Jelly: A Dynamic Hierarchical P2P Overlay Network with Load Balance and Locality

Richard Hsiao and Sheng-De Wang
Department of Electrical Engineering
National Taiwan University, Taipei 106, TAIWAN

Abstract

P2P systems based on Distributed hash table (DHT) such as CAN, Chord, Pastry, and Tapestry, use uniform hash functions to ensure load balance in each participant nodes. But the evenly distributed behavior in the virtual space destroys the locality between participant nodes. The topology-based hierarchical overlay network like Grapes, exploits the physical distance information among the nodes to construct a two-layered hierarchy, highly improves the locality, but damages the load balance property in original DHTs. In this paper, we propose a dynamic P2P overlay infrastructure, called Jelly. It can achieve both the load balancing and locality properties. Its design is based on the hierarchical overlay and uses the DHT as its routing algorithm. Because the load balancing issue in a hierarchical overlay is originated from whether the virtual hierarchy is balanced or not, Jelly uses a node joining mechanism as a fine-tuning tool and a dynamic checking mechanism as a coarse-tuning tool to balance the hierarchy. We also find that the average routing hops is a practical metric to evaluate the network size, and it is useful for Jelly's dynamic mechanism.

1. Introduction

In recent years, peer-to-peer (P2P) systems have been the burgeoning research topic in large distributed system. Gnutella [1] and Napster [2] are the most famous peer-to–peer file sharing systems among these, but both of them have the scalability problem. To address this problem, distributed hash tables (DHT) have became an fundamental part to build peer-to-peer overlay networks , CAN [3] , Chord [4] , Pastry [5] , Tapestry [6] are well-known works of these infrastructures. Many applications are layered above DHTs, such as file sharing systems [7] [8] [9], event notification services [10] [11], and application-layer multicast [12] [13] [14]. Although each of them has different location and routing algorithms, all of them have the same feature, using consistent hashing (like SHA-1) to let the participant nodes and objects distributed uniformly in its virtual space; in general condition, these systems can achieve fairly good load balancing property .

But the primitive DHT schemes have a significant disadvantage that they may violate the locality property. During the locating and routing process, the messages choose the next hop to a host regardless of the physical topology information. This produces inefficient effects in response time and overall physical path length for lookup service.

To address this problem, the DHTs should take into consideration of the relative physical position among the participant nodes. All of these systems have designed some similar approaches like [18], to exploit locality by measuring proximity metric like round trip time (*RTT*) or the IP level hops. This improvement assures the next hop selection is the relatively closer node on the underlying network that matches the routing condition, but the physical distance between the nodes looking for the object and the nodes storing that object could be still long. Grapes [15] provide the hierarchical virtual network infrastructure using physical topology information. It has two-layered overlay network, the upper layer called super-network, the lower layer called sub-network; in both layers, any DHTs routing algorithm can be used. Each sub-network has a leader joining the super-network routing and managing the sub-network. The physically nearby nodes construct the sub-network, and during each super-network query, the leader caches the object in its sub-network. Finally, a node can find the object in its sub-network with high probability, because the physical distance of any node pairs in sub-network is short, and thus this infrastructure can greatly reduce the lookup distance.

Although hierarchical overlay network like Grapes can highly improve the locality property of DHTs, it does not have the load balancing property. If DHT can provide load balance, then each leader in super-network is assigned to nearly the same load. After the lower-layered mapping, the load of each node in the entire system will no longer balance; the larger of the sub-network's size is, the lighter of the load will be assigned to its subnodes. Grapes does not provide any mechanism to adjust the size of sub-network, as a result of its node joining algorithm, producing some extremely large sub-network and a significant amount of sub-network with relatively few subnodes.

