# 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

The growth of computing power in workstations and personal computers attracts substantial interest in computing clusters and desktop grids[2][4]. They enable sharing cpu power, storage capacity, and applications among distributed computers. These computing devices span local, regional, or wide area networks. For the purpose of organizing and handling those widely distributed computing resources efficiently, distributed system platforms such as Condor[4] and Boinc[2] have been successfully used. After deploying such services, several management issues still remain. The services should be able to distribute jobs evenly among distributed nodes.[1] The system should be scalable in case new nodes join. It also has to be fault tolerant and secure in the presence of abnormal or malicious behavior of a node or churn. A management system also has to support methods for monitoring nodes and discovering resources. In the context of a cluster's scalability, self-organizing, and fault-tolerance, a P2P network[25][20][18][19] provides the ability of managing node join and departure in the system. With respect to locating resources in a P2P network, a centralized indexing server is a possible solution[2][4]. However, this approach is not scalable for large and widely distributed systems. The centralized indexing server is also a single point of failure. To remove drawbacks of a central indexing server, many approaches, such as Distributed Hash Table (DHT), have been proposed. In a structured P2P network, the DHT provides a scalable lookup method in a guaranteed number of routing hops[25][20]. However, it does not support unstructured queries or searching multiple hashed keys at once, because the original DHT lookup method is based on matching a single hashed key. To overcome limitations of DHT, SWORD[1] added small modifications to the original DHT to allow it to support multiple attributes matching in a query. Similar to SWORD, Mercury[5] supports multi-attribute range query by dividing a structured P2P network's address space according to attribute values. However, those methods impose much network traffic for resource information updates when the number of resource attributes increases. In addition, the retrieved resource information might be stale due to the characteristics of DHT entry update period. To overcome these limitations of a DHT-based resource query system, this paper presents a distributed, decentralized, and self-organizing query method system on top of a P2P network. This method uses bounded-broadcast in order to disseminate queries to the desired region. When a node receives a query, the node gets local resource information using a local monitor(e.g.,condor_startd daemon[4]). Using the information, each node checks a requirement matching using matchmaker(e.g., Condor Clas-

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

sified Advertisements (ClassAd)[21]). To process a query and aggregate replies from each node efficiently, we use the MapReduce and bounded-broadcast tree. 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 purpose of the bounded-broadcast[7] is distributing a message to a sub-region of a P2P network by using 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 Brunet node maintains two kinds of neighbor connection: an adjacent node connection and a distant node connection. Two adjacent nodes in clockwise and counterclockwise directions are registered as adjacent neighbors. Distant neighbor nodes are selected randomly, and the number of distant neighor nodes are about $0.5 * log(N)$, where $N$ is the number of nodes.

A node which is responsible for disseminating a message redistributes the message to its neighbor nodes inside its allocated sub-region while allocating new sub-regions to the neighbor nodes appropriately. Those nodes which receive the message redistribute the message using the same manner. For example, to broadcast a message over the sub-region[A,L] in Figure 1, a message initiator inserts a broadcast command to node A with sub-region information, [A,L]. Node A recognizes node B and P as its adjacent neighbor nodes, and E, K, L, and M as its distant or short-cut connection neighbors. Based on the node A's connection table, node A disseminates broadcast messages only 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.

