國 立 交 通 大 學

資訊科學系

碩 士 論 文

一合作運行之分散式記憶體快取系統

A Cooperative and Distributed Memory Cache System

研 究 生：鐘建彬

指導教授：黃俊龍　教授

中 華 民 國 九 十 八 年 九 月

一合作運行之分散式記憶體快取系統
A Cooperative and Distributed Memory Cache System

研 究 生：鐘建彬　　　　Student：Chen-Bin Chung

指導教授：黃俊龍　　　　Advisor：Jiun-Long Huang

國 立 交 通 大 學
資 訊 科 學 系
碩 士 論 文

A Thesis

Submitted to Department of Computer and Information Science

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science

September 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年九月

一 合 作 運 行 之 分 散 式 記 憶 體 快 取 系 統

學生：鐘建彬　　　　　　　　　　　　　　　　　　指導教授：黃俊龍

國立交通大學資訊科學與工程研究所碩士班

摘　　　要

記憶體快取是設計將資料和物件暫存到記憶體以減少額外的資料讀取次數如資料庫和磁碟 I/O 以增進系統效能。Memcached，是一個由 Danga Interactive 設計給 LiveJournal 且開放原始碼的記憶體快取的實作，提供了有效率的單台機器的記憶體快取功能。然而，一台 32 位元的機器頂多有 4GB 的記憶體，這些記憶體對大規模的系統來說是不足的。在這篇論文裡，我們根據 memcached 設計一個有彈性的分散式記憶體快取系統的架構以解決記憶體容量不足的問題，並且為了管理伺服器及平衡工作量設計了一個分散式的網路給記憶體快取伺服器們。實驗的結果展示了當使用相同的分散方法時，我們能給予相較於傳統方法較佳的系統效能。此外，較少的記憶體被逐出的次數證明了我們能有效的利用伺服器們的記憶體。

A Cooperative and Distributed Memory Cache System

Student：Chen-Bin Chung　　　　　　Advisor：Dr. Jiun-Long

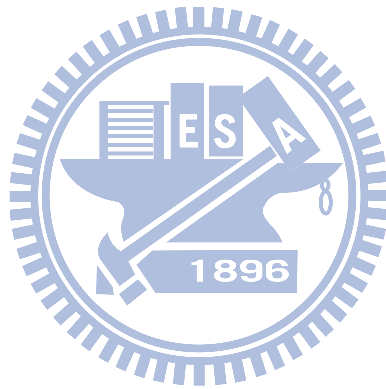
Institute of Computer Science and Engineering
National Chiao Tung University

ABSTRACT

Memory Cache is designed to speed up the system performance by caching data and objects in memory to reduce the number of times an external data source must be read such as database and the amount of disc I/O. Memcached, an open-source implementation of memory cache developed by Danga Interactive for LiveJournal, provides efficient memory cache function on a single machine. However, there is at most 4GB memory in a 32-bit machine and it does not meet the requirement of the large-scale systems. In this paper, we design a scalable distributed memory cache system architecture based on memcached to solve the problem of limited memory capacity and designs a decentralized network of memory cache server to manage the servers and balance the workload. Experimental results show that we provide a better system performance than using traditional method when a client uses the same distribution method. Besides, the fewer evictions proves that we can effectively use the memory of the servers.

誌　謝

很高興能在這兩年的碩士生活中完成了我的論文，非常感激我的朋友、家人給予我的支持與鼓勵。指導教授黃俊龍教授，在研究的過程中，總是給我點清了問題並給予我建議，讓我在研究上遇到瓶頸的時候可以順利突破，並且不以高姿態的方法進行指導，對我來說可說是亦師亦友。最後要感謝我們實驗室的其他學長和同學,和他們在實驗室兩年相處下來很愉快，讓我有了良好的研究環境和心情，才能順利的把研究進行到今天，並且讓我留下了兩年美好的回憶。



Contents

中文摘要 i
英文摘要 ii
誌謝 iii
目錄 iv
圖目錄 v

1 Introduction 1

2 Preliminaries 4
2.1 Further Introduction to Memcached 4
2.2 RelatedWork 5
2.2.1 Using Consistent Hash to Distribute Objects 5
2.2.2 Peer-to-peer Protocol 7
2.3 Design Issues of Memory Cache System 9

3 System Architecture and Protocol **11**
3.1 AOM: Assistant of Memcached 12
3.2 Client 18
3.3 Implementation of system.. 19
4 Performance Evaluation **22**
4.1 TestBed 22
4.2 The Impact of Evictions on Hit Rate.. 22
4.3 Replication.. 25
4.4 Performance on Different Machines.. 26
4.5 Utilizations of Servers.. 27
5 Conclusion.. **29**

6 Bibliography **30**

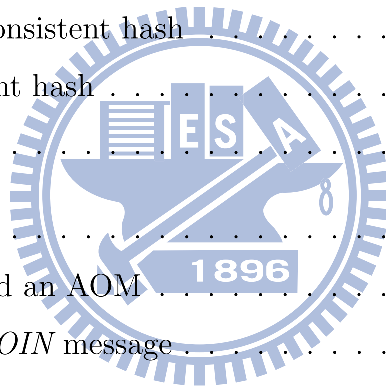
1896

List of Figures

1.1 Scenario of memory cache . . . 1
1.2 Example of the memcached protocol . . . 2
1.3 Performance of memcached . . . 2
1.4 The list of memcached clients . . . 3

2.1 Slab allocation . . . 5
2.2 Weighted consistent hash . . . 5
2.3 Time based weighted consistent hash . . . 6
2.4 An example of consistent hash . . . 7
2.5 Chord protocol . . . 8

3.1 System overview . . . 11
3.2 Structure of a client and an AOM . . . 12
3.3 Handling of an *AOM JOIN* message . . . 13
3.4 An example of *AOM JOIN* . . . 14
3.5 A special case of *AOM JOIN* . . . 14
3.6 Movement for balance . . . 15
3.7 Performance under different CPU loads . . . 16
3.8 Definition of $Load_{CPU}$ . . . 16
3.9 Definition of $Load_{utilization}$ . . . 16
3.10 Definition of $Load_{memory}$ . . . 17
3.11 An example of stale load . . . 17
3.12 Failure recovery . . . 18
3.13 System architecture . . . 19
3.14 Distribution methods . . . 20
3.15 The format of a message . . . 20
3.16 Header . . . 20
3.17 The format of content and content token. . . . 21



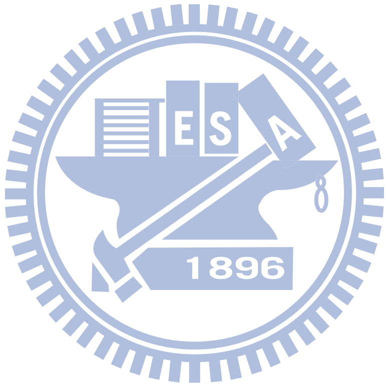
4.1 Evictions caused by traditional method and AOM 23

4.2 The cache miss caused by the evictions. 24

4.3 The relationship between replication and hit rate. 25

4.4 Performance when some server are under high workload 26

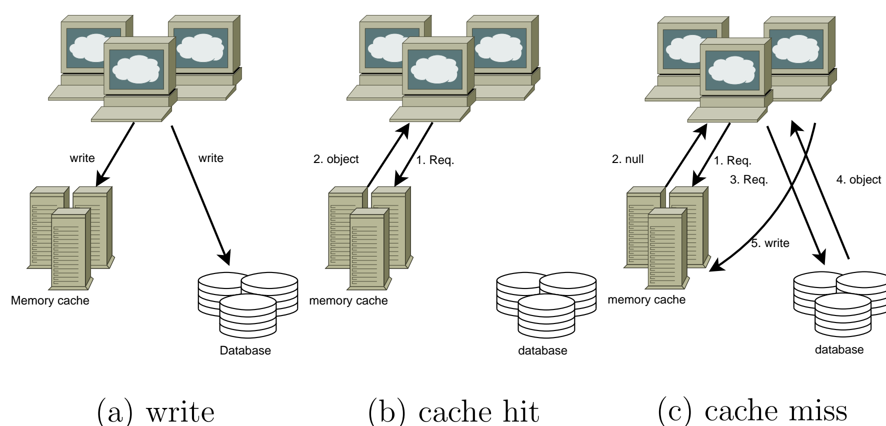
4.5 Evictions caused by traditional method and AOM 27



Chapter 1

Introduction

Large-scale websites usually store the objects or data to the parallel database system[1]. However with the growth of number of users, the performance of database systems is hard to satisfy the expectation and has become the bottleneck. To solve this problem, a website with high flow ratejk uses memory cache to cache the hot or popular objects into the memory cache server to speed up system performance. Figure 1.1 shows the common scenarios of memory cache. Besides the original database servers, we set up the memory cache server additionally. The web servers write through the objects to both memory cache servers and database servers when writing, and load the data from memory cache server. The web server visits the database only when a cache miss occurs in the cache server and it stores the result loaded from database to the memory cache servers to increase the hit rate in the future.



