Scale-adaptable Recrawl Strategies for DHT-based Distributed Web Crawling System

Xiao Xu[1], Weizhe Zhang[1], Hongli Zhang[1], Binxing Fang[1]

[1] School of Computer Science and Technology, Harbin Institute of Technology, Harbin China
smilestor@gmail.com,
{zwz, zhl}@pact518.hit.edu.cn,
bxfang@ict.ac.cn

Abstract. Large scale distributed Web crawling system using voluntarily contributed personal computing resources allows small companies to build their own search engines with very low cost. The biggest challenge for such system is how to implement the functionalities equivalent to that of the traditional search engines under a fluctuating distributed environment. One of the functionalities is incremental crawl which requires recrawl each Web site according to the update frequency of each Web site's content. However, recrawl intervals solely calculated from change frequency of the Web sites may mismatch the system's real-time capacity which leads to inefficient utilization of resources. Based on our previous works on a DHT-based Web crawling system, in this paper, we propose two scale-adaptable recrawl strategies aiming to find solutions to the above issue. The methods proposed are evaluated through simulations based on real Web datasets and show satisfactory results.

Keywords: distributed Web crawling, search engine, incremental crawl, DHT.

1 Introduction

Web search services are becoming more and more important in everyone's daily life. Their availability and effectiveness largely depend on the efficiency of the underlying crawling systems. Nowadays, due to the huge evolution of the Web in the last 10 years, building a practical and effective search engine system has become an extremely complicated task involving both intelligence and funds which keeps smaller companies out of the door.

Inspired by the concept of internet computing[1] and SETI@home[2] , large scale distributed Web crawling systems[3-5] (DWC systems for short) has been applied in practice. The basic concept of these systems is using the personal computing resources voluntarily contributed by the internet users to retrieve Web content from the internet itself avoiding the maintenance costs of huge computer clusters. The system developers either feedback part of the system's commercial profit or provide distinctive services in order to attract the computer-contributor's participation.

In the work of [6] , we have proposed a DHT-based Web crawling system which solves the node churn and load balancing problems. Like some of the existing researches on the Web crawlers published in the past[7-11] , we assumed the system is handling a periodic crawl[12] . Under this assumption, the crawling system visits the whole web until its collection reaches a desirable number of pages. However, this assumption is not quite practical for the commercial search engines which, instead, adopt the idea of incremental crawl[12] . The current main information publishers: news sites, BBS sites and blog sites are generating new Web content in a very high speed. Even the content on a fixed URL changes constantly due to the changes of the DOM structure and the increase of comments and replies[13] . Therefore, in order to keep its indexed data up-to-date, the search engine has to apply different recrawl intervals to different Web site, instead of periodically crawl all the Web sites.

Our goal in this paper is to design a fully distributed Web crawling system with the ability to incrementally crawl the Web so that the stored content of all the discovered Web sites can be continuously updated. The existing incremental crawl strategies[12, 14-18] are adopting recrawl intervals according to the change frequency, relevance and information longevity of the Web pages. The intervals may range from minutes to days so that the most important sites (such as news sites which has high update rate) can be scanned frequently meanwhile the rests are not in order to save the networking, storage and computing costs. When calculating the recrawl intervals, the existing works are all under the assumption that the system has a fixed scale (most of the works are using single-machine system). Things are different in the context of internet computing. As the real-time scale of the system is difficult to derive, the recrawl intervals calculated by the existing strategies are very likely to mismatch the system's current capacity which leads to two kinds of consequences:

Requirement exceeds system's capacity (REC). The recrawl tasks are submitted so frequently that the crawlers within the system cannot handle all of them. Many tasks are queued, timed out even removed due to the crawlers' abnormal departure. The actual update rate is below the expected recrawl rate.

Requirement underestimates system's capacity (RUC). The system has enough capacity to execute all the recrawl tasks. But the capacity is not fully used, which prevents the system from achieving a higher update rate.

In this paper, we firstly make a brief summary of our former works. Unlike the traditional dedicated crawlers, our crawler is designed to be run on contributed personal computers. We have to restrict the crawler from consuming too much bandwidth and local resources. A more detailed crawler implementation is presented including basic data structures involved in the recrawl.

Secondly, based on the system design, we propose two scale-adaptable recrawl strategies. The concept of the two strategies is periodically using the system's successful update frequency (indicating the system scale) to derive a new recrawl rate. The strategies can be used as the additional steps to the existing recrawl strategies which solely focus on the characteristics of the Web pages. The two methods are evaluated through a series of experiments simulating the two situations: REC and RUC. The results show our strategies' effectiveness.

The rest of the paper is organized as follows: Section 2 presents our system design. Section 3 proposes the self-adaptive update strategies. Section 4 provides the evaluations. Section 5 we conclude the paper.

2 System Design

First, we briefly outline our existing works presented in [6] . The paper mainly describes the system architecture of a new DWC system. The system is proposed to solve two issues:1) Scalability which is crucial for a distributed system; 2) Download time which various due to the network localities.