To address both the load balancing and locality problems, we propose Jelly, a dynamic hierarchical overlay network. Our main goal is to construct and maintain the well-balanced two-layered overlay network (the distribution of each sub-network's size within a given range), assure each participant node be assigned to similar load. Jelly's node joining mechanism is similar to Grapes. The difference is a newly-joined node not only checks the physical distance between each leader on the path in the inserting process and itself is shorter than the threshold or not, but also considers the size of the sub-network that each leader manages. If the size is larger than the given threshold, it is not appropriate to add one more node to this sub-network, because this may deteriorate the unbalance of entire hierarchy. Therefore, only when the newly-joined node finds a

Proceedings of the 24th International Conference on Distributed Computing Systems Workshops (ICDCSW'04)
0-7695-2087-1/04 $20.00 © 2004 **IEEE**

leader on the path in its inserting process, which is closer to the distance threshold and the size of its sub-network is smaller than the size threshold, then it joins to the sub-network of that leader; if there exists no such leader on the path in its inserting process to the super-network, then it will become a new leader without subnodes.

Another important topic in Jelly's design is how to maintain the balanced size of each sub-network. Peer-to-peer networks are a volatile environment, nodes joining and leaving frequently. This behavior makes the hierarchy more unbalanced, and the node joining mechanism solely is not sufficient to alleviate the unbalance. To let the hierarchy return to balanced state resiliently, Jelly provides a dynamic checking mechanism. Each leader periodically compares the size of its own sub-network to the average size of sub-network in the entire system or the size of super-network; if its size below the given allowable lower limit, it takes the sub-network merge process, which merges with another nearby and relatively small sub-network to form a larger one. In the simulation results, we will show that applying both the node joining and dynamic checking mechanism is can maintain the load balance.

Because Jelly is a fully decentralize system, it is impossible for leaders to calculate the amount of subnodes in its sub-network exactly, so we need a practical metric to evaluate the size of network. It is well known from [3][4][5][6][21] that every DHT has a monotonic increasing relationship between the number of nodes in overlay networks and the average routing path hops. Because the average routing hops within a sub-network is easy to obtain from the query messages through its leader, its tight relationship with the number of subnodes makes it be an efficient and simple metric to evaluate the size of sub-network. So even in the fully decentralize environment, we can provide a suitable metric to meet the requirement of the node join and dynamic checking mechanisms of Jelly.

The rest of this paper is organized as follows: Section 2 describes related work, Section 3 describes the fundamental hierarchical overlay network, Section 4 discusses the Jelly's design and properties, and Section 5 presents the simulation results, and concludes the paper in Section 6.

2. Related Work

In this paper, we assume that the consistent hashing implemented by original DHTs can produce good load balancing property; under this mapping mechanism, each participant node has the same load. This assumption is nearly matching the real condition with high probability, but it is still possible to result in unbalanced load distribution. Ananth Rao et al [19] proposed a scheme based on "virtual servers", every node in the system periodically check its own load, if it is overloaded, it re-arrange load with some "light" node. John Byers et al [20] suggested the "power of two choices" paradigm. In the object mapping process, a object uses more than one hash function mapping to a set of alternative nodes, and places it in the node of the least loads. Both of the above work can improve the load balancing property in DHTs significantly.

Both the Brocade [16] and SkipNet [17] are building an alternative overlay networks to address the locality problem in DHTs. In Brocade, they proposed a secondary overlay to be layered on the top of the original DHTs systems, and exploits knowledge of underlying network characteristics. The supernodes with high bandwidth and better processing power construct the secondary overlay. Each local node which wants to send wide-area messages first connect to the nearby supernode, then uses the second layer as a shortcut to the destination, this can greatly reduce the physical routing distance. Although its infrastructure is similar to our hierarchical overlay networks, not every participant node can be the supernode in Brocade, so it does not have a fully self-organizing mechanism. In SkipNet, they proposed an overlay network that can provide "content locality": limit the object to be placed in a given organization; and "path locality": guarantee the message which the source and destination nodes within the same organization will never be routed outside the organization. It realizes this property by organizing data primarily by lexicographic key ordering. It can not provide the load balance of entire system, but has a "constrained load balancing "property; in which object is evenly distributed in a defined subset of the nodes in the system.

Grapes [15] had proposed a self organizing topology-based hierarchical virtual network, our fundamental hierarchical overlay is based on Grapes, and we will describe its working principle in next section. Grapes can greatly improve the locality property from original DHTs, but it does not consider the load unbalancing effect due to the hierarchical overlay.

