Histogram-Based Global Load Balancing in Structured Peer-to-Peer Systems

Quang Hieu Vu, *Member*, *IEEE*, Beng Chin Ooi, Martin Rinard, and Kian-Lee Tan

Abstract—Over the past few years, peer-to-peer (P2P) systems have rapidly grown in popularity and have become a dominant means for sharing resources. In these systems, load balancing is a key challenge because nodes are often heterogeneous. While several load-balancing schemes have been proposed in the literature, these solutions are typically ad hoc, heuristic based, and localized. In this paper, we present a general framework, HiGLOB, for global load balancing in structured P2P systems. Each node in HiGLOB has two key components: 1) a histogram manager maintains a histogram that reflects a global view of the distribution of the load in the system, and 2) a load-balancing manager that redistributes the load whenever the node becomes overloaded or underloaded. We exploit the routing metadata to partition the P2P network into nonoverlapping regions corresponding to the histogram buckets. We propose mechanisms to keep the cost of constructing and maintaining the histograms low. We further show that our scheme can control and bound the amount of load imbalance across the system. Finally, we demonstrate the effectiveness of HiGLOB by instantiating it over three existing structured P2P systems: Skip Graph, BATON, and Chord. Our experimental results indicate that our approach works well in practice.

Index Terms—Peer-to-peer, framework, load balancing, histogram, DHT, overlay network.

1 Introduction

Peer-to-peer (P2P) systems have emerged as an appealing solution for sharing and locating resources over the Internet. Several P2P systems have been successfully deployed for a wide range of applications, such as Gnutella [2], BitTorrent [3], Overnet [4] (file-sharing systems), SETI@home [5] (computing sharing system), Groove [6] (collaborative system), and Skype [7] (Internet telephony system). In fact, a recent study showed that P2P systems dominate up to 70 percent of Internet traffic [8]. Thus, it is critical to design a P2P system that is scalable and efficient.

To build an efficient P2P system, researchers have turned to structured architectures (e.g., Chord [9], CAN [10], Tapestry [11], Pastry [12], P-Grid [13], PIER [14], and BATON [15]), which offer a bound on search performance as well as completeness of answers. However, one key challenge that has not been adequately addressed in the literature is that of load balancing. In a large-scale P2P system, nodes often have different resource capabilities (storage, CPU, and bandwidth) [16]. Hence, it is desirable that each node has a load proportional to its resource capability. Furthermore, even if the system is homogeneous (where all nodes have the same resource capability), it is difficult to ensure that the load is uniformly distributed across the system because of the dynamism in P2P systems. As a result, it is important to design mechanisms that can balance the system load.

The basic approach to load balancing is to find a pair of nodes—one that is heavily loaded and the other lightly loaded—and redistribute the load across these two nodes. However, it is far from trivial to (globally) balance the load in a P2P system. There are two main issues in P2P's load balancing: 1) how to determine if a node is overloaded or underloaded, and 2) if so, how to find a suitable partner node with which to redistribute the load. A popular solution [17], [18], [19] is to let each node in the system query for the load of an arbitrary number of other nodes periodically. If the number of queried nodes is large enough, the node can approximate the average load of the system, and hence, it can determine if it is overloaded or underloaded. If the node is overloaded (or underloaded), it redistributes its load with the queried node having the lightest (or heaviest) load since that node should be a lightly (or heavily) loaded node. The main problem with this method is that it can only guarantee the global load balance of the system with some probability. On the other hand, [20] suggests a use of a separate DHT such as Skip Graph to maintain the nodes' load distribution. Nevertheless, this solution still has a problem: it incurs a substantial cost for maintaining complete information about the load at every node in the system.

In this paper, we propose a new framework, called Histogram-based Global LOad Balancing (HiGLOB) to facilitate global load balancing in structured P2P systems. Each node P in HiGLOB has two key components. The first component is a histogram manager that maintains a histogram that reflects a global view of the distribution of the load in the system. The histogram stores statistical information that characterizes the average load of non-overlapping groups of nodes in the P2P network. These nodes are connected to P through its neighbor nodes. The histogram information can be used for two purposes. On

- *Q.H. Vu, B.C. Ooi, and K.-L. Tan are with the School of Computing, National University of Singapore, Computing 1, Law Link, Singapore 117590. E-mail: hieuvq@nus.edu.sg, {ooibc, tankl}@comp.nus.edu.sg.*
- *M. Rinard is with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Building 32-G744, 32 Vassar Street, Cambridge, MA 02139. E-mail: rinard@lcs.mit.edu.*

Manuscript received 25 Jan. 2008; revised 18 Aug. 2008; accepted 22 Aug. 2008; published online 28 Aug. 2008.
Recommended for acceptance by A. Tomasic.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2008-01-0055.
Digital Object Identifier no. 10.1109/TKDE.2008.182.

one hand, it is used to determine if a node is normally loaded, overloaded, or underloaded. On the other hand, it is used to facilitate the discovery of a lightly loaded node or a heavily loaded node for the load-balancing process when it is needed. The second component of the system is a load-balancing manager that takes actions to redistribute the load whenever a node becomes overloaded or underloaded. The load-balancing manager may redistribute the load both statically when a new node joins the system and dynamically when an existing node in the system becomes overloaded or underloaded.

We exploit the routing metadata to partition the P2P network into nonoverlapping regions corresponding to the histogram buckets. We propose mechanisms to keep the cost of constructing and maintaining the histograms low. We further show that our scheme can control and bound the amount of load imbalance across the system. Finally, we demonstrate how HiGLOB can balance the load in three existing structured P2P systems—Skip Graph [21], BATON [15], and Chord [9]. To summarize, this paper makes the following contributions:

- It proposes a general framework that uses histograms to maintain a global view of the load distribution on structured P2P systems. These histograms enable efficient load-balancing algorithms that can effectively control the amount of load imbalance across the system to globally balance the load.
- We suggest two techniques that effectively reduce the cost of constructing and maintaining the histograms.
- We show how to apply the general framework to three well-known structured P2P systems: Skip Graph [21], BATON [15], and Chord [9].
- We present experimental results that characterize the effectiveness and efficiency of the proposed techniques.

The rest of the paper is organized as follows: In Section 2, we review some related work. Section 3 presents the proposed HiGLOB framework for load balancing in structured P2P networks. In Section 4, we demonstrate how to apply HiGLOB to several existing P2P systems. Finally, we present results from an experimental study in Section 5 and conclude the paper in Section 6.

2 Related Work

Load balancing across multiple nodes has been widely studied in the context of distributed systems. Techniques that are based on *static* and/or *dynamic* methods have been developed [17]. In static methods, load balancing is triggered when either a new node joins the system or an existing node leaves the system. When a new node joins the system, it attempts to find a heavily loaded node and take over some of the load from that node. On the other hand, when a node leaves the system, it searches for a lightly loaded node to pass its current load to that node. In a different approach, dynamic methods operate when nodes that have already joined the system become overloaded or underloaded. The overloaded (underloaded) node looks for a lightly (heavily) loaded node to balance the load between them. Most P2P systems apply either one or both of these methods for load balancing. The systems differ, however, in how they find lightly or heavily loaded nodes and in how the load is redistributed.

A general technique to find a lightly (heavily) loaded node is to randomly contact a number of nodes among which the node with the heaviest (lightest) load is considered a heavily (lightly) loaded node [18], [19], [22], [23], [24], [25], [26], [27]. Since this technique is based on a randomized approach, it can only provide global load balancing with some probability. Moreover, its effectiveness depends on the number of contact nodes—while a small number of contact nodes may not lead to satisfactory load balancing, contacting a large number of nodes incurs a high overhead since it takes $O(logN)$ effort to contact a node. To keep the overhead low, Adler et al. [28] suggests that only one node is randomly contacted; other contacting nodes are just neighbor nodes of the randomly contacted node. Similarly, in BATON [15], only neighbor nodes in the routing tables are examined.

On the other hand, in [29], each node in the system repeatedly checks its neighbor nodes in the maintenance process to detect load imbalance across the system. Even though this method is able to achieve load balance of nodes when the system is in steady state, there is no guarantee of load balance when the system is in dynamic state. It is because load balancing is only done locally between neighbor nodes. In [20], a separate Skip Graph is used to maintain the distribution of the load across the nodes in the system. Even though this technique can control the balance of load among the nodes, it is expensive to maintain the Skip Graph structure when nodes join or leave, or when data is inserted or deleted.

A closely related work to our system is that in [30], which also employs histograms to keep the load distribution of the system. However, the way the system constructs and maintains histograms is totally different from our method. In this method, a node maintains its histogram by periodically sampling neighbor nodes in its vicinity to get local load distribution and exchanging this local load information with randomly selected far-away nodes to construct histogram. The process of selecting a far-away node is based on a random walk algorithm in which the node first sends a request with $logN$ hop Time-To-Live to a random neighbor node. After that, at each node along the walking path, a random neighbor node of that node is selected to forward the request. Finally, the node at the end of the walking path returns its local load information to the requester node. However, because of the randomness in selecting far-away nodes, this method also suffers from the same problems of methods that employ the randomized paradigm: it is costly to maintain histograms, and the global load balance can only be provided with some probability.

