PeerWindow: An Efficient, Heterogeneous, and Autonomic Node Collection Protocol

Jinfeng Hu, Ming Li, Hongliang Yu, Haitao Dong and Weimin Zheng
Department of Computer Science and Technology, Tsinghua University, P. R. China
hujinfeng00@mails.tsinghua.edu.cn, mingli@cs.umass.edu,
hlyu@tsinghua.edu.cn, dht02@mails.tsinghua.edu.cn, zwm-dcs@tsinghua.edu.cn

Abstract

Nodes in peer-to-peer systems need to know the information about others to optimize neighbor selection, resource exchanging, replica placement, load balancing, query optimization, and other collaborative operations. However, how to collect this information effectively is still an open issue. In this paper, we propose a novel information collection protocol, PeerWindow, with which each node can collect a large amount of pointers to other nodes at a very low cost. Compared to existing protocols, PeerWindow is 1) efficient, the cost of collecting 1,000 pointers being less than 1kbps in a common system environment, 2) heterogeneous, nodes with different capacities collecting different amounts of information, and 3) autonomic, nodes determining their bandwidth cost for node collection by themselves and adjusting it dynamically. PeerWindow can be used in many existing peer-to-peer systems and has tremendous potential for future expansions.

1. Introduction

Peer-to-peer nodes need to know the information about others. However, in most cases they have too limited knowledge about the outside world (sometimes they only hold a small routing table that contains no more than 100 pointers). In peer-to-peer backup systems, nodes need to find partners with similar [4] or different [10] operating systems, to reduce redundant data storing or to guard against simultaneous virus attack. In resource trading systems [5], nodes need to find adequate bargainers in terms of capacity, availability, physical location, bidding price, etc. In load balancing algorithms [6], heavily-loaded nodes need to find lightly-loaded ones to transfer the overload. In range query protocols [1], nodes need to gather other nodes' information for query optimization. In file sharing protocol of GUESS [19], nodes need to collect a large amount of pointers to other nodes to increase the local hit rate of submitted queries.

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

All these examples indicate that nodes in peer-to-peer systems have a desire to know others' information. However, cost-effective information-collection method is still an open issue. For the lack of general solutions, most projects design their own methods on top of an existing overlay. For example, Pastiche modifies Pastry, Mercury uses random walk on a small-world overlay, and GUESS piggybacks node pointers upon response messages within a Gnutella-like unstructured overlay.

The goal of our work is to propose a more general solution for node collection, which can be used in existing and future peer-to-peer systems. To be adaptive to the peer-to-peer environment, the protocol must hold following properties:

1. *Efficiency.* Nodes are able to collect a large amount of pointers (a *pointer* means a piece of information about another node) at a low cost. We believe efficiency is the most significant property for a node collection protocol because obviously the more pointers a node collects, the more satisfactory partners it may find locally when desired.
2. *Heterogeneity.* For the inevitable heterogeneity of peer-to-peer systems [13], nodes with different capacities should be allowed to collect different amounts of pointers (at different bandwidth cost). Heterogeneity makes it possible for the weak nodes to participate in the system, and also prevents those powerful nodes from being restricted by the weak ones.
3. *Autonomy.* We should let every node determine its cost for node collection (and thereby the amount of collected pointers) independently and be able to adjust it dynamically. This is accordant with the autonomy

spirit of "peer-to-peer" and makes the protocol adaptive to the environment changes.

The most critical problem in designing such a protocol is the maintenance of pointers: when a node joins or leaves the system, all the related nodes should update their pointers timely. There are two common ways for pointer maintenance: explicit probing (send heartbeat messages to all the neighbors[1] periodically) and multicasting (when a node joins or leaves, multicast the event to all the related nodes). Explicitly probing is not efficient because most probes get active response, and therefore have no positive effects on pointer-state updating. For example, assuming that the nodes' average lifetime[2] is about 2 hours [13] and a node probes all its neighbors every 30 seconds, then $239/240 \approx 99.58\%$ of the probes would return positively, indicating that the corresponding neighbors are still alive. Actually, these messages can be seen as a waste. If the node uses 10kbps for pointer maintenance, it can only maintain 600 pointers (assuming each heartbeat message is 500-bit in size). Compared to the scale of the whole system (perhaps comprising millions of nodes), this amount is very small.

In contrast, multicasting is more efficient because a node only receives messages when its neighbors change their state (joining or leaving). In another word, all the received messages are useful for pointer-state updating. While multicasting can achieve high efficiency, it faces another critical problem: given a heterogeneous system, how to determine which nodes hold pointers to a given node? Obviously, such information cannot be stored explicitly, because 1) it greatly complicates the protocol; 2) it is hard to determine where to place it and how to keep it available and reliable; and 3) even if we can obtain such information when needed, efficient multicast algorithm is still a hard problem.

