A Computing Resource Discovery Mechanism Over a P2P Tree Topology

D. Castellà[1], H. Blanco, F. Giné, and F. Solsona

University of Lleida, C/ Jaume II, 69, Lleida, Spain,
dcastella,hectorblanco,sisco,francesc@diei.udl.cat

Abstract. Peer-to-Peer (P2P) computing, the harnessing of idle compute cycles through Internet, offers new research challenges in the domain of distributed computing. In this paper, we propose an efficient computing resource discovery mechanism based on a balanced multi-way tree structure capable of supporting both exact and range queries, efficiently. Likewise, a rebalancing algorithm is proposed. By means of simulation, we evaluated our proposal in relation to other approaches of the literature. Our results reveal the good performance of our proposals.

Topics: Parallel and distributed computing, P2P computing.

1 Introduction

P2P paradigm takes advantage of the under utilization of personal computers, integrating thousand or even millions of users into a platform based on the sharing of computational resources [1]. Some current research projects, such as CompuP2P [2], CHEDAR [3] or CoDiP2P [4], propose using the P2P paradigm for distributed computing. P2P computing is distinguished by a mutable amount of computational resources (CPU, Memory and Bandwidth) provided by each peer. Thus, the computational resource management becomes a research challenge.

The resource discovery mechanisms in P2P computing are classified in structured and unstructured ones [5]. The unstructured algorithms are characterized by the fact that they use only local information of their neighbors. The CHEDAR's searching mechanism [3] fits in this category. Although these systems adapt easily to frequent node joins and disjoins, they do not scale very well for very large networks. On the other hand, the searching mechanisms based on structured information are generally faster and have a predictable service search time. In this set, we can find the well known Chord algorithm [6], which is used by the CompuP2P platform [2]. Although the Chord protocol is very efficient for exact queries, this is not well suited for range queries since hashing destroys the ordering of data. Recent works, such as Squid [13] supports keyword searches, including wildcards, partial keywords and ranges queries, based on DHT. It uses a locality-preserving indexing scheme based on Space Filling Curves (SFC), where each data element is indexed and shared using a set of keywords and mapped to a single point in its key space. Other works, such as [11, 7, 12], propose network

discovery services without the use of DHTs. Caron et al. [11] presents a new architecture, Distributed Lexical Placement Table (DLP), based on a Prefix Tree which supports automatic completion of partial search string, range queries and multicriteria searches. Likewise, BATON [7] proposes a balanced tree structure overlay which supports exact and range queries, also without the use of DHTs. This emphasizes that adding a small number of links in addition to tree edges, they are able to obtain an excellent fault tolerance and a balanced congestion. Finally, Harren et al. [12] take advantage of DHTs to improve the scalability and addapts it to support complex queries in relational databases. Altought these works are very optimized for resource discovery services, they are very restrictive in providing a discovery mechanism over a mutable environment. Note that, in a P2P computing environment, the shared resources change its disponibility over the time. So, it is necessary to check periodically the resources for better scheduling pourposes.

Other works related to Grid environments, [14, 15] proposes a distributed super-peer model for handling membership management and resource discovery service in large-scale Grids and exploit centralized/hierarchical information service provided by the Grid infrastructure of the local PO (Physical Organization). The related work describes that a Grid is viewed as a network interconnecting small-scale Grids, referred as PO's. For each PO, a subset of powerful nodes having high availability properties are used as super-peers. These nodes are responsible for the communications with the other POs and maintain metadata about all the nodes of the local PO. An structured P2P topology of super-peers implements the join and departure of Grid nodes and the resource discovery service. The super-peer model is similar to the manager's peer role used in our proposal, which controls a set of peers, named Areas, and stores information about discovery services. However, its organization is different because each peer of an area can be the manager of the inmediate lower level area, whereas in a super-peer model, each PO is considered as a leaf node.

In this paper, we propose a new structured computing resource discovery mechanism, which provides exact and range query facilities and scalability features with a low algorithmic cost. Our approach is oriented to the CoDiP2P (P2P Distributed Computing) system developed by our group in previous works [4]. Following the CoDiP2P architecture, the proposed lookup mechanism follows an structured architecture. It is based on a balanced multi-way tree structure capable of supporting both exact and range queries efficiently. In addition, this paper proposes a rebalancing algorithm, which allows the tree to be maintained totally balanced and re-link any isolated area. Thus, CoDiP2P exploits efficiently the well known characteristics of a tree topology ($\theta(\log_{|Area|}(N_{tree}))$ lookup length, where N_{tree} is the total number of peers, and constant linkage cost) to manage the mutability of resources. We have analyzed the performance of our proposals by means of simulation in relation to the Chord algorithm by the case of exact queries, and the BATON algorithm by the case of range queries. In both cases, the obtained results reveal the competitiveness of our proposals.

