CMSC 711: Computer Networks

Fall 2002

Distributed Object Lookup Service Over NICE

Project Report

Group Members:

Hosam Rowaihy

Mohammad Hussein

Tamer Elsayed

24 December 2002

Contents

1. Introduction

The NICE overlay structure enjoys topology aware characteristics that enhance its stress and stretch properties over other proposed peer-to-peer overlays [Chord, CAN, Pastry]. In this project, we leverage the good stretch and stress properties of NICE to provide an efficient object lookup service for NICE group members.

NICE is an application layer-multicast protocol. It was designed to support applications that multicast data to a large number of receivers. To do this, NICE builds a hierarchical structure of members to form the control path. The basic operation of NICE is to maintain this structure as members join and leave the overlay. In the case of multicast, this path is used to form the data path. The NICE hierarchy is created by assigning members to layers with the lowest layer being zero (L_0). In each layer, members are divided into clusters of sizes between *k* and 3*k*-1, where *k* is a constant design parameter. Each cluster will contain members that are relatively close to each other in the actual network topology. Further, each cluster has a cluster leader which is elected distributedly and is the graph-theoretic center of that cluster. Leaders from each cluster in L_0 will form the next layer L_1 and the same thing is repeated. Every node must be a member in an L_0 cluster even if it is a member of a higher layer. So, a node which is a member in L_i must be a member in all layers L_{i-1} …. L_0 and not only that, it must also be the leader of its clusters in all these layers. Every node in each cluster sends a periodic *heartbeat* message to assure its neighbors that it is still alive and to update them with its latest information. When a node joins, it must find its closest cluster in L_0 and to do this it starts from the top of the hierarchy. For the join process to start, NICE suggests that there will be a well-known *Rendezvous Point* (RP) which knows who are the members of the highest layer in the current hierarchy. The join query will then proceed in each layer with the cluster that is closest to itself. Distance could be measured by delay or number of hops ..etc.

Another task that NICE must do to maintain the hierarchy is to keep cluster sizes in the bound (*k* – 3*k*-1). To do this each cluster leader will periodically check for the size of its cluster if the size is greater than 3*k*-1 it will split the cluster into two clusters and if the size is less than *k* then it well merge this cluster to another cluster. In the case of a split, the leader will divide the cluster into two clusters of equal sizes where the nodes that are relatively close to each other will be in the same cluster. In the case of a merge, the leader in a cluster in L_i will choose the closest node in the layer above the one which he wants do the merge (i.e. L_{i+1}). Then this node, which of course is a leader of another cluster in L_i will merge the cluster member with it own cluster members in L_i. When any change of the structure happens (i.e. a join, a split or a merge) then the leader on the affected cluster(s) may change. In that case, the old leader must remove itself form all higher layers it is a member of and the new leader must now join the next higher layer. If it became the leader of the cluster it has just joined, then it must continue joining to the higher layer and so on. By maintaining such a structure NICE is able to reduce the state and control overhead at any member to *O*(log *N*) where *N* is the number of nodes. Moreover, NICE has *O*(log *N*) stretch, where the log is of base *k*. These last properties lead us to think about implementing a distributed object lookup service over NICE.

2. Protocol Overview

In our project we separate the multicast from the hierarchy maintenance and only use the latter to provide the new service. The distributed object lookup over NICE will support two extra basic operations. One is to insert a key into the system and the other is to retrieve a key from the system. The application that will be using this service will decide on whether the key should hold an object (e.g. a file) or pointer to the node that holds the object (e.g. IP address). Keys are generated using *consistent hashing,* of object name or any other identifier of that object. The basic property that makes consistent hashing appealing is that it tends to distribute the keys evenly throughout the key ID space. In our project, we suggest to use two different hash functions, one for what we call a *global key ID* that will map the key to a certain cluster in layer zero and the other for what we call *local key ID* to map the key to a certain node in that cluster. To be able to implement this, we will have each cluster holding a range of the global key ID space (called *Cluster Range*) and require each node to have a *local ID* in L_0 only. Cluster ranges will be represented in the form [*a b*] meaning that this cluster holds keys with global key IDs that have values between *a* and *b*. Each node in a layer will store the range of the cluster that it is the leader of in the lower layer. We will call this range *lower cluster range*. If it is not a leader of any cluster then that range can be considered as [0 0]. It will also store the range of the cluster it is currently member of which we will simply call *cluster range*. The local IDs must be unique within the cluster but not necessarily globally unique. We define the *successor* of a node as the node that directly follows it in the local ID space in the same cluster. Nodes in L_0 cluster form a logical ring so that the last node's successor is the first node. This is shown in Fig. 1 below where the numbers inside the circles indicate local IDs.

