Tourist: Self-Adaptive Structured Overlay

Jinfeng Hu
Department of Computer Science and Technology
Tsinghua University
Beijing, P. R. China
hujinfeng@mails.tsinghua.edu.cn

Ming Li
Department of Computer Science
University of Massachusetts Amherst
Amherst, Massachusetts, U. S.
mingli@cs.umass.edu

Hongliang Yu
Department of Computer Science and Technology
Tsinghua University
Beijing, P. R. China
hlyu@tsinghua.edu.cn

Weimin Zheng
Department of Computer Science and Technology
Tsinghua University
Beijing, P. R. China
zwm-dcs@tsinghua.edu.cn

***Abstract*—Structured overlays provide a basic substrate for many peer-to-peer systems. Existing structured overlays can be classified into two categories, O(log*N*)-hop ones and O(1)-hop ones. The former are suitable for large and dynamic systems, while the latter are suitable for small or stable ones. However, for the lack of adaptivity, it is difficult for a peer-to-peer system designer to choose from these two kinds of overlays because the eventual size and churn rate are not easy to predict in the design phase. To address this problem, we propose a self-adaptive structured overlay *Tourist*, which can adapt itself to the changing environment dynamically. On the one hand, Tourist can achieve 1-hop to 2-hop routing in most cases (e.g., in a 1,000,000-node system where nodes' average lifetime is only 1 hour). On the other hand, when the system size is extremely large or the nodes' churn rate is very high, Tourist can always guarantee O(log*N*)-hop routing for all the messages. Tourist nodes determine their routing table size autonomously: nodes with different capacities hold routing tables with different sizes. This makes Tourist sufficiently utilize all the nodes' allowable bandwidth to achieve as high routing efficiency as possible. Tourist also allows each node to adjust its routing table size dynamically, which is the essential reason for the self-adaptivity. Simulation results show that in a common 1,000,000-node system, Tourist can route all the messages within two hops and each node only pays no more than 1% bandwidth for its routing table maintenance. Only when the churn rate is very high (nodes' average lifetime being less than 1.5 minutes) Tourist would turn to an O(log*N*)-hop overlay where there are some messages traversing more than two hops.**

Keywords-structured overlay; peer-to-peer; DHT routing

I. INTRODUCTION

Structured overlays provide a basic substrate for many peer-to-peer systems, such as DHT-based stores [1][24][23], application-level multicast protocols [2], web caches [14], and backup systems [5]. Formally speaking, most structured overlays provide the following communication semantics:

In a virtual space, each node occupies a point, denoted as *nodeId*. Each message is assigned a destination *key*, which is also a point in the virtual space. A structured overlay guarantees that any message would be eventually sent to the node whose nodeId is the nearest to the key among all the live nodes. The concept "close" depends on the *distance* metric in the virtual space. Many existing structured overlays [25][27][15][18] use numerical space, i.e., nodes and keys are all 128-bit numbers, and define two points' distance as their numerical difference.

By far, many structured overlays have been proposed. In general, they can be classified into two categories, O(log*N*)-hop ones and O(1)-hop ones. O(log*N*)-hop overlays [25][27][10][19] let each node keep a small routing table, commonly containing log(*N*) pointers (*N* is the total number of the live nodes), and route messages via log(*N*) hops. Most O(log*N*)-hop overlays use explicit probes for routing table maintenance, i.e., each node sends heartbeat messages to all the nodes in its routing table periodically to detect whether they have left. Because routing tables are small, the cost of such probes could be kept very low [3].

On the contrary, nodes in O(1)-hop overlays keep large routing tables, commonly containing O(*N*) or O($\sqrt{N}$) pointers. Larger routing tables provide more efficient message routing, one-hop [7] or two-hop [8][9][28][12], but also induce larger bandwidth cost for routing table maintenance. Though almost all the O(1)-hop overlays use broadcast or multicast for the routing table maintenance instead of the explicit probing, their bandwidth cost is still much larger than that in O(log*N*)-hop overlays, especially when the system is very large or the churn rate is very high.

In sum, O(1)-hop overlays are more suitable for small or stable peer-to-peer systems because of their high routing efficiency, while O(log*N*)-hop overlays are more suitable for large and dynamic systems because of their low overhead. This situation makes the work of overlay selection very difficult for a peer-to-peer system designer because the eventual system size and churn rate is hard to predict in the early design phase. If an O(log*N*)-hop overlay is chosen but finally the system only attracts thousands of users and most of them usually stay a long time, such a design would be not reasonable enough because

This work is supported by National Science Foundation of China under contract 60433040.

O(1)-hop overlays are feasible and more efficient in this environment. Otherwise, if an O(1)-hop overlay is chosen but finally the system expands to a very large scale and most nodes only have a short lifetime ("*lifetime*" means the time period between a node joins the system and afterward it leaves), the routing table maintenance cost would rise to a very high level that prevents those weak nodes (e.g., modem-link nodes) from participating in the system. Furthermore, for a peer-to-peer system, redeploying its routing protocol simultaneously upon all the nodes is a hard work because nodes are numerous and autonomous. That is to say, first using an O(1)-hop overlay and afterward shifting to an O(log*N*)-hop one when the system expands to a large scale is also difficult in practice.

To address this overlay-selection problem, we design a *self-adaptivity* structured overlay, *Tourist*, which has the following properties:

1) In a small and stable system, Tourist is one-hop overlay.

2) When the system turns larger or more dynamic, Tourist adapts itself to be an overlay between one-hop and two-hop, which means that a) all the messages are routed within two hops, and b) the larger, the more dynamic the system is, the more messages are routed via two hops. That is to say, the average hop number increases along with the increasing of the system scale and the nodes' churn rate.

3) Even in a very large and dynamic system (1,000,000 nodes with their average lifetime about 1 hour), Tourist can guarantee that all the messages be routed within two hops.

4) In an extremely large and dramatically dynamic system, some messages would be routed via more than two hops. However, Tourist always provides an O(log*N*)-hop guarantee.

Generally speaking, Tourist is an overlay that dynamically adapts itself to the changing environment, always attempting to achieve as high routing efficiency as possible.

Tourist achieves the self-adaptivity by using a novel symmetric routing table structure which considers nodes' heterogeneity more carefully than previous protocols. Previous protocols treat all the nodes as uniform peers [25][27][21] or classify them into super ones and ordinary ones [31][33][29]. In contrast, Tourist nodes run at different *levels*. Nodes with larger bandwidth capacities run at higher levels, keep larger routing tables, and route their messages faster. A node can self-determine its level according to its own bandwidth capacity, and adjust it dynamically along with the change of the system environment. This heterogeneous design is more consistent with the measurement result of real peer-to-peer systems where nodes have significantly diverse bandwidth capacities because of their different link manners, and therefore can utilize all the nodes' allowable bandwidth more sufficiently than previous protocols.

Tourist nodes are allowed to adjust their levels dynamically to adapt themselves to the changing environment, which results in the self-adaptive routing efficiency from the view of the whole system. For example, when the churn rate rises, keeping original routing tables desires more bandwidth (Tourist uses multicast method for routing table maintenance). To guard against bandwidth overloading, nodes will lower their levels, i.e., shrink their routing tables. Since nodes' average routing table size decreases, message routing requires more hops. Totally speaking, it is the flexible adjustment of the individual nodes' routing tables that makes Tourist self-adaptive.

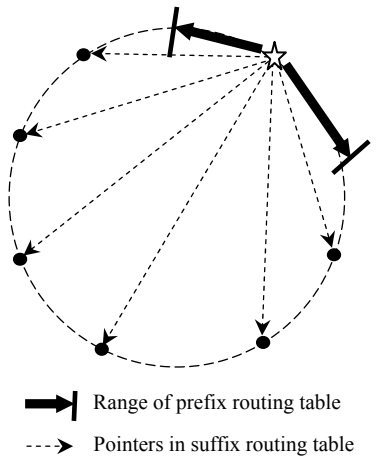

Figure1. Illustration of a Tourist node's prefix routing table and suffix routing table.

In the following, we will first give an overview of the Tourist protocol in section II, and then propose the method for routing table maintenance in section III. Section IV discusses how to realize locality-aware routing. Section V presents the simulation results. Section VI compares Tourist with some related works. Section VII makes the final conclusion.

II. PROTOCAL OVERVIEW

Like many existing structured overlays, Tourist uses numerical space, that is, every node has a 128-bit identifier *nodeId*, which is the hash result of its IP address plus port number. Each message has a destination *key*, also 128-bit long. Tourist uses XOR-based distance[1] instead of commonly used numerical distance, i.e., the distance of two 128-bit numbers, say *X* and *Y*, are defined as the value of their XOR result, namely $X \oplus Y$. Why XOR-based distance is chosen will be explained later on.

