DISTRIBUTED WEB CRAWLING: A FRAMEWORK FOR CRAWLING OF MICRO-BLOG DATA

Jie Xia[1, 2], Wanggen Wan[1, 2], Renzhong Liu[1, 2], Guodong Chen[1, 2], Qing Feng[1, 2]

[1]School of Communication and Information Engineering Shanghai, China
[2]Institute of Smart City, Shanghai University, Shanghai, China
18117348248@163com,wangwg@staff.shu.edu.cn

Keywords: Web Crawling, Distributed Framework, Social Network Analysis, Raspberry Pi, Micro-blog

Abstract

These days' social networks have attracted people to express and share their interests. We aim to monitor public opinions and other valuable discoveries by using the data collected from social network website Sina Weibo. This paper present a distributed web crawler framework called SWORM, which runs on the Raspberry Pi (cheap card-sized single-board computer) for fetching the micro-blog data and overwhelms the traditional web crawlers on efficiency, scale, scalability and cost. The framework can easily be extended according to the specific needs of the user with the help of some simple python scripts. This paper first propose a model for micro-blog network to confirm what and how our crawler will crawl from social website. Secondly it will introduce the implementation details of the whole distributed system and finally will present experimental results. We ran some crawlers within our framework on the Raspberry Pi and stored the obtained resources in Shared MongoDB which is a category of NoSQL. Experimental results demonstrated that the use of distributed framework can greatly improve the efficiency and accuracy for collecting data.

1. Introduction

Nowadays, micro-blog has been a new social networking service on the basis of web2.0 technology. People can send and receive messages instantly and conveniently with the micro-blog. With the increasing number of micro-blog users, micro-blog is becoming like a tank which is filled with a vast array of precious information. There is no doubt that we can get useful discoveries from the huge data inside micro-blog in this era of big data.
However, two obvious characters of micro-blog data: fast-updating and large-scale must be noted. With reference to these two features, the fast crawling and efficacious storage is requested. It is widely recognized that the most effective method to collect large amount of online data, is to design and apply computer programs to extract information from large-scale web pages. Thus, an effective and small-sized crawling distributed framework can be used individually or collaboratively [1]. The main idea of a distributed computing system is to obtain extra processing power by connecting idle computers to the internet. Although the earlier web crawler structures are still relatively efficient in fetching data from websites. However, Most of the existing crawling frameworks [2-4] are large-scale and it is inconvenient to apply according to the user's requirements. Which means they lack flexibility and the number of crawl applications cannot be added or reduced, and requirement of the structure of the crawl application is rigid. However, some of them can only work in a single computer and lead to the low efficiency. So in this paper, we present a distributed crawling framework which is able to run on the Raspberry Pi platform. Compared with others, our framework is distinguished from the following three aspects: high efficiency, high scalability and low cost.
High efficiency: it is important aspect to be presented in the distributed framework. We have applied the consistent hashing algorithm [5] to solve the critical issues influence the efficiency of load balancing in distributed systems. With the help of algorithm, each node in the cluster can operate with the highest efficiency. Furthermore, the cache redistribution could be minimized when the number of node changes which can be disastrous since the originating content servers are flooded with requests from the nodes. Hence consistent hashing is needed to avoid swamping of servers as well as improve the stability of the entire system. In the framework, The Bloom filter [6] a spaced-efficient probabilistic data structure that is adopted to test whether a job URL is a member of the set that contains the job URLs which has to be visited for the purpose of eliminating the chances that a newly crawled web page is a near-duplicate of a previously crawled web page.
High scalability: Users don't require to be familiar with the inside structure of the framework but put the particular crawler with the python script in the light of their own requirements into job cluster. On the other hand, it would be easy to add machines to cluster which require no change the program itself. In addition, the framework supports two crawling modes the API-based crawler and the webpage-access crawler at the same time. These three sides help expand the scalability and improve flexibility.
Low costs: Application programs run on the Raspberry Pi platform which is a cheap card-sized single-board computer. Thus, we can expand the worker nodes of job cluster with the Raspberry Pi other than the common computers to reduce the costs as well as other resources consumption.
The rest of this paper is organized as follows: Section 2 present the model of the Sina micro-blog network. Section

3 present the concrete structure of our framework as well as the algorithm design. Section 4 present implementation of proposed distributed framework. Section 5 present experimental results and analysis. Section 6 summarizes our work and presents the future work.