The outline of this paper is as follows; Section 2 revises the CoDiP2P architecture. Section 3 presents the discovery mechanism used by CoDiP2P for exact and range queries. The rebalancing mechanism used for CoDiP2P system is described in Section 4. The efficiency measurements of our proposals are performed in Section 5. Finally, the main conclusions are explained in Section 6.

2 CoDiP2P Architecture

We present a review of the CoDiP2P architecture, explained in detail in [4]. In order to describe these, some previous concepts must be introduced:

- **Area** A_i is a logical space made up of a set of workers.
- **Manager** M_i manages an area and schedules tasks over the workers.
- **Worker** W_i is responsible for executing tasks scheduled by its manager.
- **Replicated managers** RM_i**:** Each area A_i maintains a set of Replicated Managers. Each RM_i maintains a copy of the same information kept by M_i. Thus, if M_i fails, then the oldest RM_i of the same area will replace it.

Fig. 1 shows the linked structure of peers in CoDiP2P based on a tree topology with an areas size of 3 peers. Note that this type of structure allows a manager M_i of an area located at level i to be a worker W_j in an area located at level $i+1$ at the same time. In addition, this same node can be also a RM_i.The main functionalities of CoDiP2P system are *insertion and departure of peers*, *updating system information* and the *scheduling mechanism* to launch a parallel job from any peer in the system. These three algorithms are explained in detail in [4].

In order to understand better the *searching and rebalancing algorithms* explained later in this paper, some highlights of the *updating* and *departure algorithms* are needed.

2.1 Updating Algorithm

The main aim of the updating algorithm consists of maintaining the information about the computational resources available in the system.This is divided in two parts: the *manager_updating* and the *worker_updating*.

By means of the *manager_updating*, every manager M_i sends, every T seconds, a message to all the workers belonging to the same area A_i to notify that it is alive, together with the List of its Top Managers (LTM). The LTM is a list that contains the addresses of the managers located over the manager M_i in direction towards the root manager (M_1). For instance, the LTM of the manager M_8 shown in Fig. 1 would be M_4, M_2, M_1.

Whenever a $Peer_i$ receives a $message_{upd}$ from its manager M_i, it launches the *worker_updating* function. Each $Peer_i$, depending also on its role, sends to its manager M_i statistical information ($SttInf_j$) about its locally available computational resources or the available computational resources managed by such a peer (if $Peer_i$ is the manager of a lower area). Note that the information

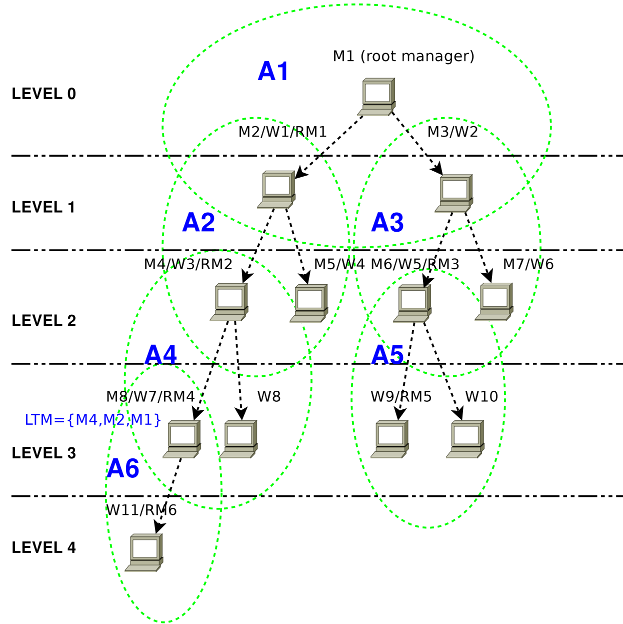

Fig. 1. Tree topology of peers in CoDiP2P.

$SttInf_j$ is only sent when it differs from the previous one sent $SttInf_{j-1}$. The cost of the algorithm is $\theta(|Area|)$, where $|Area|$ is the number of peers in one area.