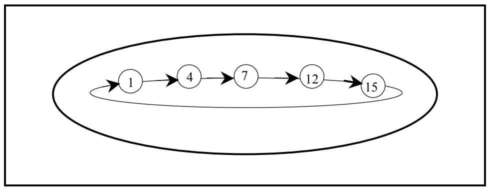


Figure 1. A Cluster in layer zero showing Local IDs.

When a key is inserted it is mapped to the cluster in L_0 that has a range in which its *global key ID* falls. Within that cluster the key is mapped to the node which has a *local ID* that directly follows the *local key ID* or in other words the successor of the *local key ID*. Similarly, when a key is retrieved, the cluster holding that key must first be located then in that cluster the node that should be holding the key is identified. The hierarchal topology of the NICE overlay can be used to efficiently locate objects in $O(\log N)$ overlay hops.

When any change in the structures happens, some keys may be transferred so that their mappings will be correct. The different cases in which keys are transferred are explained in details below. We have also modified the join, split and merge processes to reflect the added information each cluster and node should have, namely the local IDs and Cluster Ranges. The edges of a range will clearly change when the cluster splits into two clusters. In this case, the range must also splits into two smaller ranges one for each new cluster. This is also true when a merge happens in which case the cluster range of the bigger cluster resulting from the merge will either become bigger or will not change and the range of another cluster will increase. All these changes in the original NICE protocol will be discussed later.

3. Protocol Details

Several changes have been made to the original NICE protocol. First, the join process was modified mainly to enable the assignment of the newly added local IDs and cluster ranges and to do transfer of keys to the newly added node if needed. Second, the split process was changed so that the newly generated clusters are assigned their cluster ranges properly. We also differentiate between a split that happens in L_0 and a split that happen in any higher layer. Third, we modified the cluster merge process for the same reason as for the split and we also differentiate between a L_0 merge and higher layers merge. In both split and merge we may also need to transfer some keys. Finally, extra functions were added to insert, retrieve and transfer keys. The details of all these changes and more are presented in this section.

The project mainly was divided to three phases. The first phase was the task of understanding the details of the NICE protocol implementation. The second was the phase of assigning local IDs and maintaining cluster ranges even after a change in the hierarchy. The third and final phase was to introduce keys to the system and manage the insertion, retrieval and transferring of these keys.

3.1 The Join Process

In the join process of NICE, a node will first contact the RP which will give it a list of nodes in the highest layer. Then the node will choose whichever node is closest to it, in the network topology, and sends it a join query. The join query will go down the hierarchy until the joining node reaches the closest leader in L_0 and asks it to join in its cluster. At this point the join process ends in the original protocol. We added to this process that when a leader in an L_0 cluster receives a join query it will assign the joining node a, randomly chosen, unique local ID in that cluster. Clearly we need at least $3k$ local IDs in each cluster since after the number of nodes reaches $3k$ a split should happen. However, the leader checks for its cluster size after every timeout (e.g. 20 seconds), so at some point of time between two successive timeouts we can have more than $3k$ nodes in a cluster. That's why we decided to have the number of local IDs greater than $3k$. Another reason for having a larger number of randomly assigned local IDs is to have the local IDs evenly covering the local ID space in a cluster which will help to balance the number of keys held per node.

Along with assigning a local ID for the new node the leader must also provide the cluster range of this cluster. When the first node joins the system it will be assigned a local ID and its cluster range will be the whole global key ID space. In this case, the information will be provided by the RP which must be preconfigured with the global key ID space.

3.2 The Split Process

In the NICE protocol, when the leader of a cluster notices that the members in its cluster are more than $3k$-1 it will initiate the split process. The result of this process is the creation of two equal clusters each containing nodes that are relatively close to each other. In our modification, we differentiate between a split hat happens in layer zero and a split that happens in any layer above zero.

Splitting in Layer Zero

In layer zero the split process will be as the original NICE. The cluster is divided into two equal clusters. Nodes that are relatively close to each other will go into a single cluster. The range of the original cluster is divided into two equal sizes and given to the new clusters. For example a cluster holding the range [16, 23] will split into cluster [16, 19] and [20, 23]. Fig. 2 shows a similar example. But, which cluster will take [16 19] and which will take [20 23]? This can be decided according to the distances between the new clusters and the clusters adjacent to them in cluster range. For example, if there exists cluster [8 15], which is adjacent to [16 19], and cluster [24 28], which is adjacent to [20 23] then we need to measure the distance between the two new clusters and both these clusters. The new cluster which is closest to [8 15] will be assigned cluster range [16 19] and the one which is closest to [24 28] will be assigned cluster range [19 23]. This enhancement is not yet implemented.