2. Topology of Micro-blog Website

The target of micro-blog crawlers is micro-blogger's data while web crawler's is a web page. However the storage structure of data in some social networks is different from other ordinary websites. Micro-blog network is united by countless micro-bloggers, and every micro-blogger is connected with many web pages. For web crawler, crawling starts from an original web page. So the crawler of micro-blog can't be consistent with the web crawlers. Figure 1 shows the rough structure of micro-blog networks and elaborate relationship between the micro-bloggers. For the micro-blog crawler, it starts with a specific user with crawling goals and then his fans, followers and blogs. So our crawler just fetch the data containing the users' information we prefer.

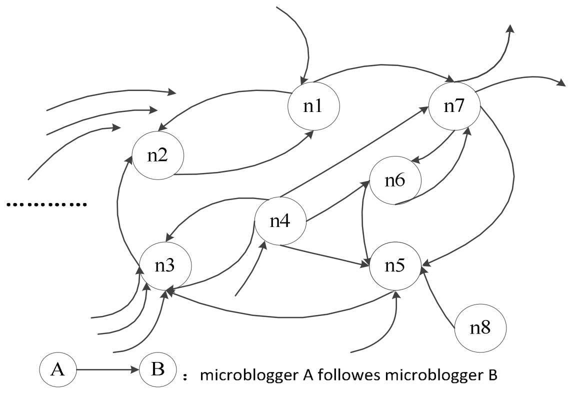

Figure 1. Topology of micro-blog websites.

2.1 Depth Control

It is true that the mechanisms of micro-blog crawler is a process of iteration in depth. It's likely to fall into an endless working when crawling the sites similar to Sina micro-blog, so the depth of iteration control must be considered. We can set a parameter named "maximum depth" on behalf of the located depth of the original target. We set the depth of micro-blogger as 0, then all his followers or fans share the same depth 1, similarly all his followers' of followers' share the same depth 2......Our crawler will stop working once the depth exceeds the "maximum depth" we have set beforehand.

2.2 Breadth Control

In order to obtain large coverage (fetch data as much as possible), our crawler employ the breadth-first algorithm. In such social networking sites, every user may be considered as a follower or a fan. For example, in the figure 1, n1 follows n2, then n1 is n2's fan and meanwhile n2 is n1's follower. Thus, if we are eager to extend the crawling breadth, we just need to crawl either the fans or the followers. Given some famous people may have countless fans, we choose to crawl just followers. We may crawl 100 common micro-blogger with one node (Raspberry Pi) in job cluster during one hour. On the other side, when we choose to crawl fans, it may not be enough for crawling just one user who has 1000000 fans. So with this choice, the crawling breadth can be extended.

3. System Design

3.1 Architecture of the Framework

Our framework separate the web crawlers and the core distributed system into two parts and enable paralleled data collection. So common users only need to write typical crawler scripts on the basis of their demands instead of learning the core structure. Figure 2 shows an overall architecture for the proposed data collection and data storage flow, which mainly consist of three key components: Master, Worker(Crawler), and the Shared MongoDB. In the whole cluster, when a task is submitted, the Master and Worker will respectively start JobMaster and JobWorker. For a particular job, each JobWorker will send feedback signal to JobMaster, and JobMaster begins running the job until all the JobWorkers complete the initialization, a URL queue will be constructed when finish a job submission. The URL queue stores the URLs of pages which are about to be crawled. When a JobWorker finishes fetching a page, it will return some new target URLs that will be put into the URLs queue. The JobWorker also take a URL from the queue to implement the next crawl. At the same time, every JobWorker has a bloom filter model for deduplication to eliminate the URLs which have been visited. However, the JobWorker in a specified job is multi-threaded downloader, and the user should set the suitable number of threads at the beginning.