Given a message with key *k*, Tourist guarantees that it would be eventually routed to the node whose nodeId is nearest to *k*, which is noted as the message's *root node*. Determining which node's nodeId is the "nearest" one depends on the XOR-based distance metric.

The core structure of Tourist protocol is a symmetric level-based routing table construction. As many previous works [8][9][28] showed, given current PCs' hardware conditions and bandwidth capacities, two-hop routing is feasible within a large range of system environment. Accordant with these works, Tourist's core structure provides an overlay that is self-tuned between one-hop and two-hop and guarantees that all the messages can be routed within two hops, even in a very large and dynamic environment, e.g., a 1,000,000-node system where the nodes' average lifetime is only about 1 hour.

[1] XOR-based distance metric is first proposed in Kademlia [19].

Furthermore, Tourist also provides a backup structure that ensures O(log*N*)-hop routing when the system is extremely large or dramatically dynamic so that the basic core structure cannot guarantee two-hop routing for all the messages. Nevertheless, under the most circumstances where the core structure is enough, the backup structure contains no real pointers and produces no overhead.

In the following, we first introduce the core structure of Tourist, the symmetric level-based routing table construction, and then complement it with the backup structure that provides the O(log*N*)-hop guarantee.

A. Core Structure

Each node has a **self-determined** value *level*, which should be an integer (0, 1, 2…). A node running at level *l* should keep two symmetric routing tables: a *prefix routing table* and a *suffix routing table*. The former comprises the pointers to all the nodes whose nodeId shares an *l*-bit common prefix with the local nodeId. Similarly, the latter comprises the pointers to all the nodes whose nodeId shares an *l*-bit common suffix with the local nodeId. Figure 1 shows an illustration. It is easy to see that prefix routing table contains pointers to those nearby nodes in the nodeId space while suffix routing table contains pointers that scatter evenly in the nodeId space.

In Tourist, a pointer consists of the corresponding node's IP address, port number, nodeId, and level. To be convenient, we call the first *l* bits of a level-*l* node's nodeId the node's *prefix eigenstring*, and the last *l* bits its *suffix eigenstring*.

Message routing is very simple. When a node receives a message, it:

1) Checks whether itself is the root node of the message. If so, reports it to upper applications; otherwise, turns to 2).

2) If the root node is in prefix routing table, forwards the message to it; otherwise turns to 3).

3) Selects a node from suffix routing table whose prefix routing table must contain the root node and forwards the message to it.

We can see that under this simple routing algorithm, a message's routing has two possibilities: one-hop, directly via a pointer in prefix routing table or suffix routing table, and two-hop, the first one via a pointer in prefix routing table while the second one via a pointer in suffix routing table.

Step 3 in the above algorithm shows a key advantage of Tourist: a node *X* can directly judge whether another node *Y*'s prefix routing table contains a pointer to a message's root node as long as *X* holds a pointer to *Y*. Assuming the message key is *k*, if *k*'s first l_Y bits is the same with *Y*'s prefix eigenstring (l_Y is *Y*'s level), *Y*'s prefix routing table must contain the message's root node. It should be noted that this advantage is gained by using XOR-based distance metric. If using numerical distance, such inference would lack completeness. The substantial reason is in the next subsection.

Tourist aims to use this heterogeneous routing table structure to guarantee two-hop routing in a large range of system environment. This demands that most nodes should have a very large prefix/suffix routing table. For example, in a 1,000,000-node system, two-hop routing requires that each prefix/suffix routing table should contain at least 2170 pointers[1]. Obviously, using explicit probing for prefix/suffix routing table maintenance under this circumstance is impractical. Therefore, like many previous two-hop overlays [8][28][12], Tourist uses multicast method: when a node joins or leaves, multicasting the event to all the nodes who should know it.

However, the multicast design in Tourist is more sophisticated than that in previous overlays because Tourist is a heterogeneous protocol. The major difficulty here is to restrict the multicast range exactly: a node's join/leave event should be only multicast to those nodes whose prefix or suffix routing table contains a pointer to it. In the multicast mechanism, a node ought not to receive a message that is useless for its routing table maintenance. How Tourist achieves it will be introduced in section III. Briefly speaking, for prefix/suffix routing table maintenance, a Tourist node only receives a message when a node in its routing table or should be in its routing table leaves or joins. The bandwidth cost of the multicast method is much lower than explicit probing. Therefore, a Tourist node is easy to maintain thousands of pointers in its prefix/suffix routing table. Thus, two-hop routing is guaranteed even in a very large and dynamic system (seeing section V for experiment results).

Due to the significant heterogeneity of the peer-to-peer nodes [26], nodes with different capacities will run at different levels. This enables Tourist to fully utilize all the nodes' allowable bandwidth to achieve as high routing efficiency as possible. The basic reason that Tourist could be self-adaptive is that when the system environment changes, nodes will adjust their levels correspondingly. A typical Tourist client tool (running Tourist protocol and helping the node control its bandwidth cost) may work as follows.

For a node *X*, it sets an upper bandwidth threshold *W* (bps). The Tourist client tool will automatically adjust the node's level to keep the actual bandwidth cost always lying between $W/2$ and *W*. For example, at some time when the system comprises *N* nodes and their average lifetime is *L* seconds, *X* runs at level l_X and pays W_X bandwidth for routing table maintenance, $W/2 < W_X < W$. When the system expands, *X*'s routing table will expand proportionally because more nodes share an l_X-bit common prefix/suffix with *X*'s nodeId. Larger routing table desires more bandwidth for maintenance, i.e., W_X rises. When W_X exceeds *W*, the Tourist client tool will shift its level to $l_X + 1$ automatically. Then both *X*'s prefix routing table and its suffix routing table will shrink to only about half of their original size. W_X will also drop down to a value between $W/2$ and *W*. Similarly, if the system shrinks to the extent that W_X drops below $W/2$, the Tourist client tool will shift *X*'s level to $l_X - 1$, enlarging the prefix/suffix routing table correspondingly.

[1] The calculation is according to our previous study [12], which shows that in a *N*-node system, each prefix/suffix routing table containing more than $\sqrt{4.7 \cdot N}$ pointers can ensure two-hop routing with a very high possibility (more than 99.9%). [28] also showed similar results.

From this paradigm we can see that when the system expands gradually, a node's prefix/suffix routing table size does not change much, which is accomplished by lowering the level periodically, preventing the maintenance bandwidth from exceeding the threshold. This directly impacts on the routing efficiency: the average number of routing hops will increase gradually along with the system expansion because fewer and fewer messages can be routed via one hop. Note that the possibility that a message from node *X* can be routed via one hop is $(P_X + S_X)/N$, where P_X and S_X are the sizes of *X*'s prefix routing table and suffix routing table respectively.

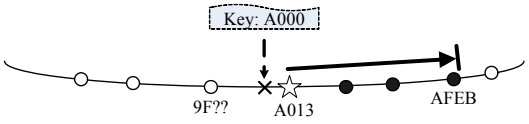

Figure 2. An instance of the routing puzzle. NodeIds are 4-bit long and noted in hexadecimal mode. If using numerical distance metric, the level-4 node A013 will be not able to determine whether itself is the root node of the message (with key A000).

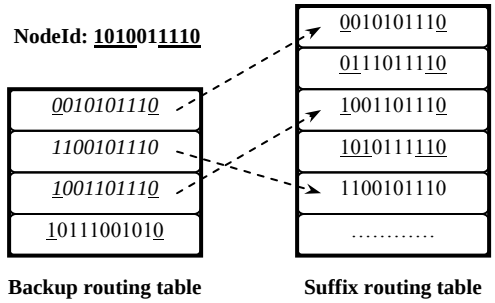

Figure 3. A possible backup routing table of a level-4 node 1010011110. NodeIds are 10-bit long and prefix/suffix eigenstrings are underlined. Dashed arrows show that most pointers in the backup routing table can be directly found in the suffix routing table.

Tourist is also self-adaptive to the change of nodes' churn rate. When nodes' average lifetime decreases, which means that more nodes join or leave the system in a given time period, then multicasts will be triggered more frequently. This means that a node will receive and send more messages per second to maintain its prefix/suffix routing table, i.e., the bandwidth cost rises. When the cost exceeds the preset bandwidth threshold, the Tourist client tool will automatically lower the node's level, shrink its routing table, and cut down the maintenance cost. Generally speaking, a node's routing table size is in inverse proportion to the system's churn rate, i.e., in direct proportion to nodes' average lifetime. This means that the lower the churn rate is, the less hops are required for message routing. An extreme example is that in a very stable system where nodes seldom join or leave, Tourist will always be a one-hop overlay (all the nodes run at level 0 and keep pointers to all the others), even if the system is composed of millions of nodes.