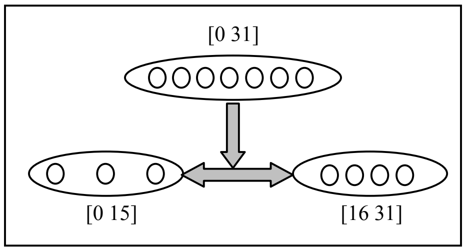


Figure 2. Example of a split in layer zero

Splitting in Higher Layers

In all layers above layer zero, the split is done by considering the ranges and not the distance between nodes. To show how this works let us consider the following example in Fig. 3. When the leader of cluster [0 63] in layer higher than zero notices that its size has more than 3*k*-1 (5 nodes in this case) it initiates the split process. To split, the leader will sort the member nodes according to the ranges each one covers. After sorting them, it will take the first half and put them in a cluster and the second half in the other cluster. The resulting range of the first cluster will be the union of all the ranges making that cluster. In our example the first range will be [0 23]. The other cluster will be also be the union of the ranges in our case it will be [24 63].

We argue that if layer zero clusters are arranged in a topology aware manner, which is the case in NICE, then with high probability clusters with adjacent cluster ranges will actually be close to each other in the underlying network topology. That's why we have decided to do cluster split in higher layers by considering the ranges and not the actual location.

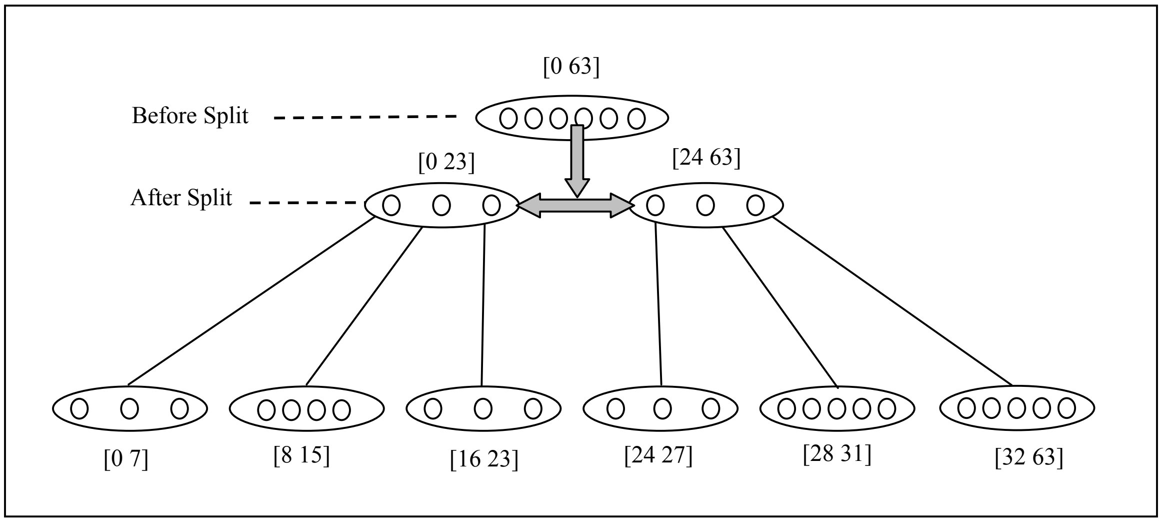


Figure 3. Example of split in a higher layer

3.3 The Merge Process

When the number of nodes in a cluster falls below *k*, the leader should merge its cluster with another cluster. In the original NICE protocol, the leader will choose its closest peer in the next higher layer in terms of distance and sends it a merge request. In our modification, as in the case of a split, we differentiate between a merge that happens in layer zero and a merge that happens above layer zero.

Merging in Layer Zero

In layer zero our modification uses the same technique used by the original NICE protocol in which the leader chooses its closest peer in the next higher layer and sends it a merge request. There are two possibilities for that peer, either it has a lower cluster range that is adjacent to the leader's lower cluster range or it has a range that is not related to the leader's range. In the first case, the two clusters will merge generating a bigger cluster with a cluster range equaling the union of the two ranges. However, if the ranges are not related, then the leader will give out its range to a cluster that is adjacent to it and then it will migrates its nodes to the closest cluster that was found. There could be two clusters that are adjacent to the cluster which wants to merge. In that case, the merging cluster will give out its range to the one that has the smaller range. This will help to balance the load (in terms of keys per node) between clusters. Examples of both cases can be seen in Fig. 6 and Fig. 7.

Merging in Higher Layers

In higher layers, when a leaders wants to merge its cluster with another cluster it will always choose a cluster that is adjacent to it. So, the generated cluster will have a range which is the union of the two ranges. When a cluster has two adjacent clusters it will choose to merge with whichever is closest in the underlying topology. As with the split, we argue that with a high probability the cluster which is the closest in the underlying topology will be the one that has an adjacent cluster range.