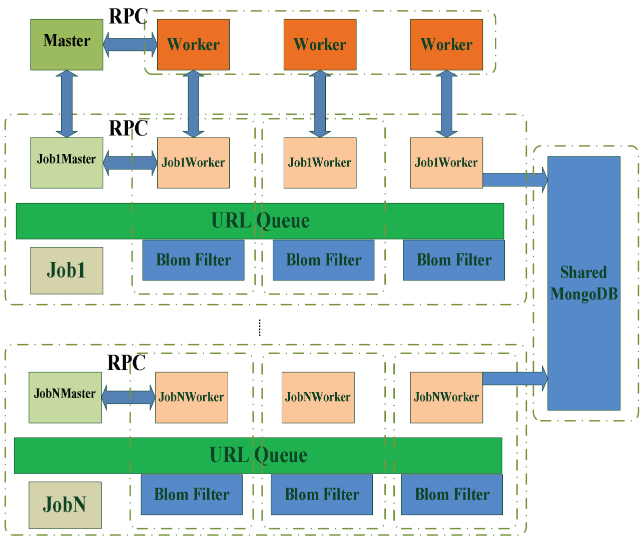

Figure 2. Overall architecture of the framework.

3.2 The Central Node Server

The central node server refers to the Master in figure 2. Master is the control center of the whole cluster. More specifically, the central node server contains one communication model, one job distribution system, the node control manager and the URL queue, as shown in Figure 3.

The communication model is responsible for the communication with all workers in the distributed cluster which play a fundamental role in the structure. This model employs the RPC protocol (Remote Procedure Call Protocol) and it makes easier for developing the network of distributed application.

The cluster holds a number of tasks, and these tasks are distributed into the cluster and managed by the job distribution system. The system distributes the different tasks to the appointed nodes based on the user's commands. It can adjusts the situation of cluster to realize loading balancing automatically. The working efficiency of a node could be greatly reduced if it operates many parallel issues. Thus, the number of jobs in one node must be appropriate. On the other side, the system take into account the duty of managing the whole jobs in the cluster. For example, the administrators inquire about the running state of a job for which nodes are running the job, e.g.

The node control manger supervises all nodes of whole cluster. It helps in adding or removing a node from cluster and monitoring the nodes' condition. All the information of one node such as the jobs running on it, the crawling speed, and the numbers of wrong and correct crawling pages or the log information of each node can be access to the administrators.

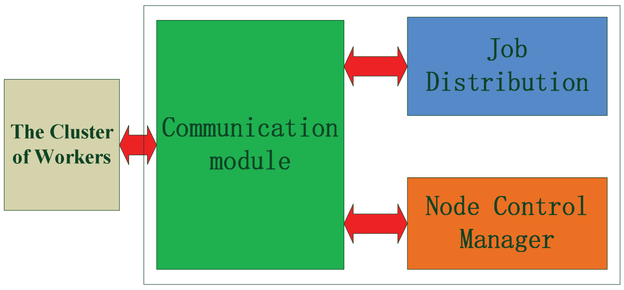

Figure 3. Structure of the Master.

3.3 The Worker Nodes

The worker nodes implement the concrete tasks of crawling. In figure 4, it contains several parts such as the communication model, watcher model, Job loader, URL controller, Job crawler, logging and login model. The communication model is responsible for the tasks of transferring information with the master or other worker nodes. Logging is the model that records real-time state of crawling job, some websites such as the Sina Micro-blog need the users to log in to get access to the page. The login model is in charge of the assignment. The watcher should monitor the condition of jobs as a worker node could carry out more than one job and the job loader starts the local jobs or the remote jobs transmitted from the master according to the command of watcher model. Next, we will talk about the URL controller and the Job crawler (crawler of Sina Micro-blog) in detail.

