An Optimized Algorithm for Reduce Task Scheduling

Xiaotong Zhang\textsuperscript{a}, Bin Hu\textsuperscript{b}, Jiafu Jiang\textsuperscript{c}

\textsuperscript{a,b,c} School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China
\textsuperscript{a} Email: xiaotong.zh@gmail.com
\textsuperscript{b,c} Email: \{hubin\textunderscore astb, jiangjjf\}@163.com

Abstract—In this paper, we propose a novel algorithm to solve the starving problem of the small jobs and reduce the process time of the small jobs on Hadoop platform. Current schedulers of MapReduce/Hadoop are quite successful in achieving data locality and scheduling the reduce tasks with a greedy algorithm. Some jobs may have hundreds of map tasks and just several reduce tasks, in which case, the reduce tasks of the large jobs require more time for waiting, which will result in the starving problem of the small jobs. Since the map tasks and the reduce tasks are scheduled separately, we can change the way the scheduler launches the reduce tasks without affecting the map phase. Therefore we develop an optimized algorithm to schedule the reduce tasks with the shortest remaining time (SRT) of the map tasks. We apply our algorithm to the fair scheduler and the capacity scheduler, which are both widely used in real production environment. The evaluation results show that the SRT algorithm can decrease the process time of the small jobs effectively.

Index Terms—mapreduce, hadoop, schedule, SRT

I. INTRODUCTION

MapReduce/\textsuperscript{[1]} is a programming model for processing large data sets on clusters of computers, the name of which is an implementation of the model by Google. A popular free implementation is Apache Hadoop\textsuperscript{[2]}. Hadoop is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. Hadoop contains two important components: Hadoop Distributed File System (HDFS) and Hadoop MapReduce. HDFS provides a distributed, scalable filesystem for Hadoop. MapReduce is used for processing parallelizable problems across huge datasets using a large number of computers (nodes). The term MapReduce actually refers to two separate and distinct tasks that Hadoop programs perform. The map tasks take a set of data and convert them into another set of data, where individual elements are broken down into tuples (key/value pairs). The reduce tasks take the output from the map tasks as input and combine those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce tasks are always performed after the map tasks. In general, MapReduce framework contains a master node (JobTracker) and some slave nodes (TaskTracker). To process a job, the client submits a job to the JobTracker, which will do some initial work for the job and add the job to a queue. When a TaskTracker has the abilities to run tasks, it sends a request (heartbeat) to the JobTracker, which will use the TaskScheduler to pick up some tasks of the jobs in the queue, and assign them to the TaskTracker. The TaskTracker lives on each node of the Hadoop cluster and is responsible for executing tasks. The TaskTracker communicates with the JobTracker through a heartbeat protocol. In essence the heartbeat is a mechanism for the TaskTrackers to announce their availability on the cluster. In addition to announcing its availability, the heartbeat protocol also includes information about the state of the TaskTracker. Heartbeat messages indicate whether the TaskTracker is ready for the next task or not. When the JobTracker receives a heartbeat message from a TaskTracker declaring that it is ready for the next task, the JobTracker selects the next available job in the priority list and determines which task is appropriate for the TaskTracker to execute.

How to schedule the tasks among numerous TaskTrackers is one of the most important issues in Hadoop, and it can directly influence the performance of the Hadoop cluster. For simple, Hadoop uses the FIFO scheduler as the default scheduler. The default FIFO scheduler is not effective in multi-user situations, besides, it can cause starvation of the small jobs. Therefore, Facebook developed another scheduler called Fair Scheduler\textsuperscript{[3]}, which supported muti-user to share a Hadoop cluster and solved the starving problem of the small jobs in some way. Another famous scheduler developed by Yahoo is the Capacity Scheduler\textsuperscript{[4]}. The map tasks and the reduce tasks are scheduled separately. When choosing a map task from a job, Hadoop tries to choose the one of which data are on the same node of the TaskTracker, in this way Hadoop can achieve data locality of the map tasks. By default, Hadoop uses a greedy algorithm to launch the reduce tasks, that means when a TaskTracker requests some reduce tasks, the JobTracker will assign the reduce tasks to it as soon as possible. Since the reduce phase has

\textsuperscript{1}This work was supported partially by National High-Tech Research and Development Program of China under Grant No.2011AA040101 and was jointly funded by the Fundamental Research Funds for the Central Universities China No.FRMP-MP-12-007A.

© 2014 ACADEMY PUBLISHER
doi:10.4304/jcp.9.4.794-801
innate relation with the map phase, this can lead to poor utilization.

