Resilient and efficient load balancing in distributed hash tables

Di Wu *, Ye Tian, Kam-Wing Ng

Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong

Abstract

As a fundamental problem in distributed hash table (DHT)-based systems, load balancing is important to avoid performance degradation and guarantee system fairness. Among existing migration-based load balancing strategies, there are two main categories: (1) rendezvous directory strategy (RDS) and (2) independent searching strategy (ISS). However, none of them can achieve resilience and efficiency at the same time. In this paper, we propose a group multicast strategy (GMS) for load balancing in DHT systems, which attempts to achieve the benefits of both RDS and ISS. GMS does not rely on a few static rendezvous directories to perform load balancing. Instead, load information is disseminated within the formed groups via a multicast protocol. Thus, each peer has enough information to act as the rendezvous directory and perform load balancing within its group. Besides intra-group load balancing, inter-group load balancing and emergent load balancing are also supported by GMS. In GMS, the position of the rendezvous directory is randomized in each round, which further improves system resilience. In order to have a better understanding of GMS, we also perform analytical studies on GMS in terms of its scalability and efficiency under churn. Finally, the effectiveness of GMS is evaluated by extensive simulation under different workload and churn levels.

1. Introduction

In the past few years, various distributed hash table (DHT) systems, such as Chord (Stoica et al., 2001), CAN (Ratnasamy et al., 2001), Pastry (Rowstron and Druschel, 2001), Tapestry (Zhao et al., 2001), and others, have been studied and applied in various applications. These systems need to provide scalable, fault-tolerant, and efficient services across a wide range of applications. However, load balancing is a fundamental problem in these systems, which can affect system performance and fairness. Load balancing strategies can be broadly divided into two main categories: migration-based and non-migration-based approaches. In this paper, we focus on migration-based load balancing strategies, which migrate data objects from overloaded to underloaded nodes to improve system performance and fairness.

Keywords:
Distributed system
Distributed hash table
Load balancing

* Corresponding author.
E-mail addresses: dwu@cse.cuhk.edu.hk (D. Wu), ytian@cse.cuhk.edu.hk (Y. Tian), kwng@cse.cuhk.edu.hk (K.-W. Ng).

1084-8045/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
DHT is now emerging as a promising approach to build efficient infrastructures for large-scale distributed applications, such as grid information services (Cai et al., 2003), Peer-to-Peer file-sharing systems (Emule project, 2006), domain naming services (Ramasubramanian and Sirer, 2004), storage systems (Kubiatowicz et al., 2000), content distribution networks (Coral Content Distribution Network, 2006), etc.

As a fundamental issue, load balancing is of great significance to the performance and fairness of DHT systems. Previous work has pointed out that, the uniformity of hash functions cannot guarantee the perfect balance between nodes, and there exists an $O(\log N)$ imbalance factor (Stoica et al., 2001) in the number of objects stored at a node. What is more, due to the node heterogeneity or application semantics, it is possible for some nodes to be heavily loaded, while others are lightly loaded. Such unfair load distribution among peers will cause performance degradation, and also provide disincentives to participating peers.

The importance of load balancing has motivated a number of proposals, e.g., Byers et al. (2003), Naor and Wieder (2003), Rao et al. (2003), Godfrey et al. (2004) and Zhu and Hu (2005), etc. Many of them are based on the concept of “virtual server (VS)” (Stoica et al., 2001). By shedding an appropriate number of VSs from heavy nodes to light nodes, load balancing can be achieved. Based on the difference in load information management and decision making of load balancing, existing migration-based approaches can be categorized into two representative strategies: (1) rendezvous directory strategy (RDS), which relies on centralized rendezvous directories to gather load information and schedule load reassignment and (2) independent searching strategy (ISS), in which nodes should search independently for other nodes with inverse load characteristics to perform load transfer.

Due to information centralization, RDS can conduct the best load reassignment and be much more efficient than ISS. Most of the existing RDS-based schemes (e.g., Godfrey et al., 2004; Zhu and Hu, 2005) only focus on the scalability problem of RDS, while paying little attention to the resilience issue. In these approaches, the positions of rendezvous directories are static and known publicly to all the peers, thus making RDS vulnerable to node-targeted attacks. In case that the rendezvous directory is occupied by malicious peers or overwhelmed by message traffic, the service of load balancing is halted. The performance degradation or failure of rendezvous directories also impacts the service of load balancing greatly. On the contrary, ISS is more resilient to the failure and attacks as there exists no central entity in the system. But its efficiency greatly depends on the searching scope. A wide scope often incurs huge traffic and becomes unscalable in a large system. In case of a narrow scope, it is inefficient to achieve system-wide load balance. Both RDS and ISS cannot achieve resilience and efficiency at the same time.

In this paper, we propose a simple yet efficient load balancing strategy for DHT systems, called group multicast strategy (GMS), which is scalable and achieves both the efficiency of RDS and the resilience of ISS. In GMS, the whole system is formed into groups, and the multicast protocol is used for load information dissemination. After load information dissemination, every group member has full load information of its own group and the utilization information of the whole system. Thus, each group member can act as the rendezvous directory to perform load reassignment within the group, but the position of the rendezvous directory is randomized to make GMS resistant to node-targeted attacks. Besides intra-group load balancing, inter-group load balancing and emergent load balancing are also allowed in GMS to achieve a more balanced system state.

