of P_i allows the number of partitions grow quickly, and as such the system can quickly tune itself to accommodate high load. The direct cost of this approach is minimal, since "adding a partition" does not actually involve any copying of data over the network.

2) Partition Shrinking: A matrix decreases the number of partitions when possible, since more partitions means more query messages are needed for a query that is sent to this matrix. Suppose LBM_i now has P_i' partitions, and before the last expansion, it has P_i partitions. P_i is also equal to the number of partitions in the ER immediately after the last expansion.

- When any node in the *last partition* of the matrix observes a low registration rate or a low number of content names, it will issue a `DEC_P` request to the head node.
- The head node again acts on only one such request: it sends a `SHRINK_P` command to all the nodes in the last partition and asks them to transfer their names containing $\{a_iv_i\}$ to the nodes in partition $P_i' - P_i$. For example, $N_i^{(P_i',1)}$ sends its names to $N_i^{(P_i'-P_i,1)}$.
- After all the transferring are confirmed successful, the head node will inform nodes in partition P_i' that they are removed from the matrix. Now partition $P_i' - 1$ becomes the last partition, and the head node decreases the size by 1, $P_i' \leftarrow P_i' - 1$. Correspondingly, the size of the current ER is also reduced by 1.

When all the partitions in the current ER are removed, and the number of partitions drop back to P_i, the head node informs the partitions from $\lceil \frac{P_i}{2} \rceil$ through P_i to become the new ER. By collapsing the matrix one partition at a time, we try to keep the matrix load balanced, and the linear decrease prevents the matrix from oscillating.

3) Replication Expansion: New replicas are added to the matrix when the query load to the matrix increases, similar to how partitions are added. The expansion region here refers to the replicas that are last added. When a node in the ER observes its query rate reaches T_q, it will send an `INC_R` request to the head node. Upon receiving such a message, the head node issues a `DUPLICATE` command to each node in the last row of the ER, asking them to replicate themselves.

The replication is also done multiplicatively to allow the matrix expand to a large size to accommodate query load. A node that receives the `DUPLICATE` message sends a copy of the names corresponding to $\{a_iv_i\}$ in its database to the newly added nodes in its column. For example, node $N_i^{(1,R_i)}$ will send its names to nodes $N_i^{(1,R_i+1)}$ through $N_i^{(1,2R_i)}$. The head node doubles R_i when all the replicas are in place, and the nodes in row $R_i + 1$ to row $2R_i$ become the new ER.

4) Replication shrinking: More replicas in the matrix means providers must register with more nodes. Thus matrices should shrink along the R dimension when the query load to this matrix drops. The replication shrinking mechanism is similar to the partition shrinking mechanism, but no data transfer is needed. When the head node receives a `DEC_R` request, it informs all the nodes in the last row to remove themselves from the matrix. The head node then decreases the number of replicas by 1.

The shrinking mechanism is important specially under "flash-crowd" type of load: when an AV-pair becomes popular due to for example a current event, its corresponding matrix will replicate quickly to accommodate the sudden surge of load. When clients lose interest in this pair, the matrix will shrink and eventually may become just one row.

5) Head Node Mechanisms: The primary job of the head node is to coordinate the matrix expansion and shrinking. The expansion and shrinking requests may come to the head node in an arbitrary order. While a matrix is in a dynamic state, i.e., expanding or shrinking, if the corresponding head node receives additional requests, it will buffer these requests and process them when the current operation completes. By serializing the operations, we ensure data consistency within the matrix.

A head node is only responsible for its own matrix, and different matrices will likely have different head nodes, which are distributed across the network. Therefore head nodes will not become the bottleneck of the system. However, when a head node leaves or crashes, vital information about its matrix, such as the size, will be lost. To prevent this from happening, live nodes in the matrix send infrequent messages with their indices (p, r) to the head node. Due to the routing properties of DHT, a new node whose ID is close to the old head node's ID will receive these messages and become the new head node. It can then recover the matrix's size based on the information it receives. In fact, the matrix expansion or shrinking requests will also reach the new head node, and they can be used to recover the matrix's dimensions as well.

D. System Properties With LBM

When LBMs are deployed in the system, both the registration and query cost are higher than in the basic system. To register a name that has n pairs, the number of registration messages needed is:

$$M_r = \sum_{i=1}^{n} R_i,$$

where R_i is the number of replicas in matrix LBM_i. $R_i = 1$ when LBM_i has no extra replicas. M_r is determined by the number of replicas each matrix has, and does not depend on the number of partitions.

To resolve a query that has m pairs, when using the query optimization mechanism, the number of query messages needed excluding the probing messages, is

$$M_q = min(P_i),$$

where $i = 1..m$, and P_i is the number of partitions in matrix LBM_i. The query cost is not affected by the number of replicas in these matrices, but depends solely on the number of partitions.

The benefits of the query optimization algorithm are two-fold. First, it will likely keep the number of query messages low by avoiding matrices that have large P. Second, by reducing the number of query messages, it also indirectly reduces the number of registration messages required in the system. By avoiding matrices that have large P, the query load on these matrices is reduced; thus it naturally limits the R-dimension expansion of these matrices. A matrix with a smaller R means that names mapped onto this matrix require fewer registration messages. Conversely, for matrices that have small P, even if they have a large R due to many queries, from the system's point of view, it will not greatly affect the average number of registration messages needed: small P implies only a small number of content providers will be affected by having to send more messages.