The reduce phase includes three subphases: shuffle, sort, and reduce. In the shuffle phase, input of a reduce task is the sorted output of the map tasks. In this phase the framework fetches the relevant partition of the output of all the map tasks, via HTTP. In the sort phase, the framework groups the inputs of the reduce tasks by keys (since different map tasks may have output the same key). The shuffle and sort phases occur simultaneously. As the map-outputs are being fetched, they will be merged momentarily. A reduce task copies all the data that belong to it from all map intermediate data, and sorts the data while copying. But it has to wait until all the intermediate data have been copied and sorted, then it begins the third stage. That means the real reduce function cannot start until all the map tasks are done. Some jobs may have hundreds of map tasks and just several reduce tasks, in which case, the reduce tasks may spend most of the time waiting for the data and keep occupying the resources. Therefore, this can lead to the starving problem of the small jobs [5].

Therefore, the role of a good scheduling algorithm is to efficiently assign each task to a priority to minimize makespan [6]. To solve this problem, Jian Tan et al. proposed a solution where the scheduler will gradually launch the reducer tasks depending on the map progress [7]. But it does not consider the factor of time. For example, a job with 80% map tasks completed may even has a longer time to finish all the map tasks than that of a job with only 20% map tasks completed. This thesis aims at providing a simple way of achieving launching the reduce tasks depending on the shortest remaining time (SRT) of the map tasks. In this way, we reduce the process time and the reduce time of the small jobs. We implement our algorithm based on the fair scheduler, which is widely used in real production environment. Our experiment results show that the reduce time of the jobs decreases a lot, especially for the small jobs (61% ~ 39 %). The process time of the small jobs can decrease by 55% at most. In addition, the process time of the large jobs can also decrease by 1% ~ 4%. The remainder of this paper is organized as follows. In Section II, related work on different scheduling algorithms of Hadoop is presented. In Section III, the implement detail of the SRT algorithm and the drawback are described. In Section IV, the experimental results and the analyses of performance of the SRT scheduler on real Hadoop environment are given. Finally, in Section V, conclusions and suggestions for future work are presented.

II. RELATED WORKS

In this section we briefly introduce some schedulers of Hadoop, including Hadoop inner schedulers (the FIFO scheduler, the fair scheduler, and the capacity scheduler) and schedulers developed by others. Specially, we focus on the fair scheduler.

A. FIFO scheduler

The FIFO scheduler is the default scheduler of Hadoop. In this scheduler, all jobs are submitted to a single queue, when a TaskTracker’s heartbeat comes, the scheduler simply picks up a job from the head of the queue, and tries to assign the tasks of the job, if the job has no tasks to be assigned (for example, all tasks have been assigned), then the scheduler tries the next job. The jobs in the queue are sorted by their priority, then submitted time.

The advantage of the FIFO scheduler is that the algorithm is quite simple and straightforward, the load of the JobTracker is relatively low. But it can result in starvation when the small jobs come after the large jobs. That means, a job has to wait until all the tasks (map, reduce) of the jobs before it have been launched.

B. Fair Scheduler

The fair scheduler is designed to assign resources to jobs such that each job gets roughly same amount of resources over time. For example, when there is only one job submitted, it uses the entire cluster, and when other jobs are submitted, tasks slots that free up are assigned to the new jobs. In this way, the problem of the FIFO scheduler that we mention above is solved: the small jobs can be finished in reasonable time while not starving the large jobs. The fair scheduler organizes jobs into pools and shares resources fairly across all pools. By default, each user is allocated a separate pool and gets an equal share of the cluster no matter how many jobs are submitted. Within each pool, either the fair sharing or the FIFO algorithm is used for scheduling jobs. In addition to providing fair sharing, the fair scheduler allows assigning guaranteed minimum shares to pools, which is useful for ensuring that certain users, groups or production applications always get sufficient resources. When a pool contains jobs, it gets at least its minimum share, however when the pool does not need its full guaranteed share, the excess is split between other pools. The fair scheduler implementation keeps track of the compute time for each job in the system. Periodically, the scheduler inspects the jobs to compute the difference between the compute time the job received and the time it should have received in an ideal scheduler. The result determines the deficit for the task. The fair scheduler will ensure that the task with the highest deficit is launched next. Disadvantages of the fair scheduler are listed below:

1) The greed algorithm of launching reduce tasks will cause reduce starvation of the small jobs.

2) Locality of the reduce tasks does not been considered.

