International Journal of Computer Science and Engineering (IJCSE)
ISSN(P): 2278-9960; ISSN(E): 2278-9979
Vol. 6, Issue 1, Dec - Jan 2017; 1-34

International Academy of Science, Engineering and Technology
Connecting Researchers; Nurturing Innovations

FOCUSED WEB CRAWLER DEVELOPMENT CHALLENGES: ECCRAWLER

FURKAN GÖZÜKARA & SELMA AYŞE ÖZEL

Department of Computer Engineering, Faculty of Engineering and Architecture,

Çukurova University, Balcalı, Sarıçam, Adana, Turkey

ABSTRACT

Nowadays, the importance of focused web crawlers is more than any time before. As the web has become massive and spam my, it is now essential to have focused web crawlers that can crawl only the targeted websites and obtain the necessary information. Instead of relying on the available public general web crawlers, today, developing a focused web crawler for the targeted web pages is preferred to increase success of information retrieval. In this paper, the challenges encountered and the proposed solutions to attempt these problems are presented, while developing an original hand-crafted, full scale, robust and effective focused web crawler for E-commerce sites, named as EcCrawler, which is developed in C# programming language by using .NET 4.5 framework and MS-SQL Server 2014 database management system. Most of the crawling challenges have been discussed before in the literature, however in this paper, practical implementation and .NET framework based solutions that includes thread pool initialization, exception handling, task parallelism, HTTP compression, duplicate web page resolution, number of concurrent connections to the same host, database communication, resource sharing between threads, etc. are presented and the proposed solutions are empirically evaluated. The experimental evaluation shows that applying the proposed solutions improve EcCrawler's crawling speed over 400% and UI responsiveness over 100%. The proposed solutions may be applicable to any software that is developed by using .NET framework.

KEYWORDS: NET Framework, Performance Tuning, Application Development, Multithreading, Web Crawling

INTRODUCTION

Web crawlers are software systems that are used to collect information from websites. Their main task is fetching websites, processing fetched source code and extracting new target hyperlinks to crawl. General web crawlers start with root URLs[1] and continue until crawling all of the hyperlinks that they can find. However, this task is tedious and not doable for small or even average scale applications. Only a fewlarge-scalecommercial general search engines (e.g. Google, Bing, Yahoo, Yandex, and so on)can cope with the challenges and the massiveness of the entire web and keep their index fresh. They need to use many different sources (e.g. being a member of ICANN[2] and getting the list of newly registered domains, and so on) to discover the new links and keep recrawling their existing URLs just tomaintain freshness. Because of these very reasons, developing focused web crawlers are much more feasible and commonly practiced.

Focused web crawlers are specialized versions of general web crawlers that crawl only certain topics or certain websites [1]. Even though they are much smaller scale than general web crawlers, still many challenges and tough tasks

[1] URL is an acronym for Uniform Resource Locator and is a reference (an address) to a resource on the Internet, http://docs.oracle.com/javase/tutorial/networking/urls/definition.html
[2] The Internet Corporation for Assigned Names and Numbers, https://www.icann.org

await the developers who are going to build focused web crawlers [2]. In this paper,it is aimed to share experiences that are gainedfrom the development of a full-scale, object oriented, robust, and fast focused web crawler that uses the latest technologies with multithreaded implementation and has a very high success rate with hand-crafted rules. The development is made in Visual C#[3]programming language based on the WPF[4] programming model by using .NET 4.5[5] framework and x64 platform. MS-SQL Server 2014[6]is used for database management. For programming Visual Studio 2013[7] IDE is used.

The challenges for which possible solutions are proposed in this study are listed as below:

- Providing communication between user interface (i.e., the main thread) and background worker threads,
- Maintenance of responsive user interfaces while running thousands of threads,
- Collaboration and resource sharing between different threads,
- Unexpected error handling,
- Maintenance of URL repository,
- Robust web page fetching,
- Keeping logs for errors,
- Providing optimized database access for performance improvement,
- Being able to process erroneous HTML source,
- Handling different HTML source page structures for data extraction from different websites,
- Compressing HTML source files to save disk space,
- Saving bandwidthby HTTP compression,
- Management of the number of threads that run simultaneously.

Since focused webcrawlers are a specialized version of general web crawlers, many of the above challenges are also the concern of general web crawlers as well. Therefore, we believe that this study can also help general web crawler developers too.

The rest of the paper is organized as follows: in the next section, the related work for the web crawler development is discussed. In the third section, the architecture of the proposed focused web crawler, and the possible solutions for the problems encountered during the implementation of our crawler are presented. The fourth section discusses the experimental results which show the effectiveness of the proposed methods, and finally, the last section concludes the study.

[3] Visual C# Resources, https://msdn.microsoft.com/en-us/vstudio/hh341490
[4] Windows Presentation Foundation, https://msdn.microsoft.com/en-us/library/ms754130
[5] .NET Framework and .NET SDK Downloads, https://msdn.microsoft.com/en-us/vstudio/aa496123.aspx
[6] SQL Server 2014 | Microsoft, http://www.microsoft.com/en-us/server-cloud/products/sql-server/default.aspx
[7] Visual Studio - Microsoft Developer Tools, https://www.visualstudio.com

RELATED WORK

In the literature, the majority of the related works are about how to design a general web crawler [3-6] or how to decide more relevant pages for a focused web crawler to reduce resource requirements such as space, bandwidth, computation power, etc. in the crawling task [7-17]. To our knowledge, there is no study that explains the challenges and the possible solutions for developing a (focused) web crawler from developers' perspective by providing implementation detail and being up-to-date. Additionally, the web has become so enormous and spam my that most of the previously proposed approaches may fail today. Google[8]displays statistical data about spam page son the web and the domains that are penalized by Google manually. These statistics clearly show an exponential increase of spam. Nowadays spammers are able to generate more sense making spun content[9] by using machine learning algorithms. They are also able to build highly related link structures. Thus, depending solely upon link relations, semantic relevancy or content analysis can lead to a failure. And for other than the multi-million budget having companies, it is not very possible to scan the entire web to determine the authoritativeness of websites or obtain enormous golden data to weigh them accurately. For the reasons outlined previously, if there is a need for a web crawler that requires high success rate, we propose that developing a task oriented, new and hand-crafted focused crawler may be the best solution.

Nisha *et al.* [7] have proposed an approach to improve the selection of seed URLs by using user interest ontology. Uzun*et al.*[8] have used decision tree learning[10]to automatically determine useful parts of the web pages. Different blocks of the web pages are identified by using HTML tags in order to improve the efficiency of web crawling and data extraction. Dahiwale*et al.* [9] have proposed a crawler that utilizes link and content-based evaluation to improve the precision of focused crawling. Yohanes*et al.* [10] have proposed a genetic algorithm by using methodology that employs adaptive and heuristic methods to improve there levancy of the crawled pages. Papavassiliou *et al.* [11] have developed a system for harvesting topic specific data from the web for both monolingual and bilingual pages. Li*et al.* [12] have proposed a framework that uses divide and conquer strategy by using double step page classifier, link evaluation, when to stop crawling strategy, etc. to enhance the focused crawling success. Kumar *et al.* [13]have developed an algorithm that is based on term frequency–inverse document frequency (TF*IDF) which improves the precision of topic targeted crawling in consecutive crawling phases. Liu *et al.* [14] have proposed a framework that calculates the distance of the newly discovered URLs from the topic by using Maximum Entropy Markov Model and Linearchain Conditional Random Field probabilistic models by exploiting multiple features such as anchor text, etc. Bedi *et al.* [15] have proposed the Dynamic Semantic Relevance system, which uses terms and links to improve crawling efficiency and precision of focused web crawlers. Maimunah *et al.* [16] have proposed a system that uses relations between parent, sibling, target, child and spouse (which links to child) documents to improve precision and recall of focused web crawling. Liu *et al.* [17] have developed an approach that uses Semantic Similarity Vector Space Model by utilizing TF*IDF and semantic similarities to calculate relevancy of the uncrawled URLs. Such related works list some efficient crawling strategies and show that system scan be extended, however, our aim in this study is different. In this research, our aim is to start crawling from a static root web site, and crawl all the pages except particular ones (e.g., blocked by robots protocol[11] or ignored by manually crafted rules, etc.) from the root domain with high success rate. To achieve this goal, we propose practical implementation based

[8] Fighting Spam – Inside Search – Google, https://www.google.com/insidesearch/howsearchworks/fighting-spam.html
[9]Article spinning – Wikipedia, https://en.wikipedia.org/wiki/Article_spinning
[10]Decision tree learning – Wikipedia, https://en.wikipedia.org/wiki/Decision_tree_learning
[11]A Standard for Robot Exclusion - Robotstxt, http://www.robotstxt.org/orig.html

solution methods which involve fine tuning of some .NET framework based parameters, and we show the performance of our proposed solutions experimentally.

Boldi*et al.*[3] have proposed a scalable web crawler called as Ubi Crawler which is similar to our study. In [3], every task is tried to be decentralized as much as possible. The system is implemented by using Java programming language, however, details about how to improve programming language related performance issues were not discussed. Also, UbiCrawler does not guarantee 100%prevention from duplicate crawling, and the system may have problems to achieve continuous crawling when it is restarted or at total application crash. Also, how to distribute crawling hosts to the agents were not explained in detail. Due to their system design, even distribution of the workload among agents is difficult to achieve. The number of the URLs among the agents is assumed to be evenly distributed during the big crawls, however, our experiments indicate that this assumption may not be valid every time. Also, due to the decentralization of the agents, they cannot communicate with the each other to access the disk, and this may cause a restriction to be put on the number of active agents to prevent hard drive from speed throttling. In EcCrawler, on the other hand, every agent is centrally controlled, thereforedistribution of the workload among agents is highly achieved and the crashes or the application restarts do not cause any problem for continuous crawling. EcCrawler's design, capability, features, and programming language that is used are highly different from UbiCrawler.

Gomes and Silva [4] have proposed a generalized web crawler and discussed some of the challenges they have encountered. Java programming language is used for implementing the crawler but no programming language related problems, their solutions, and optimizations are mentioned. Their main focus is more about explaining problems that are encountered while crawling the web and the structure of their crawler.

Heydon and Najork [5] have proposed a highly-scalable general web crawler system and discussed the fundamental problems of a general web crawler development. Their proposed system has been developed in Java programming language. However, implementation details to solve problems encountered during the crawler design were not provided.Shkapenyuk and Suel [6] have presented a system architecture of a distributed web crawler that runs on multiple connected servers. The primary concern of the study is I/O and network efficiency, and solutions for performance bottlenecks and how to obtain high performance when bulk crawling are discussed. C++ and Python programming languages are used to develop the crawler. However, implementation details and solutions for coding related problems and tunings are not discussed. Edward s*et al.*[18] have developed a fully distributed general web crawler model which mainly focuses on an efficient incremental crawling strategy. Olston and Najork [19] have made an extensive review of the web crawling literature. They have covered the majority of the web crawling topics such as crawler architecture, crawl ordering, avoiding undesirable content, and so on.

Our study is different from the previous studies that have been made in this area such that we provide source codes to solve challenging problems that may be encountered during crawler implementation. Additionally, we provide extensive empirical performance analysis for the proposed solutions.

THE PROPOSED CRAWLER SYSTEM

This section explains the general architecture of the EcCrawler as well as the challenges encountered during the development of the EcCrawler and possible solutions to these problems.

The Architecture of EcCrawler

EcCrawler starts by initializing public static variables, functions, and classes. In the second stage, main tasks that are URL Handler, Crawler, Pages Processor, and Global Statistics Handler are started independently from the main thread and thus obtain better overall application performance and the user interface (UI) responsiveness[12]. The UI thread and the main thread are the same thread in this study and this is the default setup of WPF(Windows Presentation Foundation)applications in .NET framework. Figure 1shows the workflow of EcCrawler. In the following subsections, implementation details of the main tasks such as initialization, main thread, URL handler, URL crawler, database communication, pages processor, etc. are presented.