In this paper, we propose *PeerWindow*, a novel node collection protocol that solves this maintenance problem and hence holds the above three properties (efficiency, heterogeneity and autonomy) simultaneously. Each PeerWindow node has a nodeId and a self-determined attribute *level*. A node at level l keeps about $N/2^l$ pointers (where N is the total number of the nodes). Then nodes with different capacities run at different levels. PeerWindow sets a smart mapping between a node's identifier and the pointers it should maintain, which makes it possible to judge whether a node keeps a pointer to another node by simply looking at their identifiers and levels. In this way, when a node joins or leaves, PeerWindow can figure out which nodes need to know the event without additional information storing. Furthermore, PeerWindow devises a tree-based multicast protocol that disseminates the event efficiently. PeerWindow nodes are allowed to adjust their levels dynamically to tune their bandwidth cost, which makes PeerWindow more adaptive than previous protocols.

In the following, we will first outline the PeerWindow protocol in section 2, and then discuss how to use PeerWindow in some existing peer-to-peer systems in section 3. Protocol details are presented in section 4. After reporting the experiment results in section 5, we introduce related works in section 6 and make final conclusion in section 7.

[1] If node A keeps a pointer to node B, then say B is a neighbor of A.
[2] Lifetime means the period from the time when a node joins system to the time when it leaves.

2. System Overview

Every node in PeerWindow keeps a large list of pointers to other nodes, called *peer list*. PeerWindow uses multicast to maintain the peer lists: a state-changing event, e.g., a node's joining, leaving or information changing, will be multicast to all the nodes who are interested in the changing node, in another word, whose peer list contains (or should contain) a pointer to the changing node. As discussed in the introduction, the most challenging problem is to determine which nodes keep, or should keep, such pointers. PeerWindow solves it by setting a novel mapping between a node and its peer list.

Each PeerWindow node has a unique identifier *nodeId* that is 128-bit-long, commonly the result of consistent hashing of its public key or IP address. Thereby, nodes should be evenly distributed in the nodeId space. Additionally, every node has another self-determined attribute *level*, which can be 0, 1, 2, and so on. It is demanded that the peer list of an l-level node should contain pointers to all the nodes whose nodeId's first l bits are the same with the local one.

Figure 1 shows a 10-node PeerWindow example, in which nodeIds are 4-bit long. The first l bits of an l-level node are called the node's *eigenstring*, which is underlined in figure 1. It can be seen that 0-level nodes have blank eigenstrings, 1-level nodes can be classified into two categories according to their different eigenstrings, 2-level nodes into four categories (only three of them are not empty in the example, nodes with eigenstring "11" being absent) and so on. In general, l-level nodes can be classified at most into 2^l categories with different eigenstrings. A pointer consists of the corresponding node's IP address, nodeId, level, and a

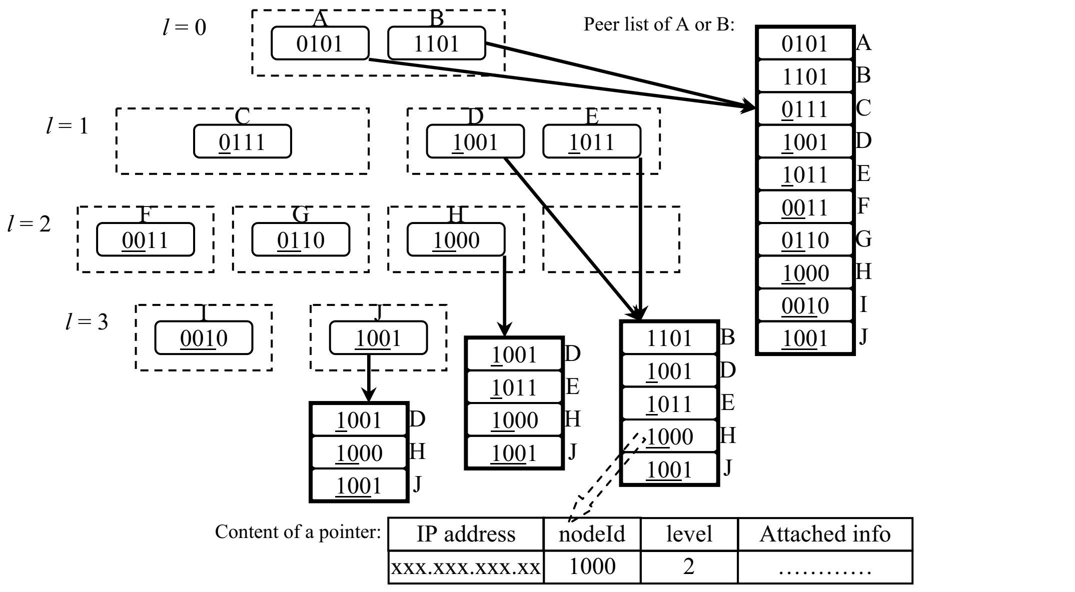

Figure 1. A PeerWindow example with 10 nodes. NodeIds are 4-bit long and eigenstrings are underlined. Peer lists of nodes C, F, G, and I are not shown for the neat of the figure

piece of attached info that can be specified by upper applications.

Peer list has the following properties:

1. Nodes with the same eigenstring must have the same peer list, e.g., nodes D and E in figure 1.

2. If a node's eigenstring is a prefix of another node's (e.g., nodes E and H), the former's peer list must completely cover the latter's. We call the former is *stronger* than the latter, in another word, the latter is *weaker* than the former.