In summary, we make the following contributions in this paper:

1. Our proposed GMS is a migration-based load balancing strategy for DHT systems, which to the best of our knowledge is the first to achieve both resilience and efficiency. By utilizing multicast-based information dissemination and randomizing the positions of the rendezvous directory, GMS can retain the efficiency of RDS, while being more resilient to the node-targeted attacks.

2. We perform an analytical study on the scalability of GMS, and the impact of system churn on the efficiency of GMS to get a better understanding. It is found that GMS can exhibit good scalability.

---

1 In this paper, we use the term “peer”and “node” interchangeably.
and the impact of churn could be minimized when the tuning parameters are properly configured.

3) We evaluate the performance of GMS via extensive simulations. Our simulation results show that GMS can achieve comparable efficiency to RDS under different scenarios. But different from RDS, our proposed GMS is also resilient and scalable.

The remainder of this paper is structured as follows. We first introduce the related work in Section 2. The detailed design of GMS is presented in Section 3. In Section 4, we analyze the scalability of GMS and the impact of churn to GMS. In Section 5, we describe the simulation methodologies and results related to the experiments conducted to verify the performance of GMS. Finally, Section 6 summarizes the whole paper.

2. Related work

The topic of load balancing has been well studied in the field of distributed systems, while the characteristics of DHT systems (e.g., node dynamics, decentralization, large system scale, etc.) pose new challenges for system designers. In this section, we briefly introduce existing approaches to load balancing in DHT systems.

In DHT systems, much research work about load balancing is based on namespace balancing. Namespace balancing is trying to balance the load across nodes by ensuring that each node is responsible for a balanced namespace (e.g., Wang et al., 2004; Naor and Wieder, 2003, etc.). It is valid only under the assumption of uniform workload distribution and uniform node capacity. For a better balancing effect, “VS” was first introduced in Chord (Stoica et al., 2001). Similar to a physical peer, a VS is also responsible for a contiguous range of the DHT identifier space, but one physical peer can host multiple VSs, and VSs are allowed to split, merge and migrate between nodes. By letting each node host multiple VSs, the $O(\log N)$ imbalance factor (Stoica et al., 2001) between nodes can be mitigated. To reduce the overhead incurred by VSs, Godfrey and Stoica (2005) propose a low-cost VS selection scheme. If node heterogeneity is considered, VSs can be allocated proportionally to node capacity (e.g., CFS Dabek et al., 2001). In case that a node is overloaded, it simply removes some of its VSs. However, such simple deletion will cause the problem of “load thrashing”, as the removed VSs will be automatically taken over by the nodes responsible for that ID space and may make those nodes overloaded. Pure namespace balancing like the above approaches does not perform load migration, thus cannot handle workload skewness well.

A more general approach is migration based, which is applicable to various kinds of scenarios and able to handle workload skewness. A number of migration-based approaches have been proposed to date (Rao et al., 2003; Godfrey et al., 2004; Zhu and Hu, 2005). They can be categorized into two main strategies: (1) RDS and (2) ISS.

In RDS, the load information of each peer is periodically published to the rendezvous directory, which can be a central entity, or organized in a distributed fashion (e.g., tree structure in Zhu and Hu, 2005). The rendezvous directory is responsible for scheduling the load reassignment to achieve load balance.

In ISS, a node does not publish its load information anywhere else, and only provides its load information upon request. To achieve load balancing, a node should perform searching (or sampling) independently to find other nodes with inverse load characteristics, and then migrate the load from the heavy node to the light node.

Rao et al. (2003), propose three simple load balancing schemes: “one-to-one”, “one-to-many” and “many-to-many”. Among them, “one-to-many” and “many-to-many” belong to the RDS category, while “one-to-one” belongs to the ISS category. To enable emergent load balancing, Godfrey et al. (2004) make a combination of “one-to-many” and “many-to-many”, and use them in different scenarios. The scheme proposed in Zhu and Hu (2005) also belongs to the RDS category, but its rendezvous directory is organized as a distributed $k$-ary tree embedded in the DHT.

In this paper, our proposed GMS differs from previous work in that, we take both resilience and efficiency into account. By disseminating the load information and randomizing the position of rendezvous directories, we can exploit the benefits of both RDS and ISS.
Besides, there are some other approaches that try to realize load balancing in DHT systems, such as object balancing based on “power of two choices” (Byers et al., 2003; Mitzenmacher, 1996). Nevertheless, all these techniques can be regarded as complementary techniques and may be combined to provide a better load-balancing effect under certain scenarios.

3. System design

In this section, we propose a GMS for load balancing in DHT systems. The objective of our design is to exploit the efficiency of RDS while improving the resilience and scalability at the same time.

The basic idea of GMS is to disperse the responsibility of load balancing to all the peers, instead of limiting to only a few ones. At the same time, in order to avoid the inefficiency caused by independent searching, load information is disseminated among peers by multicast-like protocols. In each load-balancing round, peers are selected randomly as the rendezvous directories to schedule the reassignments of VSs for a better balance.