2.2 Departure of peers

Whenever a peer leaves from the Tree, voluntarily or involuntarily, the manager of the disconnected peer detects the broken link in the updating algorithm, explained in Sec. 2.1. Whenever this happens, the manager checks if the peer is a worker node or a manager in the immediate lower level:

- In the worker case, there is no problem in restructuring the system because it is a final node, and no more work must be performed.
- In the manager case, the restructuring operation, described in Alg. 1, is applied by the replicated manager RM_i of the current area A_i. After T seconds, this detects that there is no answer from the faulting manager M_i and executes the mechanism.

Note that the algorithm selects the oldest replicated manager, by issues of peers reliability. Thus, the oldest peers, which the system considers more reliable, are in the upper levels of the tree, compared with the youngest peers, which are considered more irregular and are in the lower levels.
The cost of Alg. 1 is determined by the number of peers affected by the fall of the

peer. The worst scenario happens when the replicated manager RM_i, selected to replace an output manager is also the manager of a lower level. The cost of the algorithm is $\theta(\log_{|Area|}(N_{tree}) - 1)$, where N_{tree} is the total number of peers in the CoDiP2P system and $|Area|$ is the size of the areas.

```
procedure RM_i.Manage_Departure_in_Tree()
Input: RM_i,A_i
begin
    if |A_i − RM_i| ≠ 0 then
        M_i := RM_i ∈ A_i;
        RM_i notifies ∀workers ∈ A_i that it will be manager;
        if RM_i is also manager of a lower area A_j then
            RM_i selects the oldest RM_j ∈ A_j;
            RM_i notifies RM_j that it will be manager of A_j;
            if RM_j is manager of lower level then
                RM_j.Manage_Departure_in_Tree();
            end
        end
    else
        RM_i becomes worker of upper area of A_i;
    end
end
```

Algorithm 1: Departure of peers in CoDiP2P.

3 Searching Algorithms

We have added a new resource discovery mechanism to the CoDiP2P service layer to provide two different kinds of searching, one based on exact queries and the other one based on range queries. Both algorithms look up the addresses of peers throughtout the tree that have the desired CPU power available, although they can be used for looking up any kind of computing resources.

3.1 Exact Query Searching Algorithm

The searching mechanism is designed to take advantage of the topology and roles of peers. This algorithm is based on looking through the local database (DB) stored by each peer, which contains the computing resources characteristics of the peers located below it throughout the tree branch.

According to the Alg. 2, whenever a $Peer_i$ requests a CPU query, firstly it checks on its own DB if there is a peer with the required CPU power. In the case of a search failure, $Peer_i$ forwards the searching query to its manager M_i. If the search fails again, the next manager located on the branch continues the same search in a recursive way until it reaches the zero level (M_1). Finally, if the searching is successful then the CPU owners peer address is returned to the

```
function Peer_i.Search_EQuery(CPU_query)
Result: Peer_Address
begin
    foreach register ϵ DB.CPU_Table do
        if register.CPU = CPU_query then
            Peer_Address := register.address;
        end
    end
    if Peer_Address = NULL then
        if Peer_i ≠ M_1 then
            Peer_Address := M_i.Search_EQuery(CPU_query);
        else
            Peer_Address := NULL;
        end
    end
    return Peer_Address;
end
```

Algorithm 2: Exact Query Searching Algorithm.

$Peer_i$. Note that the cost of this algorithm is $\theta(\log_{|Area|}(N_{tree}))$.
Note that one important problem related to the tree topology is the traffic congestion produced by the routing messages of the searching algorithm and the updating algorithm. Regarding to the searching traffic, only those searches which requested peers are located in another subtree of Level 1 arrives to the root peer. So, the updating algorithm can cause even more bottleneck in the root peer than the searching one. The searching congestion is measured in the Experimental Results Section (5.1). In addition, it is worth pointing out that the updating traffic is characterized by small messages sent each T period. Thus, we must fix the value of T by balancing the congestion and the updating of information.

3.2 Range Query Searching Algorithm

The Alg. 3 shows the range query searching algorithm, to which three parameters are passed: the low and high limits of the searched CPU range values and the number of items that the algorithm has to catch. Compared with the Exact Query algorithm, Alg. 3 differs in two points. The first point is that it returns a List of Peer Addresses (LPA), which contains the desired CPU power inside the requested range. The second one is that the algorithm does not finish until it has filled the LPA up with nr_items or it has reached level zero (M_1). Note that the cost of this algorithm is also $\theta(\log_{|Area|}(N_{tree}))$.