3. A 0-level node's peer list covers the whole system, e.g. node A.

4. Two nodes at the same level, but with different eigenstrings must have entirely different peer lists, e.g., nodes C and E.

5. All nodes with the same eigenstring are fully connected through their peer lists ("fully connected" means that within a set of nodes, everyone keeps pointers to all the others), e.g., all the nodes with eigenstring "1", i.e. nodes D and E.

Attention should be paid to a special scenario where there is no 0-level node in the whole system (i.e., removing node A and B from figure 1). In this case, the system will split into two parts that are wholly unrelated to each other (nodes CFGI and nodes DEHJ). PeerWindow protocol does not rely on 0-level nodes and is able to work well in each part of a split system. To be convenient for statement, in this paper we first assume that there are 0-level nodes in the system and call them *top nodes*, and then discuss the special handling for a split system in section 4.4.

All the nodes whose peer list contains a pointer to a given node form a set, called the node's *audience set*. As discussed above, when a node changes its state, including joining, leaving, shifting its level and changing the attached info, all the nodes in its audience set must be informed. PeerWindow does not store audience sets explicitly, because they can be recognized simply by looking at the related nodes' nodeIds and levels.

For example, in figure 1, node E's audience set consists of these nodes: node A and B at level 0; node D and E at level 1 with eigenstring "1"; and node H at level 2 with eigenstring "10". That is to say, the audience set contains all the nodes whose eigenstring is a prefix of E's nodeId (1011).

Generally, the audience set of a node with nodeId "$N_0N_1N_2N_3$…" comprises all the nodes with eigenstrings of "" (blank string), "N_0", "N_0N_1", "$N_0N_1N_2$", and so on, as illustrated in figure 2. Therefore, a node in the audience set can directly judge whether another node is also in the set, by checking the node's eigenstring. In this way, a top node in an audience set can easily find out the whole set, whilst an arbitrary node in the set can find out those nodes (also in the set) at the same level or at lower levels. For example, in figure 2, 0-level nodes know the whole audience set, 1-level nodes know those at level 1, 2

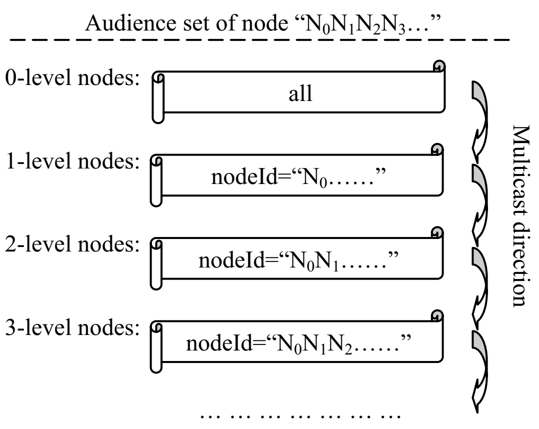

Figure 2. Composition of an audience set. The changing node's nodeId is "$N_0N_1N_2N_3$…"

and 3, 2-level nodes know those at level 2 and 3, and so on.

This leads to a useful deduction: when a top node gets a node's state-changing event, it has sufficient information to multicast it around the audience set of the changing node, as long as the message strictly flows from stronger nodes to weaker nodes during the multicast process, as illustrated in figure 2. A simple manner is by gossip: the top node first initiates a gossip around all the top nodes, and then sends the event message to a level-1 node L_1; L_1 then initiates a gossip around all the level-1 nodes, and then sends the message to a level-2 node. This process continues until all the nodes in the audience set receive the message. Here we emphasize the *feasibility* of the multicast design. Various multicast protocols can be devised in this environment, with different efficiency, reliability, and redundancy. In section 4.2, we will propose a tree-based multicast as the basic design of PeerWindow.

Noting that a multicast is always originated by a top node, a changing event must firstly be sent to a top node before being multicast around the audience set. To enable this, every PeerWindow node also keeps another list, i.e., *top-node list*, which contains pointers to *t* top nodes. Commonly we set $t = 8$.

Before turning to the protocol details, we first estimate PeerWindow's performance.

Assuming that nodes' average lifetime is *L* seconds, each node changes its state *m* times during the lifetime (including the joining and the leaving), and the multicast protocol has *r*-degree redundancy (i.e., a node receives *r* messages for each event), then a node will receive $\frac{m \times r}{L}$ messages per second on average for maintaining one pointer. Assuming that the average message size is *i* bits and a node would like to spend *W* bps bandwidth on node collection, the number of pointers it can collect (namely, the size of its peer list) is about $p = \frac{W \times L}{m \times r \times i}$. This formula shows that PeerWindow achieves the three properties proposed in the introduction:

1. *Efficiency*. Assuming that in a common peer-to-peer environment where $L = 3600$ (less than the measured result of real peer-to-peer systems [13]), $m = 3$ (a node changes its state once per lifetime), $i = 1000$ (sufficient for most applications) and $r = 1$ (a tree-based multicast is used, like that in section 4.2), a very weak node (e.g., a modem-linked node) would spend only 10% of its bandwidth, about 5kbps, on PeerWindow. Then, it can collect about $p = 6000$ pointers, which is a very large amount. For those high-bandwidth nodes, it is very easy to collect much more pointers by spending more bandwidth.

