TELKOMNIKA, Vol. 11, No. 7, July 2013, pp. 3618 ~ 3626
e-ISSN: 2087-278X

Crawling Microblog by Common-Designed Software

Gang Lu*[1], Shumei Liu[1], Kevin Lü[2]
[1]Collage of Information Science and Technology, Beijing University of Chemical Technology
15 BeiSanhuan East Road, ChaoYang District, Beijing, 100029, China
[2]Brunel University, Uxbridge UB8 3PH, UK
Corresponding author, e-mail: sizheng@126.com, smliu@mail.buct.edu.cn, kevin.lu@brunel.ac.uk

Abstract

Amount of microblogs data is needed to be crawled for research, business analyzing, and so on. However, a lot of dynamic Web techniques are used in microblog Web pages. That makes it hard to crawl data by parsing the contents of Web pages for traditional Web page crawlers. Fortunately, microblogs provide APIs. Well-structured data can be returned to users simply by accessing those APIs in form of URLs. Basing on that mechanism, researchers have obtained some data from microblogs to research. Nevertheless, no common software for crawling microblog has been published up to now. Everyone has to start designing a microblog crawler from very beginning. A common software architecture based on microblog APIs for microblog crawler is proposed in this paper, which is named as MBCrawler. Its structure, architecture, and key classes are introduced. It can be seen that MBCrawler is modular and scalable. By implementing a real microblog crawler for Sina Weibo, it is shown that MBCrawler can fit specific features of different microblogs.

Keywords*: Social Computing, microblog, crawler, Twitter*

1. Introduction

Microblogs such as Twitter and Sina Weibo have attracted attention of users, enterprises, governments, and researchers. They are interested in mining new information or finding some regular patterns of information propagation for business or research. Maybe governments can get the data by political power and enterprises by commercial behavior, but getting data of microblogs is the first issue comes to researchers. Generally, because of privacy and business reasons, microblog providers will not provide the data to researchers readily.

Early in 2001, the authors of [1] had described the common architecture of a search engine, including the issues about selecting and updating pages, storing, scalability, indexing, ranking, and so on. After that, the research about search engine technology has developed a lot, such as focused crawler [2] and detecting similar Web documents [3]. However, being different from traditional Web pages, Web 2.0 techniques such as AJAX (Asynchronous JavaScript and XML) are widely used in microblog Web pages, and the contents in microblog Web pages change too rapidly and dynamically for Web crawlers. That makes traditional crawlers for static Web pages not work well to microblog Web pages. There have been some researches on getting web content from AJAX based Web pages [4-6]. Nevertheless, they are based on the state of the application, and the technique is not that easy to implement. Fortunately, to encourage developers to develop applications about microblogs, the providers of microblogs publish some APIs in form of Web Service. Well-formatted data of microblogs can be returned by accessing the APIs in form of URLs. Except some work in which Twitter APIs were not used [7], and the work in which the authors did not state how they got the Twitter data [8] [9], most of existing research about Twitter utilizes the provided APIs [10-13].

It can be seen that Twitter APIs are widely used in the research work on Twitter. We believe that using APIs is the most popular and easy way to get data from microblogs. That motivates us to construct common software architecture to utilize the provided APIs, for automatically downloading and storing well-structured data into database. To make it convenient for researchers to obtain data from microblogs, a software architecture named as MBCrawler, which means MicroBlog Crawler, is proposed. Basing on this software architecture, a crawler using APIs of microblog with multi threads can be developed. The software

Received January 20, 2013; Revised April 5, 2013; Accepted April 15, 2013

architecture is designed to be modular and scalable, so that more functions can be added easily when more APIs are added, and details can be designed to fit different online social network services.

2. MBCrawler

2.1. Basic Structure of MBCrawler

Software architecture named as MBCrawler is proposed. This software architecture presents a main framework by which crawlers for microblog data can be easily designed, developed, and expanded. MBCrawler is designed as multi-threaded, and consists of six components, which are UI, Robots, Data Crawler, Models, Microblog APIs, and Database. The structure of MBCrawler is illustrated in Figure 1.