(a) write (b) cache hit (c) cache miss

Figure 1.1: Scenario of memory cache

Memcached[12] is an implementation of memory cache server by Danga Interactive with BSD license. Many websites like Yahoo! and Facebook use memcached as their memory cache servers[12][13]. The design principle of memcached is based on server-client architecture and the memcached server deals with the requests generated by the clients according to a simple

memcached protocol. A key-value pair is used in each object of memcached, and the key is the index to search the objects. Figure 1.2 show the simple examples of the memcached protocol. Memcached is not only simple but fast. We store 16000 different size of objects to the memcached and postgresql then measure the time spending of them. Figure 1.3 shows that the performance of memcached is good.

```
>set key 0 0 5\r\n
>hello\r\n
STORED
>
```

(a) store

```
>get key\r\n
VALUE key 0 5
hello\r\n
>
```

(b) load

Figure 1.2: Example of the memcached protocol

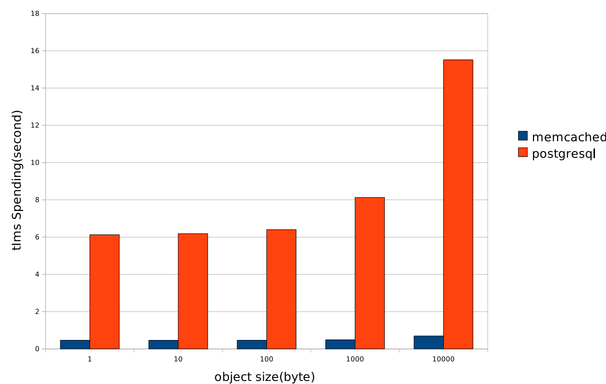


Figure 1.3: Performance of memcached

The memcached protocol doesn't contain the distribution method and there is no official

client or API of memcached thus the user can write its client or API according to the memcached protocol when it need, however, there are many different clients can be used at present and most of them have their distribution methods. We list some of clients in figure 1.4 and it includes the different programming languages.

Programming Language	Package Name	Support consistent hash?
c/c++	libmemcached	yes
	libketama	yes
PHP	PECL/memcached	yes
	PHP libmemcached	yes
Java	Spymemcached	yes
	Java memcached client	yes
Windows/.NET	.Net memcached client	yes
	beitmemcached	yes

Figure 1.4: The list of memcached clients

This work discusses the memcached and its clients and point the spot where can be improved. A good system architecture of memcached server and client should have the following properties. First, the distribution method of clients should be a balanced method. It means that the decision of data distribution should be depended on the load of servers and the memory capacity of the servers. A good hash function method balances the load natively, however, it just statically prevents the unbalance but can not dynamically balance unbalanced system. Second, a client should not take much time on distribution method to decide where to store or load the objects so the distribution method should be simple. And at last, the variant of servers should be transparent for clients. This means a client does not have to be modified when a server joins or leaves the server poll and the impact of the variant of servers to clients should be minimized. This work provides an architecture and its protocol of memcached and clients to reach the properties we mentioned. We dynamically balance the system with a simple distribution method and transparency the joining and leaving of servers for clients under this architecture with just little additional overhead.

The rest of the paper is organized as follows. The related work and design issue are discussed in Chapter 2. The system architecture, protocol, distribution method and implementation are presented in Chapter 3. Chapter 4 shows the performance evaluation. Finally, chapter 5 concludes this paper.

Chapter 2

Preliminaries

2.1 Further Introduction to Memcached

We discuss the inner mechanism and implementation of memcached in this section by listing and explaining the features of memcached according to the document, the source code of memcached and the Petrovici's research[11].

1. *Text Based and Binary Protocol:*

   A communication between memcached and its clients is based on the simple text based protocol rather than complicated XML or others, so we can use a telnet to access the memcached and we have shown the example in previous chapter. To improve the performance, developers translate the text based protocol into binary protocol later and it can be used in the recently versions of memcached. There is no security mechanism in the protocol, so memcached is proposed to setted up behind a firewall.

2. *Slab Allocation*[5]

   Memcached use slab allocation to improve its performance. The idea of slab allocation is preallocating a large memory space and dividing it into different sizes of classes. When program needs the memory, it chooses the fitted class of slab to use and unlinks the memory but not frees the memory when deallocation. Figure 2.1 shows the example of the slab allocation. Slab allocation reduces the number of times of the system call like the malloc. Because memcached is a series of operations of memory, slab allocation improves the performance of memcached substantially.

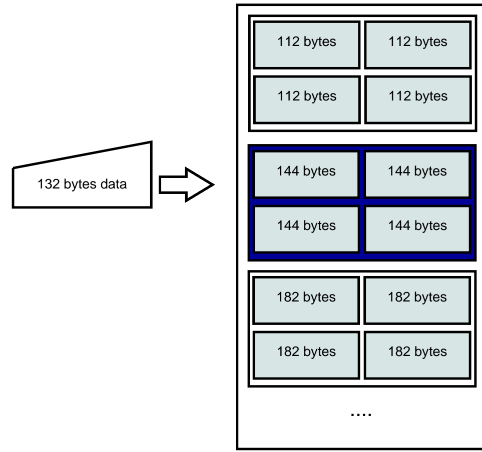


Figure 2.1: Slab allocation

2.2 Related Work

2.2.1 Using Consistent Hash to Distribute Objects

Consistent hash provides hash table functionality in a way that the addition or removal of one slot does not significantly change the mapping of keys to slots. In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped by mapping the objects according to the relative results.

Weighted Consistent Hash

Weighted distributed hash table[7] is a space considered consistent hash. There are two methods in the research, the linear method and the logarithmic method and the formulas of these method are in figure 2.2. Both of them are added the parameter of memory capacity into the functions. The parameter s is the position of the object, r is the position of the server and w is the free capacity of the server. The clients choose the server that minimizes the D_w or L_w to visit when they need. Under these methods, the clients will visit the servers with abundant free memory space in a high probability, and the server with low free space in a low probability in the contrary. This method degrade to normal consistent hash when w is 1.

$$D_w(r,s) = \frac{(s-r)mod1}{w}$$

(a) linear

$$L_w(r,s) = \frac{-ln((1-(s-r))mod1)}{w}$$

(b) logarithmic

Figure 2.2: Weighted consistent hash

Time based weighted distributed hash table[10] provides another hash function based on the same idea but there are more new elements in the function. The function shows in figure

2.3 where t is the distribution time between the node and data, b is the network bandwidth, c is the capacity and d is the physical distance between the client and data.

$$\alpha D = \frac{(s-r)mod1}{cb}dt$$

Figure 2.3: Time based weighted consistent hash

Extended Consistent Hash

Extended consistent hashing(ECH)[8] uses different approach to improve the consistent hash. It gives the client a view set and a window size w then uses this two parameter to limit or balance the distribution. ECH works as follows.

1. Each object and cache server are mapped to the hash ring.
2. Determine a suitable request window size w according to the spread of an object.
3. Requests to an object are randomly mapped to a cache in a window of size w in the view set of each client and the window starts at the point on the ring to which the object is mapped like consistent hash.

ECH is identical to normal consistent hash when the view set of each client is all servers and the window size is 1.

Discussion

Designing a complicated consistent hash method is a way to balance the system. However, the performance is dropped due to the additional computing especially in weighted consistent hash because all servers must participate in the computing and are possible to be the mapped server every time. The shortcoming of extended consistent hash is that it balances the distribution locally in each window and the servers in different windows can not help each other. Besides, increasing the window size will get the more balanced result but the hit rate will drop.

Consistent Hash Distribution Method

Most of all memcached clients work as follows. They first add the known servers into their server poll then assign each of server a position according to the hash result of the name of the server (The name can be the IP address of the server or some others). Using consistent

hash method[2] when they store or load the objects. A usage of consistent hash method is described as follows. Suppose there are n memcached servers s_1 to s_n is ready to use. A client adds they into his servers poll and assigns p_1 to p_n as their positions. An object with key k and value v will store to(load from) s_i if the result of $hash(k)$ is prior to s_i in the hash ring. Figure 2.4 shows a simple example. There are five servers and the position of them are p_1 to p_5. The object o_1 belongs to s_5 because it is prior to s_5 and object o_2 belongs s_1 following the same principle.

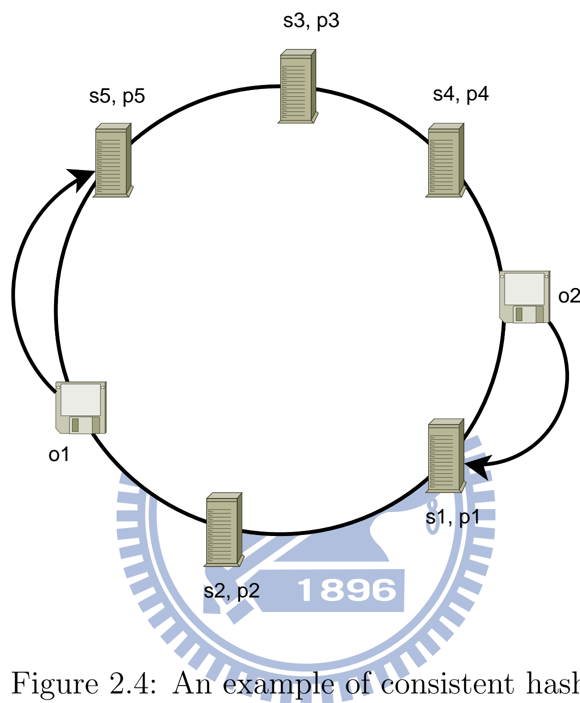


Figure 2.4: An example of consistent hash

Although it works good, however, some issues arise and we will discuss in the following section.