4 Rebalancing Mechanism

The churn of peers in a P2P environment can unbalance the tree topology. As a consequence and as we can see in Fig. 2(left), one tree branch can be much

function $Peer_i.Search_RQuery(low_CPU, high_CPU,\ nr_items)$
Result: LPA(=List Peer Addresses)
begin
    foreach $register\ \epsilon\ DB.CPU_Table$ **do**
        if $register.CPU \geq low_CPU \wedge register.CPU \leq high_CPU\ \wedge\ LPA.size < nr_items$ **then**
            $LPA.add(register.Address)$;
        end
    end
    if $LPA.size < nr_items\ \wedge\ Peer_i \neq M_1$ **then**
        $LPA+ = M_i.Search_RQuery(low_cpu, high_cpu,\ nr_items - LPA.size)$;
    end
    return LPA
end

Algorithm 3: Range Query Searching Algorithm.

longer than another (Case 1 and 2) or one area can remain isolated from the root manager (Case 3). Obviously, the unbalancing of the tree will decrease the performance of the search mechanisms explained above.

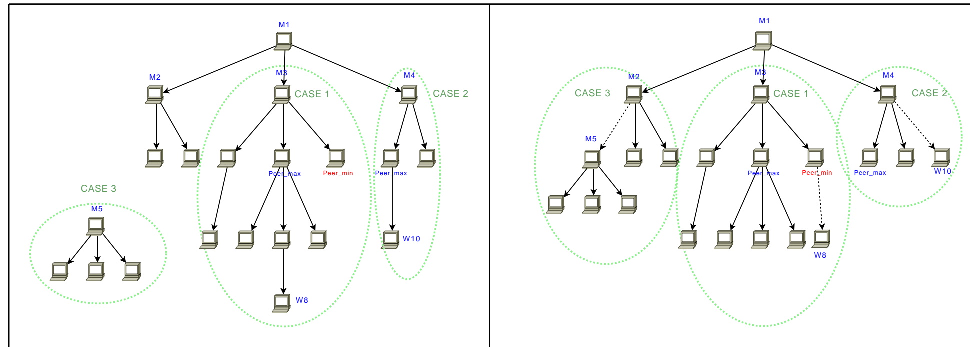

Fig. 2. Tree without rebalancing (left) and with rebalancing (right).

In order to solve this problem, a rebalancing mechanism oriented to a tree topology is proposed. Alg. 4 allows areas to be moved from one site to another or lost links to be restored.

Periodically, each $Peer_i$ of the tree launches the rebalancing algorithm. As we can see in Alg. 4, firstly the algorithm tests if any of the three cases shown in the Fig. 2(left) happens. Whenever it happens, Alg. 4 works as follows in each case:

– **Case 1**: Whenever $Peer_i$ is also a manager M_i and the area A_i is full, M_i checks from the List of its Children (LC) which are their sons with the max-

imum ($Peer_{max}$) and minimum ($Peer_{min}$) number of levels hanging down from it. If the difference between both values is greater than one level then the tree is considered to be unbalanced and as a consequence the *rebalancing_levels* procedure is called. This procedure moves the branch hanging down from the $Peer_{max}$ son with more levels to the $Peer_{min}$. As we can see in the Case 1 of Fig. 2(left), $M3$ executes the rebalancing mechanism and as a consequence the Peer $W8$ hanging down from the son of $Peer_{max}$ is linked to $Peer_{min}$. The result is shown in Fig. 2(right). Note as the example of Fig. 2 assumes an areas size of 4.
- **Case 2**: This case happens whenever $Peer_{max}$ has levels hanging down from it and the area A_i is not full. This situation means that the tree is unbalanced due to the fact that the capacity of the area is not fulled up. In this case, the half levels below $Peer_{max}$, denoted as *rlevels* in Alg. 4, are linked to $Peer_i$ by means of the same *rebalancing_levels* procedure explained above. As we can see in Case 2 of Fig. 2, $W10$ is linked to $M4$ after applying the rebalancing mechanism.
- **Case 3**: The third and the last part of the Alg. 4 is activated whenever an area A_i is isolated from its above manager. According to this goal, $Peer_i$ checks if there is an above manager (M_i) and if there is not then $Peer_i$ looks for a manager from their List of Top Managers (LTM). If the searching is success then $Peer_i$ is linked to the new manager. As we can see in Case 3 in Fig. 2, $M5$ is linked to $M2$ after applying the rebalancing mechanism.

