

Distributed Scalable Content Discovery Based on
Rendezvous Points

Jun Gao

May 2002

THESIS PROPOSAL

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Peter A. Steenkiste (Chair)
Christos Faloutsos
Srinivasan Seshan
Ellen W. Zegura (Georgia Tech.)

Abstract

A Content Discovery System (CDS) enables content consumers to discover contents available in the system provided by content providers via a set of content resolvers. CDS systems are an essential component in a wide spectrum of applications including service discovery systems, peer-to-peer object sharing systems and sensor networks.

Existing solutions to 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 only, or the search of strictly hierarchical content names. At the other end, they may offer powerful functionality such as searching of general content names, but at the expense of scalability, since often system-wide query broadcasting or full content name duplication is required.

In this thesis, I propose a distributed and scalable approach to content discovery system that can support flexible search of content names efficiently. Content names in our system are independent from each other, do not necessarily display a hierarchical property, and can be dynamic. Content resolvers form a peer-to-peer general graph of overlay network. Each content name is registered at a small set of resolver nodes, known as the Rendezvous Points (RPs). Queries are directed to the corresponding RP nodes for resolution. The RP-based scheme avoids message flooding in the overlay network and full duplication of contents at all nodes. The CDS dynamically balances load across the resolver nodes in the system to optimize system performance such as response times observed by content registration and query resolution. Our CDS utilizes existing hash-based algorithms for associating names with resolver nodes and routing within the overlay network, and the overlay infrastructure also provides system robustness.

Our CDS is designed as a generic software layer such that higher level applications can be built on top of it. I propose to implement the designed system and demonstrate the proclaimed properties of the system via simulation evaluation and Internet experimentation.

1 Introduction

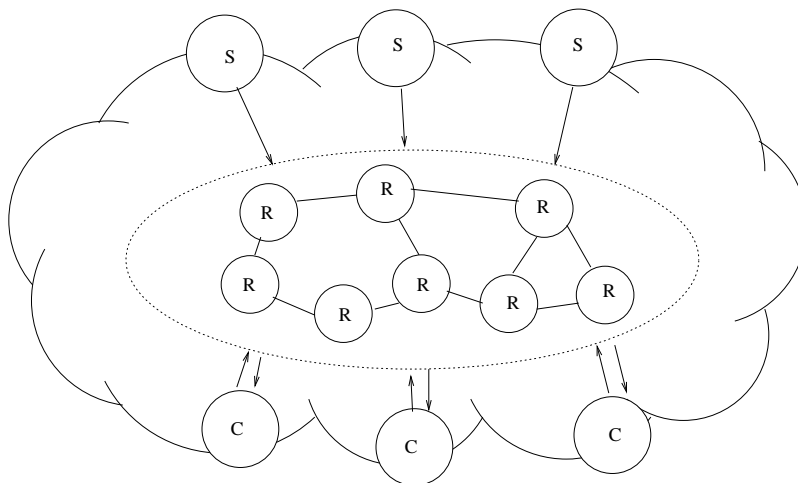


Figure 1: Content Discovery System. S: content providers, C: content consumers, R: content resolvers.

A Content Discovery System (CDS) (Figure 1) is a distributed system that typically consists of three types of entities: Content providers or servers, content consumers or clients and content resolvers. Content providers publish and provide contents. Content consumers issue queries to the content resolvers to locate content. Content resolvers connect to each other to form an infrastructure, and they jointly determine the set of content providers that have content that matches content consumers' queries. There is a wide spectrum of distributed applications that either themselves are a CDS or use a CDS as one of their major components. Content is often represented with a high level name, and depending on the context of the application, a content name may refer to different objects, e.g., the name of a file, the description of a device, or the URL of a web page. We use the term “content” and its corresponding “content name” interchangeably, and in this sense, “content discovery” means the discovery of content names.

As a first example, service discovery system is one type of CDS system. With the growing popularity of pervasive computing, service discovery systems, which allow users to discover and utilize the abundance of devices and sensors running on the Internet, are becoming increasingly important. In such a system, clients are looking for services by issuing queries that describe the search criteria. Devices or the proxies of the devices advertise their descriptions as content names, and the discovery system returns clients the list of devices(e.g., their IP addresses) that match their search criteria.

In recent years, we have witnessed one of the fastest growing applications in the computing history, peer-to-peer applications. The core component of a peer-to-peer (p2p) application, e.g., p2p file sharing, is a content discovery system. Peers can act as clients, servers or resolvers in that they may provide content to share with other peers, can discover other contents on other peers, and can also help to resolve other peers' queries.

Publication subscription systems (pub/sub) also employ CDS. Publishers are servers that advertise the descriptions of their publications. Subscribers are clients that submit their subscriptions or queries. The CDS in a pub/sub system must match subscriptions with advertisements.

Some other conventional Internet applications, e.g., web search engines and the DNS service, can also be viewed as CDS systems. Search engines are content resolvers that determine the set

of URLs published by web servers that match Internet users' interests expressed in their queries. Domain Name Service (DNS) is another type of CDS, in that DNS servers resolve domain names to their corresponding IP addresses.

The primary task of a CDS is to efficiently locate the set of contents that match a client's query. As with the wide range of applications that use CDS systems, there are a variety of solutions to the CDS system. However these existing solutions often display one of the following deficiencies: (1) The CDS may provide powerful functionalities, e.g., supporting flexible searching, but the mechanism does not scale to the wide area, in that when the amount of contents and queries increases, the computation, storage and network bandwidth requirement grow quickly to exceed the capacity of the system; (2) Some CDS systems can scale to wide area, but they typically are designed with a specific application in mind, and therefore their functionalities are limited and their techniques can not be used by other applications. For example, search engines deal with static and inter-linked web pages. DNS deals with hierarchical and well-administered domain names.

1.1 An example: a Highway Monitoring Service(HMS)

Suppose we would like to build a nation-wide highway traffic monitoring service. Devices like cameras and sensors are installed along the road side of highways. Cameras are used to monitor road conditions, such as incidents, constructions, weather(rain, snow, fog etc). Sensors are used to measure speed, and maybe other conditions like temperature and humidity. The system also includes mobile cameras and sensors that may be mounted on police cars. Cameras and sensors must frequently update their information to the system to accurately reflect the status of the highways. Users of the service may send queries like the following to the system to gather information.

- Find out the measured speed at a particular location.
- Find out the speed on all parts of the highway in a particular area to build a "speed map".
- Find a camera at a particular location and can accept new connections, so that the user can connect to it to receive live images.
- Identify the roads in an area where the observed speed is slower than 35mph or the condition is icy, so that the user can avoid them.

To build such a service, the CDS system must be able to support a large number (on the order of hundreds of thousands) of devices, high registration load due to both the number of devices and the frequent updates from these devices, and high query load because of the population scope that the service is targeting.

1.2 Proposed work

In this thesis, we propose to build a CDS system that possesses the following properties.

- Content properties
 - Searchable content
Content stored in the CDS system must be searchable. A client can locate content via the CDS without having to specify the exact canonical name of the content. Instead, it can find the content by specifying some combinations of attributes and values that describe the content.

– Dynamic content

Our CDS must support search for dynamic or frequently changing content. For example, a device, may come and go frequently as the device is turned on and off. Further, a content name itself may also change overtime, e.g., when the content name contains dynamic attributes whose values change frequently.

– Independent content

Content names are typically independent from each other. They do not link to each other in any way, e.g., like web links. They do not necessarily display hierarchical properties, e.g., like domain names.

- Scalable

Most importantly our CDS system must scale to the Internet level. By scalability we mean the performance of the CDS should remain or does not degrade significantly as the amount of available contents, the rate of content registration and the rate of queries increase. The performance metrics include: response time for registrations and queries, network and storage resource consumptions.

- Fully distributed and robust infrastructure

The proposed CDS takes on a distributed approach. Nodes in the CDS system, clients, resolvers, or servers, join or leave at their will and no centralized administration is needed. They form a general graph type of overlay network to ensure system robustness.

- Generic software layer

We observe that many applications, including service discovery, file sharing, pub/sub systems, require the functionality of a CDS, therefore we view CDS as a generic layer which applications can be built on. The CDS we design must provide a flexible interface so that different applications can utilize it.

We plan to achieve the above properties via exploring the following techniques.

- To offer flexible searchability, we represent content names and queries with attribute and value pairs. The CDS allows clients to search contents along any combination or subsets of attributes and values of the content names. To deal with the dynamic aspect of the content, content providers in our system actively register and refresh their contents.
- To ensure scalability, we propose a Rendezvous Point (RP) based scheme for registrations and queries to avoid flooding, which is the main cause of non-scalability in a general graph type of approach.
- To ensure performance, we deploy dynamic load balancing mechanisms that will automatically balance query and registration load among all the nodes in the system to maximize the system performance.
- We organize nodes in the CDS into a peer-to-peer overlay network. The CDS operates on top of existing hash-based peer-to-peer lookup schemes, where high level content names are associated with node addresses and routing within the overlay network address space is done efficiently. The system is fully distributed and no central administration is needed. The peer-to-peer network also offers robustness.

While we focus on the content discovery system in our work, we note the separation of two closely related functionalities in these applications: content discovery and content delivery. Some applications, e.g., publication subscription system and peer-to-peer file sharing systems, may employ both of these functionalities, whereas some other applications are only a content discovery system, e.g., service discovery systems. Due to the close relationship, we must understand how these two functionalities may interact within one application. Content discovery typically occurs before content delivery. Once a query is resolved and the desired content is discovered by the CDS, the mechanism for actual content retrieval or transmission may vary in different applications. There are several possibilities: (1) The resolvers may return the addresses of the content providers, and the content consumers subsequently contact the providers directly to retrieve the actual content in a peer-to-peer fashion. This mechanism is often used in the peer-to-peer file sharing applications. (2) The content resolvers may store the consumers' information, and act also as content routers: they receive actual content from the providers and relay them to the proper consumers. Pub/sub systems typically deploy this mechanism. (3) The third choice is that the resolvers can forward the queries to the corresponding providers, and the providers may then transfer content directly to the consumers.

1.3 Document organization

The rest of the proposal is organized as follows. We present related work in Section 2, where we survey existing solutions to the CDS system in various applications, and point out why these solutions do not satisfy all of our desired goals. We present an overview of the proposed CDS system in Section 3. We present the design of the basic CDS system in Section 4 and we discuss in details in Section 5 how we use Load Balancing Matrices (LBM) to address the load concentration problem encountered in the basic design. We present a brief analysis of the system in Section 6. We outline the proposed work in Section 7, which includes the implementation and evaluation plan, and research directions to improve the proposed system's performance and expand its functionality. We conclude with Time line (Section 8) and Expected contributions (Section 9).