2.2.2 Peer-to-peer Protocol

Some peer-to-peer protocols like Chord[9] provide protocols to organize and look up the network. Chord is designed for a large-scale network, and it contains the decentralized method of key location, how a new node joins or leaves and failure recovers. Here we just care about the join, leave and failure recover because look up is unnecessary.

In Chord, each node updates its successor and predecessor when it invokes $join()$ or $stabilize()$. When node n first starts, it calls $n.join(n')$, where n' is any known Chord node. Then node n finds the successor of n by itself but $join(n')$ does not make the rest of the network aware of n. Every node runs $stabilize()$ periodically to learn about newly joined

nodes and when each time a node n runs *stabilize*(), it asks the successor for its predecessor then checks whether the predecessor is not n. If so, giving the new successor the chance to change its predecessor to n. Each node also checks if predecessor is fail periodically to clear the nodes predecessor pointer if it has failed. We show the functions of the Chord protocol in figure 2.5.

```
1 n.create()
2     predecessor = nil;
3     successor = n;
```

(a) Creat

```
1 n.join(m)
2     predecessor = nil;
3     successor = m.find_successor(n);
```

(b) Join

```
1 n.stabilize()
2     x = successor.predecessor;
3     if(x in (n, successor))
4        successor = x;
5     successor.notify(n);
```

(c) Stabilize

```
1 n.notify(m)
2    if(predecessor is nil or m in (predecessor, n))
3        predecessor = m;
```

(d) Notify

Figure 2.5: Chord protocol

Although Chord is good in peer-to-peer network, it is not suitable for the cache system. All

actions like join, leave and failure recover complete after the corresponding cycle of *stabilize*() comes. In a distributed cache system, a delay may cause the additional cache miss and effect the performance. Besides, Chord does not provide mechanism of movement of nodes.

2.3 Design Issues of Memory Cache System

1. *Balance Issue:*

   The distribution of visiting the objects in the web is usually governed by Zipf's law[3]. The data distribution, especially in web cache[4] may be not an uniform distribution but others .The traditional consistent hash method may be a balanced method when the data distribution is a uniform distribution, but under the most of conditions, it is not a well balanced method. Beside, the state of unbalance will become more serious as time going if there is no dynamic balancing mechanism. The resources will be wasted and the system performance will degrade when the system is unbalanced.

   The memory capacity of one server is limited. When a new object is going to be stored to a memory cache server that the server's memory is used out, there are two choices. The one is to ignore the request, and the another is to replace some old objects with the new one by the FIFO, LRU or other cache replacement mechanism[6]. Memcached provides both choices and they can be chosen by the user when starting the memcached daemon. However, both choices will decrease the cache hit rate because there is at least one object need to be disposed, no matter it is the old object or the new object. To avoid this situation, a client should store an object to the server that the server holds relatively much free memory capacity in a relatively high probability. On the other hand, even though memcached is a I/O bounded program, it takes the CPU in a certain extent. When the CPU load is too high, the performance will sharply drop, so a client should not visit the high loading server as far as possible.

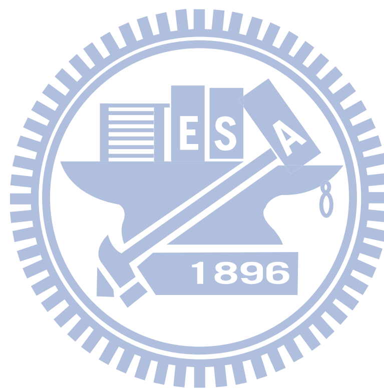
2. *Extend Ablility:*

   The simplest way to get the servers in the server poll is supposing a client has already knows all the servers. However, when the system is under high load or the memory capacity is almost used out, we may want to add new servers to share responsibility for the load or to reduce the cache miss rate caused by the insufficient capacity. At present, memcached and its clients don't possess this ability. We may have to expand the memcached protocol,or to design a front-end of the memcached server to enable the server to join or leave the server poll at will. Besides, the clients must synchronize the

server poll thus they can correctly visit the servers.

3. *Fault Tolerant:*

   Servers may fail. To decrease the impact when servers fail, the network should kick the failed server when it detects thus the clients will not visit the failed servers.



Chapter 3

System Architecture and Protocol

Figure 3.1 shows the system overview of our work. An assistant of memcached(AOM) is performed in every memcached server and it is responsible for running AOM protocol to organize the network of servers.

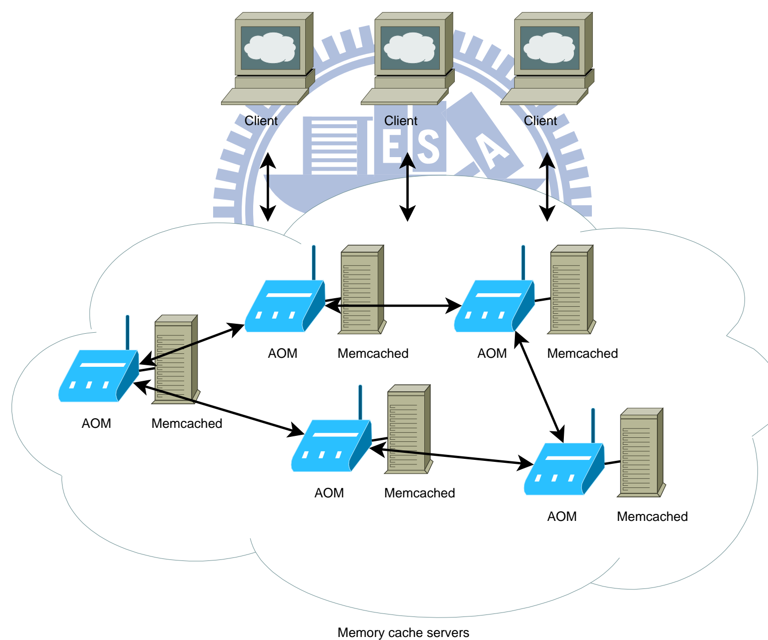


Figure 3.1: System overview

An AOM follows the memcached protocol to cooperate with the corresponding memcached server and runs the AOM protocol to organize all memcached server into a network. The client also follows the memcached protocol to operate the memcached servers and it runs the client part of AOM protocol to synchronize the servers. Besides, there is a distributed method in our client like other common memcached clients. Figure 3.2 is the diagram of structure of the AOM and our client. We first introduce the AOM and then introduce the client.