To balance the load between a lightly loaded node and a heavily loaded node [18], [19], [23], [26], [27], [31], use the concept of *virtual nodes*. In these systems, each peer may keep several virtual nodes. As a result, the overloaded node simply needs to transfer some of its virtual nodes to the lightly loaded node to achieve load balancing. In case the overloaded node cannot find any virtual node to pass to the lightly loaded node because virtual nodes are large, the node can split a large virtual node into smaller ones and pass some of them to the lightly loaded node. Rao et al. [18] further proposes that the load balancing process can also operate between a group of

several heavily loaded nodes and lightly loaded nodes at the same time. A drawback of this technique is that it takes more storage to keep additional virtual nodes, consumes more bandwidth for node maintenance, and increases query latency due to having more nodes in the system. To overcome these limitations, instead of using virtual nodes, the selected lightly loaded node can leave its current position and rejoin next to the heavily loaded node to balance the load [15], [20], [32].

Load balancing can also be handled using a *preventive* mechanism that avoids load imbalance [9], [33]. In [9], each peer node keeps exactly the same *logN* virtual nodes. As a result, with high probability, the system is load balanced. However, as discussed before, the cost of keeping additional virtual nodes at a peer is expensive. On the other hand, Byers et al. [33] proposes that for each piece of inserted data, multiple hash functions are used to find multiple nodes, and the most lightly loaded node is selected to store the data. When a data item is deleted or searched, multiple hash functions are also used to find the node storing the data. Nevertheless, this simple approach is not efficient since it incurs an expensive cost for data insertion, data deletion, and data search.

Before leaving this section, we note that there is also a class of techniques that manage overload, e.g., caching or replicating popular items/operations across several nodes [34], [35], [36], [37], [38]. We do not discuss these techniques since they are orthogonal and complementary to the load-balancing approaches presented in this paper.

3 The HiGLOB Framework

In this section, we present the HiGLOB framework. We focus on the histogram and load-balancing managers.

3.1 The Histogram Manager

The objective of the histogram manager is to maintain statistics about the load distribution across the entire P2P network. These statistics allow a node to know its own load status (in comparison with other nodes in the system) and to identify its counterpart if global load balancing is triggered.

3.1.1 Histogram Structure

In our framework, each peer node P stores an approximate histogram, keeping the load distribution of the system. The histogram contains several buckets, each of which keeps statistical information about the load of a group of nodes that is connected to P through a neighbor node. The statistical information includes the current *workload* of CPU, storage, and bandwidth of nodes in the group. Additionally, to balance the load of heterogeneous nodes, the statistical information also contains a summary of the available resources of the group (also in terms of CPU, storage, and bandwidth). The *load* of a node or group is calculated as the ratio between the current *workload* and the capability of the node or group.[1] As a result, from the histogram information, the system can balance the load of nodes according to any one of a variety of node capabilities—storage, CPU, or bandwidth. Since histogram buckets have to be disjoint, different groups belonging to different buckets have to be nonoverlapping with each other. We define a histogram as follows:

1. If nodes are homogeneous, the *load* of a node or group is simply the current *workload* of that node or group.

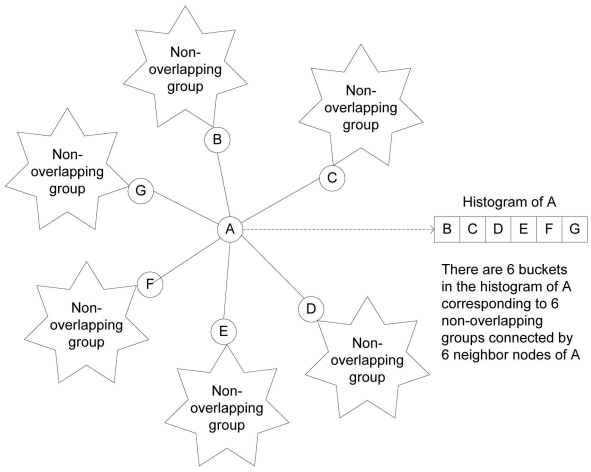

Fig. 1. Histogram structure.

Definition 1. *A histogram of a node is constructed by partitioning the network into* nonoverlapping groups, *each of which is connected to the node by a neighbor node. Each bucket in the histogram keeps statistical information of a group of nodes.*

Fig. 1 illustrates a histogram structure. In this figure, node A has six neighbor nodes, and hence, its histogram contains six buckets corresponding to six nonoverlapping groups connected by six neighbor nodes.

To build this histogram structure, we need to be able to partition the P2P network into several disjoint groups. However, this is difficult since there may be multiple paths between two nodes. In other words, a node may be connected to the histogram owner node through different neighbor nodes, and hence, it can belong to different groups (violating the nonoverlapping constraint). Our solution is based on the **key observation** that the routing/searching algorithm in structured P2P systems implicitly partitions the search space (and hence, the network) into nonoverlapping regions. When a node processes a query, it always forwards the query to the farthest neighbor node in the routing table that does not overshoot the searched key. In this way, after each search step, the search region is reduced to a group of nodes falling between the neighbor node receiving the query and the next neighbor node following that node in the routing table. In other words, neighbor nodes in the routing table virtually separate the whole network into search regions, and each node in the network belongs to exactly one of these search regions. As a result, we propose that nonoverlapping groups are formed in the same way: for each group of nodes connected by a neighbor node in the histogram, it contains all nodes in the search region limited to that neighbor node. We note that how search regions are formed depends on specific systems. We will defer this discussion to the next section. With the histogram structure, we derive an important theorem.

Theorem 1. *If the maximum imbalance ratio between the load of a node and the average load of a group of nodes in its histogram is k, the maximum load imbalance ratio of the system, which is the load ratio between the heaviest loaded node and the lightest loaded node, is k^2.*

Proof. Let N be the number of nodes in the system, x and y be respectively the heaviest loaded node and the lightest loaded node of the system, w_x and w_y be their loads, and A be the average load of the whole system.

Since there are no groups in the histogram of x, whose average load is less than $\frac{w_x}{k}$, the total load of the whole system cannot be less than

$$(N-1)\cdot\frac{w_x}{k}+w_x.$$

In other words

$$N\cdot A\geq(N-1)\cdot\frac{w_x}{k}+w_x\Rightarrow w_x\leq\frac{N\cdot A\cdot k}{N-1+k}. \quad (1)$$

Similarly, since there are no groups in the histogram of y, whose average load is greater than $w_y\cdot k$, the total load of the whole system cannot be greater than

$$(N-1)\cdot w_y\cdot k+w_y.$$

In other words

$$N\cdot A\leq(N-1)\cdot w_y\cdot k+w_y\Rightarrow w_y\geq\frac{N\cdot A}{(N-1)\cdot k+1}. \quad (2)$$

Combining (1) and (2) yields

$$\Rightarrow\frac{w_x}{w_y}\leq\frac{N\cdot A\cdot k}{N-1+k}\cdot\frac{(N-1)\cdot k+1}{N\cdot A}$$
$$\Rightarrow\frac{w_x}{w_y}\leq\frac{k^2\cdot(N-1+k)-(k^3-k)}{N-1+k}<k^2,\forall k>1.$$

□

3.1.2 Histogram Construction and Maintenance

The load distribution histogram of a node is first constructed when the node joins the system. Later, histogram values can be updated when the load distribution of the system changes. To construct a histogram, a node needs information from all neighbor nodes connected to it. To update a histogram or a bucket value, a node only needs updated information from the neighbor node connected to the corresponding group of that bucket. Specifically, when a new node joins the system, all of its neighbor nodes must send it their histogram or the part of their histogram that contains necessary information for constructing a new histogram to it. For histogram maintenance, when there is any change in the load of a node, the node informs its neighbor nodes about the change. When a node receives an update histogram message from a neighbor node, it updates the average load of the group of nodes connected by that neighbor node. After that the node continues to notify other neighbor nodes but the sender about the change. Details of how to construct and update histograms depend totally on specific systems. We shall discuss this in Section 4.

3.1.3 Improvement Techniques

While histograms are useful, the cost of constructing and maintaining them may be expensive especially in dynamic systems. As a result, we introduce two techniques that reduce the maintenance cost.

- **Reduce the cost of constructing histogram.** Constructing a histogram for a new node may be expensive since it requires histogram information from all neighbor nodes. Additionally, the histograms of the new node's neighbors also need to be updated since adding a new node to a group of nodes changes the average load of that group. Constructing and maintaining histograms may therefore be expensive if nodes join and leave the system frequently. In light of the fact that every new node in the P2P system must find and notify its neighbor nodes about its existence while these neighbor nodes need to send their information to the new node to setup connections after that, we suggest that histogram information can be piggybacked with messages used in this process. In this way, we can avoid sending separate histogram messages and totally eliminate the effect of node join on the histogram construction of the new node and histogram update of its neighbor nodes. The overhead cost of using histograms is now solely based on histogram update messages caused by changing of load at nodes in the system.
- **Reduce the cost of maintaining histogram.** Maintaining histograms can be expensive since any load change at a node causes an update to be propagated to all other nodes in the system. To avoid this propagation, we suggest that we do not need to keep exact histogram values. We only need to keep approximate values in the histogram. A node only needs to send load information to other nodes when there is a significant change in either its load or the average load of a nonoverlapping group in its histogram.