Above, we simply assume that a node adjusts its level because its bandwidth cost does not meet the requirement. In fact, a node can adjust its level for any reason. We put the focus on bandwidth because it is the scarcest resource (compared to memory, CPU and storage) for structured overlay maintenance in most cases.

```
rcv_msg(key=k, message=msg)   //receive a message.
    if begin_with(k, prefix_eigen) then
                //if the prefix eigenstring begins with k
    begin
(1)   p := getNearest(prefix_rt, k)
                //get the nearest pointer to k in the prefix routing
                //table, using XOR-based distance metric.
      if (p = self) then report(msg, upper_application)
                //self is the root node, reporting upward.
       else forward(p, k, msg)  //send the message to p.
    end else  //the root node does not in the prefix routing table.
    begin
(2)   cand := get_reachable(suffix_rt, k)
                //cand is the set of nodes that can route
                //the message to the root node via one hop.
      if (cand != null)  //two-hop routing from now on.
      begin
(3)     p := get_best(cand, k)
        forward(p, k, msg)
                //forward the message to the best one in cand.
                //the choice can be based on various preferences.
                //basically, a random one is chosen.
      end else //no suitable pointers in the suffix routing table.
      begin
(4)     p := get_nearest(backup_rt, k)
        forward(p, k, msg)
                //forward the message to the XOR-based
                //nearest pointer in the backup routing table.
      end
    end
```

Figure 4. Pseudocode of Tourist's routing protocol

B. *XOR-based Distance Metric*

We then explain why XOR-based distance metric is chosen rather than the commonly used numerical distance metric. The fundamental reason is that if using the numerical distance, there would be a "routing-puzzle" problem that needs special handling. Figure 2 shows an instance. Although prefix routing table has some similarity with the "leaf set" structure in previous protocols [25][27][15][11], they have a substantial difference: leaf set is symmetric, while prefix routing table is not. In figure 2, we assume that there is a level-4 node *X* whose nodeId is A013 and no other node whose nodeId starts with "A" and is less than A013. For statement convenience, nodeIds are assumed 16-bit long in this example and noted in hexadecimal mode. It is easy to see that *X* just stands at the left edge of its prefix routing table. Provided it receives a message with key A000, if using numerical distance, it would be puzzled by whether it is just the root node of the message, for it does not know the nodeId of its left neighbor in the nodeId space. When using XOR-based metric, this puzzle can be avoided because *X* can make sure that it is just the root node. The reason is that 1) its nodeId is nearer to A000 than that of any other node in its prefix routing table and 2) its nodeId is nearer to A000 than that of any other node out of its prefix routing table because a node out of its prefix routing table must have a nodeId that does not start with "A", which must generate a non-zero prefix at the first four bits when XORed with A000.

Similarly, if using numerical distance metric, a node *X* cannot judge whether another node *Y* holds a pointer to a mes-

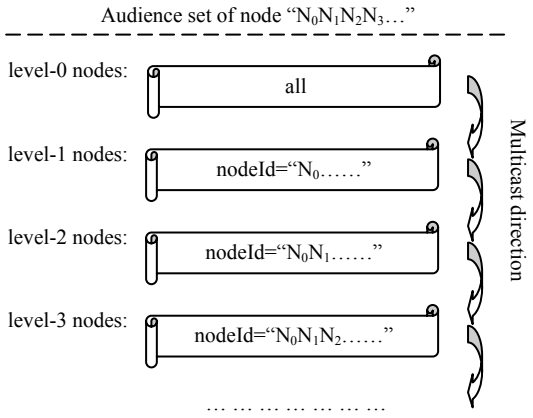

Figure 5. Composition of an audience set. The changing node's nodeId is "$N_0N_1N_2N_3$…"

sage's root node based on *Y*'s prefix eigenstring and the message's key because *Y* perhaps stands at the edge of its prefix routing table. This will disturb the inference that is used in step 3 of the simple routing protocol (seeing the previous subsection for the protocol).

C. Backup Routing Table

Although the symmetric structure of prefix/suffix routing tables and the simple routing protocol can guarantee two-hop routing in a large range of environment, in an extremely large (e.g., 10,000,000-node) or dramatically dynamic (e.g., average lifetime being several minutes) system, there will be some messages that cannot be routed to its root node eventually, i.e., the message sender cannot find a node from its suffix routing table whose prefix routing table contains a pointer to the root node. To realize the routing convergence for all the messages, each Tourist node also holds another data structure, *backup routing table*.

A level-*l* node's backup routing table has *l* items, the *i*th of which is a pointer to a node whose nodeId's first $i-1$ bits are the same with the local nodeId's, but the *i*th bit is different. Figure 3 shows a possible backup routing table of a level-4 node whose nodeId is 1010011110 (nodeIds are assumed 10-bit long in this example). It is important to note that in most cases, the pointers that satisfy the requirement of the backup routing table items can be directly found in the suffix routing table and then only virtual links are recorded, such as the first three pointers in figure 3. Only when the suffix routing table does not contain any satisfactory pointer for an item, a physical pointer is recorded, such as the fourth pointer in figure 3. Experiments show that only in a very large and dynamic system, backup routing table will contain physical pointers and make contributions to message routing (seeing section V).

It is easy to see that backup routing table resembles Pastry's routing table, which can provide an O(log*N*)-hop guarantee for message routing.

Augmented with the backup routing table, Tourist's routing protocol turns logically complete. Figure 4 shows the pseudocodes. Its only difference with the aforementioned simple protocol is to use a pointer in the backup routing table when no suitable pointer can be found in the suffix routing table (seeing line (4)). It is easy to see that Tourist's routing protocol is a greedy one: the XOR-based distance between the router's nodeId and the message key is shortened hop by hop. Therefore, the routing convergence must be ensured for all the messages.

```
rcv_event(nodeId=X, event_type=et, step=s)
            //receive a changing event of node M at step s.
(1)   adjustPrefixRT(list, et, X)
            //adjust the prefix routing table according to the event.
(2)   Rs = getAudience(list, X)
            //get X's audience set from the prefix routing table.
      for i := s + 1 to 64 do
(3)     Rn := getPrefix (Rs, localID, i − 1)
            //get all the pointers to those nodes whose nodeIds are the
            //same with X at the first i−1 bits, but different at the ith.
(4)     if Rn = null then
          continue
        fi
(5)     P := getHighestLevel(Rn)
            //get the pointer with the highest level from Rn.
            //If more than one pointers are at the same level,
            //randomly pick one from them.
(6)     send_event(P, X, et, i)
            //send the event to P, tagged step i.
      od
```

Figure 6. Pseudocde of the multicast method.

In line (3) of the protocol, a node can choose a pointer from a number of candidates whose prefix routing tables all contain a pointer to the root node. Though choosing a random one is a simple method, trying to choose a close-by[1] one in the real network may be better, for it can reduce the delay of this hop. We will exploit this way in section IV.

III. Routing Table Maintenance

As discussed above, the main issue in routing maintenance is the multicast method. Considering the symmetry of the prefix routing table and the suffix routing table, we will only put emphases on the multicast method for prefix routing table maintenance in the following statement. It can be also used for suffix routing table maintenance in a symmetric manner.

A. Tree-based Multicast

For a node *X*, all the nodes whose prefix routing tables contain a point to *X* is called *X*'s (*prefix*) *audience set*. When a node joins, leaves, or adjusts it level, the event should be multicast to all the nodes in its audience set. Before proposing the multicast method, we first analyze the composition of an audience set. Figure 5 shows an instance, from which we can see that node *X*'s audience set is composed of all the nodes whose prefix eigenstring is a prefix of *X*'s nodeId. It is important to note that within an audience set, an arbitrary node's prefix routing table must contain the pointers to all the others that are at the same level or lower levels. Therefore, once a node at the highest level in *X*'s audience set gets *X*'s changing event, it has enough information to multicast it all around the audience set.

[1] In this paper, we use the word "close-by" to denote a short distance in the real network, while use the word "near" to denote a short distance in the nodeId space.

In the instance of figure 5, the highest level is level 0, but it is not always the case. When the system is very large or very dynamic, there would be no nodes can afford the bandwidth cost of running at level 0 (seeing experiments in section V). To simplify the statement, we make the following definition first:

1) If node *X* runs at a higher level than node *Y* and *X*'s prefix eigenstring is a prefix of *Y*'s, then *X* is said to be a (*prefix*) *super node* of *Y*. Note here that *X*'s prefix routing table must cover *Y*'s prefix routing table.

2) If node *T* runs at the highest level among all the *X*'s super nodes, then *T* is said to be a (*prefix*) *top node* of *X*.