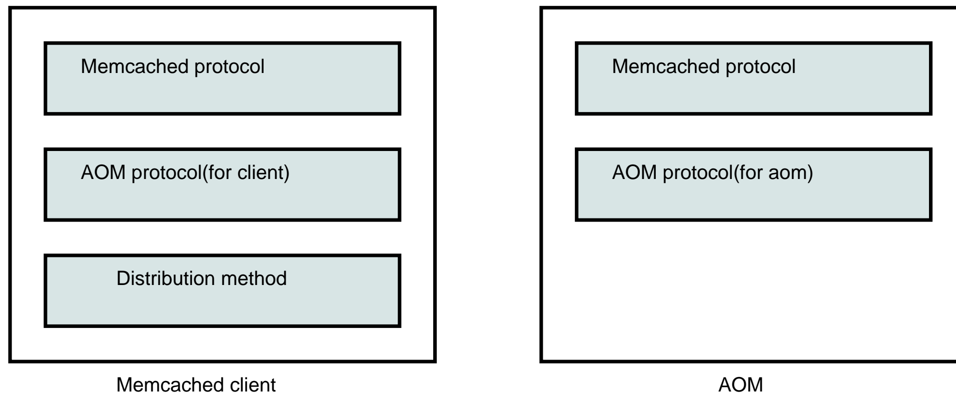


Figure 3.2: Structure of a client and an AOM

3.1 AOM: Assistant of Memcached

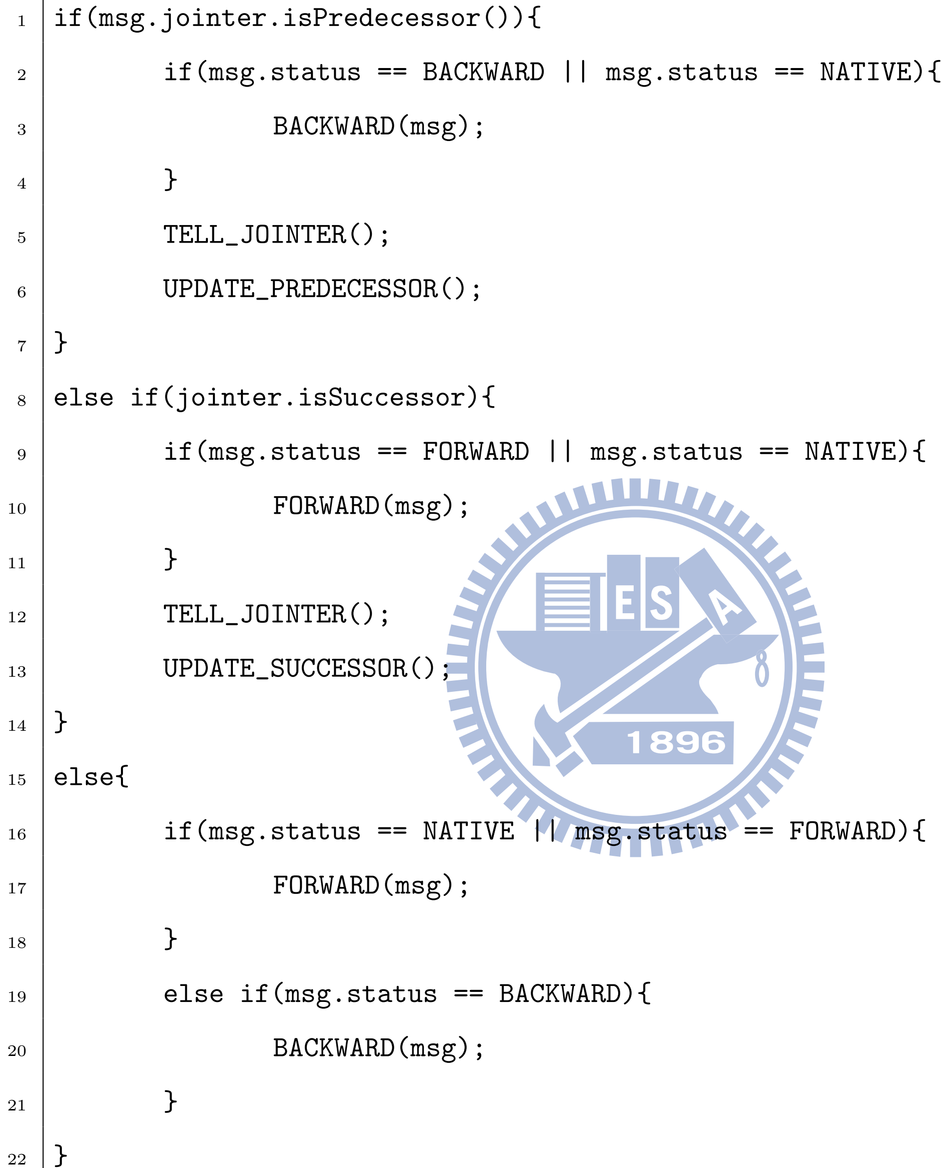
An AOM represents the memcached for foreign and it talks with other AOMs and clients. An AOM maintains a successor and a predecessor like Chord, besides, it maintains a client list and recovery information. The former is for organizing the network of servers, and the latter is for synchronizations and failure recovery. Both of them are updated by the following events and these events are happened when an AOM receives a message sent by other AOMs or clients.

1. *AOM JOIN*

   When an AOM wants to join the network of servers, it sends the *AOM JOIN* message containing its default position made by the hash(JS hash) result of its IP to any node n in the network. If n is null, the jointer becomes the first node in the network otherwise the node n checks if the jointer is its predecessor or successor and updates the predecessorconditions. How to decide the message should be pass to next or previous node is based on the status of the messages. There are three kinds of status of a message and they are "Forward", "Backward" and "Native". A message's status is "Native" when the message was born, then it becomes "Forward" when the first AOM receives the message, or becomes "Backward" if the jointer is the first receiver's predecessor. An AOM will pass the message to the successor if the status of the message is "Forward" or pass the message to predecessor if the status of the message is "Backward". The terminating conditions is simple and there are two terminating conditions. The first one is that the jointer is receiver's predecessor and the message's status is "Forward", and the second is that the jointer is receiver's successor and the message's status is "Backward".

On the other hand, when an AOM receives a "Native" *AOM JOIN* message, it passes the message to all clients for synchronizations.

The detail of handling a *AOM JOIN* message are described in figure 3.3 and the example of a *AOM JOIN* shows in figure 3.4 and figure 3.5.



```
1  if(msg.jointer.isPredecessor()){
2          if(msg.status == BACKWARD || msg.status == NATIVE){
3                  BACKWARD(msg);
4          }
5          TELL_JOINTER();
6          UPDATE_PREDECESSOR();
7  }
8  else if(jointer.isSuccessor){
9          if(msg.status == FORWARD || msg.status == NATIVE){
10                 FORWARD(msg);
11         }
12         TELL_JOINTER();
13         UPDATE_SUCCESSOR();
14 }
15 else{
16         if(msg.status == NATIVE || msg.status == FORWARD){
17                 FORWARD(msg);
18         }
19         else if(msg.status == BACKWARD){
20                 BACKWARD(msg);
21         }
22 }
```

Figure 3.3: Handling of an *AOM JOIN* message

2. *CLINET JOIN*

An AOM must know the clients thus it can tell the clients its information for synchronizations. When an AOM receives the *CLIENT JOIN* message, it pushes the new client into its client list. The AOM sends its own information to the jointer then passes

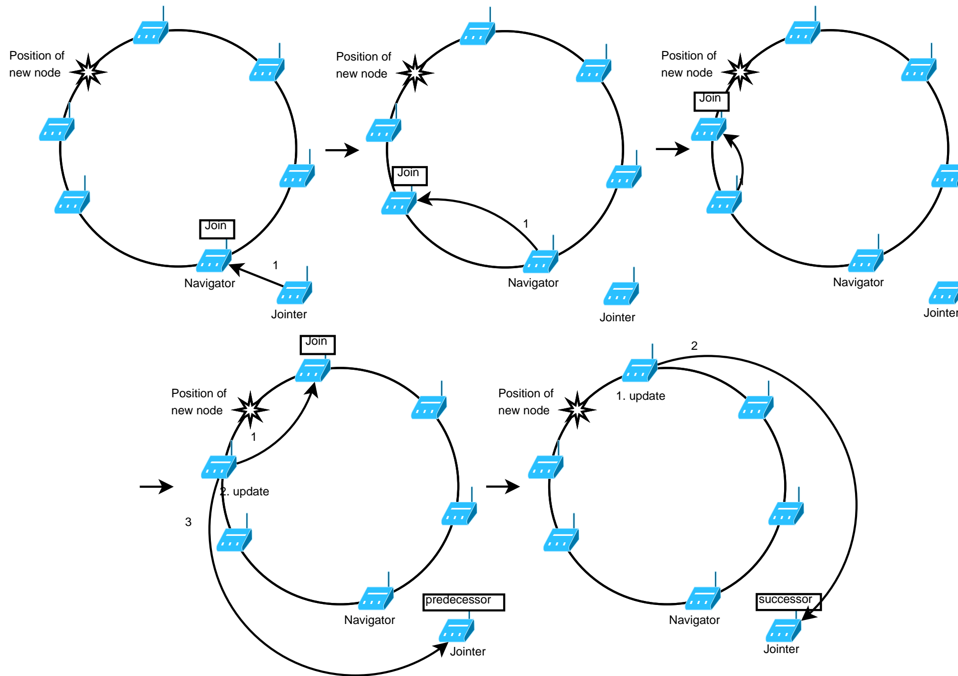


Figure 3.4: An example of *AOM JOIN*

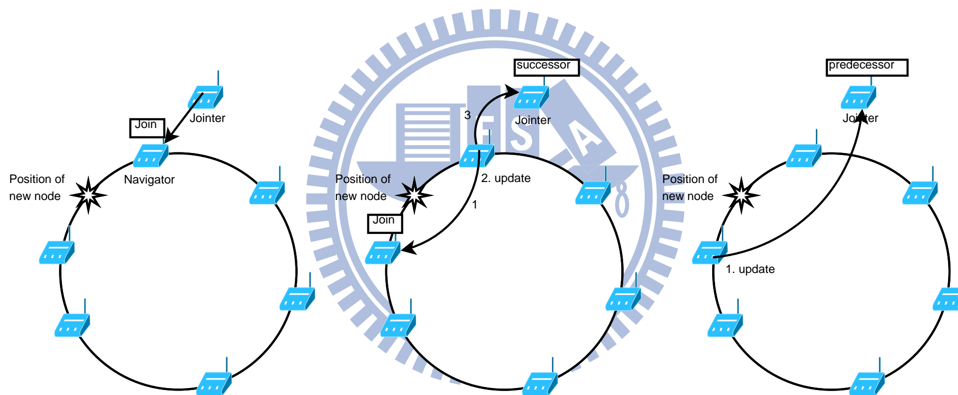


Figure 3.5: A special case of *AOM JOIN*

the message to the next node unless the message fits the terminating condition. The terminating condition is that the receiver already has the jointer in its client list.