To manage when a node should calculate and send updated information to other nodes, we define a configurable parameter θ ($\theta>1$) in our framework. A node only needs to send its updated load to its neighbor nodes if the ratio between the new load and the previous load, which was sent before, is greater than θ. When a node receives an update histogram message, it first updates its histogram. If there is any nonoverlapping group in its histogram, whose average load changes by a factor of θ compared to the previous average load, which caused update before, the node continues to update other neighbor nodes except the sender and other nodes, which may receive updated information from the sender node about the change. The effect of updating is now similar to a wave in that it propagates from the center of the region where a change happens to neighbor regions. It stops when the change reduces its effect (usually after only one or a few times of updating). As a result, only a small part of the system is updated. An example is shown in Fig. 2 in which a major change at node A may affect updating histograms at its neighbor nodes B, C, D, E, F, and G first. After that, the update process may continue at C_1 and C_2.

Clearly, if θ is set to a low value, it is still costly for updating histograms as the updating process may be triggered often. On the other hand, if θ is set with a high value, the load imbalance between nodes may be high because there may be a big difference between the real load value of a nonoverlapping group and its stored value in the histogram. As a result, depending on specific applications, an appropriate value of θ should be chosen. Since the ratio between values in the histogram and the real values can be as much as θ, the actual maximum imbalance ratio between load of a node and

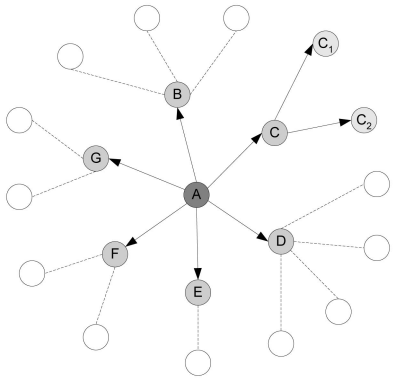

Fig. 2. Histogram update.

the average load of a group of nodes in its histogram is $k \cdot \theta$. Following Theorem 1, the maximum load imbalance ratio of the system is $(k \cdot \theta)^2$. It means that if we want to guarantee the maximum load imbalance ratio of the system is always less than an ϵ value, we can set $(k \cdot \theta)^2 < \epsilon$ or $\theta < \frac{\sqrt{\epsilon}}{k}$.

Additionally, to further reduce the cost of updating histogram, we also use the same technique as in the previous section: histogram information is piggy-backed with network messages. As a result, even though loads of nodes can be changed, update messages happen rarely.

3.2 The Load-Balancing Manager

In our framework, load balancing is done both statically when a new node joins the system or an existing node leaves the system and dynamically when an existing node in the system becomes overloaded or underloaded. In static load balancing, a new node needs to find a heavily loaded node to join as an adjacent node while an existing node wishing to leave the system needs to find a lightly loaded node to shed its workload. On the other hand, dynamic load balancing is realized by either *local* load balancing or *network* load balancing. In *local* load balancing, an overloaded or underloaded node performs load balancing with its adjacent nodes; while in *network* load balancing, an overloaded or underloaded node needs to find a lightly or heavily loaded node to do load balancing. In this case, the underloaded or lightly loaded node needs to leave its current position and joins as an adjacent node of the overloaded or heavily loaded node. In particular, a node only performs network load balancing if it cannot perform local load balancing. The rationale for this strategy is to minimize the overhead of network load balancing. In general, the load balancing algorithm aims to achieve global load balancing, which is defined as follows:

Definition 2. *The Peer-to-Peer system is globally load balanced if none of the nodes in the system is overloaded or underloaded.*

There are two issues in this algorithm: 1) how a node knows that it is overloaded or underloaded, and 2) how a lightly or heavily loaded node can be found in the system. The rest of this section introduces solutions for these issues based on histogram information.

Definition 3. *A node is overloaded if its load is greater than twice of the average load of any group in its histogram, whereas it is underloaded if its load is smaller than half of the average load of any group in its histogram.*

The above definitions imply that if the load of a node is either greater than two times or less than half of the average load of any group in its histogram, load balancing can be triggered to bring a better load balance for nodes ($k = 2$). To understand the rationale for this definition, let us assume that the load of a node x is w_x. We show that we can always do load balancing to make the system better when there is a group G in the histogram of x whose average load is either less than $\frac{w_x}{2}$ (x is an overloaded node) or greater than $2 \cdot w_x$ (x is an underloaded node). Let us consider three possible cases of G, where we assume that x is an overloaded node, and x can always shed its load to another node, i.e., x is not overloaded by a single large job (if x is an underloaded node, an opposite process can be executed).

Case 1. Group G contains only one node y. In this case, y checks the load of its two adjacent nodes. Let t be the total load of y and its adjacent nodes. If $t < 2 \cdot w_x$, y passes its load to its adjacent nodes so that each adjacent node has a load less than w_x. After that, y leaves its position and rejoins as an adjacent node of x. The system is now better balanced. On the other hand, if $t \geq 2 \cdot w_x$, y steals some load from its adjacent nodes so that both the load of y and its adjacent nodes are greater than $\frac{w_x}{2}$. As a result, x is not overloaded at all. Nevertheless, the system is also better balanced.

Case 2. Group G contains an even number of nodes. In this case, we divide all nodes in the group into pairs of adjacent nodes. Since the average load of the group is less than $\frac{w_x}{2}$, there must be a pair of adjacent nodes, whose total load is less than $\frac{w_x}{2}$. A node in this pair can pass its load to the remaining node, leave its position, and rejoin as an adjacent node of x to make the system more balanced.

Case 3. Group G contains an odd number of nodes greater than one. In this case, we divide all nodes in the group but one single node into pairs of adjacent nodes. Since the average load of the group is less than $\frac{w_x}{2}$, there must be either a pair of adjacent nodes, whose total load is less than $\frac{w_x}{2}$, or the load of the single node is less than $\frac{w_x}{2}$. The former case performs load balancing as in Case 2, while the later case performs load balancing as in Case 1. As a result, the system is better balanced.

Note that 1) in practice, usually if a node is overloaded or underloaded, it does not trigger load balancing immediately. Instead, it waits for a predefined period of time to recheck the condition again. If the node is still overloaded or underloaded, load balancing is actually triggered. 2) The reason why we choose $k = 2$ is because this value leads to the most efficient way to perform load balancing. If $k < 2$, the three above cases cannot be always applied. In this case, the system may need to perform load balancing by ripping data through adjacent nodes, which is not efficient.

From the way we construct histograms at nodes, the algorithm for finding a lightly or heavily loaded node is designed as follows: The node initializing the search process selects in its histogram the group of nodes that has the lightest or heaviest load and sends the search request to the neighbor node connected to that group. In addition to the search request, the node also specifies the *limited region*, which the receiver node can check for finding a lightly or heavily loaded node. The *limited region* is the search region falling between the receiver node and the following node in the routing table of the sender node. When a node x receives a request of finding a lightly or heavily loaded node, it checks its load and the average

loads of the groups of nodes, which fall in the *limited region*. If the load of x is the lightest or heaviest, the process stops; x is a lightly or heavily loaded node. Otherwise, x selects the lightest or heaviest group among groups in the *limited region* and continues to forward the search request to the neighbor node connected to that group.

Note that there are two reasons why the receiver node only needs to search a lightly or heavily loaded node in the *limited region*. First, it is unnecessary to make a comparison among all groups of nodes in the histogram since in the previous step, the sender node has decided that this limited region has the lightest or heaviest load compared to other places. Second, if we make a comparison among all groups, the search request may loop back to the sender node since histogram values of nodes may be inconsistent due to the fact that it takes sometime to propagate a change in load of nodes through the network when it happens.

Theorem 2. *The maximum number of steps for finding a lightly or heavily loaded node is bounded by the maximum number of steps needed for searching in the system.*

Proof. Since we form groups of nodes in the histogram as the way the search process limits its search regions by neighbor links, whenever a request of finding a lightly or heavily loaded node is sent to a neighbor node, it can be considered as a search request whose destination node is inside that region. In other words, when a node x wants to find a lightly or heavily loaded node, which is node y, the request is processed in a similar way as in the process of finding a value stored at node y and started at node x. In consequence, the maximum number of steps for finding a lightly or heavily loaded node is exactly the same as the maximum number of steps needed for processing a query in the system. □

Since the cost of searching in P2P systems is usually $O(logN)$, the expected cost of finding a lightly or heavily loaded node is also $O(logN)$. Once the lightly or heavily loaded node is found, it takes only one step to perform load balancing (the underloaded or lightly loaded node leaves its current position and joins as an adjacent node of the overloaded or heavily loaded node). As a result, the waiting time for performing load balancing is bounded at $O(logN)$.

The overall load-balancing algorithm, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2008.182, can be described as **State** describes possible states of a node; **Transition** describes when a node changes its state; **Procedure** describes in detail procedures used in load balancing process; and **Message** describes messages a node may receive and how they are processed. **P** is the current processing node. Note that this description does not include static load balancing, which happens at either the time a new node joins the system or the time an existing node leaves the system. For static load balancing, when a new node joins the system, it uses the procedure **FindHeavyNode** to find a heavily loaded node and joins next to that node to share a part of that node's workload. On the other hand, when an existing node leaves the system, it uses the procedure **FindLightNode** to find a lightly loaded node to shed its workload.

