Using Approximation Methods to Implement Quota Mechanism in Peer-to-Peer System

Huaxia Xia hxia@cs.ucsd.edu
Professor: Russell Impagliazzo
Project for CSE 208A New Directions in Algorithms, Spring 2003

Abstraction: The paper discusses the issue of abusing remote resources in peer-to-peer storage system and gives a solution using approximation methods. In the decentralized system, a mechanism that can prevent abusing the distributed resource is necessary for practical application. Our mechanism can be used without any central manager. The analysis shows the mechanism has good efficiency and accuracy.

1. Introduction

Peer-to-peer (P2P) systems have been popularized recently. Basically, P2P system is a kind of distributed system where all sites (we will use “node” instead of “site” in the remainder of the paper) have identical functionalities and responsibilities, there's no central server to control or manager the system. Since P2P systems need no central control, they are very useful in the applications where there are a large number of nodes or the nodes join/leave too frequently. P2P systems also have the advantages of high availability, high reliability, high throughput, etc.

Many research works have been done, especially in the applications of distributed storage. Intuitively, P2P systems can provide almost unlimited storage space. At least three kinds of applications have been studied: First, file-sharing applications such as Napster [1] and Gnutella [2]. In this kind of applications, each node may make some files in its own disks to be "public" so that other nodes can read the files. Second kind of applications is to write some read-only file/blocks to remote disks [3][4]. They are useful for backup or publish. The third kind of applications are distributed file systems, in which user can read/write/change a file in remote disks.

An issue related to the second and the third kinds of P2P systems is how to prevent abusing the storage space. If there's no restriction on how to use the remote storage space, a malicious user can easily break down the system by using up all the storage space; even a non-malicious user might compromise the system by accident.

On solution is to apply quota mechanisms on the storage allocation. Both CFS [4] and PAST [3] have their own quota mechanisms. But they are CFS’s method is too simple to provide efficient storage allocation, while PAST needs a central manager. We will try to use approximation methods to implement a fast, fair and decentralized quota mechanism.

The remainder of this paper is organized as follows. Section 2 introduces the characters of peer-to-peer systems. Section 3 discusses the related work in CFS and PAST systems. Section 4 describes the design of our algorithm. Section 5 analyzes the algorithm. Finally, Section 6 discusses the future works and concludes the paper.

* The paper is the work for project of UCSD CSE208A, Spring'02.
The class is from Prof. Russell Impagliazzo. This paper is submitted on June 7, 2002.

2. Problem specification

As we know, there are many peer-to-peer systems and thus there are different peer-to-peer models. We will focus on one popular category of the models, i.e., hash-based models. Before we go with the algorithms, we will describe some characters of peer-to-peer systems in this category.

Many peer-to-peer routing models are based on uniform hashes, including Chord (used by CFS) [6], Pastry (used by PAST) [5], Tapestry (used by OceanStore) [7], and CAN[8]. They use uniform hash functions to assign random node identities and random data identities; nodes are organized in a special topology according to their node IDs; data blocks are stored in nodes with closest node IDs to their data IDs.

Based on the above facts, we use the following assumptions for our problem:

- The number of nodes in the system is large. We won't think about the small systems because it is not hard to solve the problem for small systems using deterministic algorithms.
- The nodes are uniformly distributed in peer-to-peer system. Here the distribution of nodes means the distribution of nodes' identities. This is true because each node is assigned a random nodeID.
- There's no central server or a trusted third party, i.e., we are thinking about pure peer-to-peer environment.
- Each node knows a subset of the existing nodes. The subset can be considered to be a random set, but should at least include its neighbors.
- Each node knows the approximate size of the whole system. We will give an approximation algorithm in section 4.1 to estimate the size. We assume the approximation ratio to be η.
- Most of the nodes work correctly; only a few nodes, say σ percentage of all the nodes, are malicious nodes. The correct nodes have two properties: firstly a correct node won't use storage exceeding its storage limit; secondly a correct node will give correct answer if any nodes query for the information about its storage space used by a third node.

3. Related Works

3.1 Quota mechanism in PAST

PAST is a large scale, Internet-based, global storage utility that provides scalability, high availability, persistence and security [3].

In PAST, there is a central manager named “broker”. Each node has a “smartcard” that is a pair of public key/private key issued by broker. Each time a node want to use some storage space, the storage size is debited against the quota. The storage space can also be reclaimed, then the client node can use the “reclaim receipt” issued by the depository node to credit its quota.

The smartcard mechanism can control quotas accurately. But the broker may become the bottleneck of performance and also the single point of failure.

3.2 Quota mechanism in CFS

CFS is a completely decentralized peer-to-peer storage system based on Chord [4][6]. It doesn’t use any central control for quota; instead, its quota mechanism is based on the independent decision of each depository node.

CFS uses IP address as the node identities for quota. Each node limits any one IP address to using one fraction, say 0.1%, of its storage. This fraction is decrease in proportion to the total number of servers, which is estimated by Chord software (in paper [4], they didn’t say how to estimate the number).