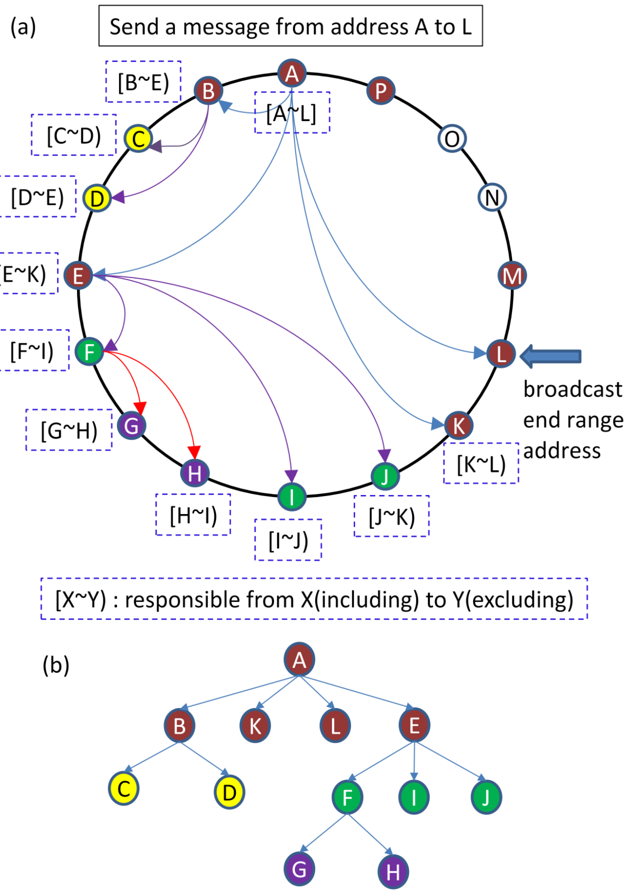

**Figure 1: Bounded-broadcast message propagation-(a)Node A broadcasts a message from node A to L. Node A first sends the message to its neighbor nodes, B, E, K, and L after setting broadcast region appropriately. The message recipient node sends the message to its neighbors inside its broadcast region recursively. (b)Generated bounded-broadcast tree after disseminating the message.**

If the broadcast message initiating node does not lie in the bounded-broadcast region, the broadcast 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 queries to nodes in the 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 $O(\log(N))$[25], but smaller than a naive flooding based broadcast method $O(N)$.

## 2.2 MapReduce

Map and reduce functions are used in the Lisp programming language and many other functional programming languages. Based on the concept of map and reduce functional programming model, Google proposed a software framework to parallelize large dataset computations efficiently[9]. 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 MapReduce framework supports distributing map and reduce tasks among nodes in a cluster to enable parallel processing, dividing input files into multiple chunks. Thus users do not have to take care of detailed management issues for distributed parallel job execution. Instead, users have to define map and reduce functions associated with their needs. The open group also provides a MapReduce function called Hadoop MapReduce[11]

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 underlying 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[25], CAN[18], or Pastry[20] if those platforms provide efficient multicast.

### 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. Similar to Google MapReduce, 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[21] 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 aggregate and 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.

### 2.3.3 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

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

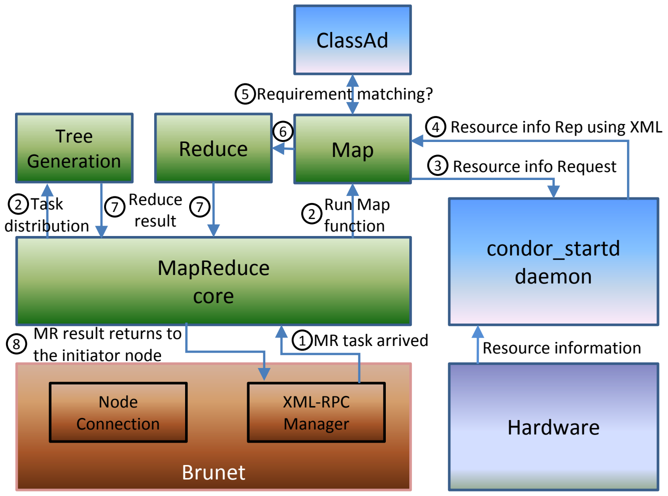

**Figure 2: MapReduce query system architecture**

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[21] 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[21] 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:

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.

## 2.4 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 the node's job is reassigned by a central manager. The Late scheduler[28] 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[26][27].

# 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

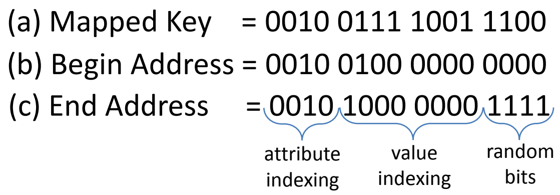
(a) Mapped Key = 0010 0111 1001 1100
(b) Begin Address = 0010 0100 0000 0000
(c) End Address = 0010 1000 0000 1111
attribute indexing
value indexing
random bits

**Figure 3: SWORD DHT key mapping**

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 range query method shares the ultimate objective with MapReduce-based query system, whose goal is to allow end users to locate a subset of nodes which satisfy users' requirement in a pool. SWORD provides a matchmaking service using DHT. The original DHT works based on a <key, value> storage. Each key is hashed to a node ID, and the appropriate node, whose node ID is the closest to the hashed key value clock-wise direction on a P2P ring structure[25][20], keeps the <key, value> pair. Mapping an 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. To overcome this limitation, SWORD maps an attribute name and a value to the DHT entry key. Among $n$ bits of a DHT key size, 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 the 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 4 shows an attribute value and query address range mappings as DHT keys. 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 Fig.4(a). To find nodes whose free disk space is between 64GB and 128GB, SWORD has to determine query begin address and end address. To decide the query begin address, it sets the attribute bits as 0100, which is a pre-determined free disk space attribute index, the value bits as 64GB, and the random bits as all 0x0s, shown in Fig.4(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 Fig.4(c). Because Fig.4(a) mapped key is located between the begin and end address, a query inside the region can find satisfying nodes. 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 Number of Visited to Complete a Query-$N_Q$, Query Latency-$L_Q$, Query Bandwidth-$B_Q$, 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 shown in the Figure.4 as $I_A$, and $I_V$, respectively. In this analysis, we assume that all nodes in a pool are evenly distributed through all address region.