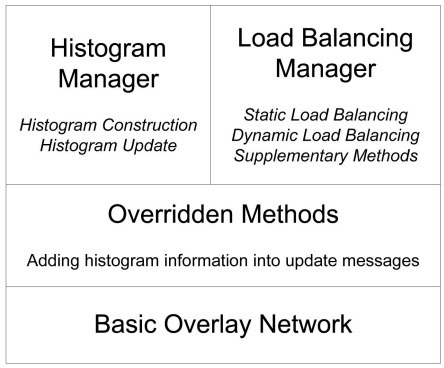

Fig. 3. The general framework.

3.3 The General Framework

To summarize, our general framework as described in Fig. 3 comprises three components built on top of the basic overlay network. The first component contains overridden methods, which extend the original methods for piggybacked histogram information, as discussed in Section 3.1.3. The other two components are the Histogram Manager, which is in charge of managing histogram information, and the Load Balancing Manager, which has a responsibility for load balancing among nodes in the system. These two components contain abstract methods to be implemented depending on specific systems.

4 Load-Balanced P2P Systems

In this section, we demonstrate how three well-known structured P2P Systems, which are very different in the overlay network, can be HiGLOB enabled to support global load balancing. These systems are Skip Graph [21], which employs a Skip List structure, BATON [15], which is based on a tree structure, and Chord [9], which utilizes a ring structure.

4.1 Skip Graph

Skip Graph [21] is based on the Skip List [39] structure, which is a multiple-sorted double-linked list. Unlike Skip List, where there is only one list at each level, a Skip Graph has many lists at a level. Each node in the system participates in a list at each level. The list to which a node belongs is controlled by a randomly assigned membership vector created when the node joins the system. In this way, a Skip Graph can be considered as a set of many skip lists, which share their lower levels. A Skip Graph structure is illustrated in Fig. 4.

Searching in Skip Graph is based on the same principle as searching in Skip List except a minor difference. Instead of sending a search query from a low-level node to a high-level node, in a Skip Graph, when a node issues a query, the search process always starts at the highest level of that node. At each step, if there is a neighbor node at the same level, which keeps a closer value to the search key, the node forwards the query to that neighbor node. Otherwise, the node continues the search process at a lower level. The destination node containing the result is found when the search process reaches the bottom level.

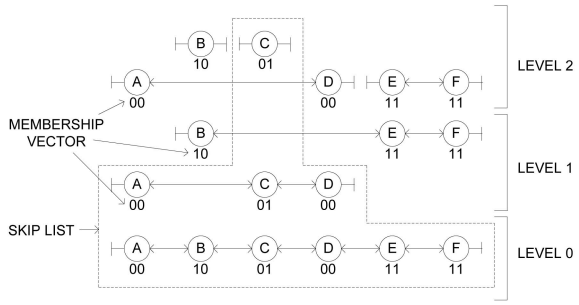

Fig. 4. A skip graph with three levels.

4.1.1 Histogram Structure

From the way a query is processed in Skip Graph, if a node x sends a query to a neighbor node y at level l, the search region is determined in one of the following ways depending on the scenarios:

1. In the first scenario, let z be a neighbor node of x at the nearest level higher than l on the same side (or direction) of x as y. Then, the search region is given by all nodes falling between y and z.
2. In the second scenario, node z as described in the first case does not exist. In this case, the search region consists of all nodes following y on the same side of x. As a result, a nonoverlapping group in a histogram in the Skip Graph can be defined as a group formed by all nodes falling between two consecutive neighbor nodes or all nodes following the farthest neighbor node of a node on the same side of that node.

As an example, consider Fig. 5. Assume that a query is sent from node D to node C at level 1, the search region is limited to all nodes between C and A since A is the next neighbor node of D at a higher level (which is level 2). As a result, the nonoverlapping group connected by node C in the histogram of node D includes two nodes C and B. On the other hand, if a query is sent from D to E at level 0, the search region is limited to all nodes following E on the same side of D since there is no other neighbor nodes of D at a higher level on the same side with E. Therefore, the nonoverlapping group connected by node E in the histogram includes two nodes E and F.

4.1.2 Histogram Construction

Based on the histogram structure described above, we now design the algorithms for constructing and maintaining histograms in Skip Graph.

As discussed in Section 3.1.2, a new node needs all of its neighbor nodes to send the necessary information to it for constructing the histogram. The question now is what information neighbor nodes should send to the new node? It is impossible for a neighbor node to send the load of every node in the nonoverlapping group connected by it because there is no way to get such information from a histogram. The neighbor node cannot send the average load of the nonoverlapping group either because it does not know which node is the next neighbor node of the new node at a higher level. Instead, a neighbor node x needs to send to the new node the summary of load and capacity of all nodes following x on the opposite side with the new

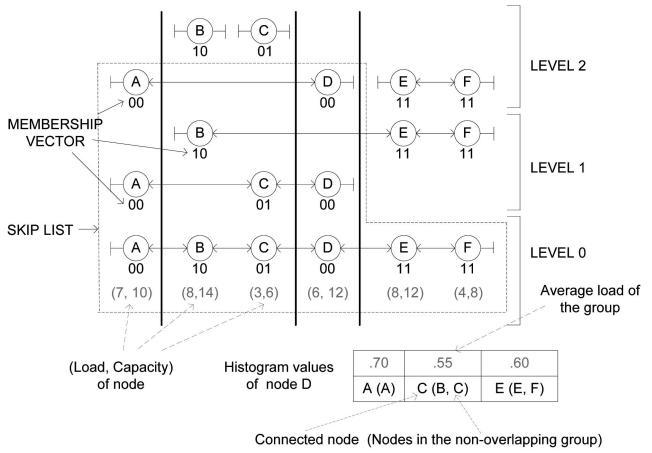

Fig. 5. Histogram structure in Skip Graph.

node together with the load and capacity of x itself. These values can be calculated from histogram values of nonoverlapping groups representing these nodes. Note that for the computation to be done, the histogram value of a nonoverlapping group must be represented by a pair of the total load and the total capacity of all nodes in that group. For example, as in Fig. 5, the stored value of a group of nodes connected through node C at D should be (11, 20) instead of the average load result 0.55. From now on, we always imply the average load as a pair of such information even though we only show the calculated average loads in the figures to simplify the presentation.

When a new node receives all the necessary information from its neighbor nodes, it can construct its histogram. The average load of a group connected by a node y is calculated from the received information from y and the next neighbor node z at the nearest higher level on the same side.

For example, assume that a new node G joins the network at a position between E and F, and it has a membership vector with prefix "01," as in Fig. 6. After joining, G has four different neighbor nodes: C, D, E, and F. As described above, C needs to send to G the average load of all nodes on the opposite side of G, which are A, B, and C. In other words, C needs to send to G the summary of load and capacity of A, B, and C, which is (18, 30)—the first parameter is the total load, while the second parameter is

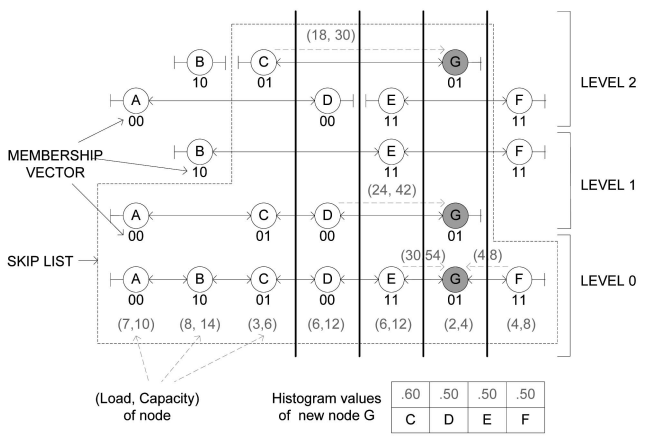

Fig. 6. Histogram construction for a new node in Skip Graph.

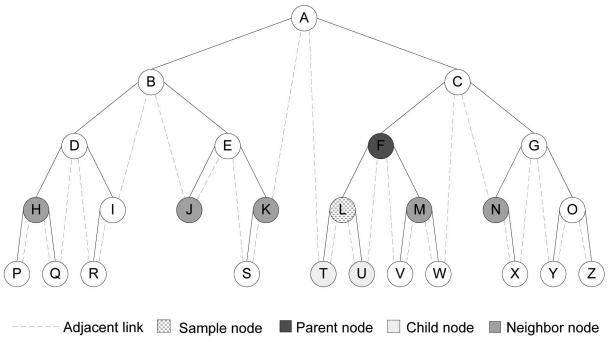

Fig. 7. BATON structure.

