An Object-oriented Design and Implementation of a Simulator for Caching with Related Content

Sarvar Abdullaev, Franz I. S. Ko
Division of Computer and Multimedia, Dongguk University, 707 Seokjang-dong, Gyeongju-si, Gyeongsangbuk-do, 780-714 Korea,
sarrtv@yahoo.com, isko@dongguk.edu

Abstract

Web caching server is one of the important components of any web site, as it makes the access of user to web content much faster while balancing the network and server load. So it is vital to deploy caching server with efficient algorithm which would cache only commonly requested content. There are many approaches have been proposed in order to solve this problem. From theory we know several caching algorithms like FIFO, LRU and LRU-min. Most of them have their advantages and disadvantages based on specific context. The purpose of this paper is to introduce the new concept of caching with related content and to show its efficiency through the use of simulation program. We will also discuss the object oriented design and implementation of caching simulator. We used 5 design patterns such as Strategy, Composite, Façade, Proxy and Flyweight while designing the object-oriented structure of our caching simulator. We also used UML tool in order to describe the static and dynamic design of our caching simulator. The purpose of implementing above mentioned simulator is to check the efficiency of proposed caching method with related content against the efficiency of other standard caching algorithms.

1. Introduction

Nowadays while the content over the Internet is growing dramatically, it is natural that the access to that web content is becoming terser and the need for efficient web caching servers are constantly growing. From theory we know that web cache is generally located between web server (origin server) and clients accessing the web content. One of its major functions is to handle the requests coming from clients. Web cache retrieves the requested content from its cached memory, but if it doesn't exist, it will request it from web server and will copy it to its own memory. With such simple mechanism, web cache could reduce the latency, because the request is satisfied from the cache (which is closer to the client) instead of the origin server, it takes less time for it to get the representation and display it. This makes the Web seem more responsive. Moreover it reduces the network load as the representations are reused and it reduces the amount of bandwidth used by a client. This saves money if the client is paying for traffic, and keeps their bandwidth requirements lower and more manageable.

In this paper, we mostly focus on proxy caches which are located remote from origin server and client. We will study different classical approaches like FIFO, LRU and LRU-min which are commonly used for caching web content. Then we will introduce new concept of caching with related content. Further we will focus on the design and implementation of simulating program for web caching. This simulating program will encompass the simulation of standard caching algorithms and a newly proposed caching with related content. Thus we can compare the efficiency of caching algorithms with and without newly proposed approach. We will build a web caching simulator using 5 design patterns such as Strategy, Composite, Façade, Proxy and Flyweight [1]. Through the use of design patterns we will make robust and scalable object-oriented design for our caching simulator. We will illustrate the design of a simulator with use of UML static and dynamic diagrams. The simulator is implemented using object-oriented programming language. In conclusion, we will show the results derived from running the simulator program and analyze them. Generally speaking, the use of object oriented concepts and the implementation of well-known design pattern in building simulator program is innovative approach by itself. But our main purpose of implementing this simulator is to highlight the efficiency of proposed web caching approach versus standard ones.

2. Related Studies

Caching algorithms also referred as replacement algorithms are optimizing instructions that computer program or hardware-maintained structure can follow to manage a cache of information stored on computer. Cache size is usually limited, and if the cache is full, the computer or caching algorithm must decide which items to keep and which to discard to make room for new items. In this section we will consider several classical caching approaches.

First In First Out (FIFO) algorithm is simply removing the topmost items in caching list if the space needed. While caching new items, we add them to the bottom of a caching list.

Least Recently Used (LRU) algorithm discards the least recently used cached content. In order to discard the least recently used item, the cached items have to be continuously ordered according to their requested time. As the cached item is requested, consequently it should be moved to the bottom of a caching list. While caching new content and if there is no more space left for caching it, we have to start removing the least recently requested items or remove the top nodes of caching list, as it frees the available space for newly requested content.