2 Related Work

We survey the CDS solutions in a range of existing applications. Based on how the resolvers are organized, we classify the solutions into two categories: centralized and distributed. We first describe the general mechanisms of each of the solution and then discuss specific systems that use these mechanisms. We compare them with our proposed system.

2.1 Centralized solutions

In the centralized solution, the resolver or the cluster of resolvers form a central data repository. The resolvers are typically administered by one organization and are co-located in the same geographical location.

Resolvers may build their database using different mechanisms: (1) Servers actively register with the central resolver, e.g., Elvin [10] in pub/sub system, and Napster [18] in file sharing; (2) Central resolvers go look for servers, e.g., a search engine like [15] typically finds web contents via periodical crawling.

Clients submit their queries to the central resolvers. The resolvers resolve queries by examining their database. This type of systems typically can support powerful functionalities, such as flexible search.

Since the central resolver must maintain all the content information in the system and also process each and every content registration and query request, the centralized solution typically does not scale well as the amount of content and the number of queries increase.

Some common techniques, such as deploying a cluster of well-connected machines as the resolvers, and using load balancing mechanisms to balance load across these resolvers, are used to make such a solution scale better. Even with these mechanisms, it is still hard for centralized solutions to address the type of applications that we have in mind, where content names are dynamic and do not link to each other and must actively register themselves. For example, it would certainly bring down Google, if the 2 billion pages must register themselves every 10 seconds.

In addition to the scalability problem, the robustness of such a system is another prominent concern. If the resolver(s) is down or being attacked (e.g. by DDoS attack), or the links near the central site are congested or down, the CDS system will not be functional.

2.2 Distributed solutions

In a distributed solution, resolvers form an application level overlay network and content providers and content consumers both connect to some content resolvers. The resolver network topology may be a peer-to-peer general graph or a hierarchical tree.

2.2.1 Graph-based without hashing support

In this type of systems, content resolvers are organized into a general graph. Based on the way content is distributed/replicated in the CDS network, we examine two types of mechanisms.

- Content flooding solution

A content provider publishes its content (name) to the resolver that it connects to. When a resolver receives such information, it inserts it into its local database, and then broadcasts to all its neighbor resolvers. Mechanisms like DVMRP [23] are typically used to avoid message looping. In this solution, content names are effectively replicated at all resolvers. A client sends its query to the resolver that it connects to, and the resolver compares the query with its database and resolves the query.

This type of mechanism is used in many systems, e.g, INS [1] for service discovery, and Siena [3] for pub/sub. The main problem with this solution is also scalability: (1) each resolver must store all the content in the system; (2) each new content will cause waves of broadcast messages throughout the network. This approach therefore does not scale with the amount of content.

Siena [3] tries to improve scalability by exploring the relationship between registrations (subscriptions), e.g., a resolver when receives a new content, broadcasts it only if the new content is not “covered” by some previous content that has been broadcast before. However, the problem is that the degree of aggregation that can be done is often limited, since the names (i) do not necessarily relate to one another, and (ii) may not display a hierarchical nature. Gryphon [2] improves the resolver’s local matching algorithm by constructing a matching tree off-line.

We observe that the main reason behind the full duplication is the “default membership” assumption: a client connecting to any resolver may be interested in any content in the system. By replicating at all resolvers, the client can get its query resolved at the resolver that it connects to. However this assumption is generally not true in the wide area, therefore

the majority of the flooding messages to replicate content are unnecessary. In our CDS, content providers do not flood the whole network, instead they only send their content names to a small set of nodes, known as Rendezvous Points (RPs).

- Query broadcasting solution

In this type of solution, content providers do not actively register their content with the resolver network. Resolvers therefore do not maintain a large database of content as in the above approach. Queries are sent to the resolvers. Resolvers, as they can not resolve queries locally, employ some kind of broadcast searching mechanisms, e.g., BFS or DFS. A query is propagated throughout the network, and eventually it will be resolved by some resolver or received by the content provider. Example systems include Gnutella [12] and Freenet [11].

The sheer number of broadcast messages clearly makes this scheme unscalable. In addition, the client may experience long delay in resolving one query, since the client does not have a good idea on where the potentially matched content would be. To improve scalability, caching is often used. However caching works well only if queries in the system display temporal and spatial correlation. Further caching does not work well for dynamic content. In our CDS, queries are sent directly to the proper RPs for quick resolution.

2.2.2 Tree-based systems

In this type of systems, resolvers are organized into a hierarchical infrastructure. The main advantage of a tree topology is that routing in the tree can be done efficiently. It is possible for each node to maintain a relatively small content database to avoid the full replication of all the content, therefore to limit the scope of registration and searching.

Hierarchical infrastructure works best for applications in which content names display a hierarchical nature. In DNS [16], hierarchical domain names allow a hierarchical organization of DNS servers, which makes the system scale to the Internet level.

Yu et al [24] designed a system for pub/sub applications, where a resolver node maintains only contents that are provided by its children nodes. For queries that the node can not resolve, it will forward them to the parent, which is effectively the default route for all other contents. In SDS [6], registrations at resolver nodes are propagated to their parents, and Bloom filters are used to compress the amount of data to be transferred. The Service Location Protocol (SLP) [13], is also based on a hierarchical organization.

However tree type of hierarchical topology has some fundamental shortcomings. Nodes high in the tree, the root node in particular, will experience high load since all queries that can not be resolved will have to go up the hierarchy. Robustness is another concern: each node could be a potential point-of-failure, in that the crash of any node will cause the partition of the tree.

The CDS we are designing is targeted for a more general set of applications, where content names are not necessarily hierarchical. We prefer a general graph type of resolver network to ensure the fully distributed nature of the system, and to achieve robustness.

2.2.3 Hash-based schemes

One reason that causes the scalability problem in above systems is that the content names generally do not have any relationship with the resolvers who host them, therefore broadcasting or flooding is often needed.

Recent hash-based lookup systems (Chord [22], CAN [19], Pastry [21], Tapestry [25]) address this problem by associating content names with the resolver nodes that will host them. Thus

system-wide flooding or broadcasting is avoided. There have been some recent works that are built on top of the above-mentioned scalable peer-to-peer look up systems, e.g., CFS [7] (file sharing system) on top of Chord [22], Scribe [4] (pub/sub system) on top of Pastry [21]. However, these systems typically work only for locating fixed content names, and do not support searchability. For example, a client must know the exact name of the content it is looking for before hand, and can not locate a content that it is interested but does not know the exact name.

There are a flurry of on-going works that are trying to support content search by building on top of the hash-based schemes. Due to the “on-going” nature, we have only limited knowledge on the specifics of these projects. But a quick read of the related websites reveals that the focus of these projects are different from ours, e.g., [14] focuses on handling complex queries in hash-based peer-to-peer networks, and [20] focuses on efficient keyword searching in peer-to-peer networks.

Our proposed system utilizes the hash-based schemes as a substrate for routing in the overlay networks. Our CDS focuses on how to support flexible search of dynamic content names, and we concentrate on how to balance load across all the resolver nodes in the CDS network to optimize performance.

3 System Overview

The proposed CDS system provides a distributed solution to the content discovery problem: nodes in the system organize themselves into a peer-to-peer overlay network. Content names are published to this network and queries are also resolved by the network. In this section we give an overview of our proposed CDS system. We first explain the node architecture from a software system’s point of view. We then present how we leverage the existing hash-based work to manage our CDS overlay network. We then define the exact meaning and representation of content names and queries used in the CDS. We lastly highlight the RP-based registration and query scheme.

3.1 Node architecture

Nodes participating in our CDS connect to each other in a peer-to-peer fashion to form an overlay CDS network. There are two different models of using CDS in applications. In the first model, the three types of entities, namely the providers, consumers, and resolvers, are a logical separation. A node can be a content provider when it provides content, and it can also be a consumer in that it can issue queries to look for other contents. Nodes also receive and store content registrations from content providers and resolve clients’ queries. In the second model, nodes in the CDS are dedicated resolver nodes for storing registrations and resolving queries. Content providers and consumers are true end systems, in that they only submit contents and queries, and are not responsible for hosting contents and resolving queries.

As discussed earlier, our CDS system is designed as a flexible module that high level applications, such as service discovery, file sharing, can use as a building block. The software architecture of a CDS node is shown in Figure 2. The programming interface (API) that the CDS provides to the application must include at least the following two methods: `publish(content)` and `locate(query)`. A node can publish its content by invoking the first method, and a node can invoke the second method to search for content.

The CDS system utilizes existing hash-based overlay network management system for overlay network related functionalities, e.g., the construction of the overlay, and the routing in the overlay, etc. For example, when the CDS layer’s `publish` method is invoked by the application, the CDS module will then decide the set of nodes in the overlay network that should receive this content,

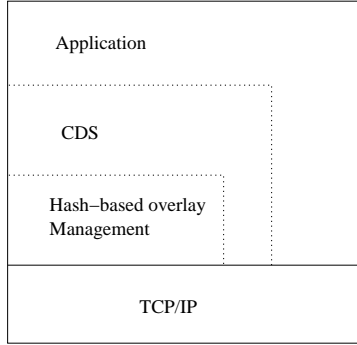


Figure 2: Node architecture

and then issue the proper commands to the overlay layer. The overlay layer subsequently sends the content to the selected destinations.

3.2 Hash-based CDS overlay network

Our CDS utilizes the recent hash-based content lookup algorithms ([22], [19], [21], [25]) to organize the resolver network and to route messages within this network. Due to the large degree of similarity in these schemes, in the following discussions, we use Chord as an example for the overlay management layer. We summarize some properties of the hash-based algorithms that are important to the CDS system.

- Addressing

Each node in the overlay network is assigned an overlay address, the node ID, which is computed via hashing. For example, a node in Chord computes its own ID by applying a consist hash function to its IP address.

- Topology forming

The neighbors of a node are determined once the ID is known. For example, in Chord, based on the node's newly computed ID, a node is logically inserted into a unit circle, and its neighbors are then determined based on the relative positions of other nodes already existing in the circle. Other systems use similar techniques.

- Routing and forwarding

Hash-based overlay networks use a form of topological routing, and do not require to run IP-like network-wide routing protocols. Each node maintains a relatively small routing table, e.g., the size of the routing table is $O(\log n)$ in Chord, where n is the number of nodes in the network.

