MapReduce Resource Discovery and Monitoring Using Self-Organizing Multicast Trees

Kyungyong Lee
ACIS Lab. Dept. of ECE
University of Florida
klee@acis.ufl.edu

Tae Woong Choi
ACIS Lab. Dept. of ECE
University of Florida
twchoi@ufl.edu

Arijit Ganguly
Amazon Web Service
Amazon
aganguly@gmail.com

P. Oscar Boykin
ACIS Lab. Dept. of ECE
University of Florida
boykin@acis.ufl.edu

Renato Figueiredo
ACIS Lab. Dept. of ECE
University of Florida
renato@acis.ufl.edu

ABSTRACT

Resource monitoring and discovery are important processes for building a large computing system. This paper presents a MapReduce-based resource query method, which runs on top of a structured Peer To Peer (P2P) network. A self-organizing bounded-broadcast method allows our system to query the entire network efficiently with the latency cost of $O(log^2(N))$, where N is a number of nodes. By using the concept of Map and Reduce functional programming model, our query system performs a matchmaking in each node with the local resource information, and the matching result is summarized and aggregated in a distributed fashion at each node of the bounded-broadcast tree during the reduce phase. Analysis and simulation results prove that our system is scalable with respect to the number of nodes. A performance comparison with SWORD, a DHT based resource query algorithm, shows that our system imposes less update overhead when the number of resource attributes increases while providing more timely matching result. MapReduce-based resource query system also supports new attribute addition without reconfiguring a previous state. Our MapReduce-based query system is currently deployed on PlanetLab while built upon the Brunet P2P network. To our best knowledge, our system is the first demonstrated implementation of MapReduce-based resource monitoring system that runs on top of a P2P network.

1. INTRODUCTION

Desktop grids[2][4] using commodity computers connected over a local or wide-area network have been proven to be excellent platforms for high-throughput computing while providing sharing CPU power, storage capacity, and applications. These systems should be scalable in case of installing new machines. It also has to be fault tolerant and secure in the presence of abnormal or malicious behavior of a user or churn. These systems must be able to distribute jobs evenly among nodes[1] to achieve maximum utilization and alleviate hot stops. Middleware solutions such as Condor[4] and Boinc[2] for managing distributed resources often rely on a centralized master server that has global view of available resources to make efficient scheduling decisions. Worker nodes publish their resource information to the central master node that matches a job requirement against available resource information to determine the most suitable node that can execute a job. The central master server approach often imposes significant administrative, uptime, and scalability issues, and the failure of the master can render all its managed resources unusable.

Distributed Hash Table (DHT)[23][19] offers a decentralized information system that scales to a large number of nodes, self-configuring in presence of node failures, and efficient object storage and retrieval based on a key. Resource discovery often involves querying resources using attributes (CPU speed, disk space, RAM, and etc) by their desired values, often involving range queries on attributes. Thus, DHTs are often inadequate for these requirements. SWORD[1] and Mercury[5] enhance the DHT to range and multi-attribute queries, but these systems cannot capture complex queries involving unstructured data, such as partial string matching or regular expression matching. To prevent hot spots where only a few nodes might end up storing bulk of information, they have to support a load balancing mechanism, which results in uneven node distribution. Additionally, those systems require nodes to periodically publish their resource information, which can impose significant network overheads even when no jobs are being submitted to the system. Furthermore, the retrieved resource information might be stale due to characteristics of DHT entry update period. Finally, the usability of these systems in an often Network Address Translation (NAT) constrained environment is unknown. In this paper we present a decentralized, self-organizing resource discovery system based on MapReduce that overcomes several of the above limitations of existing resource discovery systems. Our system stores only resource information extracted by condor_startd daemon[4] locally at a node, and leverages a structured P2P network to

---

[1]We use the terms, *node*, *peer*, *resource* and *machine* interchangeably.

create a multicast tree for efficient query dissemination and result aggregation over a large subset of nodes. A query to discover resources is propagated down the tree where each node performs a matchmaking using Condor Classified Advertisements (ClassAds)[20] with requirements defined at Map function - the outcome of which is a rank that represents its ability to meet the job requirements; and each node propagates the query down to its child nodes in the multicast tree. Once leaf nodes have computed their Map functions, the results are communicated up the tree. Aggregation happens at each intermediate node by computing a Reduce function over the results obtained from its children.

We present experimental results from a deployment of MapReduce based query system on PlanetLab[8] with a structured P2P network, Brunet[6]. We also compare the performance of our query system with SWORD[1] analytically and through simulations. The experiment results show that MapReduce-based query system completes querying 600 PlanetLab nodes within 15 seconds for 80% of queries. Our query system's characteristics and contributions are the following:

1. No resource information update traffic: Each query is resolved using local up-to-date resource information.
2. Bounded-broadcast trees provide an efficient way to disseminate queries and reduce response message size.
3. By using well-known Map and Reduce functional programming model, programmers can implement parallel resource monitoring jobs in a distributed system easily.