In a system where level-0 nodes exist, only level-0 nodes are top nodes. In a system where there is no level-0 node, nodes are split into several parts. Prefix routing tables of two nodes in different parts must have no intersections with each other. Each part has its own top nodes, i.e., the highest-level ones in this part. To maintain the prefix routing table, each node should keep a (*prefix*) *top-node list*, which contains *t* pointers to the top nodes in its part. Commonly we set $t = 8$.

When a node *X* changes its state (joins, leaves, or adjusts its level), the event should be first reported to a top node via a message called *event-report*. Then the top node initiates a tree-based multicast that distributes the event to all the nodes in *X*'s audience set. Figure 6 shows the pseudocode. The key idea of the multicast method is as follows. When a top node starts to multicast an event, it first sends the event to a node whose nodeId's first bit is different with the local nodeId. Thus there will be two nodes that have received the event, with the first bit of their nodeIds different. After that, each of these two nodes sends the event to another node whose nodeId has the same first bit but different second bit with the local nodeId. After that, all the four nodes that have received the event have different first two bits with each other in their nodeIds. In general, at step *s*, every node that has received the event should send the event to another node whose nodeId has the same first *s* bits and a different (*s*+1)th bit with the local nodeId (seeing line (3) of figure 6). This process continues until no appropriate node can be found at some step (line (4)).

It must be noted that at each step the local node always chooses a target node with the highest level from all possible nodes (line (5)). This ensures the proper multicast direction that is from higher-level nodes to lower-level nodes. We proved that this multicast method is complete (all the nodes in the audience set can receive the event) and non-redundant (each node only receives the event once) elsewhere [13]. Furthermore, it also has the following properties:

1. Different nodes have different out-degrees. Higher-level nodes have more out-degrees than lower-level ones.

2. An event can reach all the nodes in the audience set through about $\log_2 A$ steps, where *A* is the size of the audience set.

3. The multicast tree is neither unique nor pre-determined. Every node dynamically chooses the next target in the multicast tree at runtime (seeing line (5)).

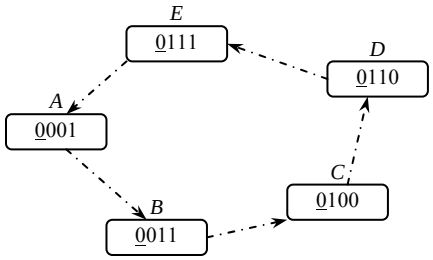

Figure 7. Illustration of the failure detection mechanism.

To guard against stale pointers in the prefix routing table, acknowledgements are required for all the multicast messages. When a message gets no response after three continuous attempts, the corresponding pointer will be removed from the prefix routing table and the message will be redirected to a new target node (i.e., turn back to line (3)).

B. Failure Detection

When a node joins or adjusts its level, it reports the event to a top node, randomly chosen from its top-node list, by itself. Before a node leaves the system, it should also send a report to a top node. However, peer-to-peer nodes may leave the system quietly, without notification to others. Therefore, a failure detection mechanism is desired. Figure 7 shows an illustration of the failure detection for prefix routing tables. Note that all the nodes with the same prefix eigenstring are fully connected through their prefix routing tables (i.e., each keeps pointers to all the others). Thus, all the nodes with a given prefix eigenstring can be seen as a virtual circle based on their nodeIds. It is demanded that each node in this circle probe its right neighbor periodically ("right" means the direction from small to large). In the example of figure 7, there are five nodes with prefix eigenstring "0". Then each of them probes the node whose nodeId is just larger than it, while the one with the largest nodeId (namely *E*) probes that with the smallest nodeId (*A*).

Once a node, say *A*, detects the failure of its right neighbor, namely *B*, it immediately reports the event to one of its top nodes (*A*'s top node must also be *B*'s top node) and redirects its probe to the next right neighbor, *C*.

It should be noted that this failure detection mechanism is resilient to concurrent failures. For example, when node *B* and *C* concurrently leave the system, *A* will first detect *B*'s failure and remove *B* from its prefix routing table. After that it will redirect the probe to *C*, immediately detect *C*'s failure and redirect the probe to *D*.

A special scenario that should be noticed is that when there is only one node with a certain prefix eigenstring in a system, it will not be probed by any other node. When it leaves, the event will not be reported to a top node. Then the pointers to it in other nodes' prefix routing table will become stale pointers. Removing such stale pointers is accomplished by the refreshing mechanism presented in the next subsection.

C. Refreshing

Because of the Internet asynchrony, no application-level multicast can be absolutely reliable. Therefore, there must be some errors in prefix routing tables, which fall into two types:

absent pointers and stale pointers. Both of them are only of a very small fraction and do no harm the system substantially. An absent pointer would be automatically revised when the corresponding node leaves the system, while a stale pointer would be removed when being used for multicast process and getting no response. However, these errors would accumulate before being revised, from the view of whole system.

To guard against the accumulation, Tourist employs a refreshing mechanism. Every node measures the lifetime of all the nodes in its prefix routing table, and calculates the average lifetime of the nodes at each level, noted LT_i, where *i* denotes the level value. A level-*l* node multicasts its state around its audience set every $2 \cdot LT_l$ (by reporting to a top node). A level-*m* pointer that has not been refreshed for a period of $3 \cdot LT_m$ will be directly removed from the prefix routing table. This mechanism can limit the accumulation of both the absent pointers and the stale pointers, producing an upper bound for the error rate of prefix routing tables. In practice, most nodes never perform such refreshing multicast because their lifetimes are much shorter than twice the average lifetime.

This refreshing mechanism is also used in Calot [28], in which it is called "re-announcement".

D. Top-node List Maintenance

Top-node lists are maintained in a lazy manner. When a node *X* sends an event-report to a top node, the top node returns an acknowledgement, piggybacking $t-1$ pointers to other top nodes, which helps *X* refresh its top-node list. If the event-report does not get an acknowledgement, *X* redirects it to another top node within its top-node list. If all the top nodes in its top-node list are unavailable, it asks another node in its prefix routing table for his top-node list as a substitution.

E. Suffix Routing Table Maintenance

Nodes' prefix routing tables and suffix routing tables are maintained independently. Above we showed the maintenance of prefix routing tables. Suffix routing table maintenance is in a completely symmetric way.

Each node keeps a (*suffix*) *top-node list*. A node's changing event is first reported to a (*suffix*) *top node* and then multicast around the node's (*suffix*) *audience set* through a suffix-based multicast method. Failure detection is based on the probes in virtual circles among the nodes with the same suffix eigenstrings. Refreshing mechanism is also employed.

In general, a node's changing event will trigger two multicasts, one in its prefix audience set and the other in its suffix audience set.

F. Joining and Level-adjustment

A new node *X* contacts a bootstrap node *B* that is already in the system for joining. *B* first helps *X* determine its initial level and then finds a prefix top node T_P and a suffix top node T_S for *X*. *X* determines its initial level as $l_X = \left\lceil l_B + \log_2 \frac{W_B}{W_X} \right\rceil$, where l_B is *B*'s level, W_B is *B*'s recent bandwidth cost that is dynamically measured, and W_X is *X*'s bandwidth threshold.

B finds T_P by the following way: if *B*'s prefix top nodes are also *X*'s top nodes, then *B* choose a random node from its prefix top-node list as T_P; otherwise, *B* forwards the top-node finding task to a node *C* in its suffix routing table whose prefix top nodes are also *X*'s top nodes. This can be judged by a) *C*'s prefix eigenstring is a prefix of *X*'s prefix eigenstring, b) *X*'s prefix eigenstring is a prefix of *X*'s prefix eigenstring, or c) *C* and *X* have the same prefix eigenstring. After that, *C* offers *X* one of its top nodes as T_P. *B* finds T_S for *X* in a similar way.

After finding T_P and T_S, *X* downloads its prefix/suffix routing table and top-node list from them. T_P and T_S do not always transmit the prefix/suffix routing table by themselves. They can ask some other *X*'s super nodes to transmit it in parallel, different nodes in charge of different fractions. After routing table transmission, T_P and T_S multicast *X*'s join event around *X*'s prefix audience set and suffix audience set, respectively.

A joining node can also first set a low level so as to start working in a relatively short time, and then ask super nodes for a large prefix/suffix routing table. After completing the background downloading, it raises it level and reports the level-shift event to a top node. We call this process *warm-up*.

A node can adjust its level at runtime. When it raises its level, it should first download those required pointers from super nodes and then report a level-shift event to a top node. When it lowers its level, it could remove those useless pointers from its routing table directly and then send an even-report to a top node.