V. Evaluation Methodology

We describe our simulator implementation and the evaluation methodology.

A. Simulator Implementation

We developed an event-driven simulator to evaluate the CDS system. Each node uses a first-come-first-serve queue to process registrations and queries with exponentially distributed service rates. A node measures the registration and query rates it observes using a sliding window of a certain number of recently received registrations or queries. In the simulation, we use a window size of 20. Matrix size discovery is done by probing head nodes.

The simulator assumes the existence of a DHT-based overlay mechanism for routing and forwarding and it uses a 24-bit name space for node IDs and the hashed values of AV-pairs. 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. In practice, this can be achieved by using techniques such as assigning multiple "virtual" node IDs to one node [1]. The hash function used by the CDS system must generate values uniformly distributed in the name space and be insensitive to the input. In our implementation, we use the cryptographic function SHA-1 as the system-wide hash function.

The simulator uses an exponential distribution with a mean value of 50 ms [1] to model the one-way network delay between any two nodes. In DHT systems such as Pastry [3], by employing proximity metric into the routing rules, the overlay delay between two nodes can be limited to within 1.4 times of the physical network delay. In our simulation, we conservatively set the average overlay delay between two overlay nodes to be twice of the physical network delay between them, which results in a mean of 100 ms.

B. Experiment Setup

In the experiments we conducted, we assume that each node has approximately $500Kbps$ available link bandwidth (DSL level) dedicated to content name registrations and queries. Corresponding to this bandwidth, assuming a 1000-byte registration packets size and a 250-byte query packet size, each node sets up a threshold of $T_{reg} = 50reg/sec$ as the maximum sustainable registration rate and $T_q = 200q/sec$ as the maximum sustainable query rate. T_{CN} is set to be 4000. When a node observes that one of these thresholds is reached, it will issue a matrix expansion request to the corresponding head node. In our experiments, T_{reg} is always reached before T_{CN}. To enable us to study the effectiveness of the load balancing mechanism, the maximum number of partitions and replicas a matrix can use are configurable in the simulator. The matrix will stop expanding along a dimension if that dimension reaches its maximum value. In our experiments, we focus on how the system behaves when load increases. The load distribution does not change within each simulation run, and the matrix shrinking mechanism is not triggered.

The processing of registrations and queries on a node is exponentially distributed with a mean rate of $1000reg(query)/sec$, which can easily be achieved by modern PCs on a database with a size on the order of 10^5 entries. With these assumptions, a node's performance is limited by its available link bandwidth.

To register a name, registration messages are sent to the RP nodes corresponding to the name's AV-pairs concurrently. Upon receiving a registration, the RP node either inserts the name into its local database and replies the registering node with a success, or rejects the name and replies with a failure. A registration may fail at a node for two reasons: (1) the registration rate this node observes, r_{node}, exceeds the set threshold, i.e., $r_{node} > T_{reg}$, or the number of names it is hosting exceeds T_{CN}; (2) the corresponding matrix is in a dynamic state such as expanding. For instance, a node has sent a replica to a new node, but the success of the replication has not been confirmed, and during this time period, any registrations arrive at the replicating node will be rejected and result in a failure. The registration succeeds when *all* the pairs registered successfully.

Similarly, a query is sent to one RP node in each partition of the chosen LBM concurrently. The RP node rejects the query if the query rate this node observes, q_{node}, exceeds the set query rate threshold, i.e., $q_{node} > T_q$, by replying to the query node with a failure message. Otherwise, it accepts the query, examines its database and sends the querying node the set of content names that match the query. Note that the set may be empty. From the querying node's point of view, a query succeeds when *all* the corresponding RP nodes accept the query.

We evaluate the CDS system using the following metrics: the *registration/query success rate* and the *registration/query response time*. The success rate is defined as the percentage of successful registrations or queries in one simulation run. Since the *system throughput* equals to the product of the *system load* (registration/query rate) and the *success rate*, the success rate is used as an indicator of the system's throughput: the throughput increases as load increases, if the success rate remains high. For a successful registration or query, we define the response time as the time between when the last reply message is received and when the registration or query messages (probe messages, when we must probe the matrix size) are first sent.

C. Workload

In the following experiments, we consider a CDS network that has 10,000 (N_c) nodes. There exists 50 attributes in the system, each of which can take on 200 values; this results in 10,000 (N_d) distinct AV-pairs. On average, each node is responsible for 1 (= N_d/N_c) AV-pair.

We generate two sets of content names for registration and one set of queries as workload to drive the simulations.