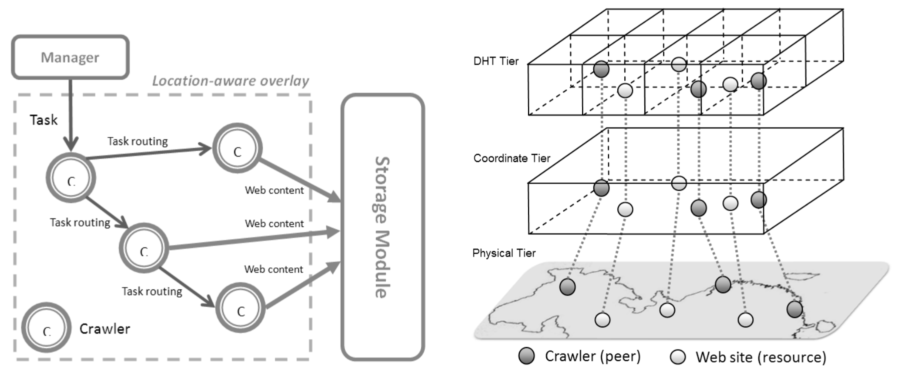


Fig. 1. The architecture of the proposed DWC system. From left to right, the first figure shows the main system modules; the second figure shows the 3-tier mapping.

As shown in Fig.1, the system consists of a Manager node and a network-location-aware overlay containing all the crawlers. The task of the manager is to maintain the Web site (or channel which will be introduced later in this paper) database and submit recrawl tasks to the overlay. The network-location-aware overlay is achieved by adopting a 3-tier mapping. As is demonstrated in Fig.1, from bottom to top, the 3 tiers are physical tier, coordinate tier and DHT tier. On the coordinate tier, a network coordinate service (NC) maps the physical locations (measured using the network latencies) of the crawlers and the Web hosts to the coordinates in a multi-dimensional network coordinate space. Using the coordinates as IDs and keys, on the DHT tier, all the crawlers join a DHT overlay (currently implemented on CAN), while the Web hosts are inserted to the overlay. By combining the network coordinate space and DHT, we finally achieve a network-location-aware task scheduling method. Because both the DHT tier and coordinate tier are self-organized, the manager doesn't need to choose to which crawler a task should be assigned, as well as monitor the state of the crawling tasks and the behavior of the crawlers. The Web pages downloaded are not stored on the crawler side. They are sent to a distributed storage system (the storage module) after a series of pre-processing.

Because the detailed design of the manager and the crawler which is crucial for this paper is not presented in [6] , we describe them in the following literature.

2.1 Task Design

A task is the description of a or a part of a Web host which we call a channel. A channel may represent 3 kinds of Web entries: 1) the whole Web host (such as "sports.sina.com.cn"); 2) a Web host's sub-directory (such as "news.sina.com.cn/w/");

3) a set of sub-directories on a Web host. The manager maintains a database of channels. When a channel needs to be recrawled, the manager patch the channel into a recrawl task and submit the task the overlay which finally appoints a crawler to run the task. On receiving a recrawl task, the crawler simply run the task according to the task's fields. A task contains the following fields:

The list of seed URLs. The seed URLs are used as the starting point of the crawl. Mostly these URLs points to the index pages. The list also contains the URLs (pointing to this channel) that were discovered during the crawl of other channels (we call these URLs the inter-site URLs).

The list of regular expressions to be used as the URL filter. The crawler only downloads the URLs which matches the regular expressions. In case there are many kinds of expressions, we use a list to include all of them.

The number of Web pages derived according to historical record. The number is used to decide the size of URLseen data structure on the crawler. As all the downloaded data are transferred to it, the storage module knows the completion of each crawl task and feeds back the latest statistics of the channels to the manager. The feedbacks are also used later in Section 3.

The maximum depth to crawl. Defines how the crawl process stops.

The array of page content digest. The array contains the 32bit SHA-1 digests of all known pages which is also fed back by the storage module. The array is used to filter the pages that haven't changed since the last crawl so that the crawler doesn't have to extract their contents and send them back to the storage module. According to our experience, the array occupies at most 400kB.

The digests of discovered inter-site URLs. On discovering inter-site URLs the crawler first checks if they are in this discovered set. If true, the URLs are ignored, else the URLs are transferred back to the storage module with the downloaded contents and finally fed back to the manager.

2.2 Crawler Design

Different from the dedicated crawlers, our crawler is designed to be run on the machines contributed by the ordinary internet users who may want to do other things besides crawling. As a result, small data structures, restricted downloading scheme have to be applied. The whole program is run on a low system priority. As is shown in Fig.2, a crawler consists of 3 kinds of modules: task queue, task manager and task threads.

Task queue. Crawl tasks submitted to the crawler are first put into the task queue by the task manager.

Task thread. Each task thread is a complete crawler thread responsible to download a channel. As the target is only one channel, the size of each data structure involved is quite small compared with the dedicated crawlers. A task thread contains 4 main sub-modules: URLtodo, URLseen, Pageseen, Robotsfilter and downloader.

1) **Downloader.** The Downloader is responsible for downloading the Web pages. In order to limit the bandwidth usage of each task thread, the downloader opens one

TCP persistent connection to the target Web host and doesn't use pipelining. At the first stage of downloading, the downloader first downloads the Web site's robots.txt and builds a robots filter. After downloading each Web page, the downloader extracts the URLs in it. All the extracted URLs are matched with the Robotsfilter and the URLseen. If the URLs can pass the two filters, they are inserted into the URLtodo. After downloading, the Web pages are filtered by Pageseen. Then the downloader write the textual content to a temporary disk file.