Note that all searching operations, whose peers are affected by one of the three rebalancing cases, will be aborted. They will be resumed when the rebalancing procedure was finished.
The cost of this is $\theta(\log_{|Area|}(N_{tree}) - 1)$ because the maximum number of hops is equal to the length of a branch from the root manager M_1 to a leaf node W_i.

5 Experimentation

The performance of our proposals was tested by means of GridSim [8] and SimJava [9] simulators. In order to simulate our P2P platform with GridSim, peers were modelled as user entities by means of threads. All entities (peers) were connected by network links, whose bandwidth and latency can be specified at the start time. SimJava features provide the management of events and the mechanism for discovering peers.

All tests were performed with 10,000 peers and a total of 125,000 searches, which follows a Poisson distribution with a mean frequency of 125 $searches/s$ by default. According results obtained previously in [4], the updating procedure is continuously executed in periods of 20 seconds. A summary of the experimental results is shown in next section.

5.1 Experimental Results

First of all, we tested the influence of the rebalancing algorithm over the searching algorithms. Likewise, the impact of the number of replicated managers (RM)

procedure $Peer_i.rebalancing()$
Data: M_i(=Manager of $Peer_i$), A_i(=Area of M_i), LC(=List of Childs),
LTM(=List of Top Managers)
begin
  if $Peer_i = M_i$ **then**
    $\{Peer_{max}, Peer_{min}\} := Peer_i.max_min_child_peers(LC)$;;
    `// Case 1`
    if $Peer_{max}.levels_below - Peer_{min}.levels_below > 1 \ \wedge \ A_i.isfull()$
    then
      $rlevels := \lfloor (Peer_{max}.levels_below - Peer_{min}.levels_below)/2 \rfloor$;
      $Peer_{max}.rebalancing_levels(rlevels, Peer_{min})$;
    end
    `// Case 2`
    if $Peer_{max}.levels_below \geq 1 \ \wedge$ ***not*** $A_i.isfull()$ **then**
      $rlevels := \lfloor Peer_{max}.levels_below/2 \rfloor$;
      $Peer_{max}.rebalancing_levels(rlevels, Peer_i)$;
    end
  end
  `// Case 3`
  if $\nexists M_i$ **then**
    for $j := 0$ **to** $LTM.size$ **do**
      $Peer_j := LTM.get(j)$;
      if $\exists Peer_j$ **then**
        new $Peer_i \longrightarrow Peer_j$;
        break;
      end
    end
  end
end
procedure $Peer_i.rebalancing_levels(rlevels, Peer_j)$ **begin**
  if $Peer_i.levels_below \geq rlevels$ **then**
    $Peer_k := Peer \in LC \mid MAX_{k=1}^{|LC|}(LC.get(k).levels_below)$;
    $Peer_k.rebalancing_levels(rlevels, Peer_j)$;
  else
    new $Peer_i \longrightarrow Peer_j$
  end
end

Algorithm 4: Rebalancing Algorithm

was also evaluated for both cases, with and without rebalancing. Fig. 3 shows the percentage of unsuccessful searches in relation to the percentage of failed peers for 1 an 3 RMs. In the non-rebalancing case, we can see as the failures scaled well and the results ranged up to 40% with only 1 RM and 25% with 3 RMs. When we applied rebalancing, the results were very satisfactory with a rate of unsuccessful searches below 5% for both cases, 1 and 3 RMs. In this case, the number of RMs practically does not affect the behavior of the searches and the influence of the percentage of faulting peers on the system is imperceptible. Therefore, these results prove the well performance of our rebalancing algorithm.

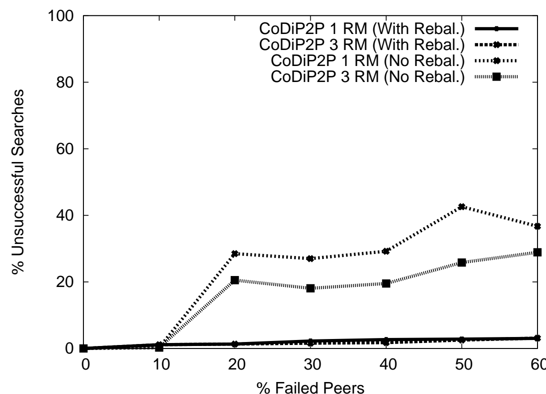

Fig. 3. Comparative of unsuccessful searches in terms of failed peers with and without Rebalancing Algorithm.