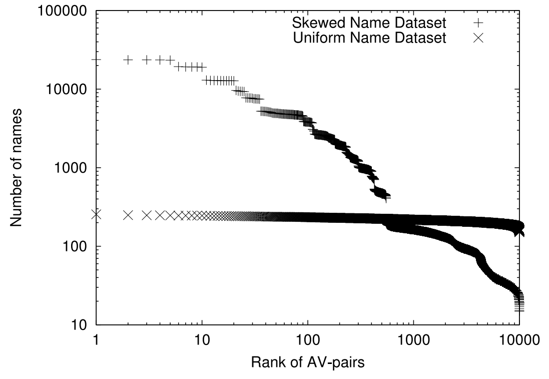


Fig. 7. AV-pair distribution in two sets of content names.

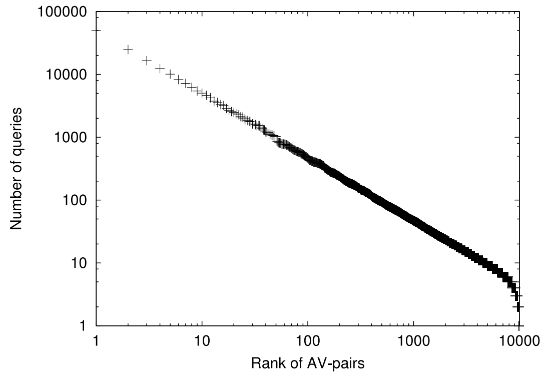


Fig. 8. AV-pair distribution in queries.

Each name dataset contains $100,000$ names and each name is comprised of $n = 20$ AV-pairs. The AV-pair distributions in names are shown in Figure 7. In the uniform dataset, each AV-pair is equally likely to appear in a name, and on average each AV-pair occurs in about 200 names. The uniform dataset is primarily used for comparison. 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($\alpha \sim 0.88$). The top 5 most popular AV-pairs are contained in about 24,000 names. The query dataset (Figure 8) contains $99,473$ queries and is generated based on a Zipf distribution with $k = 0.5$ and $\alpha = 1$ in Equation 1. The number of AV-pairs in a query ranges from 1 to 10, and on average each query consists of 4 AV-pairs. The most popular AV-pair occurs in about 50,000 queries. The sender of a name or a query is selected randomly from the nodes in the system and both the arrival times for names and queries are modeled with a Poisson distribution.

VI. Simulation Results

We conducted extensive simulations to evaluate the properties of the CDS system. We show the system's performance with regard to registration load in Sections VI-A, VI-B and query load in Section VI-C. We analyze the load balancing behavior in Section VI-D. Finally, we study the cost introduced by LBMs in Section VI-E.

A. Registration Success Rate

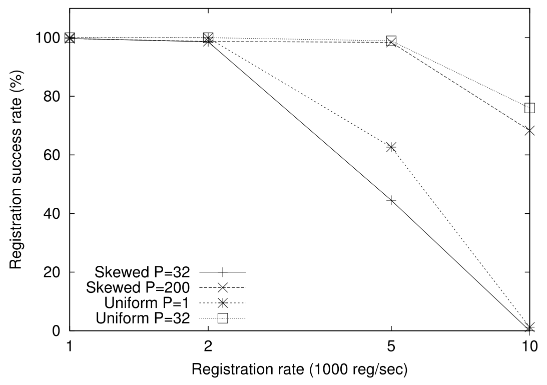


Fig. 9. Registration success rate comparison.

We first examine how the system behaves as the registration rate increases. For each experiment, we inject either the skewed dataset or the uniform dataset into the system with a certain arrival rate r_{system}. Each experiment is carried out with a different configuration of P value, the maximum number of partitions a matrix may use.

Figure 9 compares the success rate in these experiments after all the matrices stop expanding. We observe that for a given P value, when the registration rate is low, the registrations succeed on the existing set of partitions, and the success rate is 100%. As load increases, the success rate starts to drop, because without further expanding, nodes in the matrices become saturated and start to reject registrations. By increasing P, for the same registration load, the success rate is improved significantly. As load is further increased, all curves eventually drop.

For the uniform dataset, since AV-pairs are distributed evenly in content names, registration load is distributed fairly evenly among nodes in the system. Compared with the skewed load, to maintain the same success rate, fewer partitions are needed for the same registration load. The basic system ($P = 1$) performs well until r_{system} reaches $2000 reg/sec$, and after that the success rate drops quickly to near 0%. The reason is that the hash function may map multiple AV-pairs onto the same node, and when r_{system} increases, registration rate on such nodes will reach T_{reg} earlier than others, and cause registration failures.

We study the data points corresponding to the highest registration load, where $r_{system} = 10^4 reg/sec$. In these experiments, since there are no queries, and thus no replications in the system, each name is registered at $n = 20$ nodes, the average registration rate observed on a node is:

$$r_{node} = \frac{r_{system} \cdot n}{N_c} = 20 reg/sec.$$

The success rate under this registration load is 76% for the uniform load with $P = 32$, and 68% for the skewed load with $P = 200$. What it means is that on average each node in the system operates at 40% of its link capacity while maintaining a fairly high success rate. These experiments show that the system can be scaled to near its capacity even for skewed load: the load balancing mechanism effectively spreads the excessive load to underutilized nodes in the system.