2) **URLtodo.** The URLtodo is a FIFO queue used to store the URLs to be downloaded. In order to avoid the queue becoming too long, we restrict the length of the queue to 1000. The URLs exceeds the limit is temporarily written into disk files.
3) **URLseen.** The URLseen is an array of integers storing the SHA-1 hash of the already discovered URLs in the current crawl process.
4) **Pageseen.** The Pageseen is the array of page content digest mentioned in 2.1. We use sequential search to find if a Web page is in the Pageseen.
5) **Robotsfilter.** The Robotsfilter is a list of forbidden paths on the Web host.

When the downloading process reaches the maximum depth to crawl, the downloader stops, and transforms to an uploader. The uploader uploads the extracted Web contents (stored in the disk files by the downloader) and the statistics of the channel to the storage module. After that, the task thread terminates itself and release the space of all its data structures.

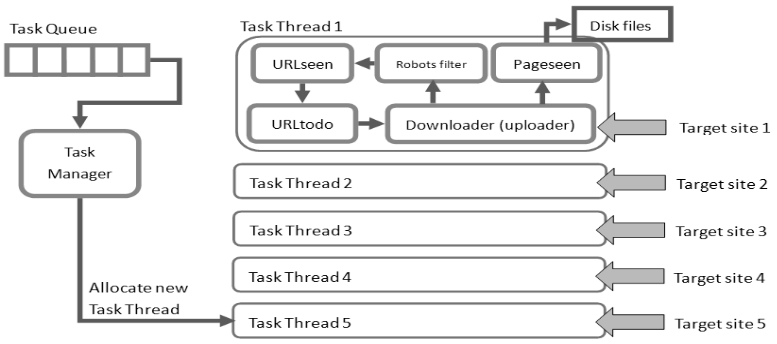


Fig. 2. The data structures on a single crawler (machine). A crawler consists of 3 main part: the task queue, the task manager and the task threads. Each task thread implements a crawler responsible to download a certain channel.

Task manager. Task manager is responsible for allocating the task threads. It ensures that at any time there are only a restricted number of task threads running in order to save the cost of system resources. When a task thread terminates, task manager fetches a new task from the task queue and allocate a new task thread to run it.

3 The Scale-adaptable Recrawl Strategies (SARSS)

In our system, channels are classified according to their change frequency into several ranks we call the priority groups (PGPs). Channels with higher change frequency are assigned to PGP with higher rank. The average change frequency of the channels in each PGP is used as the PGP's initial recrawl time interval. Therefore, channels with higher change frequency should have shorter recrawl intervals than the others. Here, we don't focus on how the change frequency is derived as there's previous works[12, 14-18] concerning the issue. Instead, we focus on how the system scale affects the PGP's initial recrawl time interval. As is discussed in Section 1, the recrawl intervals solely derived from the characteristics of the Web sites may mismatch the system's capacity (the proof is presented as the simulation result in Section 4). As a result, we have to develop an algorithm to automatically adjust the recrawl intervals to the system's scale. The initial time intervals are only used as the start point of the adjusting process. However, one cannot directly calculate the PGP's recrawl intervals according to the scale of the system as there are no reliable real-time statistics on the system scale. Thus the adjustments have to be done according to the records of the manager. In this section, we propose two scale-adaptable recrawl strategies to automatically adjust all the channels' recrawl intervals in order to maintain high update rate. In the following literature, an update of a channel indicates a successful recrawl in which the content of the channel stored in the storage module can be updated. Due to the node churn within the system, not all recrawls are successful. Therefore, the update rate does **NOT** equal to the recrawl rate. Instead it indicates the rate of successful recrawls.

3.1 Iterated Interval Reassignment (IIR)

The basic idea of the IIR strategy is to periodically reassign each PGP a new update interval derived from the PGP's latest update record. The reassignment process is called when 50% channels with the highest rank has completed more than 5 successful recrawls (updates) since the last reassignment. The process is run in an isolated thread so that it doesn't stop the system's crawl sequence.

First, we add 3 records to each channel. 1) historyInterval (*hInterval*): the average time gap between updates since the last reassignment; 2) historyRecrawlCount (*hSubCount*): records the number of recrawl tasks (on that channel) submitted to the crawlers since the last reassignment; 3) historySuccessCount (*hSucCount*): records the number of updates since the last reassignment. Then, each PGP is reassigned a new recrawl interval which is the mean of all channel's historyIntervals. To this point, we only get intervals from the historical records. In order to achieve a higher update rate, we add a encourage mechanism to the above process. We define the *successRate* of a PGP as quotient of (*hSucCount* / *hSubCount*) and add a static variable *lastSuccessRate* to record the *successRate* calculated during the last reassignment. If the *successRate* is larger than *lastSuccessRate*, the system would make a positive attempt by decreasing the update interval by a pre-configured parameter δ (<1). The final step is to check the value of each PGP's new recrawl interval to ensure that the PGPs with lower ranks never have smaller recrawl intervals than the PGPs with higher ranks, at the same time, to prevent the recrawl intervals from growing too large. The above process is written as psuedocode below.