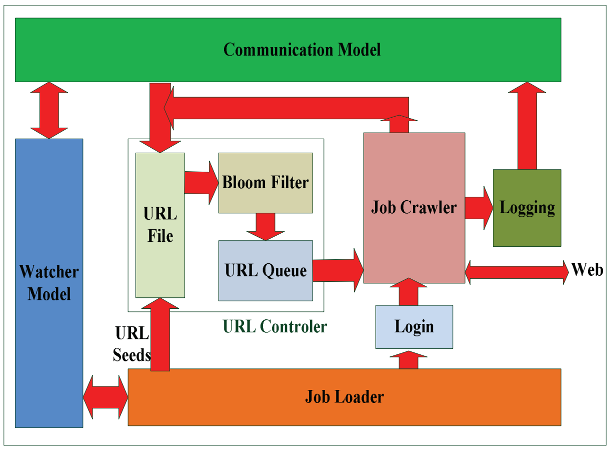

Figure 4. Structure of the Worker Node.

3.3.1 URL Controller

The model URL Controller is a system that stores and distribute the target URLs. It consist of three parts URL file, Bloom Filter and the URL queue. The URL queue is a set of target URLs which is about to been fetched which is shared by all the nodes. From figure 4, we can observe that the URL has two sources. One is local and the second is other nodes in the cluster. How can we determine the distribution of the URL to the target node? How can we realize load balancing and improve working efficiency? If the system can't schedule the tasks well, some nodes may be busy with crawling, while others are leisure. As a result, the whole working efficiency is extremely low. Thus the URL queue adopts the consistent hashing scheme to implement load balancing well [7]. Furthermore, the pages that have been fetched must be excluded. It needs our system to test whether the URL is a member of the URLs which have been visited. So the data structure of Bloom Filter is utilized to improve efficiency of the testing. Next, we describe consistent hashing and Bloom Filter in more detail.

A: Consistent Hashing Algorithm

Our system is based on the consistent hashing, a scheme developed in a previous theoretical paper [8]. Here, we motivate consistent hashing and describe its simple implementation. First, we will outline its theoretical justification, and then describe our implementation of the algorithm.

We can choose some standard base hash function that map strings to the number range [0-N]. Dividing by N, think of it as a hash function that maps to the range [0-1], which can in turn be thought of as the unit circle. Each URL is thus mapped to a point on the unit circle. At the same time, mapping every node in whole cluster to a point on the unit circle. Now assign each URL to the first node which point it encounters moving clockwise from the URL's point. An example is shown in Figure 5.

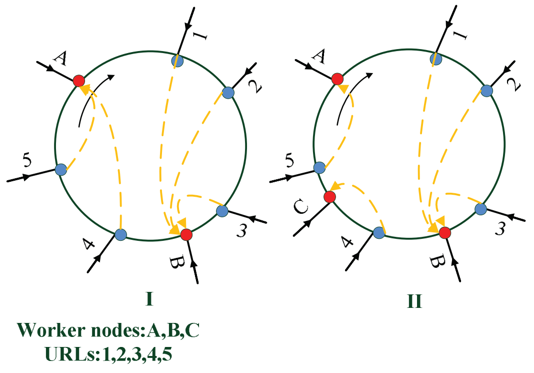

Figure 5. An example for consistent hashing.

(I) Both URLs and nodes are mapped to points on a circle using a standard hash function. A URL is assigned to the closest node moving clockwise around the circle. Items 4 and 5 are mapped to cache A. Items 1,2 and 3 are mapped to cache B. (II) When a new node is added the only URLs that are reassigned are those closest to the new cache going clockwise around the circle. In this case when we add a new node only items 4 move to the new node C. Items do not move between previously existing nodes.

Consistent hashing is easy to implement. All the points of nodes can be stored in a binary tree, and the clockwise successor to a URL's point can be found (after hashing the URL point) via a single search in the binary tree. However, some Consistent Hashing implementation method also uses the idea of a virtual node. When using a general hash function, then the distribution map location server is very uneven. Therefore, the use of virtual nodes thought it is quite important to make a small number of copies of each node point--that is, to map several copies of each node to different random points on the unit circle. So the uneven distribution can be suppressed in some level, to minimize the redistribution for adding or removing nodes. This produces a more uniform distribution of URLs to worker nodes.