B. Effectiveness of Partitions

To better understand the effectiveness of adding partitions on improving the registration success rate, we conducted another series of experiments. We inject the skewed content name dataset into the system with a fixed arrival rate of $r_{system} = 5000reg/sec$. We vary the configured P value in each experiment, and for each P we run the experiment twice: (1) during the *initial run*, as names arrive, partitions are created when needed, and (2) the same dataset is sent to the system again in the *stable state*, when all the partitions have been created, and no new partitions are added.

The number of partitions needed, P_i, for a pair $\{a_i v_i\}$ can be analytically computed as follows:

$$P_i = \frac{r_{a_i v_i}}{T_{reg}} = \frac{r_{system} \cdot p_i}{T_{reg}} \quad (2)$$

where $r_{a_i v_i}$ is the arrival rate of $\{a_i v_i\}$, and p_i is the pair's probability of occurring in names. In the skewed name dataset, for the top 5 most popular pairs, $p_i = 0.24$. With $r_{system} = 5000reg/sec$ and $T_{reg} = 50reg/sec$, from Equation 2, we know to accommodate names that contain these pairs, each of the corresponding matrices needs at least $P = 24$ partitions.

Figure 10 shows the success rate of registrations, under different P value. To interpret the figure, we first classify registration failures into four types:

1) *capacity failure*. Failures due to not having enough partitions allocated to a matrix to accommodate a pair's registration load.
2) *compulsory failure*. In the simulation, it takes one RTT to add new partitions to a matrix, and registrations arriving during that time period are rejected.
3) *conflict failure*. Since multiple AV-pairs may be mapped onto one node, a registration may fail at a node because some other pair introduces high registration load there.
4) *statistical failure*. Failures due to statistical variations, e.g., failures caused by bursty arrival of registrations of the same pair on one node.

In Figure 10, when $P \leq 20$, the success rate is very low primarily due to the large number of capacity and conflict failures caused by the popular pairs. In particular, $P = 1$ corresponds to our basic system, and the poor performance shows that using one RP node for each AV-pair can not handle highly skewed load.

When $P = 32$, the success rate is still below 50% though seemingly there should be enough partitions. The failures come mainly from conflicts: since we have 10,000 distinct AV-pairs and 10,000 nodes, it is possible that two AV-pairs are mapped onto the same node. As the system allows more partitions to be used by a matrix, the conflict failures are overcome and the success rate increases significantly. The reason is that when a node observes high registration load caused by two different pairs, it will prompt the expansion of both of their corresponding matrices (at different times), thus reducing the load observed by partitions within each of the two matrices. The gap between the initial curve and the stable curve represents the percentage of compulsory failures. When enough partitions are allowed, the success rates in the stable run are substantially higher than those in the initial run since there are no compulsory failures. We observe that the success rate stays above 95% for $P > 100$ under this load.

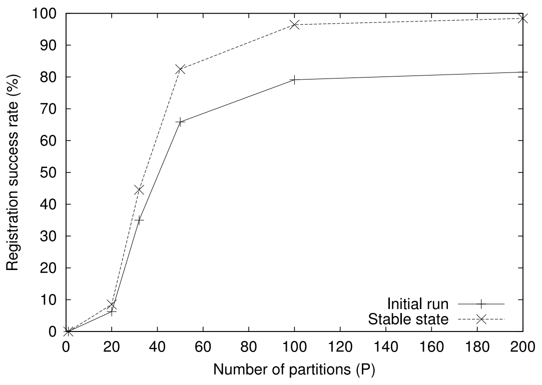


Fig. 10. Effect of number of partitions. Skewed dataset with $r_{system} = 5000reg/sec$.

In summary, by allowing the matrix to expand along the P dimension, the system can successfully recruit lightly loaded nodes to share concentrated registration load, thus increasing the system success rate.

C. Query Success Rate

In this section, we study how the system scales as query load increases. In the following experiments, we first inject into the system the skewed name dataset with $r_{system} = 2000reg/sec$, and then issue the Zipf queries with different arrival rate q_{system}. We run the simulation under two schemes: (1) random, where a query is sent to a matrix that corresponds to a random pair in the query, and (2) using query optimization. In these experiments, a matrix may replicate as many times as necessary.

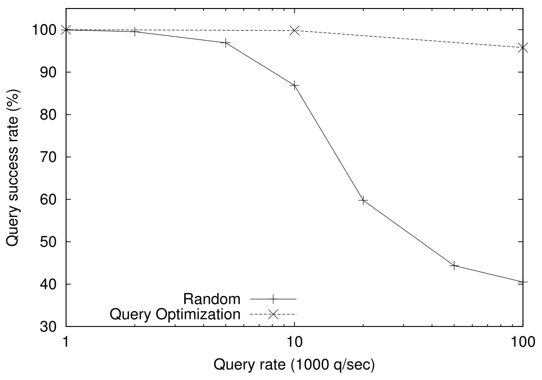


Fig. 11. Query success rate comparison.

Figure 11 shows the query success rates. In the random scheme, by selecting a random pair for each query, the system tries to spread load to different matrices, and the success rate stays above 90% for rates as high as $5000q/sec$. However, when q_{system} is further increased, the success rate starts to drop sharply. The reason is that since popular AV-pairs appear in many queries, and each query contains only a few pairs, it is possible that many queries select the same AV-pair and are sent to the same matrix, which will cause compulsory failures and the replication of these matrices.