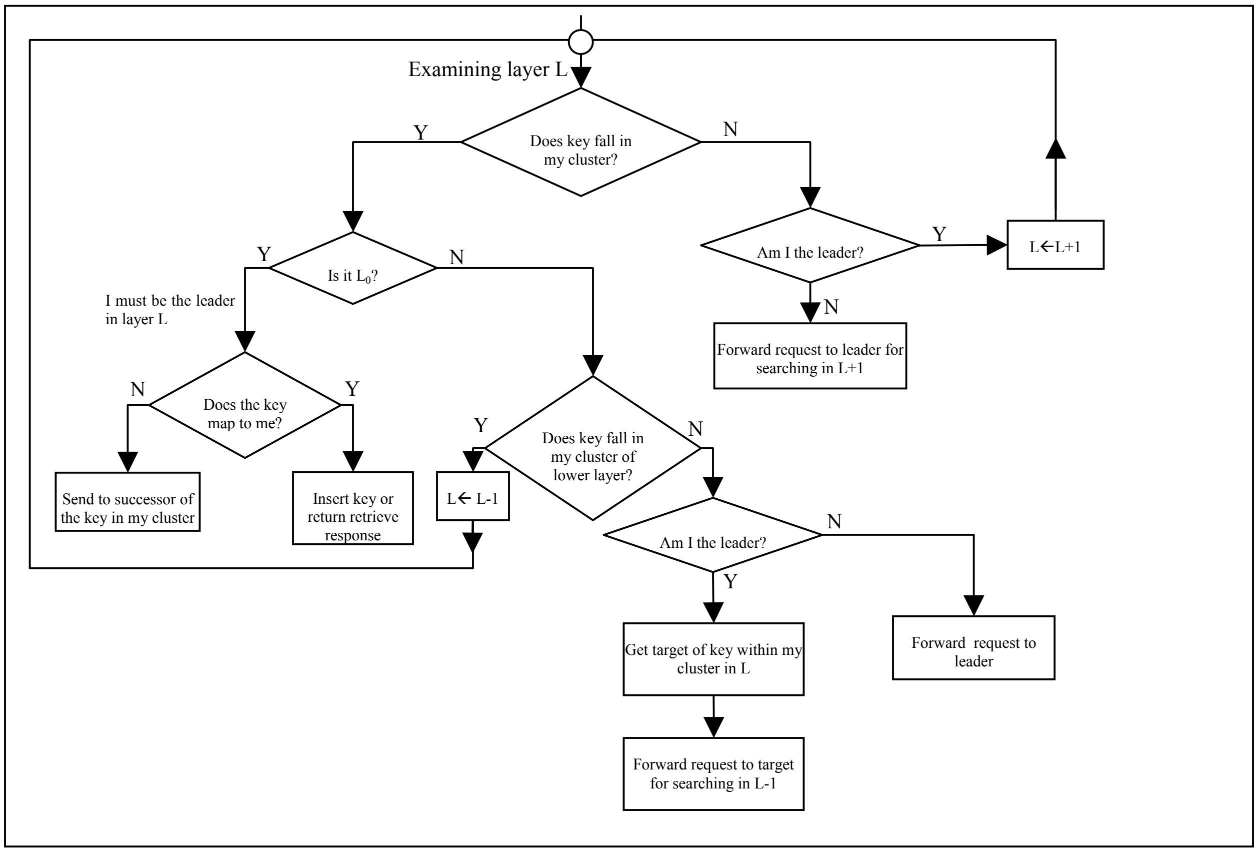
3.4 How to Deal with Keys

The main contribution of the project is this part. We have added facilitates to the NICE protocol to handle key operations namely insert key, retrieve key and transfer key. In this section each of these new operations are discussed in details. One thing to notice is that all key operations are initiated in L_0 and not in any other layer. This is clear since all nodes are members of L_0.

3.4.1 Insert and Retrieve Key

Both insert key and retrieve key must do the same thing; i.e. find the node that a certain key is mapped to. In the case of a key insertion, when this node is found the key is stored in it. On the other hand, if we are retrieving a key, then after finding the node that the key map to, that node will return the address of the node that has the actual object. Of course, this can be modified so that nodes will hold keys and the actual object in which case when we are retrieving a key we will get back the object itself. The response will also specify if the key is not found in the expected location.

When a node wants to insert or retrieve a key it would first check it has the key locally if not the request need to be routed. The way the routing is done is shown in the next flow chart (Fig. 4):



Examining layer L
Does key fall in my cluster?
Y
N
Am I the leader?
Y
L←L+1
N
Forward request to leader for searching in L+1
Is it L_0?
Y
N
I must be the leader in layer L
Does the key map to me?
N
Y
Send to successor of the key in my cluster
Insert key or return retrieve response
Does key fall in my cluster of lower layer?
Y
N
L← L-1
Am I the leader?
N
Y
Get target of key within my cluster in L
Forward request to leader
Forward request to target for searching in L-1

Figure 4. The algorithm for routing key insert and key retrieve messages. Target means the node in layer L that has cluster range that contains the global key ID.