We have designed a class HashRing to implement the consistent hashing algorithm. And the class consists of several core methods.First the method generate_circle( ) which helps hash the worker nodes using the md5_constructor to construct the unit circle according to the nodes' weights. The different weights determine the number of virtual nodes talked last page. At last, we can get a dictionary ring stores the <key,node> pairs and a list contains the sorted keys.Another fundamental method is iterate_nodes( ). Given a string key (the URL object) it returns the nodes as a generator that can hold the key. The generator iterates one time through the ring starting at the correct position. So the method support us distribute a URL to a worker nodes efficiently to achieve good load balancing result.

B: Bloom Filter

If you need to judge whether an element is in a collection, we usually make the practice to save all the elements down, and then we can get the result by comparing with set that has saved.

Bloom filter is a method of using a hash function to map an element to a point on a filter array with length m. When this point is 1, then the element is in the set, on the contrary not within the set. False positives are possible, but false negatives are not. Elements can be added to the set, but not removed (though this can be addressed with a counting filter). The more elements that are added to the array, the larger the probability of false positives. We can extend the method to employ k hash functions to map the element to m points and then identify all these m points to determine the result. We can calculates the minimum number of bits (m) represents an element in filter array and number of hash functions(k) given a number of entrants (capacity) and the desired error rate (false positives) with the equation 1 and 2.

$$m = \frac{capacity \cdot \log(error)}{\log(\frac{1.0}{2.0^{\log(2.0)}})} \tag{1}$$

$$k = \frac{\log(2.0) \cdot \mathrm{m}}{capacity} \tag{2}$$

For example, if we set the capacity 3000 and the error rate 0.01, then the (m, k) is (28756, 7).

Overall, within the uniform model, the sequence diagram of how the worker node operates under guidance from SWORM distributed framework is shown in Figure 6.

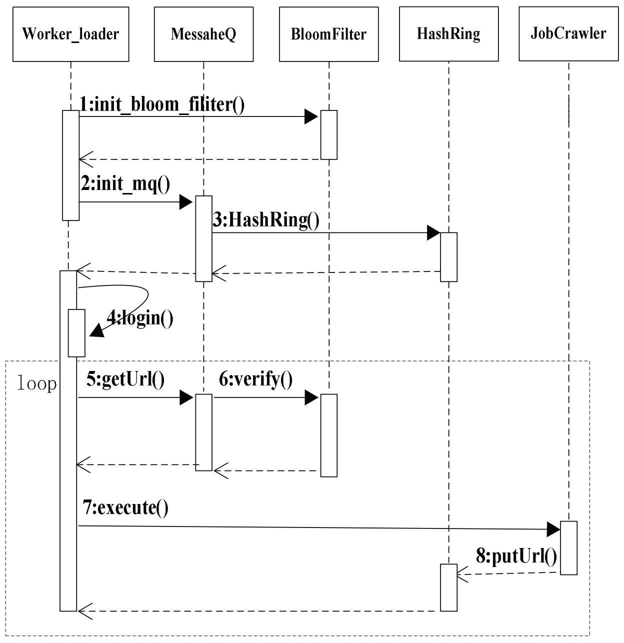

Figure 6. The sequence of the running of worker node.

3.3.2 The Crawler of Sina Micro-blog

The crawler of Sina Micro-blog is core model of the system. It completes the work of data fetching under the SWORM framework. We focus on the realization of micro-blog crawler as we need data of micro-blog. The users can bring about their own crawlers on the basis of their real needs and then run these crawlers on the SWORM. We have designed a crawler covering the whole

Sina Micro-blog website that get all the basic information, relationship information of micro-bloggers (including the followers and the fans) as well as all the information of micro-blogs have delivered.