In the following, we will introduce the system design in details.

3.1. Load information dissemination and aggregation

GMS is built on top of ring-like DHTs, such as Pastry (Rowstron and Druschel, 2001), Chord (Stoica et al., 2001), etc. Based on the system size, one or more load balancing groups are formed among peers.

For systems with only a few hundred peers, only one group is formed; but for systems with thousands of peers, peers form into multiple groups, each of which corresponds to a continuous region with equal size in the ring. All the peers within the same group share the same prefix, which is referred to as the “GroupID”. The adjustment of groups is decentralized, and does not require global cooperation. The details of group formation procedure will be presented later in Section 4.1.

In GMS, any group multicast protocol can be used for load information dissemination within the group. Here, to reduce the maintenance overhead, a virtual multicast “tree” embedded in the DHT is used for information dissemination. This tree does not require explicit maintenance and just expands based on local information. The node that wants to disseminate its load information is the root of the tree. It sends the message containing load information to all the peers with the same GroupID in its routing table. Subsequently, when an intermediate node \( j \) receives a message that it has not received before from another node \( i \), \( j \) forwards the message to every node \( k \) in its routing table satisfying that \( \text{prefix}(i, j) \) is a prefix of \( \text{prefix}(j, k) \). (see Fig. 1). Here, the function \( \text{prefix}(x, y) \) is defined as the maximum common prefix between nodes \( x \) and \( y \). Through the above approach, we can prevent redundant forwarding.

In every update period \( T_u \), every node should publish its load information once. The load information of a node \( i \) to be published includes:

- The node’s capacity: \( C_i \), which is in terms of load units. Here, load can be storage load, bandwidth load or computation load, depending on the specific applications.

![Fig. 1. Message dissemination.](image)
• The set of VSs hosted by node $i$: $\langle i_1, i_2, \ldots, i_m \rangle$, where $i_k$ refers to the $k$-th VSs hosted by node $i$, $k = 1..m$.
• The load vector of VSs hosted by node $i$: $\langle L_{i_1}, \ldots, L_{i_m} \rangle$, where $L_{i_k}$ refers to the load on VS $i_k$, $k = 1..m$.

Let $L_i$ be the total load on the node $i$, $L_i = \sum_{k=1}^{m} L_{ik}$.
• The node’s IP address.

For a small system with only one group $G$, the load information of each peer can be delivered quickly to all members within a short period. After that, each peer has the load information of all the other peers, and knows the total capacity $C_G$ and the total load $L_G$ of the whole system by $C_G = \sum_{i \in G} C_i$ and $L_G = \sum_{i \in G} L_i$, respectively. Thus, the system utilization $\mu$ can be easily calculated by $\mu = L_G / C_G$.

For a single node $i$, its utilization is given by $\mu_i = L_i / C_i$. Based on the ratio between $\mu_i$ and $\mu$, each node can be classified into three types:

• heavy node, if $\mu_i > \mu$;
• light node, if $\mu_i < \mu$ and
• normal node, if $\mu_i = \mu$.

In practical implementation, as $\mu$ and $\mu_i$ are real numbers, we will round the value of $\mu$ and $\mu_i$ before comparing their value. For example, if $\mu$ equals to 0.67, it will be rounded to 0.7. Similarly, all the $\mu_i$ in the range of $[0.65, 0.75)$ will be rounded to 0.7 and identified as normal nodes.

Another possible approach is to use a range for the identification of “heavy node”, “light node” and “normal node”. For instance, when $0.9\mu < \mu_i < 1.1\mu$, that node is classified as “normal node”; when $\mu_i \leq 0.9\mu$, it is classified as “light node”; when $\mu_i \geq 1.1\mu$, it is classified as “heavy node”.

In case of a large system, there exist multiple groups. Each node $i$ should run at least two VSs (see Fig. 2).

$i_1$ Primary Server of node $i$, which is with the identifier $id_1$ generated by hashing the node’s IP address
$i_2$ Secondary Server of node $i$, which is with the identifier $id_2$ generated by hashing $id_1$.

Each node only publishes load information via its primary server. During load reassignment, the primary and secondary server cannot be moved to other nodes, but their size can be changed.
Given two arbitrary groups \( S \) and \( T \), the probability that there exists at least one node \( i \) whose primary server \( i_1 \) is in one group \( S \) and secondary server \( i_2 \) belongs to another group \( T \) is given by \( 1 - e^{-c} \) (Tang et al., 2005), where \( c \) is the ratio between the group size and the number of groups. With \( c = 5 \), the probability is as high as 0.993.

It implies that, for each group, there exists at least one group member whose secondary server is in any other group with high probability. It provides us with a good opportunity to aggregate load information of other groups and calculate the system utilization.

The aggregation process is as follows: through multicasting in each group, for a given node \( i \), it can have the full load information of at least two groups: group \( S \), where its primary server \( i_1 \) is in; and group \( T \), where its secondary server \( i_2 \) is in. To let other group members in group \( S \) learn the load status of group \( T \), the node \( i \) also propagates the information about the total capacity \( C_T \) and the total load \( L_T \) of group \( T \) within group \( S \) by piggybacking with its own load information. As the secondary servers of group members in \( S \) exist in almost all other groups, after each node disseminates the group load status, every member in group \( S \) learns the load status of all other groups, and is able to estimate the system-wide utilization independently.