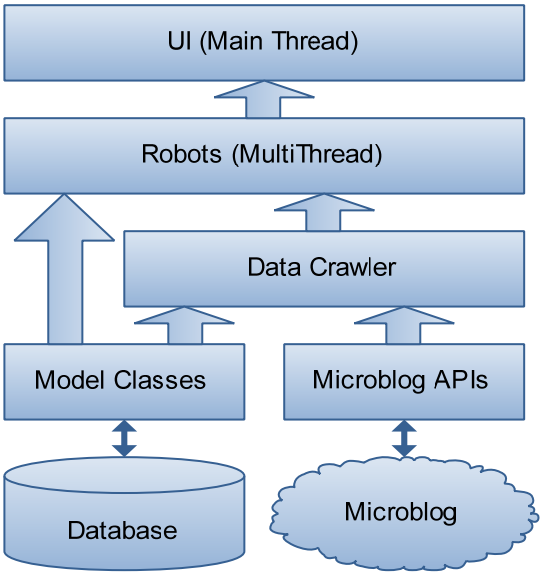

Figure 1. Basic Structure of MBCrawler

(1) Database

At the most bottom of MBCrawler is a relational database, in which crawled data is stored. The database tables are designed according to some entities, which are easily designed because microblogging APIs return well formatted result in format of XML or JSON. There are following main tables:

(a) A table named as users.

This table stores the basic information of users of microblogging services, such as user ID, nickname, location, numbers of followers and following users, and so on.

(b) A table named as user_relations.

This table stores the relationships between users of microblogging services. Relationships between users are recorded in the form of Source User follows Target User, by the user IDs. As a result, there are main columns of this table, which are source_user_id and target_user_id. For example, if User 1 follows User 2, where 1 and 2 are the two users' IDs, the source_user_id of the relationship will be 1, and the target_user_id will be 2.

(c) A table named as statuses.

This table stores the statuses crawled down, in-cluding the status ID, the content of the status, user ID of the poster, the time when the status is posted, and so on.

(d) A table named as comments.

In microblogging services, users can comment on statuses. This table stores the comments of statuses, including status ID, comment ID, the content of comments, the time of comments, the user IDs of the commenters, and so on.

Above are the four main tables in the database of MBCrawler. Different microblogging services may have different things, but users, user relationships, statuses, and comments are the most basic items of microblogging services. Different items can be added in, but these four items are necessary. There are two more tables, named as invalid_users and invalid_relations. They are used to store banned users' IDs and relationships which may have been canceled.

(2) Model Classes

Each entity like user and status has a database table correspondingly, as previously introduced. On the other hand, entities appear as Model Classes to the upper layers of the architecture, such as Robots and Data Crawler. Model Classes provide methods to manipulate data in database by the model classes.

(3) Microblog APIs

Microblogs provide some APIs which meet REST (Representational State Transfer) requirements. By calling those APIs, specific data will be returned in the format of XML or JSON. The layer of Microblogging APIs includes simple wrapped methods of the APIs. The methods submit URLs with parameters by HTTP requests, and return the result string in the format of XML or JSON.

(4) Data Crawler

Data Crawler plays a role as a controller between Robots and Microblogging APIs. From Robots, it receives commands indicating what data to crawl, and then invokes specific method in Microblogging APIs. When Data Crawler receives the result string from Microbloggings, it transforms the result strings from the format of XML or JSON into instances of certain model class, which will be used by the upper layer, Robots.

(5) Robots

MBCrawler needs different threads with different robots working in them to crawl different data at the same time. According to the four main tables given in database, the layer of Robots includes at least four main robots, which are User Relation Robot, User Information Robot, Status Robot, and Comment Robot. Each robot inherits from a base class of robots. Every robot calls Data Crawler to crawl data online by Microblogging APIs. All robots have their own waiting queues. Some are queues of users' ID, some are queues of IDs of statuses, and some are of IDs of comments. When a robot has crawled the data of head-of-queue, the head will be move to the end of the queue. So the waiting queues are all circular queues. Being different from robots of Web page, items in waiting queues of MBCrawler are not labeled to avoid being processed again, if they have been processed by robots. The reason is that data on microblogging services, such as user relationships, users' information, etc., always changes. When an item becomes the head-of-queue again, its data will be crawled again, in that the data may have changed, and should be updated. Following is the description of the four main robots.