G. Backup Routing Table Maintenance

When a node joins the system, after downloading its prefix/suffix routing table, it selects appropriate pointers in the suffix routing table as virtual pointers to fill in the items of the backup routing table. If there are some items that cannot be satisfied by any pointer in the suffix routing table, it then asks a super node (or a top node, if no super node found in the prefix/suffix routing table) for proper physical pointers. The maintenance of those physical pointers is by periodically probing, commonly once per 30 seconds. When a stale pointer is detected, a fresh one will immediately take its place, which is also obtained from a super node or a top node. Since only in a very large and dynamic system, the backup routing table will contain a few physical pointers, the probing cost is very low.

H. Summary

Totally speaking, to maintain its routing table, a Tourist node pays bandwidth for three objectives: 1) participating in the event-distribution multicast for prefix/suffix routing table maintenance, 2) probing another node for failure detection, and 3) probing physical pointers in the backup routing table. The first one occupies the most of the bandwidth cost. It ranges from several messages per second (for weak nodes) to hundreds of messages per second (for powerful nodes). The second one is a fixed value, 0.2 messages per second if messages are sent once every 5 second. In most cases the third one is zero. In a system that is very large and dynamic, weak nodes' backup

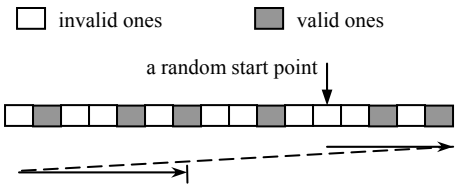

Figure 8. An illustration of the next-hop selection from the suffix routing table. The search starts from a random point and collects N_{cand} (5 in this example) valid candidates. The GNP-closest one among them will be finally chosen as the next hop.

routing tables will contain several physical pointers. The probing cost will be no more than 0.5 messages per second (assuming 10 physical pointers). From this analysis we can see that the latter two probing cost is negligibly slight compared to the multicast cost used for prefix/suffix routing table maintenance. This confirms the validity of Tourist's basic idea, letting each node adjust the size of its prefix/suffix routing table to control its bandwidth cost, which results in the self-adaptivity of the whole system.

IV. Locality-aware Routing

Recall that in Tourist's routing protocol (figure 4), when choosing the next hop from the suffix routing table, there may be many candidates that can route the message to the root node via one hop (seeing line (2) and (3)). Choosing a close-by one from them will greatly reduce the network latency of this hop, which makes significant sense for the whole message routing that only requires two hops totally.

To accomplish this locality-aware routing, a node needs to know the network distance between itself and the nodes in its suffix routing table. Due to the large size of the suffix routing table, explicit probing is unrealistic, so we prefer the distance-predict technique. From many proposed methods in the distance-prediction area [20][6][4], we choose GNP [20] because of its simplicity.

Every node measures its network distances to 16 well-known nodes and attaches the 16-dimension coordinates into its pointers. When there are many candidates for the next hop, a node will choose the one whose coordinates are the closest to its own in the 16-dimension Euclidean space.

This locality-aware method improves the routing efficiency greatly, but it costs too much CPU time when the candidates are too many. Experiments (in section V) show that in a common 1,000,000-node system, there will be more than 5,000 candidates on average. Therefore, a Tourist node will compromise a tradeoff between the routing efficiency and the CPU time in practice, as illustrated in figure 8.

When X needs to choose a node in its suffix routing table that can route the message to the root node via one hop, it searches the suffix routing table from a random start point, seeing figure 8. The search continues until it has collected N_{cand} valid candidates (whose suffix routing tables contain the root node) or all the items in the suffix routing table have been checked. After that, it chooses the one with the closest GNP coordinates from the collected candidates as the next hop. Tourist allows nodes to determine their N_{cand} values independently, according to their different CPU capacities. Obviously, larger N_{cand} provides higher routing efficiency but costs more CPU time.

V. Experiment Results

The basic goal of our experiment is to simulate Tourist protocol in a system with 1,000,000 nodes whose lifetime distribution is consistent with the measurement result of real peer-to-peer systems [26]. Our experiment environment is a 16-server cluster. To make the parallel simulation possible, we build our experiments on top of ONSP [30], a general platform for overlay protocol simulation. ONSP is based on parallel discrete events and uses MPI for server communication. By using ONSP, our experiments obtain a time scale of 1/26: simulating 1-hour Tourist running requires one day or more.

Tourist nodes keep large routing tables. Therefore, it is impossible to store all the nodes' routing tables in memory. Noting the fact that Tourist nodes with the same prefix/suffix eigenstring should contain the same pointers in their prefix/suffix routing tables, we store a "correct" routing table for each kind of prefix/suffix eigenstring (comprising pointers to all the live nodes whole nodeIds start/end with the eigenstring) and only store the error items of the actual prefix/suffix routing table for each node. This optimization enables the experiments to be executed wholly in memory, avoiding the data transmission between memory and disk.

A transit-stub model of Internet topology is embedded in ONSP, which is generated by the tool of GT-ITM [32]. We set the parameters as the follows. There are 120 transit domains, each containing 4 transit nodes. Every transit node has 5 stub domains, each containing 4 stub nodes. Every Tourist node is connected to a random stub node. The intra-stub latency is 1ms. The stub-to-transit latency is a random value between 10ms to 30ms. The transit-to-transit latency is a random value between 50ms to 150ms.

In the experiment, we first launch 1,000,000 nodes whose lifetime distribution is consistent with the measurement result of Gnutella, i.e., figure 6 of [26]. Then we begin the event-driven process and let new nodes join the system continuously in a Poisson procedure. The expectation of node-joining rate is equal to the average node-leaving rate, that is, 1,000,000 nodes per 2.3 hours (2.3 hours is the average lifetime in figure 6 of [26]). By this way, the system scale is kept stable at 1,000,000-node in the whole simulation process.

We assume that each node sets an upper threshold for its input bandwidth, which is 1% of its total available input bandwidth, but cannot be less than 500bps. We believe such a slight bandwidth cost will hardly impact other applications negatively. The distribution of the nodes' available input bandwidth is also consistent with Gnutella's measurement result, i.e., figure 3 of [26].

Arguments in Tourist protocol are set as follows. In failure detection mechanism, every node probes it right neighbor every 5 seconds. If not getting an acknowledgement for three continuous times, it reports a node-leaving event to a top node. Every node probes the physical pointers in its backup routing table every 30 seconds. If not getting an acknowledgement

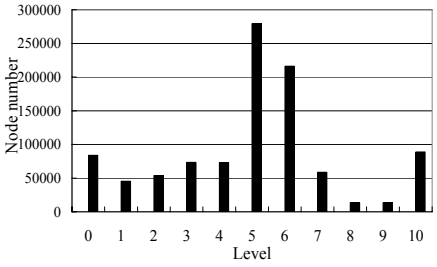

Figure 9. Node distribution.

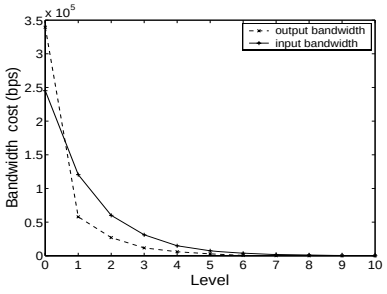

Figure 10. Bandwidth cost.

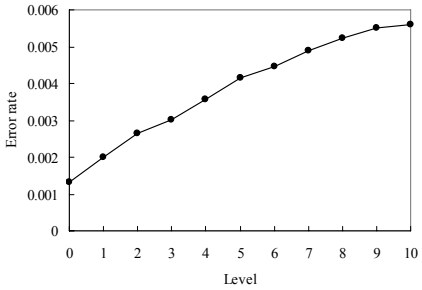

Figure 11. Error rate of prefix/suffix routing tables.

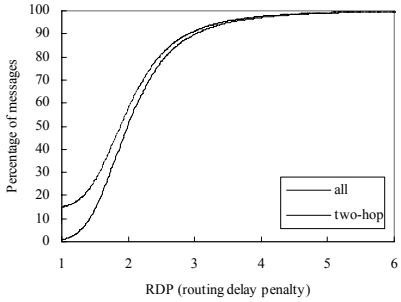

Figure 12. CDF of routing delay.

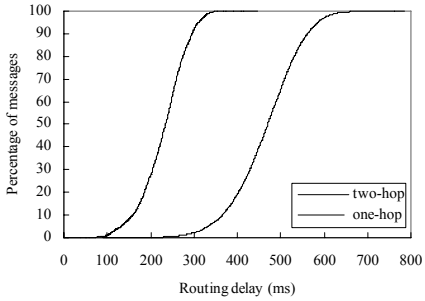

Figure 13. CDF of RDP

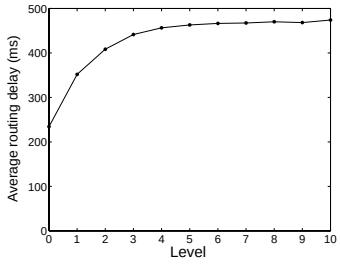

Figure 14. Routing delay of different-level nodes

from some node, it then continues to probe it twice, once per 5 seconds. If still not getting an acknowledgement, it asks a super node for another pointer as a substitution. In the routing process, when a node needs to choose the next hop from its suffix routing table, if multiple candidates exist, it chooses a random one from them. This is accomplished by setting $N_{cand} = 1$ in the method shown in figure 8.