Forwarding in such a network is solely based on node ID. A node knows which outgoing overlay link to use to forward a message by simply examining the destination ID. The length of a path between any two nodes in the overlay network is $O(\log n)$ in terms of overlay hops.

- Support for name lookup

Unlike the traditional IP network, where the contents can be arbitrary on a given IP host, i.e., content and address do not relate to each other, in the hash-based systems, a content name is also hashed to get a node ID, and the content, or the content name will be stored

on that node. Therefore once a content name is known, its location, or which node has it, is also known. By associating content names with node IDs, locating a content is equivalent to compute the content's hash.

A node may join or leave the overlay at any time and may crash. At those times only a small portion of the names in the system need to be shifted.

3.3 Attribute-Value pairs

To enable flexible search, content names and queries are represented with attribute-value pairs (AV-pairs). AV-pairs describe the multiple aspects of contents. For example, AV-pairs can be used to describe a service in a service discovery system; AV-pairs can be used to represent events and subscriptions in a pub/sub system; and AV-pairs can be used to represent files in an object sharing system. Such a naming scheme is searchable in that clients can find a content by specifying partial description of the content. AV-pairs enable more powerful searches than just keywords. This type of naming scheme, or similar schemes have appeared in other works, e.g., in INS, SDS, Siena, etc.

3.3.1 Content name representation

In the following discussion, we use service discovery system as our example application. The mechanisms can be readily applied to other applications. A server may have one or multiple services to offer and each service is described with a service description (SD), which is represented with AV-pairs. Service descriptions are “content names” in this type of applications, and they are in the following format:

$a_1 = v_1$	
	$a_{11} = v_{11}$
	...
	$a_{12} = v_{12}$
	...
	...
$a_2 = v_2$	
...	
$a_n = v_n$	

where a_i 's are attributes and v_i 's are values. We use the following format to represent an SD: $SD : \{\{a_1 = v_1, a_{11} = v_{11}, a_{12} = v_{12}\}, a_2 = v_2; \dots, a_n = v_n\}$, or $SD : \{\{av_1, av_{11}, av_{12}\}, av_2, \dots, av_n\}$, where av_i represents $a_i = v_i$. Note that in the initial system, we only consider the case where attributes take on exact values, and we will consider inequality assignment of values to attributes in the future. An example service description for a camera in our highway monitoring service may be like the following:

Camera Number = 5562
Camera Type = quick cam
Connection Status = available
Highway Number = 279 S
Exit Number = 4
State = PA
City = pittsburgh
Speed Measured = 45mph
Road Condition = dry

There are two types of relationships between attributes: orthogonal or dependent. In the above example, a_1 and a_2 are orthogonal attributes, and a_{11} and a_{12} are dependent attributes of a_1 . Two attributes are orthogonal to each other if they can independently exist in one name. In comparison, dependent attributes do not exist by themselves. They must co-exist with their parent attributes, e.g., a content name may have attribute a_{11} only if it has attribute a_1 . An attribute is defined to be the dependent attribute of some other attribute if in the system we do not allow queries to use that attribute individually as a search criteria. For example, “Exit Number” is a dependent attribute of “Highway Number”, since we define that it is not meaningful to search for cameras based on the exit number only.

There are two types of attributes in describing a device: static attributes and dynamic attributes. For a particular device, the value of a static attribute does not change during the life time of the device, e.g., in the above example, Camera Number and Camera Type are static attributes. On the other hand, values of dynamic attributes may change frequently for a device. For example, the speed observed at a section of the road is constantly changing, the road condition may be dry or wet. Even the location related attributes may take on different values if we consider that cameras mounted on vehicles. The dynamic aspect of content names requires names to be frequently refreshed with the system.

3.3.2 Queries and Subset Matching

Queries are issued by clients, and they also consist of AV-pairs, and the matched content must simultaneously satisfy all the AV-pairs presented in the query. For example, a query may be $Q : \{av_1, av_2\}$, which means the matched content must satisfy $a_1 = v_1$ AND $a_2 = v_2$. If a query contains dependent attributes, it must also contain the corresponding parent attributes. For example, $Q : \{\{av_1, av_{11}\}, av_2\}$ is a valid query, but $Q : \{av_{11}; av_2\}$ is not a valid query.

A query matches a content 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. For example, all the descriptions that have $a_1 = v_1$ and $a_2 = v_2$ will be matches of Q . It is possible that the content description may have other components in it. The AV-pairs that are in the content name but not in the query are considered as “don’t care” by the clients.

We now compute the number of possible matched subsets given a content name. Suppose the content name consists of n orthogonal AV-pairs only, then the number of subsets (excluding the empty set), or the total number of different queries that can be matched by this content name is

$$\binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{n-1} + \binom{n}{n} = 2^n - 1.$$

Since we consider a dependent AV-pair and its parent AV-pair(s) together as one “combined” AV-pair, if the name contains dependent AV-pairs in the set, the actual number of matched subsets will be smaller, e.g., we would not have $\{av_1, \{av_1, av_{11}\}\}$ as a possible subset, instead we would only have $\{av_1, av_{11}\}$.

3.4 RP-based registration and query scheme

One of the fundamental problems in designing the CDS is where to store the content names such that when the system is stable, content in the system can be discovered. Existing broadcast search or content flooding solutions may work well in a local area setting when the load is low, however they do not scale to the wide area.

This phenomenon has been observed in another application: IP multicast. In the original design of multicast protocol [8], the group information is pushed to every router with the concern that a host connecting to that router may join the group. This leads to the explosion of group states that must be maintained on every router. The PIM-SM [9] wide area extension to the original protocol distributes the group information only to a small set of routers, called Rendezvous Points, and the assumption is that clients are not interested in the groups by default, and a client who wants to join a group must do so by explicitly contacting the RP points.

In the CDS system, we believe that using RP points is also a scalable approach for distributing content names. In our CDS, content providers actively register contents with selected sets of nodes in the system, known as the Rendezvous Points (RPs). RPs are the resolvers for the contents that are registered with them. Content consumers (clients) direct their queries to the proper RPs to get them resolved. Nodes in the CDS also act as message routers to propagate registration and query messages within the CDS network. This approach has two benefits: (1) contents will not be replicated at all resolvers, thus no flooding of content names; (2) clients in search of content can wisely visit the RP nodes, instead of blindly sending broadcasting searching messages through the whole network, thus no broadcast searching.

In RP-based system, it typically requires a separate RP discovery mechanism to locate the RP points, e.g., PIM-SM uses bootstrap configuration mechanism. The use of hash-based lookup mechanism as the substrate of the CDS system further helps the RP-based system. We select the set of RPs for each content based on the content's name. We assign content names to the set of RPs using the hash-based lookup algorithm. Therefore a client immediately knows the corresponding RPs for a given query. This way we do not need to establish a separate RP discovery mechanism.

4 Basic CDS system

In this section, we explain the operations in the CDS.

4.1 Registration mechanism

Content providers actively register their contents with RP resolvers in the CDS. They first must determine the addresses/IDs of the RPs, and then send the full content names to the RPs. The resolvers maintain the names in a soft state fashion, thus the providers must refresh their registered contents periodically.

There are two goals in the registration process: (1) The content must be registered at nodes in a way that no matter what the queries are, the matched content must be found; (2) We must limit the number of RP nodes used for one content name.

4.1.1 Determine the RP set

Suppose a provider must register the following name: $SD : \{\{av_1, av_{11}, av_{12}\}, av_2, \dots, av_n\}$. The provider determines the IDs of the nodes where it should send the registration to by applying the system-wide hash function \mathcal{H} , to the AV-pairs in the description. For the first level attributes, a_i , for $i = 1, n$, the provider computes

$$\mathcal{H}(a_i = v_i) = N_i.$$

For a dependent attribute, the hash function is applied to it and its parent AV-pairs together. For example, in the above example, attribute a_1 has two dependent attributes a_{11} and a_{12} , which

take on values v_{11} and v_{12} respectively. The following computation is carried out:

$$\mathcal{H}(\{a_1 = v_1; a_{11} = v_{11}\}) = N_{11}.$$

$$\mathcal{H}(\{a_1 = v_2; a_{12} = v_{12}\}) = N_{12}.$$

N_i 's are node IDs in the network. For an SD that has n AV-pairs (including dependent AV-pairs), the hashing computation is carried out n times, which will yield n different node IDs, when we assume no hash collisions. These n nodes constitute the RP set for this SD. Therefore the size of the RP set is n for an SD that has n AV-pairs.

As an example, suppose we have the following two devices:

SD1	SD2
Location = WeH7110	Location = WeH7110
Device = Digital Camera	Device = Digital Camera
Make = Canon	Make = Minolta
Model = PowerS30	Model = S304
Owner = jg	Owner = lw

By applying the hashing technique to SD1, we get:

$$\mathcal{H}(\text{Location} = \text{WeH7110}) = N_1$$

$$\mathcal{H}(\text{Device} = \text{DigitalCamera}) = N_2$$

$$\mathcal{H}(\text{Device} = \text{DigitalCamera}; \text{Make} = \text{Canon}) = N_3$$

$$\mathcal{H}(\text{Device} = \text{DigitalCamera}; \text{Make} = \text{Canon}; \text{Model} = \text{PowerS30}) = N_4$$

$$\mathcal{H}(\text{Owner} = \text{jg}) = N_5$$

Thus, the RP set for SD1, $RP_1 = \{N_1, N_2, N_3, N_4, N_5\}$. Similarly, for SD2, we have $RP_2 = \{N_1, N_2, N_6, N_7, N_8\}$. Notice that RP_1 and RP_2 share two nodes N_1 and N_2 , since two of the AV-pairs in SD1 and SD2 are the same. Once the node IDs of the RP set are determined, the service provider sends the full content name to each node in the RP set (Figure 3).