This scheme is simple, but is not very good. Since data blocks are randomly stored on the nodes, it is expected that much more blocks will be on some nodes than the other nodes. Using this quota mechanism, storage requests from a client node might be rejected even if the storage it is using is much less than its quota.

4. Approximation Algorithm Design

4.1. Problem specification

Before we go with the approximation algorithm, we define some functions that can be called by the nodes to query the topology and storage usage information:

- GetNodeDomain(NodeID)

  As we specified in section 2, each data block will be stored in the node with closest node ID to the data ID. In other words, each node covers a set of data ID. We name the size of the set as “domain” of the node, D(node). Each node can easily calculate its domain according to its neighbors’ IDs.

  For example, in 1-dimensional topology of Chord, a node’s domain is its distance from it’s upper neighbor; in Pastry or Tapestry, a node’s domain can be half distance between its two neighbors; in multi-dimensional topology of CAN, each node corresponds to a multi-dimensional cube, its domain is volume of the cube.
- (TotalSpace, UsedSize) ← GetStorageInfo(DepositoryNodeID,ConsumeNodeID)
  The function gets the information from DepositoryNode about its total storage space and its space consumed by ConsumeNode.
- NodeIDSet ← GetNodes(DepositoryNodeID)
  The function returns all the NodeIDs that the DepositoryNode knows.

- NodeID ← SelectNode(NodeIDSet)
  The function randomly selects a node from a set.

4.2 Estimate the total size N

In a large, dynamic, peer-to-peer system (without central server), no node knows exactly the global status due to the limited communication ability. But we can still estimate the global status using a small number of queries.

To get the total size, we use the assumption that the nodes in P2P systems are usually distributed uniformly. Since the length of node IDs is fixed, we have known the size of the node ID space. By sampling the density of node IDs, we can easily estimate the number of nodes. This density can be got from the "GetNodeDomain" function.

We have the following algorithm to estimate the total size:

```
TotalDomain ← 0;
For a proper integer k;
Repeat k times the following:
    Randomly select a node v = ($x_1$, … , $x_2$);
    Request its Domain D(v);
    TotalDomain ← TotalDomain + D(v);
AverageDomain = TotalDomain/k;
EstimateSize = SpaceSize/AverageDomain.
```

In practical system, each node can periodically run the algorithm to make the latest estimation.

4.3 Detect the quota violation

Given the estimated system size N, each node is expected to use at most λ=1/N of total storage.

Although one node might use much more storage on some nodes than on the others, if we check its storage usage over a relatively large number of nodes, the average ratio should approach its ratio over the whole system.

We use the above property to sample the nodes. When a depository node, say u, receives a storage allocation request from node v, the algorithm for it to check violation is:
We define the following variable:

Total: the total space size in the nodes we have visited
Used: the space used by node v
NodeSet: the nodes whose neighbors haven't be visited

The algorithm is:

```
Total ← u's total space;
Used ← u's space used by v;
NodeSet ←{u};
Mark u as "visited";    // we will discuss this in section 4.4.
```

```
repeat:
        If Used/Total $\leq \lambda$, then we stop and approve v’s request;
        If we have visited all the nodes (or many), then stop and reject v’s request;
        Otherwise:
        Randomly select a node w from NodeSet;
        NodeSet_w ← GetNeighborInfo(w)
        For each node t in NodeSet_w, if it isn’t marked before, do:
                (Total_t, Used_t) ← GetStorageInfo(t, v)
                Total ← Total + Total_t;
                Used ← Used + Used_t;
                NodeSet ← NodeSet + {t};
                Mark t as “visited”;
        ENDFOR
        NodeSet ← NodeSet – {w};
ENDREPEAT
```

In most cases, since v doesn’t violate the quota, the algorithm will stop quickly; only for those few cases the algorithm needs more time to reject the request. We’ll analyze it in detail in section 5.2.

4.4 Mark visited nodes

An easy way is of course to keep the visited nodes in a binary tree. The time for query or insertion is O(logN). The space requirement is O(N), where the coefficient could be 24 if we use two 4-byte pointers and one 16-byte ID for each node. This is affordable for current systems (the size of the Napster network is 160,000 simultaneous users in July 2001 and the number of Gnutella users per day is between 20,000 and 50,000 in January 2001 [10] [11]).

For even larger P2P systems, we can use another approximation method with less accuracy but better space efficiency. We use the tool of “Bloom filters”[12]. Bloom filters are a bit-vector of length *w* with *L* independent hash functions, each of which maps an element to an integer in [0, *w*).

In the beginning, the *w* bits are all initialized to be ‘0’. Each time a node is visited, we hash its ID using the *L* hash functions and got *L* integers in [0, *w*]. If any of the *L* corresponding bits in the bit-vector are zero, we set them to be ‘1’ and consider the node to be a new node; otherwise, we think that the node has been visited before.

Obviously, it’s possible that all the corresponding bits for a new node have been set before its arrival and the new node will be considered as an old node, which we call “false hit”. We need to find proper values of *w* and *L*. We’ll do this in section 5.1.

5. Algorithm Analysis

5.1 Analysis for bloom filters

Now we want to know choose a proper value for the length *w* of bit-vector and number of the hash functions *L*. The objective is to reduce false hits as well as to reduce the storage requirement.