Based on our knowledge and survey, MapReduce-based query system is the first demonstrated implementation of MapReduce application deployed on a structured P2P network. The rest of this paper is organized as follows. Section 2 discusses the MapReduce-based query system architecture. Section 3 evaluates our query system and compares the performance with that of SWORD by simulation and analysis. Section 4 discusses related works. Section 5 summarizes and concludes this paper.

2. SYSTEM ARCHITECTURE
In this section, we describe the overall architecture of the MapReduce-based query system and the bounded-broadcast method. We will also discuss how we use Map and Reduce functions to handle query and response processing.

2.1 Bounded-broadcast
The key to put a resource discovery message is bounded-broadcast[7] that disseminates a message over a sub-region of a P2P network by leveraging a structured P2P system to create a self-organizing tree. Bounded-broadcast is currently implemented on top of Brunet[6], which implements a 1-d Kleinberg's small-world network[16]. Each node maintains two types of connection: (1) constant number of near connections to its left and right neighbors on the P2P ring, and (2) about $0.5 * log(N)$, where N is the number of nodes, far connections to random nodes in the P2P ring; such that the number of hops between nodes is $O(log^2(n))$.

A node is allocated a sub-region of the P2P ring over which to disseminate a message. The node then redistributes the
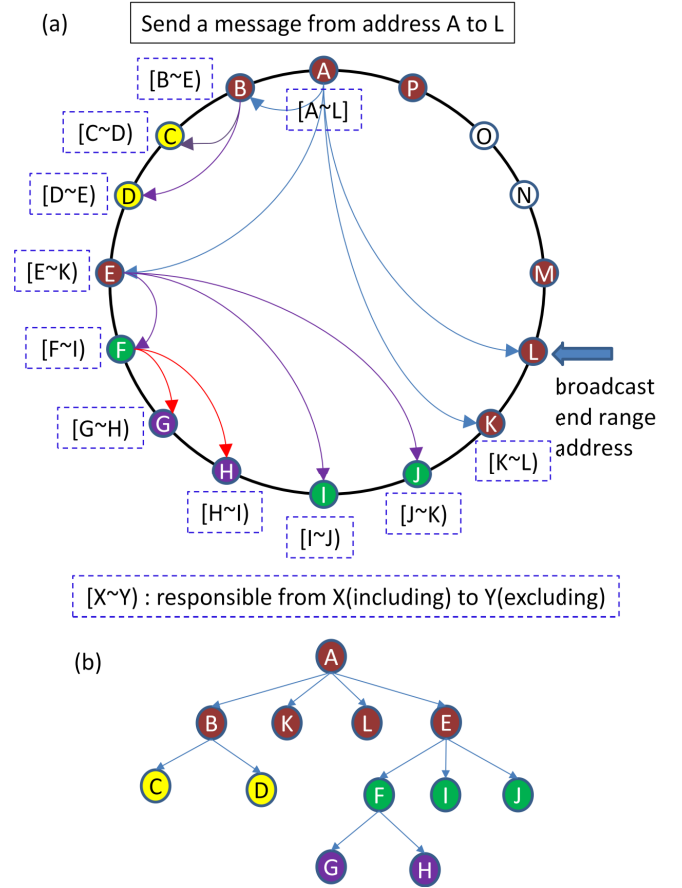
Figure 1: Bounded-broadcast message propagation-(a)Node A broadcasts a message from node A to L. Node A first sends the message to its neighbor nodes, B, E, K, and L after setting broadcast region appropriately. The message recipient node sends the message to its neighbors inside its broadcast region recursively. (b)Generated bounded-broadcast tree after disseminating the message.

message to its neighbor nodes inside its allocated sub-region while allocating new sub-regions to the neighbor nodes. This process continues until the message is disseminated over the entire sub-region. For example, to broadcast a message over the sub-region[A,L] in Figure 1, a message initiator issues a broadcast command to node A with sub-region information, [A,L]. Node A recognizes node B, P, E, K, L, and M as neighbors. The node A then broadcast messages to its neighbors, by specifying broadcast range as [B E), [E K), [K L), [L] to node B, E, K, and L, respectively. After receiving the message, node B, E, K, and L broadcast the message only to their neighbor nodes inside the specified sub-region recursively. After disseminating the broadcast message until the leaf node, a graph like Figure 1(b) is formed.

If a node initiating a message does not lie within the bounded-broadcast domain (sub-region of the P2P ring), the message is first routed to a center node inside the bounded-broadcast region by using greedy routing. The bounded-broadcast method is responsible for distributing arbitrary queries to

nodes in a P2P pool and dealing with lagging nodes whose response time takes longer than the others. The latency cost of bounded-broadcast is $O(\log^2(N))$[7], which is larger than a DHT system $O(\log(N))$[23], but smaller than a naive flooding based broadcast method $O(N)$.