Figure 7 shows the overall working flow. First, it will read the configure file to initialize the system, such as the number of threads, port of communication, and injects the seed URLs to URL queue, e.g. Then the crawler will access to the site according to the user name and password in the configure file by simulating the browser to get permission. During a crawling loop, if the URL queue is empty which means no more followers to be crawled, the crawler will come to the end. If it is not, it will start fetching pages of the seed URLs. There are two phases to complete the fetching work. One is downloading the HTML pages and another is parsing the HTML to fetch the valid data and merge and save it to MongoDB. The fetched data includes micro-blogger's nick name, his micro-blogs and comments, fans' name_list, followers' name_list and so on. When appearing failures or faults during downloading the HTML pages, the crawler will rollback. If no faults appear, it will start parsing the HTML. And the new valid URLs parsed from the pages who are under depth control and never been crawled will be added into the URL queue for updating. And it will make several backup of the valid URLs in case of crawling failure.

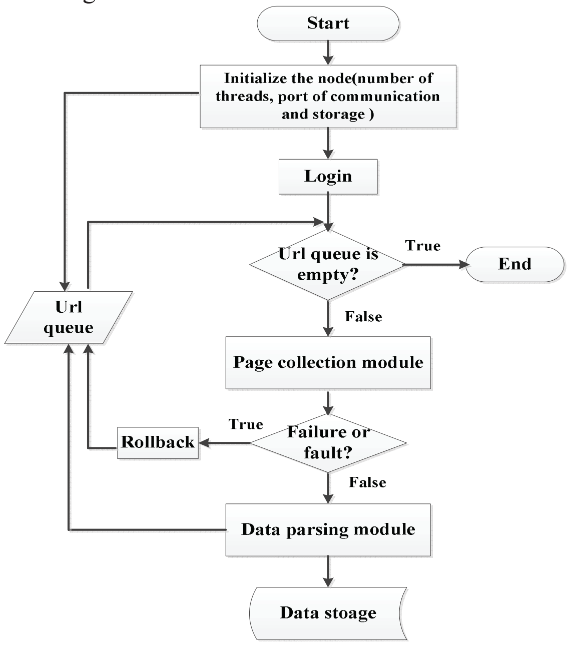

Figure 7. The workflow of the crawler.

3.4 Sharded MongoDB

We have applied Sharded MongoDB to store a great of collected data in NoSQL format for the convenience of data reuse and data processing, as shown in figure 8 [9]. In addition, the NoSQL databases show strong performance in reading and writing data, especially the large amount of data. It is unnecessary to establish the table with specific data fields in advance. The users can store the data as their own format at any time to overcome the looseness of fetched data from different sources topics in websites. We partition the data and store the data in the different MongoDB instance in the cluster and this process is almost transparent to application.

From the figure 8 we can see that there are four components: mongos, config server, shard, application driver. All requests are coordinated through mongos and it is a request distribution center, which is responsible for forwarding the data request to the shard server. Config server, as its name implies, stores all the meta-information of database such as the routing and shard information. Last, shard, it stores the data on several different machines. As long as we set up the rules of fragmentation, and then mongos instance automatically forwards the requests of data operation to the corresponding shared machine. Through the operation of the database corresponding data operation requests to the corresponding slice machine. It must be sure that the distribution of these requests is uniform. The whole Shared MongoDB Cluster may consist of a group of computers that deploy config server instance runs on port 27019 and deploy mongos instance runs on port 27017 at the same time.

In combination, the SWORM framework, the web crawler and the shared MongoDB Cluster constitute the whole proposed paralleled data collection and data storage system.

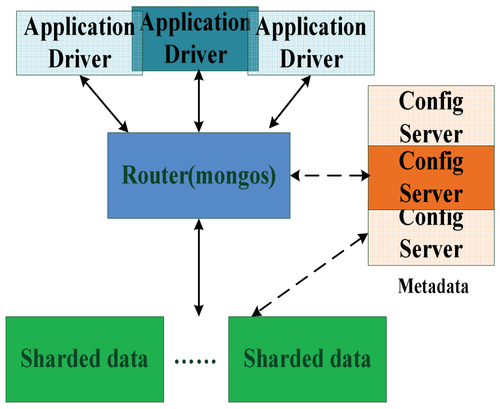

Figure 8. Architecture of Sharded MongoDB.