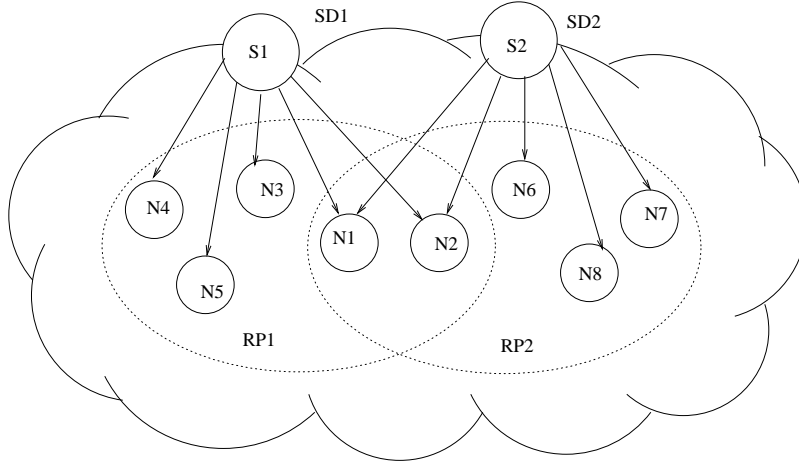


Figure 3: Registration with RP set

Content providers must refresh their registrations for several reasons: (1) the resolver node keeps names in a soft state fashion, therefore without timely refreshing, the names will expire; (2) since content names have dynamic attributes whose values may be changing frequently, it is then necessary for the service to register frequently to update its information. Note that in this case, when the value of an attribute changes, the hash function will generate a different RP node ID; (3)

a resolver node may leave or crash, and in this case, the refresh message will allow the content to go to a node that is alive in the system.

4.1.2 Discussions

We justify the design decisions we make in the registration process.

- Hash attribute and value together

In our system, attribute and its corresponding value are hashed together to determine the set of RP nodes. An alternative way may be hashing the attribute only to decide the rendezvous nodes. The primary reason that we hash attribute and value together is to reduce the database size on the RP node and therefore the computation (insertion and search) overhead on the RP node, since many names may contain the same attributes but they take on different values, e.g., the attribute “location”. In addition, queries are in the form of `attribute = value`, and search is carried out on the combination not attribute alone.

One main downside is that hashing AV-pair together adds difficulty to range search. We will explain how we handle range search in the proposed work section.

- Hash each AV-pair individually

By registering with nodes that correspond to each individual AV-pair in the name, it is guaranteed that any query that consists of at least one of the AV-pair of the content name can find it. The compromise in our scheme is that since a node is only specialized in one AV-pair, when it receives a query, it must compare the rest of the AV-pairs in the query with the names in its database.

We treat dependent AV-pair together with its parent AV-pairs as one entity and hash them as a whole. The system remains correct in terms of resolving queries, since as we have defined earlier, queries do not contain solely dependent AV-pairs.

As a comparison, if we map the combination of some orthogonal AV-pairs together onto one node, the node becomes more specialized and that will improve the efficiency of a query resolution. However the trade off is: it is possible that some query may not be able to find the content. For example, if the name is registered at a node that corresponds to the hash of av_1 and av_2 together, query that has only av_1 will not be able to find it, as the hash of av_1 and av_2 is different from the hash of av_1 alone.

One extreme way of ensuring that a content can be found by any query that is a subset of the name is to register the content at nodes that correspond to the all combination of the AV-pairs. As shown earlier, the total number of RP nodes that will be involved will then be $2^n - 1$. In this approach, the client can simply hash the whole query to get the RP node ID, and on that RP node, no further comparison of AV-pairs is needed. However in this scheme, the number of RP nodes grows exponentially with respect to the number of AV-pairs, n , in the name. For a name that has 10 AV-pairs, it would need 1023 RP nodes.

We plan to explore how we may combine AV-pairs intelligently to improve the system’s performance in the proposed future work.

- Registering the full name with RP node.

The reason that the full name is registered at a RP node is that when the node receives a query, it can resolve it completely. In comparison, if the resolver node only has part of the full content name, a client must send a query to multiple nodes to get it resolved.

4.2 Query mechanism

Clients in the CDS issue queries to locate content. As we discussed earlier, our CDS system allows subset matching, a name should be a match for queries that include any selection or combination of the AV-pairs in this name. We illustrate the client mechanisms again by using the example $SD : \{\{av_1, av_{11}, av_{12}\}, av_2, \dots, av_n\}$.

In order to have their queries resolved, clients must determine based on their queries the set of RPs that may contain the potentially matched content. Suppose the query is $Q : \{\{av_1, av_{11}\}, av_2\}$. As we discussed in the registration section, content names that contain av_1 (and maybe other AV-pairs) are stored in the node that corresponds to av_1 , content names that contain av_2 (and maybe other AV-pairs) are stored in the node that corresponds to av_2 , and content names that contain $\{av_1, av_{11}\}$ are stored at another node that corresponds to $\{av_1, av_{11}\}$. Thus the query that contains all of its AV-pairs can be sent to any one of these nodes to get it resolved. The client computes the three RP nodes' IDs as follows:

$$N_1 = \mathcal{H}(av_1)$$

$$N_{11} = \mathcal{H}(av_1, av_{11})$$

$$N_2 = \mathcal{H}(av_2)$$

The clients must choose a RP node that has a good performance, e.g., small response time. In selecting a RP node, client uses the following strategies:

- Client prefers nodes corresponding to dependent attributes.

The client can send to any one of these three nodes to get its query resolved. However, it is clear that node N_{11} must contain a smaller database (assuming no other AV-pair maps onto this node) than the one N_1 has, since all the names N_{11} has must also appear in N_1 . With the assumption that with a smaller database, the query will be resolved faster, between N_{11} and N_1 , client would choose N_{11} .

- In determining whether to use N_{11} or N_2 , client has several possibilities:

- Query one node

A client may select a node randomly. For example, it may choose N_{11} or N_2 . The client may pick one node that has good performance in the history, e.g., it may be the case that the client received query resolution back faster from N_{11} than N_2 before, and the client may continue to send its query to N_{11} .

In the above example, suppose a query is $\{\text{Location} = \text{WeH7110}; \text{Device} = \text{Digital Camera}\}$. If the client chooses to hash the first AV-pair, it will get the ID of N_1 , and it then sends its query to N_1 . N_1 finds two SDs, SD1 and SD2, in its database that match the other AV-pair (Device = Digital Camera) in the query. (Figure 4)

- Query multiple nodes

The client may select a node using a two-pass algorithm: it first probes the related RP nodes to get a count of how many content names they each hold and then sends its query to the node that has the least amount of names. For example, if N_1 has less content names than N_2 , the client will send its query to N_1 . Alternatively, the client may use a parallel mechanism, in that it can concurrently send query messages to all the corresponding nodes and wait for the first node that replies.

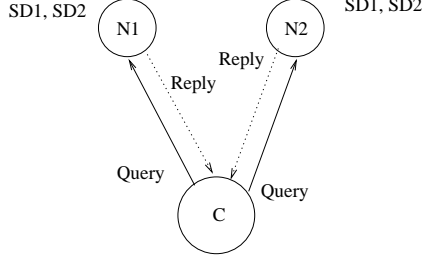


Figure 4: Query resolution

4.3 Resolver mechanism

A resolver node is responsible for receiving name registrations and resolving queries. A node stores all the content names that share the AV-pair that the node is responsible for. As different AV-pairs may have the same hash value, one node may be the host of multiple AV-pairs. A node becomes specialized in the particular AV-pair that it gets mapped onto in that it will host all the contents that have this AV-pair in their names. In our previous example, N_1 becomes the specialized node for $\{\text{Location} = \text{WeH7110}\}$. We also observe that for some common AV-pairs that are shared by many names, these names will end up being mapped onto the same node. In the above example, N_1 and N_2 will host both SD1 and SD2. A RP node logically stores a database as the following:

<i>AV - pairs</i>	<i>Content name</i>	<i>Provider</i>	<i>Attribute list</i>				
$a_1 = v_1$			a_1	a_2	a_3	a_4	...
	SD_1	$S1$	v_1	v_2	v_3	—	...
	SD_2	$S2$	v_1	v_2	v_3'	v_4	...
	...		v_1
$a_i = v_i$			a_1	a_2	a_3	a_4	...

In this example, the RP hosts two AV-pairs: $a_1 = v_1$ and $a_i = v_i$ (assuming $\mathcal{H}(av_1) = \mathcal{H}(av_i)$). In this type of data organization, a new name can be inserted into the database efficiently. To resolve a query, the node must perform pair-wise comparison between the AV-pairs in the query and the AV-pairs in each of the name in the database. To speed up the matching computation, by taking advantage of the fact that we are only considering exact matching in the basic design, we use a fast algorithm based on hashing to manage the content name database.

A RP node organizes content names according to their AV-pairs. When RP receives a registration message for a new name, it hashes (unrelated to hash function \mathcal{H}) each AV-pair in the name, and inserts it into corresponding slots in the hash table. For example, SD1 is inserted into two places, $h(a_2 = v_2)$ and $h(a_3 = v_3)$ and SD2 is inserted into three places (Figure 5).

When the RP node N_i receives a query, it first makes sure that it is responsible for the AV-pair of which the client used to locate this RP node. The node then hashes the rest of the AV-pairs in the query and merge the corresponding lists in the hash table. If matches are found, the RP node can reply to the client with the results. In the basic system, the resolver may just return the matched name's IP address to the client. In a more sophisticated system, the resolver may behave more like a general-purpose database, e.g., client may specify the desired attribute values of the matched content. We will address this aspect in the proposed future work.

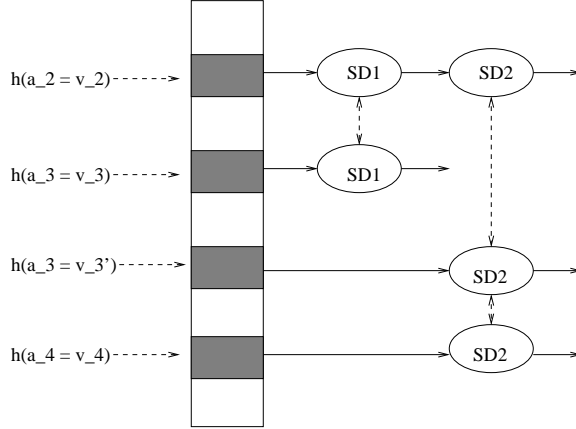


Figure 5: Data structure for matching based on hashing

5 System with load balancing

5.1 The load concentration problem

One problem with basic RP-based approach is that the system performance will degrade if the RP points are overloaded, e.g., getting too many registrations or queries. To maximize the performance of the system, we must make sure that it is not the case the some nodes in the system are overloaded while other nodes remain idle.

The basic system tries to split registrations (and queries) among nodes in the network by using a different set of RP points for different content names. However these sets may still overlap if the names share some common AV-pairs, or with a smaller chance that two different AV-pairs have the same hash value. Similarly, for two queries that share one common AV-pair, they may be both sent to the node that corresponds to the common AV-pair. Therefore, it is still possible that some nodes may get (1) lot of registrations if it corresponds to a popular AV-pair shared by many content names, (2) lot of queries if the AV-pair it is responsible for appears in many queries.