LRU-min algorithm is removing the least recently used item whose size is larger or equal than new item to be cached. So instead of deleting several items from bottom of caching list, it is enough to find a cached item with the same size or more than desired space and has not been requested for long time. If there is no item with the same size or larger than newly cacheable item, we search for half of the size of an item to be newly cached. This process should be continued until the desired space for new item will be freed [2, 3, 4].

The topic of caching data with related content is previously researched by Michael Rabinovich et al [5] and he calls this method as **prefetching**. Prefetching refers to performing work in anticipation of future needs. The idea of prefetching Web pages has surely occurred to many as they used their browsers. It often takes too long to load and display a requested object; by the same token, several seconds usually elapse between consecutive requests by the same user. It is natural to wonder if the time between two requests could be used to anticipate and prefetch the second Web object so that it could be displayed with little or no delay. The goal of prefetching is to display Web objects on the user's screen faster than if prefetching were not employed and the objects were demand-fetched, that is, downloaded after the user requested them. Prefetching mechanisms are user-transparent, meaning that prefetching takes place without the user being involved or even aware of it. User-transparent prefetching is probably the only practical approach because of the highly dynamic and wide-ranging nature of many browsing sessions; usually, the user is unable to predict the URLs of objects to be visited, except possibly for the top-level object of a Web site. A transparent prefetcher is necessarily speculative, meaning that the prefetching system makes guesses about a user's future object references. For example, a prefetcher could infer future user behavior from past references to the same or similar objects made either by the same user or many users. Another source of information for the prefetcher is the content of the Web object that is currently viewed by the user. For example, hyper-links in an HTML object are candidates for prefetching since a user might click on them.

Any speculative prefetcher will make some wrong guesses and, therefore, will make more requests than a nonprefetching system that is presented with the same stream of user requests. The extra requests contribute to the two costs of prefetching: the extra load placed on origin servers and the extra network bandwidth consumed. It is important to quantify the costs because they can lead to worse performance for the prefetching client, other clients, or both. Besides evaluating the costs through obvious measures such as total extra requests and total extra bytes, some studies attempt to be more precise, evaluating how prefetching affects the burstiness of requests and what portions of the network become bottlenecks because of the extra bandwidth demanded by prefetching [6].

Therefore the terms *precision* and *recall* are used to refer to how often the users address prefetched data. Precision is the percent of prefetched objects that are subsequently requested. It is a measure of the accuracy of the prefetching algorithm. Recall is the percent of client requests that were prefetched. It is a measure of the usefulness of prefetching: prefetching would not be worth much, even if highly precise, if few objects were prefetched (that is, if prefetching had low recall). We call these metrics abstract because they reflect only the algorithmic aspect of prefetching, that is, the quality of prediction. In practice, prefetching is not purely an algorithmic problem, because even if an algorithm correctly predicts a future access, prefetching of the predicted object would be successful only if the object can be retrieved before the user actually requests it.

There is substantial danger in prefetching. It is inherently difficult to predict the future actions of a user who does not provide any special information to

the prefetcher, and incorrect guesses impose extra load on shared facilities. Therefore, it is valuable to know how much possible advantage prefetching can deliver and how accurate prefetching must be in order to succeed. An important study by Kroeger et al. [7] establishes bounds on the latency reduction achievable by prefetching into a shared proxy cache. These are not mathematical bounds, but rather the results of simulations applied to substantial traces (approximately 24.6 million requests) under idealized conditions. Their most notable result is that, even employing an unlimited cache, having a prefetch algorithm that knows the future, allowing up to 1Mbps of bandwidth for prefetching, and assuming that a major portion of end-to-end latency (77 to 88 percent) is incurred between server and proxy rather than proxy and client, prefetching into the shared intermediate proxy can reduce the total latency by no more than 60 percent. The primary reason for such limited latency reduction under such favorable conditions is the high number of uncacheable objects that are not cached or prefetched in the study.