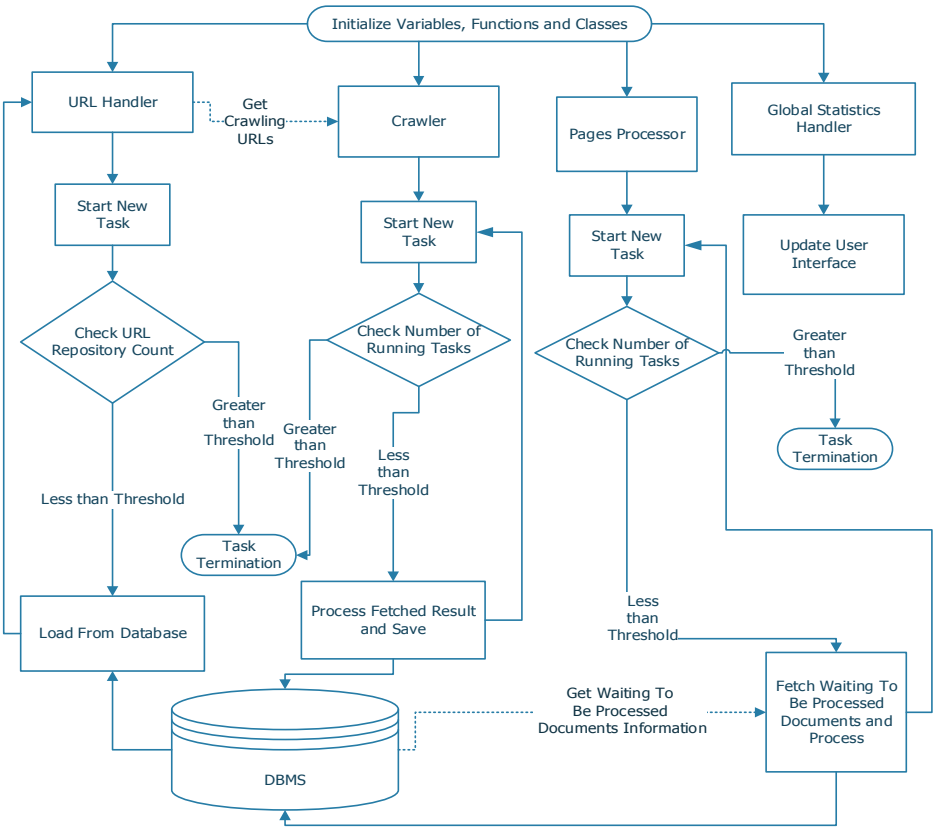

Figure 1: Flowchart of EcCrawer

Initialization

Initialization process sets up the global variables; manages continuously running Tasks[13]and functions; performs thread pool initialization; controls global unhandled exception handler; and runs close handlers.

URL Handler, Crawler, Pages Processor, and Global Statistics handler are Tasks that run continuously. These Tasks continuously start new Tasks in a pre determined time interval, and they continue to operate until application termination.

[12]Chapter 6 — Using Multiple Threads, https://msdn.microsoft.com/en-us/library/ff649143.aspx
[13]Task Class. Represents an asynchronous operation. Microsoft Developer Network, https://msdn.microsoft.com/en-us/library/system.threading.tasks.task

Thread Pool Initialization

.NET framework determines the number of threads that is spawned automatically according to the application demand. However, this may yieldpoor results if the software is running on apowerful hard ware when the hardware demand is high. To be able to start a pre-determined number of threads and utilize the system resources better, we propose to set the thread pool count manually before spawning the threads as shown in Code Snippet *1*.

```
ThreadPool.SetMaxThreads(100000, 100000);
ThreadPool.SetMinThreads(100000, 100000);
```

Code Snippet 1: How to initialize the Thread Pool

SetMax Threads sets the maximum number of threads for the thread pool, and if there is more demand than the pre-determined maximum count, the demands are queued until the thread pool becomes available. SetMin Threads sets the minimum number of threads for the thread pool before switching an algorithm to manage the thread pool. In both functions that are shown in Code Snippet *1*,the first parameter is the number of worker threads in the thread pool, and the second parameter is the number of asynchronous I/O threads in the thread pool. Worker threads are used for the active works done in the thread pool while I/O threads are more likely to wait for an external operation to complete such as a data receive from the network. We propose to set both SetMax Threads and SetMin Threads to anequal high number to obtain maximum available resources for the thread management system.

Exception Handling

Setting up proper exception handling enables developers to discover not properly handled errors during the program run. While our software is running in a real case scenario, we propose always to setup global exception handling to prevent from accidental application crashes. Moreover, errors are need to be properly logged for future analyses. Close handlers are also necessary to ensure that no data is lost when the application is terminated by running a command or clicking the close button at the right top of the UI. Moreover, close handlers can be set to run when an unexpected application termination happens for preventing any possible data loss. Code Snippet *2*shows how to setup global unhandled exception handling and application of close handlers.

```
AppDomain currentDomain = AppDomain.CurrentDomain; //set the event domain

Application.Current.DispatcherUnhandledException += new
//setup crashing event handler for sub threads
      DispatcherUnhandledExceptionEventHandler(
            CloseCrashHandlers.AppDispatcherUnhandledException);

currentDomain.UnhandledException += new
//setup crashing event handler for main thread
      UnhandledExceptionEventHandler(CloseCrashHandlers.CrashCloseHandler);


Closing += new CancelEventHandler(CloseCrashHandlers.CloseHander);
//setup closing event handler
```

Code Snippet *2*: How to Setup Global Unhandled Exception Handler and Close Handlers

UI Independent Tasks

Executing every task independently from the main thread can ensure application responsiveness, because none of the tasks would block the main thread. The best way to achieve this is to start a new independent Task for every process that application is going to do. We propose that only UI updates should run in the main thread by using polling methodology. Thus, Task. Factory method as shown in Code Snippet *3*to spawn new threads can be used. This method is one of the most optimal ways of doing multi-threading in C# WPF applications when using .NET 4.5 because Tasks are more light weighted when they are compared to Threads[14]. Additionally, using Task Scheduler. Defaultparameter will ensure that all of the spawned Tasks will be independent of themain thread.

```
Task.Factory.StartNew(() =>
{
   StartTasks();
}, CancellationToken.None, TaskCreationOptions.LongRunning,
     TaskScheduler.Default);
```

Code Snippet *3*: How to Start New Threads Independently from the Main Thread

Timer-Based Polling Methodology

We propose that polling methodology based system design provides both performance and robustness. Polling based system can be achieved by starting all of the tasks as different threads from the main thread during the initialization phase, and doing timer-based function calls in these child threads as shown in Code Snippet *4*. Timer[15] method also starts another Task and even if this spawned Task is terminated unexpectedly, the method continues to spawn new Tasks. Therefore, unexpected errors do not cause the application gets terminated, and unexpectedly terminated Tasks are properly eliminated.

```
private static Timer _timer;
private static int howManySeconds = 1;

public static void func_StartCrawlingWaitingUrls()
{
   _timer = new Timer(wrapper_func_CheckWaitingUrls, null,
                      PublicSettings.irTimers_Delayed_Start_MiliSeconds,
                      howManySeconds * 1000);
}
```

Code Snippet 4: How to Start Timer-Based Polling Functions

Main Thread

Main thread is important for developers or clients to manage the crawling process. Displaying various statistics in the UI helps us greatly during the development of the crawler. Some developers may find that it is not essential for a web crawler to have a fancy UI. However, our proposed system can be applied to all kinds of software that require highly responsive UI. Running all of the tasks separate from the main thread can ensureUI responsiveness. However, updating the screen still requires accessing the UI thread. How to start polling threads from the main thread and refresh the interface properly without blocking the UI thread for a period of time are shown in Code Snippet 5. The _timer object ensures UI is updated with an interval and the Dispatcher event in this method ensures UI is not blocked. This methodology can minimize the risk of UI blockage.

[14]Task Parallelism (Task Parallel Library), https://msdn.microsoft.com/en-us/library/dd537609(v=vs.110).aspx
[15]Timer Class – MSDN, https://msdn.microsoft.com/en-us/library/system.timers.timer(v=vs.110).aspx

In EcCrawler, a global static class is designed to store the events (e.g. URLs of the latest crawled pages, number of page-processing tasks that are running, how many crawling errors that happened, and so on). Polling threads constantly check this global static class and update the user interface without blocking the UI thread. However, even if this methodology is followed, UI freeze can still happen under certain situations. We noticed that this problem happens due to the Garbage Collector (GC) ofthe .NET framework. When the GC runs in the default mode which is workstation, if there are too many objects being constructed and destructed in the running software, it may pause all of the threads, including the main thread for along time, and thus cause UI freeze, and software performance degrades. During the development of the EcCrawler, figuring out and solving this problem has taken some time. A possible solutiontoperformance degradation issue is to set GC mode to server which does not pause all of the threads and support multi-threaded garbage collection. This setting is especially useful when the software is running on a system that has multi-core CPU. Code Snippet 6shows how to set GC mode. After this mode change, according to our observations, even if thousands of Task srun at the same time, UI responsiveness is not affected as long as there are sufficient system resources to update the UI. Also, changing the GC mode improves overall performance significantly. Experiments about this modification are presented in section 0.

```
private static System.Threading.Timer _timer;

_timer = new Timer(updateGlobalStatistics, null,
                        PublicSettings.irTimers_Delayed_Start_MiliSeconds,
                        PublicSettings.ir_RefreshUI_MS);//Start polling thread

object updateLock = new object();

 private void updateGlobalStatistics(object sender)
    {
        if (Monitor.TryEnter(updateLock))
        {
            try
            {
                Application.Current.Dispatcher.Invoke(new Action(() =>
                {
                  //Get data from global static class and update the interface
                }));
            }
            finally
            {
                UiRefreshed();
                Monitor.Exit(updateLock);
            }
        }
    }
```

Code Snippet 5: How to Refresh the User Interface Without Causing UI Freeze

```
<configuration>
  <runtime>
    <gcAllowVeryLargeObjects enabled="true"/>
      <!–If the application uses over 2GB objects-->
    <gcServer enabled="true"/> <!--Set garbage collector mode-->
  </runtime>
</configuration>
```

Code Snippet 6: How to Set Garbage Collector Mode

URL Handler

One of the main components of a web crawler is a URL handling system [20]. The main job of this system is to queue which URLs will be crawled next. Using MS-SQL server makes easy to build the URL handler system and itprovides some benefits which are explained in the below paragraphs.