We first explain the three factors that determine “load” in the CDS system.

1. Number of content names.

A common AV-pair may be contained in a large number of content names. The result is that the node that corresponds to this pair must maintain a large database. Large database causes large computation(matching) overhead.

The distribution of AV-pairs in content names is likely to be skewed. At one extreme, for attributes that have only 2 possible values, on average there are half of the names in the system will take one of the two values, e.g., for attribute “road condition”, which can be either dry or wet, As another example, it is possible that most of the cameras in the system observe (speed=45mph). At the other extreme, for some rare values, only a few names contain those AV pairs, e.g., the pair (speed=100mph).

For the node who is responsible for an extremely popular AV-pair, it must host a large number of names, e.g., half of the total names in the system. On the other hand, a node may be very empty if it corresponds to a rare combination of AV-pair.

2. Content registration rate.

A node may observe higher registration rate than others if it maintains a larger name database. It is also possible that though the database is small, registration rate may still be high because the names it is hosting may have to update themselves frequently due to their changing values. In either case, a RP node may be swamped by the high registration rate.

3. Client query request rate.

AV-pair distribution in queries may also be skewed. Some AV-pairs may be contained in many queries, which may cause the queries to be sent to the same nodes. The skewed effect may be a transient phenomenon, e.g. in the case of “flash crowd” effect, or it may be persistent in that common interests are shared. A node may be overloaded by high query rate, while other nodes see few queries.

The observation is that while some nodes in the system are overloaded, some nodes may still be under utilized. The idea to avoid the “hot spot” problem is to spread the load around all nodes in the system so that one node will not be swamped by service registrations or queries.

5.2 Load Balancing Matrix (LBM)

In this section, we present our mechanism using load balancing matrix (LBM). Unlike in the lightly loaded case, where registrations and queries can be sent directly to the RP nodes, if the system gets loaded by the above factors, nodes in the system that are less utilized must be discovered and used to share the load. In particular, two techniques, partition and replication, are used. The content names are partitioned among multiple nodes if the supply of content or the content registration rate is high. We replicate content names at multiple nodes when the query rate is high.

We make sure that a node only holds a limited number of names, and the extra content names that come to this node will be automatically redirected to other nodes. This also applies to situations when registration rate is high. Similarly, the content on a node may be dynamically replicated if it experiences high query rate.

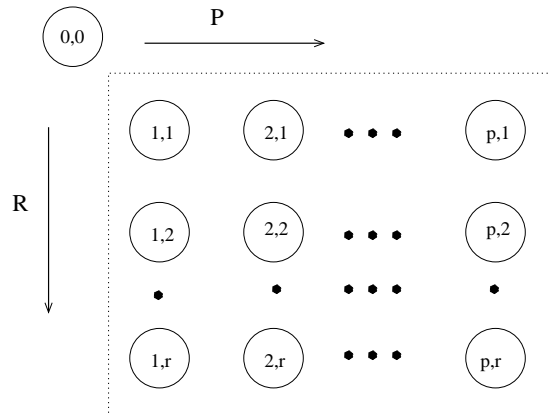


Figure 6: Load balancing matrix

A popular AV-pair may be shared by a large number of content names and may appear in many clients’ queries. The CDS system may have to use a set of nodes instead of one to store the SDs that have this AV-pair in their descriptions. This set of nodes are organized into a logical matrix, and it is specially named the load balancing matrix, since its purpose is to balance load and provide good performance.

The matrix has two dimensions: P and R , where P represents Partition, and R represents Replication. Figure 6 shows the layout of the matrix corresponding to AV-pair av_i . A node in the matrix is denoted by $N_i^{(p,r)}$, where p is the column index and r is the row index, and i is the index of the AV-pair that this matrix is responsible for, av_i . The IDs of the nodes are determined by: $N_i^{(p,r)} = \mathcal{H}(av_i, p, r)$. For each matrix, there is a special “head” node, $N_i^{(0,0)} = \mathcal{H}(av_i, 0, 0)$. Head node stores the current size of the matrix along the two dimensions, p and r , which are initialized to be 1 at the system start up time. The first name that contains av_i will be registered at node $N_i^{(1,1)}$. Each column contains a subset of the content names that has av_i , which is why they are called partitions. The nodes in each column are replicas of each other, in that they host the same set of names that contain av_i .

The matrix is dynamic in that it grows or shrinks depending on the load along its two dimensions. The matrix may consist of only one node when load is low, as in the basic system. It may also be in the shape of one row, when registration is high, or one column, when query request is high. The nodes must maintain the following statistics to grow and shrink the matrix: the total number of names it has; the rate of name registration, and the rate of client query. In correspondence to these statistics, nodes must maintain three thresholds.

1. T_{SD} . A resolver maintains the maximum number of content names it can hold.
2. T_{reg} . The maximum rate of registration the resolver can sustain.
3. T_q . The maximum query rate from clients the resolver can sustain.

We next explain the registration and query operations with matrix and the matrix management mechanisms.

5.3 Operations with LBM

5.3.1 Registration

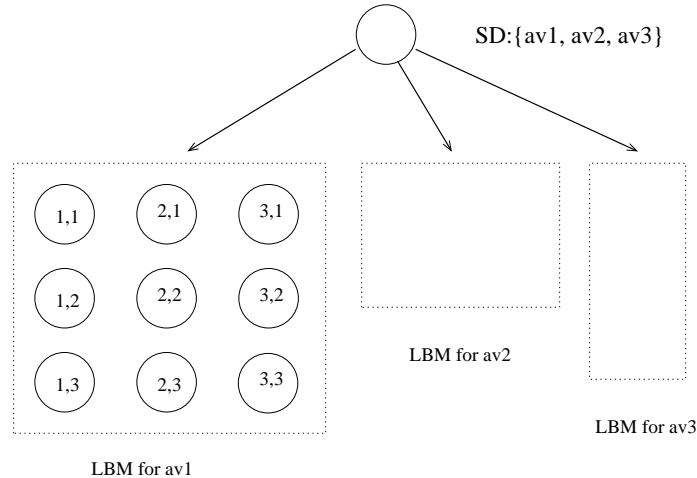


Figure 7: Registration with Load balancing matrices

In contrast to in the basic scheme where a provider can register with each node that corresponds to one of the AV-pair in its name, with LBM, a provider must register with corresponding matrices.

Content providers determine the rendezvous matrices (RM) the same way as they determine the RP nodes by applying hash function to its AV-pairs. The difference is the content provider must register its name with one column of nodes in each matrix. It determines which column by first discovering the current values of p and r . We have the following mechanisms for content provider to discover p and r .

1. Retrieve from the special head node.

The values of p and r are stored at the head node corresponding to the AV-pair in question.

The content provider computes the ID of the head node for pair av_i as follows, $\mathcal{H}(av_i, 0, 0) = N_i^{(0,0)}$, and then retrieves the values of p and r from it.

2. Discovery via binary search

However the above scheme alone may not be sufficient, since the head node may be down or become a bottleneck. For example, since all the registrations and queries that contain the same AV-pair will come to the same head node first, it may be overloaded. (Note that no matching computation is involved on the head node.)

Under these conditions, a provider can still find out the current value of p by sending out probes to the corresponding matrix. A node when receives such a probe can reply with whether it can accommodate more names or accept new registrations. The provider will register its name at the RP node that is available. Since the nodes in the matrix are ordered, the dimension can be discovered efficiently using binary search.

3. Based on cached values

Content provider may also explore caching mechanisms. This is particularly useful when it does frequent refreshing. For a particular AV-pair, the content provider caches its corresponding LBM's p and r values and reuse them without rediscovering them. It must rediscover these values when its registration fails using the cached values.

Once the size of the matrix is found, registration is sent to the nodes in the last column. There are two choices for the registration process: (1) it registers with the first node in the last column

$$\mathcal{H}(a_i = v_i, p_i, 1) = N_i^{(p_i,1)},$$

for $i = 1, n$, where p_i is the P dimension value for av_i 's matrix. Then the first node propagates the registration to other nodes in the column; or, (2) it registers with all the nodes in the column itself, $N_i^{(p_i,r)}$, where r varies from 1 to r_i .

Content registered at one node will expire after a certain time period, and must therefore be periodically refreshed. The provider must go through the same discovery and computation period to re-register the content. Notice that at different times the values of p and r may be different, thus the set of RP nodes may be different at each registration/refresh time.

5.3.2 Queries

When LBM is being used, a client must determine which matrix it needs to use to send its queries. As we discussed in the basic system, any matrix that corresponds to the AV-pairs in the query can be used for query resolution. The clients must choose a RP matrix that has good performance. In selecting a matrix to send its query, the client may use the same mechanisms that was presented in the basic system. It may pick a random AV-pair from its query and send to that pair's corresponding

matrix. It may send its query to all the matrices. Alternatively, client may probe the head node of each corresponding matrix to get the size of each matrix and send its query to the one that has the smallest number of columns.

Once a matrix is selected, suppose the matrix corresponds to av_1 , the client must send the query to all the columns of the matrix because nodes in each column contain a partition of all the names that have av_1 in them. To do that, client must first discover the size of this matrix. Client does so by using the same mechanisms used by the content providers as described above. For example, it applies the same hash function \mathcal{H} to av_1 to get the head node's ID $\mathcal{H}(av_1, 0, 0) = N_1^{(0,0)}$. Client then receives the values of p_1 and r_1 from $N_1^{(0,0)}$.

Since the nodes in the same column are replicas of each other, the client's query needs only to be sent to one node in each column. To balance the load of the nodes in each column, it picks a random node as the query resolver. Client computes the following IDs: For each $p = 1, p_1$, choose a random r between 1 and r_1 (inclusive), and compute $\mathcal{H}(av_1, p, r) = N_1^{(p,r)}$. The client then sends p_1 query messages to these p_1 nodes.

5.4 Matrix management mechanisms

Each load balancing matrix grows or shrinks based on the system's load status.

5.4.1 Matrix P-dimension expansion mechanism

There are two possible reasons that the matrix may expand along the P dimension: when the number of names on a node reaches T_{SD} or the registration rate at the node reaches T_{reg} .