3. Caching with Related Content

In previous section we considered various replacement algorithms from theory. We have also discussed about the concept of prefetching. This section will describe the new approach for caching web content. The main difference between prefetching and out approach is that we deal the requested data and its related content as a whole. So we can apply any replacement algorithm to cached data and at the same time to its related content. Similarly, in our method we separate the notion of data in to two classes. One is the data type representing its own content and second is the data type standing for the reference to the first data type and including necessary information such as file name and its size. This approach will help us to share the same data by many complex objects including requested data and its related content. It will prevent the caching server requesting from origin server extra times for the same data which is cached already. Consequently it will reduce the burst of requests between origin server and caching server. We will apply our replacement algorithm only to referential data type, checking if the data it refers is also referred by another object. If it is not referred by another object, we will remove the actual data from our caching memory. Playing with referential data types is lot faster than replacing the actual data. Therefore the replacement algorithms could be used efficiently and improve the internal processing of caching server.

It is generally known that every web page includes hyperlinks to some other web resources. So in our method, we parse the HTML content of cacheable web page and cache the related resources along with requested web page. By predicting the possible items to be requested and caching them for users beforehand, we can increase the hit rate of cache servers and deliver the requested resources much faster. In this approach we do not go into deep to some predicting algorithm, so we just use the parsed links within requested web page. So if the web page is requested and it exists in cache server, we give to it some priority not to be removed by using some replacement algorithm. While deleting the cached web content, we also delete its related data unless it has reference with other pages. If it has a reference with other cached objects, it will not be removed from caching list. The philosophy behind replacement techniques might be inherited from any algorithm mentioned above, but the concept of data to be cached is fundamentally changed. So the web page is now bound to its related web content and cached as single package.

From the concept above, we mentioned that one compound item of some cached web page can be referenced by other web pages too. So instead of caching the same data and binding it into different packages, the same data is shared by different packages. Preventing the duplication of same data makes the caching memory used much efficiently. In order to implement this technique we have to introduce two types of web data. One is actual data which will be stored in caching memory and the other one is referential data which references the actual data in caching list. So we have to maintain two lists for both types of data. We keep referential data in referential caching list and the actual data will be stored in real caching list. Replacement algorithm will use referential caching list in order to find what data to delete and the actual data from real caching list will be deleted.

In caching the web objects along with their related content, we consider the web object and its related content as one encapsulated object. This object includes the list of image and text references. Image and text references are referential data types which have to be derived from actual caching list. Moreover encapsulating object includes the main text reference object referring to the requested web page itself. So the referential caching list will store the encapsulated objects including references to actual caching list.

The related content for web page could be found through the use of special parser which searches for hyperlinks. The caching simulator implements the “compile” function which compiles the web page and its related content into one encapsulated object. It has

to parse the content of requested web page and compile the list of related web resources. Parser has to identify the reasonable links and report them. The implementation of this kind of parser is somewhat advanced topic and has to be considered in further researches, so the parser has to implement some sophisticated prediction algorithm anticipating the users next request and marking it as related content. The idea discussed above is summarized into Figures 1 and 2. The Figure 1 shows the regular caching method without prefetching the next user request. The Figure 2 shows how the proposed caching with related content works.

The algorithm of caching along with related content is initiated by the client's request. After receiving the request from client, cache server searches the requested page inside actual caching list. If it finds the requested web page, it derives the actual content from real caching list and returns it to the client. Meanwhile it applies some caching algorithm to referential list, not to actual caching list. If it cannot find the request page from actual caching list, it sends new request to origin server and starts compiling new package of related content for requested page. Whenever compiling new package, the cache server checks if some related resources exist in actual caching list. If they exist, compiler simply makes reference to them and does not request it from web server. If they do not exist within real caching server, they will be downloaded from web server.