2.2 MapReduce

Map and reduce functions are popular paradigms in functional programming languages (Ruby, Python, LISP). A map function transforms each element (x_i) in a collection $(x_1, x_2...x_n)$ into a different key element (y_i), thus creating a collection of intermediate elements $(y_1, y_2...y_n)$. A reduce function computes an aggregation (z) over this collection. Based on the concept of map and reduce functional programming model, software frameworks (Hadoop[11] and Google[9]) have been developed to parallelize computations on large datasets efficiently. Map function usually works on (Key/Value) pairs to create intermediate (Key/Value) results. Reduce functions work on intermediate (Key/Value) results while aggregating intermediate values associated with the same intermediate keys. The framework distributes map and reduce tasks among nodes in a cluster to enable parallel processing, while the users only have to specify the appropriate map and reduce functions associated with their computation without having to worry about details of distributed parallel job execution.

In our MapReduce-based query system, we define the Map function as checking requirement matching and calculating a rank value. The Reduce function is defined as aggregating and ordering the Map result based on the matching result and rank value. These are presented in detail next section.

2.3 MapReduce Query System Architecture

MapReduce-based query system is divided into 5 modules. They are P2P network module, MapReduce core, Map and Reduce function, local resource information monitor, and matchmaking module.

2.3.1 P2P Network Module

The P2P network module is responsible for handling new node join and departure, connection management with neighbor nodes, and routing messages. The current version of MapReduce query system is developed on top of Brunet[6]. However, our system can be deployed on the other P2P platforms, such as Chord[23], CAN[18], or Pastry[19].

2.3.2 MapReduce Core Module

The MapReduce core module takes responsibility for distributing Map and Reduce functions using bounded-broadcast. When a user initiates a MapReduce task, the request is conveyed to the MapReduce core module through the underlying P2P network's Remote Procedure Call(RPC) module. The MapReduce core module checks the Map argument, the Reduce argument, and the broadcast region argument. The Map and Reduce argument will be passed to the Map and Reduce function, respectively. The broadcast region argument describes a bounded-broadcast region that the node is responsible for. The MapReduce core module disseminates the task to nodes which reside under its responsible region after manipulating broadcast region argument appropriately.

A user has to define own Map and Reduce functions associated with his needs. After a node processes the Map task, the node returns the result to the Reduce function of itself. The Reduce function aggregates its own Map result and child nodes' Reduce results. After completing the Reduce function, the node returns the result to the parent node's reduce function. For the resource discovering purpose, Map and Reduce functions are defined as follows:

Map function: In a Map function, a resource requirement and a rank criteria are delivered as a Map argument . The following example illustrates a Map argument based on Conodr ClassAd.[2]

$Requirement$=(Memory>2048) && (KeyboardIdle>300) && (SoftwareInstalled.Contains("Matlab")) (1)
$Rank$=(Memory)+(KeyboardIdle*10) (2)

State (1) describes the resource requirement. It means that the target resource's memory has to be bigger than 2GB and the keyboard idle time is more than 300 seconds. The target machine also has to contain string *Matlab* in a SoftwareInstalled attribute. When a node receives a resource matching request, it first checks whether it satisfies the requirement or not. If it does, it calculates the rank value whose purpose is to order candidate nodes. As users' demands change, it is highly likely that new attributes are added to the original resource attribute set. In this situation, using ClassAd allows users to add new attributes to the original resource attribute set easily. Though a DHT-based query system usually supports adding new attributes to the original attribute set, it may result in performance degradation or rearranging whole DHT entries for new attributes. Statement (2) shows the rank criteria. Every node which satisfies the requirement calculates its rank value using a given rank criteria, and this value is used to select optimal candidates. As we can see from this example, a user can easily specify its requirement and rank value arbitrary. For a matchmaking purpose, we use Condor ClassAd[20] to exploit its regular expression and arbitrary matching support. Using Condor ClassAd allows us ordering requirement satisfying nodes with more flexibility than related approaches such as [15], which supports only a static ordering method(i.e., based on an average queue size and CPU speed)

Reduce function: In the Reduce function, the number of desired nodes and a rank ordering method are delivered as an argument. Assuming that a node sends MapReduce tasks to n nodes using bounded-broadcast, n reduce results will be returned from child nodes, and one Map result will be returned from itself. The Reduce function will summarize those results based on the number of desired nodes, an ordering method, and rank values. The number of desired nodes specifies how many nodes the user wants to find in the pool, and an ordering method specifies a rank value alignment method. If the number of desired nodes is k, and the ordering method is *ascending*, the rank value is aligned in the ascending order, and top k rank value nodes are returned from the Reduce task. As with Map, Reduce is processed at each node in a tree, and results are propagated back through the tree.

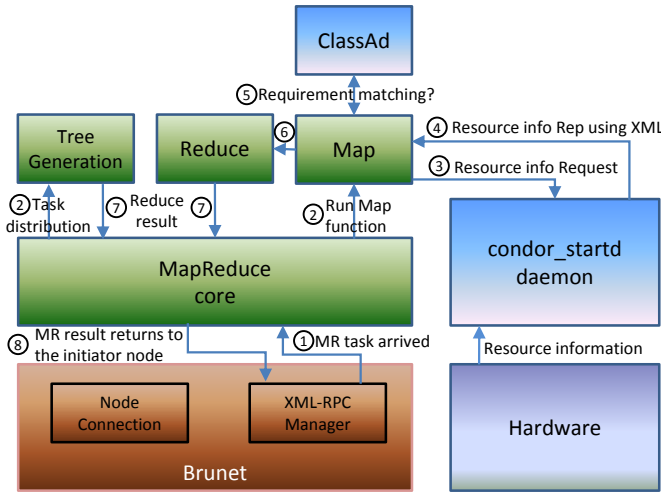[2]We modified the ClassAd syntax in this example to increase readability.