The Psuedocode of IIR. The code implements the main function *InheritedIntervalReassignment* of the IIR. We assume K PGPs, and In PGP[K][], the first dimension indicates the priority number 0..K-1, the second dimension stores the list of channels under a certain priority number

```
const EncourageRate = δ;
static LastSuccessRate[K] = {0, 0, …, 0};
program InheritedIntervalReassignment ( PGP[K][] )
  var  P:-1..K-1;SuccessRate[K];ActualExecInterval[];NewIntervals[K]:Real;
       RecrawlCount, SuccessCount, C: Integer;
  begin
    P := -1;
    repeat
      P:= P + 1; C := -1;
      repeat
        C := C + 1; ActualExecInterval[C] := PGP[P][C]. hInterval();
        RecrawlCount := RecrawlCount + PGP[P][C].hSubCount;
        SuccessCount := SuccessCount  + PGP[P][C].hSucCount;
      until C = number of channels in PGP[P]
      NewIntervals[P] := mean(ActualExecInterval[0..C-1]);
      SuccessRate[P] := SuccessCount / RecrawlCount;
      if SuccessRate[P] > LastSuccessRate[P]; then { encourage };
        NewIntervals[P] :-= NewIntervals[P]*EncourageRate;
      end if
      LastSuccessRate[P] = SuccessRate[P];
    until P = K-1;
    call checkPriorities(NewIntervals[]);
    Set NewIntervals[] as manager's recrawl interval;
end.
```

The Psuedocode of *checkPriorities*. The code is the implementation of function *checkPriorities* which is called by *InheritedIntervalReassignment*.

```
const SmallestMultiple = σ ;
const BiggestMultiple = μ ;
program CheckPriorities ( NewIntervals[K] )
  var upperLimit: Real; P: Integer;
  begin
    P := 0;
    repeat
      P := P + 1;
      if NewIntervals[P] < NewIntervals[P-1]; then
        NewIntervals[P]:=NewIntervals[P-1]*SmallestMultiple;
      end if
      upperLimit := NewIntervals[P-1] * BiggestMultiple;
      if NewIntervals[P] > upperLimit; then
        NewIntervals[P] = upperLimit;
      end if
    until P = K-1;
end.
```

3.2 Iterated Capacity Reassignment (ICR)

The basic idea of the ICR strategy is to periodically evaluate the system's capacity and distribute the capacity to each PGP. In this method, the recrawl interval of each PGP shares a common divisor ΔT_0^{PGP} which is the recrawl interval of the PGP with the highest rank (rank 0) and ω_i which is the multiple between the initial recrawl interval of the *i*th and the *(i+1)*th PGP. Then, the recrawl interval of the PGP with rank *k* equals to $\Delta T_k^{PGP} = \prod_{j=0}^{k-1} \omega_j \times \Delta T_0^{PGP}$. ΔT_0^{PGP} is set to Δt at the system's initial stage. We define $\Delta T_0^{PGP} = \theta \times \Delta t$. So in each ICR process our goal is to derive a new θ.

In the following literature, we present the derivation of how to calculate θ during each assignment process. All the records used in 3.1 are inherited. We assume that there are *N* channels in the manager's database; the channels are classified into *S* ranks (PGPs); the *j*th channel has an update interval ΔT_j; the channel's update rate is $\lambda_j = 1/\Delta T_j$; the channel's average download time is $\bar{t}_l$. Here we add a record to each channel to reserve its download times during the updates. According to [6] , our scheduling algorithm guarantees the channel's recrawl task is always assigned to a crawler with low latency. Therefore, unless some exceptional event such as congestion occurs, there won't be significant difference between the download times.

We also assume that during ΔT_j, there are on average *M* crawlers in the system; the number of task threads on the *i*th crawler equals to CL_i. If we want to update each channel *j* on schedule, the best choice is to complete all the recrawl tasks submitted during ΔT_j. Under the assumption that the system's DHT-based scheduling can sufficiently balance the load of all crawlers, we have the following formula.

$$\sum_{l=1}^{N} (\lambda_l \times \bar{t}_l) \times \Delta T_j \times \frac{1}{\sum_{k=1}^{M} CL_k} \le \Delta T_j \tag{1}$$

We replace "$\le$" with "=", then (1) can be transformed to:

$$\sum_{l=1}^{N} (\lambda_l \times \bar{t}_l) = \sum_{k=1}^{M} CL_k \tag{2}$$

The right side of (2) is not an accurate value as the set of crawlers subjects to constant change. But, it can be treated as an approximation of the latest system capacity. To implement an iteration process, we replace the right side of (2) with a history-related value derived from the record of each channel's previous *L* updates.

$$\frac{1}{\theta \times \Delta t} \times \sum_{k=0}^{S-1} \left(\frac{1}{\prod_{j=0}^{k-1} \omega_j} \sum_{PGP(k)} \bar{t}_l \right) = history\left(\sum_{l=1}^{N} (\lambda_l \times \bar{t}_l) \right), \quad \prod_{j=0}^{-1} \omega_j = 1 \tag{3}$$

$$\theta = \sum_{k=0}^{S-1} \left(\frac{1}{\prod_{j=0}^{k-1} \omega_j} \sum_{PGP(k)} \bar{t}_l \right) \Bigg/ \left(\Delta t \times history\left(\sum_{l=1}^{N} (\lambda_l \times \bar{t}_l) \right) \right), \quad \prod_{j=0}^{-1} \omega_j = 1 \tag{4}$$

Then θ can be calculated according to (4). In the implementation, we also add the encourage mechanism and checkPriority function at the end of the process.

4 Experiments

All the experiments are done under complicated simulations. First, we adopt P2Psim's King dataset[20] to simulate the network latencies between the crawlers and the Web hosts. Because the size of the dataset (only 1740 nodes) is relatively small, we map the nodes into a coordinate space using Vivaldi[21] and replicate a set of new nodes around each original node. The distance between the original nodes and its replica is a random value scaling from 10% to half of the distance between the original node and its nearest neighbor. Accordingly, we generate two larger datasets: 1) **16000 Web hosts and 1400 crawlers.** The original 1740 nodes are divided into 1600 and 140 and we replicate 10 new nodes around each node. 2) **16000 Web hosts and 2800 crawlers.** Replicating 1 more node around each of the 1400 crawler nodes.