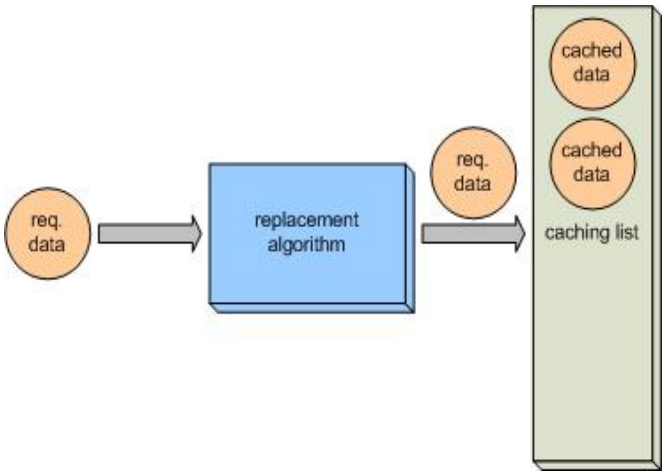

Figure 1. Ordinary caching method

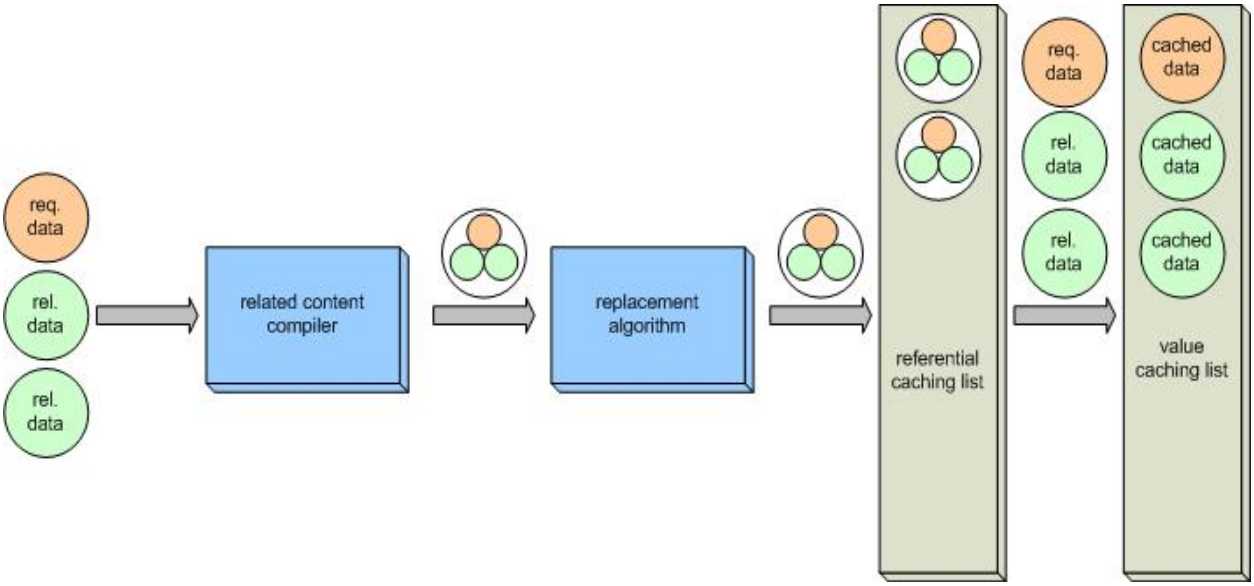

Figure 2. Caching with Related Content

While allocating them to real caching server and creating references for them, cache server will check for available memory. If there is no available memory, it will continually remove encapsulated packages with related content according to some replacement algorithm from referential caching list as well as removing them from real caching server until the available memory reaches desired size. After the needed space freed for new batch of related web resources, it will be placed to real caching server. The related web resources will be placed to real caching server separately while to referential caching server as one encapsulated object.

4. Conceptual Design of the Simulator

The design of simulator which tests caching with related content is illustrated using UML diagrams. We can use UML Class Diagram in order to represent the static relationships among classes. The classes are the definition of objects which we are going to use in our object-oriented application. Also in UML Class Diagram we can see that some classes inherit attributes and methods from their parent class. It is a very important feature of object-oriented paradigm and it is used very intensively in this simulator. We also used object composition in order to delegate the functions in run-time.

On the other hand, in order to describe the dynamic relationship of objects, we use UML Process Flow Diagram. The process flow diagram shows the actual interactions between different class objects and indicates the order and time frame of each action. The simulator implements three caching algorithms such as FIFO, LRU and LRU-min. So we can compare the performance of classic caching algorithms versus same algorithms but if cached with related content. We will present a Process Flow Diagram for both ordinary caching and caching with related content. To avoid presenting the same thing with different replacement algorithms, we chose the simplest one – FIFO, as a base algorithm for replacement.