### 3.2. Intra-group load balancing

After load information dissemination, every member within a group holds the load information of its own group and the information of system-wide utilization. Each of them has the capability to act as the rendezvous directory to classify the nodes and schedule load reassignment in the group.

However, in order to be resilient to the node-targeted attacks, it is expected that the position of the rendezvous directory can be randomized in each round of load balancing actions. Although some sophisticated randomized algorithms and security mechanisms can be used to make the position selection more secure, we adopt a simple but effective approach. It is based on the assumption that, a node-targeted attack is often costly and the probability to compromise a node within a short period is low.

The idea of our approach is as follows: we associate each round of load balancing with a sequence number \( \text{seq}_lb \), which is known by all group members. \( \text{seq}_lb \) will be increased by one every load balancing round. In each round, every node locally generates a random key by hashing \( \text{seq}_lb \) with a common hashing function. The prefix of a generated key should be replaced by \( \text{GroupID} \) to guarantee the node responsible for the key exists within the group. The node that hosts that key is selected as the rendezvous directory.

However, as there is no centralized time server to synchronize the time on each node, it is possible that nodes within the same group may have different \( \text{seq}_lb \). In our design, we implement the quasi-synchronization of \( \text{seq}_lb \) in a distributed manner. After the completion of load balancing actions, the node who acts as the rendezvous directory increases \( \text{seq}_lb \) by one and disseminates the new \( \text{seq}_lb' \) to all the group members via group multicast for synchronization. The receiving node updates the value of its local \( \text{seq}_lb \) only when the new \( \text{seq}_lb' \) is within the range \( [\text{seq}_lb, \text{seq}_lb + h] \), where \( h \) is a small integer (e.g., 30). By only accepting a bigger \( \text{seq}_lb \), no node can always occupy the position of rendezvous directory via disseminating the same \( \text{seq}_lb \). At the same time, due to the randomness of hash function, it is also not easy for a node to generate a new \( \text{seq}_lb \) with the same hash value while within the range of \( [\text{seq}_lb, \text{seq}_lb + h] \). In the next round, another node will be chosen as the rendezvous directory due to the randomness of hashing functions.

Besides receiving synchronization updates from the rendezvous directory passively, the node also performs active synchronization periodically. Every \( T_{syn} \) period, the node queries its neighboring nodes within the same group for the latest \( \text{seq}_lb \). Using a method similar to Majority Voting (Majority voting, 2008), the node updates its own \( \text{seq}_lb \) as the \( \text{seq}_lb \) in the majority of replies. It is used to handle the situation where the node has not received any update for a period of time or only received wrong \( \text{seq}_lb \) for a number of consecutive rounds.

Although the position of the rendezvous directory is still publicly known, it becomes random and dynamic. Even if the current rendezvous directory is compromised by a malicious node, the service of load balancing can quickly be recovered. As there is no fixed rendezvous directory in the group, GMS
is more resilient than RDS to the node-targeted attacks. The details of intra-group load balancing is given by the pseudocode in Algorithm 1.

**Algorithm 1.** Intra-group load balancing at node $i$.

1: $key \leftarrow \text{Hash}(seq_{lb})$
2: $key \leftarrow \text{PrefixReplace}(\text{GroupID}, key)$
3: 
4: if $key \in i$'s ID space then
5: //node $i$ acts as a rendezvous directory
6: Invoke Greedy VS Matching algorithm to calculate VS reassignments;
7: Disseminate VS reassignments to related nodes;
8: $seq_{lb} \leftarrow seq_{lb} + 1$;
9: Disseminate $seq_{lb}$ to all the group members;
10: else
11: //node $i$ acts as a normal peer
12: if receiving VS reassignment from node $j$ then
13: if $key \in j$'s ID space then
14: perform load migration according to VS reassignment from node $j$;
15: else
16: ignore. //j is not the correct rendezvous directory.
17: end if
18: end if
19: if receiving new $seq_{lb}$ then
20: if $seq_{lb}$ in $[seq_{lb} - seq_{lb} + h]$ then
21: $seq_{lb} \leftarrow seq_{lb}$
22: else
23: $seq_{lb} \leftarrow seq_{lb} + 1$
24: endif
25: endif
26: Query neighbors about their $seq_{lb}$ every $T_{syn}$ interval, update local $seq_{lb}$ as $seq_{lb}$ in the majority of replies;
27: endif

As the problem of computing an optimal reassignment of VSs between heavy nodes and light nodes is NP-complete, a simple greedy matching algorithm (see Algorithm 2) is adopted here to perform VS reassignments. After creating the reassignment decision, the rendezvous directory will send the decision to the related nodes. In order to avoid being overloaded by load reassignment, the node will also check the reassignment decision before performing real transfer. If the transferred load will overload the node itself, the node can reject the transfer of VSs. This is to prevent the node from being overloaded by mistake or malicious reassignment of VSs.