3. *AOM LEAVE*

   When an AOM going to leave the network, it sends the *AOM LEAVE* message to its predecessor and successor. There are three informations in a *AOM LEAVE*, the leaver's information, the leaver's predecessor's information and the leaver's successor's information.

4. *CLIENT LEAVE*

   When an AOM receives a *CLIENT LEAVE* message, it removes the client from the client list if there is a corresponding client.

5. *AOM MOVE*

The initial position of the server is decided by the hash result of its IP address, however, the distances between each AOM and its predecessor may be very different, and it causes the unbalance. To solve the balance issue, the server moves clockwise on the hash ring when it detects that its load is relative higher than its successor's load. An AOM checks whether the server's load is too heavy periodically. If the server's load exceeds its successor's load by a given value, the server will move to balance the system. The distance between a server and its predecessor on the hash ring should be inversely proportional to the node's load. Figure 3.6 is the formula of computing the movement. D_n means the distance between self and n, and L_n means the load of n.

$$\frac{D_{predecessor} - x}{D_{successor} + x} = \frac{L_{successor}}{L_{identity}},\ x = \frac{L_{identity} * D_{predecessor} - L_{successor} * D_{successor}}{L_{identity} + L_{successor}}$$

Figure 3.6: Movement for balance

The load should be defined according to the balance of the CPU load, utilization of memcached and the free capacity of memory of the server so we defined the load as the weight sum of $Load_{CPU}$, $Load_{utilization}$ and $Load_{memory}$. All of them are the integer between 1 to γ and the larger number means that the load is heavier. γ is the maximum value of load and we and we define γ as 5 in our work.

To define the $Load_{CPU}$, we first do an experiment to find the relationship between the CPU load and the throughput to find the suitable parameter to define the load and figure 5 is the result of the experiment. The experiment is designed as follows. We store and load 30000 objects (1KB) under different CPU loads and try to find the impact of the CPU load on throughput. The throughput goes down rapidly when the CPU load start from 0 to 3 then it becomes smooth. According to the result of this experiment, we define the $Load_{CPU}$ in figure 3.8. α is a argument of the thresh hold of CPU load and m is the slope of the line started from (0, 1) to (α, 5). In our work, α is set to 3 and m is 4/3.

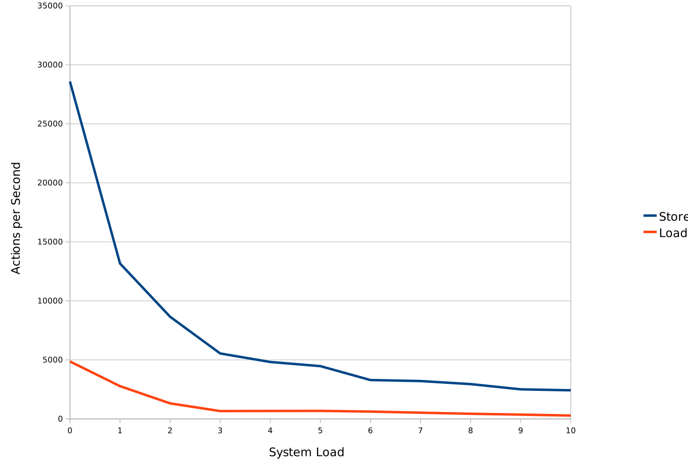


Figure 3.7: Performance under different CPU loads

$$Load_{CPU} = 1, \text{ if CPU load} = 0$$
$$Load_{CPU} = \gamma, \text{ if CPU load} > \alpha$$
$$Load_{CPU} = 1 + m * \text{CPU load, if } 1 <= \text{CPU load} <= \alpha$$

Figure 3.8: Definition of $Load_{CPU}$

$Load_{utilization}$ is defined by referring to the utilization of memcached. $Load_{utilization}$ is 1 when the utilization is 0% and it is γ when the utilization is 100%. Figure 3.9 shows the formula of $Load_{utilization}$.

$$Load_{utilization} = 1 + (\gamma - 1) * utilization$$

Figure 3.9: Definition of $Load_{utilization}$

The last member of load is $Load_{memory}$. When a percentage of server's free memory capacity is less than a given value β (We set β to 10% in our work), we raise the $Load_{memory}$ to at most 5. Figure 3.10 is the formula we defined.

The movement x must be positive, because if it can be positive or negative, an load error may occur. When a client wants to load an object, it expects that the data it got is up-to-date however, if an AOM moves forward then backward or in the contrast, the client may get the old object. We use an example to explain it in figure 3.11.

This limit means each AOM just cares about whether its load is too high but not too low. An AOM tries to reduce its load when it detects its load is too heave but does not

$$Load_{memory} = 1, \text{ if free} > \beta$$

$$Load_{memory} = \gamma - (\gamma - 1)/\beta * free, \text{ if free} <= \beta$$

Figure 3.10: Definition of $Load_{memory}$

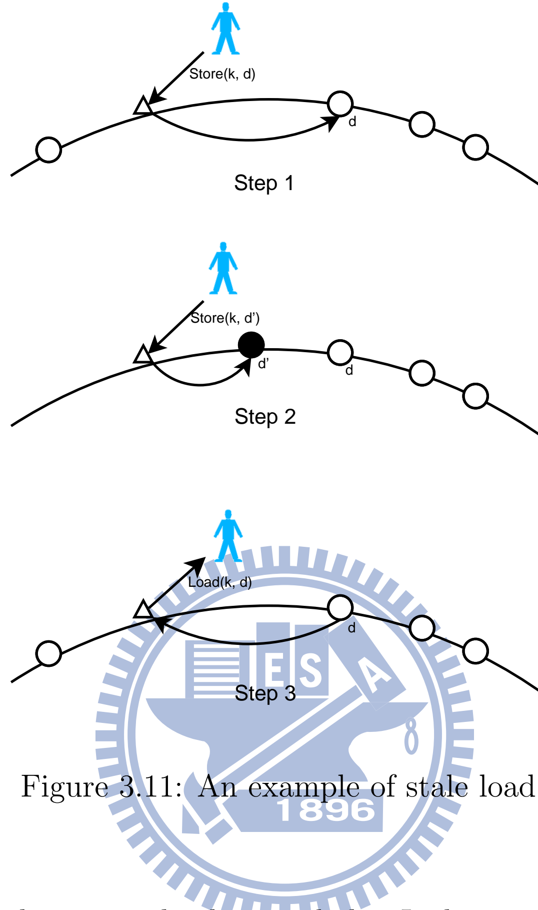


Figure 3.11: An example of stale load

do anything when it detects its load is too light. It does not mean we turn a blind eye to this case, because another node with relatively high will try to reduce its load when detect its load is too high.

The moving will not change the sequence of servers on the ring, so that the predecessor or successor of each AOM is not changed but it may cause the additional cache miss because the server an object is store to may move to other positions when the client tries to load.

6. *AOM RECOVERY*

   Servers may fail, and it causes errors. An AOM n tests if its predecessor and successor fails periodically and sends the *AOM RECOVERY* message to the another neighbor and it contains informations of its neighbor. When an AOM receive this message, it checks the status of the message. If the status is "Native", the AOM just passes the message to the next, otherwise it saves this information.

   When an AOM detects the failed neighbors, it tries to ask the informations of conrre-

sponding node according to the recovery information to fix its failed neighbor.

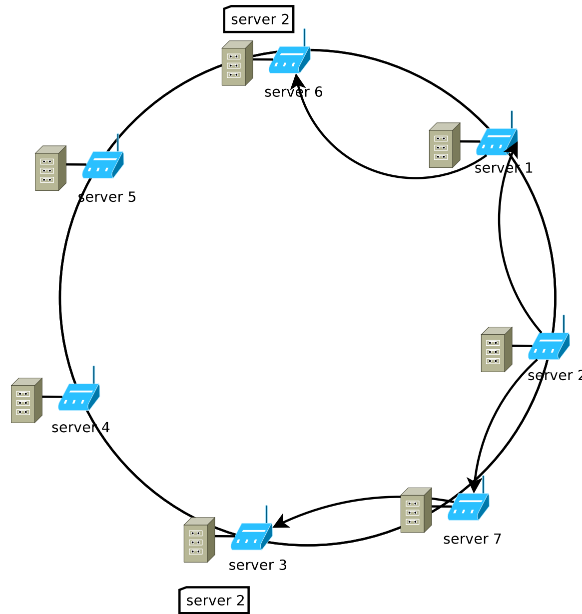


Figure 3.12: Failure recovery

3.2 Client

There are three components in our client, the memcached protocol, the distribution method and the AOM protocol. The memcached protocol is responsible for operating the memcached servers and it is used by the user API, the distribution method. The AOM protocol for clients synchronizes the servers in the server poll by communicating with AOMs. Figure 3.13 shows the architecture of servers and clients.

When a client starts, it sends the *CLIENT JOIN* message to any AOM in the network according to the AOM protocol, then it will receive the servers list of the network in a short period of time. When the client needs to load or store the object to a server, it runs the distribution method to decide which server is the target, then it uses the memcached API to visit the memcached server.