(a) User Relation Robot must work, because MBCrawler has to crawl data along the social network constructed by user relationships. User Relation Robot starts working from a specific user. It crawls the IDs of the user's following users, and adds them to its waiting queue. After that, it crawls the IDs of the user's followers, and adds them to its waiting queue, too. It can be seen that it's BFS (Breadth First Search) on the social network.

(b) User Information Robot can be set to work or not. When it works, its waiting queue increases as the one of User Relation Robot, because User Relation Robot adds new items into it, too. User Information Robot crawls users' basic information, including the user's ID, nickname, gender, number of followers, number of following users, and the user's other basic information. If the information of a user is newly crawled, it will be saved into database. Otherwise, the data in database will be updated with the new one. If a user is found not to exist (may be banned user), the user ID will be recorded in the table of invalid_users.

(c) Status Robot can be set to work or not. If it is set to work, its waiting queue increases as the one of User Relation Robot, because User Relation Robot adds new items into

it, too. Status Robot crawls users' statuses and save them into database. Status Robot queries the last status stored in database for a certain user. If there have been some statuses in database for that user, the robot will crawl statuses after the last one. Otherwise, the robot will crawl statuses as very beginning as possible. When Status Robot has crawled some statuses of a user, it arranges them into a queue, and then crawls statuses which retweet them one by one. If some retweeting statuses are found, they are added to the status queue, too. Meanwhile, Status Robot stores the statuses into database. On the other hand, crawled statuses include data of the users who post them. Status Robot will try to add the user IDs into the waiting queues of User Relation Robot, User Information Robot, and itself.

(d) Comment Robot can be set to work or not. As it works, in its waiting queue there are status IDs. When Status Robot crawls statuses, the status IDs will be added into the waiting queue of Comment Robot. As a result, it can be inferred that Comment Robot has to work together with Status Robot, but working Comment Robot is not necessary for Status Robot. Comment Robot crawls comments of a status and save them into database. Crawled comments include data of the users who post them. Comment Robot will try to add the user IDs into the waiting queues of User Relation Robot, User Information Robot, and Status Robot.

There is another issue we should consider. A crawler should run continuously. However, some cases may interrupt its running. For example, the network disconnects, or even the computer on which the crawler is running has to reboot. As a result, a crawler should remember which point it has come to before it stops. For that reason, every robot will record the user ID or status ID before it start to crawl the information of a user or comments of a status. When the crawler restarts, the robots can find the user ID or status ID recorded before stopping, and then start from it.

(6) UI

UI means User Interface. It's the layer of representation, which shows some basic information and makes users able to interact with the program, namely control the program. The user who is using the program has to login the microblogging service by the program. After that, the user's basic information can be shown on UI. All working robots return messages during their processes to UI, and the messages are shown on UI. UI provides some simple functions, such as searching user online or in database. Necessary options of the program also can be set by UI. For example, it is usually needed to provide the information of database to the program by UI, so that the program can communicate with database. UI works in the main thread. By selecting which robot to work together with User Relation Robot, multi sub-threads with robots working in them are generated and started. What is the most important is that robots can be started, paused, continued, or stopped at any time by the buttons on UI.

2.2. Multi-Threads Structure of Robots Layer

MBCrawler is multi-threaded. Besides the main thread of UI, more threads with robots working in them are generated, so that different robots can work in parallel. The structure of the multi-threaded robots and their queues is as in Figure 2 shows.