To see how this routing algorithm works, let us do the following example in Fig. 5:

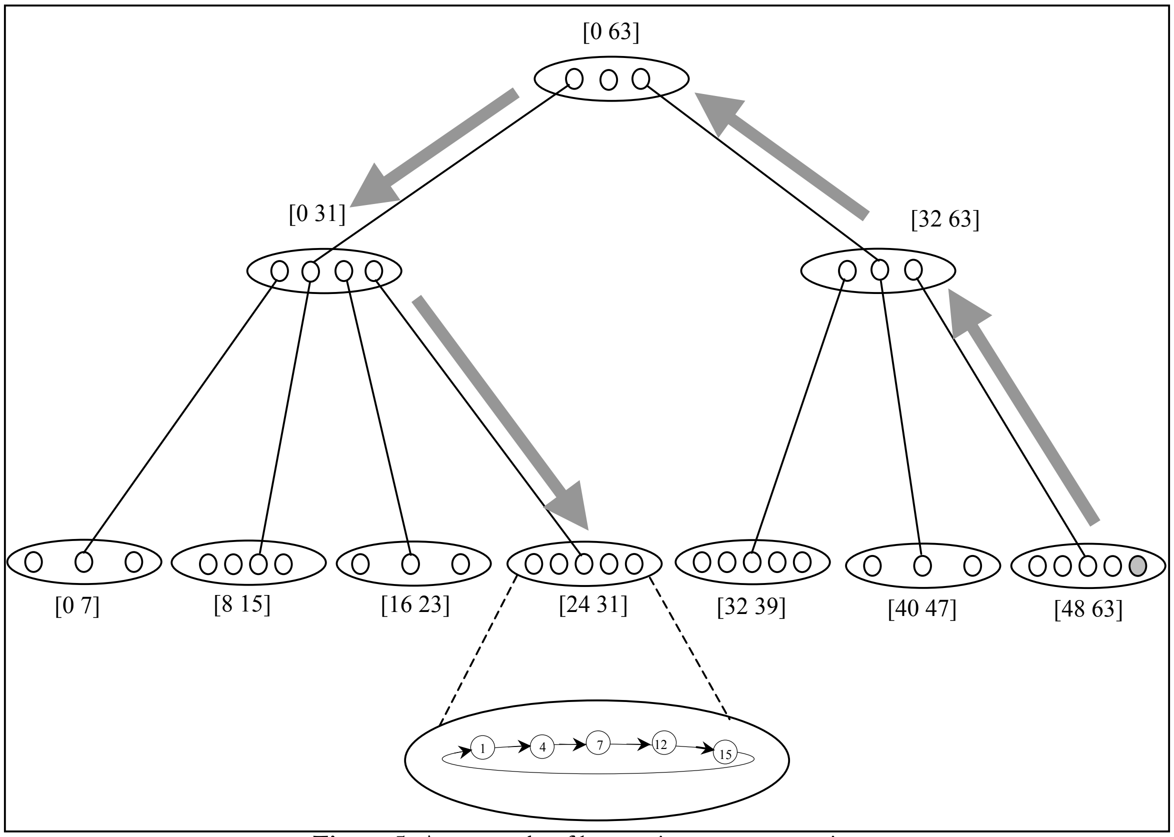


Figure 5. An example of key retrieve request routing

First the grey node on the rightmost cluster issues retrieve key request. Let us assume that the key is (29, 5) where the first number is the global key ID and the second number is the local key ID. Both these numbers are generated by hashing a quantity such as the object name using two has functions. First, the node who requests the key will look if the key falls in the range of its cluster. Clearly this is not the case here so it will send the request to its leader. The leader will notice that the key is not in its range so it will forward the request up the hierarchy to the leader of its next higher layer. Of course, the same node could have been also the leader in the next higher layer but in our case it is not. Now, the leader of the cluster [32 63] will check if the key falls in its range which clearly is not true. So, it will forward the request to the next layer. In the next layer, the leader discovers that the key (global key ID) falls in its range. Next, it finds to which of its cluster members the request should be forwarded. That member should have a lower cluster range in which the required global key ID falls. It finds out that the required member is the leader of cluster [0 31], so it forwards the request to it. Next, the request is forwarded to the leader of cluster [24 31]. Now that the request has reached the required cluster, the leader will look at the local key ID and find its successor. Since the local key ID is 5 its successor will be node 7 (see the blown up figure). So, the request is forwarded to node 7. When node 7, finally, receives the request it will figure out that it is the required node so it will look into its key list and send the requester the address of the node holding the actual object. If this request was insert key instead of retrieve key then the required node would have stored the key in its key list and may return a confirmation to the requester assuring it that the key is stored.