Assume X_i to be the number of "1" bits after the i-th new node is visited. We think about the visiting on the i-th new code. Since there are $(w - X_{i-1})$ "0" bits, each hash function will hit a "0" bit in the probability of $\frac{w - X_{i-1}}{w}$. Thus *L* hash functions can hit:

$$X_i - X_{i-1} = L * \frac{w - X_{i-1}}{w}$$

$$\Rightarrow X_i - (1 - \frac{L}{w}) X_{i-1} - l = 0$$

$$\Rightarrow X_i - w = (1 - \frac{L}{w})(X_{i-1} - w)$$

Since $X_0 = 0$, we can get:

$$X_i = w - w * (1 - \frac{L}{w})^i$$

Each new node has the probability $(X_i/w)^L$ of false hit. So the expected number of total false hits is:

$$E = \sum_{i=1}^{N} (\frac{X_i}{w})^L = \sum_{i=1}^{N} (1 - (1 - \frac{L}{w})^i)^L \approx \sum_{i=1}^{N} (1 - (1 - i * \frac{L}{w}))^L \approx \sum_{i=1}^{N} (\frac{iL}{w})^L$$

Thus if we let L=logN, w=2NlogN, then the expected number of total false hits is

$$\sum_{i=1}^{N} (\frac{iL}{w})^L = \sum_{i=1}^{N} (\frac{i \log N}{2N \log N})^{\log N} < \sum_{i=1}^{N} (\frac{1}{2})^{\log N} = 1$$

The storage space for the bit vector is only NlogN/4 bytes. For example, for 1 million nodes, we only need 5MB. And I guess we can decrease w to be NlogN or NlogN/2 and still get good accuracy. (I have no time to calculate it carefully:-P).

5.2 Analysis for quota violation detection

It's easy to know that the algorithm will always give correct answer for node that doesn't exceed its quota (assume we make a reasonable estimation on total size).

For node that does exceed the quota, we want find out the probability that the algorithm gives wrong answer.

I'll give up the credits for this part of analysis because I haven't found out how to calculate some of the probabilities. But the main idea is like:

1. Assume the quota is *qn* blocks. A malicious node is using storage $m=(1+\delta)qn$ blocks.

When it requests for a new block storage, the probability that a validation is motivated is equal to the probability that the new block is on a depository node where there are at least *q* blocks from the malicious node.

2. If a validation is motivated, what's the probability that the malicious node luckily escapes being detected on its violation? A node may escape if the depository node queries some nodes where much less storage is used by the malicious node. In this case, the validation queries will follow a path where the accumulated average storage keeps exceeding the limit until the last depository node is counted in. We need to know the probabilities of such paths with different lengths. Then the sum of them is the "escape probability" for one new block.

3. We then multiply the escape probabilities from qn blocks to $(1+\delta)qn$ blocks. This is the probability that a malicious node can successfully use $(1+\delta)$ time of its quota without being detected the violation.

Intuitively, although the malicious node can escape the detection on each single block in a relatively high probability, it will be detected when it exceeds the limit many blocks.

6. Future Work and Conclusions

One of the future works is to complete the analysis. I believe that the bit-vector length for bloom filters can be much less than $w=2N\log N$. A good way to do this is to draw a chart of the expected false hits on different w and L.

The violation detection analysis should be continued.

The detection time for legitimate nodes is also worth to analyze, because this gives the overhead of our algorithms.

This paper describes some approximation algorithms to implement quota mechanism in large decentralized peer-to-peer systems. The analysis shows that the algorithms are efficient and accurate.

References

[1] Napster. http://www.napster.com/

[2] Knowbuddy's Gnutella FAQ, http://www.rixsoft.com/Knowbuddy/gnutellafaq.html

[3] P. Druschel and A. Rowstron, "*PAST: A large-scale, persistent peer-to-peer storage utility*", HotOS VIII, Schoss Elmau, Germany, May 2001

[4] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, Wide-area cooperative storage with CFS, 18th ACM Symposium on Operating Systems Principles (SOSP'01), October 21-24, 2001 Chateau Lake Louise, Banff, Alberta, Canada

[5] A. Rowstron and P. Druschel, "*Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*". IFIP/ACM International Conference

on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November 2001

[6] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications,ACM

[7] SIGCOMM 2001, San Diego, CA, August 2001.

[8] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker. A Scalable Content-Addressable Network. In Proceedings of the ACM SIGCOMM, 2001.

[9] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing, U. C. Berkeley Technical Report UCB//CSD-01-1141, April 2000.

[10] Morpheus Out of the Underworld, by Kelly Truelove and Andrew Chasin, http://www.openp2p.com/pub/a/p2p/2001/07/02/morpheus.html

[11] Gnutella: Alive, Well, and Changing Fast, by Serguei Osokine, http://www.openp2p.com/pub/a/p2p/2001/01/25/truelove0101.html

[12] B. Bloom, Space/time trade-offs in hash coding with allowable errors, in *Communications of the AC*M, July 1970, vol. 13(7), pp. 422–426.