Figure 2: MapReduce query system architecture

1. When a MapReduce task arrives to a node, it is delivered to the MapReduce core module first.
2. The MapReduce core module redistributes the task to nodes inside allocated sub-region and waits results from the child nodes. MapReduce core also initiates a local Map function.
3. The Map function requests local resource information to *condor_startd* daemon.
4. The *condor_startd* returns up-to-date resource information expressed as an XML.
5. The Map function converts the returned XML-resource information to a ClassAd object and checks requirement matching using ClassAd.
6. A Reduce function is performed using the local node's Map result.
7. The local node's Reduce result and child nodes' Reduce results are returned to the MapReduce core, and results are summarized.
8. The MapReduce core module returns the aggregated Reduce result to the MapReduce task initiating node.

2.3.3 Local Resource Information Monitor Module

There are several ways to gather resource information. The naive way would be using */proc* file system on a Linux machine. By using the *proc* file system, we can get a kernel, process, CPU, and memory information. Though this method is simple and easy to implement, it provides a limited number of information. In addition, this information is not accessible on operating systems other than Linux. While several monitoring systems can be integrated in our framework, we use Condor[4] *condor_startd* daemon to monitor and gather resource information. This daemon periodically sends a machine's ClassAd[20] to a *condor_collector* daemon. The machine's ClassAd is used to evaluate matchmaking by a *condor_negotiator*. Condor allows running each condor daemon in a standalone mode without installing an entire Condor pool. *Condor_startd* daemon provides summarized metrics, such as load average and total idle time, as well as basic information provided by an operating system.

2.3.4 Matchmaking Module

We use Condor ClassAd[20] to check whether a resource's capacity satisfies an user's requirement or not. With a connection to the *condor_startd* daemon, ClassAd provides an interface to interact with the resource information provided by *condor_startd*. After getting local resource information through the condor_startd as an XML file, a ClassAd library converts the XML file into a ClassAd object, which is suitable for a matching purpose. Two ClassAd objects(i.e., resource information ClassAd and job requirement ClassAd) match if both ClassAds contain the requirement field, and the requirement value evaluates to *true* to the other ClassAd. For example, *requirement=other.NumberOfCPU>2* will match with a resource which has more than two processors. Because we use the ClassAd library intact, our query system follows most of the ClassAd library characteristics.(e.g., supporting range query, regular expression match, and etc.)

Using five modules described from section 2.3.1 to 2.3.4, Figure 2 architecture is formed. MapReduce-based resource discovery processing step is as follow:

2.4 Comparison with Data-Processing MapReduce Framework

The goal of data processing systems like Google MapReduce[9] and Hadoop[11] is sharing computing power in order to process large dataset. Due to large input and output dataset transfer, the system is appropriate in a LAN environment. Our MapReduce-based query system, on the other hand, targets for monitoring and querying nodes in a WAN environment. Data processing MapReduce systems run a central manager which is responsible for assigning map and reduce tasks and dealing with a worker failure. On the contrary, our MapReduce-based query system has no central manager node. Instead, a self-organizing bounded-broadcast tree is responsible for committing map and reduce tasks.

For efficient data processing, Hadoop MapReduce detects a lagging node based on a progress score. If a node's progress score is less than a threshold value, which is decided based on the average Map and Reduce task execution time, the node is marked as a straggler. The node's job is reassigned by a central manager. The Late scheduler[26] detects lagging nodes that will finish the farthest into the future based on *Remaining Job Portion/Progress Rate*, which considers both how fast the node is processing the task and how much amounts of work remain. In Hadoop and Late algorithm a central manager is obligated to monitor Map and Reduce task processing nodes. In MapReduce-based query system RPC timeout will distinguish lagging nodes. If a parent node detects a RPC timeout from one of its child nodes, the parent node would prune the retarding node from a bounded-broadcast tree.

Data processing MapReduce consumes much network bandwidths to transfer input and output data, and they need fast data transmission for an efficient job processing. Oppositely, MapReduce-based query system consumes small amounts of network bandwidth, because a Map argument(i.e., query requirement) is usually less than several-hundreds bytes, and a Reduce result is aggregated at each node in a broadcast tree. The hierarchical information aggregation method, which applies to the Reduce result accumulation, provides scalability in a distributed information management system[24][25].