The size of an event-distribution message (used in multicast) is set 1000-bit long, which includes the event type, the changing node's nodeId, IP address, port number, level and GNP coordinates, and about 400 bits reserved for upper applications. During the tree-based multicast process, every median node delays the message for 0.5 second that is spent on receiving the message, calculating the next hop and sending the message.

All our experiments are performed on a 16-server cluster that is connected by 2Gbps Myrinet. Each server has four 700MHz Xeon CPUs and 1GB memories, running an operating system of Linux Redhat 7.3.

A. The Common Case

Figure 9 shows the distribution of the nodes in the common environment. There are totally 11 levels. The majority of the nodes lie in two parts, level 5 to 6 and level 10, which are the cable-linked ones and the modem-linked ones. They account for 44% and 23% of the total nodes respectively (seeing figure 4 of [26]). This distribution shows the great heterogeneity of a real peer-to-peer system, confirming the basic design principle of Tourist.

Figure 10 depicts the bandwidth cost of the nodes at each level. To keep the whole state of the system, level-0 nodes spend about 350kbps input bandwidth (note that their available input bandwidth is more than 35Mbps). This cost decreases by half once the level decreases by one. A level-10 node only spends 340bps input bandwidth, which is under the preset threshold of 500bps. Considering the message size of 1000 bits, this means that these weakest nodes only receive 0.5 messages per second.

Most nodes' output bandwidth is lower than their input bandwidth, except the level-0 ones. This is because of the unbalanced maintenance multicast (seeing section III) in which higher-level nodes send more messages than lower-level ones. Nevertheless, the output bandwidth cost of level-0 nodes is no more than 2% of their total available output bandwidth yet, which is acceptable in most cases.

Although the prefix/suffix routing tables are large, they have very few errors. As figure 11 shows, higher-level nodes have fewer error items in their routing tables. This is because the multicast direction that is from higher-level nodes to lower-level ones. We can see that even the error rate of the weakest nodes' routing tables is less than 0.6%. When a node leaves, the event will be detected after 15 seconds (three probes with no acknowledgement) and then reported to a top node. The multicast from the top node to each node in the audience set takes about $\log_2 1{,}000{,}000 \approx 20$ steps. Assuming each step requires 250ms, all the nodes in the audience will receive the event within $15 + (1+20)\cdot(0.5+0.25) = 30.75$ seconds. That is to say, a pointer will remain stale for no longer than 30.75 seconds. Compared to the average lifetime (135 minutes), the error rate will be less than $30.75/(135\times 60) \approx 0.0037$, which is accordant with the experiment result. Other O(1)-hop overlays also showed similar results [8][28].

Since the prefix/suffix routing tables contain stale pointers, using these pointers for message routing will render the message to be lost. To guard against it, Tourist nodes ask for acknowledgement in each routing hop. When a node finds a stale pointer in its suffix routing table (forwarding a message to it

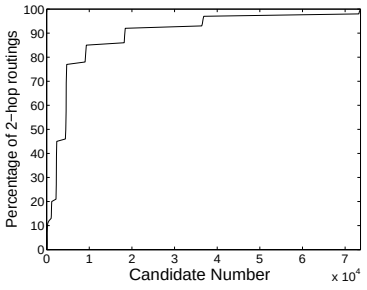

Figure 15. CDF of the number of candidates when choosing the next hop from the suffix routing table

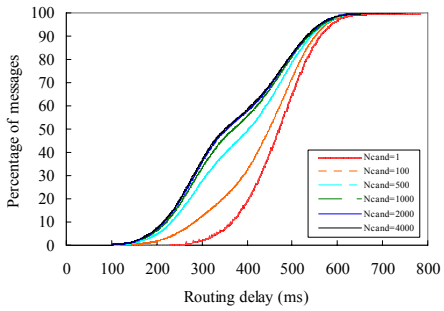

Figure 16. CDF of locality-aware routing delay

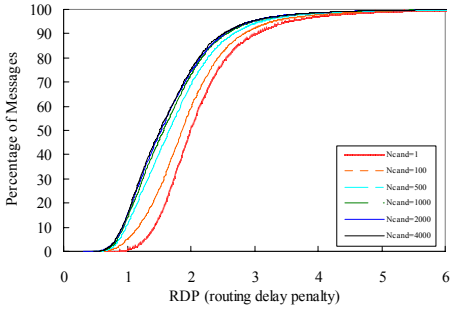

Figure 17. CDF of locality-aware RDP

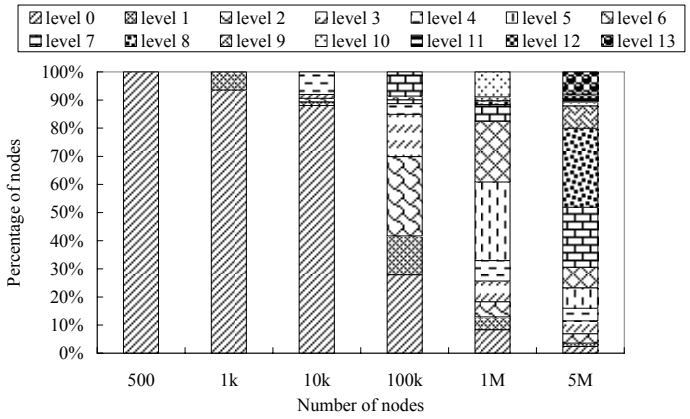

Figure 18. Level distribution vs. system scale.

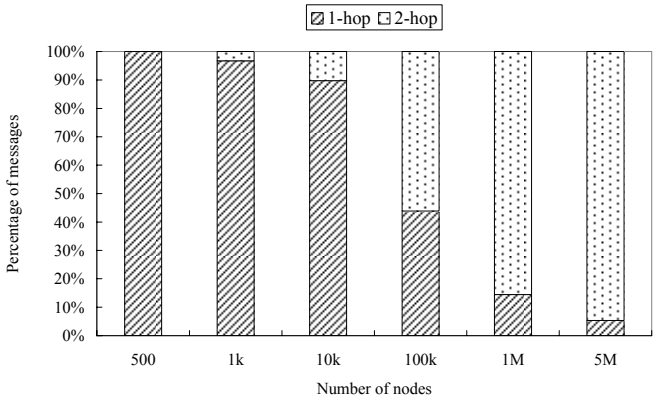

Figure 19. Hop distribution vs. system scale.

and getting no acknowledgement), it removes the stale pointer and redirects the message to another candidate (whose prefix routing table contains the root node). When a node finds a stale pointer in its prefix routing table, which must point to the former root node of this message, it removes it and sends the message to the new root node.

To measure Tourist's routing efficiency, we launched 200,000 message routings from random initiator with random keys. All these messages were finished within two hops. The CDF (cumulative distribution function) of their routing delays is depicted in figure 12. We distinguish one-hop messages and two-hop messages for clarity. Most one-hop messages traverse two transit domains and cost 200-300ms. Two-hop messages take longer time, most of which require 400-600ms, almost twice that of one-hop messages. The routing delay penalty (RDP) is depicted in figure 13. One of the two lines is for all the messages and the other only for two-hop ones. Considering that RDPs of one-hop messages must be 1, the start pointer of the all-hop line on the y-axis indicates that 15% messages are routed via one hop. We can see that about 50% of the two-hop messages' RDPs are larger than 2.0, which indicates that there is sufficient room for locality-aware optimization.

In Tourist, different nodes run at different levels, maintain different sizes of routing tables, and therefore route their messages with different delays. Figure 14 shows this effect. We can see that since level-0 nodes keep pointers to all the other nodes, their messages can always be routed via one hop, the delay of which is the same with the IP layer. For level-1 nodes, only half of their messages can be routed via one hop, while the other half require two hops. Since the two-hop delay is approximately twice as long as the that of one-hop, level-1 nodes have an average routing delay of about 360ms, which is 50% longer than that of level-0 nodes. The lower level a node runs at, the longer delay its messages require. For those weakest nodes, almost all their messages can be only routed via two hops (seeing the high tail of the line).