### *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 is $N$. SWORD can optimize this by manipulating the number of bits for an attribute and a value indexing bits in Figure 4. As the number of bits for an attribute and a value indexing increases 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 Latency For a Query*

The latency of a query depends on the number of visited nodes to complete a query. Because MapReduce-based query system propagates queries using bounded-broadcast, the latency is closely related to the tree-depth of bounded-broadcast. According to DeeToo[7], tree depth of bounded-broadcast is $O(log^2(N))$. Assuming SWORD query is propagated 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 system 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 a query and response message size 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 checks query matchings at nodes inside the query region based on periodically updated remote node's resource information. Otherwise, MapReduce-based query system performs matchmaking based on local resource information. In this section, we are going to analyze SWORD's bandwidth costs to update resource information to remote nodes. Every node has to update its resource information, whose size is $A_N * A_S$(neglecting serialization overload), $A_N$ times, because any nodes mapped to the hashed key have to provide query matching service. All nodes($N$) in the pool will perform updates periodically. In addition, each DHT entry has to be routed to a proper hashed key node, which takes $O(log(N))$ hops[25][20] or $O(\frac{1}{k}log^2(N))$,where $k$ is number of short-cut connection[6]. Accordingly, bandwidth for a resource information 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, such as operating system. When it comes to dynamic information, such as current cpu load or free memory size, the stale information is highly likely to be useless. Because a DHT entry age is dependent on the DHT entry update period, SWORD can make a update period shorter to keep resource information fresher. However, 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. This simulator provides us a controlled experiment environment, and also allows us to check the correctness of our system implementation. We used King data set[12] to set network latency between nodes. Using Archer[10], we could run simulations on a decentralized computing cluster efficiently. On the simulation, we ran 1-simulated hour after the 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 Complete Time*

Figure 5.(a) shows query completion times. As we can see from the figure, the query time grows as the number of nodes increases. To highlight a relationship between a query complete time 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, so we can conclude from this pattern that the order of our query system latency is less than $O(N)$, but bigger than $O(log(N))$. 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, while the query region is decreased to 0.3%. It may show different latency values for some other P2P systems, but we can conclude that the cost for querying the entire network would be small if we design a broadcast method while considering each P2P network's characteristics carefully, which is a part of our future work.

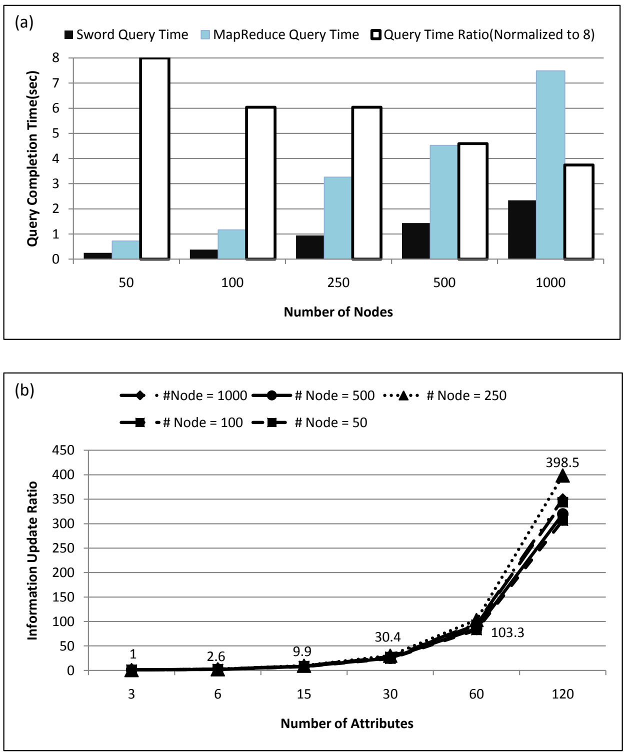

**Figure 4: MapReduce-based query system and SWORD simulation result (a)Query completion time. Query time ratio = (Query complete time/Number of nodes)*1.6. (b)SWORD resource information update size. Ratio=(BW consumption)/(BW consumption when the number of attribute is 3)**

### 3.3.2 *Query Bandwidth Usage*

Table. 1 shows a bandwidth consumption to complete a query. The *Ratio* is calculated as $\frac{SWORDBandwidth}{MapReducebandwidth}$. The ratio value decreases as the number of nodes increases, but it is still bigger than 0.3%, which is the fraction of the SWORD query region to the entire network. One of reasons for this is multiple hops to route a message to a node inside a query region. A probability of finding a node in SWORD 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 in a query region is usually 0 when the number of nodes in the entire network 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

**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 |

region.