Note that a node receives new registration only if it is a node in the last column. If the amount of names on the node reaches T_{SD} , or if it sees the registration rate approaches T_{reg} , it will spawn off a new node by sending a message to the corresponding head node to increase the p value by 1. In fact, if the matrix has multiple rows, a new column of nodes will be recruited for the new registration. The nodes in the new column have the following IDs: $\mathcal{H}(av_1, p + 1, r) = N_1^{((p+1),r)}$, where r is the number of rows(replicas). Thereafter, future registrations will start to fill up the nodes in this column.

Consider the nodes in the first row of the matrix, each node holds different set of content names, but they all share the common AV-pair, av_i . Therefore the content names that have av_i in them are partitioned among the nodes in the same row.

5.4.2 Matrix R-dimension expansion mechanism

Since each column contains a partition, to guarantee that all the matched content names will be found, clients' queries must be sent to all columns. We examine the first column in the matrix to illustrate the mechanisms.

Initially all queries will go to $N_i^{(1,1)}$, when the query rate reaches T_q , $N_i^{(1,1)}$ will spawn off a new node, $N_i^{(1,2)}$, and duplicate its content at this node. $N_i^{(1,1)}$ contacts $N_i^{(0,0)}$ to increase the r value by 1. As explained earlier, since the clients pick a random node in each column to send its query to, once $N_i^{(1,2)}$ is created, the queries will be shared by these two nodes. The nodes in each column observes the same query rate, therefore anyone of them may prompt the birth of a new node. (Note that when a node is responsible for multiple AV-pairs, it has to spawn off a new replica for each of the pair.)

This mechanism brings in new nodes additively. An alternative to the additive increase may be the multiplicative increase, in that when the nodes in one column decide to expand, they double the amount of existing nodes, i.e, increase from 1 to 2 to 4, and so on.

From the whole matrix point of view, every query is sent to all the columns, therefore, the query rate is uniformly observed on all these nodes. To maintain the matrix property, during transient time, one column may decide to increase the value of r before others. The head node after increasing r must inform the new nodes in other columns to retrieve data from earlier nodes in their respect column.

5.4.3 Matrix P-dimension compaction mechanism

The matrix should be kept as small as possible to minimize the number of replicas and partitions, since a query has to be sent to all the columns and a registration has to be sent to all the nodes in one row. We use two mechanisms to compact the matrix along P dimension.

- Push

The first node in the last column, $N_i^{(p,1)}$, probes other first row nodes and if these earlier nodes can accommodate content names, it will push its content to these nodes.

- Pull

Nodes in the first row except the last one, $N_i^{(1,1)}$ to $N_i^{(p-1,1)}$, when they have extra space, actively probe the last node $N_i^{(p,1)}$ and retrieve content from it.

When the last column is not holding any content, it will remove itself from the matrix. It informs $N_i^{(0,0)}$ to decrease p by 1. The first row nodes must ensure the consistency in their respective columns. When they get new content names, or when they remove content names, the first row nodes send update messages down along its column to keep the replicated data synchronized.

5.4.4 Matrix R-dimension compaction mechanism

Since the query rates observed by the nodes along this dimension are uniform, the last node in each column when observes the rate below a low threshold, will remove itself. Again due to the uniform distribution, in fact, the whole last row will be removed. The head node decreases the value of r by 1.

6 System Analysis

In this section, we present a brief analytical analysis of the proposed CDS system. A comprehensive performance evaluation based on a real implementation is part of the proposed future work. As we presented in the design sections, we first examine the system’s properties when no load balancing matrices are used. We then analyze the system when load balancing matrices are deployed. In the following analysis, we assume the resolver nodes are homogeneous, in that they have the same computation, storage and network capacity.

6.1 System performance without LBM

Load to the CDS is determined by three parameters (1) the number of content names, (2) the registration rate and (3) the query rate. Load balancing matrices are not needed if the load

observed on each resolver node is the same, i.e., each node receives the same number of name registrations and queries.

6.1.1 Registration cost

As described earlier, the RP-based scheme allows each content name to be registered with a small set of RP nodes. For a content name, $SD : \{av_1, \dots, av_n\}$, which has n AV-pairs, the content provider must register it with the n nodes corresponding to the n AV-pairs, which requires n registration messages. Typically n is a small number (e.g., $n = 20$) compared to the number of resolvers in the CDS.

The registration latency is determined by the underlying overlay network and the processing speed on the resolver nodes. Since the provider can send its registrations out concurrently to all the n RP nodes, the latency to register one name will be the RTT ($O(\log N_c)$ in terms of overlay hops in Chord) to the “furthest” node.

6.1.2 Query cost

To resolve a query that has m AV-pairs, $Q : \{av_1, \dots, av_m\}$, the client only needs to send it to 1 RP node that corresponds to one of the m AV-pairs. In the case that the client probes the m corresponding RP nodes first and then selects one that has the best response time, the total number of messages needed for one query is then $m + 1$.

6.1.3 RP node database

Suppose there are N_s content names in the system. The hash-based scheme tries to distribute these names to different RP nodes, and each node is only responsible for a subset of the names. The size of the content name database on a RP node, or the number of content names, is determined by two factors: (1) the number of AV-pairs that are mapped onto this node, and (2) for each of these AV-pairs, the number of content names that contain that AV-pair.

Since AV-pairs in our system are the “keys” that are hashed, suppose there are N_d different AV-pairs and N_c number of resolver nodes, on average, each node would be responsible for $\frac{N_d}{N_c}$ AV-pairs. Suppose a resolver node, N_i , is responsible for k AV-pairs, av_1, av_2, \dots, av_k , then the total number of names, t_i , on this node is:

$$t_i = \sum_{j=1}^k N_{av_j},$$

where N_{av_j} is the number of names that contain av_j .

6.1.4 System scaling property

When load is balanced, it is relatively straightforward to show the system’s scaling property. Corresponding to the three thresholds, T_{SD} , T_{reg} , and T_q , the system may reach its capacity when on each resolver node, the number of names, the registration rate or the query rate reaches threshold.

The system can be evaluated from two angles. (1) Suppose the CDS system has a fixed number of RP nodes, N_c , we look at how many content names the system can support, how much registration rate and query rate the system can sustain. (2) Given a fixed load, e.g., the maximum number of content names, registration and query rate, we look at how many RP nodes we need to satisfy this load.

For content names, we have the following condition:

$$T_{SD} \cdot N_c \geq N_s \cdot n.$$

On the left hand side, $T_{SD} \cdot N_c$ is the maximum number of names the CDS can support. On the right hand side, N_s is the number of names that the system receives, since each name must be registered n times, from the CDS's point of view, it must store $N_s \cdot n$ names. Directly from above, we have:

$$N_c \geq \frac{N_s \cdot n}{T_{SD}},$$

and

$$N_s \leq \frac{T_{SD} \cdot N_c}{n}.$$

As an example, suppose $T_{SD} = 10,000$ and $n = 20$. To support $N_s = 100,000$ names, the system needs at least $N_c = 200$ nodes. Interpreted the other way, a CDS that has $N_c = 200$ nodes can support up to $N_s = 100,000$ names.

Similarly, given a fixed N_c , the maximum registration rate, R , that the CDS can support is

$$R = \frac{T_{reg} \cdot N_c}{n}.$$

And the maximum query rate, Q , is

$$Q = T_q \cdot N_c.$$

6.1.5 A numerical example

In the Highway Monitoring Service (HMS) example, suppose there are 100,000 (N_s) cameras, and on average each camera is described by 20 (n) AV-pairs. Suppose among the 100,000 cameras, there are collectively 10,000 (N_d) different AV-pairs. Each camera must update its information at a certain rate, e.g., once every 100 seconds.

Consider the case where each camera is connecting to a computer. These computers form the CDS network, and each computer can act as a resolver. The number of resolver nodes is 100,000 (N_c). Since $N_c \gg N_d$, each resolver node will be responsible for at most 1 AV-pair, and only 10,000 ($= N_d$, assuming no hash collision) nodes will be receiving registrations and queries. Assuming an even distribution of AV-pairs in names, on average each node who receives registrations will be responsible for 200 names ($= \frac{N_s \cdot n}{N_d}$).

In comparison to some basic schemes, our scheme has clear advantages: in a flooding scheme, each node would host all the 100,000 names, and receive updates from each of these names; in the broadcast search scheme, each query must reach each of the 100,000 nodes; and in the centralized scheme, the bandwidth requirement at the central site can easily be on the order of gigabits per second.

6.2 System properties with LBM

When the load to the system is imbalance, or increases, load balancing matrices will be deployed for popular AV-pairs. As a result, a name that has n AV-pairs may have to be registered with more than n nodes, and a query may be sent to more than 1 node to get resolved. We examine how the system behaves in these situations.

6.2.1 Analysis of one matrix

To simplify our analysis, we assume that one node is responsible for only one AV-pair. (Note that this is a reasonable assumption when $N_c \gg N_d$, but it is generally not true when $N_c < N_d$, which we will consider in the future evaluation.)

With that assumption, the number of replicas (rows) for an AV-pair increases linearly as a step function with respect to the query rate for this AV-pair:

$$r_i = \left\lceil \frac{Q_{av_i}}{T_q} \right\rceil,$$

where r_i is the number of replicas, and Q_{av_i} is the system-wide query rate for the pair av_i . As an example, when $T_q = 1000$ queries/sec, 1 extra replica is needed for this AV-pair, when the system observed query rate increases by 1000 queries/sec.

The number of partitions for av_i , p_i , is a function of (1) N_{av_i} , the number of names that contain the AV-pair av_i , and (2) the registration rate of this AV-pair. For simplicity, we first consider them separately. Suppose the registration rate for this AV-pair is under threshold, and we again assume each node is responsible for only one AV-pair, p_i is simply determined by:

$$p_i = \left\lceil \frac{N_{av_i}}{T_{SD}} \right\rceil.$$

On the other hand, when the registration rate of this AV-pair is higher than a node's registration threshold, the number of partitions is:

$$p'_i = \left\lceil \frac{R_{av_i}}{T_{reg}} \right\rceil,$$

where R_{av_i} is the system-wide registration rate for the pair av_i . The number of partitions needed for the av_i is $p_i = \max(p_i, p'_i)$.

The total number of nodes needed for av_i , or the size of the matrix for av_i , M_i , is:

$$M_i = r_i \cdot p_i.$$

We believe in practice, the system typically will not have matrices that have both large r_i and p_i . With the query optimization mechanism, queries are sent to the matrices that have the fewest number of partitions. With less number of queries arriving at them, the matrices that have a lot of partitions do not need many replications (rows). With fewer replications, registrations can still be done quickly. On the other hand, at matrices that have fewer names or registrations, they may receive large amount of queries, which means more replicas are needed. However, the overhead associated with replicating these nodes is small since the databases on these nodes are smaller and they receive less frequent updates.