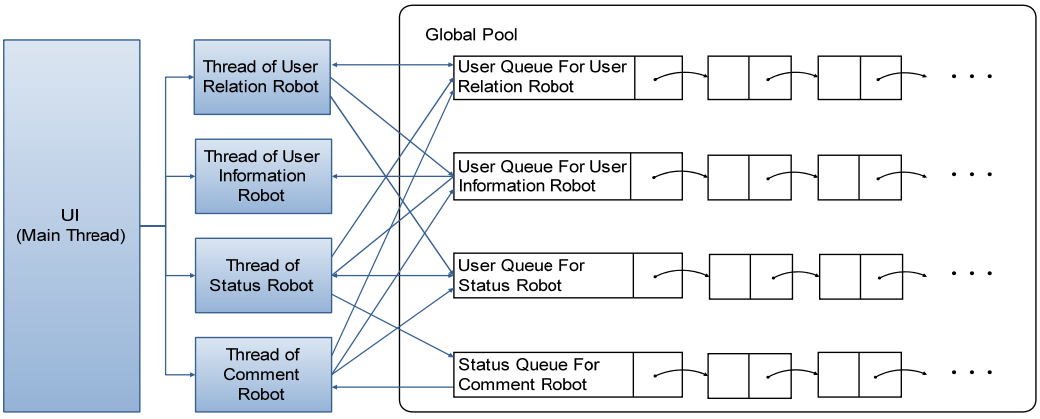

Figure 2. Multi-Threaded Robots and Their Queues

Each robot has its own waiting queue. The queue of Comment Robot is of status IDs, and others are of user IDs. All of the queues can be accessed by any robots.

The arrows between robots and the waiting queues in Figure 2 show the data flow direction between them. Basically, every robot fetches the head item from its own queue to crawl, and then move the item to the end of the queue. By crawling followers and followings IDs of a user, User Relation Robot processes a BFS in the social network of microblogging service. According to which robots are working, User Relation Robot will make the waiting queues of itself, User Information Robot and Status Robot grow at the same time, by adding new items to the queues synchronously. User Information Robot only fetches head item from and move it into the end of its waiting queue, but not add new items into any queues. Status Robot crawls the statuses posted by users as many as possible. Before saving a status into database, Status Robot crawl more statuses which retweet it. When a status is crawled while Comment Robot is working, Status Robot adds the ID of the crawled status into the waiting queue of Comment Robot. Status Robot also adds the user IDs included in the data of crawled statuses into its own queue as well as queues of User Relation Robot and User Information Robot if they do not exist in them. Like Status Robot, Comment Robot also adds commenters' user IDs included in crawled comments into the queues of User Relation Robot, User Information Robot and Status Robot.

As it is known, the waiting queues exist in memory. As they grow, they will occupy more and more memory. Considering that the amount of memory is limited, the size of the queues in memory should be limited, too. Solution for this issue is to limit the size of queues in memory, and let them grow in disk when their sizes in memory reach the limit. Database tables are provided for each queue separately. When the size of a queue reaches the limit, new items will be added into the corresponding table. The time when the items are added is also stored in the tables, so that items in the tables can be sorted in the order of their added time. When a queue extends into disk, namely uses its database table to store new items, head item will be moved into the database table as well. In that way, the part in memory and the part in the table work together as a whole queue. When all the items in memory have been moved to the end of a queue in database table, a series of items in front of the queue in database table will be moved from the table into memory to be processed.

2.3. Key Classes of MBCrawler

There are some key classes in MBCrawler. Those classes make up some modules to contribute to the implementation of main functions of MBCrawler.

(1) Model Classes.

In order to separate functions of software from specific database operation, Object Relational Mapping (ORM) is used by defining several model classes. The member variables of those classes correspond to the properties of entities, such as users' ID, users' name, etc., which are stored in tables in database. Besides, Model Classes also defined the methods for basic CRUD operation in database. They take charge of transforming data between database and Data Crawler or the robots. They make the logic functions of software not need to care about specific database operation.

(2) Database Factory Classes.

MBCrawler needs a database to store the crawled data. In order to not to be limited to a specific type of database, the Factory Pattern is used to design the classes for database accessing.

The class Database is an abstract class, which defines the methods necessary for database operation. According to specific type of database, different subclasses can be defined by inheriting the root class Database, and implementing the methods defined in class Database. For example, SqlDatabase and OracleDatase both inherit Database, but they implement the methods for MS SQL Server and Oracle respectively. If it is needed, more subclasses for other types of databases can be added.

By class DatabaseFactory, the design pattern Factory is used to create instance of class Database. DatabaseFactory returns an instance of Database for any type of database. In that way, the change of database type is limited in class DatabaseFactory, but not distributes everywhere in the code.

(3) Robot Classes.

MBCrawler generates multi threads with robots working in them. That should be a scalable structure, so a basic class for robots is designed and new robot classes can be easily extended from it. The base class RobotBase defines the necessary queues and the methods. Subclasses for specific robots inherit from it, and implement their own methods, especially Start(), in which how the robots work is specified.

(4) Queue Classes.