2. *Heterogeneity*. A PeerWindow node can determine its bandwidth cost on node collection by itself. Thus powerful nodes will never be restricted by the limit of the low bandwidth of those weak nodes.

3. *Autonomy*. Nodes can adjust their levels to be adaptive to the changing environment. Essentially, this is because of the direct proportion between peer list size *p* and nodes' lifetime *L*: peer lists will automatically expand when the system turns stable gradually. For instance, in a given PeerWindow system, a modem node sets an upper bandwidth threshold 5kbps, collecting about 6000 pointers, at level *l*. Then the system gradually turns stable (i.e., the average lifetime *L* increases), resulting in fewer events and less bandwidth cost. Once the bandwidth cost drops to a value below 2.5kbps, the node will automatically shift the level to $l-1$ and the peer list will inflate accordingly. Thereafter, the bandwidth cost returns to 5kbps, with about 12000 pointers in the peer list.

3. Usage

PeerWindow endows every node with a large number of pointers to other nodes, which can facilitate upper applications in many ways. In this section, we present a brief discussion on how to utilize PeerWindow to serve different requirements.

Looking at the level value for powerful nodes. A simple and direct way is finding powerful nodes by looking at the level value in the pointers. Practical experience shows that nodes with higher bandwidth (at high levels[3] in PeerWindow) also tend to stay longer and contribute more resources [15].

[3] "Higher level" means smaller level value, i.e., 0 is the highest level.

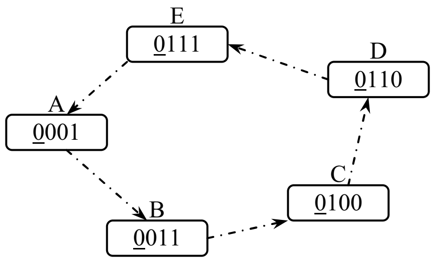

Figure 3. illustration of failure detection

Directly using the attached info. Some applications need to exchange some brief information among the nodes. They can directly attach the information into the pointers. For example, GUESS [19] protocol can attach the number of shared files to the pointers. Backup systems [4][10] can attach operating system versions. Range-query systems [1] can attach load distribution, node-count distribution, and query selectivity. Bidding systems [5] can attach nodes' basic status, such as storage space, bandwidth, availability, software/hardware summary, approximate bid, etc.

Using compression techniques to express more info. PeerWindow pointers should be kept small, because large pointers will finally deflate the peer lists. Therefore, if nodes need to express much about their status, some compressing techniques should be combined. For example, Pastiche [4] can attach the content abstracts into the pointers. LOCKSS [11] can use bloom filter [2] to indicate whether a node contains a given digital document and attach the filter results into the pointers.

4. Protocol Details

4.1 Failure Detection

Peer-to-peer nodes can leave the system without notification. Therefore, leaving events must be detected by some means (then multicast around the audience set).

Recall that PeerWindow nodes can be divided into multiple groups according to their different eigenstrings. Nodes in each group are fully connected through their peer lists. Thus, all the nodes in a given group can be seen as a circle based on their nodeIds. It is demanded that a PeerWindow node always probe its right neighbor in the circle ("right" means the direction from small to large). Figure 3 shows an example, in which there are five nodes with eigenstring "0" and every node sends heartbeat messages to the node whose nodeId is just larger than it.

```
rcv_event(nodeId=M, event_type=et, step=s)
            //receive change event of node M at step s.
(1)  adjustList(list, et, M)
            //adjust the peer list according to the event.
(2)  Rs = getAudience(list, M)
            //get M's audience set from the peer list.
     for i := s + 1 to 64 do
(3)    Rn := getSuffix(Rs, localID, i − 1)
            //get all pointers to the nodes whose
            //nodeIds are the same with the local
            //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 of the
            //same (highest) level, randomly pick one.
(6)    send_event(P, M, et, i)
          //send the event to P, tagged as step i.
     od
```

Figure 4. Pseudocde of multicast

Once a node detects the failure of its right neighbor, it immediately reports the event to a top node, randomly chosen from its top-node list, and redirects its probing to the next neighbor. Note that such detection mechanism is resilient to concurrent failures. For instance, node B and C concurrently leave the system in figure 3. Then node A will first detect B's failure and report it to a top node. After that it removes B from its peer list and redirects its probing to node C, and then immediately detects C's failure and reports it.

4.2 Tree-based Multicast

As mentioned in section 2, multicast protocol can be designed in many ways. In this subsection we present a tree-based method as PeerWindow's basic design, the pseudocode of which is shown in figure 4.

The fundamental principle of the protocol 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 one. Thus there will be two nodes having received the event, with different first bits in their nodeIds. 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 one. Then, all the four informed nodes have different first two bits with each other in their nodeIds. In general, at step s, every informed node sends the event to another node whose nodeId has the same first s bits and different (s+1)th bit (see figure 4(3)). This process continues until no more appropriate node can be found (see figure 4(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, as shown in figure 4(5).