Next, we evaluated the impact of the searching frequency on the exact query case. The performance achieved by our proposals, denoted as CoDiP2P, was compared with Chord. Samples were collected for CoDiP2P with 1 and 3 Replicated Managers (RM) and 1 and 3 successors in the Chord algorithm.

Fig. 4(left) and (right) show the results of the exact query search with a high (12500 $searches/sec$) and low (125 $searches/sec$) searching frequencies, respectively. In general, both plots showed that CoDiP2P obtained better results than Chord, specially when the frequency was high. On the other hand, we saw how the influence of the successors in the Chord case was higher than the use of replicated managers in CoDiP2P. This was because the Chord's successors are active elements in the searching process, whereas in the case of CoDiP2P, the replicated managers do not play such an important role in the searching algorithm. Focusing on the influence of the searching frequency, we saw that the CoDiP2P and Chord results ranged up to 40% and 80% with a high frequency (see Fig. 4(left)), whereas they ranged from 15% to 5% with low frequency respectively (see Fig. 4(right)). This behavior is due to the fact that a low searching frequency gives both systems enough time to recompose system tables and links. However, CoDiP2P continues giving better results than Chord because the system has

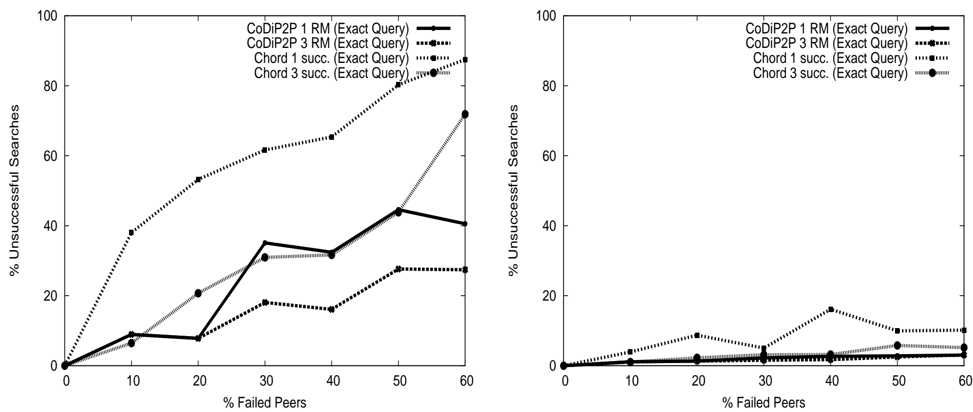

Fig. 4. Exact queries searching Algorithm versus Chord with a $Freq = 12500\ searches/sec$ (left) and $Freq = 125\ searches/sec$ (right).

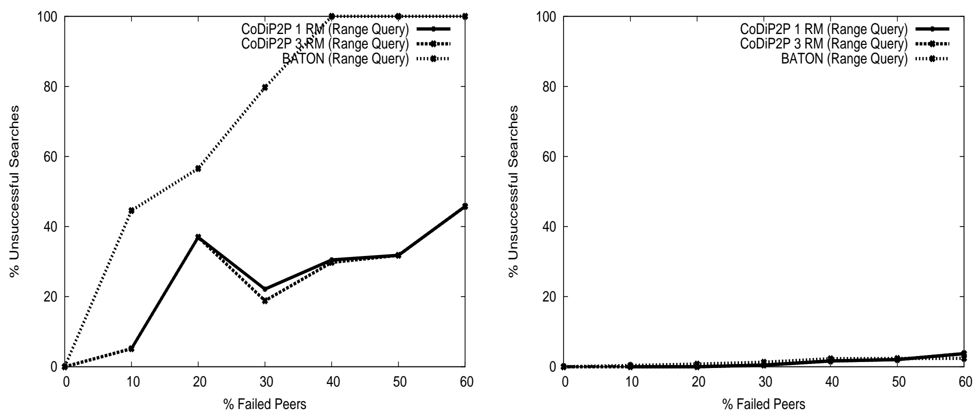

Fig. 5. Range queries searching Algorithm versus Baton with a $Freq = 12500\ searches/sec$ (left) and $Freq = 125\ searches/sec$ (right).