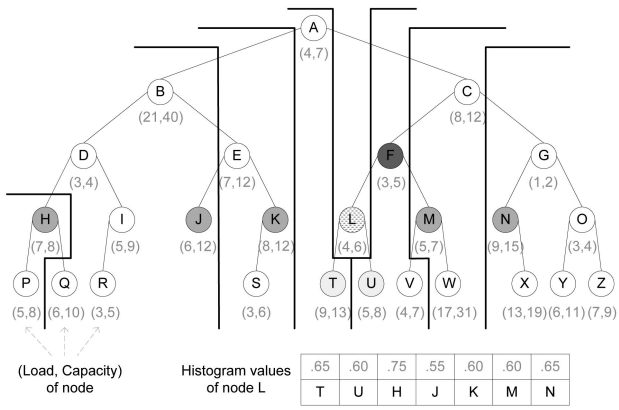

Fig. 8. Histogram structure in BATON.

the total capacity. Similarly, D, E, and F respectively send to G (24, 42), (30, 54), and (4, 8). Finally, from the received information, G can calculate the average load of these four nonoverlapping groups connected by C, D, E, and F to build its histogram.

4.1.3 Histogram Maintenance

When the load of a node changes by a factor θ, the node has to send updated information to all of its neighbor nodes. Similar to the information a node has to send to a new node, the node sends to every neighbor node on its left side the summary of load and capacity of all nodes on its right side, including its load and capacity, and vice versa. When a node receives updated information request from its neighbor x, it recalculates the histogram value associated with x by subtracting the received value with the summary value of all nodes following x on the same side of that node. After that, if the new average load value of the nonoverlapping group connected by x changes by a factor θ compared to the previous average load, which triggered histogram update before, the node continues to send updated information to every neighbor node in the opposite side, with x satisfying two conditions: 1) the neighbor node is not connected directly to x, and 2) the nonoverlapping group connected by the node at that neighbor node includes node x. These two conditions can be checked easily at the node.

For example, let us continue the previous example in Fig. 6 and assume that θ is set to 1.5. Now, if the load of E increases to 9, E has to send an updated information message to all of its neighbor nodes. In particular, E sends (33, 54) to G and F, and (15, 24) to D and B. When G, F, and B receive the updated information message, they recalculate the average load of the region connected by E and the process stops. However, when node D receives the updated information message, since C satisfies two conditions described above while the average load of the nonoverlapping group connected by E at D also changes 1.5 (this group contains only one node E), D needs to send updated information to C. As a result, D calculates and sends an updated information message (21, 36) to C. Finally, C recalculates the average load of the nonoverlapping region connected by D and the process stops. Note that in this example, we can also see the wave effect of the histogram update process in the system: a change of load at node E is propagated to G, F, D, and B, then continues to C from D.

4.2 BATON

BAlanced Tree Overlay Network (BATON) [15] is a structure based on a binary balanced tree in which each peer in the network maintains a node of the tree. A node has connections to other nodes by four different kinds of links: a parent link pointing to the parent node; child links pointing to child nodes; adjacent links pointing to adjacent nodes, which maintain adjacent ranges of values; and neighbor links pointing to selected neighbor nodes, which are nodes at the same level, having distances equal to a power of two from the node. In BATON, data stored at nodes is ordered increasingly from the left to the right of the tree. An example of BATON is shown in Fig. 7.

In BATON, when a node x processes a query, if the searched key does not fall into the range of values managed by x, x forwards the query to the farthest neighbor node in the routing table, which is nearer to the searched key. If there is no such neighbor node, x forwards the query to either a child (if it exists) or an adjacent node of x in the search direction. In particular, if x is a leaf node without a full routing table on the search direction, x always forwards the query to its parent node for processing.

4.2.1 Histogram Structure

A difficulty arises in defining nonoverlapping groups for a histogram in BATON. The search region, which is bounded by a query when it is sent from a node to its child, always includes the adjacent node of that node on the same search direction (the adjacent node is either a descendant node of the child node or the child node itself). As a result, there is no way to define nonoverlapping groups connected by both child link and adjacent link of a node at the same side. Furthermore, if a node has full routing tables, its parent link is never used in query processing, and hence, there should be no group connected by the parent link. It means that we cannot strictly follow the rule of the general framework, which requires every link to be associated with a nonoverlapping group to form a bucket in the histogram structure. Instead, we suggest that a node only uses a subset of links to build its histogram given that the union of all nonoverlapping groups connected by links in this subset still covers the whole system. In particular, we only create nonoverlapping groups connected by child links and neighbor links, as in Fig. 8, with two exceptions. First, if a node does not have a child, the nonoverlapping group,

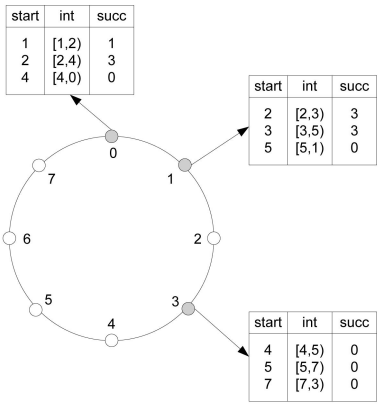

Fig. 9. A Chord ring.

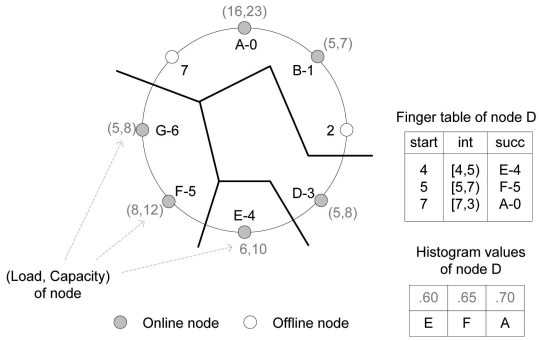

Fig. 10. Histogram structure in Chord.

which contains nodes falling between the node and the first neighbor node on the direction of the missing child, is connected by the adjacent node on that direction.

For example, node J in Fig. 8 has a histogram with nonoverlapping groups connected by nodes B, E, H, I, K, L, and N (B and E are adjacent nodes of J). The values are respectively 0.53, 0.56, 0.75, 0.61, 0.66, 0.61, and 0.65. Second, if a node does not have full routing table, it always asks its parent to find a lightly or heavily loaded node as in the search algorithm, and hence, it does not need to keep exact histogram values for this node.

4.2.2 Histogram Construction and Maintenance

The way histogram is constructed and maintained in BATON is similar to that of Skip Graph. When a new node joins the system, all of its neighbor nodes have to calculate and send the summary of load and capacity of themselves and all nodes following their position on the same side of the new node from their histogram values. However, in BATON, the histogram values are not recalculated right away if the new node does not have full routing tables. These values are kept until the node has full routing tables. At that time, histogram values are recalculated. The histogram values of nonoverlapping groups, which contain nodes falling between the node and the first neighbor node, are calculated with additional information from histogram values of the parent node. When the load of a node is changed, it sends updated information, which is calculated in the same way as in the process of histogram construction, to all of its neighbor nodes. The following update propagation is also similar to that of Skip Graph.

4.3 CHORD

Chord [9] is built on a structure in which nodes are ordered to form a ring—an identifier circle modulo 2^m according to their identifier. A data item with identifier d is assigned to the first node x whose identifier is equal to or follows d in the identifier space (x is called successor of d). For routing purposes, each node n in the network maintains an m-entry key routing table called finger table in which the ith entry contains the identifier of the first node s that succeeds n by at least 2^{i-1} on the identifier circle. In other words, s is the successor of $(n + 2^{i-1})$. Besides the finger table, each node also maintains a link to its predecessor node, which is the immediate node before the node in the Chord ring. Fig. 9 shows the structure of an eight-node Chord ring with three online nodes.

Since a data item in Chord is stored at its successor node, the target of the search algorithm is to find such a node. As a result, when a node n issues or receives a query request q, if n is not the successor node x of the query value v, it has to send q to x. However, if n does not know x, it has to send q to the immediate predecessor y of v first since the successor of the immediate predecessor of v is exactly the successor v, which is x. After that, y can continue to forward q to x. Nevertheless, if n does not know y either, it has to look into its finger table to find a node closest to and before v in the identifier space and forwards q to that node. By repeating this process, q is forwarded to y and then to x for processing.

It is important to note that since the original version of Chord uses consistent hashing that distributes data uniformly, it cannot support range queries. To avoid this problem, many variants of Chord such as [40] have been proposed. These extended versions support range queries by using order-preserving hash functions. This way, however, invalidates some properties of Chord and cannot guarantee uniform data distribution.

4.3.1 Histogram Structure

In the search algorithm described above, whenever a node sends a query to a neighbor node in its finger table, the search region is limited to that node and the following node of that node in the finger table since that node is a predecessor of the searched key while the following node is a successor of the searched key. In case the node receiving the query is the last node in the finger table, the search region is limited to that node and the immediate predecessor of the current node. Therefore, the non-overlapping groups in a histogram are formed in the same way and are described in Fig. 10.