The multicast protocol has four major properties:

1. Event messages always flow from the stronger nodes to the weaker nodes.

2. Different nodes have different out-degrees. Stronger nodes have more out-degrees than weaker ones. The root node of the multicast tree has approximately $\log_2 N$ out-degrees.

3. An event message can reach all the nodes in the audience set through about $\log_2 N$ steps.

4. The multicast tree is not pre-determined. Every node dynamically chooses the next target in the multicast tree at runtime (see figure 4(5)).

To guard against stale pointers in the peer lists, acknowledgement is required for all the multicast messages. When a message gets no response after three continuous attempts, the corresponding pointer will be removed from the peer list and the message will be redirected to a new target node (i.e., turn back to the line (3) in figure 4).

4.3 Joining Process and Level Shifting

A new node contacts a bootstrap node that is already in the system for joining. Four steps are needed: 1) finding out a top node, 2) determining the joining node's level, 3) downloading the peer list and top-node list, and 4) multicasting its joining event around its audience set. The key problem here is how to determine which level is suitable for its capacity before practical running. The estimation is by this way: the top node tells the new node its own level l_T, as well as its current bandwidth cost W_T that is dynamically measured. Then the new node estimates its level l_X based on these two values, as well as its own permitted bandwidth cost W_X: $l_X = \left\lceil l_T + \log_2 \frac{W_T}{W_X} \right\rceil$. More detailed description of these four steps can be found in [8].

A new node can also first set a low level so as to start working in a relatively short period, and then ask stronger nodes for a larger peer list. After completing the background downloading, it raises it level and reports the state-changing event to a top node. We call this process *warm-up*.

A node can adjust its level at runtime due to the change of the system environment or the upper bandwidth threshold set by the user. When a node raises its level, it should first download those required pointers from stronger nodes and then reports the event to a top node. When a node lowers its level, it removes those useless pointers from its peer list and reports the event.

4.4 Split PeerWindow

When the system is very large or very dynamic, no node can afford the bandwidth cost running at level 0. In this circumstance, the system will split into two parts, one comprising all the nodes whose nodeId starts with "0" and the other comprising all the nodes whose nodeId starts with "1". It can be easily seen that these two parts are wholly independent to each other (a node in one part must keep no pointer to any node of the other part) and each one is a complete PeerWindow (all the protocols proposed above in this paper are still suitable for each part).

To be general, PeerWindow is made up of several parts that are independent to one another. The highest-level nodes in each part are called *top nodes*. Every node maintains *t* pointers (in its top-node list) to the top nodes in its part. When a node changes its state or detects failure of another node, it reports the event to a top node, which will then multicast the event around the changing node's audience set, using the tree-based multicast protocol presented in section 4.2.

A top node's top-node list does not contain pointers to top nodes of its own part. Instead, it contains pointers to some top nodes of other parts, *t* pointers for each part. When a joining node X and its bootstrap node Y are not in the same part, X needs to find a top node of its own part first (step 1 of the joining process, seeing section 4.3). X accomplishes this by asking a top node in Y's part, say Z. Z's top-node list must contain *t* top nodes of X's part.

4.5 Top-node List Maintenance

Top-node lists are maintained in a lazy manner. When a node M reports an event to a top node, the top node should return a response, piggybacking $t-1$ pointers to top nodes, which will help M refresh its top-node list. If the report does not get a response, M will redirect the report to another top node within its top-node list. If all the top nodes in its top-node list are unavailable, it will then ask another node in its peer list for his top-node list as a substitution.

A top node's top-node list is maintained similarly. When a top node T works for another node's joining process, it chooses a live pointer from its top-node list and asks the corresponding node for $t-1$ pointers to top nodes of that part. If all the pointers in T's top-node list are stale, it will ask another top node of its own part for help.

4.6 Accuracy Improvement

Because of the Internet asynchrony, the multicast protocol can never be absolutely reliable. Therefore, there must be some errors in the peer lists, which fall into two types: absent pointers and stale pointers. Both of them are only of a very small fraction and do no substantial harm. An absent pointer would be automatically revised when the corresponding node leaves the system, while a stale pointer would be removed when being used during multicast procedure and getting no response. But these errors would accumulate before being revised, from the view of whole system.

To guard against this accumulation, PeerWindow devises a refreshing mechanism. Every node measures the lifetime of all the nodes in its peer list, and calculates the average lifetime of each level, noted LT_i, where *i* denotes the level. An *l*-level node multicasts its state around its audience set every $2 \cdot LT_l$ (by reporting to a top node). An *m*-level pointer that has not been refreshed for a period of $3 \cdot LT_m$ will be directly removed from the peer list, without explicit probing. This mechanism can limit the accumulation of both absent pointers and stale pointers, and make the error fraction of peer list convergent. In practice, most nodes never perform such refreshing multicast because their lifetimes are much shorter than twice the average lifetime.

5. Experiment Results

Our basic experiment goal is to simulate a 100,000-node PeerWindow in a common environment, where the distributions of node capacity and lifetime are both accordant with the measurement result of Gnutella in [13], and the Internet topology is generated by the Transit-Stub model [20]. After that, we examine PeerWindow's scalability and adaptivity, i.e., how PeerWindow varies when the system scale or the nodes' lifetime changes.