Several key aspects are observed when designing a URL handler system:

- Both the original URL sand their normalized versions that can be obtained by applying some methods like SHA256, should be stored since some websites work with case-sensitive URLs.
- Newly discovered URLs should be saved to the hard disk as batches in order to obtain better performance in terms of disk access time. Both versions of the URLs should be kept in a properly designed data structure such as hashsets or dictionaries until they are written to disk. Not-properly designed structures can reduce performance of the crawler.
- Since different threads concurrently access to the data structure that store URLs, using private read only objects to lock the data structure can ensure thread safety and data consistency. An example of such case can be seen in Appendix Code Snippet 7.
- URLs that are failed to be crawled for a particular time should be disabled to prevent crawler trapping at the same URLs. Disabling the failed URLs must be done both in the database and in the URL repository of the software. In EcCrawler, we have set a variable which is called as retry count to 3 to solve this problem. In our crawler, after three consecutive failures, the URL is disabled for 24 hours. We did not conduct extensive experiments to find optimal parameters for these settings. Therefore, optimal value for retry count can be determined as future work.
- Another vital point is to provide load balance between the different E-commerce websites. Size and capacity of each site are distinct from each other, so the number of the crawling tasks for each website should be different and this value should be determined separately for each site to ensure both politeness and maximum performance of the crawling task. One of the major mistakes that were made during the early development stage of EcCrawler was the nonexistence of any load balancing system between the E-commerce websites. The URL handler was fetching the next batch of the crawling URLs from the database by the earliest discovery date. Simply doing breadth-first search [21] which is one of the very first proposed approaches [22] in the literature, caused that a few huge sized E-commerce sites get all of the available crawling agents and thus decrease the overall application performance and violate the politeness rule of the crawling. One methodology proposed to solve this problem is to use the completion time of the previous crawling session [21]. By considering the crawling speed of each website individually, the number of agents for each website and the pause period for each consequent crawling can be individually determined. This approach requires determining threshold values for these parameters. This method favors very fast websites, and this may result in several enormous and very fast websites to get the majority of the available agents. One of the possible ways of solving this problem is to assign a unique static number for the maximum number of fetching agents at a time for each one of the E-commerce sites. When retrieving next crawling URLs from the database, a specified number of links for each E-commerce site is retrieved according to the pre-determined thresholds. The URL handler must also keep track of the list of URLs that are crawled currently to maximize the number ofcrawling agents for each website. When the crawler asks the URL handler to send the next batch of the queued links, a pre-determined number of URLs need to be provided for each website

to accomplish load balancing. We defined different number of links to be crawled for each website according to our empirical analysis of early crawling results. Additionally, public search engines can be used to determine these values. All the main search engines (e.g. Google, Yahoo, Bing, Yandex, and so on) support "*site:mysite.com*" queries which provide the number of estimated results each of which can be considered as a different page. Although these results are not very accurate and reliable (e.g. they don't display exact statistics, or some sites could have duplicate pages because of incorrect URL structures, and so on), they can be used to roughly estimate the size of the crawled websites and determine the number of crawling agents proportionally for each site (e.g., if there are 100 available agents at a time, distribute them according to these size values).

```
private static readonly object _lockObject = new object();
//object to use for locking

public static void funcitonName(object inPut)
{
    lock (_lockObject)
      //lock the object so non-thread safe objects are now thread-safe
    {
            //execute necessary procedures on non-thread safe objects such
            as dictionaries
    }
}
```

Code Snippet *7*: An Example for Usage of Data Structures and Locking to Ensure Thread Safety and Data Consistency

- After load balancing strategy is decided, the question arises which links should be crawled next. Several different techniques have been proposed in the literature to determine which links should be the next. These methods can be summarized as follows:(a) First discovered is first crawled [21];(b) Giving priority by using the count of incoming hyperlinks [23] to crawl more relevant pages first; This technique is now obsolete due to the massive amount of link spamming; (c) Giving priority by some page scoring such as Page Rank [23-28]. However, most of the advanced page ranking algorithms may perform poorly against current web pages because of the newly developed advanced search ranking manipulation techniques[16] (e.g., link schemes, sneaky redirects, affiliate programs, paid links, and so on). Google uses over 200 different factors[17] to determine the Page Rank of web pages today to fight against these techniques and deliver high-quality search results. Since in our work, the websites that are going to be crawled are pre-determined, and there is a finite number of pages; we did not use any page ranking algorithm to give priority to pages which are crawled first. Instead of some page ranking mechanism, we have employed manually constructed filtering rules to prevent crawling unnecessary pages. For example for the E-commerce site VatanBilgisayar, if the newly discovered URLs contain "*/webapp/*"or "*search Query=*" strings, they are discarded. Because, a URL, which contain these two strings, cannot be a category page or a product page. For our task, we only need to crawl and process the category pages which list the product pages, and the product pages themselves. Currently, a domain expert analyses the discovered URLs and decides which strings are used for filtering the web pages. Whenever structures of the product and category pages are changed, we need to update our rules which can determine the product and category pages. Therefore, automatic

[16]Quality guidelines – Google, https://support.google.com/webmasters/topic/6001971
[17] Google Inside Search, Algorithms, https://www.google.com/insidesearch/howsearchworks/algorithms.html

determination of product pages and category pages is needed and this can be done by employing a classifier. As our main aim in this study is to show .NET framework implementation based solutions for the problems encountered, we plan to include a classifier to our crawler as a future work.

URL Crawler

URL crawler (i.e., page fetching) is another core part of a web crawler [19]. Tasks of a URL crawler can be summarized as follows: crawling the given link, extracting the hyperlinks from the crawled page, and returning both the extracted links and the source code of the crawled page. In the EcCrawler, only product pages are processed so the crawler checks whether the page is a product page or not. If it is not a product page, the source code is not saved. Also, all of the extracted URLs are processed by applying techniques like “replace words”, “ignore words”, “trimming”, “not allowed types” (e.g., PNG, JPG, PDF, and so on), “disallowed by robots protocol”, and so on. Some of these techniques are manually defined after inspection of the targeted E-commerce websites. Eliminating unnecessary pages (e.g., duplicate pages, irrelevant pages, etc.) from processing makes a huge impact on the overall performance of crawling. When the crawler tasks complete their operation, they return the discovered URLs and the source of the crawled page to the URL handler so that they can be saved in the database system; and the discovered new URLs can be added to the crawling queue.

The performance of the web crawlers is highly depended on the used settings. However, in most cases these settings are not mentioned in the related literature (e.g.,[6, 18, 19]). To possibly obtain better performance, we adviseusing Http Web Request class in .NET framework because this class allows tuning of many different parameters. From these parameters Headers[18], Encoding[19] of the Stream Reader, KeepAlive and the Response Uri are necessary. Headers are used to enable HTTP compression[20] when fetching pages from the server and thus significantly reduce the network usage. HTTP compression is achieved by adding gzip[21] and deflate[22] parameters to the headers by using “Accept-Encoding” key as shown in Code Snippet *8*.Encoding of the stream is essential for parsing of the fetched source code properly.It is provided to the Stream Reader[23]function, and it determines the character encoding of the fetched source code[24]. Keep alive[25] feature may significantly affect the overall network performance according to the crawling scenario. When the Keep alive feature is enabled, it uses a persistent HTTP connection[26] to the crawled server for multiple HTTP requests instead of establishing a new connection session when each time a page is fetched. The only disadvantage of Keep alive is, it causes extra memory usage (especially for the web server) therefore, when there is multiple access to the server, the Keep alive feature should be enabled.

Also, there will be many pages that redirect to another page. Therefore,it is important for acrawler to handle the redirected URLs for achieving proper crawling and preventing duplicate crawling.HttpWebRequestclass by default allows automatic redirections. So if any redirection happens during the HTTP request to the server, it is handled by Http Web

[18] Headers, https://msdn.microsoft.com/en-us/library/system.net.httpwebrequest.headers(v=vs.110).aspx
[19]Encoding Class – MSDN, https://msdn.microsoft.com/en-us/library/system.text.encoding(v=vs.110).aspx
[20]HTTP compression – Wikipedia, https://en.wikipedia.org/wiki/HTTP_compression
[21] Gzip – Wikipedia, https://en.wikipedia.org/wiki/Gzip
[22]DEFLATE – Wikipedia, https://en.wikipedia.org/wiki/DEFLATE
[23] StreamReader – MSDN, https://msdn.microsoft.com/en-us/library/system.io.streamreader(v=vs.110).aspx
[24]Character encoding – Wikipedia, https://en.wikipedia.org/wiki/Character_encoding
[25]KeepAlive – MSDN, https://msdn.microsoft.com/library/system.net.httpwebrequest.keepalive(v=vs.110).aspx
[26]HTTP persistent connection – Wikipedia, https://en.wikipedia.org/wiki/HTTP_persistent_connection

Request class automatically, and the final crawled URL can be obtained from Response Uri .Absolute Uri parameter of the Web Response class.

All settings can be done as shown in Code Snippet 8. Also, EcCrawler shows respect to the canonical URLs[27] system of Google. Canonical URLs show absolute links of pages to prevent duplicate pages[28]. If the source code has canonical URL parameter and the URL of the fetched page does not match with the canonical URL, then the URL handler only returns the canonical URL of the fetched page. If this returned canonical URL has been seen before by the crawler, the fetched URL is marked as "*duplicate by canonical*". For example, assume that the discovered URL is "http://www.buroteknik.com/Cross-883-3-Atx-Bazalt-Siyah-Versatil_151274.html" while the canonical URL in the page source is "http://www.buroteknik.com/en-ucuz-cross-883-3-atx-bazalt-sc4b0yah-versatc4b0l-fiyati-ozellikleri_151274.html". As can be seen, the URLs are different however they are the same pages.

One other critical problem is infinite-URLs generation. Some websites use bad link structure which results in generating infinite amount of URLs. Infinite-URLs generation happens by adding more directory level to the URLs. The following URL can be given as an example of such case: http://www.mysite.com/toys/computers/toys/computers. To possibly overcome this problem, a maximum directory depth for each website can be set manually after a domain expert inspects URL structure of each website. However, this may cause false rejects if the URL itself contains "/" character instead of for directory level. On the other hands, the gainsthat have been obtained by setting a max directory depth value are much greater than these very rare false rejects when developing EcCrawler. As many E-commerce websites generate an infinite number of links they cause crawling of too many duplicate pages.

```
HttpWebRequest request = (HttpWebRequest)WebRequest.Create(srUrl);
//init request
request.KeepAlive = true; //keep connection alive
request.Accept =
      "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8";
      //set request headers
WebHeaderCollection myWebHeaderCollection = request.Headers;
myWebHeaderCollection.Add("Accept-Language", "en-gb,en;q=0.5");
myWebHeaderCollection.Add("Accept-Encoding", "gzip, deflate");
//set accepting http compression

request.AutomaticDecompression = DecompressionMethods.Deflate |
      DecompressionMethods.GZip; //enable automatic decompression
using (WebResponse response = request.GetResponse()) //start response
{
    using (Stream strumien = response.GetResponseStream())
    {
        Encoding myEncoding; string srContentType = "";
        if (response.ContentType != null)
        {
            srContentType = response.ContentType;
            if (srContentType.Contains(";"))
            {
                srContentType = srContentType.Split(';')[1];
            }
            srContentType = srContentType.Replace("charset=", "");
            srContentType = //try to auto get encoding type
                PublicStaticFunctions.func_Process_Html_Input(srContentType);
        }
        try { myEncoding = Encoding.GetEncoding(srContentType); }
        catch { myEncoding = irCustomEncoding == 0 ? Encoding.UTF8 :
                                    Encoding.GetEncoding(irCustomEncoding); }
                                    //set recognized encoding
        using (StreamReader sr = new StreamReader(strumien, myEncoding))
        {
            srBody = sr.ReadToEnd(); //read the response
            srFinalUrl =
                  PublicStaticFunctions.Return_Absolute_Url(
                        response.ResponseUri.AbsoluteUri.ToString(),
                        response.ResponseUri.AbsoluteUri.ToString());
        }
    }
}
```

Code Snippet 8: A Shortened Version of EcCrawler's Fetching Function

In today's web,there is a vast amount of web pages to crawl and the number of web pages also increases

[27] Use canonical URLs, https://support.google.com/webmasters/answer/139066
[28]Duplicate content – Google, https://support.google.com/webmasters/answer/66359

exponentially.To crawl all these pages,more concurrent connections are neededto be opened to the same host by spawning multiple crawling agents. However, when the default settings are used, .NET framework only allows having two connections concurrently to the same server. Two connections concurrently are the default limit defined by HTTP/1.1[29] protocol in 1997. At that time, when the capabilities of web server hardware and software are taken into account, this limit can be considered as reasonable. However, nowadays, the power of web server hardware and software has significantly increased. Additionally, sizes of websites have been exponentially increased. As a consequence of the sestated reasons, to obtain better crawling performance we need to use multiple crawling agents to the same host, therefore this limit needs to be increased. Moreover, many of the modern web browsers already allow more than two persistent connections for each host (e.g. Mozilla Firefox allows six current connections by default as it can be seen in Figure 2). Concurrent connection limit per host can be increased by setting up Default Connection Limit parameter as shown in Code Snippet 9 for .NET framework when using C#. This parameter must be set before the web request is made hence; EcCrawler sets this parameter at the start of the application.