The migration cost of VS is also an important issue that should be addressed during system design. Normally, the migration cost involves two kinds of cost: (1) movement cost (mainly bandwidth cost), which depends on the type of moved objects and the movement distance (e.g., the number of hops) and (2) maintenance cost, which refers the message cost to reconstruct the overlay structure, and normally small. The type of moved objects is quite different in different applications. In DHT storage systems, we need to move the stored files, which is normally in large size; but in DHT index service, we only need to move the metadata information, which is normally in small size. For simplicity, we can define the movement cost $C$ as the product of the size of moved objects and the movement distance (e.g., the number of hops between the source and the destination). For the benefit of load redistribution, it is expected to define a utility function in terms of transferred load to quantify the benefit, e.g., $Util(load)$. The reassignment is only allowed when $Util(load) > C$, but it is not easy to provide a reasonable definition of utility function. Another approach is to set a threshold for
the movement cost. When the movement cost in a reassignment is beyond a predefined threshold, we simply abandon that load reassignment, and continue to choose another destination node. In this way, we can prevent the costly load movement between two far nodes.

Algorithm 2. Greedy VS Matching Algorithm.

1: INPUT: Load information of all the nodes in group \( G \)
2: Initialize \( pool \) = \( \emptyset \)
3: for each node \( i \in G \) do
4: \hspace{1em} while \( i \) is heavy do
5: \hspace{2em} select the least loaded virtual server \( i_k \)
6: \hspace{2em} if \( i_k \) is not primary/secondary then
7: \hspace{3em} move \( i_k \) to \( pool \).
8: \hspace{2em} end if
9: \hspace{1em} end while
10: while \( i \) is still heavy do
11: \hspace{2em} split primary/secondary server into half, move the virtual servers not hosting \( i_1 \) and \( i_2 \) to \( pool \)
12: \hspace{1em} end while
13: end for
14: Sort virtual servers in the pool from heaviest to lightest
15: for each node virtual server \( v \in pool \) do
16: \hspace{2em} assign \( v \) to the node \( j \) which has the minimum utilization after reassignment and satisfies that the movement cost is less than a threshold.
17: end for
18: return the virtual server reassignment.

3.3. Inter-group load balancing

Inter-group load balancing is performed only when there exist multiple groups in the system. Since the distribution of light nodes and heavy nodes may vary in different groups, it is possible that even after intra-group load balancing, some groups may still have many light nodes, while other groups may have many heavy nodes. To handle this situation, inter-group load balancing is allowed in GMS.

In a group \( S \) with many heavy nodes, its average utilization is higher than the average utilization of the whole system. Suppose node \( i \) to be the current rendezvous directory of group \( S \), due to load information dissemination, node \( i \) has the full load information of its own group and the load status (i.e., total capacity, total load) of all other groups with high probability. Node \( i \) can select one group whose free capacity is slightly bigger than the required amount to perform inter-group load balancing.

The process is as follows: Given that inter-group load balancing is performed between group \( S \) and group \( T \), the rendezvous directory \( i \) of group \( S \) first finds another node \( j \) within group \( S \) whose secondary server exists in group \( T \), and then transfers the responsibility to node \( j \). Since node \( j \) has the full load information of both group \( S \) and group \( T \), it can perform the best-fit load reassignment among the nodes of the two groups. The algorithm of VS reassignment is similar to Algorithm 2.

GMS can also provide emergent load balancing. In case that the utilization of one heavy node is beyond a threshold, it can directly contact the lightest node within its group; if there exist no light nodes in its own group, based on collected load information of other groups, it selects the group with the most amount of free capacity and sends a request to one random node in that group, which will return a light node to relieve its load.

With the combination of intra-group load balancing, inter-group load balancing and emergent load balancing, GMS can achieve a system-wide balanced state.
4. Performance analysis

In this section, we perform an analytical study on the scalability and efficiency of GMS. For convenience, the related notations are summarized in Table 1.

4.1. Scalability

The main concern is the scalability of GMS, which determines the applicability of GMS in real large systems.

Define \( g \) to be the group size and \( N \) to be the system size. In case of a small system with a few thousands of peers, one group is formed and the group size \( g \) equals \( N \). The message traffic that a node (including virtual node) experiences per interval is given by \( W_{GMS} \approx O(g/Tu) = O(N) \). For example, given the update period \( Tu = 60 \text{ s} \), \( N = 1000 \) and message size = 30 bytes, \( W_{GMS} \) is about 0.5k bytes/s.

If a node hosts multiple VSs, the traffic increases proportionally to the number of VSs. However, the traffic is still rather low and tolerable for normal users. Even for a powerful node with 20 VSs, the message traffic is about 10k bytes/s. To limit the traffic towards a node, each node can maintain a threshold to limit the maximum number of VSs.

In a large system, multiple groups are formed. Define \( c \) to be the ratio between the number of nodes in a group and the number of groups, we have \( g = \sqrt{cN} \). In case that \( c \ll N \), the message traffic for a peer per interval is given by \( W_{GMS} \approx O(g/Tu) = O(\sqrt{N}) \). Given the update period \( Tu = 60 \text{ s} \), \( N = 250,000 \), \( c = 4(c \ll N) \) and message size = 30 bytes, \( W_{GMS} \) is about 0.5k bytes/s, which is comparable to the small system.