3. The Fundamental Hierarchical Overlay

In this section, we present the backbone architecture of our system, and this concept is first proposed by Grapes [15]. This overlay has two layers, the nodes physically near each other construct the sub-network, and the super-network is composed of the leaders of each sub-network. Both of the virtual networks can use any original DHTs (CAN, Chord, Pastry, Tapestry) as the locating and routing algorithm, and every subnode keeps a field recording its leader.

We assume that the newly-joined node can obtain one already existed node in the system, and then it uses this node as bootstrap node to insert into super-network. The inserting process is similar as original DHTs, but the newly-joined node checks the physical distance to each leader on its inserting path in super-network successively. If it meets a leader physically closer to the threshold, then it inserts into the sub-network of that leader and completes the inserting process. If no such leader on the path, it inserts into super-network, and become the leader of a new sub-network without any subnodes.

While a node inserts object into the system, it first inserts the object into its own sub-network by the hashed key of that object, then it request the leader of its sub-network to insert the object into super-network; its own leader finds the associated leader in the super-network's virtual space by the hashed key of that object. Finally, that leader inserts the object into the corresponding position in its sub-network, completes the inserting process of that object.

While one node looks for an object, it first searches the sub-network by the key of that object. If it fails, then it searches the super-network through its leader. After it finds the object outside its sub-network, it caches the object in the corresponding position in its sub-network. Consequently, every node will find the object in its sub-network with high probability.

When a leader fails, its own subnodes will aware this immediately during the routing process. Once a subnode discovers, it advertises the leader failing information to other subnodes in this sub-network, then the spare leader becomes the new leader and notices the rest of subnodes with leader-changing. The spare leader is selected from the subnodes; it maintains the leader's neighbor information in the super-network. So the spare leader can easily takes over the position of the ex-leader, and its original position is transferred to one of its neighbor in sub-network.

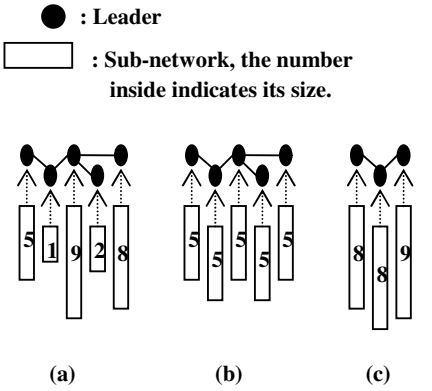

Figure 1: Compare three different situation of the hierarchical overlay. (a) Unbalanced hierarchy (b) Perfectly-balanced hierarchy (c) Partially-balanced hierarchy

4. The Design of Jelly

From the previous section, we have known that the topology-based hierarchical overlay is an appropriate approach to improve the locality property in peer-to-peer systems, but it does not preserve the inherent load balancing property of original DHTs. Here we make an assumption that, by the hashing scheme of original DHTs, all objects in the system distributed in the virtual space evenly. So if we wish to preserve the load balancing property, we should keep the hierarchical overlay balanced, and making the size of virtual space assigned to each node the same. Namely, the same load in each node. In Figure 1, we sketch three different possible states of hierarchy overlay; each state has 25 subnodes in entire system. In (a)'s hierarchy, the size of each sub-network differs greatly. As a result, the loading of the only subnode in the second sub-network (from the left side) will be nine times more than the subnodes in the third sub-network, and we should avoid the occurrence of such unbalanced hierarchy. Both (b) and (c) can maintain the load balance of each subnode in the entire system, because the size of each sub-network is almost equal, and (c) is sufficient to meet our requirement. But in (b) the size of the super-network is the same as the size of each sub-network, and (c) is not, this characteristic in (b) lets it roughly has $\sqrt{n}$ leaders in super-network and $\sqrt{n}$ subnodes in each sub-network with totally n nodes in the system. If the original routing algorithm guarantees $O(\log n)$ hops, (b) can guarantee $O(\log \sqrt{n})$ hops , greatly reduced the logical hops in the routing path. In Jelly, we can keep the partially-balanced hierarchy in (c) or the perfectly-balanced hierarchy in (b) easily, the only difference of these two keeping approaches is the system size metric, and we will discuss it next.