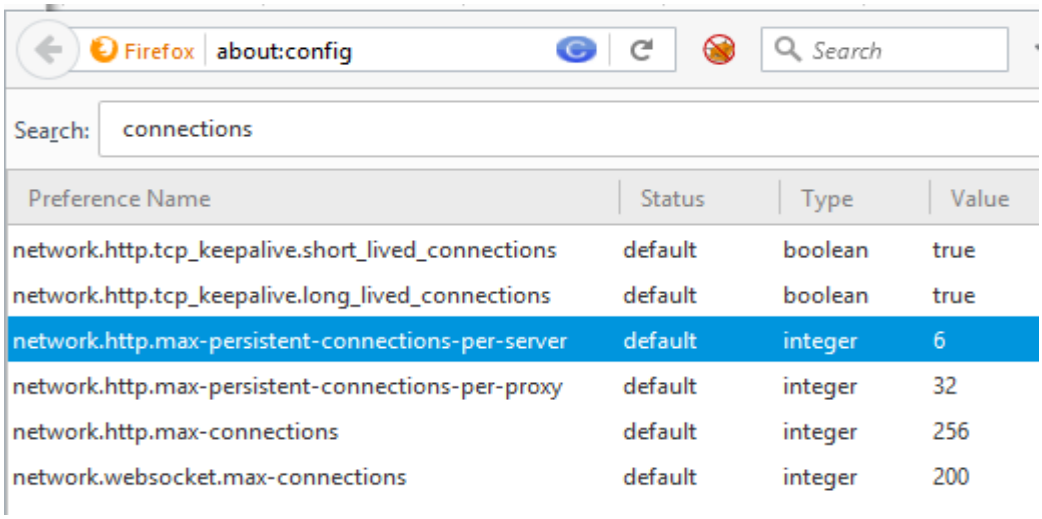

Figure 2: Default Max Persistent Connections per Server Settings for Mozilla Firefox Version 44.0.2

```
ServicePointManager.UseNagleAlgorithm = true;
//https://msdn.microsoft.com/en-
us/library/system.net.servicepointmanager.usenaglealgorithm
//Used to reduce network traffic by buffering small packets of data
// and transmitting them as a single packet

ServicePointManager.Expect100Continue = false;
//Setting this false will more likely to improve performance because
// some servers does not support it

ServicePointManager.CheckCertificateRevocationList = false;
//If certificate revocation is not important for developed crawler
// setting this false more likely to improve performance

ServicePointManager.DefaultConnectionLimit = 1000;
//This sets to maximum number of concurrent connections to same host
```

Code Snippet 9: How to Increase the Maximum Number of Concurrent Connections to the Same Host and Improve the Fetching Performance of the Crawler by Tuning the Network Configuration

For further analysis, we have examined statistics of the top 10 Turkish E-commerce websites in Turkey by using

[29]Hypertext Transfer Protocol -- HTTP/1.1, https://www.w3.org/Protocols/rfc2616/rfc2616.html

Alexa[30] and Google services to see whether maximum two simultaneous persistent connection for each host is suitable or not. Our aim in this analysis is to test whether these targeted E-commerce websites' daily fresh content could be obtained while obeying concurrent connection count limitation of the HTTP/1.1 protocol. So estimated index size for each website is collected by querying "*site:mysite.com*" from Google. Then, average loading speed of each of these websites are gathered from Alex a which calculates page loading speed by computing how long does it take the user's browser to load DOM (i.e., the structure of the page which does not include images or CSS styles) of the page. Loading speed of browser can be slower than page source fetching of the crawler however, since Alex a takes only DOM loading into consideration and has enormous statistical data about these websites, we believe that this measure can give an overall idea. According to the Cho and Garcia-Molina [29], more than 40% of the pages in the *.com*[31]domain changes every day. So to keep the freshness of the E-commerce websites, we have calculated minimum number of daily page crawling requirement for each site and this is computed by multiplying the estimated page count of the site by 40%.Moreover,we have calculated how many pages can be crawled for each E-commerce website as follows:

$$HPCBC(e) = \frac{H \times M \times S \times CC}{ALS(e)},$$

where $HPCBC(e)$ is the number of pages that can be crawled for the selected E-commerce website in a day, H is hours in a day (i.e., 24), M is minutes in a hour (i.e., 60), S is seconds in a minute (i.e., 60), CC is the concurrent connection count (i.e., 2), and $ALS(e)$ is the average load speed in seconds of the selected E-commerce website which is directly taken from the Alexa's statistics. The results are displayed in the Table 1where estimated index size is provided by Google and the coverage is $\frac{PCD(e)}{HPCBC(e)}$ where $PCD(e)$ is the minimum #of page to be crawled per day to re-crawl the updated content each day.

Table 1: Statistical Analysis of Concurrent Connection Limit on E-Commerce Websites

E-commerce Site	Estimated Index Size	Average Load Speed ($ALS(e)$)	#of Pages to be Crawled per Day ($PCD(e)$)	How Many Pages can be Crawled per Day ($HPCBC(e)$)	Coverage
Gittigidiyor	788,000	1.682	315,200	102,734	%32.59
N11	781,000	1.079	312,400	160,148	%51.26
Hepsiburada	778,000	1.605	311,200	107,663	%34.59
Vatanbilgisayar	3,230,000	1.758	1,292,000	98,293	%7.60
Trendyol	570,000	2.053	228,000	84,169	%36.91
Teknosa	1,070,000	3.153	428,000	54,804	%12.80
Markafoni	473,000	1.999	189,200	86,443	%45.68
Sanalpazar	552,000	0.986	220,800	175,253	%79.73
Kitapyurdu	627,000	1.088	250,800	158,823	%63.32
Tozlu	1,110,000	1.141	440,000	151,446	%34.41

When the results in Table 1areanalyzed, it can be concluded that two concurrent connections at a time are not enough to keep up daily content freshness in today's web. However, these are rough estimations, and detailed experiments may be required to obtain more accurate results because of several reasons:

[30] An Amazon company that provides public statistics about websites, http://www.alexa.com/
[31]COM = Commercial, any commercial related domains meeting the second level requirements, http://tools.ietf.org/html/rfc920#page-2

- Google statistics are not entirely accurate thus, there may be more pages;
- Statistics that are provided by Alexaare estimations of various parameters, and they may not be very accurate;
- Page crawling speed is highly depended on the network connection speed and the time of the day when the web server is accessed;
- It is not possible to crawl only updated pages without re-crawling the other pages as well;
- 40% update estimation is based on the experiments conducted in 1999 and today it may be very different percent;
- In some cases not daily but hourly, even minutely freshness may be required, which means crawling for more pages may be necessary;
- There are some websites that contains much more content and thus requires much more frequent crawling sessions (e.g. Amazon[32] has 161,000,000 estimated index size, and so on).

Even though these statistics are not perfect, it is clear that only twoconcurrent connections is not sufficient currently, and it should be increased according to the tasks that are running, and the system resources used by the tasks. Therefore, we have set this limit to a large number which is equal to 1000. On the other hands, to prevent any bottleneck on the E-commerce websites, wemake sure that we have never exceed 50 concurrent connections, as our maximum crawling agent count per website has been set to 50.We have set this number based on our empirical analysis of the crawling speed by testing several different values.

Database Communication

In EcCrawler, all of the database queries are executed by using a single database handler class. Using single class makes database handling system easier to manage. Several different functions are written to handle specific cases according to EcCrawler requirements. One of the database management functions designed for EcCrawler is shown in Code Snippet 10.In the displayed function there are several key parameters:

- when integrated security=SSPI is used, no user ID or password is required since the Windows's current credentials is used for authentication;
- Max Pool Size=20000 sets the connection pool size of the SQL-Server for that client and if the application opens a lot of concurrent connections to the SQL-Server this parameter should be set to a high number;
- enabling Pooling[33] improves overall performance by preventing from opening a physical channel each time a new connection is opened;
- Connection Timeout determines the timeout of connection opened before the connection is automatically closed and this should be set to an appropriate value according to the scenario (e.g. if exhausting queries are being executed, this timeout should be set to a high number in order to prevent connection problems);
- CommandTimeout is a different parameter than the Connection Timeout thus it should also be set to anappropriate value.

[32]http://www.amazon.com/
[33]SQL Server Connection Pooling, https://msdn.microsoft.com/en-us/library/8xx3tyca(v=vs.100).aspx

Performance improvement of this function comes from using a single database connection for executing multiple updates or insert operations. The function gets a list of parameters and objects of multiple queries and then save them in the database by using Sql Command. This kind of batch inserts and updates may improve overall performance by preventing multiple server connection authentications. To minimize the number of server connection authentication, as ingle static connection is shared a mong all of the threads to prevent authentication overhead. However, SQL connection is not thread safe and need to be locked each time it is accessed. Lock methodology can decrease overall performance when there are too many threads and, therefore, it is better to use a new connection for each thread.

```
public static string srConnectionString = "server=localhost;database=dbBame;
integrated security=SSPI; Max Pool Size=20000; Pooling=True;Connection
Timeout=3000;";//define connection string

public static bool batch_execute_CMD_update_delete(string srCommandText,
List<string> lst_Command_Variable_Names,
List<List<object>> lst_Command_Parameters, CommandType cdType)
//a generic function that can be called from any thread
        {
            bool blErrorHappened = false;

            try
            {
                using (SqlConnection connection = //init SQL connection
                   new SqlConnection(DbConnection.srConnectionString))
                {
                    connection.Open(); //open SQL connection

                    for (int i = 0; i < lst_Command_Parameters.Count; i++)
                    {
                      using (SqlCommand cmd =
                       new SqlCommand(srCommandText, connection))
                        {
                            try //catch if any error happens
                            {
                              cmd.CommandTimeout = //set execution timeout
                               PublicSettings.irCommandTimeOutSettings_Second;
                              cmd.CommandType = cdType; //set command type

                              for (int k = 0;
                                    k < lst_Command_Parameters[i].Count; k++)
                                    //add command variables with an order
                                {
                                  cmd.Parameters.AddWithValue(
                                   lst_Command_Variable_Names[k],
                                   lst_Command_Parameters[i][k]);
                                }

                                  cmd.ExecuteNonQuery(); //execute SQL command
                            }
                            catch (Exception E) { //log errors here }
                        }
                    }
                }
            }
            catch (Exception E) { //log errors here}

            return !blErrorHappened;
        }
```

Code Snippet 10: A Shortened Version of Batch Command SQL Query Execution Function of EcCrawler

We present experiments on the effects of batch access to database and static connection performance in section0.Additionally,it is a good practice to update the same tables from the same threads always. This methodology can

prevent crucial table locks and improve the overall performance. However, select statements, in general, are directed to the same table from different threads. Thus, there can be occasions that both select and update statements are executed on the same table at the same time from possibly different connections. In such case, "*transaction was deadlocked on lock resources with another process and has been chosen as the deadlock victim*" error may happen and SQL server always undoes the least amount of work required by the transaction. By default, priority of select transactions is lower than priority of update transactions, so select transactions are denied. The possible solutionto this problem is to check SQL server errors and if there is a resource deadlock victim error when thes elect statement is executed, the same select statement is repeated a pre-determined number of times with a delay. Moreover, every unexpected error should be logged and handled in a proper way.

Pages Processor

Pages processor class processes pages, and translates the semi-structured data into the structured form. These specific processors are also known as wrappers [30] in Information Retrieval (IR)area and they are one of the most important parts of the focused web crawlers. General web crawlers also have page processing tasks (e.g., for link extraction) however, when focused web crawling is made, the task is usually much more complex because the aim of focused crawling is to obtain structured data from unstructured targeted domains.