As it is shown in Section 2.2, each robot has a waiting queue. A waiting queue is divided into two parts, which are the part in memory and the part in database. Every robot maintains the handles of some queues of user IDs or status IDs. The class QueueBase defines queue handles and the basic operations.

The whole queue is designed as a circular queue, to ensure that every node will be process repeatedly for information updating. As a result, the queues will grow longer and longer as the crawler works. However, the limited memory of computer cannot contain unlimited queues. To deal with that issue, a queue is departed into two parts. One part is maintained in memory, which is defined as *lstWaitingID*. The other part is stored in disk, namely in database, which is defined as *lstWaitingIDInDB*. *lstWaitingID* and *lstWaitingIDInDB* both are member variables of class QueueBase. *lstWaitingID* is a linked list, whose length is limited by *iMaxLengthInMem*. *lstWaitingIDInDB* is an instance of class QueueBuffer. When a robot starts to work, it will create a temporary table in database, in order to extend the queue handled by it in disk. When the length of the whole queue is longer than *iMaxLengthInMem*, new node will be added into the temporary table. In this case, all nodes of *lstWaitingID* will be moved into *lstWaitingIDInDB* at last. At this time, the first *iMaxLengthInMem* nodes of *lstWaitingIDInDB* will be moved into *lstWaitingID*. QueueBufferFor is an enumeration type that helps QueueBuffer to determine for which robot to create *lstWaitingIDInDB*. Figure 3 illustrates the structure and working process of a queue.

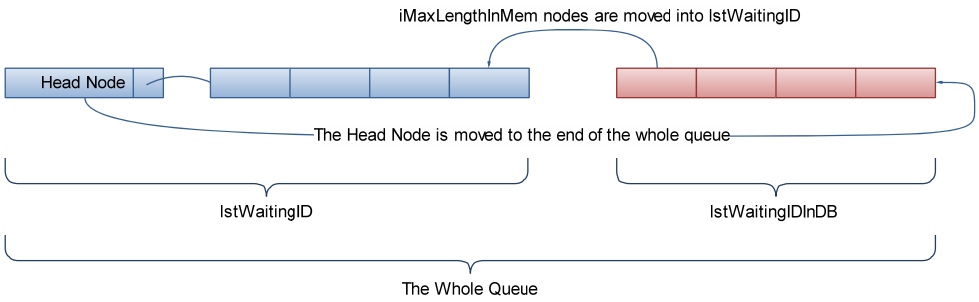

Figure 3. Structure and Working Process of a Queue

However, the previous process of queue managing maybe not good enough. When *lstWaitingIDInDB* is very long, that means the table storing the queue has too many records. If *iMaxLengthInMem* is set to a big number, the process of moving *iMaxLengthInMem* IDs from *lstWaitingIDInDB* to *lstWaitingID* will take a long time. That may cause the response of database timeout. An alternate way to solve the problem, is adding an additional thread to take the charge of moving IDs between the two parts of the queue. Producer-Consumer model can be used here. Figure 4 shows the improved process.

As Figure 4 illustrates, a model called *Queue Coordinator* and a queue named *Back Buffer* are added. *Queue Coordinator* fetches IDs from *lstWaitingIDInDB* to *lstWaitingID*. A crawling robot fetches ID from *lstWaitingID* to crawl. After that, it sends the crawled ID to *Back Buffer*. At the same time, *Queue Coordinator* fetches crawled IDs from *Back Buffer* and pushes them to the end of *lstWaitingIDInDB*. The crawling robot and *Queue Coordinator* work asynchronously in parallel. From the view of *lstWaitingID*, *Queue Coordinator* is the producer, and the crawling robot is the consumer. On the contrary, from the view of *Back Buffer*, *Queue Coordinator* is the consumer, and the crawling robot is the producer. Of course, each robot needs a *Back Buffer*, but only one *Queue Coordinator* can manage all of the queues.