4.1 The Size Metric

The size metric is a key concept to construct and maintain the balanced hierarchy of Jelly. But in the fully decentralize environment, no single node has the capacity to exactly calculate the number of nodes in any subset of the systems, so we must find an alternative approach to evaluate the network size approximately.

In [3][4][5][6][21], the analyses show that the routing hops are bounded by some functions that increase with the number of nodes *N*. In Chord, the bound is $O(\log_2 N)$, CAN is $O(N^{1/d})$ for a given dimension *d*, and Pastry is $O(\log_{2^b}(N))$ for a given base *b*. And the simulation and experimental results in these papers also indicate that the average routing hops in each DHTs is monotonically increasing with the number of nodes. That is to say, if the average routing hops in one overlay network is more than another one, then its network size is larger than another with high probability. This explicit characteristic in DHTs inspires us to use the average routing hops as a metric to evaluate the network size.

To obtain the average routing hops, we just need adding a field in querying message to record the number of hops. While messages traverse to next node, the field will be added by one. Here we examine the revised lookup process in the hierarchical overlay step by step: If the source node cannot find the object in its sub-network, it will send the querying message to its leader. In this step, the leader can obtain the previous routing hops in sub-network, then the message will be routed to the leader of the destination sub-network, and this leader can obtain the routing hops in super-network. Finally, the messages will be routed to the destination subnode, and it can also send the hops record in its sub-network to its leader. So during a querying

process, the leaders can get three routing hops information at most, and one information at least (routed inside the sub-network).

To evaluate the average routing hops, leaders need not to keep the hops information for a whole period. Considering the space usage and volatile behavior in peer-to-peer system, leaders just need to record a reasonable quantity of recent hop information, and it is sufficient to evaluate the "current" network size.

We list all the notations and definitions of Jelly's system size parameters in Figure 2, they are all related to the system size metric, so the notations of them all starts with "***S***", to indicate the size. Jelly has two different system size metric, one is the size of super-network (*#super*) for perfectly-balanced hierarchy, the other is the average size of sub-network (*#avgsub*) for partially-balanced hierarchy. *#super* and the size of each sub-network (*#sub*) are easy to evaluate by the average hops by previous discussion. To evaluate *#avgsub*, leaders must obtain other leaders' *#sub* information. It is simple to satisfy this requirement by an additional message field of routing process in super-network. Every source leader appends its *#sub*, and consequently, each leader in routing path can gather others' hop information. Keep the current list, now it is easy to evaluate the *#avgsub*.

Notation	Definition
***S**threshold_ join*	The size threshold in node joining
***S**low_dc*	The lower size limit in dynamic checking
***S**threshold_sc*	The size threshold in selecting candidate
***S**up_merge*	The upper size limit in merging

Figure 2: Notations and definitions of the Jelly's system size parameters

```
node.Join(bootstrapnode)
   nodeID1=Hashing(node);
   leader=bootstrapnode.leader;
   suitableleader=null;// is a container to fill the
   leaders of suitable joining sub-networks
   while(the super-network routing not finished)
       d=Relativedistance(node, leader);
       s=leader.#sub;
       // sm is the system size metric
       sm=leader.#avgsub; // or sm=leader.#super
       if ( d < distancethreshold AND s < sm ×
                   Sthreshold_join)
           suitableleader.add( leader );
       else
           leader=leader.Nexthop(nodeID1);
   end while;
   if(suitableleader is null) //cannot find the suitable
leader
       joins supernetwork with nodeID1;
   else
       minleader=Choosemin(suitableleader);
       // choose the leader with minimum sub-network
size to join
       nodeID2=Hashing(node);
       joins minleader's subnetwork with nodeID2;
```