Commercial systems usually require very high success rate of obtaining targeted data. Obtaining such success rates by using automatic or semi-automatic wrapper generation methodologies (e.g. [31, 32]) are very hard to achievein practice today because of various reasons: (i) browsers have become so robust in the recent years that they display even very erroneous HTML source data correctly and thus data are not presented as structured as before; (ii) complexity of the web pages has significantly increased and the amount of ordinary structured data (e.g., table usage, etc.) has decreased.Therefore, in order to achieve a very high success rate, we decided to develop the wrappers manually for EcCrawler and we obtained very high success rate (i.e., about 100%).As our crawler only visits a pre-determined set of e-commerce websites, we can develop some rules by manually inspecting the source codes of the product pages to extract information about the products. When the manual wrapper crafting methodology is chosen, the system design is extremely important for rule generation. Proper system design makes the rule generator's job extremely easy and improves the software performance. For example, for VatanBilgisayar E-commerce website, if a nobject, that satisfies the below conditions, exists in the source code of the crawled page, the page is determined as a candidate product page and it is processed by page processor agent:

- HTML Object Type: span
- HTML Object Identifier: class
- HTML Object Identifier Name: aktKod

Main task of pages processor is to translate the desired information from raw HTML source code into a structured form by using the wrappers. For this task, regular expressions[34] (regex) can be utilized. However, due to the erroneous nature of the source codes and the complexity of the regexes that are required, using an advanced HTML parsing library

[34]Regular expression – Wikipedia, http://en.wikipedia.org/wiki/Regular_expression

such as Html Agility Pack[35] can be a better solution. Htm Agility Pack is capable of fixing HTML errors and generate properly structured Document Object Model (DOM)[36]. It is open source and written in .NET framework. EcCrawler's wrappers use Html Agility Pack for information extraction task. The system design is completely up to the developer. For EcCrawler, XPath[37] queries are defined for each E-commerce website and executed iteratively to get all of the necessary information (e.g., product title, product categories, etc.) in a structured way. There are several key aspects when building HTML parser with Html Agility Pack framework. Html Document and Html Node are one of the most commonly used classes and they are not System. I Disposable. Because of this reason, setting them to NULL when the object is not necessary anymore, improves the overall performance. One other important aspect is, to check whether an object is NULL or not before using it due to the unpredictable nature of the web crawlers. Additionally, in some cases, parsing JavaScript Object Notation (JSON) data is also necessary. The usage of JSON significantly increases performance by providing more asynchronous pages and better user experience. However, Html Agility Pack is not capable of parsing JSON data. For parsing JSON data, another open source and free library Json.NET[38] is used.

Collaboration and Resource Sharing Between Different Threads

Collaboration and resource sharing between different threads can be done either by using a database system as a middleware or having public functions to access private variables. Using the database as middleware can degrade the system performance due to connection authentication overhead and accessing to the hard disk instead of using RAM. Resource sharing between different threads requires data access to the same objects from multiple threads at the same time. Collaboration between multiple threads can be ensured by properly locking thenon-thread-safe objects before accessing them.

There is a dedicated class in EcCrawler, which holds statistics and events' logs to show on the user interface and save to the hard disk. For keeping events' logs, private static List objects are used and accessed via public functions. For statistics such as how many pages are crawled, public static long objects are used. However, operations on the numerical objects (e.g., integer, long, etc.) are also not-thread-safe, and thus, they need to be either locked or more properly Interlocked method is needed to be used when accessing them for both read or update. Also, to ensure thread safety, it is a necessity to use the same locking object for the same non-thread-safe objects. If multiple objects are used for locking to ensure thread safety of a non-thread-safe object, this may result in either error or data inconsistency. It may even cause deadlocks when a thread locks the first object and at the same time, another thread locks the secondobject, and no locks are released until the same thread locks both of the objects.Code Snippet *11*shows an example usage of Inter locked and object-based locking method.

[35]"*HtmlAgilityPack is an agile HTML parser that builds a read/write DOM and supports plain XPATH or XSLT*", https://htmlagilitypack.codeplex.com/
[36] Document Object Model – Wikipedia, http://en.wikipedia.org/wiki/document_object_model
[37]XML Path Language (XPath) – W3C, https://www.w3.org/TR/xpath/
[38] Popular high-performance JSON framework for .NET, http://www.newtonsoft.com/json

```
Interlocked.Increment(ref GlobalStats.long_GlobalPageFetchFailCount);
//thread safe increment
Interlocked.Read(ref GlobalStats.long_GlobalPageFetchFailCount);
//thread safe read
private static readonly object _lockObject = new object();
//read only object to have lightweight locking

private static void functionName ()
{
    lock (_lockObject) //ensures thread safety inside
    {
        //do operations here, they are thread safe
    }
}
```

Code Snippet 11: Examples for Proper Locking to Provide Thread Safety

Error Log System

In software development,unexpected errors can always happen and in the worst cases they may cause unexpected application terminations. Alsoit is very hard to test the application for all cases and fix all of the possible errors before the public release. In order to minimize these possible errors, software developers tend to follow software release life cycle[39] (e.g., alpha, beta, release candidate, etc.).Therefore, logging every error, and handling unexpected errors are crucial and important. In EcCrawler, an entire class is designed for error-logging. This class uses Stream Writer[40]class to log errors into the text files. The database system can also be used to log errors however for faster logging we have used text files. Stream Writer object is not a thread safeobject and does not Flush[41] after every write automatically by default. So locking is necessary when accessed by multiple threads. Also, it has to be either manually flushed or auto-flush mustbe set to true not to lose any data in case of unexpected application terminations.

Thread Count Management System

The thread count management system is an important part of developing a multi-threaded software. The importance increases when particular tasks need a certain number of active threads constantly. For example, to keep a steady number of active crawling agents for each host individually, we propose that one possible solution is to use the polling methodology by timer based checks. In this method, spawned threads are kept in a thread list. Whenever the timer ticks, the number of unfinished threads is checked, and the required number of new threads are started as shown in Code Snippet 12. This strategy properly ensures that at any given time there exist a certain number of active threads for the task. For multi-threading in .NET 4+, two different classes which are Task[42]and Thread[43]classes can be used. Managing and starting Tasks are relatively easier and have better performance[44] than using Threads. Thus, Task class is preferred starting from .NET 4+ when writing multi-threaded applications. Therefore, EcCrawler uses Task class instead of Thread class.

[39]Software release life cycle – Wikipedia, https://en.wikipedia.org/wiki/Software_release_life_cycle
[40]Stream Writer Class – MSDN, https://msdn.microsoft.com/library/system.io.streamwriter(v=vs.110).aspx
[41]"*Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream*"40
[42]Task Class – MSDN, https://msdn.microsoft.com/en-us/library/system.threading.tasks.task(v=vs.110).aspx
[43]Thread Class – MSDN, https://msdn.microsoft.com/en-us/library/system.threading.thread(v=vs.110).aspx
[44]Task Parallelism – MSDN, https://msdn.microsoft.com/en-us/library/dd537609(v=vs.110).aspx

```
private Timer _timer;
private void Button_Click(object sender, RoutedEventArgs e)
{
    Task.Factory.StartNew(() => //start task as a background thread
    {
        startCheckingTimer();
    }, CancellationToken.None, TaskCreationOptions.LongRunning,
       TaskScheduler.Default);
      //TaskScheduler.Default ensures seperate than main thread
}

private void startCheckingTimer()
{
    _timer = new Timer(doStuffParallel, null, 100, 1000);
      //constantly starts new separate than the main thread tasks
      //with a timed interval even if error happens in the started task
}

private List<Task> startedTaskList = new List<Task>(); //init task list

private int irMaximumNumberOfParallelTasks = 100; //set # of parallel tasks
private void doStuffParallel(Object state)
{
    startedTaskList.RemoveAll(tsk => tsk.Status == //remove completed tasks
                              TaskStatus.RanToCompletion);
    for (int i = 0;
            i < irMaximumNumberOfParallelTasks - startedTaskList.Count; i++)
    {
        Task myTask = Task.Factory.StartNew(() =>
        {
            //do operation here
        }, CancellationToken.None, TaskCreationOptions.LongRunning,
            TaskScheduler.Default);

        myTask.ContinueWith(t => LogError("log error here"),
            TaskContinuationOptions.OnlyOnFaulted);
            //even if unhandled error happens it will be handled and
            //application continue to run
        startedTaskList.Add(myTask);
    }
}
```

Code Snippet 12: An Example for Multi-Thread Management System that Ensures Responsiveness of the UI

EXPERIMENTAL RESULTS AND DISCUSSIONS

In this section experiments performed and their results are presented to evaluate the efficiency and the performance of the proposed methods. All of the experiments are performed on a personal home computer having the below configurations:

- **Operating System**: Windows 8.1 (6.3) Enterprise Edition 64-bit (Build 9600)
- **CPU**: Intel Core i7-2600K CPU @ 3.40GHz, 1 CPU - 4 Cores - 8 Threads, Frequency4544.05 MHz (45 * 100.98

MHz) (Information received by CPU-Z[45] 1.72 x64)

- **RAM**: 32768 MB, DDR3-1346 - Dual Channel, Frequency 673.2 MHz (1:5), Timings 9-9-9-24 (Information received by CPU-Z 1.72 x64)
- **Hard Disk**: 2x OCZ Vertex 4, Raid 0 setup with stripe size 64KB, Write-Cache Buffer Flushing: Enabled, Cache mode: Write back, Sequential read 742 MB/s, write 772 MB/s, Random 512 KB read 482 MB/s, write 486 MB/s. Tests are done at CrystalDiskMark[46] 3.0.4 x64. Access time: 0.127 milliseconds (access time tested by HD Tune Pro[47] 5.50). Other than from synthetic tests, when reading a large amount of data (30 GB+) from the SQL server, the windows task manager displayed630 MB/s read speed with 6000 milliseconds average response time
- **Internet Connection (Network Speed)**: 50 Mbit/s download, 5 Mbit/s upload, personal home fiber connection, provider: Turkcell Superonline[48]
- **Best Performance Settings**: HTTP compression is enabled, initialization of the thread pool is enabled, garbage collector mode is set to server, and concurrent connections limit is increased.

All of the tests (except source code compression) are executed for 30 minutes and in each minute statistics of the conducted tests are collected. In order to collect network traffic statistics, Fiddler[49] framework is used. The RAM usage and the CPU usage statistics are collected by using Performance Counter class and1-second polling methodology which are also used by Windows Task Manager. We use a measure which we call as "UI responsiveness" to evaluate the performance of our system. The main task of our system is called as UI and there fore UI responsiveness is the response rate of the UI to the UI changes. UI responsiveness is a direct indication of the system performance when all of the tasks are started independent from the UI thread since UI may lag only when the application itself is throttling in such scenario. The UI responsiveness is measured by the following strategy:(i) only a class is used to refresh UI by Monitor. TryEnter locking strategy as shown in Code Snippet 5; (ii)after each UI refresh is completed, the time of the refresh is recorded; (iii)the UI is updated per 0.5 seconds. We have chosen updating per 0.5 seconds because "*high update speed*" profile of Windows Task Manager also uses0.5seconds interval to refresh the statistics. So after the application is started, each second has exactly 2 times UI refresh process. If the corresponding second has any missing refresh, it is counted as UI freeze, if the second has more than 2 times UI refresh, it is counted as UI lag (delayed update consecutively). The total responsiveness of the UI is measured as the number of correct UI refresh matches in each second divided by the number of total expected UI refresh. So if the UI is refreshed 2 times within each second, the UI responsiveness is 100%. Refresh count over than 2 or less than 2 in a second means there is either UI lag or UI freeze which both reduce UI responsiveness.