enough time to rebalance the system completly and thus better performance is obtained. This is due to the maintaining cost of the overall system in CoDiP2P is $\theta(|Area| \cdot \log_{|Area|} N_{tree})$ whereas the maintaining cost of Chord is higher, $\theta(N \cdot \log^2 N)$. In general, we conclude that in any case, the CoDiP2P rebalancing algorithm performs better than the restructuring of the Chord DHT structure.
Our next evaluation was to compare the performance of our range query approach in relation to the BATON algorithm, following the same methodology described above. Fig. 5 shows the percentage of unsuccessful searches in relation to the percentage of failed peers for high (Fig. 5(left)) and low (Fig. 5(right)) searching frequencies. As we can see in both figures, our approach improves the BATON results. This improvement is very signicant for the case of high frequency, achieving a maximum difference around 70% (see Fig. 5(left)). This difference is due to the fact that each peer on the BATON system has stored less keys than CoDiP2P and as a consequence BATON needs to do more hops than CoDiP2P for searching a specific set of values. In addition, BATON has a major dependency of the neighbourhood and by this reason the drop of a neighbour has a high repercussion on the searching process.
Next, we compared the response time of CoDiP2P for both cases, exact and range query, in relation to the Chord algorithm. The discovery mechanism needed a minimum interval, called "response time", to update the system completely after a peer fault. According to our aim, we obtained the percentage of unsuccessful searches for an hypothetical massive drop of 50% of the total peers at the same time. Thus, we could compare the robustness of both approaches, CoDiP2P and Chord. From Fig. 6 (left), we can see that CoDiP2P took 255 units of time to recover the parts of the system affected by the faulting peers in both cases, whereas Chord needed more than 800 units of time. This is because Chord updates one entry of its finger table in each updating step and as a consequence, it needs more time to update the tota1 of 160 entries in its finger tables. In contrast, CoDiP2P only needs a number of jumps equal to the number of levels of the tree to update the network completely after a massive drop of peers. In general, we can conclude that CoDiP2P has a response time approximately 3 times faster than Chord.
Finally, Fig. 6 (right) shows the congestion (percentage of messages) achieved throughout the levels of the tree topology for different areas sizes, when the searching mechanism is applied. Level 0 represents the root of the tree. We can see as the congestion increases with the areas size. As it was expected, congestion is critical in the root and decreases when ascending levels. With an areas size of 21 peers, congestion is around 100%, causing a bottleneck in the root node. In order to reduce this congestion, the trees level and the areas size should be limited below a specific threshold. Therefore, in order to maintain the scalability of the system, we should group the peers according to its characteristics in a set of different small trees, which would be connected by a second level topology with good scalability properties. This will be the main goal of future work.

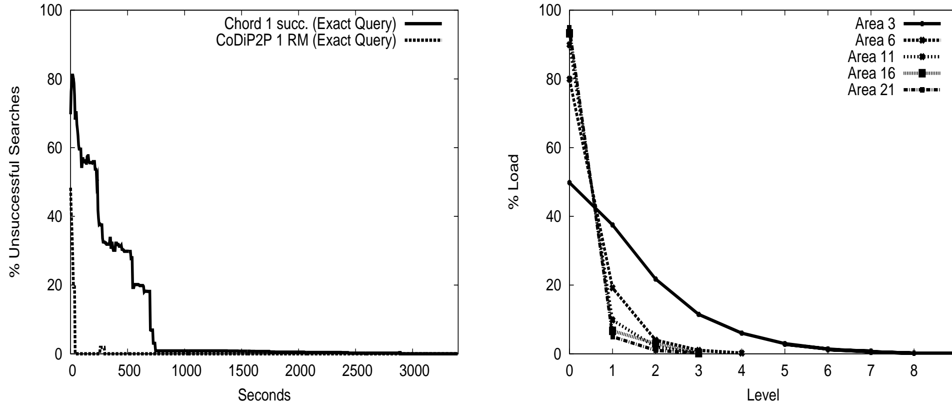

Fig. 6. CoDiP2P and Chord comparative of system time response due to a 50% of Failed Peers (left) and Tree topology congestion (right)

6 Conclusions and Future Work

In this paper, a computing resource discovery mechanism oriented to the CoDiP2P system is presented. Following the CoDiP2P architecture, the lookup mechanism is based on a balanced multi-way tree structure capable of supporting both exact and range queries efficiently. A rebalancing algorithm is also proposed.