Figure 3: The pseudocode of Jelly's node *Join* algorithm

4.2 Node Joining Mechanism

Node joining mechanism in Jelly is not only a constructing scheme for balanced hierarchy, but can also be regarded as a fine-tuning mechanism for maintaining balanced hierarchy. Figure 3 shows the *Join* algorithm, the algorithm is similar to the one in original hierarchy overlay that we discussed earlier. It needs checking the physical distance information to join the relatively closer sub-network. The two significant differences are that we add a new size checking condition to assure the joining process will not make the size of that sub-network larger than ***S**threshold_join*×system size metric. The other chance is that we examine all the leaders in the routing path, not only single suitable leader, and pick the suitable leader of the smallest size as the leader of the newly-joining nodes. These two changes in joining process can reduce the amount of sub-networks with extremely large and small size. Another interesting point in *Join* algorithm is that the system size metric obtained from leaders can be *#super* or *#avgsub*, by which balanced hierarchy we want to construct, and this principle is the same for the following operations in Jelly.

4.3 Dynamic Checking Mechanism

It is not enough for only relying on node joining mechanism to balance the size of each sub-network; we need a more aggressive approach to let sub-networks actively adjust their size. Namely, splitting their sub-network or merging with other sub-network. But the splitting operation will cause relatively much smaller sub-networks in the system, and this condition let one AS has many sub-networks. It will gain less benefit of the locality in hierarchical overlay. So we only consider sub-network merge in Jelly.

And this aggressive approach is Jelly's dynamic checking mechanism. Every leader of the sub-network periodically compares the *#sub* that it evaluates from the hops information in its sub-network with ***S**low_dc*. If its *#sub* is smaller than ***S**low_dc*, it will start the Sub-network Merge process, merging its sub-network with other one.

The periodicity of checking can be adjusted to reflect the volatility of the changing networks, if a leader finds its *#sub* varying slightly from the recent checks, it can increase the period for future checking to reduce its processing loading; on the contrary, if it finds its *#sub* varying greatly from the recent checks, it should check more frequently to reflect the volatile environment.

The Sub-network Merge process will cause the redistribution of the virtual space of sub-network, so we should alter the routing table of these subnodes.

Fortunately, it will not complicate and just takes moderate bandwidth usages. If we use CAN algorithm, it can maintain the original relative position, and scales the coordinate in one dimension to fit the modified virtual space. The only nodes that need to alter its neighbor set are the ones which are adjacent to the merging "plane", the leader must notify these nodes that some neighbors should add to their neighbor sets. The majority of subnodes need not to change. In Chord's case, the remaining consecutive ring space makes the finger table partially correct, all we need to do is informing the nodes in the cutting edge (heads and tails) to correct their successors. Once the successors are corrected, the new sub-network can function correctly, and the stabilization process in Chord will update all the finger pointers in parallel. Regardless which DHTs algorithm we use, we just keep the original useful routing information and modify some parts of them. This modifying process is far more efficient than reinserting the subnodes to sub-network one by one.

4.4 Sub-network Merge

In the merging process, the leader of the sub-network which wants to merge with one "suitable" sub-network must have a feasible scheme to find the "suitable" sub-network. So each leader maintains a candidate list to store the suitable sub-networks' leaders; one leader can easily measure the physical distance form itself to these leaders, and then the information of physical distance and *#sub* of other leaders decide whether they are suitable merging candidates of that leader. We let the distance threshold be an invariant in all the operations, to assure the strong bond locality. To save the bandwidth usage and reflect the volatile environment, one leader does not need to measure the physical distance every time when a new message comes. It can apply variable measuring period, just like the dynamic checking mechanism we discussed earlier. If one leader measure that a leader is physically closer than the threshold and *#sub* of that leader is lower than ***S****threshold_sc* multiplied by the system size metric, then it is a suitable merging partner, and the leader will put it in the candidate list. The candidate list only keeps some recent entries, and all entries be sorted by candidate leader's *#sub* increasingly. The top entry in the list is the most suitable merging candidate by current information for future merging.