(a) Mapped Key   = 0010 0111 1001 1100
(b) Begin Address = 0010 0100 0000 0000
(c) End Address   = 0010 1000 0000 1111

          attribute   value   random
          indexing   indexing   bits

Figure 3: SWORD DHT key mapping for free disk space: (a)key mapping of 121GB free disk space (b)Query begin address for 64GB free disk space (c)Query end address for 128GB free disk space

3. EVALUATION

In this section, we evaluate MapReduce-based query system in a decentralized and heterogeneous environment using an analytical method and a simulation. To address the feasibility of MapReduce-based query system in the real-world, we deployed the system on PlanetLab. To compare our MapReduce-based query approach against a DHT based range query algorithm, we evaluate SWORD[1] analytically and via a simulation.

3.1 SWORD

SWORD provides a resource discovery system that allows end users to locate a subset of nodes which satisfy users' requirement in a pool, using DHT. The original DHT allows keeping a <Key, Value> pair on distributed nodes. Each key is hashed to a node ID, and the appropriate node, whose node ID is the closest to the hashed key value clockwise direction on a P2P ring structure[23][19], keeps the <key, value> pair. However, mapping a resource attribute name(e.g.,free memory) to a DHT entry key and associating an attribute value(e.g.,2GB) to a DHT entry value cannot support a resource discovery based on a value range. To overcome this limitation, SWORD maps an attribute name and a value to a DHT entry key. Among n bits of a DHT key, it allocates m bits for attribute indexing, where $n > m$. Of the remaining $n - m$ bits, k bits are designated as value expression bits, and a random value is filled in remaining $n - m - k$ bits. The mapping is performed once per an attribute for every information update event, and the entire resource information is conveyed to the calculated DHT key as a value. Figure 3 shows SWORD DHT key mappings. In this example, n is 16 bits, m is 4its, and k is 8 bits. Let's assume that value 0010 indicates the free disk space attribute. A node whose free disk space is 121GB sets its free disk attribute key as in Figure 3(a). To find nodes whose free disk space is between 64GB and 128GB, SWORD first decides the query begin address by setting attribute bits as 0100, value bits as 64GB, and random bits as all 0x0s, shown in Figure 3(b). For the query end address, it sets attribute index bits as 0100, value bits as 128GB, and random bits as all 0xFs as in Figure 3(c). Nodes inside between the begin address and end address will satisfy the query. If multiple attributes need to be considered, one representative attribute is selected randomly, and the query range is determined based on the requirement of the randomly selected representative attribute. Other than a naive resource matching, SWORD provides an *optimizer* module to select optimal resources among multiple candidate nodes.

3.2 Performance Analysis

In this section, we are going to compare the performance of MapReduce-based query system and SWORD analytically. Our analysis focuses on the (1) Number of Visited Nodes to Complete a Query-N_Q, (2) Query Latency-L_Q, (3) Query Bandwidth-B_Q, and (4) Resource Information Update Bandwidth-B_U. We express the total number of nodes in a pool as N, the number of published attributes as A_N, the size of each attribute as A_S, and the number of attribute and value indexing bits as I_A, and I_V, respectively. In this analysis, we assume that all nodes are evenly distributed.

3.2.1 Number of Visited Nodes to Complete a Query

MapReduce-based query system visits all nodes in a pool to complete a query, which means that the number of visited node is N. SWORD can optimize this value by manipulating the number of bits for an attribute and a value indexing bits shown in Figure 3. As those bits increase by 1 bit, the query region decreases in half. Thus, the number of visited nodes to complete a query is $\frac{N}{2^{I_A + i_v}}$, where $0 < i_v < I_V$. i_v means the number of same bit value between query begin and end value. For example, if a query wants to find machines whose attribute value lies between 0x0000 and 0x00FF, i_v is 8.

3.2.2 Query Latency

The latency of a query depends on both the number of visited nodes to complete a query and query propagating method(i.e., bounded-broadcast). According to [7], latency of bounded-broadcast query dissemination follows $O(log^2(N))$. Assuming SWORD propagates query using bounded-broadcast, the query latency will be $O(log^2(\frac{N}{2^{I_A + i_v}}))$. Otherwise, the latency of MapReduce-based query is $O(log^2(N))$.

3.2.3 Bandwidth Usage For a Query

Let's assume that a query message size is S_Q, and a response message size is S_R. Using bounded-broadcast, a message is routed only to 1-hop neighbors, so bandwidth consumption between a parent and a child node is $S_Q + S_R$. Thus, the bandwidth usage to complete a query is $(N_Q - 1) * (S_Q + S_R)$. In case of MapReduce-based query system, Reduce results from child nodes are aggregated whenever a parent node processes a Reduce function, so S_R of MapReduce-based query system does not increase linearly as Reduce results propagate through the bounded-broadcast tree.

As we can see from above three metrics, the query performance is closely related to the *Number of Visited Node to Complete a Query*. By using bounded-broadcast, we can complete a query at the cost of $O(log^2(N_Q))$, which is smaller than a flooding based broadcast cost($O(N_Q)$).