As we can see from the example above the request may travel up and down the hierarchy to reach its target. Although it may need a formal proof, it is clear that any request will travel through $O(\log N)$ nodes to reach it target.

It may happen in some cases when the network is still stabilizing after a change in the hierarchy that a certain key is not found although it is in the system. This may happen rarely and since the NICE system stabilizes fairly quickly (about 20 seconds) a simple retry will find the required key.

3.4.2 Transfer Keys

Keys are transferred in three different situations that may alter the L_0 structure. First, when a new node joins the system in which case it may need to get some keys from its successor. The second situation is when a split occurs in L_0. The transferred keys in this case are the ones that have global key IDs that do not fall in the ranges of the newly created clusters. Transfers will only be between the new cluster and it will not involve any other part of the structure. Third, when a merge happens in $L_{0.}$ In case all the keys from the cluster that is merging are transferred either to the cluster which it is merging with or to another cluster according to how did the merge happen. Each case is explained in details below. Although transfer key may seem as a different kind of operation it is actually implemented as insert key and will use the same routing scheme described above. However, transfer key is something that is done automatically without any interaction from the user.

Join Transfers

Each node will periodically check if it has a new predecessor in the local ID space. If one is found keys that belong to this new node are transferred to it. For example, looking at the Fig. 1 above, let us assume that the node with local ID 7, node 7 for short, has just joined this L_0 cluster. Before node 7 joined the cluster node 12 had all the keys that have $4 <$ local key ID ≤ 12. So, when node 12 first learns about node 7 it will check in its key list for keys that should be mapped to node 7 instead of itself. These keys are the ones having $4 <$ local key ID ≤ 7. Of course, we can have can have two or more new nodes that joined between a node and its predecessor. In that case, keys will be transferred to the appropriate nodes according to their local key IDs. Clearly join transfers happen within a single cluster and does not affect any other cluster.

Split Transfers

When a split happens in an L_0 cluster the range is divided into two equal sizes and given to the newly created clusters. After splitting, each node in both new clusters will sense the change of cluster range by the heartbeat messages it receives from the leader. When this happens it needs to go over all its keys to check if any of them has a global key ID that falls in the other half of the range in which case they will be transferred to the other cluster. Since consistent hashing is used for generating global key IDs on average half the keys would satisfy this. These split key transfers happen between the newly created clusters only and do not affect any part of the hierarchy. As an example let us look at Fig 2. First, there was one cluster with range [0 31]. When the leader of that cluster figured out that the number of nodes had exceeded $3k$-1 it decided to split the cluster. Two cluster were created the right one with range [0 15] and the left one with range [16 31]. After the split, nodes in [0 15] must look for all the keys with global key IDs ∈ [16 31] and transfer them to the cluster on the left. Also, nodes in the other cluster must look for keys with global key IDs ∈ [0 15] and transfer them to the right cluster.

Merge Transfers

A merge in L_0 can happen in two different ways. One is when the leader of the cluster which wants to merge finds out that his closest peer in L_1 has a lower cluster range that is adjacent to its own lower cluster range. In this case the two clusters will merge forming a bigger cluster and the range of that cluster will expand to include both ranges. When this happens nodes in the merging cluster will be assigned local IDs in the cluster which they merged with. Nodes will notice that a merge has happened by the heartbeat messages sent by the leader. Although that keys will stay in the resulting cluster (all keys will fall in the bigger range), keys maybe transferred inside that cluster to nodes with appropriate local IDs. Considering Fig. 6 below, when a merge happens between cluster [0 15] and cluster [16 31] the resulting cluster will be [0 31]. The nodes in that cluster must check if the keys still map to them after the merge. Specifically, they need to make sure that each one of them is the successor of the keys it holds in the bigger cluster. If there are any keys wrongly mapped they must be transferred to the appropriate nodes. Clearly, these transfers will only affect cluster [0 31] and no other cluster.

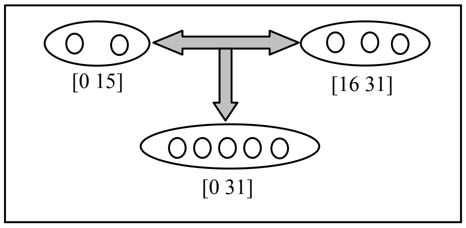


Figure 6. Example of merging with an adjacent cluster

In the other way, the leader of the cluster that wants to merge may find its closest peer in L_1 to have a lower cluster range which is not adjacent to its own lower cluster range. In that case, the leader will give its own cluster range to one of the clusters that is adjacent to it in cluster range. Then the cluster will merge with the closest cluster found and the resulting range will not change (i.e. it will remain as if no merge has happened in the cluster). After the merge, nodes will sense the merge by messages received from the leader. Then, nodes that sense they have a totally new cluster range will transfer all their keys to the cluster that got their old range. From there the keys will be given to the specific nodes with the appropriate local IDs. For example, if we look at Fig. 7 below, we see that cluster [0 15] wants to merge and found that its closest cluster, in actual network topology, is [48 63]. In this case, it will give its range to the cluster with the closest range which is [16 31]. After the merge, [16 31] becomes [0 31] and nodes in [0 15] will merge with cluster [48 63]. Now, all nodes from the old [0 15] will transfer their keys to cluster [0 31] since the global key IDs will map to it.