The exact query proposal was compared with another exact query algorithm widely used in the literature, Chord. Given that Chord does not implement range queries, we also compared our range query proposal with a binary-tree structure, named BATON. In general, our results show that CoDiP2P performs much better than the other approaches, specially for high frequencies. In this case, CoDiP2P achieves half the rate of unsuccessful searches as the others. Likewise, the results obtained reveal the good performance of our rebalancing algorithm, improving the unsuccessful searches rate by 50%. Robustness was another goal of our work. In doing so, the response time to update the system completely after a massive drop of peers was measured. Our results show that CoDiP2P has a response time approximately 4 times faster than Chord.

The future trend is directed towards extending the tree topology with a second level topology, which will allow to increase the scalability of the platform. Each tree would group a set of peers according to any common characteristic (i.e. locality, computational resources, etc.), whereas the second level would connect the set of trees by means of a Bruijn graph[10], which is characterized by its high scalability and low congestion. Thus, we maintain the effectiveness of the tree topology for searching and overcome its main drawback, the congestion of the root levels for huge systems. Another important trend is testing and monitoring the topology and algorithms under real conditions and network. This will allow to measure the real traffic congestion and bottlenecks caused by the updating

algorithm, the communication times taken by the departure algorithm and the effectiveness and overhead messages of the rebalancing algorithm.

References

[1] I. Foster and A. Iamnitchi. "On death, taxes and the convergence of peer-to-peer and grid computing", 2nd. Int'l Workshop on P2PSystems, 2003.

[2] R. Gupta, V. Sekhri and A. Somani. "CompuP2P: An Architecture for Internet Computing Using Peer-to-Peer Networks", IEEE Transactions on Parallel and Distributed Systems, Vol. 17, No.11, 2006.

[3] N. Kotivalainen, M. Weber, M. Vapa and J. Vuery. "'Mobile Chedar - A Peer-to-Peer Middleware for Mobile Devices", 3rd Intl Conf. on Pervasive Computing and Communications Workshops (PerCom Workshops), 2005.

[4] D.Castellà, J.Rius, I.Barri, F. Giné and F. Solsona "CoDiP2P: a New P2P Architecture for Distributed Computing", Conference on Parallel, Distributed and Network-based Processing (PDP 2009), pp. 323-329, 2009.

[5] E. Meshkova, J. Riihijrvi, M. Petrova and P. Mhnen. " A Survey on Resource Discovery Mechanisms, Peer-to-Peer and Service Discovery Frameworks", Journal of Computer Networks, Vol. 52, pp. 2097-2128, 2008.

[6] D. Karger, F. Kaashoek, I. Stoica, R. Morris and H. Balakrishnan. "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications", In 2001 ACM SIGCOMM Conference, pp. 149-160, 2001.

[7] H. Jagadish, Beng Chin Ooi and Quang Hieu Vu. "BATON: A Balanced Tree Strcuture for Peer-to-Peer Networks", In the 31st Int'l Conference on Very Large Data Bases (VLDB) Conference, 2005.

[8] R. Buyya and M. Murshed, "GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing", Concurrency and Computation: Practice and Experience (CCPE), Vol. 14, Issue 13-15, pp.1175-1220, 2002.

[9] SimJava, http://www.dcs.ed.ac.uk/home/hase/simjava/, 1998.

[10] D. Loguinov, A. Kumar, V. Rai and S. Ganesh. "Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience", In Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications, 2003.

[11] E. Caron, F. Desprez and C. Tedeschi. "A Dynamic Prefix Tree for the Service Discovery Within Large Scale Grids". In the sixth IEEE International Conference on Peer-to-Peer Computing (P2P2006), 2006.

[12] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker and I. Stoica. "Complex Queries in DHT-Based Peer-to-Peer Networks". In the first International Workshop on P2P Systems (IPTPS'02), 2002.

[13] C. Schmidt and M. Parashar. "Squid: Enabling Search in DHT-Based Systems". Journal of Parallel and Distributed Computing, Vol. 68, N 7, pp. 962-975, 2008.

[14] C. Mastroianni, D. Talia and O. Verta. "A super-peer model for resource discovery services in large-scale Grids", Future Generation Computer Systems, Vol. 21, pp. 1235-1248, Elsevier Science, 2005.

[15] C. Mastroianni, D. Talia and O. Verta. "Designing an information system for Grids: Comparing hierarchical, decentralized P2P and super-peer models", Parallel Computing, Vol. 34, Issue 10, pp. 593-611, 2008.