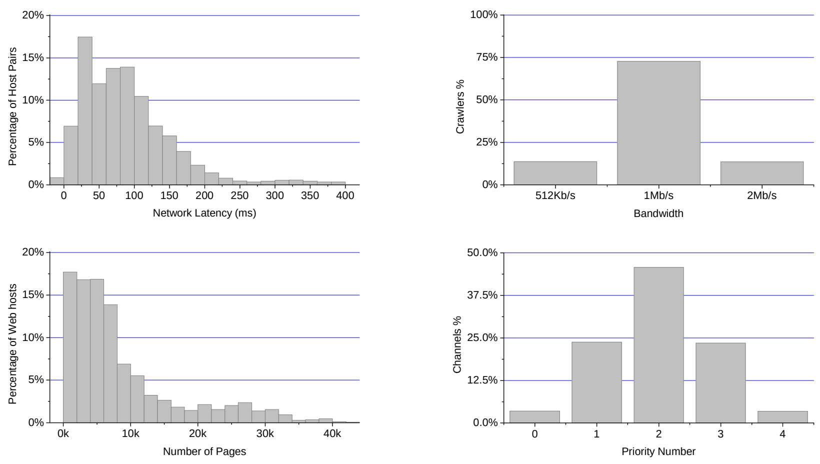


Fig. 3. Statistics on the datasets. left to right, top to bottom, the figures show the distribution of: network latencies, bandwidth, page numbers and number of channels on each PGP.

Second, the bandwidth assigned to each crawler has 3 levels: 512Kb/s, 1Mb/s and 2Mb/s. The number of crawlers under each level is decided according to standard normal distribution 1.5*N(1, 0.5). Each Web host is treated as a channel. The number of Web pages on each channel is assigned according to a Web dataset we collected in 2008. The dataset is the result of 1738 crawling tests containing 15,187,511 URLs. Because the dataset only contains 1738 Web hosts, we also have to increase its size using replication. We divide the set of Web hosts (channels) into 5 PGPs. The number of channels under each rank is decided by 1.25*N(2, 1). To each channel in the *i*th PGP, we assume a change frequency obeying Poisson distribution with $\lambda = 1/\theta_i$, $i = 0,1,2,3,4$. θ_i indicates the average time interval when most of the Web pages have changed on the channel. For the 5 PGPs, we assume $\theta_0 = 60\min$, $\theta_1 = 120\min$, $\theta_2 = 240\min$, $\theta_3 = 480\min$, $\theta_4 = 960\min$. All the above setups are illustrated in Fig.3.

The simulation is based on time steps. Each step represents 1 minute. The whole simulation involves 90000 steps (1500 hours, 62.5 days). The crawlers are all added to the system at the first step. The living time of each crawler obeys the normal distribution N(1000, 200) which means 95% crawlers' living time is between 608 steps and 1392 steps. After the living time, 1/3 crawlers leave the system and notify the leaving event to their neighbors; the tasks both running and waiting are migrated to the neighbors; the rest 2/3 crawlers leave the system without notifying. The tasks on these crawlers are completely lost. Each dead crawler will rejoin the overlay within a time gap obeying the normal distribution N(600, 200). The maximum number of task threads on each crawler is set to 5. The maximum length of task queue is set to 5. When the task queue is full the newly submitted task is forwarded to a less loaded crawler according to [6] . If a task's waiting time exceeds the time limit which equals to the channel's recrawl interval, it is canceled by the crawler.

The download time of each Web page is calculated by adding the RTTs, data transfer time and a wait time (fixed to 200ms) involved. The RTT equals to 2 times the network latency in order to simulate the time cost of a GET request under HTTP persistent connection without pipelining. The data transfer time is calculated through dividing the size of the Web page by the speed. The speed equals to the bandwidth of the crawler divided by the number of task threads. Because the 1-minute-per-step setup shields us from directly simulating each downloading process. Therefore, instead, the total download time of all pages on a Web host is pre-calculated once it is submitted to a certain crawler.

4.1 Evaluation Criterias

In the simulation, 3 recrawl strategies are compared. They are 1) FIXED: the recrawl interval is fixed to the change frequency of the Web site, ignoring the fluctuation of the system's capacity. 2) IIR. 3) ICR. We use the following evaluation criterias:

The total number of recrawl tasks. The number indicates the load of the manager. Under both REC and RUC, the value should be minimized in order to reduce the manager's unnecessary load.

The number of queued tasks. The number indicates the number of tasks contained in the crawlers' task queue. Under both REC and RUC, it should be minimized.

The system's throughput. The value indicates the total download rate of the system. Under both REC and RUC, the value should be maximized so that all the capable crawlers are efficiently utilized.

Update quality. To each channel, its change rate (the reciprocal of the channel's change frequency, and the FIXED strategy is using this change rate as the channel's recrawl rate) is labeled as $\lambda_i^{\exp ected}$. In the real run, due to the node churn, the actual update rate (λ_i^{actual}) is always lower than $\lambda_i^{\exp ected}$. Assuming that each PGP *i* has a weight R_i, then the update quality calculated by Formula(5) indicates how well the recrawl strategy can perform against the channels' actual change frequency. Here we apply weights (used in Table 2 and 4) to PGPs (4~0): $2^4=16$, $2^3=8$, $2^2=4$, $2^1=2$, $2^0=1$. Under both REC and RUC, the update quality should be maximized.

$$updateQuality = \sum_{i=1}^{N}\left(R_i \times \frac{\lambda_i^{actual}}{\lambda_i^{\exp ected}}\right) \Big/ N \tag{5}$$

In addition, there are other metrics such as the number of task migrations, load variations, etc. Due to the size of the paper, we don't present them.