In addition, in the workload, the pairs that are popular in queries are also common in registrations, which means their corresponding matrices have many partitions. The time it takes to replicate a large matrix is high, since the new replicas can be used only after all the partitions replicate successfully. Queries arriving during the replication time period are likely to be rejected, since they must be sent to the existing replicas, which have already being saturated. This phenomenon is displayed most clearly when the arrival rate is extremely high ($q_{system} > 50,000q/sec$); in this scenario, all the queries arrive before a matrix can complete two rounds of replications.

On the other hand, the query optimization mechanism successfully spreads query load to matrices with few partitions. This is specially important for high query load, where using the load balancing mechanism alone is not effective. Figure 11 shows that even under the highest load, $q_{system} = 10^5q/sec$, with query optimization, by avoiding large matrices and thus long replication time, the system's query success rate remains above 95%. Most matrices do not need to replicate at all, and the largest matrix replicated twice ($R = 4$).

D. Load Distribution

In this section, we evaluate the system's load balancing property by examining the name distribution and observed load on nodes. We report results corresponding to registration load; the results for query load are similar.

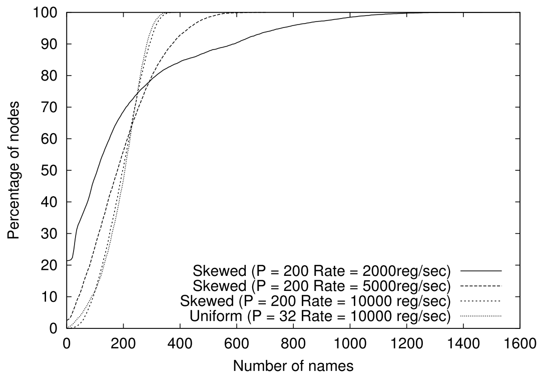


Fig. 12. Comparison of the Cumulative Distribution Function of the number of registered names on nodes.

1) Content Name Distribution: Figure 12 shows the Cumulative Distribution Function (CDF) of the final number of names on nodes under four scenarios, corresponding to four experiments in Figure 9. The first three curves correspond to the experiments where the skewed dataset is used with $P = 200$, and various registration rates. The fourth curve is from the uniform dataset with $P = 32$. Since the success rates are different in these experiments, for comparison purpose, we normalize the number of successfully registered names in each experiment to 10^5. Thus on average each node should receive 200 names.

In our experiments, since T_{reg} is always reached before $T_{CN}(= 4000)$, matrix expansions are therefore caused by the high registration rates observed on nodes, and not the high number of names. At the end of each experiment, the average registration rate on each node can be simply computed by dividing the final number of names it has by the simulation time. Thus we use the number of names to represent the registration load on a node.

With low registration rate, the system can accommodate the registrations successfully using a small number of partitions for each matrix, which means many nodes in the system may receive none or a small number of names. For example, when $r_{system} = 2000reg/sec$, 21% of nodes receive no registrations. In the mean time, some nodes in the system accumulate large number of names, as exhibited by the long tail in the distribution. Note the maximum number of names on a node is still less than T_{CN}. As registration rate increases, names are spread to more nodes, due to the expansion of matrices. In Figure 12, when $r_{system} = 10^4reg/sec$, we observe that the CDF grows very quickly and no nodes receive more than twice of the average number of names. A distribution that is "more vertical" represents a more load balanced system.

More quantitatively, we use the metric *Coefficient of Variance (CV)* [14] to evaluate the load balancing property. In our context, CV is defined as:

$$CV\left[n_i\right] = \frac{\sigma\left[n_i\right]}{E\left[n_i\right]}$$

where $i = 1..N_c$, n_i is the number of names node N_i holds, and σ and E are the standard deviation and mean of n_i. A smaller CV indicates a more load balanced system. As load increases, the load balancing mechanism successfully balances load across all nodes across the system. The CV decreases from 1.242 to 0.369 as r_{system} increases from $2000reg/sec$ to $10^4reg/sec$. As a reference, when $r_{system} = 10^4reg/sec$, the CV in the skewed load case matches the $CV(= 0.366)$ in the uniform load case.

2) Observed Load on RP Nodes: We now take a closer look at the load distribution within different partitions of a matrix. Figure 13 shows the observed registration rate as time progresses at three different partitions of a matrix that corresponds to one of the most popular AV-pairs. In this experiment, the skewed name set with $r_{system} = 2000reg/sec$ is used. Initially there is only one partition in the system, and it receives the entire registration load corresponding to this pair. The maximum observed registration rate approaches $450reg/sec$. As partitions are added to the matrix to share the registration load, the rate observed by the first partition begins to drop quickly, as shown in the figure. The 16th and 32nd partitions are introduced around time 2000 ms and 3700 ms respectively. Once all the partitions are in place, as expected, the load on each partition stays under the set threshold of $T_{reg} = 50reg/sec$. In fact, since the load is shared by 32 partitions, each node observes about $15reg/sec$.