Average application threads count is calculated by using1-second polling methodology and Process. GetCurrent Process().Threads. Count[50] method. Maximum 300 concurrent crawling agents are used for experiments and maximum crawling count for each website is limited individually. We have used 50 different E-commerce sites for the experiments

[45]CPU-Z is a freeware that gathers information on some of the main devices of your system, http://www.cpuid.com/softwares/cpu-z.html
[46] CrystalDiskInfo is a HDD/SSD utility software (open source) which supports S.M.A.R.T and a part of USB-HDD, http://sourceforge.jp/projects/crystaldiskinfo/
[47] http://www.hdtune.com
[48]http://www.superonline.net/kesfet/fiber-internet
[49] The free web debugging proxy for any browser, system or platform, http://www.telerik.com/download/fiddler
[50] MSDN, https://msdn.microsoft.com/en-us/library/system.diagnostics.process.getcurrentprocess(v=vs.90).aspx

and some of them are listed in *Table 1*.When we have tested the limit for the default number of connections per host, a single website is used for experimenting so that the soft ware can possibly reach the number of concurrent connections per host threshold. Every crawled page is processed to extract URLs in the crawled page, determine the status of crawled page as canonical or a product page or not, etc. However, page processing that is applied to product pages involves extraction of product title, product categories, product features, product comments, etc. To determine type of the page that is processed, EcCrawler needs to determine whether the crawled pages are “a product page” or “not” and this is decided at the link extraction phase by applying manually generated rules. If the page is not a product page, it is not saved for further processing. During the tests, the number of page processing tasks’ count was set to 100.

The tests, other than the source code compression, depend on the network capacity, the hard disk performance, and the SQL-Server caching. Because of the sere asons, all of these experiments are conducted after 1 am local time for politeness and more objectivity. In addition, before the test, the database is reset (returned to initial values), the computer’s IP address is changed and the computer is restarted. These actions were taken in order to solve the following possible issues:

- It can be expected that the E-commerce websites’ server load will be lower after the midnight due to decreased visitors’ activity. After midnight crawling also improves the chances of not harassing the crawled websites.
- Both home’s internet connection network load and the crawled E-commerce websites’ server network load changes throughout the day. Because of this reason, making all such network load based tests after 1 AM local time should improve the reliability of the experiments.
- Windows 8.1’s disk system, SQL-Server, and Windows 8.1’s DNS system make heavy caching of many different requests to improve the performance. Conducting experiments consequently can yield unfairness among the various tests because of these caching. After each one of the tests, the computer is restarted to conduct the tests in equal conditions.
- Since the home network is used to carry out the tests and the ISP provides dynamic IP address, the IP address is changed after each one of the tests to prevent any possible IP blockage. However, this may not have been necessary becausewhile conducting the experiments, we did not notice any IP blockage or we did not receive any complaint from any of the crawled web servers’ administrators. Not receiving any complaints or not getting blocked may show that the methods used are sufficient to handle multiple connections from each client without any trouble.

Database Communication

In order to verify the effectiveness of the proposed database connection system, we have conducted the following 3 different tests: (i) in the first test scheme, no batch database access is used, and for each database access a new connection session is opened and closed; (ii) in the second test scheme, batch database access methodology is used and for multiple database accesses, a single connection session is used and closed; (iii) in the third test scheme, a static connection is employed for all of the database accesses between all of the different threads by using locking methodology to ensure thread safety, and it is never closed. The difference from the batch database access mode is, it uses a single connection to execute multiple different queries. So it should not be confused with batch SQL queries. Therefore, it reduces overhead of SQL-Server connection session handling.

Database communication experiments are conductedon the EcCrawler system. However, we did not observe any significant difference between these three modes. The reasons for this observation may be that the system has a lot of very heavy system demanding tasks when compared to the database session generation and there may be not enough number of database connection procedures to observe the effectiveness of the proposed systems. Additionally, the executed queries in the EcCrawler were already heavily tuned to obtain maximum performance. Because of these reasons, we decided to conduct synthetic tests that will heavily use the database connections.

In order to carry out the synthetic tests, a new table is designed which have a single Integer 32(int) primary key column. The numbers between 1 and 10,000 are inserted into this table as primary keys. Then a new software is developed to execute the tests. This software spawned 250 Tasks (each one is separate from the main thread) and each one of these Tasks continues to work constantly until the application is terminated. These Task squeries the database and selects a single Data Row from the synthetic test table by generating a random primary key predicate. Then database accesses of the Tasks are counted cumulatively. In the batch database access methodology, a single new database connection for each Task is used 50 times before it is closed. The table and a sample query used in the experiments are presented in Figure 3. The reason for choosing such a high number of Tasks is to maximize the system resources usage in all cases. These values are not optimal values for the best system performance. Also, very simple database design is chosen to evaluate the database connection performance rather than the database querying performance. The results of these experiments are shown in Table 2.

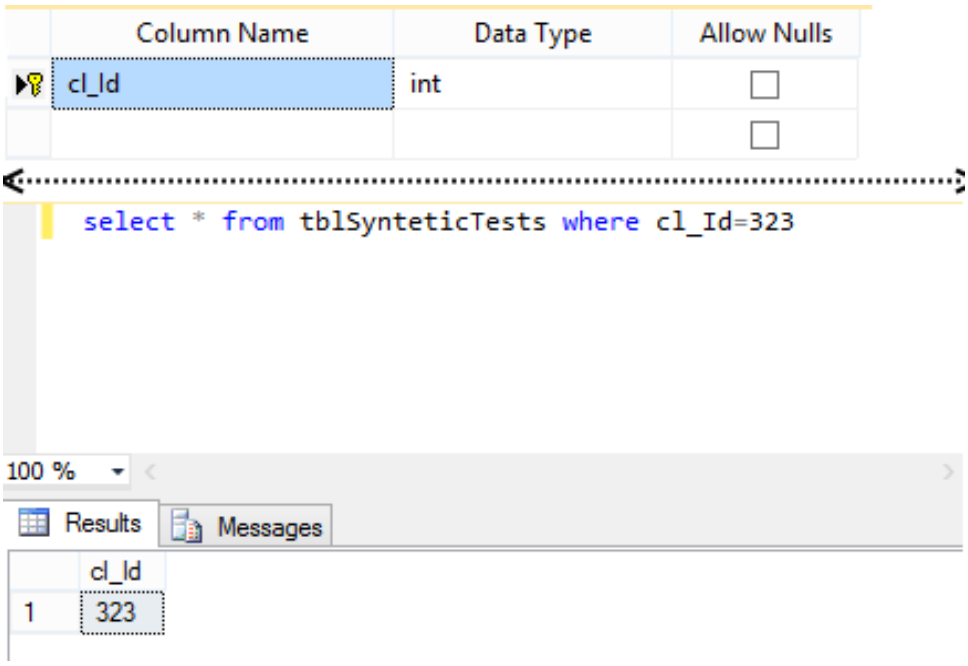

Figure 3: Table and Query Used in the Synthetic Tests

Table 2: Experimental Results for Synthetic Database Connection

Test Scheme	Average CPU Usage of the Test Software	Average CPU Usage of the SQL-Server	Average Number of Completed Queries per Minute
Scheme 1 - Single Queryper Database Connection	53.90%	45.12%	3,120,492

Scheme 2 - Batch Mode – Multiple Queriesper Database Connection	51.21%	45.65%	3,982,170
Scheme 3- A Single Static Connection for all of the Threads and the Queries	99.17%	0.49%	58,891

When the results in Table 2areanalyzed, Scheme 2 is the best performer by almost equalizing the CPU utilization of both software and SQL-Server. Thus, it also achieves to obtain highest number of queries per minute. When Scheme 1 and Scheme 2are compared, the CPU usage of software in Scheme 1 is slightly higher than that of the Scheme 2. The possible reason of this can be that the software spends more time to establish a new session for each query instead of using the same session for multiple queries. When the results of Scheme 3are interpreted, we see that the test software uses a massive amount of the available system resources, and SQL-Server utilization is poor. The possible reason of this is, each time a Task demands a database access, it has to lock the static database connection. If the connection is already locked, it has to wait. This locking process is significantly system demanding, and this is probably the main cause of unbalanced CPU usage between the test software and the SQL-Server and results in a lower performance.

Source Code Compression

Page processing is a much more challenging task when it is compared to crawling pages and saving their source code task. Because of this reason, page processing and page source saving task shave to be asynchronous. Since the crawling task is much faster than the page processing task, to achieve asynchronous execution, source codes of the web pages have to be kept until they are processed. However, storing the raw source code of the crawled pages takes huge space both on the disk and in the RAM. Finding the optimal compression algorithm for the developed crawler can help to reduce space requirement. If the disk space usage is more crucial than the CPU requirement for the developed software, a more aggressive compression can be chosen. However, the obtained experimental results indicate that one of the compression methods clearly surpasses every other method by the *CPU_requirement/compression* ratio.

For the compression tests, the dataset which is generated by EcCrawler is used. 100,000 pages are randomly chosen from the dataset. The features of the pages in the dataset are as follows: maximum source code size (UTF-8) of a single page is 8.54 Megabytes, the average size is 86.68 Kilobytes, the minimum size is 23.68 Kilobytes, and total size of all uncompressed source codes size is 8.26 Gigabytes. When testing Gzip compression algorithm, .NET frame work is used because it natively supports Gzip. For testing the other compression algorithms, 7-Zip[51]library is used with Compression Level set to Normal. To use 7-Zip DLL file in C#, Seven Zip Sharp[52]is used as a wrapper.

The results of thecompression experiments are presented inTable 3 where **Min**,**Max**,and **Avg Compressed Source Size** are the minimum,the maximum, and average size of the compressed source code of acrawled page in kilobytes;**Min**,**Max**, and **Avg Compression Time** are the minimum,the maximum, and average elapsed time for compressing source code of a crawled page in milliseconds;**Min**,**Max**, and **Avg Decompression Time** are the minimum,the maximum, and average elapsed time for decompressing source code of a crawled page in milliseconds;**Min**,**Max**, and **Avg Compression Benefit** are the minimum,the maximum, and averageof the percentage difference between the size of the compressed andraw source codes of a crawled page;**Avg CPU Usage** is the average CPU

[51]Open source 7-Zip is a file archiver with a high compression ratio, http://www.7-zip.org
[52] Managed 7-zip library written in C#, https://sevenzipsharp.codeplex.com/

usage of the system while compression experiments are running.

Table 3: Comparison of the Compression Methods

Mode:	Gzip	BZip2	Deflate	Deflate 64	Lzma	Lzma2	Ppmd
Min Compressed Source Size (KB):	9.01	9.30	8.75	8.75	**8.74**	8.75	8.08
Max Compressed Source Size (KB):	8,471	**8,458**	8,472	8,472	8,470	8,472	8,577
Avg Compressed Size (KB):	24.97	24.09	22.78	22.78	22.78	22.79	**21.80**
Min Compression Time (milliseconds):	**0**	13	27	15	22	27	13
Max Compression Time (milliseconds):	**366**	637	4,700	1,977	4,679	4,718	3,930
Avg Compression Time (milliseconds):	**1.53**	42.03	61.27	39.20	57.99	62.07	38.37
Min Decompression Time (milliseconds):	**0**	10	20	9	19	20	11
Max Decompression Time (milliseconds):	**178**	732	541	527	567	538	4,369
Avg Decompression Time (milliseconds):	**0.39**	31.14	57.93	30.57	57.49	57.15	30.66
Min Compression Benefit (%):	2.42	2.40	3.14	3.14	**3.16**	3.14	1.94
Max Compression Benefit (%):	91.77	93.33	92.93	92.93	92.93	92.93	**93.50**
Avg Compression Benefit (%):	71.18	72.20	73.70	73.70	73.71	73.70	**74.84**
Avg CPU Usage (%):	17.51	16.67	**16.22**	19.16	16.26	16.43	17.31

In Table 3, the best values for each performance evaluation parameter are written in bold face. According to the results that are presented in Table 3, Gzip clearly outperforms the other methods when both the compression ratio and the compression times are considered. Even though the Gzip obtains lower average compression ratio than all of the other methods, it is at least 15 times faster than the other methods for the compression task and 78 times quicker than the other methods for the decompression task. The compression performance gain of the other methods is extremely low when compared with their increased time requirement and because of this reason EcCrawler uses Gzip.