Once a leader starts the Sub-network Merge process, it first proceeded the *FindMergingPartner* algorithm (shown in Figure 4) to find the exact leader and its sub-network for merging. The algorithm first picks the top entry of its candidate list, it is the best candidate, and the leader will send a message to this candidate. If it is alive, it sends a message back appending its *#sub*, the leader uses this information to calculate the merged size. If the merged size is smaller than ***S****up_merge*×system size metric, this candidate leader and its sub-network are the exact partner of merging process. If this candidate is not alive or fails the size checking, the algorithm will pick the second, the third…….. suitable candidate to repeat the same procedure described above. If all the entries in candidate list are not suitable partner, the leader will increase ***S****threshold_sc* to loosen the restriction of selecting candidate, and the infinite loop in the algorithm will do this progressively until it finds the partner.

While the leader find its merging partner, its subnodes and itself will merge into this partner's sub-network to become the new subnodes. It takes a leaving process in super-network to inform one corresponding leader and its sub-network taking over its original loads. After the Sub-network Split process, the overall hierarchy will be much balanced and system still preserves the locality property.

```
leader.FindMergingPartner( )
  while( )// start infinite loop
    while(leader.Candidatelist NOT NULL)
      // pop the fittest candidate leader of list
      c =leader.Candidatelist.firstelement;
      // check the candidate is alive or not
      if (c is alive)
        s =leader.#sub+c.#sub;
        // the total size after merging can not be too large
        if (ts < Sup_merge×leader.#avgsub)
          // return the partner of merging process
          return c;
      end inner while;
      // if can not find the partner
      increase Sthreshold_sc;
      wait a short period of time;
    end outer while;
```

Figure 4: The pseudocode of *FindMergingPartner* algorithm

5. Simulations

Now we present some simulation results to evaluate the load balancing and locality performance in the Jelly design. In both super-network and sub-network routing, we used CAN [3] algorithm with dimension 3 and more uniform partitioning scheme.

To simulate the real world network environment, we selected BRITE [22] topology generator to generate 1000 Autonomous Systems (ASs), and used the heavy-tailed policy to place these ASs, observing the power-law distribution. We assumed that there are 10000 nodes in each AS. In each time of our experiment, we randomly picked 100 ASs of these 1000 ASs, and a given percentage of nodes in each AS to join the system.

Before the experiment, we had set the distance threshold and the system size parameters listed in Figure 2, we fixed the distance threshold to be 100, the ***S****threshold_join* and the ***S****up_merge* all 1.26, the ***S****low_dc* and ***S****threshold_sc* 0.79. The reason for setting the system size parameters in these values is based on the CAN algorithm, the average routing

hops grows as $n^{1/d}$; n represents the number of nodes, and d is the dimension of virtual space, here is 3. All the system size parameters are the upper limits or the lower limits of average path hops. So we set the upper limits $2^{1/3} = 1.26$, means double the size; the lower limits $(1/2)^{1/3} = 0.79$, means half the size.

First, we evaluated the load balancing performance by examining the volume of each node's zone in virtual space. Because we assume the uniform hashing can distribute the objects evenly in the coordinate space, so the volume of each node's zone represents the load stored on that node. Here we defined V, the average volume calculated from dividing entire coordinate space by the number of nodes in the system. We had experimented seven tests and the results was shown in Figure 5. In all the joining tests, the entire system had 16384 nodes, and we left 8192 nodes for the leaving tests, and inserted adequate amounts of querying messages during each 1000 joining or leaving messages in all Jelly's tests to let the leaders gather enough path hops information.