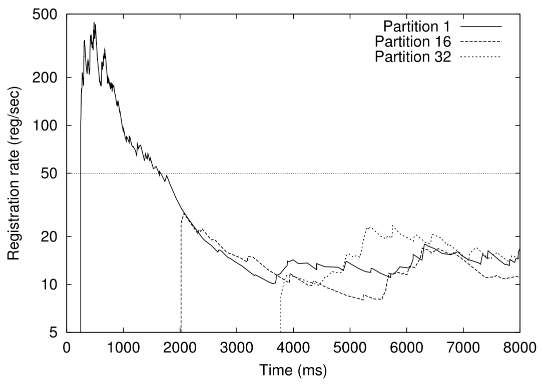


Fig. 13. Load within a matrix. Skewed dataset with $r_{system} = 2000reg/sec$.

E. Registration and Query Cost

In this section, we evaluate the system from content providers and query issuers' point of view: we examine the response times, and the number of messages needed for registrations and queries.

In this set of experiments, registrations and queries arrive simultaneously with the arrival rates of $r_{system} = 1000reg/sec$ and $q_{system} = 5000q/sec$. The workload consists of about 17,000 skewed names and 83,000 Zipf queries. Instead of devoting its full bandwidth to serve either registrations or queries, each node allocates 50% of the bandwidth to queries and 50% to registrations. Correspondingly, the thresholds are set as follows: $T_{reg} = 25reg/sec$ and $T_q = 100q/sec$. We again run the simulation under two schemes: random and with query optimization.

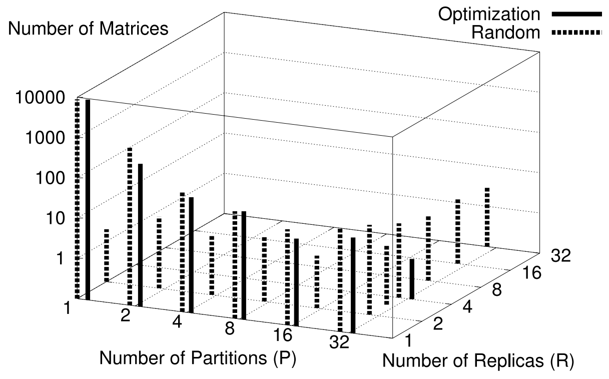


Fig. 14. Matrix size distribution. All the axes are in logarithmic scale.

1) Matrix Size Distribution: As discussed in Section IV-D, the sizes of the load balancing matrices affect the cost of registrations and queries. After each simulation run, we tally the sizes of all the matrices. Figure 14 is a 3-D presentation of the distribution. Each bar corresponds to the number of matrices that have that particular size (P, R). Since there are 10,000 distinct AV-pairs in the system, there are 10,000 LBMs in total. All the results fall on the vertical planes that correspond to powers of two, because the dimensions are increased multiplicatively and there is no matrix shrinking in the experiments.

In the random scheme, 89.2% of the matrices have 1 partition and 1 replica, $(P = 1, R = 1)$. As we discussed earlier, matrices that have large P may still get many queries, which means they must replicate themselves frequently. Figure 14 confirms our analysis in that some matrices with large P, e.g., $P = 32$, also have a large R. The largest matrix has a size of $(P = 32, R = 32)$.

In contrast, with query optimization, more matrices (94.3%) have the minimal size, $(P = 1, R = 1)$. In all the matrices, the maximum number of replicas a matrix has is 4. It is worth noting that the matrix that has 4 replicas also has 32 partitions. The explanation is that the AV-pair corresponding to this matrix is also popular in queries. In particular, it appears frequently by itself in queries, which makes query optimization not applicable and replication necessary.

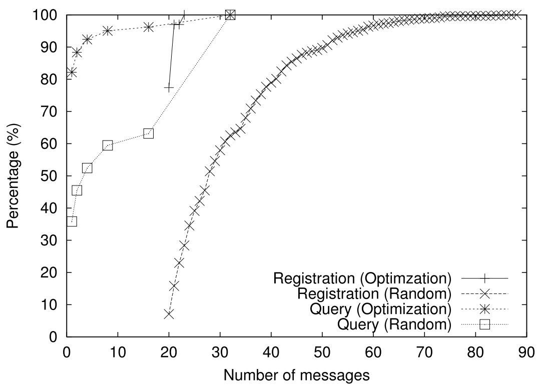


Fig. 15. CDF of registration and query messages.

2) Registration Cost: Figure 15 shows the CDFs of the number of registration and query messages under the two schemes.

With query optimization, 77% of the content names need to register with only 20 nodes because the corresponding matrices have only 1 replica. The maximum number of registration messages is 23, and the average is 20.3, i.e., a less than 1 message increase over the minimal registration requirement of the system. However, in the random case, 93% of the registrations need more than 20 messages, which means they involve at least one matrix that has multiple replicas, The average number of registration messages goes up to 32.3, and the maximum is 88.

The two curves on the right side in Figure 16 compare the registration response time of the two schemes. Sending more registration messages in the random scheme results in a longer response time: the average is 901 ms, whereas the average is 859 ms in the optimization scheme.