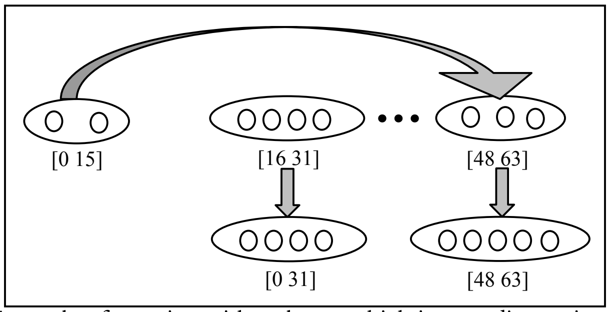


Figure 7. Example of merging with a cluster which is not adjacent in cluster range

3.5 Dealing with Node Failures

The problem with node failures is that when a node fails all the keys that it holds will be lost. To solve this problem one can use *r* different hash functions to generate r different key IDs which means that each key will be stored in *r* different locations. This will require the failure of *r* nodes for the key to be lost. One advantage of this method over, for example duplicating keys in the same cluster, is that keys will be distributed all over the network and the failure of a part of the network (which may be a whole cluster) will not lead to losing all the replicas of a key. It is clear that geographically close nodes have a higher probability of failing together than nodes that are far apart. Another advantage of using replication is that when retrieving a key we can always choose the closest replica to return. To do this, when a node wants to retrieve a key it will request all the *r* replicas instead of only requesting one key ID. Then, when proceeding with routing the key request nodes will forward only requests that fall within their ranges. If none falls in its range, then it will forward all requests to appropriate nodes as shown in Fig. 4 above. This will guarantee getting the closest replica first which will improve performance. However, when we are inserting a key we must proceed with inserting all the *r* key replicas. Clearly, this will require modifying the routing algorithm mentioned above. Techniques to deal with node failures are suggested but not yet implemented.

3.6 Added Overhead

To implement object lookup service over NICE we added some extra types of messages along with new state values that must be stored in each node. However, the order of states stored in a node is still $O(\log N)$. As of the new required state, each node must store the following in addition to what the original NICE does:

1. *Local ID*: the nodes local ID in L_0.
2. *Cluster Range*: the range of the cluster this node is member of.
3. *Lower Cluster Range*: if the node is a member in any layer above L_0, this will store the range of the directly lower layer cluster which the node is leader of.
4. *Key List*: a list of the keys currently stored in the node.

The new message types are as follows:

1. *Peer Info*: is used to inform a new node of its local ID and the cluster range to which it is joining.

2. *Insert Key*: is used to insert a key in the NICE system.
3. *Retrieve Key*: is used to find a certain key in the system.
4. *Retrieve Key Response*: contains the key value, which in our implementation is the address of the owner of that key. It also returns "key not found" if the key could not be located.

Some messages of the original NICE where modified to enable offering object lookup mainly the refresh messages (*heartbeats*). In refresh message, we added information about the peers in the same cluster such as their local IDs and their lower cluster ranges.

5. Modification of the Original NICE Protocol

While we were studying the implementation of NICE we noticed what could be considered a flaw in the splitting algorithm used. The algorithm which is used is as follows:

1- Pick 2 nodes and try them as new centers (leaders).
2- Partition all cluster nodes into 2 separate sets S1 and S2 according to their distance from centers (nodes closer to center 1 will go to S1 and nodes closer to center 2 will go to S2).
3- Measure max-radius of each set. If it is the smallest so far then keep it.
4- **If** size of one set is less than the required size.

(partitioning is unbalanced in size; i.e. we have one set larger than the other)

Then a- sort nodes of the larger set in ascending order according to their distance from the center of the **larger** set.

b- transfer the first *k* nodes (*k* = minimum cluster size – the smaller set size) from the larger set into the smaller set.

5- Repeat for all other pairs of nodes

We noted the following about this algorithm: If transfer is needed (as shown in Fig.8) then it is possible for the radius of the smaller set to increase after transfer (refer to Fig. 9). This is clearly not the wanted result. The algorithm didn't even update the smallest radius after transferring some nodes from larger to smaller sets. To solve this we suggest to sort nodes of the larger set according to their distance from center of the **smaller** set and update the radius after this operation. Resulting clusters after transfer using our suggestion are shown Fig. 10.