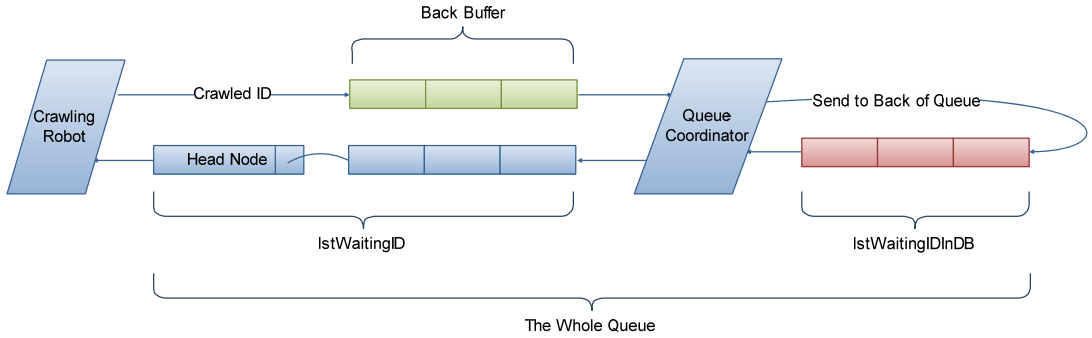

Figure 4. Producer-Consumer Based Queue Managing

(5) Other Utility Classes.

There are other utility classes wrapping some functions to make them easy to use. For example, classes about the settings make the options such as the information of database connection saved into and loaded from an encrypted file by serializing and deserializing. There are also GlobalPool and PubHelper classes. All public objects which can be accessed by every part of MBCrawler at any time are organized in GlobalPool class in form of static members. PubHelper encapsulates some gadget functions and provides easy-to-use methods of them.

3. Implementation as Sinawler

Basing on MBCrawler, a crawler program named as Sinawler (Sinawler is the combination of the words Sina and crawler) for Sina Weibo, is developed. Its code can be found at http://code.google.com/p/sinawler. Sina Weibo is one of the most popular microblogs in China. It has some more features than Twitter. For example, users of Sina Weibo can label themselves with no more than 10 words, which reflect the users' hobbies, profession, and so on. These words are called as tags. As a result, a new robot takes the charge of crawling the data about tags is added in Sinawler, which is named as UserTagRobot.

In our research work, Sinawler initially ran for months. During that time, it crawled 8,875,141 users' basic information, 55,307,787 user relationship, 1,299,253 tags, 44,958,974 statuses, and 35,546,637 comments. The data is stored in a SQL Server database. However, in that time, Sinawler was frequently stopped to be modified because of bugs and updates. That makes the data some dirty due to some testing result. From 15:54:46 on May 30th, 2011 to 11:44:26 on January 7th, 2012, a stable version of Sinawler was running uninterruptedly. The new data obtained this time is listed in Table 1.

Table 1. Crawled Data by a Stable Version of Sinawler

Robot	Data Content	Total Records	ACR (Records/Minute)
UserInfoRobot	User Information	6,571,955	20
UserRelationRobot	User Relationship	37,902,219	118
UserTagRobot	Tag	1,068,060	3
	User Owning Tags	8,031,712	25
StatusRobot	Status	32,627,963	102
CommentRobot	Comment	26,884,365	84

ACR is the abbreviation for Average Crawling Rate. It is different for each robot, because they access different Sina Weibo APIs. For example, only one user's information can be obtained by UserInfoRobot for each invoking the relative API, while at most 5000 user relationships can be obtained by UserRelationRobot for one time. ACR is also related to the data. For instance, many users don't set their tags, so UserTagRobot can't get their tags. That lowers the ACR of UserTagRobot.

4. Conclusion and Future Work

MBCrawler, which is a software architecture for microblog crawler, is proposed here. By dividing the whole architecture into several levels and applying some simple design patterns, the structure of the architecture is highly modularized and scalable. The different parts of the architecture are loose coupling, and each part is high cohesion. That makes it easy to be modified and extended.

A crawler software for Sina Weibo is implemented basing on MBCrawler, which is named as Sinawler. A robot about users' tags are easily added according to Sina Weibo API. Because of the careful design of the architecture, we have easily upgrade Sinawler according to Sina Weibo API 2.0, in which only JSON format is used and some new properties are added to users, tags, and so on. A large number of data from Sina Weibo has been crawled by Sinawler.

Comparing to traditional Web page crawlers, MBCrawler has its own features. Microblog APIs are the foundation of the design and implementation of MBCrawler, so MBCrawler does not need to download Web pages to store and parse. No indexing module for Web pages is needed as well. Well-structured data is returned by microblog APIs and is stored into databases directly. As a result, complex technical issues resulted from AJAX are avoided. In addition, the indexing mechanism of the used database can be utilized to enhance the performance of the database.