4.2 The Case of REC

We firstly examine the performance of IIR and ICR under the case of REC. Here we choose the dataset containing 16000 channels and 1400 crawlers.

Table 1. Number of recrawl tasks and average recrawl intervals (steps) under REC. The left 4 colums show the number of recrawls. The rest 4 colums show the average re-recrawl intervals.

Rank	FIXED	IIR	ICR	Rank	FIXED	IIR	ICR
4	0.905E6	0.251E6	0.522E6	4	60	216	104
3	3.295E6	1.212E6	1.719E6	3	120	326	230
2	2.979E6	1.512E6	1.499E6	2	240	473	477
1	0.795E6	0.471E6	0.393E6	1	480	811	972
0	0.056E6	0.038E6	0.027E6	0	960	1412	1964
Total	8.030E6	3.485E6	4.160E6	Mean	60	216	104

The left side of Table 1 shows that IIR and ICR dramatically reduce the number of recrawl tasks. To explain the reason, the right side of Table 1 shows the recrawl intervals. As the system's capacity cannot match the channels' change frequency (used as the recrawl frequency of the FIXED strategy), the IIR and ICR all increase the recrawl intervals of each rank. From the table, we observe that the IIR narrows the gaps between different ranks; meanwhile, the ICR widens them. However, the above characteristics don't cause continuous narrowing or widening. We observe that during the whole simulation process, the recrawl intervals don't change in linear patterns.

From Table 1, we observe that there are much more tasks submitted under FIXED strategy. However, such high recrawl rate doesn't lead to high efficiency. Under FIXED strategy approximately 70% tasks are waiting in the task queue. According to Fig.4, if we assume that the average number of online crawlers is 700, then the length of each crawler's task queue under FIXED strategy is about 17. In such situation, most queued tasks are timed out and cancelled. On the other hand, the lengths under IIR and ICR are about 2.8 and 1.4 which indicate 83% and 92% reduction on the task queue's memory cost. IIR has less queued tasks because IIR's recrawl intervals on rank 0 to rank 2 are larger than that of the ICR's.

The throughput in Fig.4 is calculated through dividing the total size of pages downloaded in 10000 steps by the total time cost (including RTTs, data transfer time and wait time). Because the task threads are not always downloading pages, the throughputs don't equals to the number of crawlers multiplied with the bandwidth. Instead, they only equals to half of the product. In addition, the curves all present obvious decline in the first 20000 steps. This is due to the decline of the number of crawlers, since we added all the crawlers to the system at the first step. Moreover, although the number of tasks is significantly reduced under IIR and ICR, the throughput of the system doesn't change dramatically. The throughput under the ICR almost matches that under FIXED strategy. On the other hand, as IIR's recrawl intervals on rank 4 to rank 2 are larger, the throughput under IIR declines about 14%.

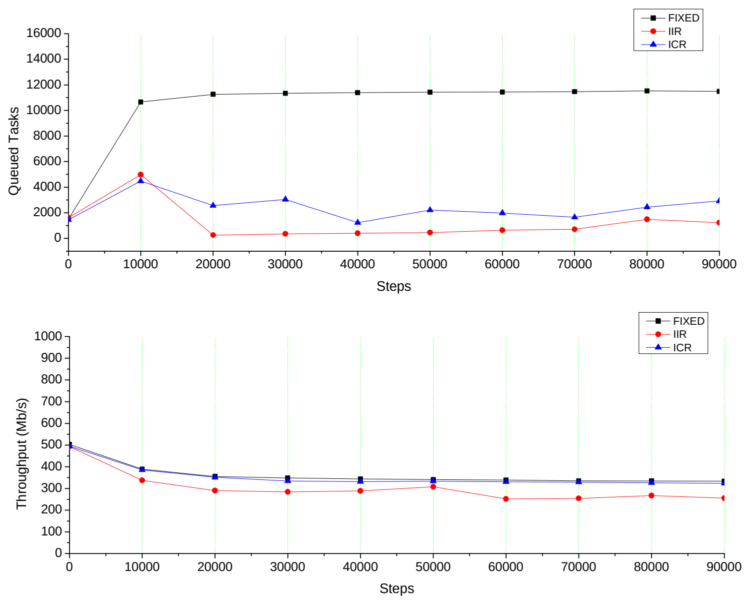


Fig. 4. The system performance throughout the simulation under REC. From left to right, the first figure shows the number of queued tasks; the second figure shows the system's throughput.

Table 2. Update quality under REC. The left three columns labeled "w" use the weights assumed in 4.1; the right three columns use 1 as each PGP's weight.

Rank	FIXED (w)	IIR (w)	ICR (w)	FIXED	IIR	ICR
Update Quality	1.9987	2.0266	2.0176	0.5571	0.4936	0.4331

Table 2 shows the update quality of the fixed recrawl strategy, IIR and ICR. As the FIXED strategy represents a very radical approach, its update quality can be treated as the maximum capacity the system can achieve under REC. IIR and ICR's weighted update qualities are very similar to that of FIXED, which means the update quality has been maximized. Meanwhile, their non-weighted update qualities are lower. We find that the reason is that, as the system's capacity cannot meet the required recrawl rate, both IIR and ICR decrease the recrawl rates of the lower ranked channels (which accordingly decreases the update rates) in order to ensure high update rate of the higher ranked channels. Since the non-weighted update qualities are calculated without concerning the channels' importance, they become inevitably smaller.