4.1 Class Diagram

In Figure 3, we can see the Class Diagram illustrated. It is worthwhile to start understanding this class diagram, from core class named Web Data. This class represents any type of data which is managed by cache server. This type is abstract and cannot be instantiated. This class includes two main attributes, name and size, that are common to its subclasses. There are 2 subclasses inheriting from Web Data class, one is ValData which stands for value data and RefData which represents the reference data type. Both subclasses are also abstract classes and cannot turn into objects. The ValData type includes some additional attributes such as Content and NoRef. The first attribute stands for the actual value of an object and the second attribute NoRef stores the number of references to this object. If it turns to 0, it has to be deleted from caching list. On the other hand, RefData type aggregates ValData and could perform several operations on its referencing object. For example increment the NoRef attribute of referred ValData object when the RefData object is created and do vice verse when it is removed. Going down, we can see 2 subclasses for ValData class, and 3 subclasses for RefData class. ValImg and ValText classes inherit from ValData class and include the same attributes which ValData holds. RefImg and RefText data types are successors of RefData type and also hold the same attributes and methods which parent class possesses. These two subclasses are the fundamental elements of RefData class. Additionally we have another subclass of RefData class named as RelatedContent which represents the composition of RefImg and RefText classes. This design technique is inherited from Composite pattern which will be discussed more deeply later in this section. The RelatedContent class includes CompileRC method which is used to fill the RC_List attribute of the class. This method involves web server and value caching list as an argument. It also takes the maximum number of related files to compile as a related content. In our simulator, the CompileRC method randomly assigns the relationship between requested page and its related content. So it uses the getAny method of Web Server object. The getAny method of Web Server object sends random web data to an object that invoked it.

There is another type named as Replacement Algorithm illustrated in Class Diagram. It defines the parent class for all replacement algorithms. Its subclasses concretely implement the replacement algorithm. This design technique is thoroughly discussed in Strategy Pattern later in this section.

On the other side, we have CachingList abstract class which aggregates Web Data class with one-to-many relationship. So the CachingList class includes the object of LinkedList type which stores the data to be cached. The concrete subclasses of CachingList are the ValCachingList which stands for value caching list and RefCachingList which represents the list of referential data types. RefCachingList aggregates ValCachingList object and carries out several operations on it. For example, when the encapsulated packages with related content are removed from

RefCachingList, the ValCachingList should be updated. These two concrete caching lists have the same interface, so they will be treated by ReplacementAlgorithm class evenly. The CachingList class includes memorySize and usedMemory attributes in order to indicate the limitation of caching memory. Also it holds several methods like removeTop, remove, replaceToBottom, addToBottom, findByName and findLRUmin which are used by replacement algorithms.

The CacheServer is a central class which combines all composes all classes into one and provides interface for external classes like Client and WebServer. This design technique is the implementation of Façade Pattern which will be discussed in detail later in this section. The CacheServer includes all methods needed to external classes and handles the objects inside it. It includes attributes like hit count, miss count, request count, cache_with_RC and curRelList. The first 3 attributes are the indicators of cache server's performance. The 4th attribute defines weather to cache with related content or not. And the last attribute is used by Client component while making decision on choosing related content. The curRelList stands for the list of hyperlinks within opened page. So the user might click on one of these links and the pre-cached data will be returned to the user.

The Client and Web Server classes stand for Client and Web Server components described in Module Architecture section. Both of these classes share the SitesBase singleton object which keeps the domain of web data. SitesBase class stores the web objects that are viable to be requested through this simulator. At the beginning of the program, the SitesBase object is initialized and the domain of objects to be used is defined while it gets instantiated.

One of the robustness presented in design of a simulator is the use of Design Patterns described by GoF [1]. We will discuss the used Design Patterns in deep and explain how they improve the reusability and scalability of our design. So we have used 5 Design Patterns such as Strategy, Composite, Façade, Proxy and Flyweight very effectively in order to build pattern language working toward to solve our problem.