In our approach, \( c \) is an important tuning parameter. In order to guarantee efficient information dissemination between groups, \( c \) cannot be too small, and normally we set \( c \geq 5 \). As \( c = g^2/N \), we can guarantee the above requirement by keeping the group size \( g \) bigger than \( \sqrt{cN} \). Therefore, given the maximum possible system size \( N_{\text{max}} \), we can define two thresholds: the minimum group size \( g_{\text{min}} \), which is to guarantee a bigger \( c \); and the maximum group size \( g_{\text{max}} \), which is to limit the message traffic towards a node.

In the evolution process of the system, the groups are split only when the group size is beyond the threshold \( g_{\text{max}} \). The group split does not require global cooperation between nodes. As each node knows exactly the same number of group members due to information dissemination, when the group size is beyond \( g_{\text{max}} \), it will increase its GroupID by one bit locally. With the changes happening in each node, the system is reorganized into more groups. When the group size is below the threshold \( g_{\text{min}} \), the merging of groups is incurred by self-adaptation similarly. Each node decreases the length of GroupID by one bit. The process is stopped when the length of GroupID reaches zero, which means there exists only one group in the system.

<p>| Table 1 |
| Notations |</p>
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>The system size, i.e., the total number of nodes</td>
</tr>
<tr>
<td>( g )</td>
<td>The group size</td>
</tr>
<tr>
<td>( c )</td>
<td>The ratio between the group size ( g ) and the number of groups ( N/g )</td>
</tr>
<tr>
<td>( W_{GMS} )</td>
<td>The message traffic that a node experiences per interval</td>
</tr>
<tr>
<td>( Tu )</td>
<td>The load information update period</td>
</tr>
<tr>
<td>( N_{\text{max}} )</td>
<td>The maximum possible system size</td>
</tr>
<tr>
<td>( g_{\text{max}} )</td>
<td>The threshold of maximum group size</td>
</tr>
<tr>
<td>( g_{\text{min}} )</td>
<td>The threshold of minimum group size</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>The arrival rate of peers</td>
</tr>
<tr>
<td>( F(x) )</td>
<td>CDF of node lifetime distribution</td>
</tr>
<tr>
<td>( R )</td>
<td>The residual lifetime of a node</td>
</tr>
<tr>
<td>( p_{\text{real}} )</td>
<td>The probability that a peer is a “real” live peer</td>
</tr>
</tbody>
</table>
4.2. Efficiency under churn

The phenomenon of peer dynamics is called “churn”, and may cause the failure of load reassignments. There exist some peers that may leave after publishing their load information. If the rendezvous directory does not know about their departures and still take them as live peers to perform load reassignment with other peers, it will result in the failure of some load reassignments and impact the efficiency of load balancing. Only the peers that are really alive at the time of load balancing are useful and we call them “real” live peers.

What we want to know is that, to what degree does the churn impact the efficiency of GMS, and how to tune the parameters to mitigate its impact?

Based on the measurement results (e.g., Saroiu, 2004; Stutzbach and Rejaie, 2005, etc.) from real Peer-to-Peer systems, we adopt the Pareto lifetime distribution in our analysis, in which the cumulative distribution function (CDF) is given by

\[ F(x) = 1 - \left( 1 + \frac{x}{\beta} \right)^{\frac{1}{\alpha}} \]

where \( \alpha \) represents the heavy-tailed degree and \( \beta \) is a scale parameter. Under the Pareto distribution, most peers have short lifetimes, while some peers stay much longer in the system.

When performing load balancing, we assume that the rendezvous directory only considers the peers that have made updates within the latest update period. By analysis, we have the following result:

**Theorem 4.1.** Under the steady system state, given the update period \( T_u \), the probability that a peer is a “real” live peer in the peer set for balancing at the rendezvous directory is given by

\[ p_{\text{real}} = \frac{1}{((\alpha - 1)/\beta)T_u + (\beta/(2 - \alpha)T_u)((1 + (T_u/\beta)^{1-\alpha} - 1)] \]

for Pareto lifetime distribution with \( \alpha > 2, \beta > 0 \).

**Proof.** Denote \( t_i \) to be the time for the \( i \)-th round of load balancing. Within the latest update period \([t_i - T_u, t_i]\), we denote the set of new joining peers as \( X \). For the peers that leaves the system within this period, based on their live time and departure time, we divide them into three sets (see Fig. 3): (1) \( Y_1 \), the set of nodes that are alive at time \( t_i - T_u \), but leave the system without making any update in \([t_i - T_u, t_i]\); (2) \( Y_2 \), the set of nodes that are alive at time \( t_i - T_u \) and depart before \( t_i \), but make one update before departure and (3) \( Y_3 \), the set of nodes that join the system during \([t_i - T_u, t_i]\), but depart again before \( t_i \).

During the load balancing action at \( t_i \), the nodes in \( Y_1 \) will not be considered, for they do not make any update within \([t_i - T_u, t_i]\). But for the nodes in \( Y_2 \) and \( Y_3 \), the rendezvous directory has no knowledge about their departure and still thinks they are alive. The nodes in \( Y_2 \) and \( Y_3 \) belong to “faked” live peers.