### *3.3.3 Resource Information Update Bandwidth*

Figure 5(b) shows a resource information update bandwidth consumption when the number of published attributes 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 $\frac{n-attribute's\ bandwidth}{3-attribute's\ bandwidth}$. When the number of attribute is small, the increasing rate is not $O((Number\ of\ Attribute)^2)$ due to the initial resource information serialization overhead. As the serialization overhead effect becomes negligible, the rate follows $O((Number\ of\ Attribute)^2)$

Based on the analysis and simulation results, we conclude that the novel bounded-broadcast allows propagating query to the entire P2P network eligible within a reasonable amount of time. Considering that propagating a query is much less expensive than publishing large 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 6 shows MapReduce-based query system's latency in completing queries 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 until they completed, 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 P2P network is our future project.

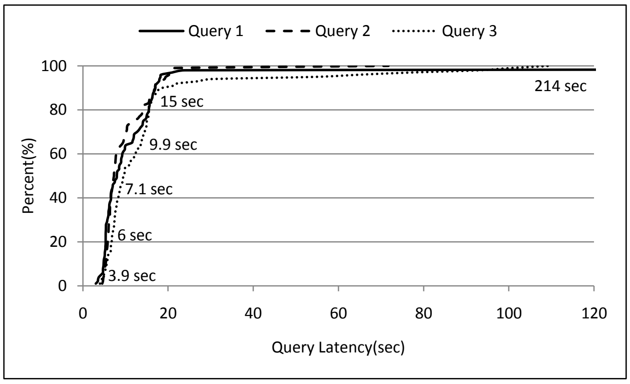

**Figure 5: Cumulative distribution of query latency on the PlanetLab with about 600 nodes**

# 4. RELATED WORK

We have already discussed a range of related works which are closely tied to MapReduce-based query system in previous sections. In this section, we will discuss some related works on P2P network range query systems and cluster monitoring systems.

**Range Query System on a P2P network**: Kim et. al[15] and Artur et. al[3] discuss resource locating methods using a multiple dimensional 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 a local node's information update to neighbor 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 neighbor 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[25] as an underlying P2P network, and a resource information propagation tree is constructed based on the node id. Each node needs periodic resource information update 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. However, this query system does not support arbitrary matching, such as regular expression match or partial string match.

**Cluster Monitoring System**: Blue Eyes[23], Ganglia[22], and Supermon[24] are hierarchically structured cluster monitoring systems. Blue Eyes[23] 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[22] uses gmond and gmetad to aggregate local resource information. Ganglia gmond gathers 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 update to the gmond, and the system is not reliable in case of gmetad failure. In Supermon[24], the mon process is responsible for local resource information archiving and filtering, and the supermon process aggregates multiple mon processes' data and presents the aggregated data as a single data sample. The Supermon's hierarchical structure is not self-configurable, because the relation between mon server and supermon server has to be registered manually by a system administrator. 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 the MapReduce-based query system, which uses a self-organizing broadcast tree to spread queries to the entire network. To our best knowledge, this system is the first demonstrated implementation of MapReduce application deployed on top of a P2P network. We described how our system adapted the concept of Map and Reduce functional programming model and differences between Google MapReduce system and our MapReduce-based query system. With the aid of self-organizing bounded-broadcast method, we could distribute resource locating queries to the entire network neatly. The use of condor_startd daemon to collect local resource information and ClassAd to process matchmaking allow us to perform various matching method(regular expression, partial string match) with plentiful resource information. Our system 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 complete 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 practicability 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 cluster 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 distributed computing infrastructure for computer architecture research and education. In *Collaborative Computing: Networking, Applications and Worksharing*, volume 10, pages 70–84, 2009.

[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] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

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

[21] 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.

[22] 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.

[23] 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.

[24] M. J. Sottile and R. G. Minnich. Supermon: A high-speed cluster monitoring system. In *CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing*, Washington, DC, USA, 2002. IEEE Computer Society.

[25] 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.

[26] 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.

[27] P. Yalagandula and M. Dahlin. A scalable distributed information management system. *SIGCOMM Comput. Commun. Rev.*, 34(4):379–390, 2004.

[28] 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.