Nevertheless, there is an important condition for using MBCrawler. To access the APIs, MBCrawler has to act as an application registered at the microblog provider. Fortunately, it is easy to register applications for it. After that, a unique pair of App Key and App Secret will be given to access APIs by the application. We registered five applications for the five robots in our Sinawler. As a result, each robot can work as an independent application, and they will not share the same access frequency restriction. That makes Sinawler work more efficiently.

In the future, we think MBCrawler can be improved mainly in two aspects. Firstly, a focused module can be designed and added. So that we can tell the robots what type of data to crawl. For example, the robots can be told to crawl the information of users who are in a specific city, or the statuses including specific words. Secondly, we would like to design a ranking module. It is impossible to crawl the whole social graph due to the large scale of it. A ranking module will help the crawler to select more important users to crawl. Some researches about social network crawling strategy can be embedded in this module, too. Additionally, because database is loosely coupled with MBCrawler, different databases can be easily tried, such as some NoSQL databases according to the practice requirements. For example, using any one of graph databases such as Neo4j to store the social graph would be interesting. New types of databases may bring new useful features.

Acknowledgments

This work is supported by the Fundamental Research Funds for the Central Universities grants ZZ1224.

References

[1] Arasu A, Cho J, Garcia-Molina H, Paepcke A, Raghavan S. Searching the web. *ACM Transactions on Internet Technology*. 2001; 1(1): 2-43.
[2] Maimunah S, Sastramihardja HS, Widyantoro DH, Kuspriyanto NFN. CTFC: more Comprehensive Traversal Focused Crawler. *KOMUNIKA Indonesian Journal of Electrical Engineering*. 2012; 10(1): 189-198.
[3] Das S, Mathew M, Vijayaraghavan P. An Efficient Approach for Finding Near Duplicate Web pages using Minimum Weight Overlapping Method. *International Journal of Electrical and Computer Engineering*. 2011; 1(2): 187-194.
[4] Mesbah A, van Deursen A. *Invariant-based automatic testing of AJAX user interfaces*. Proceedings of the 31st International Conference on Software Engineering. Washington, DC, USA. 2009; 210-220.
[5] Peng Z, He N, Jiang C, Li Z, Xu L, Li Y, et al. Graph-Based AJAX Crawl: Mining Data from Rich Internet Applications. *2012 International Conference on Computer Science and Electronics Engineering*. Hangzhou, China. 2012; 590-594.
[6] Mesbah A, van Deursen A, Lenselink S. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Transactions on the Web*. 2012; 6(1):1-30.

[7] Weng J, Lim E-P, Jiang J, He Q. T*witterRank: finding topic-sensitive influential twitterers*. Proceedings of the third ACM international conference on Web search and data mining. New York, USA. 2010; 261-270.
[8] Mendoza M, Poblete B, Castillo C. *Twitter Under Crisis: Can we trust what we RT?* Proceedings of the First Workshop on Social Media Analytics. Washington, DC, USA. 2010; 71-79.
[9] Sriram B, Fuhry D, Demir E, Ferhatosmanoglu H, Demirbas M. *Short text classification in twitter to improve information filtering*. Proceeding of the 33rd international ACM SIGIR conference on research and development in information retrieval. Geneva, Switzerland. 2010; 841-842.
[10] Kwak H, Lee C, Park H, Moon S. *What is Twitter, a social network or a news media?* Proceedings of the 19th international conference on World Wide Web. Raleigh, USA. 2010; 591-600.
[11] Wu S, Hofman JM, Mason WA, Watts DJ. *Who says what to whom on twitter*. Proceedings of the 20th international conference on World Wide Web. Hyderabad, India. 2011; 705-714.
[12] Bakshy E, Hofman JM, Mason WA, Watts DJ. *Everyone's an influencer: quantifying influence on twitter*. Proceedings of the fourth ACM international conference on Web search and data mining. Hong Kong, China. 2011; 65-74.
[13] Li R, Lei KH, Khadiwala R, Chang KC-C. TEDAS: A Twitter-based Event Detection and Analysis System. *International Conference on Data Engineering*. Los Alamitos, CA, USA. 2012; 1273-1276.