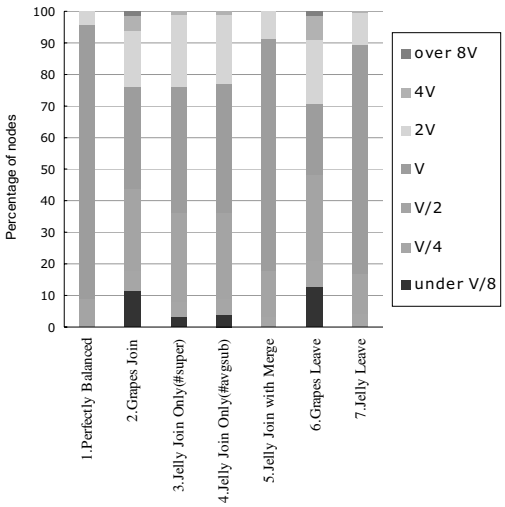

Figure 5: Comparing the loading distribution among the seven tests

In the Perfectly Balanced test, we joined 16384 nodes into 128 sub-networks to make each sub-network have the same size. Due to the CAN algorithm, the percentage of the nodes assigned to the volume equal V is not 100%, but about 85%. The largest volume is 4V and the smallest volume is V/4. We can regard it as the ideal case. In the Grapes Join test, only 32% of nodes with volume V, and has significant percentage of nodes over 8V and under V/8, the load distribution is so uneven in Grapes. In test 3 and 4, we used only the node joining mechanism in 4.2. Although the percentage of V has only a little improvement, the percentage over and under the limits is significantly reduced. Due to the size checking scheme in the node joining mechanism avoids producing the sub-network with extremely large size, and the little effort we paid in super-network joining makes the extremely small size sub-networks relatively few. And from the figure, we noticed the effects of choosing #avg*sub* or *#super* as the system size metric are similar. In test 5, we used both the node join scheme and dynamic sub-network merge scheme described in 4.3 and 4.4. Due to dynamically merge the smaller sub-networks, the heaviest load dropped to 4V, and 74% of the nodes are assigned to the zones with volume V. This experimental result shows that Jelly can produce good load balancing property with the joining process.

In the leaving test, we randomly chose half the nodes leaving successively to examine the performance of the dynamic checking mechanism in Jelly. From the Grapes Leave test, we noticed the leaving process deteriorates the load balance, due to lack of self-adjusted mechanism. And because of the high performance of Jelly's dynamic checking mechanism, in test 7, the loading distribution is almost the same with test 5. This showed that Jelly can dynamically adjust its hierarchy to reflect the variation in networks, and preserve the good load balancing property in original DHT.

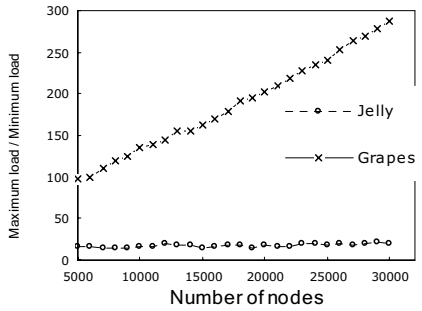

Figure 6: Comparing the load diversity in Jelly with Grapes

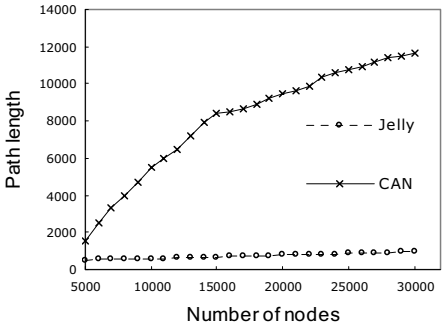

Figure 7: Comparing object lookup physical path length in Jelly with CAN

Figure 6 presents the load diversity property by examining the factor of maximum load divided by minimum load, while the total system size scaled from 5000 to 30000 nodes. By this experiment, we conclude that the good load balancing property can be preserved in

hJelly, even the system scales to relatively large size. Contrarily, in Grapes, the load distribution will be more diverse as the larger system size, due to the fully static constructing scheme.

Figure 7 presents the locality property of Jelly. To display the replicate scheme of hierarchical overlay network in limited simulation environment, we assumed the entire system stores 500 kinds of objects. In each system size condition, every node randomly picked an object from the 500 kinds to lookup. Due to the replications in sub-network, the path length in Jelly is much shorter than the physical path length in CAN, and scales well as the original hierarchical overlay, achieving good locality property.

6. Conclusion

Load balance and locality are two important issues in the design of current peer-to-peer systems. Jelly applies topology-based hierarchical overlay network as its fundamental infrastructure to improve the locality property of DHTs. We design a node join mechanism as a fine-tuning tool, and a dynamic checking mechanism as a coarse-tuning tool to balance the size of each sub-network, making the system can exploit the load balancing characteristic in DHTs. Our simulation results have demonstrated that Jelly is a scalable peer-to-peer overlay network with load balancing and locality properties.