6.2.2 Registration and query cost

The size of the matrices affects the overhead of name registration and query resolution.

- Registration

The number of nodes who will receive the registration for name $SD : \{av_1, av_2, \dots, av_n\}$, increases from n to $\sum_{i=1}^n r_i$, where r_i is the number of rows that the matrix corresponding to av_i has.

- Query

Suppose the query is: $Q : \{av_1, av_2, \dots, av_m\}$ and the number of partitions for the matrix corresponding to av_i is p_i . In our system, client selects the AV-pair, av_k , where $p_k = \min(p_i)$, for $i = 1, m$. If $p_k = 1$, the query only needs to be sent to the node that is responsible for av_k , which is the same as in the no LBM case. When $p_k > 1$, the query must be sent to one node in each of the p_k partitions. Thus, the total number of nodes that will receive the query is p_k .

6.2.3 Load balanced

Load to the CDS system is balanced across all resolver nodes in the system to eliminate hot spots and ensure registration and query processing performance. The system does not reject registrations or queries when there are nodes in the system that are not fully utilized.

- Content names

Hash function ensures the even distribution of AV-pairs over resolver nodes. However this mechanism alone is not sufficient, since a skewed distribution of AV-pairs in names will cause some nodes receive significantly more names than other nodes.

When the number of names on a node reaches threshold, T_{SD} , the LBM mechanism discovers nodes in the system that may be still available by applying hash function with different parameters, and future names will be registered at these nodes. Thus the number of names on each node is maintained under the desired threshold while the number of names received by the system increases.

- Registration load

The proposed CDS system also balances the registration load to the system across all the nodes. The system automatically discovers and redirects registrations to nodes that can still accept registrations when one node reaches threshold T_{req} . Thus, the registration rate at any node will remain under its allowed threshold as the registration load to the system increases.

- Query load

Similarly, for the query load, as the query rate observed at a node approaches the threshold, T_q , new replicas will be utilized to share the queries. Query rate observed at a node is kept under the threshold.

Given that the load to the CDS system is spread around all the nodes in the system, from a resolver node's point of view, since the number of names it hosts, the registration rate and query rate are all maintained under thresholds, it can process registrations and queries that arrive at it efficiently.

6.2.4 A numerical example

Again we use the HMS as an example to illustrate how the CDS system may behave when LBM must be used.

We use a skewed distribution, Zipf distribution, to model the AV-pair distribution in names:

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

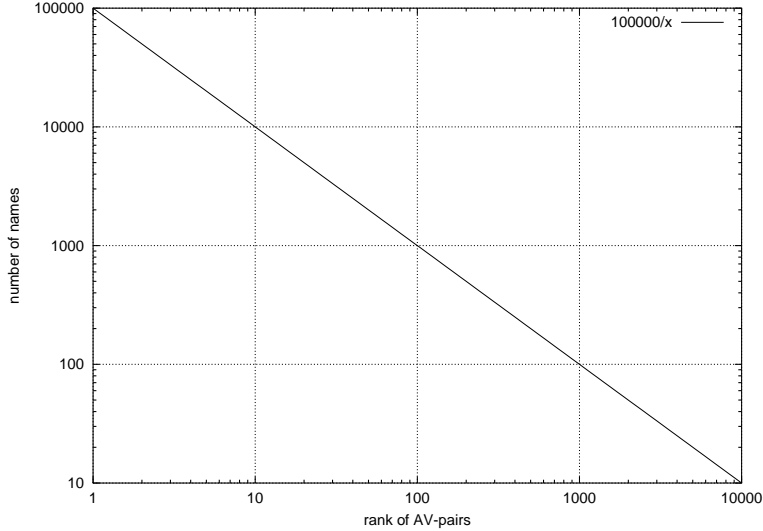


Figure 8: Number of names vs. AV-pair rank

for $i = 1, N_d$, where, i is the rank of AV-pair av_i in terms of occurring frequency in names. $i = 1$ corresponds to the AV-pair that is contained in the most number of names. k and α are some constants, and α is close to 1. For simplicity, we choose $k = 1, \alpha = 1$, with $N_s = 100,000$, and we have:

$$N_{av_i} = 100,000 \cdot \frac{1}{i}$$

Figure 8 shows the distribution on a log-log plot. Suppose in the system, the thresholds for each node are: $T_{SD} = 10,000$, $T_q = 1,000$ queries/sec, and $T_{reg} = 1,000$ reg/sec. Given this type of distribution, it is easy to see that the basic design will not work, since for example, the most popular AV-pair appears in 100,000 names, no single resolver can host that many names. This pair will need 10 partitions. The top 10 AV-pairs each is contained in more than 10,000 names, and will need more than 1 partitions. Also observe that 9000 out of the 10,000 AV-pairs have only less than 100 names, and nodes that are responsible for these pairs will have room to accommodate other AV-pair's names.

Similarly, without LBM, the system can only support the maximum registration rate of 1000 reg/sec, and query rate of 1000 queries/sec for any AV-pair. Whereas, with LBM, when rates grow beyond thresholds, the system will continue to work by spreading the load until all the nodes in the system become busy.

7 Proposed research

The proposed work includes a full implementation of the CDS system, a comprehensive evaluation of the system, system performance enhancement, and functionality improvement.

7.1 Implementation and evaluation plan

7.1.1 Implementation plan

Immediately upon the completion of the thesis proposal, I will start the implementation of the designed CDS system. I plan to implement two versions of the system: a simulator and an actual implementation. The simulator implementation will allow us to evaluate the different aspects of our system in a controlled environment. The actual implementation is expected to be run on the Internet. As we described earlier, the CDS system will be implemented as a generic software module so that other applications can be built on top of it. In this regard, we would also develop at least a couple of real applications, e.g., similar to the highway monitoring application we described, to test and evaluate the system running in the real world. We expect to release the software to the research community and the general public.

7.1.2 Evaluation plan

Since CDS is a distributed and dynamic system, and there are many parameters that can be tuned, the evaluation of such a system itself is a difficult task. By doing comprehensive evaluation, we would also gain experience in making design decisions.

The critical piece in evaluating such a system is to generate realistic work load. The following factors contribute to the work load: distribution of AV-pairs in names and queries, registration and query rate. To obtain work load, we plan to do the following:

- Generate synthetic work load based on known distributions, e.g., for the distribution of AV-pairs in names, we plan to use Zipf distribution as one possibility to model it.
- Use benchmarks. Benchmarks exist for evaluating similar applications, e.g, TPC benchmarks [5] for databases. Many of these benchmarks can be conveniently converted and used to test our system. For example, it is reasonable to assume that the query distribution in our system resembles the query pattern observed by a large database.
- Collect traces from other applications. There exist a wide spectrum of applications that are also CDS systems in various forms, e.g., Morpheus [17], Gnutella, search engines, and web caches. Functionality-wise, they may be different from our CDS, but the user load can be borrowed to evaluate our system. For example, we may translate keyword searches received by a search engine into AV-pair searches by assigning each keyword a proper attribute.

With the work load, we plan to evaluate the following aspects of the system.

- Scalability

Scalability is the first goal in our designed system. The analytical analysis we presented only provides a starting point, and we will examine the system's behavior as load increases and load balancing matrices are deployed.

- System dynamics

Issues associated with such a dynamic system, such as consistency, stability, and robustness, must be addressed. For example, data in replicas may be inconsistent with each other in transient states, which will affect the correctness of query results. System stability is another concern, since the matrices are expanding and compacting depending on the load status. The system must avoid oscillation and ensure fast convergence. Further, the system must handle

node crashes properly. For example, we must evaluate the impact of the “head node” of a matrix going down or being overloaded.

- System heterogeneity

The analysis we presented does not take into account the fact that nodes are heterogeneous in network connection, computing power and storage space. To deploy CDS in the real world, we must take these factors into consideration and see how the system behaves in a heterogeneous environment.

- Overlay network vs. physical network

We will explore the relationship between the overlay network and the physical network. Since our CDS layer is decoupled from the underlying overlay management layer, we would like to see the effects of different overlay mechanisms on the content discovery system.

7.2 System improvement

We plan to extend our system in several directions.

7.2.1 Specialized resolvers

The basic protocol consists of features that allow clients to do query optimization to some degree. Clients first retrieve the size of each matrix, and then send queries to the smallest matrix. Selecting the smallest matrix reduces the query resolution time. This scheme is beneficial if there exist small matrices corresponding to some AV-pairs in the query. However it does not help if all the involved matrices are large. We plan to improve the system’s performance when this is the case.

As an example, a query may be: $Q : \{ \text{device} = \text{digital camera}, \text{location} = \text{Wean 7110} \}$. The two matrices corresponding to $\{ \text{device} = \text{digital camera} \}$ and $\{ \text{location} = \text{Wean 7110} \}$ are both large as there are lots of digital cameras in the system, and there are many devices in Wean 7110. But there are only 2 devices that match both of these criteria. The idea is to store these two devices in a specialized node, and queries that contain these two AV-pairs can go directly there without having to go through either one of the big matrices. The benefits include: (1) reduced load on big matrices; (Since they are big, it is likely that they get more registrations and queries.) and (2) reduced resolution time for queries. If the queries hit the specialized nodes, they can be resolved faster.

Registering at nodes corresponding to all possible combinations of the AV-pairs in one name would provide a solution to this problem. However this is not practical when the number of AV-pairs in the name is large. As we discussed earlier, when the name has 10 AV-pairs, there will be $2^{10} - 1 = 1023$ different combination of AV-pairs.

If the system has some knowledge, either based on statistics or *a priori*, on the query pattern, then it may be possible to actively combine some AV-pairs and register them together. For example, if we know that queries always contain two or more AV-pairs, it makes sense to actively register all the combinations of two AV-pairs together instead of registering each individual AV-pair.

The set of names corresponding to some combination of AV-pairs may be small, as in the example above. However, if no queries are interested in this combination, there would be no point of combining them and registering them at a separate node. In other words, whether a “combination node”, or a specialized resolver, should be used should be dependent on the query load.