To solve the first problem, Jian Tan et al. proposed a scheduler called Couping Scheduler which will gradually launch the reduce tasks depending on the map progress [7]. But the Couping Scheduler does not consider the factor of time. The jobs that have larger rate of progress of the map tasks may have longer time to finish all the map tasks. We implement our algorithm based on the shortest remaining time of the the map tasks and apply it to the
fair scheduler, which is widely used in real production
environment. The detail of our algorithm is provided in
section III.

C. Other schedulers

The capacity scheduler is designed to guarantee a
minimum capacity of each organization. The central idea
is that the available resources in the Hadoop cluster are
partitioned among multiple organizations who collectively
fund the cluster based on computing needs. In the capacity
scheduler, several queues are created, each of which is
assigned a guaranteed capacity (where the overall capacity
of the cluster is the sum of each queue’s capacity). The
queues are monitored, and if a queue is not consuming its
allocated capacity, this excess capacity can be temporarily
allocated to other queues. Given that the queues can
represent a person or larger organization, any available
capacity is redistributed for other users. To improve the
data locality of the fair scheduler, Matei Zaharia et al.
developed a simple technique called delay scheduling [8].
But it still ignored the locality of the reduce tasks. M.
Hammad et al. addressed this problem by incorporating
network locations and sizes of the partitions of the
reduce tasks in the scheduling decisions, and mitigated
network traffic and improved MapReduce performance
[9]. Lijing Li et al. [10] proposed a new improvement
of the Hadoop relevant data locality scheduling algorithm
based on LATE [11]. Besides, authors of [12] and [13]
proposed schedulers which managed the jobs to meet
deadlines. Similarly, the schedulers that were suitable for
real-time circumstance were proposed [14] [15]. Sheng-
bing Ren and Dieudonne Muheto proposed strategies to
schedule tasks in OLAM (Online Analytic Mining) using
MapReduce [16]. Specially, Xiaoli Wang et al. took full
consideration of the relationship between the performance
and energy consumption of servers, and proposed a new
energy-efficient task scheduling model based on MapRe-
duce, and gave the corresponding algorithm [17].

III. ALGORITHM DESCRIPTION

A. Algorithm Implement

Suppose we have M jobs waiting to be processed, we
use J to represent the set of them.

\[ J = \{ job_i | 1 \leq i \leq M \} \]  \hspace{1cm} (1)

For every \( job_i \) in \( J \), we represent the map tasks of \( job_i \)
by \( U_i \).

\[ U_i = F_i \cup R_i \cup P_i \]  \hspace{1cm} (2)

The symbols \( F_i \), \( R_i \), and \( P_i \) represent the set of the fin-
ished map tasks, the running map tasks, and the pending
map tasks of \( job_i \). In the fair scheduler, the pools are
sorted by fair sharing algorithm while the jobs in a pool
can be scheduled by either the FIFO algorithm or the fair
sharing algorithm. We apply the SRT algorithm to the
fair scheduler in this way: we change the way the fair
scheduler sorts the jobs within a pool when scheduling
the reduce tasks. To achieve that goal, we provide a
comparator called the SRT comparator to compare two
jobs when sorting jobs according to their remaining time
of the map tasks. To calculate the remaining time of the
map tasks of \( job_i \), we have to estimate the remaining time
of the tasks in both \( R_i \) and \( P_i \). Let \( f(U_i) \) represent the
remaining time it takes to finish all the tasks in \( U_i \). We
calculate \( f(U_i) \) like below:

\[ f(U_i) = \max[f(R_i), f(P_i)] \]  \hspace{1cm} (3)

In fact, we have three methods to calculate \( f(R_i) \): we
can treat the running tasks as the pending map tasks,
calculate the remaining time of each running task
according to its progress and add them together, or we
can simply ignore them. There is no mechanism through
which we can obtain the process of a specified running
task in Hadoop. Therefore, the second method will be
difficult to achieve unless we modify the underlying
architecture of Hadoop. Therefore we choose the third
way to calculate \( f(U_i) \). Note that, two jobs with the same
priority will have the same number of the running tasks
according to the fair scheduler. Besides, each job will
have a small and limited number of the running tasks
when many jobs are processed simultaneously, in which
case \( f(R_i) \) is much smaller than \( f(U_i) \). Suppose \( job_i \) has
run for time \( T_i \), the number of the finished map tasks is
\( f_i \), the number of the pending map tasks is \( p_i \), \( f(U_i) \) can
be expressed like below:

\[ f(U_i) = f(P_i) = \begin{cases} +\infty & f_i = 0 \\ T_i/f_i * p_i & f_i \neq 0 \end{cases} \]  \hspace{1cm} (4)