In this paper, we regard the distance threshold as the invariant in the node joining and sub-network merging process to ensure the locality. In the future work, we consider designing an adaptive distance threshold mechanism, and constructing and maintaining a more flexible hierarchy to accommodate ever changing network environment.

7. References

[1]Gnutella http://www.gnutella.com

[2]Napster http://www.napster.com

[3]S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A Scalable Content-Addressable Network," In *Proceedings of SIGCOMM 2001*, ACM.

[4]I. Stoica, R. Morris, D. Karger, M F. Kaashoek and H.Balakrkshnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," In *Proceedings of SIGCOMM 2001*, ACM.

[5]A. Rowstron and P. Druschel, "Pastry: Scalable Distributed Object Location and Routing for Large-scale Peer-to-peer Systems," In *Proceedings of IFIP/ACM Middleware 2001*.

[6]B. Y. Zhao, J. D. Kuibiatowicz, and A. D. Joseph, "Tapestry: An Infrastructures for Fault-tolerant Wide-area Location and Routing," Tech. Rep.UCB/CSD-01-1141, UC Berkeley, EECS, 2001.

[7]F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, 2001.

[8]P. Druschel and A. Rowstron, "Past: Persistent and anonymous storage in a peer-to-peer networking environment," In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS 2001)*, 2001.

[9]J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, "OceanStore: An Architecture for Global-Scale Persistent Storage," In *Proceedings of ACM ASPLOS*, 2000.

[10]A. Rowstron, A. M. Kermarrec, M. Castro, and P. Druschel, "Scribe: The design of a large-scale event notification infrastructure," In *Proceedings of the Third International Workshop on Networked Group Communication,* 2001.

[11]L. F Cabrera, M. B. Jones, and M. Theimer, "Herald: Achieving a global event notification service," In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS 2001)*, 2001.

[12]Y. H. Chu, S. G Rao, and H. Zhang, " A case for end system multicast," In *Proceedings of ACM SIGMETRICS 2000*, 2000.

[13]S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "Application level multicast using content-addressable networks," In *Proceedings of the Third International Workshop on Networked Group Communication*, 2001.

[14]S. Zhuang, B. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz, "Bayeux: An architecture for wide-area, fault-tolerant data dissemination," In *Proceedings of NOSS-DAV'01*, 2001.

[15]K. Shin, S. Lee, G. Lim, H. Yoon, and J. S. Ma, "Grapes: Topology-based Hierarchical Virtual Network for Peer-to-peer Lookup Services," In *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW' 02)*, 2002.

[16]B.Y. Zhao, Y. Duan, and L. Huang," Brocade: Landmark Routing on Overlay Networks," In *Proceedings of the 1st International Workshop on Peer-to Peer Systems*, 2002.

[17]N. J.A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, A.Wolman, "SkipNet: A Scalable Overlay Network with Practical Locality Properties," In *Proceedings of 4th USITS*, 2003.

[18]M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron, "Exploiting network proximity in peer-to-peer overlay networks," presented at *International Workshop on Future Directions in Distributed Computing (FuDiCo)*, 2002.

[19]J. Byers, J.Considine, and M. Mitzenmacher, "Simple Load Balancing for Distributed Hash Tables," In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, 2003.

[20]A. Rao, K. Lakshminarayanan, S. Surana, R. Karp and Ion Stoica, "Load Balancing in Structured P2P Systems," In *Proceedings of the 2nd International Workshop on Peer-to Peer Systems*, 2003.

[21]M. Kelaskar, V. Matossian, P. Mehra, D. Paul, and M. Parashar, "A Study of Discovery Mechanisms for Peer-to-Peer Applications," In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID' 02)*, 2002.

[22]A. Medina, A. Lakhina, I. Matta, and J. Byers, "Brite: Univeral Topology Generation from a User's Perspective," Tech. Report BUCS-TR-2001-003, Boston University, 2001.