Based on the above observations, we now sketch out an algorithm that dynamically introduces specialized resolvers. As discussed in the basic system, a node monitors the rate of queries. We

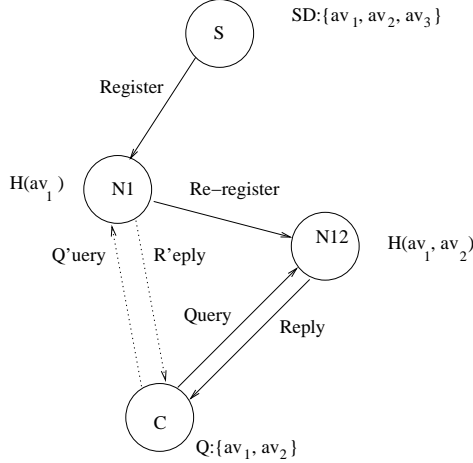


Figure 9: Register and query with specialized node

extend this scheme such that the node not only monitors the overall query rate, but also monitors the rate of each individual AV-pair being queried. For example, for the node that corresponds to av_1 , it maintains a counter for each additional AV-pair in the query, e.g., for query $Q : \{av_1, av_2, av_3\}$, the node would increase the counter for av_2 and the counter for av_3 . It computes the rate of each AV-pair being queried. The node declares a particular AV-pair “popular” if its rate of being queried reaches a threshold. For example, if many queries contain $\{av_1, av_2\}$, the node will decide that $\{av_1, av_2\}$ is a popular query. From then on, when future registrations (including refreshments) that contain both av_1 and av_2 arrive at this node, in addition to accepting them locally, the node will also send them to a new node that corresponds to the hash of $\{av_1, av_2\}$ together, $N_{(1,2)} = \mathcal{H}(av_1, av_2)$ (Figure 9).

In this scheme, clients with queries that include $\{av_1, av_2\}$ would send queries to node $N_{(1,2)}$ first. If it does not exist, clients can fall back to the original scheme by sending queries to the first level nodes.

A specialized node may remove itself under the following conditions: when it observes low query rate, or when the amount of names on it drops to a certain level. The master node may probe the specialized node to retrieve its observed query rate. If it is low, the master node may decide not to forward new names to it and this will eventually lead to the removal of the node. A specialized node corresponding to 2 AV-pairs may further create more specialized node that corresponds to 3 AV-pairs.

7.2.2 Improve search efficiency within an LBM

The problem is that if the matrix’s P-dimension is large, sending queries from the client to all the columns in the matrix may create problems. For example, the client may experience “reply implosion”, when all the nodes reply with matched contents. We explore the following mechanism to remedy this aspect.

We organize the columns in a matrix into a set of logical trees. Suppose there are p columns in the matrix. Nodes in the first column are root nodes, and for nodes in column i , their children are nodes in column $2i$ and $2i + 1$. The node in one column picks a random node from its children columns as its children. For example, node $N^{(1,r)}$, may pick a random r_2 and r_3 from 1 to r , to determine its children, $N^{(1,r_2)}, N^{(1,r_3)}$.

In this scheme, a client sends its query to a node in the first column, which is the root node of

a search tree. This node then sends the query to its two children, and the children nodes forward along, until the query reaches the leaf nodes. At that time, the leaf nodes resolve the query and send their results to their parents. When a node receives the resolutions from its children, it will resolve the query locally and add its results to the received results and forward to its parent. This will go on until all the results reach the root node. The root node then sends the summary of the results to the client (Figure 10).

Some further optimization can be done to the above basic scheme. For example, a node may terminate the propagation of queries and resolutions “early” when enough results have been collected. This is particularly useful for queries that may have a large set of matches.

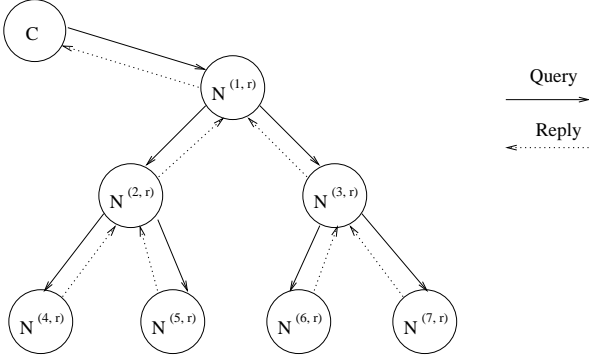


Figure 10: Search tree within a matrix

7.2.3 Support range search

Hashing AV-pair together fundamentally makes range search difficult. In our design, to resolve a query $Q : \{a_1 > v_1, a_2 = v_2\}$, since there is no node in the system that corresponds to the hash of $\{a_1 > v_1\}$, the client must hash $\{a_2 = v_2\}$, and then sends the query to the node that corresponds to $\{a_2 = v_2\}$. However system can not handle the case in which all the AV-pairs in the query are inequalities.

The reason that hashing AV-pair together does not work is because the resulting nodes do not know each other, even when they share the same attribute. For example, the node $N_1 = \mathcal{H}(a_1 = v_1)$, is not related to $N'_1 = \mathcal{H}(a_1 = v'_1)$ in any way.

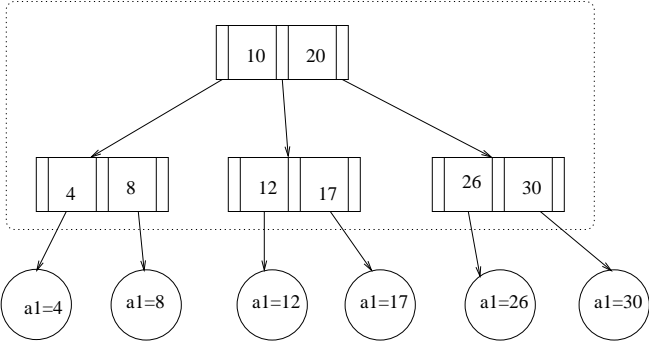


Figure 11: Example B-tree for range search

There are different ways of making the hash-based scheme to work for range search. We consider the following scheme. We introduce an auxiliary indexing data structure to link nodes that share

the same attribute together. We use a B-tree (or B+ tree) for this purpose, and the B-tree is stored on one node that corresponds to the hash of the attribute that allows inequality tests.

Consider the attribute a_1 . We use $N = \mathcal{H}(a_1)$ to store the data structure. As in the basic system, the registration message of a new service, e.g., $SD : \{a_1 = v_1\}$, is sent to node N_1 , which corresponds to the hash of AV-pair, $a_1 = v_1$. In addition to this, in the new scheme, the registration is also sent to Node N . Node N uses the value v_1 as the key to insert the node ID, N_1 , in this case, into the B-tree it maintains. For another name that has the same value v_1 , it will be registered at N_1 , but at N , no action will take place, since this value has already been inserted before.

Exact queries that contain a_1 will still be hashed and sent to the nodes normally. Queries that contain range search, e.g., $Q : \{v_1 < a_1 < v'_1, a_2 = v_2\}$, will be sent to N . Node N will use v_1, v'_1 as search keys and then return the client all the corresponding node IDs. Client will then send its query to those IDs to get the query resolved (Figure 11).

We plan to address problems brought in by this approach, e.g., the node that is holding the data structure may become a bottleneck and point-of-failure.

7.2.4 CDS as a distributed database

We plan to extend our CDS from a simple name resolution service to a system that allows more sophisticated database-type of operations. For example, the CDS should be able to not only return the address of a matched device, but also some other aspects of the device depending on the need of the client. Operations like “select” or “projection” will be incorporated into the CDS.

8 Timeline

Tasks	Summer'02	Fall'02	Spring'03	Summer'03	Fall'03
Basic CDS simulator implementation	<----->				
Incorporate load balancing mechanisms	<----->				
Synthetic load and benchmark evaluation	<----->				
Actual implementation	<----->				
Collect traces and comprehensive evaluation	<----->				
System improvement	<----->				
Internet evaluation	<----->				
Writing	<----->				

9 Expected Contributions

I expect the following contributions upon the completion of the thesis.

- From a system’s point of view, we will demonstrate via implementation and evaluation that

the proposed CDS provides an important solution to the distributed content discovery problem, which has become one of the most important applications on the global Internet.

- From an architecture's point of view, we will show that content discovery is a critical layer in building a wide range of distributed applications.
- From a software's point of view, we will release our CDS software and the applications associated with it to the research community and general public. We hope to see it being used in building real Internet applications.

Acknowledgement

I would like to thank my advisor, Peter Steenkiste, for his guidance and support. I would like to thank my colleague and friend, Syed Umair Ahmed Shah, for the many valuable discussions. I would like to dedicate this work to my parents, my brothers in China and my wonderful wife, Shuheng Zhou, for their love, care, encouragement, inspiration and belief in me.

References

- [1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The Design and Implementation of an Intentional Naming System. In *Proceedings of SOSP 1999*, Kiawah Island, SC, December 1999.
- [2] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching Events in a Content-based Subscription System. In *Principles of Distributed Computing*, 1999.
- [3] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.
- [4] M. Castro, P. Druschel, A-M. Kermarrec, and A. Rowstron. SCRIBE: A Large-scale and Decentralised Publish-subscribe Infrastructure. Submitted, September 2001.
- [5] Transaction Processing Performance Council. <http://www.tpc.org/>.
- [6] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An Architecture for a Secure Service Discovery Service. In *Proceedings of Mobicom 99*, Seattle, WA, August 1999.
- [7] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of SOSP 2001*, Banff, Canada, October 2001.
- [8] S. Deering. Multicast Routing in Internetworks and Extended LANs. In *Proceedings of SIGCOMM 1988*, 1988.
- [9] 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.
- [10] Elvin. <http://elvin.dstc.edu.au/>.
- [11] Freenet. <http://freenet.sourceforge.net/>.

- [12] Gnutella. <http://gnutella.wego.com/>.
- [13] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol. IETF, RFC 2165, November 1998.
- [14] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-based Peer-to-peer Networks. In *Proceedings of IPTPS'02*, March 2002.
- [15] Google Inc. <http://www.google.com/>.
- [16] P. Mockapetris. Domain Names - Concepts and Facilities. IETF, RFC 1034, November 1987.
- [17] Morpheus. <http://www.morpheus-os.com/>.
- [18] Napster. <http://www.napster.com/>.
- [19] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proceedings of SIGCOMM 2001*, pages 161–172, San diego, CA, August 2001.
- [20] P. Reynolds and A. Vahdat. Efficient Peer-to-Peer Keyword Searching. Unpublished draft.
- [21] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. Submitted, May 2001.
- [22] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari 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.
- [23] D. Waitzman, C. Partridge, and S. E. Deering. Distance Vector Multicast Routing Protocol. IETF, RFC 1075, November 1988.
- [24] 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.
- [25] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, U. C. Berkeley, April 2001.