4.3.2 Histogram Construction

Among the three systems, Chord is the most difficult system to construct and maintain a global histogram. Unlike Skip Graph and Baton where neighbor nodes of a new node always share the end horizon that is either the minimum or maximum value, Chord has no end because of the circle structure. As a result, we cannot construct a histogram for a new node using the same technique of Skip Graph and Baton. Alternatively, realizing that in the search algorithm, at each step, the sender node can calculate the average load of the region between it and the receiver from its histogram,

we propose that the average load of a nonoverlapping group connected by a neighbor of a new node can be calculated by aggregating the load values during the search process. In other words, the average load of the nonoverlapping group connected by the first node in the finger table is calculated on the way a query is sent from the first node to the second node. Similarly, the average load of the second nonoverlapping group is calculated on the way a query is sent from the second node to the third node and so on. Finally, the average load of the last nonoverlapping group is calculated from the load information of the whole system, which can be calculated from a histogram of any existing node, and loads of other nonoverlapping groups, which are calculated before. Note that, in this way, the algorithm for constructing the finger table of a new node has to be modified if we do not want to separate the process of constructing the finger table and the process of constructing histogram. Basically, a neighbor node at position $i+1$ in the finger table is always looked up from a neighbor node at position i instead of from the predecessor node, as in the original method. Nevertheless, this modification does not affect the cost of constructing the finger table.

4.3.3 Histogram Maintenance

Histogram updating in Chord is also different from other systems. When the load of a node changes by a factor of θ, the node does not send an updated information message to its neighbor nodes since neighbor nodes are not nodes having links to it. Instead, the node sends the update message to $logN$ nodes, which are nodes preceding its 2^i positions in the Chord ring (if there is a position without an available node, the nearest node preceding that position is sent the message). If a node x receives an update request targeting at it, x checks its routing table to see if the sender is in or not. If the sender node is a neighbor of x, x recalculates its histogram and continues to send an update message to other nodes if necessary. An issue with this process is that it is expensive since a node needs approximately $logN$ messages for every update operation. To deal with it, we have two solutions. The first solution is to use a variant of Chord, which supports bidirectional links such as that in [41], and hence, update messages can be sent directly to neighbor nodes and are processed as in Skip Graph and BATON. The second solution is to change the histogram update process as follows: When the load of a node changes by a factor of θ, the node sends updated information to only its predecessor node. When a node receives an update message from its immediate successor node, it recalculates its histogram and continues to update its predecessor node if there is any signification change in nonoverlapping groups. Even though the second solution cannot tighten the ratio between the real average load of a group and the stored value in a histogram within θ with high probability, this ratio is still in the range. It is because a load change of a node x often has a significant impact to only a few preceding nodes of x. For far away nodes, nonoverlapping groups containing x usually are big groups, and hence, a load change of a node does not always have significant impact. Nevertheless, if many nodes in the same group change their load, the updating process should also be propagated long enough for updating histograms of these far away nodes.

5 Experimental Study

To evaluate the performance of our proposal, we built a simulator for our framework on which we implemented the three structured P2P systems: Skip Graph [21], BATON [15], and Chord [9]. For Chord, we implemented the second solution for histogram update since we want to employ our system on the original Chord structure. As noted in Section 4.3, Chord has two main versions, and since the extended version has been widely used for supporting range queries, we shall evaluate the performance of such extension as well. We call the extended version, Chord^*, to distinguish it from Chord.

The simulator is used to test the storage load distribution on large-scale networks from 1,000 to 10,000 nodes over PlanetLab [42], a testbed for large-scale distributed systems. For each network size N, we insert a data set of $100 \cdot N$ data items each of which has the same size. In this way, the work load of a node is simply the number of data item stored at that node. We tested the system in three different network types:

1. *Type-I* is a homogeneous system with skewed data distribution,
2. *Type-II* is a heterogeneous system with uniform data distribution, and
3. *Type-III* is a heterogeneous system with skewed data distribution.

Note that since the original version of Chord uses consistent hashing, which distributes data uniformly, this version can only be used to test with Type-II network.

To generate heterogeneous nodes, we used a real data set from that in [16], which measures the amount of shared data at nodes in the Gnutella system (we assume that the storage capability of nodes is proportional to the amount of data they share or store in the data set). To generate skewed data distribution, we used the Zipfian method with parameter 1.0. The default values of the network size and θ are 10,000 and 2.0.

5.1 Load Balancing

We first study the effect of our load-balancing method on the three different network types. We use the following simple load balancing method for comparison: a node contacts all nodes in its routing table to find a lightly or heavily loaded node to balance its load. The results of the load distribution of nodes of both methods are shown in Figs. 11a, 11b, and 11c. In these figures, the x-axis displays the percentage of nodes, while the y-axis displays the percentage of load stored by the corresponding percentage of nodes. The *ideal* line represents the capacity distribution of nodes in the systems. Note that in Fig. 11a, nodes are ordered decreasingly by their load, while in Figs. 11b and 11c, nodes are ordered decreasingly by their capacity. The results confirm that in all network types, the HiGLOB-enabled system outperforms its simple counterpart, i.e., our methods have a much better load balance than the simple method. A better result is also reflected in the maximum

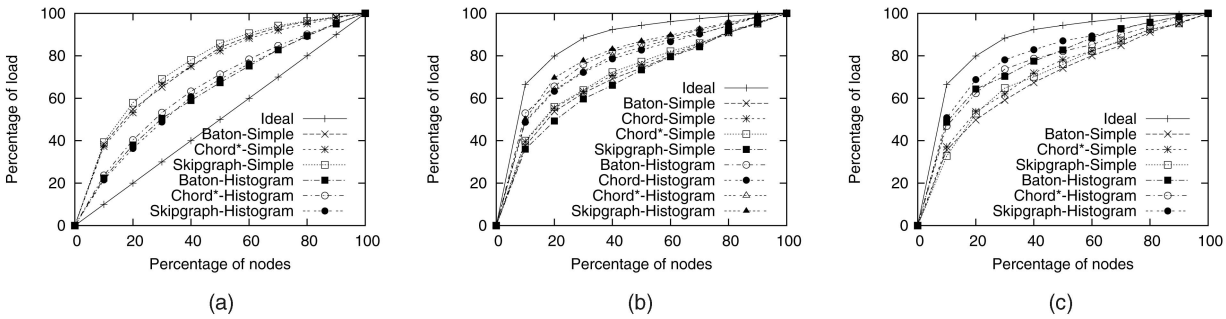

Fig. 11. Load distribution of nodes. (a) Type-I network (homogeneous nodes, skewed data distribution). (b) Type-I network (homogeneous nodes, skewed data distribution). (c) Type-III network (heterogeneous nodes, skewed data distribution).

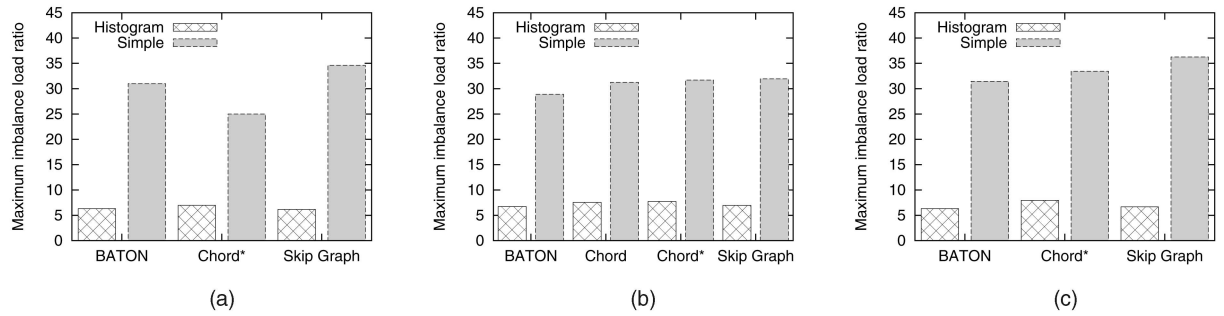

Fig. 12. Maximum load imbalance ratio of nodes. (a) Type-I network (homogeneous nodes, skewed data distribution). (b) Type-I network (homogeneous nodes, skewed data distribution). (c) Type-III network (heterogeneous nodes, skewed data distribution).

load imbalance ratio of nodes in the systems, as displayed in Fig. 12a, 12b, and 12c. In the HiGLOB-enabled methods, this maximum load imbalance ratio value varies from 6 to 8, while in the simple method, the value varies from 25 to 36. We further look into the details of the change of this value along the time line of the experiment of Type-II network in Fig. 13. The result shows that this maximum load imbalance ratio value of the simple method increases significantly along the time line, while this value in our systems only increases slightly in the first half of the experiment and does not change much in the second half of the experiment. The increase of this value along the time line is caused simply by having more loads in the system (in the experiments data, items are inserted to the system gradually). However, note that this value does not increase in our systems after sometime, while it still increases in the simple method. Furthermore, as expected, the maximum load imbalance ratio value of in our systems is always controlled within a boundary—less than $(2 \cdot \theta)^2 = (2 \cdot 2)^2 = 16$.

5.2 Cost of Load Balancing

We next study the overhead of realizing global load balance. We compare our HiGLOB-enabled method with three other methods as follows:

1. In the first method, a node selects a lightly or heavily loaded node when needed by randomly asking $logN$ nodes (they are not nodes in the routing table as in the simple load-balancing scheme). By randomly picking nodes in the system, the method improves the chance of finding a globally underloaded or overloaded node, and hence, load balancing is better than that of the simple load balancing scheme. However, the cost is more expensive since each look-up node requires $logN$ efforts.
2. In the second method, we use a separate Skip Graph for keeping load distribution of nodes, as in [20].
3. In the third method, we also use histograms for keeping load distribution of nodes. However, in this method, nodes construct and maintain histograms by sampling neighbor nodes and exchanging these sample loads results with randomly selected far-away nodes, as in [30].