3.5 Performance Evaluation for the Proposed SWORM framework

A: Experimental environment

We deploy the SWORM framework for experiment with 4 worker nodes shown in figure 9. The two nodes are Raspberry Pi B+ which includes an ARM1176JZF-S 700 MHz processor, VideoCore IV GPU, and was shipped with 512 megabytes of RAM. And other two are computer with the CPU of Intel i3 M350 and 4GB of RAM, the Operating System: Windows7 x86_32. And for each worker node the maximum number of threads is 4.

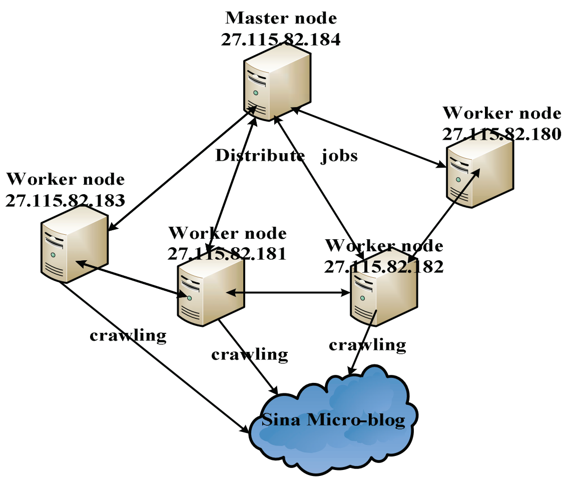

Figure 9. Experimental deployment.

B: Results and analysis

In our experiments, we crawl Sina micro-blog with hundreds of million active users. First, we set two seed URLs (the homepage of two micro-bloggers) and 2 threads of each worker node in the configure file. And entire distribute system runs almost 15 hours, and crawled about 1,334,528 URLs, all information (fans, followers and the micro-bloggers) of about 56,363 micro-bloggers. The statistical results show in table 1.

Table 1. Statistics of SWORM based crawler

total crawling time	54,312s
number of URL seeds	2
total number of URLs	1,334,528
number of URLs crawled successfully	1,110,951
number of failed URLs	223,557
speed of getting pages(URLs/s)	20.5

From table 1, we can learn the crawling correct rate is about 90%, which means our framework can acquire a good stability. We also see that the system can get 20.5 pages per second. And the crawling speed has something to do with the number of nodes and number of threads in each node.

And we have done another experiment to evaluate the impacts on speed with different number of threads. The figure 10 shows the result. We conclude that the crawling speed changed a little when the number of threads is greater than 8. The speed can be improved in a certain level with the increase of number of threads. However, the speed is also affected by some other factors such as the bandwidth limitation of network. What's worse, once the crawling speed is greater than the limitation, the Sina site will prohibit you to visit pages by your IP number. Based on these factors, we can explain the phenomenon shown in figure 10.

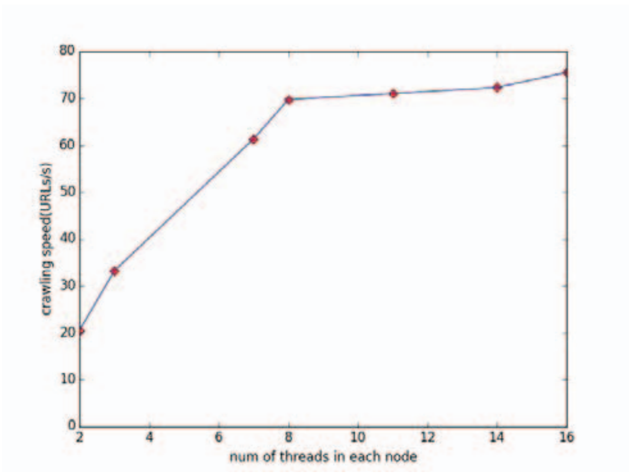

Figure 10. Different speed with the variable threads.