3.2.4 Cost for Resource Information Update

SWORD has to publish a node's resource information to remote nodes in order to enable distributed matchmaking. Every node has to update its resource information, whose size is $A_N * A_S$(neglecting serialization overhead), to A_N nodes, because any nodes mapped to the hashed key have to process matchmaking. In addition, each DHT entry has to be routed to a proper hashed key node, which takes $O(log(N))$ hops[23][19] or $O(\frac{1}{k}log^2(N))$,where k is number of short-cut connection[6]. Accordingly, bandwidth for a resource infor-
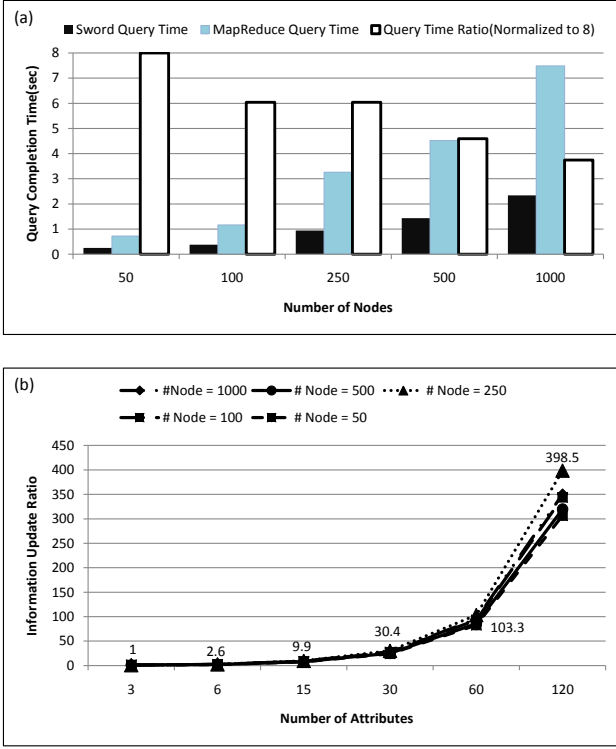
Figure 4: MapReduce-based query system and SWORD simulation result (a)Query completion time. Query time ratio = (Query complete time/Number of nodes)*1.6. (b)SWORD resource information update size. Ratio=(BW consumption)/(BW consumption when the number of attribute is 3)

mation update per one update period is:

$$N * A_N * (A_N * A_S) * (Number\ of\ Hops)$$

It shows that the bandwidth consumption is $O(A_N^2)$, and $O(N * log(N))$, which is not scalable in case of increasing number of resource attributes.

3.2.5 Query Result Correctness

A resource information in a DHT entry is not usually up-to-date. This may not hurt SWORD performance if a query requirement is static information(e.g., operating system). When it comes to dynamic information(e.g., current CPU load), the stale information is highly likely to be useless. SWORD can make an update period shorter to keep resource information fresher, but it will result in more frequent DHT entry update and more bandwidth consumptions.

3.3 Simulation Results

To prove correctness of analysis and compare the performance of MapReduce-based query system and SWORD, we implemented an event-driven single-threaded simulator, which uses Brunet[6] routing, DHT, and node management. We used King data set[12] to set network latency between nodes and Archer[10] in order to run simulations on a decentralized

Table 1: Bandwidth Usage Per One Query

# Nodes	Query BW(Bytes)		Ratio
	Sword	MapReduce	
50	382	8,444	0.045
100	711	17,069	0.041
250	998	44,087	0.022
500	1,281	89,878	0.014
1000	1,841	180,428	0.01

and distributed computing resources efficiently. On the simulation, we ran 1-simulated hour after a P2P pool is formed to make the pool stable. And then, target operation(i.e., resource query or DHT resource information update) is initiated at each node for 2-simulated hours. For MapReduce-based query system, each node initiates a query every 5 minutes while setting a query range as the whole network. In case of SWORD, we allocated 4 bits for attribute indexing, and a value field is filled with two randomly generated integer or double value to specify a desired region. We set a Time To Live(TTL) of SWORD's DHT entry as 30 minutes, and the entry update is performed every 15 minutes. After running 10 simulations with different parameters, we calculated an average value.

3.3.1 Query Latency

Figure 4(a) shows query latency. As we can see from the figure, the query latency grows as the number of nodes increases. To highlight a relationship between a query latency and a number of total nodes in the pool, we added a query time ratio. The ratio is calculated as $\frac{Query\ Complete\ Time}{Number\ of\ Nodes}$ and multiplied by a constant value to make the value fit for the graph. The ratio value decreases as the number of nodes increases, which means that the query latency is less than $O(N)$. Though SWORD shows less query latency than MapReduce-based query system, the effect of SWORD's smaller query region is not so remarkable. When we set the attribute indexing size to 4 bits, SWORD query region is about 0.3% of the entire network. However, the query latency decreased only about 30% of an entire network query latency.

3.3.2 Query Bandwidth Usage

Table 1 shows bandwidth consumption to complete a query. The *Ratio* is calculated as $\frac{SWORD Bandwidth}{MapReduce bandwidth}$. The ratio value decreases as the number of nodes increases, but it is still bigger than 0.3%, which is the fraction of SWORD's query region to the entire network. One possible reason is multiple hops to route a message to a node inside a query region. A probability of finding a node in a query region decreases when the number of nodes in the pool decreases, so it will take more hops to route a message to a node in the desired region. In our simulation, the number of nodes inside a query region is usually 0 when the number of total nodes is small. In case there is no node in the query region, SWORD should provide additional methods to query nodes which are located near the query region.