We configure parameters so that the expected maximum load imbalance ratio is approximately equal in all the three systems, and hence, the load distribution is also approximately equal with high probability. In particular, the maximum load imbalance ratio of these three systems in this experiment is displayed in Fig. 14, while the cost of load balancing of these three systems (in terms of the average number of extra messages required at each node in the whole experiment) is displayed in Fig. 15. The results show that our HiGLOB enabled systems have a much lower

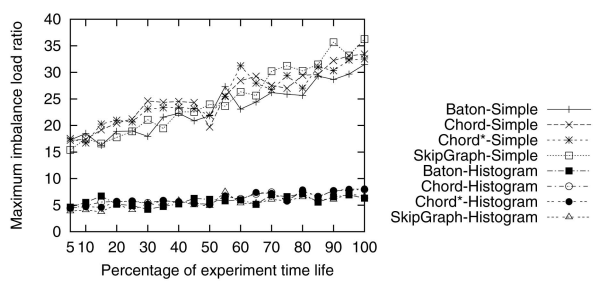

Fig. 13. The change of the maximum load imbalance ratio value along the experiment time line in Type-II network (heterogeneous nodes, uniform data distribution).

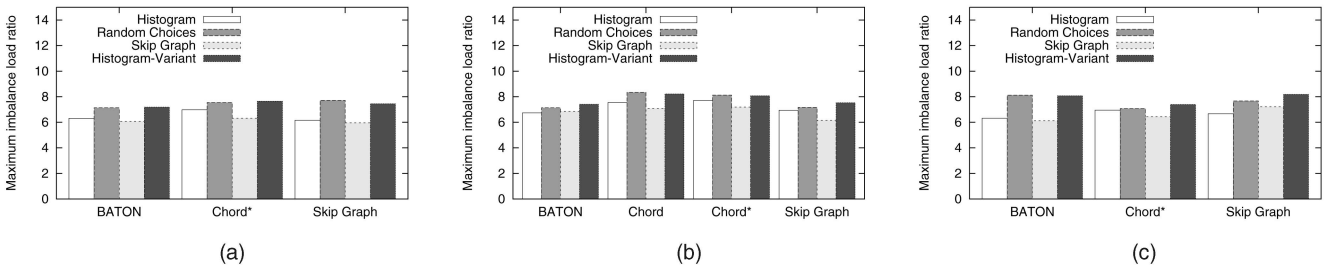

Fig. 14. Maximum load imbalance ratio of nodes. (a) Type-I network (homogeneous nodes, skewed data distribution). (b) Type-II network (heterogeneous nodes, uniform data distribution). (c) Type-III network (heterogeneous nodes, skewed data distribution).

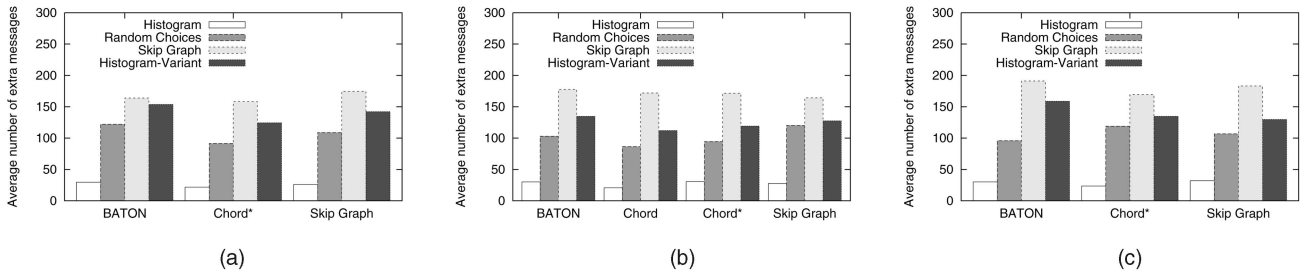

Fig. 15. Cost of load balancing. (a) Type-I network (homogeneous nodes, skewed data distribution). (b) Type-II network (heterogeneous nodes, uniform data distribution). (c) Type-III network (heterogeneous nodes, skewed data distribution).

overhead in comparison to the other two methods: approximately 30 extra messages at each node. We note that in the random choices and histogram-variant methods, there are some points in the experiment timeline when the maximum load imbalance ratio still goes beyond 20. These values we put in the graphs are just values measured at the end of experiments after the system becomes stable.

5.3 Effect of Varying θ Value

In this experiment, we investigate the effect of varying θ value on the cost of load balancing as well as the maximum load imbalance ratio of nodes in the system while keeping the network size unchanged. Since the effect is similar in different network types, we only show the results of Type-II network in Fig. 16. The figure displays the average number of extra messages required at each node for load balancing. The result shows that when θ is increased, the load balancing cost is reduced. It is because it takes longer time for a histogram update request to be triggered. On the other hand, when θ is increased, the difference between the real load of a region and the load of that region stored in a histogram is higher. As a result, the maximum load imbalance ratio of nodes in the system is increased too (see Fig. 16b).

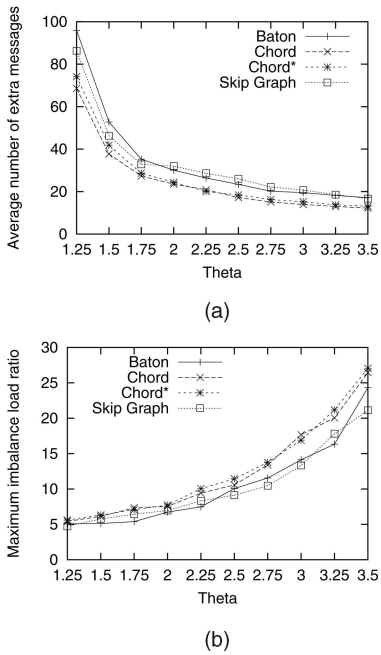

Fig. 16. Effect of varying θ value in Type-II network (heterogeneous nodes, uniform data distribution). (a) Cost of load balancing. (b) Maximum load imbalance ratio.

It is not surprising that Chord and Chord* produce similar results since their structures are identical. However, compared to BATON and Skip Graph, the results are different. This is because these systems have different histogram structures and different strategies for constructing and maintaining histograms, as presented in previous sections. In particular, in terms of the load balancing cost, Chord versions have the lowest cost compared to BATON and Skip Graph. This is because among three structures, Chord has the smallest average histogram size due to the smallest routing table size. The maximum number of different neighbor nodes in a routing table in Chord is just $logN$ while that in BATON and Skip Graph can be up to $2 \cdot logN$. On the other hand, since the size of histogram in BATON and Skip Graph are approximately equal, the load balancing cost of BATON and Skip Graph are indistinguishable. In terms of the maximum load imbalance ratio, the result of Chord versions is the worst, while those of BATON and Skip Graph are better. However, this result is not mainly caused by the smaller histogram size in Chord. Instead, the main reason comes from the delay of the update strategy in Chord since the histogram update of a node is only forwarded to the predecessor node step by

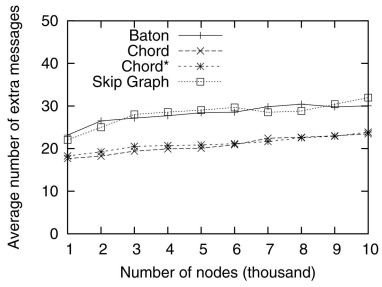

Fig. 17. Effect of varying network size in Type-II network (heterogeneous nodes, uniform data distribution).

step. Therefore, it takes longer time to update histogram values of a nonoverlapping group from a far away node. Nevertheless, the difference is not much. The results of maximum load imbalance ratio of BATON and Skip Graph are again indistinguishable since these two systems apply the similar strategy in constructing and maintaining histograms.

5.4 Effect of Varying Network Size

Finally, we study the effect of varying network size on the load-balancing cost while keeping θ unchanged. Fig. 17 shows the load balancing cost of Type-II network in terms of the average number of extra messages required at each node, as in previous graphs. The result shows that the cost increases logarithmically with the network size. This follows from the fact that histogram update requests are propagated via neighbor nodes, whose total number depends on the network size $(logN)$.

6 Conclusion

In this paper, we proposed a framework, HiGLOB, to enable global load balance for structured P2P systems. Each node in HiGLOB maintains the load information of nodes in the systems using histograms. This enables the system to have a global view of the load distribution and hence facilitates global load balancing. We partition the system into nonoverlapping groups of nodes and maintain the average load of them in the histogram at a node. We also proposed two techniques to reduce the overhead of maintaining and constructing histograms. Even though the proposal is a general framework, it is possible to deploy different kinds of P2P systems on it. We demonstrated this by building three well-known structured P2P systems: Skip Graph, BATON, and Chord on our proposal. Our performance evaluation shows that our HiGLOB enabled systems are superior over other methods.

Acknowledgments

Dr. Vu and Dr. Tan are supported in part by ASTAR SERC Grant 072 101 0017 as part of the S3 project [1].