As our analysis is under the steady state, the number of arrival peers equals the number of departure peers, thus we have \(|X| = |Y_1| + |Y_2| + |Y_3|\). Let \( M \) be the number of live peers at time \( t_i - T_u \), then the number of peers in the peer set for balancing at time \( t_i \) can be given by \( M' = M + |X| - |Y_1| = M + |Y_2| + |Y_3| \). At time \( t_i \), among \( M' \) peers, only \( M \) peers are “real” live nodes and \( p_{\text{real}} \) can be given by the ratio between \( M \) and \( M' \). In order to compute \( p_{\text{real}} \), we need to compute \(|X|\) and \(|Y_1|\) first.

Let \( \lambda \) be the arrival rate of peers under the steady state. By applying Little’s Law (Haverkort, 1998), we have \( \lambda = (\alpha - 1)/\beta \) for Pareto lifetime distribution. Then the number of new arrival peers within \([t_i - T_u, t_i]\) is given by \(|X| = \lambda T_u \). After getting \(|X|\), we then proceed to calculate \(|Y_1|\).

Supposing that a peer is alive at time \( t_i - T_u \), then according to (Leonard et al., 2005), given the CDF of peer lifetime distribution as \( F(x) \), the CDF of its residual lifetime \( R \) is given by

\[ F_R(x) = P(R < x) = \frac{1}{E[T]} \int_0^x (1 - F(z)) \, dz. \]
For Pareto lifetime distribution, we have \( F_R(x) = 1 - (1 + (x/b))^{1-z} \). Due to the randomness, the update time \( T \) of a peer since \( t_i - T_u \) can be assumed to follow a uniform distribution in \([0, T_u]\), with the PDF given by \( f(x) = 1/T_u \). Define the probability that the peer's residual lifetime \( R \) is less than its first update time \( T \) as \( p_{Y_1} \), then \( p_{Y_1} \) can be calculated by

\[
p_{Y_1} = P(R<T) = \int_0^{T_u} P(R<x) f(x) dx = \frac{1}{T_u} \int_0^{T_u} F_R(x) dx
\]

Under Pareto lifetime distribution, we have

\[
p_{Y_1} = 1 - \frac{\beta}{(2-z)T_u} \left[ \left(1 + \frac{T_u}{\beta}\right)^{2-z} - 1 \right]
\]

From the above, we can deduce \(|Y_1| = M \times p_{Y_1}\) and obtain \( M' \) accordingly. Thus, we have

\[
p_{\text{real}} = \frac{M}{M'} = \frac{1}{1 + zT_u - p_{Y_1}} = \frac{1}{((z-1)/\beta)T_u + (\beta/(2-z))T_u((1 + (T_u/\beta))^{2-z} - 1)}
\]

To get a better understanding, we take a real DHT-based Peer-to-Peer file-sharing system Kad (Emule project, 2006) as a case for study. According to the measurement results in Stutzbach and Rejaie (2005), the peer lifetime distribution in Kad follows a heavy-tailed distribution with the expectation of 2.71 h, which can be approximated by a Pareto distribution with \( z = 2.1, \beta = 3.0 \). Based on Theorem 4.1, we get the numerical results of \( p_{\text{real}} \) under different update periods (as shown in Table 2).

It can be observed that, by tuning the update period to be smaller, we can keep the value of \( p_{\text{real}} \) very high. In a group with 1000 nodes, by setting \( T_u = 60 \) s, there are less than 4 "faked" live peers in the balancing set on average. In the real environment, if the update period is properly configured, the impact of churn will not be a big issue for GMS.

5. Experimental evaluation

5.1. Methodology

We evaluate the performance of GMS on top of a discrete-event Chord simulator, which is enhanced to support VSs. In our experiments, we simulate a system with up to 8192 nodes, which is bigger than that in the previous simulation settings (e.g., 4096 nodes in Godfrey et al., 2004). The
group thresholds $g_{\text{max}}$ and $g_{\text{min}}$ are set as 1024 and 256 respectively. Synthetic traces are generated to simulate the system churn. The peer lifetime satisfies a heavy-tailed Pareto distribution with $\alpha = 2.1$, which is close to the real Peer-to-Peer systems (e.g., Saroiu, 2004; Stutzbach and Rejaie, 2005). $\beta$ is adjusted to simulate different churn levels.

In the experiment, the node capacity follows a Gnutella-like distribution (Saroiu, 2004), and the average node capacity is able to serve 100 requests per second. Initially, each node is assigned five VSs. Similar to Zhu and Hu (2005), we also assume the size of a VS satisfies the exponential distribution. The update period and the load balancing period are set as 60 and 180 s, respectively.

The workload is generated by the requests from all the nodes uniformly, but the destinations are chosen from either the uniform or Zipf distribution. To serve a request will put one unit of load on the destination node. In case that a request is served by an overloaded node, the request is called “ill request”. We run the simulation for a rather long period, and take the average percentage of “ill requests” among all the requests as the metric to evaluate the effectiveness of load balancing algorithms.