We predict the remaining time of the map tasks in a job
with the average time of map tasks multiplied by the
amount of the pending map tasks. For example, suppose
that a job has been running for 4 minutes, and it has 2
finished map tasks and 1 pending map task, we calculate
the remaining time of the map tasks like this: 4 min / 2 * 1 = 2 min. Some other methods to estimate the
remaining time of the jobs are provided in [18] [19] [20].
We let \( \tau_s(i) \) and \( \tau_e(i) \) represent the start time and end
time of \( job_i \) respectively, and let \( \sigma_s(i) \) and \( \sigma_e(i) \) represent the start
time and end time of the reduce phase of \( job_i \)
respectively. Therefore,

\[ Average_J = \frac{1}{M} \sum_{i=1}^{M} [\tau_e(i) - \tau_s(i)] \]  \hspace{1cm} (5)

is the average process time of all jobs. And

\[ Average_R = \frac{1}{M} \sum_{i=1}^{M} [\sigma_e(i) - \sigma_s(i)] \]  \hspace{1cm} (6)

is the average process time of the reduce phases of all
jobs. Once the request of a TaskTracker comes, we sort
the jobs in the waiting queue, and assign tasks to the
TaskTracker in order. The steps of assigning the reduce
tasks using the SRT algorithm are shown in Fig. 1.

Step 1 The TaskTracker sends a request for the reduce
tasks.
Step 2 The JobTracker forwards the request to the TaskScheduler.
Step 3 The TaskScheduler sorts the pools according to the fair scheduler. In this way the advantage of the fair scheduler can be achieved.
Step 4 The TaskScheduler selects a pool from the ordered pools (for example, pool 2), and sorts the jobs in pool 2 using the SRT algorithm.
Step 5 The TaskScheduler picks up a reduce task of the job with the shortest remaining time of the map phase.
Step 6 The TaskScheduler returns the reduce task to the JobTracker.
Step 7 The JobTracker assigns the reduce task to the TaskTracker.

The SRT comparator takes the status information of two jobs as input, and output an integer which indicates the order of two jobs. The detail of the SRT comparator is shown in Algorithm 1 below.

With the SRT algorithm, we make the task of the shortest remaining time of the map phase get lunched before others. The remaining time of the map phase is changing dynamically according to the feedback. If two jobs both finish their map phases, we let the job with less map tasks lunch its reduce tasks ahead of another. Therefore, the small jobs will not be starving and the process time of the small jobs decreases greatly. The SRT algorithm focuses on scheduling the reduce tasks, therefore it can be easily applied to many other schedulers of Hadoop.

B. Drawback

The SRT algorithm does not consider the time that the reduce tasks spend on the shuffle and the sort phases, which will sometimes lead to resource underutilization. For instance, if job1 has a shorter map remaining time than job2 does, we make job1 ahead of job2 according to the SRT algorithm. But if job2 has more intermediate data to copy, then it may be better to launch the reduce tasks of job2, because these reduce tasks may spend much time copying intermediate data and when they finish copying the intermediate data of previous map tasks, all map tasks may have been finished, so the reduce tasks do not need to wait. On the contrary, job1 may have less intermediate data to copy and block for waiting new map intermediate data. Figure 2 below illustrates the problem. To remedy the defect as we describe above, a more precise method to calculate the remaining time of the map and the reduce phases is needed. Since optimal solutions are hard to compute (NP-hard) [21], we propose an approximation algorithm. In this model, the SRT algorithm will work fine and reduce the average process time of the small

**Algorithm 1 SRT comparator**

Require:
- The status of two jobs(job1, job2):
  - \( f_i \): the number of the finished map tasks of job1
  - \( p_i \): the number of the pending map tasks of job1
  - \( T_i \): the time that job1 has spent

Ensure:
- -1 (schedule job1 prior job2);
- 0 (keep the current order of job1 and job2);
- 1 (schedule job2 prior job1).