Note that the average response time is greater than two RTTs (400 ms), which is how long it would take to register one AV-

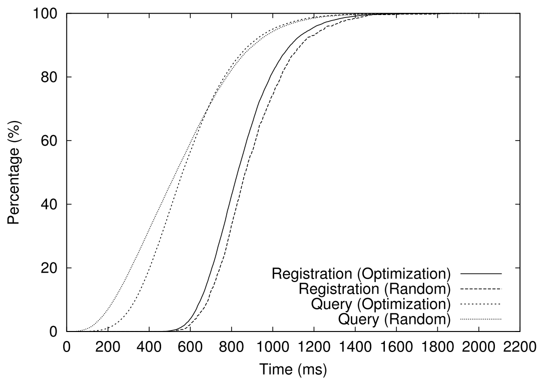


Fig. 16. CDF of registration and query response time.

pair (matrix size probing and the actual registration). The main reason is that the response time is computed only when all the 20 AV-pairs' registrations are confirmed. More formally, this is equivalent to sampling an exponential distribution 20 times and take the maximum value instead of the average value.

3) Query Cost: From query issuers' point of view, using query optimization, the average number of query messages (excluding the probing messages) is 2.7. This means on average a query is sent to matrices that have less than 3 partitions. In particular, from Figure 15, we observe that 82% of the queries are sent to matrices that have 1 partition, and only 3.7% of queries use the maximum number (32) of partitions. In comparison, in the random scheme, the average number of query messages is 13.8, only 36% of the queries are sent to matrices that have 1 partition, and 36.8% of queries are sent to matrices that have 32 partitions.

The cost of query optimization is the larger number of probing messages: instead of probing one matrix to get the size, the querying node has to probe all the matrices corresponding to the pairs in the query. This results in a slightly longer average response time for the query optimization scheme (597 ms vs. 594 ms). The two curves on the left in Figure 16 compare the CDF of the response times with and without query optimization. The CDF of the optimization scheme initially lies on the right side of the random scheme, but it has a steeper slope owing to a more uniform distribution. In practice, a query initiator can cache the size of different matrices to reduce the number of probing messages for its future queries.

In summary, by using query optimization, while the system is accommodating high skewed load, both registration and query costs are kept near the minimum cost as defined by the basic RP-based system with no partitions and replications.

VII. Related work

There exist many systems that can be considered as CDS systems, ranging from web search engines and directory services to peer-to-peer file sharing systems. We classify these systems based on how the content resolvers are organized and compare them with our system. Centralized systems, such as Napster[15] and Google[6], use a set of central servers to index contents and resolve queries. These servers may become the bottleneck as load increases, and form the single point-of-failure of the system, thus making it vulnerable to censors and attacks such as Denial-of-Service. Our CDS is distributed and uses a more robust overlay resolver network.

Content resolvers may be organized hierarchically into a tree structure, e.g., in DNS [5] and SDS [16]. In general, these systems are designed for hierarchical content names, such as domain names and directories [17]. To prevent overloading resolvers high in the tree, DNS relies on caching to scale to the Internet level and SDS uses bloom filter to reduce load propagated up the tree. In contrast, our system is designed to handle more general content names that do not necessarily have a hierarchical nature.

Systems based on an unstructured general resolver network such as INS [8], Siena [9], Gnutella[7], and Freenet [18] require flooding the network at either content registration time or query resolution time. Hence these systems do not scale with the number of content names and queries. More recent systems such as KaZaA [19] scale better by leveraging a two-tier infrastructure and relying on "supernodes" to suppress the flooding. In our system, we eliminate network-wide flooding at both registration and query time by establishing Rendezvous Points.

A hash-based peer-to-peer system such as Chord [1], CAN [2], Pastry [3], and Tapestry [4], uses a scalable protocol to form a self-organizing structured overlay network. While not directly supporting general content searchability, these systems provide an efficient solution to content name lookup by binding a complete content name, such as a file name, to a specific node in the system using a hash function. These systems relate to our work in two ways. First, the DHT abstraction [11] in these mechanisms provides the CDS system a scalable and robust substrate for building the CDS overlay network and for routing CDS messages. Second, our CDS system extends the basic lookup functionality and supports content searchability by using AV-pairs.

Several projects built systems on top of DHT to support searchability. In [12], the focus is on efficient keyword-based searching. Unlike our system, a query is sent to each node that is responsible for one of the keywords in the query, and partially matched results are first collected over the network and then "join" operations are performed to get the final matches. Techniques such as bloom filters and caching are used to reduce the network bandwidth consumption. We avoid the transmission of potentially large number of partially matched results by storing complete content names (all keywords of a document in [12]'s context) on RP nodes to allow full resolution locally.

Twine [20] is a resource discovery system built on top of Chord. Resource descriptions are separated into "strands" and then mapped onto nodes in the resolver overlay network, similar to our basic system. A resolver that corresponds to a random strand in the query is used to resolve the query. Twine simply rejects registrations that correspond to a popular strand once a threshold on the corresponding node is crossed. In our system, we show query optimization is important to this type of system's performance and we use load balancing matrices to deal with skewed load distribution.