3.3.3 Resource Information Update Bandwidth

Figure 4(b) shows SWORD's resource information update bandwidth consumption when the number of published at-

tributes is 3, 6, 15, 30, 60, and 120. To show the effect of increased number of attributes, we normalized each bandwidth value to that of 3-attributes. Thus, n-attribute value's ratio is calculated as $\frac{n-attribute's\ bandwidth}{3-attribute's\ bandwidth}$. When the number of attribute is small, the increasing rate is not $O(A_s{}^2)$, where A_s means number of attributes, due to a resource information serialization overhead. As the serialization overhead effect becomes negligible, the rate follows $O(A_s{}^2)$

Based on the analysis and simulation results, we conclude that bounded-broadcast allows propagating queries to the entire P2P network eligible within a reasonable amount of time. Considering that propagating a query spends less bandwidths than publishing resource information, it is more desirable to propagate queries to the entire network than publishing resource information periodically. In addition, propagating queries to the entire network allows each node to process matchmaking using own fresh local resource information, which claims advantages over DHT-based query system where saved resource information might be stale.

3.4 PlanetLab Evaluation

Figure fig:plab shows MapReduce-based query system's query latency on PlanetLab. The test was performed on 31. January. 2010, and about 600 nodes were included in the experiment pool. For the experiment, 3 kinds of queries were performed 100 times each.

Query 1: Requirement=(Memory>1024) && (KeyboardIdle > 300) && (OpSys=="LINUX"). Rank = Memory + KeyboardIdle*10. Return 5 nodes whose rank is the highest.

Query 2: Requirement=no requirement. Rank=no rank value. Return randomly selected 200 nodes.

Query 3: Requirement=(PhysicalLocation.Longitude>0) && (PhysicalLocation.Latitude>0). Rank=CpuBusyTime. Return 5 nodes whose rank value is the lowest.

As we can see from the figure, 80% of queries completed in 15 seconds regardless of the query type. However, some queries took very long to complete, because our query system waited for a reply from lagging response nodes until the underlying P2P system issues a RPC timeout. Supporting a method to handle lagging nodes other than relying on a RPC timeout of underlying P2P network is our future project.

4. RELATED WORK

In this section, we will discuss related works on P2P network range query systems and cluster monitoring systems.

Range Query Systems on a P2P network: Kim et. al[15] and Artur et. al[3] discuss resource locating methods using a multiple dimension P2P network. Kim et. al[15] maps each resource attribute to one dimension in CAN[18]. For matchmaking, a requirement conforming zone is created based on the criteria described in a query, and nodes inside the requirement satisfying region are candidate ones for the requirement. Artur et. al[3] converts multiple dimension spaces into one dimensional ring space. Using the correlation between multiple dimensions and an attribute value, the resource matchmaking is performed. Above methods need
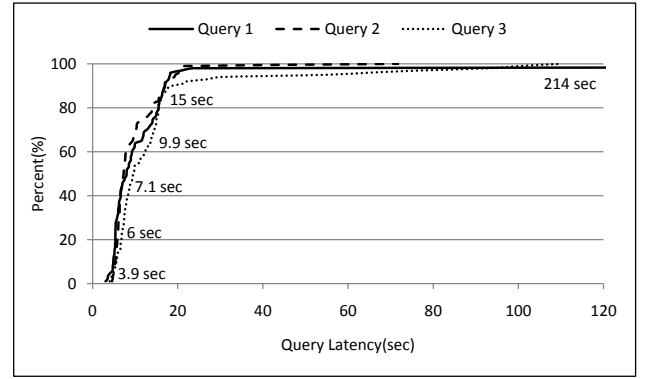
Figure 5: Cumulative distribution of query latency on the PlanetLab with about 600 nodes

a local node's information update to neighboring nodes, because they use the information in order to select optimal requirement matching nodes. In addition, adding new resource attributes results in additional dimensions which bring an increased number of neighboring nodes and more management issues.

Similar to our work, Kim et. al[14] and Armada[17] use a tree-structure to check requirement matching. Kim et. al[14] uses Chord[23] as an underlying P2P network, and a resource information propagation tree is constructed based on the node ID. Each node needs periodic resource information updates to the parent node, which shares same drawbacks with SWORD. Armada[17] assigns an Object ID based on an attribute value, and a partition tree is constructed based on the proximity of the object ID. To locate nodes, only the desired region of Object ID needs to be scanned. All above methods do not support arbitrary matching, such as regular expression matching or partial string matching.