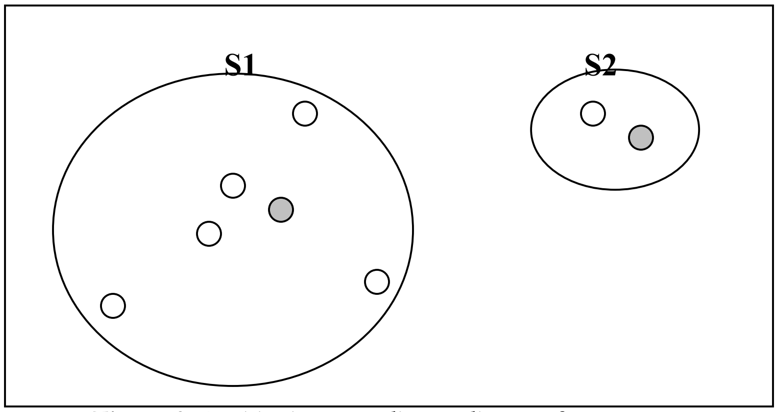


Figure 8. Partitioning according to distance from centers

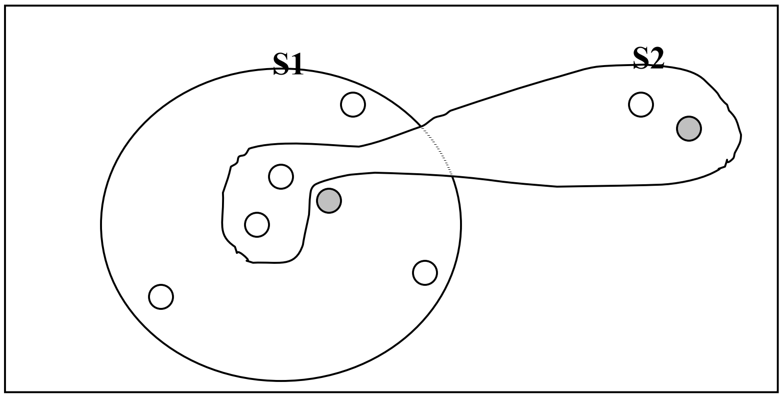


Figure 9. Transfer of current algorithm. Note that radius of S2 is increased

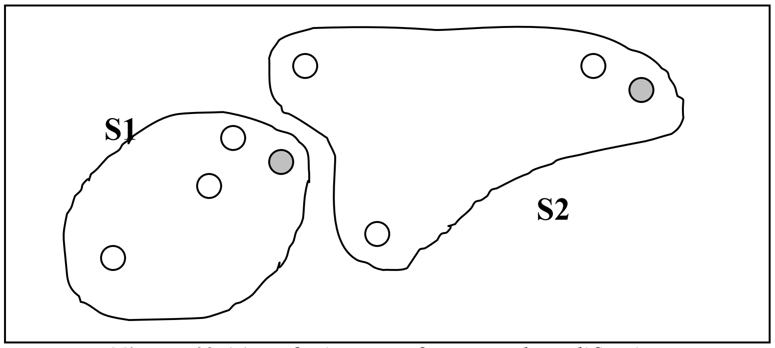


Figure 10. Transfer in case of suggested modification

6. Member Contributions

It is fairly difficult to say who did what in this project since most of the time we worked as a team. The first part was the discussion and brainstorming to come up with a solution to the main problem which is how to modify NICE to let it handle keys. Every team member had his input in the final idea so it is not a single person thing. As of the second part, which is the implementation, it was divided into three main phases. The first was the understanding of the NICE implementation which was done mainly as a group. The second phase was the assignment of local IDs and cluster ranges. In this phase we worked on parallel, namely Hosam worked on modifying the join process, Mohammad worked on modifying the merge process and Tamer worked on modifying the split process. The third phase was the management of keys. In this phase, Tamer implemented the functions to handle key insertion, retrieval and routing. Hosam continued to work on the join process and dealt with key transfers that occur after each join, while Mohammad worked on the transfer of keys that happens after a split or a merge. Debugging was mainly done by Mohammad and Tamer while the final report was written by Hosam.

7. Conclusion

In this project we have implemented a simulation of an object lookup service built over the NICE hierarchy. Our proposed system leverages the good stretch and stress properties of NICE to provide an efficient service. The stretch properties of NICE help our structure to insert and retrieve keys in $O(\log N)$ hops. Although many other peer to peer object lookup services were proposed, we think that using NICE to provide this service has some better properties since the structure is topology aware.

Completing this project was very challenging and we think that we still have many things to do. First, we would like to have a complete performance analysis of the project. We would also like to compare the results we obtain with that of Chord, CAN, Pastry and Tapestry. As, we mentioned in the report there are several enhancements that can be made, so we would like to try implementing these enhancements and see if they would boost the performance as we expect.