To make large-scale experiments possible, we first developed ONSP [17], a general platform for large-scale overlay simulation on a homogeneous cluster. ONSP is based on parallel discrete events and uses MPI for machine communication. Transit-Stub model is naturally integrated into ONSP as its network topology.

Considering that PeerWindow nodes with the same eigenstring would have the same peer list, we record all the correct peer lists in a centralized data structure, and only record erroneous items in nodes' individual data structures. This method has two advantages: making it possible to run the whole experiment in memory and facilitating the calculation of the error rate of the peer lists. Meanwhile, it does not harm the validity of the experiment results.

Based on ONSP, our experiment program comprises 1,600 lines of C++ code and is 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.

5.1 Common PeerWindow

We simulate a common PeerWindow with 100,000 nodes. That is to say, we first create 100,000 nodes on the ONSP platform and then let new nodes join and existing nodes leave, with almost the identical joining and leaving rates. In this experiment, the following characteristics are hold:

- Distribution of nodes' lifetime meets the measurement results of Gnutella (figure 6 of [13]), in which the average lifetime is about 135 minutes.
- Distribution of nodes' available bandwidth meets the measurement results of Gnutella (figure 3 of [13]).
- The user-set upper (input) bandwidth threshold is 1% of the node's total bandwidth, but cannot be less than 500bps (a small value that is affordable even for modem-linked nodes).
- The Transit-Stub network model is generated by the tool of GT-ITM [20], in which there are 120 transit domains, each containing 4 transit nodes. Every transit node has 5 stub domains, each containing 2 stub nodes. Thus, there are totally 4800 stub nodes. To reach the required 100,000-node scale, each stub node is assigned with about 20 PeerWindow nodes. Common latency parameters are set as follows: transit-to-transit latency is 100ms; transit-to-stub is 20ms; stub-to-stub is 5ms; and node-to-node is 1ms.
- Nodes join the system in a Poisson process, with the expectation of the time interval of two successive node joining events is 100,000/135 minutes.
- The event message size is 1,000 bits.
- During the multicast procedure, every medium node delays the message for 1 second that is spent on receiving, calculating and sending.

Figure 5 plots the distribution of the nodes at different levels. Somewhat surprisingly, there are more

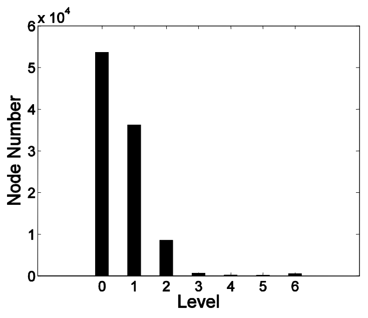
Figure 5. Node distribution in common PeerWindow

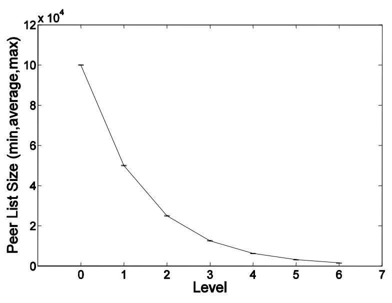
Figure 6. Size of peer lists at different levels

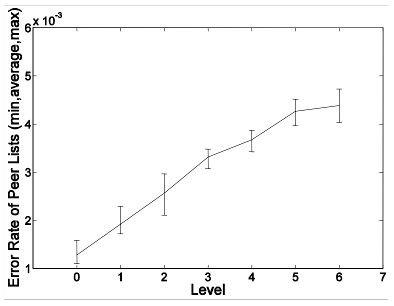
Figure7. Error rate of the peer lists at different levels

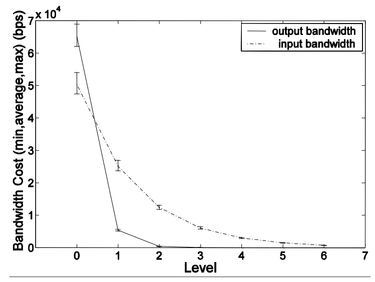
Figure 8. bandwidth cost at different levels

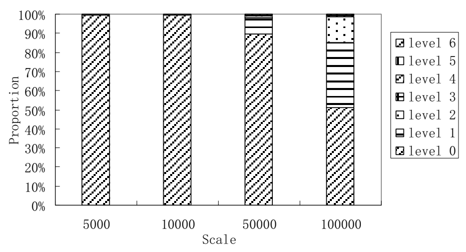
Figure 9. Node distribution in different system scales

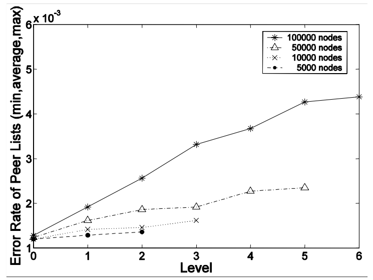
Figure 10. Average peer list error rate in different scales

than half of the nodes running at level 0. It seems quite a lot. However, it is really consistent with the measurement result of real peer-to-peer systems (seeing figure 3 of [13]) in which only 20% nodes' available bandwidth is less than 1Mbps. Perhaps our intuition that a large portion of Internet nodes are weak ones is somewhat questionable.