Using Compression While Fetching Pages

Today, the majority of the web servers supports compressed data transferring, however by default, .NET does not make the HTTP requests with compression enabled. Necessary headers are needed to be includedin the requests, and automatic decompression needs to be enabled as shown in Code Snippet 8, to fetch web pages with HTTP compression. The results of the HTTP compression enabled tests, and the default Http Web Request class tests are presented in Table 4 where the gray highlighted rows are the best performance settings (i.e., compression enabled) and the not highlighted (white) rows are the results of the same settings when HTTP compression is deactivated.

In Table 4;**Crawl Success** is the number of successfully crawled web pages;**Crawl Fail**is the number of unsuccessfullycrawled web pages;**Avg # of Pages per Minute Crawl**is the average number of successfully crawled web pages per minute;**Avg # of Processed Pages per Minute** is the average number of processed pages per minute;**Avg Download Speed per Second** is the average bandwidth in kilobytes used by the application per second to download web pages;**Avg Upload Speed per Second**is the average bandwidth in kilobytes used by the application per second to upload

data (upload is used for only connecting the websites) to the web servers;**Discovered Unique URLs**is thenumber of discovered unique web pages. Other metrics used in the table are self-explanatory.

Table 4: The Effect of Compression Enabled HTTP Requests Experiment

HTTP Compression (Minutes)	5	10	15	20	25	30
Crawl Success	30,917	62,595	98,092	134,875	171,821	210,189
	19,860	40,824	62,178	84,260	106,032	126,735
Crawl Fail	114	459	1,348	1,622	1,697	1,754
	199	486	740	991	1,828	2,818
Avg# of Pages per Minute Crawl	6,086	6,195	6,485	6,693	6,827	6,960
	3,914	4,046	4,115	4,184	4,213	4,198
Avg# of Processed Pages per Minute	815	805	780	738	715	704
	701	722	699	693	679	670
Avg Download Speed Per Second (KiloBytes)	2,536	2,550	2,665	2,633	2,631	2,660
	5,645	5,759	5,790	5,769	5,791	5,805
Avg Upload Speed Per Second (KiloBytes)	38	39	42	44	44	45
	21	21	21	22	22	22
Discovered Unique URLs	153,994	207,914	259,737	309,515	346,910	383,869
	115,564	168,024	211,293	236,519	261,804	291,895
Avg RAM Usage (MegaBytes)	1,003	1,106	1,177	1,229	1,275	1,292
	1,031	1,118	1,153	1,179	1,198	1,215
Avg CPU Usage (%)	59.74	62.07	63.36	63.89	64.35	64.71
	32.78	32.45	33.01	33.01	33.11	33.04
Avg# of Threads Spawned	542	555	553	554	558	562
	623	644	663	696	725	756
UI Responsiveness (%)	98	98.41	98.22	97.95	97.83	97.80
	98.5	98.58	98.66	98.62	98.56	98.61
UI Freeze (%)	2	1.58	1.77	2.04	2.81	2.19
	1.5	1.41	1.33	1.37	2.16	1.38
UI Lag (%)	0.33	0.16	0.11	0.16	0.2	0.16
	0	0.08	0.05	0.08	0.06	0.05

When Table 4 is analyzed the benefit of HTTP compression is clearly observed. As mentioned in the testing platform specifications, the maximum download speed of the internet connection was 50 Mb its per second, which is roughly equal to 6400 Kilo Bytes per second. Once the HTTP compression was disabled, the testing system network load was maximized, and this hindered the number of pages fetched per minute. Enabling HTTP compression increases the number of pages crawled per minute by 65.79% and decreases the bandwidth usage by 54.17%. Another important aspect of these tests is the number of processed pages per minute. In the first 5 minutes, the number of processed pages per minute is significantly higher than the HTTP compression disabled mode. However, this difference decreases as the time passes. The reason of this is, at the beginning, there are not enough product pages to process until the page processing tasks

countis maximized. Thus, the faster page crawling mode is obtained by processing more pages. However, as more pagesare crawled, the number ofobtained product pages increases and the maximum capacity of the page processing tasks are surpassed and the gapis closed and theaverage number of pages processed per minute for both modes almost becomes equal.

The reason of having differencebetween the two modes in terms of average number of threads spawned is that, when the HTTP compression is enabled, it fetches pages faster. Thus, the operation is completed faster, and the crawler thread is terminated quicker than the HTTP compression is disabled mode. When the HTTP compression is enabled, the UI responsiveness reduces slightly due to the fact that, in the case of compression, more pages are processed, thus heavier processing is done.

Initialization of Thread Pool

By default, .NET framework spawns a certain number of threads by the pre-defined heuristics to maximize system performance. However,.NET framework heuristics may result in lower performance by using as mallerportion of the available resources. Hence, initialization of thread pool manually may improve the system performance greatly. The initialization of the thread pool is shown in Code Snippet 1. In Table 5,the results of the initialization of the thread pool experiment are presented. In the table, the gray highlighted rows include results of the best performance settings in which thread pool is initialized, and the not highlighted (white) rows display results of the same settings except that the thread pool is not initialized.

Thread pool experiment clearly shows the performance gain at the UI responsiveness. Even though the number of active threads increasesover the time as the framework optimizes the count of the activethreads, the performance loss of the default settings is still significant. Initialization of the thread pool utilizes the application for using the system resources more efficiently and achieves better performance.

Table 5: Result of the Thread Pool Initialization Experiment

Thread Pool (Minutes)	5	10	15	20	25	30
Crawl Success	30,917	62,595	98,092	134,875	171,821	210,189
	16,727	45,068	71,034	97,986	121,094	131,132
Crawl Fail	114	459	1,348	1,622	1,697	1,754
	110	252	547	715	969	1,182
Avg# of pages per Minute Crawl	6,086	6,195	6,485	6,693	6,827	6,960
	3,017	4,260	4,550	4,744	4,691	4,244
Avg# of Processed Pages per Minute	815	805	780	738	715	704
	377	585	678	692	701	664
Discovered Unique URLs	153,994	207,914	259,737	309,515	346,910	383,869
	99,360	155,312	195,540	228,970	252,695	263,187
Avg RAM Usage (Mega Bytes)	1,003	1,106	1,177	1,229	1,275	1,292
	838	1,056	1,139	1,186	1,210	1,236
Avg CPU Usage (%)	59.74	62.07	63.36	63.89	64.35	64.71
	36.68	40.88	40.41	39.67	38.14	34.14
Avg # of Threads Spawned	542	555	553	554	558	562
	358	417	448	466	478	500
UI Responsiveness (%)	98	98.41	98.22	97.95	97.83	97.80

	41.5	58.33	64.66	68.04	69.46	67.47
UI Freeze (%)	2	1.58	1.77	2.04	2.81	2.19
	58.5	41.66	35.33	31.95	30.53	32.52
UI Lag (%)	0.33	0.16	0.11	0.16	0.2	0.16
	22.33	19.25	17.33	17.20	17.16	18.47

Garbage Collector Mode Experiment

By default, the garbage collector mode is set to workstation, however, this mode is designed for the single core processors. Nowadays, the majority of the computers sold in the market have multi-core processors. We did not have any single-core computer to test the mode difference, however, on the testing platform, changing workstation mode to server mode improves overall performance significantly. The garbage collector mode is set as shown in Code Snippet 5. In Table 6,the gray highlighted rows present results for the best performance settings in which garbage collector mode is set to server and the not highlighted rows include results for the same settings except garbage collector mode is workstation.

If all of the tasks are running as background threads, the UI thread is not used for any processes, and if there exist senough system resources to refresh the UI, there should not be any UI responsiveness problem. However, if the application utilizes many threads and these threads heavily uses system resources as in EcCrawler, default settings may fail to supply necessary system resources to the application, and this may cause the UI to freeze or lag. We propose that this problem can be solved by manual initialization of the thread pool and setting the garbage collector mode.

Table 6: Results of the Garbage Collector Mode Experiment

Garbage Collector Mode (Minutes)	5	10	15	20	25	30
Crawl Success	30,917	62,595	98,092	134,875	171,821	210,189
	7,811	17,563	25,740	32,789	39,868	48,147
Crawl Fail	114	459	1,348	1,622	1,697	1,754
	29	85	104	121	131	162
Avg# of Pages per Minute Crawl	6,086	6,195	6,485	6,693	6,827	6,960
	1,529	1,730	1,690	1,616	1,568	1,580
Avg# of Processed Pages per Minute	815	805	780	738	715	704
	518	622	627	633	619	618
Avg Download Speed Per Second (KiloBytes)	2,536	2,550	2,665	2,633	2,631	2,660
	518	550	545	510	495	498
Avg Upload Speed Per Second (KiloBytes)	38	39	42	44	44	45
	9	10	10	10	10	10
Discovered Unique URLs	153,994	207,914	259,737	309,515	346,910	383,869
	70,157	102,738	126,387	139,490	149,314	163,116
AvgRAM Usage (MegaBytes)	1,003	1,106	1,177	1,229	1,275	1,292
	643	648	666	699	734	751
Avg CPU Usage (%)	59.74	62.07	63.36	63.89	64.35	64.71
	25.64	28.21	28.60	28.20	28.22	28,72
Avg# of Threads Spawned	542	555	553	554	558	562
	598	606	617	624	627	632
UI Responsiveness (%)	98	98.41	98.22	97.95	97.83	97.80
	75.33	71.25	67.94	64.70	62.43	61.61
UI Freeze (%)	2	1.58	1.77	2.04	2.81	2.19
	24.66	28.75	32.05	35.29	37.56	38.38
UI Lag (%)	0.33	0.16	0.11	0.16	0.2	0.16
	2.5	3.08	2.94	2.62	2.53	2.36

The experimental results that are presented in Table 6 clearly display the huge performance improvement when we set garbage collector mode to server. The gray highlighted rows present the results obtained when garbage collector mode is set to server, and other rows show the results for the default case. As it can be seen from the table, this mode change boosts the number of crawled URLs by 336%, and improves the UI responsiveness by 58.74%.

Concurrent Connections Limit Experiment

In .NET framework the limit for the number of concurrent connection sper host is two by default. However, crawlers are more likely to need more than two connections at the same time for each host when craw ling gigantic websites. EcCrawler distributes crawling tasks for each host individually by using pre-assigned values which are10 by default. However with having 10crawling tasks per host, it is quite hard to reach maximum consistent two connections per host limitation because of the application processing delays. Therefore, for this experiment, one of the biggest E-commercewebsites alone is used with 300concurrentcrawling agents.