We also notice that even under IIR and ICR which has lower recrawl rate, the average success rate of the tasks is only 30% (the fact is also true under the case of RUC). The phenomenon is mainly caused by the crawler's non-notified departure in which all the tasks assigned to the crawler are completely lost. We consider this low rate a necessary cost under a fluctuating distributed environment. On the other hand, under FIXED strategy, since there are a lot more tasks timed out in the queue, the success rate become even worse and is less than 15%.

4.3 The Case of RUC

We secondly examine the performance of IIR and ICR under the case of REC. Here we choose the dataset containing 16000 channels and 2800 crawlers.

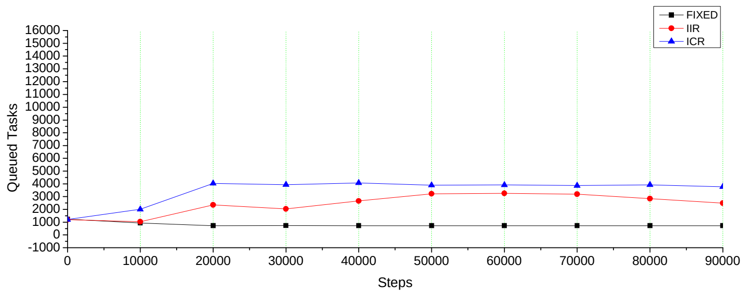
Table 3. Number of recrawl tasks and average recrawl intervals (steps) under RUC. The left 4 colums show the number of recrawls. The rest 4 colums show the average recrawl intervals.

Rank	FIXED	IIR	ICR	Rank	FIXED	IIR	ICR
4	0.905E6	0.738E6	0.975E6	4	60	74	56
3	3.295E6	3.506E6	3.428E6	3	120	110	112
2	2.979E6	3.388E6	3.805E6	2	240	214	190
1	0.795E6	0.943E6	0.691E6	1	480	399	544
0	0.056E6	0.075E6	0.052E6	0	960	782	1119
Total	8.030E6	8.650E6	8.953E6	Mean	287	246	284

Different from the case of REC, the number of recrawl tasks under IIR and ICR in Table 3 don't decline. Instead, they are increased by 8% and 11.5% compared with the FIXED strategy. The fact indicates that, under the case of RUC, in order to fully utilize the system's capacity, there's no way to reduce the load of manager. The bottleneck should be solved by either upgrading the manager's capacity or deploy a number of managers to distribute the loads. We also found that IIR and ICR perform differently on deciding the recrawl intervals. On one hand, the IIR increases the intervals of higher-ranked (rank 4) channels and decrease that of lower-ranked (rank 3 to 0) channels. On the other hand, ICR decreases the intervals of higher-ranked (rank 4 to 2) channels and increase that of lower-ranked (rank 1 to 0) channels. Through the comparison we conclude that ICR is more likely to ensure the high re-scrawl rate of high-ranked channels, while IIR performs better if taking all ranks into consideration.

Both Fig.5 shows the increase in the number of queued tasks and system throughput under IIR and ICR. To our surprise, the increase in Fig.5 (300%-400% in the left figure and 200% in the right figure) significantly exceeds the increase (8%-11.5%) in Table 3. On one hand, we consider the increase in queued tasks no harm to the crawlers. If we assume that the average number of online crawlers is 1400 (half of the 2800 crawlers), then the length of each crawler's task queue under FIXED, IIR and ICR is about 0.36, 2 and 3. The numbers are low enough to prevent the time out. On the other hand, by re-adjusting the recrawl intervals of each rank, the system's download capacity is fully used. Under IIR and ICR, the system's throughput reaches 2 times of the throughput achieved under the case of REC (which also indicates the growth in the number of crawlers). But under the FIXED strategy, the system's throughput doesn't change compared with REC.

Table 4 shows that IIR and ICR significantly increase the weighted update quality by 91% and 72.6% compared with FIXED strategy, which is satisfactory. We further notice that the non-weighted update qualities of IIR and ICR exceeds 1, which means the update rate exceeds the channels' change frequency. Nevertheless, we believe the freshness of the Web contents can still be improved since the Web pages on the channels don't change all at once (instead, their change frequency is supposed to obey the Poisson distribution[12]). Another important issue is that if a crawler visits a Web host too often, it brings additional load to the Web host. A feasible solution is to add an upper limit to the recrawl rate (e.g. 2 times of the channel's change frequency) and, if the system's capacity permits, submit more channels to crawl.



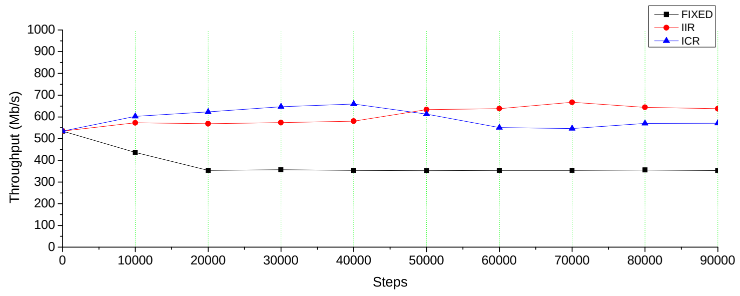


Fig. 5. The system performance throughout the simulation under RUC. From left to right, the first figure shows the number of queued tasks; the second figure shows the system's throughput.