1: if \( f_1 = 0 \land f_2 \neq 0 \) then
2: return 1
3: else if \( f_1 \neq 0 \land f_2 = 0 \) then
4: return -1
5: else if \( f_1 = 0 \land f_2 = 0 \) then
6: if \( p_1 > p_2 \) then
7: return 1
8: else if \( p_1 < p_2 \) then
9: return -1
10: else
11: return 0
12: end if
13: else
14: if \( p_1 = 0 \land p_2 = 0 \) then
15: if \( f_1 > f_2 \) then
16: return 1
17: else if \( f_1 < f_2 \) then
18: return -1
19: else
20: return 0
21: end if
22: end if
23: RemainingTime1 ← \( T_1/f_1 \times p_1 \)
24: RemainingTime2 ← \( T_2/f_2 \times p_2 \)
25: if RemainingTime1 > RemainingTime2 then
26: return 1
27: else if RemainingTime1 < RemainingTime2 then
28: return -1
29: else
30: return 0
31: end if
32: end if
IV. EXPERIMENT A RESULTS

In order to evaluate the performance of the SRT algorithm, we conduct a series of experiments. We use gridmix2 as a tool to make the tests. Gridmix2 is a benchmark of Hadoop. It can automatically generate and submit some jobs, which can be configured in the kind and the amount of the reduce tasks. There are two kinds of jobs in our experiments: small and large. The large jobs have more map tasks and will take a longer time to finish their map phases. Without loss of generality, we firstly submit all the jobs with random order. We compare the average reduce time, average job process time, and the total time for finishing all the jobs using both the SRT scheduler and the fair scheduler. We repeat each test for twenty times. Secondly, we conduct an experiment in which jobs are submitted with the order from large ones to small ones. In this way we can find out how the algorithm works in the worst situation. Thirdly, we repeat the first experiment under different number of slave nodes. Finally, we apply the SRT algorithm to the capacity scheduler. The hardware configuration in our experiments is showed in Table I.

### Table I. The Hardware Configuration

<table>
<thead>
<tr>
<th>No.</th>
<th>Type</th>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Master</td>
<td>8-core, 2.13GHz</td>
<td>16G</td>
</tr>
<tr>
<td>1</td>
<td>Slave</td>
<td>4-core, 2.13GHz</td>
<td>4G</td>
</tr>
<tr>
<td>2</td>
<td>Slave</td>
<td>4-core, 2.13GHz</td>
<td>4G</td>
</tr>
<tr>
<td>3</td>
<td>Slave</td>
<td>8-core, 2.13GHz</td>
<td>16G</td>
</tr>
<tr>
<td>4</td>
<td>Slave</td>
<td>8-core, 2.13GHz</td>
<td>4G</td>
</tr>
<tr>
<td>5</td>
<td>Slave</td>
<td>16-core, 2.66GHz</td>
<td>48G</td>
</tr>
<tr>
<td>6</td>
<td>Slave</td>
<td>16-core, 2.66GHz</td>
<td>48G</td>
</tr>
<tr>
<td>7</td>
<td>Slave</td>
<td>16-core, 2.66GHz</td>
<td>48G</td>
</tr>
<tr>
<td>8</td>
<td>Slave</td>
<td>4-core, 2.13GHz</td>
<td>4G</td>
</tr>
<tr>
<td>9</td>
<td>Slave</td>
<td>4-core, 2.13GHz</td>
<td>4G</td>
</tr>
<tr>
<td>10</td>
<td>Slave</td>
<td>4-core, 2.13GHz</td>
<td>4G</td>
</tr>
</tbody>
</table>

#### A. Random Order

In this experiment, we submit the jobs by a random program. The test bed that we use contains five nodes: No.0 ~ No.4. We conduct four groups of tests, each group has a different amount of jobs, while the numbers of different kinds (small, large) of jobs in the same group are the same. Each group has $10 \times 2$, $20 \times 2$, $30 \times 2$, $40 \times 2$ jobs respectively. For example the first group will have 10 small jobs and 10 large jobs. The tests in each group will be repeated for twenty times. Figure [3, 4] show the results of the average job process time of two kinds of jobs when using the SRT scheduler and the fair scheduler.

It is clear that the process time of the small jobs can decrease effectively when using the SRT algorithm. When the number of jobs is not that large ($10 \times 2$), the process time of the small jobs can even decrease by 55%. Figure 4 shows that the process time of the large jobs keeps nearly the same using two different schedulers. The process time of large jobs can decrease by 1% ~ 4%. We can see that the SRT algorithm has different effects on different jobs because it has different effects on reducing the process time of the reduce tasks. To see whether the SRT algorithm really has an influence on the reduce phase, we calculate the average reduce time of the jobs, figure [5, 6] show the results.

The reduce time of both small jobs and large jobs decreases. Specially, the reduce time of the small jobs gets a decrease of 61% to 39%. Figure 7 shows the compare of the process time total jobs.