B. Locality-aware Routing

When a node chooses a pointer from its suffix routing table for the next hop, there may be many candidates. Figure 15 depicts the CDF of the candidate count. In 80% cases, there are more than 2000 candidates. Choosing a close-by one from them will reduce the delay of this hop greatly, which also makes significant sense for the whole routing efficiency, for most messages only involve two hops.

As discussed in section IV, we use GNP coordinates as guidance for close-by node selection. In this experiment, 16 nodes are selected as the landmark nodes, which are far from one another. Each node measures its distances to these landmark nodes, getting a 16-dimenstion coordinates and attach them into the pointers. When choosing the next hop from its suffix routing table, a node will choose the GNP-closest one from N_{cand} candidates using the method proposed in section IV. Figure 16 and figure 17 show the improvement of the routing

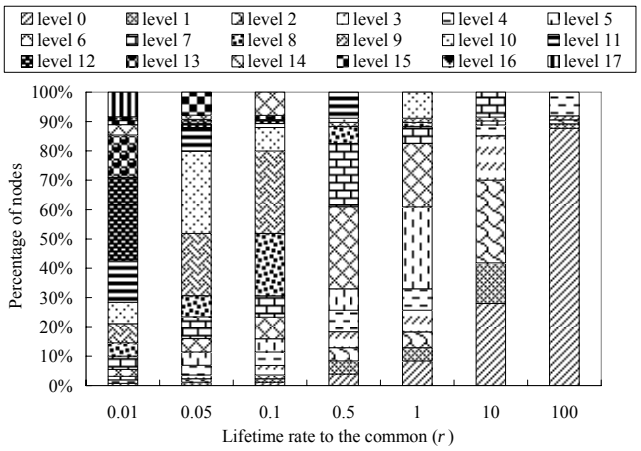

Figure 20. Level distribution vs. lifetime.

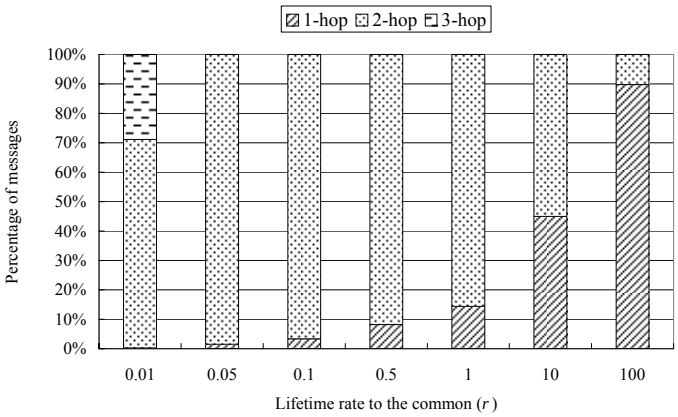

Figure 21. Hop distribution vs. lifetime.

efficiency by this method. Only two-hop messages are considered in this experiment. From the result we can see that:

- A dramatic improvement can be obtained even with a small value of N_{cand}, such as 100.
- The larger N_{cand} is, the shorter delay a message requires.
- Setting N_{cand} as 2000 can enable half of the two-hop messages to be routed within 320ms, with RDPs less than 1.4. But even larger N_{cand} only can make little further improvement.

Considering that larger N_{cand} desires more CPU time, we recommend a value of 1000 or 2000 for N_{cand}. In practice, nodes are allowed to set N_{cand} by themselves.

C. *Scalability*

Tourist's routing efficiency is tightly related to the system scale. As discussed in section II, when the system expands, nodes lower their levels gradually to keep the bandwidth cost under the threshold. Consequentially, more messages require two hops. Figure 18 and figure 19 show this effect.

In figure 18, we use different patterns to show the percentages of nodes at different levels. In a 500-node system, all the nodes run at level 0, which means that Tourist turns to be a one-hop overlay in which all the messages can be sent to the root node directly through the IP layer. When the system turns larger, nodes lower their levels, results in more levels and fewer nodes at high levels. For example, in a 100,000-node system, there are totally eight levels and only 30% nodes have enough bandwidth capacities to run at level 0.

Figure 19 shows the variation of the routing-hop distribution along with the system expansion. In a 500-node system, all the messages are routed via one hop. When the system expands, more and more messages require two hops, indicating that the average hop number increases gradually. When the system reaches a scale of 5,000,000 nodes, in which there are 14 levels totally (seeing figure 18), all the messages can still be routed within two hops. However, the proportion of the one-hop ones drops to only 5.34%.

D. *Adaptivity*

When the system's churn rate changes, nodes also adjust their levels to control their bandwidth cost, which makes Tourist self-adaptive ultimately.

Figure 20 shows the variation of the level distribution along with the change of nodes' lifetime. Also different patterns denote percentages of nodes at different levels. Accordingly, Figure 21 shows the variation of the routing-hop distribution. In this experiment, we let every node's lifetime be r times of that in the common case (the experiment above in subsection A). It is easy to see that higher churn rate (namely shorter average lifetime) produces more levels. This is because when nodes join and leave the system more frequently, maintaining the same routing table desires more bandwidth. To guard against bandwidth overloading, nodes lower their levels automatically.

In a system where the nodes' average lifetime is 9.4 days (r=100), 90% nodes run at level 0, keeping pointers to all the other ones, even though the current system size is 1,000,000-node. Under this circumstance, 90% messages can be routed via one hop. This result confirms our argument that one-hop overlay is always feasible as long as the system is stable enough, even if the system size is very large.

At the other end, in a system where the nodes' average lifetime is as short as 1.35 minutes (r=0.01), the weakest nodes can only run at level 17. Another important matter that should be noticed is the absence of level-0 nodes. In fact, the most powerful nodes only run at level 3. From figure 21 we can see that in this dramatically dynamic system, two-hop routing cannot be guaranteed any more. There are about 30% messages that are routed via three hops. This means that the first hop must be via a pointer in backup routing table.

VI. RELATED WORK

Tourist is a heterogeneous and self-adaptive structured overlay. It borrows some ideas from other previous protocols but have significant difference with them.

The idea that utilizing nodes' heterogeneity to achieve higher routing efficiency has been widely used [31][33][29]. These protocols classify nodes into powerful ones and weak ones and construct a structured overlay only among the powerful ones. A weak node only connects to a close-by powerful one. Tourist considers heterogeneity more carefully, dividing nodes into different levels. By this way, Tourist can utilize nodes' allowable bandwidth more sufficiently.

In the area of O(1)-hop overlays, Gupta et al. proposed a one-hop overlay and a two-hop overlay [8]. In their schemes, nodes' state-changing events are distributed through a unique pre-determined tree. This requires some powerful nodes that always work as slice leaders. Though Tourist also uses a heterogeneous structure in which nodes run at different levels, it does put the constraint of being powerful ones on any node. A node can run at any level it prefers. In fact, Tourist can work without powerful nodes, i.e., when all the nodes are at the same level.

Kelips [9] is another well-known two-hop structured overlay. In Kelips each node maintains a routing table with $O(\sqrt{N})$ pointers and gossips to one another to maintain the pointers. The gossip method is not efficient enough because a node may receive the same event multiple times. Additionally, it takes a long time for an event to reach all the nodes that should know it. By contrast, the tree-based multicast in Tourist distributes an event more efficiently and has no redundancy. Furthermore, Kelips has no adaptivity. It always provides two-hop routing, no matter how greatly the system environment is changing.

There are also some works that aim to make O(log*N*)-hop overlays more adaptive. Castro et al. [3] present an optimized version of Pastry (MSPastry) that self-tunes its probing frequency to adapt itself to the churn rate. Bamboo [22] uses a fixed-period recovery method instead of traditional active recovery method to save the bandwidth cost under high churn. Tourist also pays extensive attention to the bandwidth control, especially when the churn rate is high. However, it uses a wholly different way with the above two protocols. Tourist nodes will shrink their routing tables to defend against the bandwidth overloading.

SmartBoa [11] is our preliminary design of Tourist. Essentially, SmartBoa's structure is a suffix routing table plus a leaf set. By using the prefix routing table instead of the leaf set, Tourist achieves higher routing efficiency. This is because at the same bandwidth cost, a node can keep more pointers when using a prefix routing table, which is maintained by multicast, than using a leaf set, which is maintained by explicit probing.

Twins [12] can be seen as a homogeneous version of Tourist, in which nodes' routing table format is the same with Tourist, but all the nodes run at the same level. Z-Ring [17] extends Twins with an improvement that handles the scenario of the whole-system update: when the system should be updated from level 7 to level 6, there would be a transient period when level-6 nodes and level-7 nodes coexist. Z-Ring proposes a gossip-based method for routing table maintenance in this transient period. However, it does not treat the system as an essentially heterogeneous one yet. It still assumes that nodes run at the same level in the most of time (with almost the same cost) and does not propose a maintenance protocol for a system with nodes at totally different levels.