To test the capability of load balancing of the whole system, another group of experiments is also designed to evaluate the performance of the whole system with the passage of crawling time. We have counted the number of tasks assigned to each node in five different times. Figure 11 shows the results, with the passage of time (increase of total number of distributed URLs), the Standard Deviation between 4 nodes is becoming smaller. It means the numbers of URLs assigned to each worker node tend to be equal. Therefore we can conclude that the proposed system is robust in load balancing.

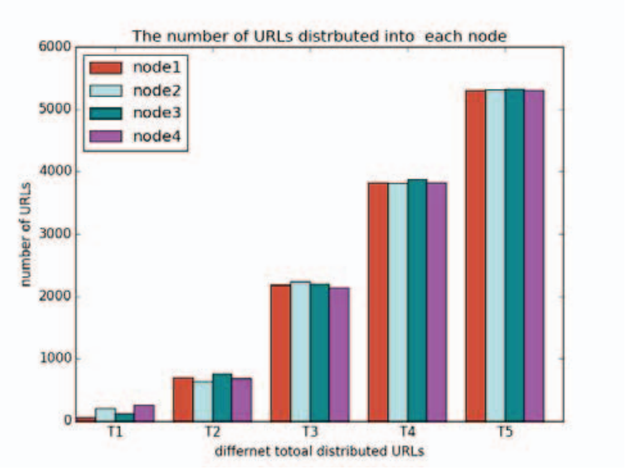

Figure 11. The number of URLs distributed into each node

Social networks is a valuable resource for data mining. However there has been little focus on the field of efficient and accurate data collection in previously published papers. There are at least three major challenges that some existing distributed frameworks of crawling could not deal with. One is the greater speed needed in the handling of the large mass of social networks' data. And Second is the need for a uniform collection model to deal with multiple web sources. Last is the simplification of implement whole system. Motivated by these challenges, this paper proposes a distributed web crawler framework based on Raspberry Pi called SORM. Our design realizes the separation of the distributed framework and web crawler. Thus the users can write their own crawler based on their needs. In addition, the efficient consistent hashing algorithm is applied to improve capability of load balancing. In order to prevent from crawling one page repeatedly, our system adopt the structure of bloom filter to eliminate the URLs which have been visited. The results of experiments on collecting data from Sina micro-blog, show that the proposed SWORM system could optimize the system resource scheduling efficiently and is effective in speeding up the collection of a large amount of data from multiple web sources.

Acknowledgments

This paper is supported by the National High Technology Research and Development Program of China (863 Program), under Grant (No.2013AA01A603-03) of National Nature Science Foundation of China (No.61373084), and Innovation Program of Shanghai Municipal Education Commission (No.14YZ011).

References

[1] J. Bar-Ilan, Data collection on the web for informetric purposes --- a review and analysis, Scientometrics, vol. 50(1), pp. 11 – 30, 2001.

[2] V.Shkapenyuk, T.Suel, Design and implementation of a high-performance distributed web crawler, ICDE,2002.

[3] ZHOU Shilong, CHEN Xingshu, Nutch crawling optimization from view of Hadoop, Journal of Computer Applications, 2013, 33(10) : 2792 —2795

[4] Zhenguo Xuan, Load Balancing Technology Based On Consistent Hashing For Database Cluster Systems, CITCS 2012.

[5] Yang Gaizhen. The Application of MapReduce in the Cloud Computing. Intelligence Information Processing and Trusted Computing (IPTC), 2011.

[6] Lei LIN, An URL Lookup Algorithm Based on MPHF and Bloom Filter, 8th Annual Conference of China Institute of Communications, 2011: 813 — 817.

[7] M.C. Ferris, M.P. Mesnier and J.J. More, NEOS and Condor: solving optimization problems over the internet, ACM Transactions on Mathematical Software, vol. 26(1), pp. 1 – 18, 2000.

[8] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin and Rina Panigrahy. Consistent hashing and random trees: Distributed cachine protocols for relieving hot spots on the World Wide Web. In Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pages 654-663 , 1997.

[9] Yan Zhao, Research on MongoDB Design and Query Optimization in Vehicle Management Information System,Applied Mechanics and Materials, 2013, Vol.2095 (246), pp.418-422.