References

[1] S3: Scalable, Shareable and Secure P2P Based Data Management System, http://www.comp.nus.edu.sg/~s3p2p, 2008.
[2] Gnutella, http://www.gnutella.com/, 2008.
[3] BitTorrent, http://www.bittorrent.com/, 2008.
[4] Overnet, http://www.overnet.com, 2008.
[5] SETI@home, http://setiathome.berkeley.edu/, 2008.
[6] Groove, http://www.groove.net, 2008.
[7] Skype, http://www.skype.com/, 2008.
[8] A. Madhukar and C. Williamson, "A Longitudinal Study of P2P Traffic Classification," *Proc. Int'l Symp. Modeling, Analysis, and Simulation of Computer and Telecomm. Systems (MASCOTS),* 2006.
[9] D. Karger, F. Kaashoek, I. Stoica, R. Morris, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *Proc. SIGCOMM '01,* pp. 149-160, 2001.
[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content Addressable Network," *Proc. SIGCOMM '01,* pp. 161-172, 2001.
[11] B.Y. Zhao, J.D. Kubiatowicz, and A.D. Joseph, "Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing," Technical Report CSD-01-1141, Univ. of California, Apr. 2001.
[12] A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems," *Proc. 18th IFIP/ACM Int'l Conf. Distributed Systems Platforms (Middleware '01),* pp. 329-350, 2001.
[13] K. Aberer, "P-Grid: A Self-Organizing Access Structure for P2P Information Systems," *Proc. Int'l Conf. Cooperative Information Systems (CoopIS),* 2001.
[14] R. Huebsch, B. Chun, J.M. Hellerstein, B.T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A.R. Yumerefendi, "The Architecture of PIER: An Internet-Scale Query Processor," *Proc. Conf. Innovative Data Systems Research (CIDR),* 2005.
[15] H.V. Jagadish, B.C. Ooi, and Q.H. Vu, "Baton: A Balanced Tree Structure for Peer-to-Peer Networks," *Proc. Very Large Databases Conf. (VLDB '05),* pp. 661-672, 2005.
[16] S. Saroiu, P.K. Gummadi, and S.D. Gribble, "A Measurement Study of Peer-to-Peer File Sharing Systems," *Proc. Multimedia Computing and Networking Conf. (MMCN),* 2002.
[17] M. Mitzenmacher, "The Power of Two Choices in Randomized Load Balancing," *IEEE Trans. Parallel and Distributed System,* vol. 12, no. 10, pp. 1094-1104, Oct. 2001.
[18] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Structured P2P Systems," *Proc. Int'l Workshop Peer-to-Peer Systems (IPTPS),* 2003.
[19] D. Karger and M. Ruhl, "Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems," *Proc. ACM Symp. Parallelism in Algorithms and Architectures (SPAA),* 2004.
[20] P. Ganesan, M. Bawa, and H. Garcia-Molina, "Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems," *Proc. Very Large Databases Conf. (VLDB '04),* pp. 444-455, 2004.
[21] J. Aspnes and G. Shah, "Skip Graphs," *Proc. 14th Ann. ACM-SIAM Symp. Discrete Algorithms (SODA '03),* pp. 384-393, 2003.
[22] D. Karger and M. Ruhl, "Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems," *Proc. Int'l Workshop Peer-to-Peer Systems (IPTPS),* 2004.
[23] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Dynamic Structured P2P Systems," *Proc. INFOCOM,* 2004.
[24] K. Kenthapadi and G.S. Manku, "Decentralized Algorithms Using Both Local and Random Probes for P2P Load Balancing," *Proc. ACM Symp. Parallelism in Algorithms and Architectures (SPAA),* 2005.
[25] G. Giakkoupis and V. Hadzilacos, "A Scheme for Load Balancing in Heterogenous Distributed Hash Tables," *Proc. ACM Symp. Principles of Distributed Computing Conf. (PODC),* 2005.
[26] J. Ledlie and M. Seltzer, "Distributed, Secure Load Balancing with Skew, Heterogeneity, and Churn," *Proc. INFOCOM,* 2005.
[27] S. Surana, B. Godfrey, K. Lakshminarayanan, R. Karp, and I. Stoica, "Load Balancing in Dynamic Structured P2P Systems," *Performance Evaluation,* vol. 63, no. 6, pp. 217-240, 2006.
[28] M. Adler, E. Halperin, R.M. Karp, and V.V. Vazirani, "A Stochastic Process on the Hypercube with Applications to Peer-to-Peer Networks," *Proc. 35th ACM Symp. Theory of Computing (STOC '03),* pp. 575-584, 2003.
[29] K. Aberer, A. Datta, and M. Hauswirth, "Multifaceted Simultaneous Load Balancing in DHT-Based P2P Systems: A New Game with Old Balls and Bins," *Self-* Properties in Complex Information Systems,* 2005.

[30] A.R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: Supporting Scalable Multi-Attribute Range Queries," *Proc. ACM SIGCOMM '04,* pp. 353-366, 2004.
[31] P.B. Godfrey and I. Stoica, "Heterogeneity and Load Balance in Distributed Hash Tables," *Proc. IEEE INFOCOM,* 2005.
[32] M. Bienkowski, M. Korzeniowski, and F.M. Heide, "Dynamic Load Balancing in Distributed Hash Tables," *Proc. Int'l Workshop Peer-to-Peer Systems (IPTPS),* 2005.
[33] J. Byers, J. Considine, and M. Mitzenmacher, "Simple Load Balancing for Distributed Hash Tables," *Proc. Int'l Workshop Peer-to-Peer Systems (IPTPS),* 2003.
[34] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System," *LNCS,* vol. 2009, pp. 46-66, July 2001.
[35] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-Area Cooperative Storage with CFS," *Proc. 18th ACM Symp. Operating Systems Principles (SOSP),* 2001.
[36] M. Roussopoulos and M. Baker, "CUP: Controlled Update Propagation in Peer-to-Peer Networks," *Proc. USENIX Ann. Technical Conf.,* 2003.
[37] V. Ramasubramanian and E.G. Sirer, "Beehive: Exploiting Power Law Query Distributions for O(1) Lookup Performance in Peer to Peer Overlays," *Proc. First USENIX Symp. Networked Systems Design and Implementation (NSDI '04),* pp. 331-342, 2004.
[38] L. Yin and G. Cao, "DUP: Dynamic-Tree Based Update Propagation in Peer-to-Peer Networks," *Proc. 21st Int'l Conf. Data Eng. (ICDE),* 2005.
[39] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Comm. ACM,* vol. 32, no. 10, pp. 668-676, 1990.
[40] M. Abdallah and H.C. Le, "Scalable Range Query Processing for Large-Scale Distributed Database Applications," *Proc. Int'l Conf. Parallel and Distributed Computing Systems (PDCS),* 2005.
[41] J.J. Jiang, F.L. Tang, F. Pan, and W.N. Wang, "Using Bidirectional Links to Improve Peer-to-Peer Lookup Performance," *J. Zhejiang Univ. SCIENCE A,* vol. 7, no. 6, 2006.
[42] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: An Overlay Testbed for Broad-Coverage Services," *ACM SIGCOMM Computer Comm. Rev.,* vol. 33, no. 3, 2003.

Quang Hieu Vu received the BEng degree in computer science from HCMC University of Technology, Vietnam, in 2001 and the PhD degree in computer science from Singapore-MIT Alliance (SMA) in 2008. He is currently a research fellow in the School of Computing, National University of Singapore. He was a lecturer at the HCMC University of Technology and a software engineer at TTNK software company for three years and a half. His research interest includes query processing in distributed systems especially P2P systems. He is a member of the ACM and the IEEE.

Beng Chin Ooi received the BSc (First Class Honors) and PhD degrees from Monash University, Australia, in 1985 and 1989, respectively. He is a professor of computer science at the School of Computing, National University of Singapore. His research interests include database performance issues, indexing techniques, XML, P2P/parallel/distributed computing, embedded system, Internet, and genomic applications. He has served as a PC member for a number of international conferences (including SIGMOD, VLDB, ICDE, WWW, EDBT, DASFAA, GIS, KDD, CIKM, and SSD). He is an editor of *GeoInformatica*, the *GIS Journal*, *ACM SIGMOD Disc*, and *VLDB Journal* and a guest editor for the special issue (section) of *IEEE TKDE* on P2P-based data management. He is the recently elected editor-in-chief of *IEEE TKDE*. He is a cofounder and director of GeoFoto Pte., a company providing imaging solutions, and BestPeer (on the way), a company specializing in P2P computing technology.

Martin Rinard received the BSc degree (*magna cum laude* and with honors) in computer science from Brown University in 1984 and the PhD degree in computer science from Stanford University in 1994. He was with two start-up companies, Ikan Systems and Polygen Corp. He was with the Computer Science Department, University of California, Santa Barbara, in 1994, as an assistant professor. He is currently with the Massachusetts Institute of Technology, as an assistant professor in 1997 and then promoted to associate professor and full professor in 2000 and 2006, respectively.

Kian-Lee Tan received the BSc (*Hons.*) and PhD degrees in computer science from the National University of Singapore in 1989 and 1994, respectively. He is currently a professor in the Department of Computer Science, National University of Singapore. His major research interests include query processing and optimization, database security, and database performance. He has published more than 200 conference/journal papers in international conference proceedings and journals. He is also the coauthor of three books. He is a member of the ACM.