Our client runs the client part of the aom protocol to receive the server information. Clients will receive three types of messages *AOM JOIN*, *AOM LEAVE* and *AOM MOVE*. A client adds the server into its servers list when receiving the *AOM JOIN* message, deletes the server when receiving the *AOM LEAVE* message and updates the server when receiving the *AOM MOVE* message.

The distribution method is the API for users, and it is the normal consistent hash method with the argument of replications r when store and the argument tries t when load. The client

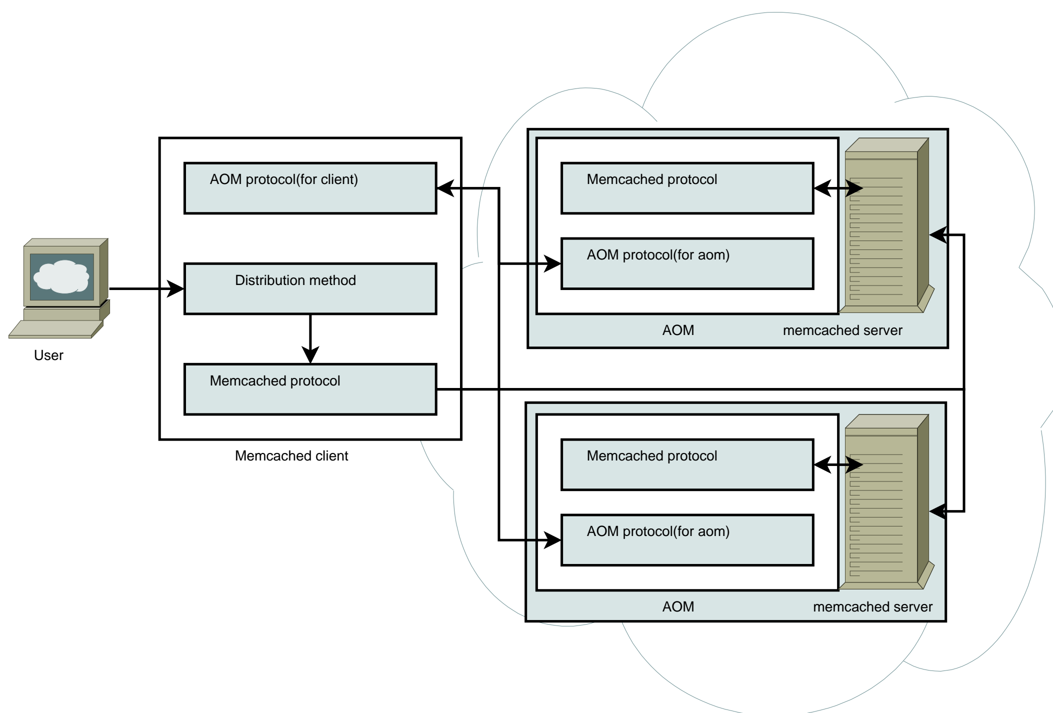


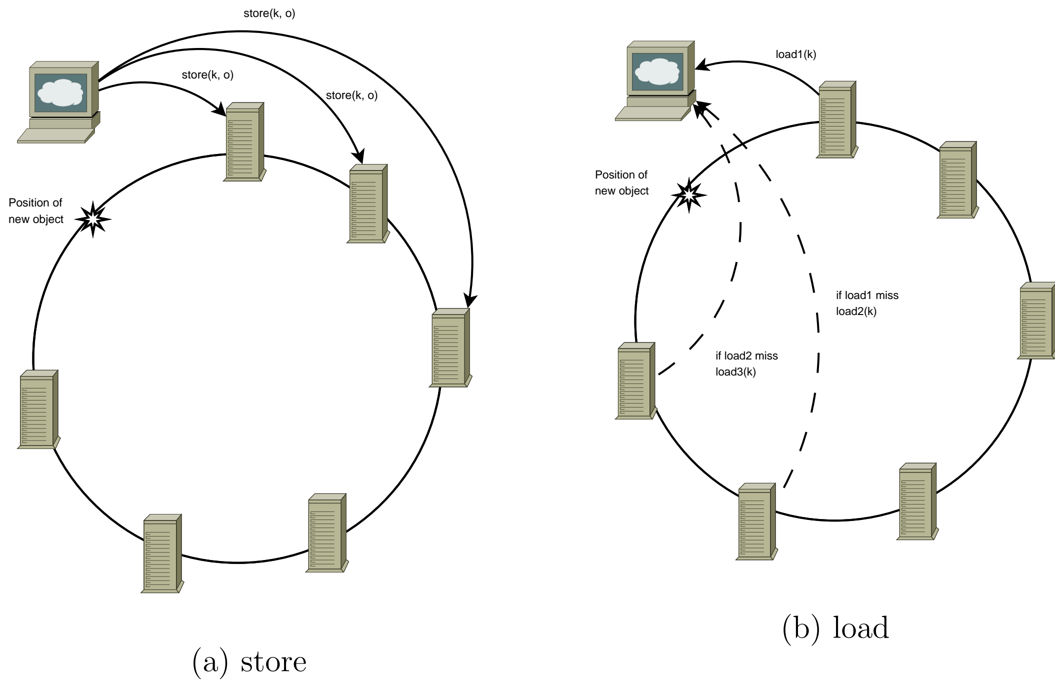
Figure 3.13: System architecture

stores an object into continuous r servers started from the server picked up by the normal consistent hash method along the searching direction and tries to load the object from the server picked up by the normal consistent hash method. If it loads miss, it tries to load the object from the previous node(predecessor) until cache hit or the number of times reaches the tries t. In our system, there are two cases that a cache miss will happen. The one is the object a client tries to load is not existed, and another is that the server stores the object has moved. We choose the predecessor as the next node to load because the movement of servers is counterclockwise on the ring. Figure 3.14 shows the example of distribution method of store and load suppose r and t are both 3.

3.3 Implementation of system

Both AOMs and clients are implement in C language with "libevent" library. The libevent API provides a mechanism to execute a callback function when a specific event occurs on a file descriptor or after a timeout has been reached. Furthermore, libevent also supports callbacks due to signals or regular timeouts. Memcacehd uses libevent to develop the part of networking, too.

The AOM protocol relies on the exchanging of the message that we mentioned before. The

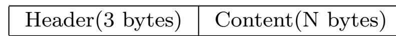


(a) store

(b) load

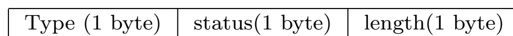
Figure 3.14: Distribution methods

message format is in figure 3.15. There is a header(figure 3.16) and a content in a message. The header describes the type, status of the message and length of the content. Type points the action of the producer, and it can be join, leave or something we mentioned before. Status defines the direction if the message must be pass. Length of message writes the size of content in the message. The content writes the position, load and IP of the nodes even more. These nodes can be the producer of message or the neighbors of the producer. The content is composed by the minimum unit called "content token"(figure 3.17). Every message ends by "\ n".



Header(3 bytes)	Content(N bytes)

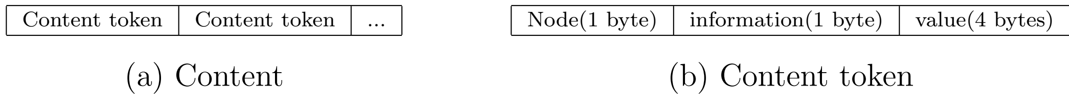
Figure 3.15: The format of a message



Type (1 byte)	status(1 byte)	length(1 byte)

Figure 3.16: Header

Once the AOM or the client receives a message, it parses the message and forward or backward the message according to the column "status" in the header, then it handles the message according to the "type" in the header and the content of message according to the AOM protocol we talked in the previous chapter.



(a) Content (b) Content token

Figure 3.17: The format of content and content token.

The communication of AOM and memcached depends on the ”stats” in memcached protocol. An AOM can get many informations of memcached by ”stats” but it just cares about the ”bytes” row and the ”limit_maxbytes” row at present. The first points out the used memory and the second points out the total memory.

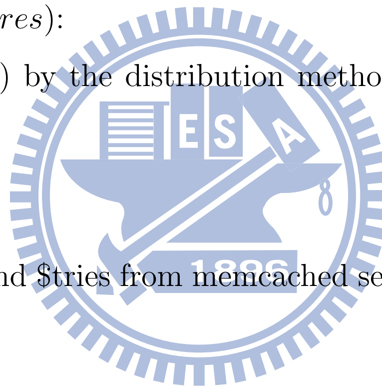
The user API of our client is listed as follows. User must call *start*() at begining and *stop*() in the end.

1. *start*():

   Initial and start the client.

2. $store(key, data, length, stores)$:

   Store the object(key, data) by the distribution method of store with the argument of $stores.



3. $load(key, item, tries)$:

   Load the item with $key and $tries from memcached servers by the distribution method of load.

4. *stop*():

   Stop the client.