Figure 6 shows the size of the peer lists of the nodes at different levels. According to the PeerWindow protocol, an *l*-level node collects the pointers to all the nodes whose nodeId has an *l*-bit common prefix with the local nodeId. Because nodes are evenly distributed in the nodeId space, the peer lists of the nodes at a given level are almost of the same size. (Figure 6 plots the maximum and the minimum values, but they are hard to be distinguished.)

Although the peer lists are large, they have very few errors. As figure 7 shows, the error rate is less than 0.5%. This is because a changing event will be reported to the top node immediately when it is detected and multicast around the audience set without delay. The multicast needs $\log_2 100{,}000 \approx 16.6$ steps. Assuming that each step costs 500ms on average, all the nodes in the audience set will receive the event within $(1+0.5)\times 16.6 = 24.9s$, that is to say, a point will be kept stale for no longer than 25 seconds. Compared to the average lifetime of the nodes (135 minutes), the error rate will be no more than $25/(135\times 60) \approx 0.0035$, which is accordant with the experiment result.

Higher-level nodes have peer lists with fewer errors than lower-level nodes. This is because the multicast process ensures the higher-to-lower direction of the message flow, which indicates that higher-level nodes can revise their peer lists earlier than lower-level ones.

Figure 8 shows the input and output bandwidth for the peer list maintenance. As participated, the input bandwidth is in proportion to the peer list size. The input-bandwidth cost for every 1000 pointers is about 500bps. As discussed in section 4.2, higher-level nodes will have larger output-bandwidth cost. In this case, almost all the messages are sent from 0-level or 1-level nodes. But their output cost is only a little more than the input, also very light for these powerful nodes.

5.2 Scalability

The main impacts of the system scale are the distribution of the nodes and the error rate of the peer lists. Figure 9 depicts the variation of the percentage of the nodes at each level when the system scale changes, using different figure patterns for different levels. In a 5000-node PeerWindow, all the nodes run at level 0. When the system expands, there comes out more levels and more nodes tend to work at lower levels. This is because those weak nodes cannot afford the bandwidth

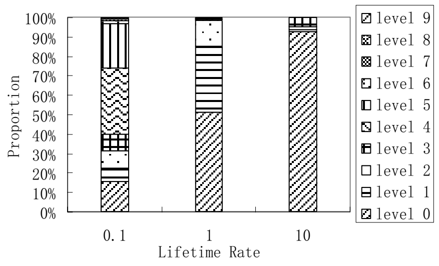

Figure 11. Nodes distribution with different lifetime rate (to the common PeerWindow)

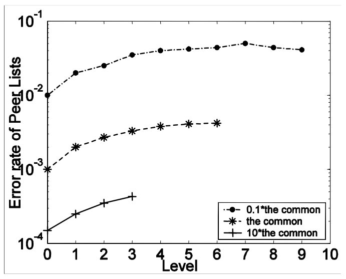

Figure 12. Average peer list error rate in the systems with different lifetime rates

cost at high levels in a large system. The error rate of the peer lists also rises (seeing figure 10), because multicast needs longer time and the errors in the peer lists are revised less timely. But the change is very slight.

5.3 Adaptivity

Nodes in different peer-to-peer systems will have different lifetimes, which are essentially determined by the usage model. Even in a given system, the nodes' lifetime may vary along with time. We assume that nodes' lifetimes are *Lifetime_Rate* times of that in the common case (section 5.1). The node distribution and the error rate of peer list at different *Lifetime_Rate* are shown in figures 11 and 12, respectively. Note that figure 12 uses the logarithmic scale on the y-axis.

When the *Lifetime_Rate* is 0.1 (this means that the average lifetime is 13.5 minutes) there comes out 10 levels and only about 15% 0-level nodes. This is because when the lifetime turns short, more state-changing events will occur in a given time interval. Therefore, a node can only maintain a small peer list and run at a low level.

Figure 12 shows that the peer lists' error rate also increases when lifetime turns shorter. This is because the error rate is proximately determined by the formula $error\_rate = multicast\_delay / lifetime$. Since the system scale does not change, the number of multicast hops ($\log_2 N$) also does not change. Thus the lifetime will be approximately in inverse proportion to the average error rate. As shown in figure 12, in a system with $Lifetime\_Rate = 0.1$, the average peer list error rate is about 10 times of that in the common case ($Lifetime\_Rate = 1$), which is between 1% and 5%. However, such a result can hardly turn to the reality because of its very short average lifetime.

6. Related Work

Most previous projects devised their node collection protocol based on some existing overlay structures.

RanSub [9] is based on an application-level multicast tree. Using information collection and distribution, RanSub offers every node O(log*N*) pointers. By explicit probes through these pointers, every node changes it parent node dynamically. In this way, the multicast tree is optimized piece by piece. GUESS [19] is based on a Gnutella-like unstructured overlay. By piggybacking some known pointers on the response of a ping or query message, every node can collect a large number of pointers, which are used for non-forwarding search. Pastiche [4] uses a modified Pastry to collect pointers to those nodes who are storing similar data with the local node. Mercury [1] is built on top of a small-world overlay. To optimize the attribute-base query and load balancing, Mercury deploys random walk upon the overlay to collect other nodes' information, including load distribution, node-count distribution, and query selectivity. Compared to these previous protocols, PeerWindow is not based on any existing overlays and simultaneously holds the properties of efficiency, heterogeneity, and autonomy. We believe PeerWindow can also be used in the above systems and works well.