The process time of total jobs is determined by when the last job finishes. We can see that the process time of total jobs of the SRT algorithm is a little more than that of the fair scheduler. For example, when we submit $40 \times 2$ jobs, the process time of total jobs of the SRT algorithm
is 6% more than that of the fair scheduler. This may be due to the large jobs being pushed behind the small jobs and taking a longer time to finish.

B. Small jobs come after large jobs

In this experiment, we submit all small jobs after large jobs. The test bed contains one master node (No.0) and four slave nodes (No.1 ~ No.4). We submit 10 large jobs, wait for 5 seconds, and then submit 10 small jobs. The time interval between two submission is used to ensure that the jobs submitted earlier can be initialized before those submitted later so that all the jobs are in the right order we want. Figure 8 shows the results of the average job process time of two kinds of jobs.

We can see that the small jobs can get 39% decrease of process time with the SRT algorithm. And the job process time of the large jobs is nearly the same under two schedulers.

C. Scalability

In this experiment, we repeat the first experiment under different number of slave nodes to see the scalability of the SRT scheduler. We conduct four groups of tests, each group has 4, 6, 8, 10 slave nodes respectively. For example, in the fourth group of tests, the test bed contains one master node (No.0) and ten slave nodes (No.1 ~ No.10). In each test we submit 40 small jobs and 40 large jobs. Each group of tests are repeated for twenty times. Figure [9,10] show the results of the average job process time of two kinds of jobs.

We can see that the SRT scheduler still uses less time to finish the small jobs than the fair scheduler does. With the number of slave nodes getting larger, the process time of the small jobs of two schedulers is getting closer. This is a normal phenomenon, since we can assume that we have thousands of slave nodes, then all the tasks will be running at the same time, therefore, the process time of each job is invariable no matter what scheduler we use.
D. Apply the SRT algorithm to the capacity scheduler

The capacity scheduler is another popular scheduler of Hadoop, and we apply the SRT algorithm to it and see how it can reduce the process time of the small jobs. As we see from the scenario of Fig. 11, the SRT algorithm can perfectly reduce the process time of the small jobs when working with the capacity scheduler.

V. CONCLUSIONS

In this paper, we propose a simple algorithm for scheduling the reduce tasks based on the shortest remaining time of the map tasks. We implement our algorithm by applying the SRT algorithm to the fair scheduler and the capacity scheduler, which have been proved by several experiments. The results of the experiments show that the SRT algorithm decreases the reduce time and process time of the small jobs without affecting the performance of large jobs. However, the process time of total jobs of the SRT scheduler is a little more than that of the fair scheduler. In the future, we will improve our algorithm by considering the time of the reduce phase and using a more precise method to estimate the remaining time of both the map and the reduce phases. We also plan to consider several high-level objectives, such as the intelligent scheduler based on genetic algorithm and energy saving.

ACKNOWLEDGMENT

This work was supported partially by National High-Tech Research and Development Program of China under Grant No.2011AA040101 and was jointly funded by the Fundamental Research Funds for the Central Universities China No.FRF-MP-12-007A.

REFERENCES


**Xiaotong Zhang** received the M.S., and Ph.D. degrees from University of Science and Technology Beijing, in 1997, and 2000, respectively. He was a Professor in the Department of School of Computer and Communication Engineering, University of Science and Technology Beijing, from 2000 to 2013. He was as a visiting scholar supervised by Prof. Liang Cheng in LONG Lab of Deportment of Computer Science and Engineering of Lehigh University during 2010. His industry experience includes affiliation with Beijing BM Electronics High-Technology Co., Ltd. from 2002 to 2003, where he worked on digital video broadcasting communication systems and IC design. His industrial cooperation experience includes BLX IC Design Co., Ltd, NorthCommunications Corporation of PetroChina, and Huawei Technologies Co., Ltd.Etc. His research includes work in quality of wireless channels and networks, wireless sensor networks, networks management, cross-layer design, resource allocationof broadband and signal processing of communication.

**Bin Hu** is a student of School of Computer & Communication at University of Science and Technology Beijing. He pursued his Ph.D. in the field of Cloud Storage platforms and Massive Data Mining.

**Jiafu Jiang** is currently a graduate student and working towards his M.E. degree at University of Science and Technology Beijing, China. His major is Computer Science and Technology. His current research interest includes cloud computing and parallel computing.

© 2014 ACADEMY PUBLISHER