Chapter 4

Performance Evaluation

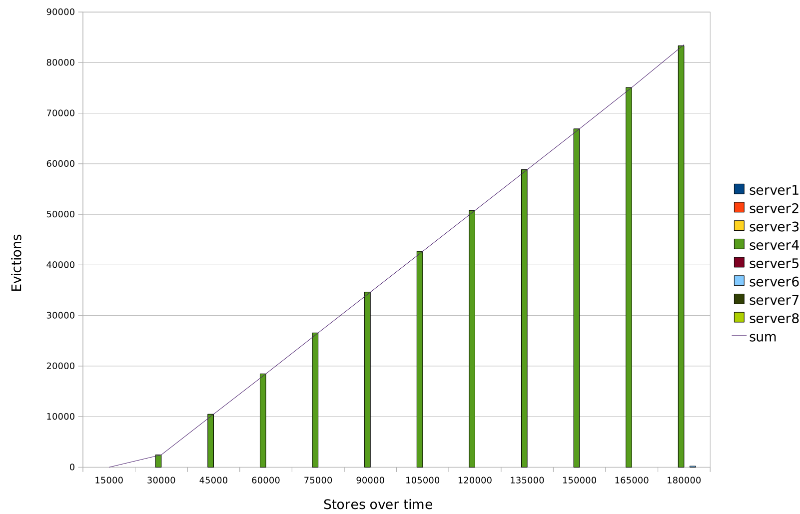
4.1 Test Bed

Each memcached and AOM are performed on the virtual machine(VirtualBox-3.3) in Intel Core2 2.4GH system on Linux(Kernel 2.6.28-15).There are total 3GB memory installed in the host and each vm takes 128MB. The version of memcached is 1.2.8 and libevent is 1.4.10.

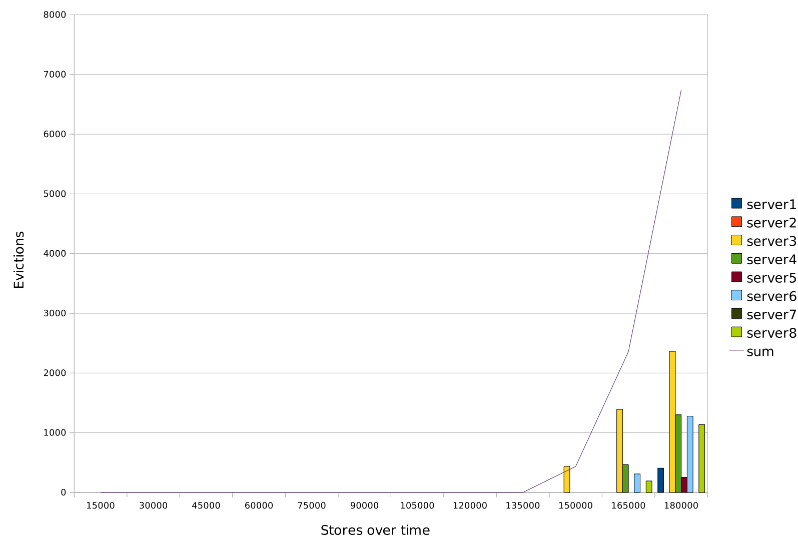
4.2 The Impact of Evictions on Hit Rate

The evictions are caused by the insufficient of memory space of the server. It comes an eviction when a new object is stored to a server that the server has used out its memory and the evictions will cause the cache miss. AOM protocol detects the memory and it tries to reduce the load of the server when the server runs out the memory. We first monitor the numbers of eviction in a traditional system and in our system. We use 8 servers and each of them uses 32MB memory as the cache(256MB at all).

We use a data set contains 180000 different keys with uniform distribution. Every object is 1800 bytes and the replication r is set to 1, therefore, the total size of all objects is 324MB and it exceeds the total memory capacity for a little. Figure 4.1 shows the result of this experiment.



(a) Traditional method



(b) AOM

Figure 4.1: Evictions caused by traditional method and AOM

Even though when the number of objects is small, there are evictions in the server 4 when using traditional method because the results of distribution of objects are gathered in the server 4 and the number of evictions of server 4 grows to more than 80000 at last. We have not seen this situation in an AOM system. The number of evictions is 0 until there is no free memory space in all server because AOMs balance the server.

This experiment proves that the AOM's low memory detection reduces the evictions of the system and it is helpful for increasing the hit rate because the evictions cause cache miss when we try to load the objects from servers that the object has been evicted . Figure 4.2 shows the number of cache miss. Two cases are considered, sufficiently and insufficiently total

memory capacity(8 MB, 256 MB). We use Zipf distribution with exponential parameter 1.0. The number of total objects is 18000(Each object is 180000 bytes) and load 18000 times with cache is empty at beginning.

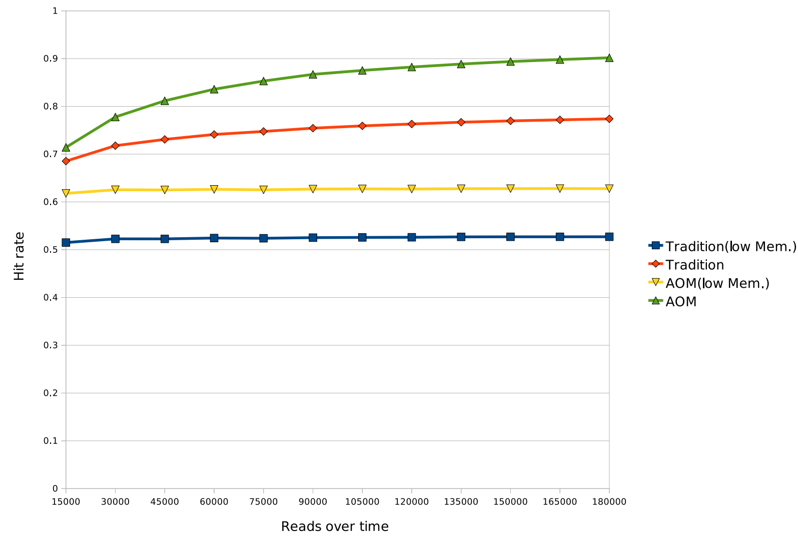


Figure 4.2: The cache miss caused by the evictions.

The result shows that we can have the higher cache hit rate when using our system. Both of the hit rate of traditional method and our method grow up progressively because the number of cached objects becomes as time going.

4.3 Replication

Our "replication" means the number of times that the clients store each object to the servers, and the our "tries" means the times a client tries to load an object from the next server if it loads miss from the present server. In the common use, both "replication" and "tries" are set to 1 to reach the better performance but higher replication provides higher hit rate because some servers may leave or move.

We tries to kick the servers when a client loads a series of object then measure the relationship between replication and the hit rate. The result shows in figure 4.3. The number of objects is 180000 and the size of each object is 1800 bytes. To eliminate the cache miss caused by eviction, we increase the memory space of each server to 48MB.

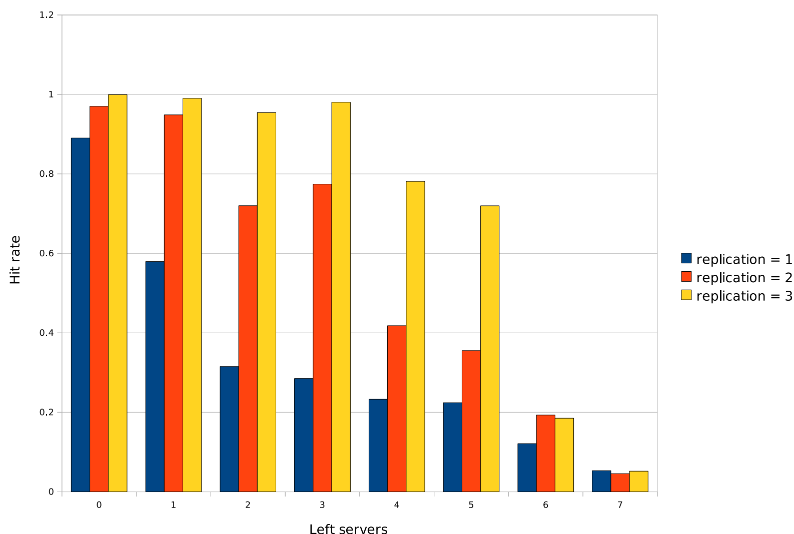


Figure 4.3: The relationship between replication and hit rate.

The hit rate is very low when there are servers leaving the network when replication is 1. If we set the replication to 2 or more than 2, the hit rate is 80% even if there are half of server left. We can get the hit rate of 80% even when 5 servers leaves when the replication is set to 3, however the overhead grows with the replication and too high replication may decrease the system performance.

4.4 Performance on Different Machines

We measure the speed performance when the servers are under different system load because memcached may be performed on different server with different ability. We store and load 18000 objects and measure the performance. We distribute objects Zipf distribution to simulator the behavior of objects in the web of the real world, and the exponential parameter of Zipf is set to 1.2.

The impact of system load on performance has been described before, when the average load exceeds 3, the performance goes down obviously. So we choose specific numbers of servers and let them work at the high workload, then measure the performance. The result shows in figure 4.4.