For the reason of comparison, we also simulate four other strategies: (1) RDS: only one powerful rendezvous directory exists in the system, which can perform the best-fit load reassignment; (2) ISS: the light node samples the ID space for the heavy node, and the sample size $k$ per interval is set as 1; (3) Proportion strategy: the nodes remove or add VSs to make the load proportional to its capacity, similar to the approach adopted in Dabek et al. (2001) and (4) no load balancing for the system.

5.2. Experimental results

5.2.1. Impact of workload level

In this experiment, we examine the effectiveness of load balancing strategies under different workload levels.

By varying the request rate initiated by every node, the system workload level is changed accordingly and we record the percentage of ill requests to measure the effectiveness of load balancing strategies. Two kinds of workload distribution are considered: one is uniform distribution, another is skewed distribution (Zipf distribution with parameter 1.2).

The results are shown in Fig. 4. Under the uniform workload, Proportion, ISS, RDS and GMS can all guarantee a low percentage of ill requests even with a high request rate. GMS is slightly worse than RDS, but outperforms ISS and Proportion.

Under the skewed workload, all the strategies perform worse, but GMS and RDS still can retain a low percentage of ill requests. The reason lies in that, both GMS and RDS can utilize the global load information to achieve a more balanced system state, while Proportion and ISS can only perform balancing statically or based on limited information.

5.2.2. Impact of workload skewness

Under some scenarios, the workload may be highly skewed. We also study the effectiveness of load balancing strategies under different workload skewness. We vary the skewness by adjusting the parameter $\alpha$ of Zipf distribution. When $\alpha = 0$, it corresponds to a uniform distribution without skewness.

Fig. 5 plots the percentage of ill requests under different workload skewness. It can be observed that, with the increase of skewness, the effectiveness of all the load balancing strategies is impacted
greatly. When the workload is highly skewed (i.e., $\alpha = 3$), even RDS cannot handle the skewed workload well. However, under different skewness, GMS always has similar efficiency as RDS.

5.2.3. Impact of workload shift

In the above experiments, there is no shift on the workload distribution. It is desirable to know how the load balancing strategies respond in case of workload shifts. We still use the skewed workload (Zipf distribution with $\alpha = 1.2$), but we change the set of destinations in the middle.

Fig. 6 shows the impact of workload shift on the four load balancing strategies. The experiment lasts for 120 min, and the destination set is changed after 60 min. We can find that, when the workload shift happens, there is a burst of ill requests due to the workload imbalance. After some time, all four strategies can smooth out the burst, but the response time is a bit different. Compared with RDS and GMS, Proportion and ISS need more time to converge to the level before workload shift happens. GMS has a short response time, but it is still a bit longer than RDS. It is because that, besides
intra-group load balancing, GMS also needs to perform inter-group balancing in order to balance the whole system.

5.2.4. Impact of system size

Although the scalability of GMS has been analyzed in Section 4, we further validate our results by simulation. In the experiment, the system size is varied from 500 to 9000. We measure the average message overhead per VS.

From Fig. 7, we can observe that the message overhead per VS is less than 0.5kbytes/s most of the time, even with the increasing of the system size. It is because that, when the group size reaches a threshold, the group will be split to keep the traffic low. The effect of group splitting can be found in the figure. The message overhead increases continuously with the increasing of system size, but it drops down when reaching a threshold.

![Fig. 6. Percentage of ill requests under workload shift.](image)

![Fig. 7. Message overhead per virtual server under different system size.](image)
5.2.5. Impact of system churn

We also study how the churn impacts on the load balancing strategies by varying the average peer lifetime. We run the experiments under both uniform and skewed workload distributions (Zipf distribution with $\alpha = 1.2$).

Fig. 8 shows the percentage of ill requests under different churn rates. We can observe that, under both uniform and skewed workload, in spite of the changes of churn rate, GMS and RDS can always lead to a more balanced system state and minimize the percentage of ill requests. GMS and RDS perform slightly worse when the churn rate is higher. The reason is that, a high churn rate will increase the convergence time to the balanced state due to the existence of “faked” live peers, which cause the increase of ill requests accordingly.

In summary, we find that GMS can achieve almost the same efficiency as RDS. But different from RDS, GMS also achieves resilience and scalability at the same time.

6. Conclusion

In this paper, we propose a new load balancing strategy, called group multicast strategy (GMS), to realize both the efficiency of rendezvous directory strategy (RDS) and the resilience of independent searching strategy (ISS). Instead of reporting load information to the rendezvous directory or searching independently, the load information is disseminated via the multicast protocol. Multicast makes it hard to stop the load-balancing service by simply overwhelming a few rendezvous directories. In GMS, every peer can perform load scheduling within its group, but the responsibility is randomized to resist node-targeted attacks. Besides intra-group load balancing, GMS also supports inter-group load balancing and emergent load balancing. The performance of GMS is analyzed analytically, and we also perform extensive simulation to evaluate the effectiveness of GMS under different scenarios.

References

Coral Content Distribution Network (http://www.coralcdn.org/) (December 2006).
Emule project, web site (http://www.emule-project.net/) (December 2006).