Table 7: Results for the Concurrent Connection Limit Experiment

Concurrent Connections Limit (Minutes)	5	10	15	20	25	30
Crawl Success	36,804	76,486	118,736	145,322	175,070	203,082
	24,841	54,223	83,540	110,386	138,644	178,467
Crawl Fail	57	76	95	100	104	105
	0	3	4	4	4	4
Avg# of Pages per Minute Crawl	7,265	7,588	7,865	7,221	6,961	6,731
	4,898	5,375	5,532	5,488	5,517	5,919
Avg# of Processed Pages per Minute	968	823	761	708	695	676
	781	779	734	698	680	674
Avg Download Speed Per Second (KiloBytes)	1,582	1,643	1,699	1,558	1,501	1,450
	1,075	1,167	1,194	1,182	1,185	1,269
Avg Upload Speed Per Second (KiloBytes)	47	48	49	45	43	41
	30	32	33	33	33	35
Discovered Unique URLs	87,375	126,071	154,488	174,936	200,291	222,008
	70,057	110,788	136,557	156,776	176,680	203,628
AvgRAM Usage (MegaBytes)	763	798	829	862	884	903
	734	797	839	859	870	882
Avg CPU Usage (%)	54.84	54.18	54.16	47	44.01	41.26
	31.08	33.89	33.98	33.46	33.11	36.67
Avg # of Threads Spawned	568	567	558	565	567	570
	571	569	569	570	570	563
UI Responsiveness (%)	98	98.08	98	97.91	97.86	97.86
	98.83	98.66	98.72	98.79	98.8	98.8
UI Freeze (%)	2	1.91	2	2.08	2.13	2.13
	1.66	1.33	1.27	1.2	1.2	1.194
UI Lag (%)	0	0	0	0.08	0.06	0.08
	0	0.16	0.16	0.12	0.13	0.11

How to increase the maximum limit and the other parametersare shown in Code Snippet 9. The results of this experiment are given in Table 7, where gray highlighted rows show the results of the best performance settings

(i.e., maximum consistent connections per host limit is 1000) and the other rows display results of the same settings except that the connection limit is 2.When the results given in Table 7areanalyzed, there is a significant performance boost in the first 5 minutes. Then, this performance gain decreases gradually, because, the number of the URLs to be discovered decreases as a greater portion of the website is crawled. At the 30th minute, since the majority of the sites were already crawled, the number of the crawled pages almost become equal for the two methods because there were not left many pages to crawl.

All Default Settings Experiment

In the final experiment, the application is run without any advanced optimizations to show the effects of the proposed methods. To disable all of the optimizations, HTTP Compression and KeepAlivefacilities are removed from the web page fetcher function; thread pool initialization and the limit on the maximum consistent connection per host are removed; and finally, garbage collector mode setup is removed from the APP. config. The results of this experiment are presented in Table 8 where the gray highlighted rows display the results of the best performance settings whi lethe other rows show the results of alldefault settings.

When Table 8 is analyzed, a lot of significant improvements in the application performance can be observed. The proposed configurations in this paper increase the number of crawled pages by 472%, decrease the UI freeze by 95.77%, reduce the UI lag by 96.39%, and improve the UI responsiveness by 102% with respect to default settings. Overall application performance isimproved and the application is able to utilize the available system resources.

Table 8: Results for all Default Settings Experiment

All Default Settings (Minutes)	5	10	15	20	25	30
Crawl Success	30,917	62,595	98,092	134,875	171,821	210,189
	4,232	8,628	14,546	21,628	29,445	36,728
Crawl Fail	114	459	1,348	1,622	1,697	1,754
	57	81	105	151	233	325
Avg # of Pagesper Minute Crawl	6,086	6,195	6,485	6,693	6,827	6,960
	800	818	934	1,038	1,127	1,173
Avg # of Processed Pages per Minute	815	805	780	738	715	704
	295	395	463	514	547	555
Avg Download Speed Per Second (KiloBytes)	2,536	2,550	2,665	2,633	2,631	2,660
	1,146	1,117	1,233	1,348	1,437	1,458
Avg Upload Speed Per Second (KiloBytes)	38	39	42	44	44	45
	3	3	4	4	5	5
Discovered Unique URLs	153,994	207,914	259,737	309,515	346,910	383,869
	45,317	62,624	79,247	90,788	103,691	110,341
AvgRAM Usage (MegaBytes)	1,003	1,106	1,177	1,229	1,275	1,292
	787	942	918	880	856	869
Avg CPU Usage (%)	59.74	62.07	63.36	63.89	64.35	64.71
	17.80	19.74	21.64	23.15	24.29	24.66
Avg # of Threads Spawned	542	555	553	554	558	562
	369	450	470	483	488	497
UI Responsiveness (%)	98	98.41	98.22	97.95	97.83	97.80
	18.16	20.08	32.22	40.66	45.43	48.19

UI Freeze (%)	2	1.58	1.77	2.04	2.81	2.19
	81.83	79.91	66.77	59.33	54.56	51.80
UI Lag (%)	0.33	0.16	0.11	0.16	0.2	0.16
	9.5	8.91	6.66	5.54	5.03	4.44

CONCLUSIONS

The methods we have presented in this study for the focused web crawler development process can significantly help .NET developers for any software development. According to our experiments, using .NET framework without fine tuning can significantly hinder the potential application performance and decrease the utilization of system resources. There can be even more performance tunings, however during EcCrawler development, source code compression, HTTP compression while fetching web pages, initialization of the thread pool, garbage collector mode setup and the maximum concurrent connection limit are the major ones that are discovered and applied. The proposed methods and experimental results presented in this study will help .NET developers to boost their software performance and save their time to figure the bottlenecks that can affect the expected performance of their application. For the future work, EcCrawler design and implementation may be modified to make it compatible for running on a cluster of computers rather than on a single machine. Additionally, a classifier may be embedded to the EcCrawler to determine product pages automatically.

ACKNOWLEDGEMENTS

This work was supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK) scholarship 2211-C.

REFERENCES

1. Chakrabarti S, Berg MVD, Dom B. Focused crawling: a new approach to topic-specific Web resource discovery. Computer Networks 1999;31:1623–40. DOI:10.1016/s1389-1286(99)00052-3.
2. Gupta, Satinder Bal. The Issues and Challenges with the Web Crawlers. International Journal of Information Technology & Systems 2012;**1** (1):1-10.
3. Boldi P., Codenotti B., Santini M., and Vigna S. Ubicrawler. A scalable fully distributed web crawler. Software: Practice and Experience 2004; **34**(8): 711–726. DOI:10.1002/spe.587
4. Gomes D., and Silva M.J. The Viúva Negra crawler: an experience report. Software: Practice and Experience 2008; **38**(2): 161–188. DOI:10.1002/spe.v38:2
5. Heydon, Allan, and Marc Najork. Mercator: A scalable, extensible web crawler. World Wide Web 2.4 (1999): 219-229. DOI:10.1023/A:1019213109274
6. Shkapenyuk Vladislav, and Torsten Suel. Design and implementation of a high-performance distributed web crawler. Data Engineering, 2002. Proceedings. 18th International Conference on. IEEE, 2002.DOI:10.1109/icde.2002.994750.
7. Nisha J, and Sundareswari K. Clustered Based User-Interest Ontology Construction for Selecting Seed URLs of Focused Crawler. International Journal of Innovative Research in Computer and Communication Engineering 2015; **3** (2): 827–830.

8. Uzun E., Güner E.S., Kılıçaslan Y., Yerlikaya T., and Agun H.V. An effective and efficient Web content extractor for optimizing the crawling process. Software: Practice and Experience 2014; **44**(10): 1181–1199. DOI:10.1002/spe.2195

9. Dahiwale P., Raghuwanshi M.M., and Malik L. PDD Crawler: A focused web crawler using link and content analysis for relevance prediction. In the Proceedings of SEAS-2014 – Third International Conference on Software Engineering and Applications November 7–8, 2014, Dubai, UAE. arXiv preprint arXiv:1411.4366

10. Yohanes B.W., Handoko H., and Wardana H.K. Focused Crawler Optimization Using Genetic Algorithm. TELKOMNIKA Telecommunication, Computing, Electronics and Control 2011; **9**(3): 403–410. DOI:10.12928/telkomnika.v9i3.730

11. Papavassiliou V., Prokopidis P., and Thurmair, G. A modular open-source focused crawler for mining monolingual and bilingual corpora from the web. In the Proceedings of the Sixth Workshop on Building and Using Comparable Corpora August 8, 2013, Sofia, Bulgaria, pp.43–51.

12. Yanni L., Wang Y., and Du J. E-FFC: an enhanced form-focused crawler for domain-specific deep web databases. Journal of Intelligent Information Systems 2013; **40** (1): 159–184. DOI:10.1007/s10844-012-0221-8

13. Kumar M., and Vig R. Focused crawling based upon tf-idf semantics and hub score learning. Journal of Emerging Technologies in Web Intelligence 2013; 5(1): 70–77. DOI:10.4304/jetwi.5.1.70-77

14. Liu H., and Milios E. Probabilistic models for focused web crawling. Computational Intelligence 2012; **28** (3): 289–328. DOI:10.1111/j.1467-8640.2012.00411.x

15. Bedi P., Thukral A., Banati H., Behl A., and Mendiratta V. A multi-threaded semantic focused crawler. Journal of Computer Science and Technology 2012; **27**(6): 1233–1242. DOI:10.1007/s11390-012-1299-8

16. Maimunah S., Sastramihardja H.S., Widyantoro D.H., Kuspriyanto K. CT-FC: more Comprehensive Traversal Focused Crawler. TELKOMNIKA Telecommunication, Computing, Electronics and Control 2012; **10**(1): 189–198. DOI:10.12928/telkomnika.v10i1.777

17. Liu W., and Du Y. An improved topic-specific crawling approach based on semantic similarity vector space model. Journal of Computational Information Systems 2012; **8**(20): 8605–8612.

18. Edwards, Jenny, Kevin McCurley, and John Tomlin. An adaptive model for optimizing performance of an incremental web crawler. Proceedings of the 10th international conference on World Wide Web. ACM, 2001. DOI:10.1145/371920.371960

19. Olston, Christopher, and Marc Najork. Web crawling. Foundations and Trends in Information Retrieval 2010;**4** (3): 175-246. DOI:10.1561/1500000017

20. Castillo, Carlos. Effective Web Crawling. Ph.D. thesis (2004).

21. Najork, Marc, and Janet L. Wiener. Breadth-first crawling yields high-quality pages. Proceedings of the 10th international conference on World Wide Web. ACM, 2001. DOI:10.1145/371920.371965

22. Pinkerton, Brian. Finding what people want: Experiences with the WebCrawler. Proceedings of the Second International World Wide Web Conference. Vol. 94. 1994.

23. Cho, Junghoo, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through URL ordering. (1998).
24. Cho, Junghoo, and Hector Garcia-Molina. Effective page refresh policies for web crawlers. ACM Transactions on Database Systems (TODS) 28.4 (2003): 390-426. DOI:10.1145/958942.958945
25. Cho, Junghoo, and Uri Schonfeld. Rankmass crawler: a crawler with high personalized pagerank coverage guarantee. Proceedings of the 33rd international conference on Very large data bases. VLDB Endowment, 2007.
26. Abiteboul, Serge, Mihai Preda, and Gregory Cobena. Adaptive on-line page importance computation. Proceedings of the 12th international conference on World Wide Web. ACM, 2003. DOI:10.1145/775152.775192
27. Baeza-Yates, Ricardo, et al. Crawling a country: better strategies than breadth-first for web page ordering. Special interest tracks and posters of the 14th international conference on World Wide Web. ACM, 2005. DOI:10.1145/1062745.1062768
28. Chien, Steve, et al. Link evolution: Analysis and algorithms. Internet mathematics 1.3 (2004): 277-304. DOI:10.1080/15427951.2004.10129090
29. Cho, Junghoo, and Hector Garcia-Molina. "The Evolution of the Web and Implications for an Incremental Crawler." In Proceedings of the 26th International Conference on Very Large Data Bases, pp. 200-209. Morgan Kaufmann Publishers Inc., 2000.
30. Kushmerick, Nicholas. Wrapper induction for information extraction. Diss. University of Washington, 1997.
31. Doorenbos, Robert B., Oren Etzioni, and Daniel S. Weld. A scalable comparison-shopping agent for the world-wide web. Proceedings of the first international conference on Autonomous agents. ACM, 1997. DOI:10.1145/267658.267666
32. Yang, Jaeyoung, et al. A More Scalable Comparision Shopping Agent. In Proc'EIS2000 Engineering of Intelligent Systems. 2000.