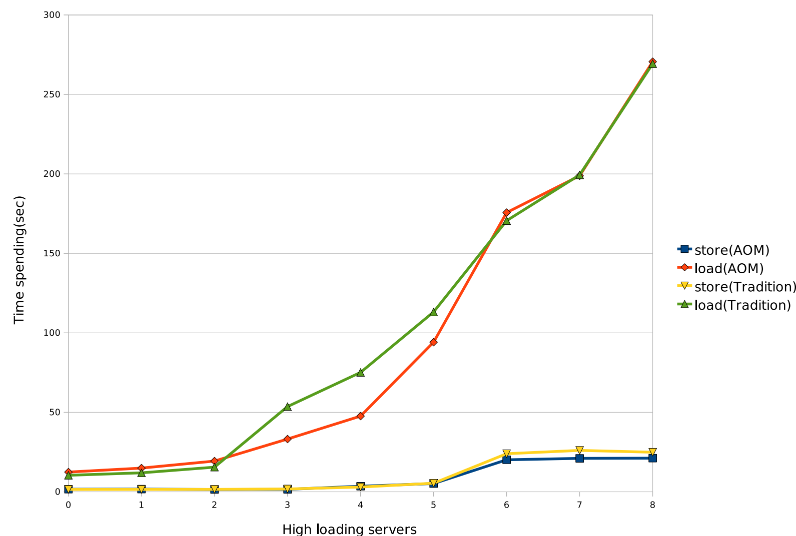


Figure 4.4: Performance when some server are under high workload

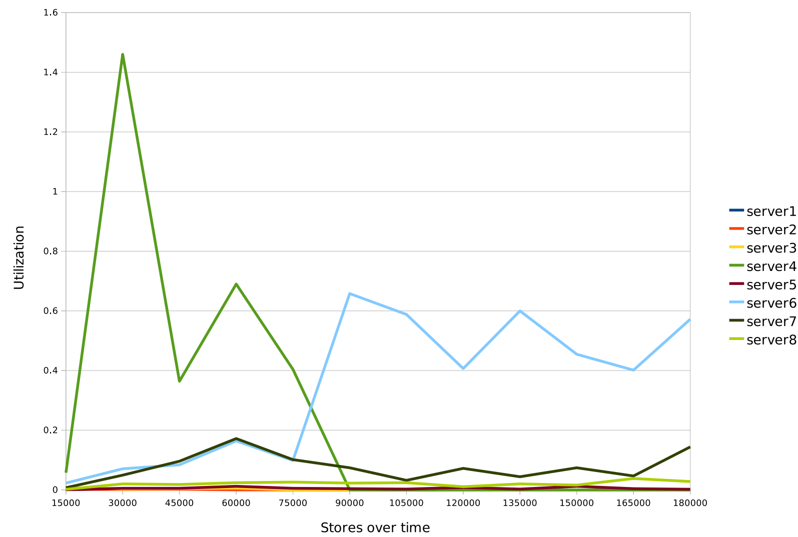
In figure 4.4, the time spending of load of traditional system increases quickly when the number of high loaded sever is 3, that may mean the server 3 is a high loaded server, but the client chooses it to store the objects aplenty. The time spending increases comparatively moderate in an AOM system because an AOM system will tries to reduce the workload of the servers with high workload.

It seems that the CPU resource is comparatively not important when load. The time spending of load increases very slow even if there are more servers at high workload.

4.5 Utilizations of Servers

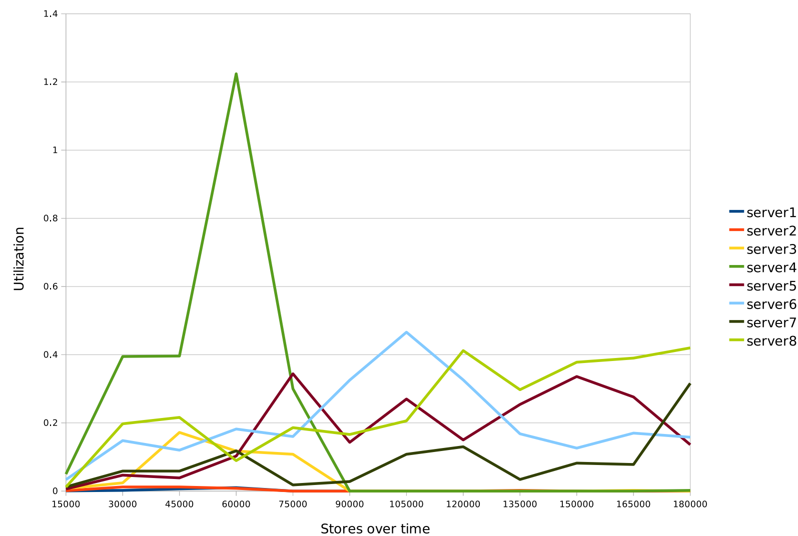
In the real world, objects in the web distributes in Zipf distribution. It causes the situation that the utilization of the server might be specially high if the server is the web cache server.

We compute the utilization of memcached server by storing objects in Zipf distribution with the exponential parameter is set to 1.2 to the server. The number of objects is 180000 and we kick 4 servers after storing 50000 objects to increase the workload to see the variations of utilizations of the servers. The result shows in figure 4.5.



1896

(a) Traditional method

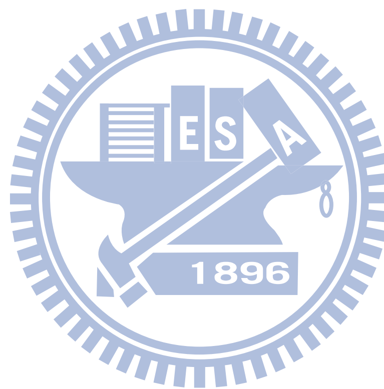


(b) AOM

Figure 4.5: Evictions caused by traditional method and AOM

The utilizations of servers are more average in the AOM system than in traditional method

because the AOM balances the utilization of a servers. Besides, the sum of utilizations of servers in AOM system is higher than in traditional method, that means the system performance is better in the AOM system.

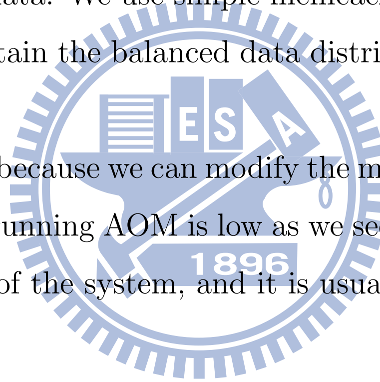


Chapter 5

Conclusion

In this work, we design a distributed architecture of memory cache server and client. Using our system, we can transparently change the members of memory servers and client. The load balance issue can be solved by the AOM, so the clients can use simplest and fastest distribution method when store or load the data. We use simple memcached client and memcached server which is not be modified and obtain the balanced data distribution and good performance on our system.

This architecture is scalable, because we can modify the mechanism of load balance method when we need, The overhead of running AOM is low as we see in the previous chapter because it is just a independent process of the system, and it is usually in sleep.



Bibliography

[1] P. Valduriez. "Parallel database systems: The case for shared-something". In Proc. of International Conference on Data Engineering, Pages: 460 - 465, 1993.

[2] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim; L. Matkins, Y. Yerushalmi. "Web caching with consistent hashing". In Proc. of International Conference on World Wide Web, Pages: 1203-1213, 1999.

[3] D.N. Serpanos, G. Karakostas, W.H. Wolf. "Effective caching of Web objects using Zipf's law". In Proc. of International Conference on Multimedia and Expo, Pages: 727 - 730, 2000.

[4] G. Karakostas, D.N. Serpanos. "Exploitation of different types of locality for Web caches". In Proc. of International Conference on Computers and Communications, Pages: 207 - 212, 2002.

[5] J. Bonwick, S. Microsystems. "The Slab Allocator: An Object-Caching Kernel Memory Allocator". In Proc. of USENIX Summer, 1994.

[6] K.Y. Wong. "Web cache replacement policies: a pragmatic approach". In Proc. of International Conference on Network, Pages: 28 - 34, 2006.

[7] C. Schindelhauer, G. Schomaker. "Weighted distributed hash tables". In Proc. of ACM Conference on Parallelism in algorithms and architectures, Pages: 218 - 227, 2005.

[8] S. Lei, A. Grama. "Extended consistent hashing: an efficient framework for object location". In Proc. of International Conference on Distributed Computing Systems, Pages: 254 - 262, 2004.

[9] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, Hari Balakrishnan. "Chord: a scalable peer-to-peer lookup protocol for Internet applications". In IEEE/ACM Transactions on Networking, Pages: 17 - 32, 2003.

[10] R. Lang, Z. Deng. "Data Distribution Algorithm using Time based Weighted Distributed Hash Tables". In Proc. of International Conference on Grid and Cooperative Computing, Pages: 210 - 213, 2008.

[11] J. Petrovic. "Using Memcached for Data Distribution in Industrial Environment". In Proc. of International Conference on Systems, Pages: 368 - 372, 2008.

[12] http://www.danga.com/memcached/

[13] http://developers.facebook.com/opensource.php

[14] http://developers.facebook.com/opensource.php