Another peer-to-peer system in which nodes collect a large amount of pointers is the one-hop DHT [7], compared to PeerWindow, one-hop DHT treats almost all the nodes as homogeneous peers and costs too much for weak nodes when the system is very large and dynamic or some application-specified information should be attached into the pointers.

There are also some previous works aiming at peer-to-peer node information *aggregation* (not collection), such as SOMO [21], SDIMS [18] and Willow [12]. The main difference between these protocols and PeerWindow is that they summarize the state of the whole system (e.g. the total load of the current system), while PeerWindow simply presents individual nodes' information to others.

PeerWindow uses a prefix-based multicast for event notification. Prefix-based multicast has been proposed for a long time [16] and was also introduced into the global multicast service recently [14]. However, the multicast in PeerWindow has a substantial difference with previous protocols. In previous protocols, a message must be sent to all the nodes whose nodeIds have a common prefix (the groupId in I3 [14]), while in PeerWindow all the nodes receiving a given message do not have a common prefix, but their eigenstrings must be prefix of a given identifier (the changing node's nodeId).

Also there are some application-level multicast protocols that use prefix-based relationship to construct a multicast tree, e.g., Scribe [3]. In them, every node within a group needs to maintain the states of its parent node and children nodes, which is not desired in PeerWindow's multicast protocol.

7. Conclusion

The spirit of peer-to-peer system is collaboration, intercommunion, and resource exchanging among different nodes. All these operations are based on mutual understanding of the nodes. Therefore, letting peer-to-peer nodes know each other is very important. In this paper, we propose a novel node collection protocol PeerWindow that simultaneously holds the fine properties of efficiency, heterogeneity, autonomy, dynamical adjustability, self-organizing, and adaptivity. PeerWindow can be used in many existing peer-to-peer systems and we believe it can also serve well for future peer-to-peer system constructions.

Acknowledgement

We thank Professor Jie Wu for some valuable suggestions on the paper. We thank the anonymous reviewers for their useful comments. We thank Zheng Zhang, Ben Y. Zhao, Xuezheng Liu, Shuming Shi and Yun Mao for their discussion with us on the basic idea of PeerWindow. We also thank Yinghui Wu for his ONSP platform that simplifies our experiment significantly.

References

[1] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. SIGCOMM 2004. August 2004.

[2] B. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7), pages 422–426, July 1970.

[3] 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.

[4] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making Backup Cheap and Easy. OSDI '02. December 2002.

[5] B. F. Cooper and H. Garcia-Molina. Bidding for Storage Space in a Peer-to-Peer Data Preservation System. ICDCS '02. Junly 2002.

[6] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Dynamic Structured P2P Systems. INFOCOM 2004. March 2004.

[7] A. Gupta, B. Liskov, and R. Rodrigues. One Hop Lookups for Peer-to-Peer Overlays. HOTOS IX. May 2003.

[8] J. Hu, M. Li, H. Dong, and W. Zheng. PeerWindow: An Efficient, Heterogeneous, and Autonomic Node Collection Protocol. Full report, available at http://hpc.cs.tsinghua.edu.cn/granary/.

[9] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using Random Subsets to Build Scalable Network Services. USITS '03. March 2003.

[10] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A Cooperative Internet Backup Scheme. USNIX '03. June 2003.

[11] P. Maniatis, M. Roussopoulos, T. Giuli, D. S. H. Rosenthal, M. Baker, and Y. Muliadi. Preserving Peer Replicas By Rate-Limited Sampled Voting. SOSP '03. October 2003.

[12] R. van Renesse and A. Bozdog. Willow: DHT, Aggregation, and Publish/Subscribe in One Protocol. IPTPS '04. February 2004.

[13] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. MMCN '02. January 2002.

[14] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. SIGCOMM 2002. August 2002.

[15] B. Wilcox-O'Hearn. Experiences Deploying a Large-Scale Emergent Network. IPTPS '02. March 2002.

[16] J. Wu and L. Sheng. Deadlock-Free Routing in Irregular Networks Using Prefix Routing. PDCS '99. Auguest 1999.

[17] Y. Wu, M. Li, and W. Zheng. ONSP: Parallel Overlay Network Simulation Platform. PDPTA '04. June 2004.

[18] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. SIGCOMM 2004. August 2004.

[19] B. Yang, P. Vinograd, and H. Garcia-Molina. Evaluating GUESS and Non-Forwarding Peer-to-Peer Search. ICDCS '04. March 2004.

[20] E. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. INFOCOM 1996. June 1996.

[21] Z. Zhang, S. Shi, and J. Zhu. SOMO: Self-organized Metadata Overlay for Resource Management in P2P DHT. IPTPS '03. November 2003.