Table 4. Update quality under RUC. The left three columns labeled "weighted" use the weights assumed in 4.1; the right three columns use 1 as each PGP's weight.

Rank	FIXED (w)	IIR (w)	ICR (w)	FIXED	IIR	ICR
Update Quality	4.7873	9.1474	8.2611	1.0013	1.9827	1.9701

5 Conclusions

In this paper, we propose the design of a distributed incremental crawling system. The system is designed to utilize the machines contributed by ordinary internet users as the crawlers. Based on this design, we propose two new recrawl strategies to adapt the recrawl frequency to the system's capacity in order to make full use of the contributed machines. The strategies show their efficiency under a series of simulations.

Acknowledgments. This paper was partially supported by the National Natural Science Foundation of China under Grant No.60703014; the National Grand Fundamental Research 973 Program of China under Grant No.G2005CB321806; the Specialized Research Fund for the Doctoral Program of Higher Education, SRFDP No. 20070213044; the National High-Tech Research and Development Plan of China under Grant Nos. 2007AA01Z442; the National High-Tech Research and Development Plan of China under Grant Nos. 2009AA01Z437.

References

1. Foster, I.: Internet Computing and the Emerging Grid. Nature. 2000.
2. Werthimer, D., Cobb, J., Lebofsky, M., Anderson, D., Korpela, E.: SETI@HOME---Massively Distributed Computing for SETI. Comput. Sci. Eng., vol. 3, pp. 78--83. (2001)
3. YaCy Distributed Web Search, http://yacy.net
4. FAROO Real Time Search, http://www.faroo.com
5. Majesti-12: Distributed Web Search, http://www.majestic12.co.uk
6. Xu, X., Zhang, W.Z., Zhang, H.L., Fang, B.X., Liu, X.R.: A Forwarding-based Task Scheduling Algorithm for Distributed Web Crawling over DHTs. In: the 15th International Conference on Parallel and Distributed Systems (ICPADS'09), pp. 854--859. IEEE Computer Society, Shenzhen, China (2009)
7. Heydon, A., Najork, M.: Mercator: A Scalable, Extensible Web Crawler. World Wide Web, vol. 2, pp. 219--229. (1999)
8. Boldi, P., Codenotti, B., Santini, M., Vigna, S.: Ubicrawler: A Scalable Fully Distributed Web Crawler. Software—Practice & Experience, vol.3(8), pp. 711--726. (2004)
9. Loo, B.T., Cooper, O., Krishnamurthy, S.: Distributed Web Crawling over DHTs. Technical report, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley 01 Feb (2004).
10. Singh, A., Srivatsa, M., Liu, L., Miller, T.: Apoidea: A Decentralized Peer-to-Peer Architecture for Crawling the World Wide Web. In: the SIGIR Workshop on Distributed Information Retrieval, pp. 126--142. (2004)
11. Lee, H.T., Leonard, D., Wang, X., Loguinov, D.: IRLbot: Scaling to 6 Billion Pages and Beyond. In: the 17th International Conference on World Wide Web, pp. 427–436. (2008)
12. Cho, J., Garcia-Molina, H.: The Evolution of the Web and Implications for an Incremental Crawler. In: the 26th International Conference on Very Large Data Bases (VLDB'00), pp. 200--209. San Francisco. (2000)
13. Adar, E., Teevan, J., Dumais, S.T., Elsas, J.L.: The Web Changes Everything: Understanding the Dynamics of Web Content. In: the 2nd ACM International Conference on Web Search and Data Mining (WSDM'09), pp. 282--291. ACM, Barcelona, Spain. (2009)
14. Edwards, J., McCurley, K., Tomlin, J.: An Adaptive Model for Optimizing Performance of an Incremental Web Crawler. In: the 10th International Conference on World Wide Web, pp. 106--113. ACM, Hong Kong. (2001)
15. Wolf, J.L., Squillante, M.S., Yu, P.S., Sethuraman, J., Ozsen, L.: Optimal Crawling Strategies for Web Search Engines. In: the 11th International Conference on World Wide Web, pp. 136--147. ACM, Honolulu, Hawaii, USA. (2002)
16. Cho, J., Molina, H.G.: Effective Page Refresh Policies for Web Crawlers. ACM Trans. Database Syst., vol. 28, pp. 390–426. (2003)
17. Pandey, S., Olston, C.: User-centric Web Crawling. In: the 10th International Conference on World Wide Web, pp. 401--411. ACM, Chiba, Japan. (2005)
18. Olston, C., Pandey, S.: Recrawl Scheduling based on Information Longevity. In: the 17th International Conference on World Wide Web, pp. 437--446. ACM, Beijing, China. (2008)
19. Brin, S., Page, L.: The Anatomy of A Large-scale Hypertextual Web Search Engine. Computer Networks and ISDN Systems, vol. 30, pp. 107--117. (1998)
20. P2PSim-Kingdata. http://pdos.csail.mit.edu/p2psim/kingdata/
21. Dabek, F., Cox, R., Kaashoek, F., Morris, R.: Vivaldi: A Decentralized Network Coordinate System. In: SIGCOMM'04, pp. 15--26. ACM, Portland, Oregon, USA. (2004)