Cluster Monitoring System: Blue Eyes[22] and Ganglia[21] are hierarchically structured cluster monitoring systems. Blue Eyes[22] provides a reliable monitoring system by running multiple management servers, which are constructed as a self-organizing hierarchical tree using a management server list. For high system availability, monitoring data is replicated into multiple backup servers, and this replication requires much bandwidth consumption and may cause data consistency problems. Ganglia[21] uses gmond and gmetad to aggregate local resource information. Ganglia gmond gathers a local cluster node's information using multicast, and the gmetad accumulates inter-cluster information by collecting the gmond information. It also consumes much network bandwidth for local node information updates to the gmond, and the system is not reliable in case of gmetad failure. Intemon[13] is a server-client model monitoring system. It uses the SNMP to collect resource information and supports automatic data analysis based on the historical resource correlation pattern. Due to its static server-client relationship, it is not scalable in case multiple new clients join the monitoring system. The system also has to consume much bandwidth for periodical information update.

5. CONCLUSIONS AND FUTURE WORK

This paper presents and evaluates MapReduce-based query system, which uses a self-organizing broadcast tree to spread queries to the entire network. We described how our system adapted the concept of the Map and Reduce functional programming model and comparison with data processing MapReduce system. With the aid of self-organizing bounded-broadcast trees, we can distribute queries to the entire network efficiently. The use of condor_startd daemon to collect local resource information and ClassAd to process match-making allow us to perform various matching methods such as regular expression and partial string matching with plentiful resource information. Our system also supports adding new or user-defined resource attributes easily after deploying a pool without affecting performance. The analysis and simulation results show that dispersing queries to the entire network with the bounded-broadcast method makes queries to finish in a reasonable time, and they also show scability of our system for the increased number of nodes and resource attributes. PlanetLab deployment of our system and its performance metric show that MapReduce-based query system is a feasible and attractable solution for a wide area resource monitoring system. Next steps for MapReduce-based query system are showing the practicality of a bounded-broadcast method on top of other P2P systems, such as Chord, Pastry, Can, and etc. We will also work on deploying our MapReduce-based query system in a real system for monitoring and querying purpose.

6. REFERENCES

[1] J. Albrecht, D. Oppenheimer, A. Vahdat, and D. A. Patterson. Design and implementation trade-offs for wide-area resource discovery. *ACM Trans. Internet Technol.*, 8(4):1–44, 2008.

[2] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *Fifth IEEE/ACM International Workshop on In GRID*, pages 4–10, 2004.

[3] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In *P2P '02: Proceedings of the Second International Conference on Peer-to-Peer Computing*, page 33, Washington, DC, USA, 2002. IEEE Computer Society.

[4] J. Basney and M. Livny. *Deploying a High Throughput Computing Cluster*. Prentice Hall PTR, 1999.

[5] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev.*, 34(4):353–366, 2004.

[6] P. Boykin and et al. A symphony conducted by brunet, 2007.

[7] T. W. Choi and P. O. Boykin. Deetoo: Scalable unstructured search built on a structured overlay. In *Seventh International Workshop on Hot Topics in Peer-to-Peer Systems*, 2010.

[8] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[10] R. J. Figueiredo and et al. Archer: A community

[11] A. S. Foundation. `http://hadoop.apache.org/`.

[12] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: estimating latency between arbitrary internet end hosts. *SIGCOMM Comput. Commun. Rev.*, 32(3), 2002.

[13] E. Hoke, J. Sun, J. D. Strunk, G. R. Ganger, and C. Faloutsos. Intemon: continuous mining of sensor data in large-scale self-infrastructures. *SIGOPS Oper. Syst. Rev.*, 40(3):38–44, 2006.

[14] J.-S. Kim, B. Bhattacharjee, P. J. Keleher, and A. Sussman. Matching jobs to resources in distributed desktop grid environments. 2006.

[15] J.-S. Kim, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman. Using content-addressable networks for load balancing in desktop grids. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 189–198, New York, NY, USA, 2007. ACM.

[16] J. M. Kleinberg. Navigation in a small world. *Nature*, 406, August 2000.

[17] D. Li, J. Cao, X. Lu, and K. C. C. Chen. Efficient range query processing in peer-to-peer systems. *IEEE Trans. on Knowl. and Data Eng.*, 21(1):78–91, 2009.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[19] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, 2001.

[20] S. M. Ruman. R, Livny. M. Matchmaking: distributed resource management for high throughputcomputing. In *In proc. 7th IEEE symp. on High Performance Distributed Computing*, pages 140–146, 1998.

[21] F. Sacerdoti, M. Katz, M. Massie, and D. Culler. Wide area cluster monitoring with ganglia. In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 289–298, Dec. 2003.

[22] S. Song, K. D. Ryu, and D. D. Silva. Blue eyes: Scalable and reliable system management for cloud computing. *Parallel and Distributed Processing Symposium, International*, 0:1–8, 2009.

[23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[24] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[25] P. Yalagandula and M. Dahlin. A scalable distributed

distributed computing infrastructure for computer architecture research and education. In *Collaborative Computing: Networking, Applications and Worksharing*, volume 10, pages 70–84, 2009.

information management system. *SIGCOMM Comput. Commun. Rev.*, 34(4):379–390, 2004.

[26] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. Technical Report UCB/EECS-2008-99, EECS Department, University of California, Berkeley, Aug 2008.