Load balancing using partitions and replicas can trace its roots to early work in parallel databases, e.g., Gamma [21]. DDS [22] explores these ideas further in the domain of designing backend for Internet services in a server cluster setting. Upon receiving a request, the front end server selects a replica within a partition to best serve the request. Our system works in a peer-to-peer setting, and the selection of which node serves a request (query or registration) is done by the end points locally.

In the context of Content Distribution Networks (CDN), [23] proposes schemes where a request redirector can select a server replica from a dynamic list of servers to serve a URL request. The selection is based on the load of the servers, and the redirector may decide to grow the list of servers if the number of requests increases. This scheme is similar to one dimension of our load balancing mechanism, the replication expansion. However, in our system, the expansion is done in a distributed fashion by using high local query load to indicate the need of expansion, and no centralized entity like the redirector is needed. In addition, we also consider load balancing for registration.

VIII. CONCLUSIONS

In this paper, we presented a distributed and scalable approach to the content discovery problem. The RP-based content registration and discovery mechanism allows the CDS system to scale with the number of content names and queries by avoiding network-wide flooding. AV-pair based content representation coupled with subset matching allows flexible searches. Load balancing matrices are deployed to improve the system's throughput by eliminating hot-spots. Our approach is distributed in that nodes in the system can make load balancing decisions based on their local load information. The even distribution of registration and query load in LBMs is achieved via hashing and requires no centralized control. Our extensive simulation results validated the system's scalability and load balancing properties. In particular, our system scales to near its operational capacity under extremely skewed load. Finally, the extra cost introduced to registrations and queries by load balancing remains low when the query optimization algorithm is applied.

ACKNOWLEDGMENT

The authors would like to thank Umair Shah, Christos Faloutsos, Shuheng Zhou and Leejay Wu for their valuable comments.

REFERENCES

[1] 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*, San Diego, CA, August 2001, pp. 149–160.

[2] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," in *Proceedings of SIGCOMM 2001*, San Diego, CA, August 2001, pp. 161–172.

[3] A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems," in *Proceedings of Middleware 2001*, Heidelberg, Germany, November 2001.

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

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

[6] Google Inc., http://www.google.com/.

[7] Gnutella, http://gnutella.wego.com/.

[8] 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.

[9] A. Carzaniga, D. Rosenblum, and A. Wolf, "Design and Evaluation of a Wide-Area Event Notifi cation Service," *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 332–383, August 2001.

[10] J. Gao, G. Tzanetakis, and P. Steenkiste, "Content-Based Retrieval of Music in Scalable Peer-to-Peer Networks," in *Proceedings of ICME 2003*, Baltimore, MD, July 2003.

[11] Project IRIS, http://iris.lcs.mit.edu.

[12] P. Reynolds and A. Vahdat, "Effi cient peer-to-peer keyword searching," in *Proceedings of Middleware 2003*, Rio de Janeiro, Brazil, June 2003.

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

[14] D. Thaler and C. V. Ravishankar, "Using Name-Based Mappings to Increase Hit Rates," *IEEE/ACM Transactions on Networking*, vol. 6, no. 1, pp. 1–14, 1998.

[15] Napster, http://www.napster.com/.

[16] 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.

[17] M. Wahl, T. Howes, and S. Kille, "Lightweight Directory Access Protocol (v3)," December 1997, IETF, RFC 2251.

[18] Freenet, http://freenet.sourceforge.net/.

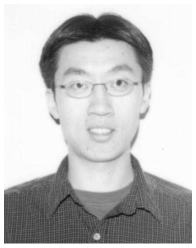
[19] Kazza, http://www.kazaa.com/.

[20] M. Balazinska, H. Balakrishnan, and D. Karger, "INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery," in *Proceedings of Pervasive 2002*, Zurich, Switzerland, August 2002.

[21] D. Dewitt and et al, "The Gamma Database Machine Project," *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, no. 1, March 1990.

[22] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler, "Scalable, Distributed Data Structures for Internet Service Construction," in *Proceedings of OSDI 2000.*

[23] L. Wang, V. Pai, and L. Peterson, "The Effectiveness of Request Redirection on CDN Robustness," in *Proceedings of OSDI 2002*, Boston, MA, Dec. 2002.



Jun Gao is currently a Ph.D. student in the Computer Science Department at Carnegie Mellon University. He received double B.S. degrees in engineering physics and computer science from Tsinghua University, Beijing, China, an M.S. degree in nuclear engineering from the University of Virginia, and an M.S. degree in computer science from Carnegie Mellon University, in 1995, 1997 and 1999, respectively. His research interests lie in the areas of computer networking and distributed systems. More information can be found at http://www.cs.cmu.edu/∼jungao.

Peter Steenkiste is a professor in the School of Computer Science and the Department of Electrical and Computer Engineering at Carnegie Mellon University. He received a B.S. degree in electrical engineering from the University of Ghent, Belgium, in 1982, and M.S. and Ph.D. degrees in electrical engineering from Stanford University in 1983 and 1986. His research interests are in the areas of networking and distributed systems. More information can be found at http://www.cs.cmu.edu/∼prs.