Tang et al. proposed 2h-Calot [28] as a self-adaptive two-hop overlay. In 2h-Calot, nodes expand or shrink their routing tables to keep an $O(\sqrt{N})$ size along with the change of the system scale. The most significant difference between 2h-Calot and Tourist is their different design goals. 2h-Calot aims to minimize the total bandwidth cost of the whole system in a given environment while keeping 2-hop routing. By contrast, Tourist aims to minimize the routing hops when each node's bandwidth cost is prelimited (by an upper bound). Another difference is that 2h-Calot nodes are uniform whose routing tables are almost of the same size, while Tourist nodes' routing tables are of extensively different sizes. In addition, 2h-Calot can only adapt to a system with a large scale (million-node) and moderate churn rate (with average lifetime of 1 or 2 hours). In an even larger or more dynamic system, it cannot turns to an O(log*N*)-hop overlay, like that in Tourist.

The concurrent work Accordion [16] shares the same goal with Tourist, but uses a completely different method. An Accordion node's routing table contains the pointers that scatter in the nodeId space adhering to the small-world distribution. Powerful nodes (with higher bandwidth thresholds) keep larger routing tables (i.e., their pointers scatter more densely) than weak nodes. A node-leave event will not be distributed to others. Instead, every node uses the lifetime model to predict whether a pointer in its routing table has been stale. Those pointers that have a large possibility of being stale will be removed from the routing table directly. This method reduces the maintenance cost greatly. However, it heavily depends on the accuracy of the lifetime model for prediction. Since different peer-to-peer systems' lifetime distributions are significantly different, obtaining an appropriate lifetime model for a given system is not rather simple. Because Accordion uses the small-world model to construct routing tables, it essentially is a self-adaptive O(log*N*)-hop overlay. From the view of the whole system, the argument that it adjusts along with the environment changes is the logarithm base *B*, which impacts the average hop number ($\log_B N$) straightly. In contrast, Tourist is self-adaptive O(1)-hop overlay in most cases, which adapts its routing efficiency between one-hop and two-hop.

VII. CONCLUSION

When each node's bandwidth cost is limited by an upper threshold, how to design a structure overlay that fully utilizes all the nodes' available bandwidth to achieve as high routing efficiency as possible? We proposed Tourist as a possible solution.

Since different nodes will set different thresholds, Tourist is designed as a heterogeneous protocol, in which different nodes can run at different levels and keep routing tables with different sizes. Even if a node does not change its level, its bandwidth cost for routing table maintenance will change along with time because the system environment (especially the system scale and the churn rate) is always changing. Therefore, Tourist allows nodes to adjust their levels dynamically, which makes the whole system self-adaptive to the environment. To be efficient, Tourist uses multicast method for routing table maintenance,

which enables each node to keep a large routing table at a low cost. Large routing tables provide high routing efficiency. In most cases, Tourist can route all the messages within two hops. In an extremely large and dramatically dynamic system where it is difficult to guarantee two-hop routing completely, Tourist employs backup routing tables to provides an O(log*N*)-hop guarantee and route some messages via more than two hops.

We believe Tourist, along with some concurrent works, is just a start in the direction of making structured overlays self-adaptive. Further work includes real deployment, security research, especially on the application-level multicast for routing table maintenance, resilience to network partition, theory analysis to show the upper bound of the routing efficiency that can be obtained in a given environment, etc. Currently, we are deploying Tourist protocol into Granary [34], an object-oriented peer-to-peer storage system. We hope its practical experience would direct our future work.

ACKNOWLEDGMENT

We thank Professor Jie Wu, Professor Ben Y. Zhao, Zheng Zhang, Xuezheng Liu, Shuming Shi, and Haitao Dong for their discussion on the basic idea of this paper. We also thank Yinghui Wu for his ONSP platform that simplifies our experiment significantly.

REFERENCES

[1] F. Bek, M. F. Kaashoek, D. Karger, R. Morris, and I Stoica. Wide-area cooperative storage with CFS. 18th ACM Symposium on Operating Systems Principles (SOSP '01). October 2001.

[2] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. IEEE JSAC, 20(8). October 2002.

[3] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. The International Conference on Dependable Systems and Networks (DSN 2004). June 2004.

[4] M. Costa, M. Castro, A. Rowstron, and P. Key. PIC: Practical Internet coordinates for distance estimation. In 24th International Conference on Distributed Computing Systems (ICDCS 2004). March 2004.

[5] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. 5th Symposium on Operating Systems Design and Implementation (OSDI '02). December 2002.

[6] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. ACM SIGCOMM 2004. August 2004.

[7] A. Gupta, B. Liskov, and R. Rodrigues. One hop lookups for peer-to-peer overlays. 9th Workshop on Hot Topics in Operating Systems (HOTOS IX). May 2003.

[8] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. First Symposium on Networked Systems Design and Implementation (NSDI '04). March 2004.

[9] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: building an efficient and stable P2P DHT through increased memory and background overhead. 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03). February 2003.

[10] N. J.A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and Alec Wolman. SkipNet: A scalable overlay network with practical locality properties. 4th USENIX Symposium on Internet Technologies and Systems (USITS 2003). March 2003.

[11] J. Hu, M. Li, W. Zheng, D. Wang, N. Ning, and H. Dong. SmartBoa: Constructing p2p overlay network in the heterogeneous Internet using irregular routing tables. 3rd International Workshop on Peer-to-Peer Systems (IPTPS '04). February 2004.

[12] J. Hu, H. Dong, W. Zheng, D. Wang, and M. Li. Twins: 2-hop structured overlay with high scalability. The International Conference on Computer Science (ICCS '04). June 2004.

[13] J. Hu. Research on large-scale distributed storage system based on peer-to-peer architecture. Ph. D thesis (in Chinese). June 2005. Available at http://hpc.cs.tsinghua.edu.cn/granary/.

[14] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. 15th International Conference on Parallel and Distributed Computing Systems (PODC 2002). July 2002.

[15] M. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal hash table. 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03). February 2003.

[16] J. Li, J. Stribling, R. Morris, and M. Frans Kaashoek. Bandwidth-efficient management of DHT routing tables. Second Symposium on Networked Systems Design and Implementation (NSDI '05). May 2005.

[17] Q. Lian, W. Chen, Z. Zhang, S. Wu, and B. Y. Zhao. Z-Ring: Fast prefix routing via a low maintenance membership protocol. 13th IEEE International Conference on Network Protocols (ICNP 2005). September 2005.

[18] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: distributed hashing in a small world. 4th USENIX Symposium on Internet Technologies and Systems (USITS 2003). March 2003.

[19] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. 1st International Workshop on Peer-to-Peer Systems (IPTPS '02). March 2002.

[20] T. S. E. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. IEEE INFOCOM 2002. June 2002.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. ACM SIGCOMM 2001. August 2001.

[22] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. 2004 USENIX Annual Technical Conference (UESNIX '04). June 2004.

[23] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. ACM SIGCOMM 2005. August 2005.

[24] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale persistent peer-to-peer storage utility. 18th ACM Symposium on Operating Systems Principles (SOSP '01). October 2001.

[25] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. International Conference on Distributed Systems Platforms (Middleware 2001). November 2001.

[26] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. Multimedia Computing and Networking 2002 (MMCN '02). January 2002.

[27] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. ACM SIGCOMM 2001. August 2001.

[28] C. Tang, M. J. Buco, R. N. Chang, S. Dwarkadas, L. Z. Luan, E. So, and C. Ward. Low traffic overlay networks with large routing tables. ACM SIGMETRICS 2005. June 2005.

[29] R. Tian, Y. Xiong, Q. Zhang, B. Li, B. Y. Zhao, and X. Li. Hybrid overlay structure based on random walk. 4th International Workshop on Peer-to-Peer Systems (IPTPS '05). February 2005.

[30] Y. Wu, M. Li, and W. Zheng. ONSP: Parallel overlay network simulation platform. The 2004 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '04). June 2004.

[31] Z. Xu, M. Mahalingam, and M. Karlsson. Turning heterogeneity into an advantage in overlay routing. IEEE INFOCOM 2003. April 2003.

[32] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an Internetwork. IEEE INFOCOM 1996. June 1996.

[33] B. Y. Zhao, Y. Duan, L. Huang, A. Joseph, and J. Kubiatowicz. Brocade: Landmark routing on overlay networks. 1st International Workshop on Peer-to-Peer Systems (IPTPS '02). March 2002.

[34] W. Zheng, J. Hu, and M. Li. Granary: Architecture of object-oriented Internet storage service. IEEE International Conference on E-Commerce Technology for Dynamic E-Business (CEC-EAST 2004). September 2004.