Strategy Pattern: While making the cache server capable of choosing either caching method, we used Strategy pattern to separate the basic algorithm for replacement of objects and implemented it nicely so it does not depend on the other parts of the program and could be extended easily. We have defined "Replacement Algorithms" abstract class which provides the abstraction of concrete replacement algorithm. So the simulator application can choose dynamically which replacement algorithm it wants to use. We can see three concrete replacement algorithms: FIFO, LRU and LRU-min underneath the "Replacement Algorithms" abstract class. The strategy pattern perfectly fits this situation and solves the problem of dynamically choosing different algorithms during the run-time very expertly.

Composite Pattern: This pattern is used to represent the "Related Content" class which stands for the composition of other sibling classes with the same type as its siblings. It is the exactly the same situation which is described in intent part of Composite pattern. The related content is a batch of web objects along with the requested page. This batch contains referential data derived from cached data. So the batch is composed of referential objects and stored as a one with the same interface as others. This is the core idea of this paper, to store the requested pages with their related content as a one object and apply various replacement algorithms on that objects. Basically we do our replacement manipulations with referential data type which references the actual data. So by working with collection of referential data types inside the related content batch, we do not have to reorder the position of actual data in the caching list. While deleting the related content batch, all referencing components inside it will check weather the data they are referencing is not used by another object. If not, they will delete the data from actual cache and finally discard themselves. Loosing its components, related content batch is also removed from referential caching list.

Façade Pattern: It could be seen from the class diagram that two different objects such as client and web server are communicating with the caching subsystem through a single cache server object. The "CacheServer" object summarizes the use different classes behind it and provides an interface for other objects which interact with subsystem. The intent of Façade Pattern ideally fits to this situation where the objects with different semantics should communicate through single object providing an interface with the use of the other objects behind it. So the objects out of the semantic should use only the available functions of Façade object. It eases the implementation and gives much simpler design. Of course on the other hand, it is absurd to see when client or web server objects have direct access to the internals of caching system. It gives some sort of transparency to the user, so he or she does not see how his request is processed.

Proxy Pattern: It is a perfect solution for defining a surrogate objects which represent the real data. We used the proxy pattern in order to make referential data types and referential lists. This data type actually refers to some other object, but includes some useful

information about the object that it refers. As we mentioned above, we use referential types for caching manipulations. Also the use of referential type could prevent the duplication of same data in caching list. It enables the objects to share the same data. Proxy pattern provides a placeholder for objects which share the same data. It is something close to C programming language's pointer object, but it is much more powerful than that. The one point is that the proxy object might include some useful information about the data it references. So we can find out which data we are using without digging down to the content of it. The pointers include only the physical memory address and nothing more. In proxy pattern, when we need the actual data we can get it upon demand.

Flyweight Pattern: The caching system uses very large number of fine-grained objects, in order to save the system resources we have to use Flyweight Pattern in this system. This pattern is not exactly reflected in class diagram, but the core idea of system is don't let the duplication of already cached data. In this situation, we are using referential data types which might have less weight than actual data objects. Anyway the number of objects will be reduced to some extent, as we prevent the duplication of same data in caching lists. In case, when different web pages use the same images or text, they could be logically shared by the creation of the referential data type to that data. So that referential data will goes into the batch of related content.

The used design patterns are not the only ones that could be implemented here. Many other design patterns could be used to make this design more reusable and extensible. For example, the ***Iterator Pattern*** is the perfect solution to traverse the "cached list" object uniformly. We used this pattern in terms of LinkedList object. This data container already includes this pattern, so we found not to present the Iterator pattern in our system. Generally speaking, we look forward to further improvements on this model and think that other patterns like ***Visitor*** and ***Command*** could be successfully used in our design. For example, the command pattern would be useful to encapsulate the requests of client into an object, while visitor pattern could be handy when cache server or referential cached list wants to do some operation inside the valued cached list.

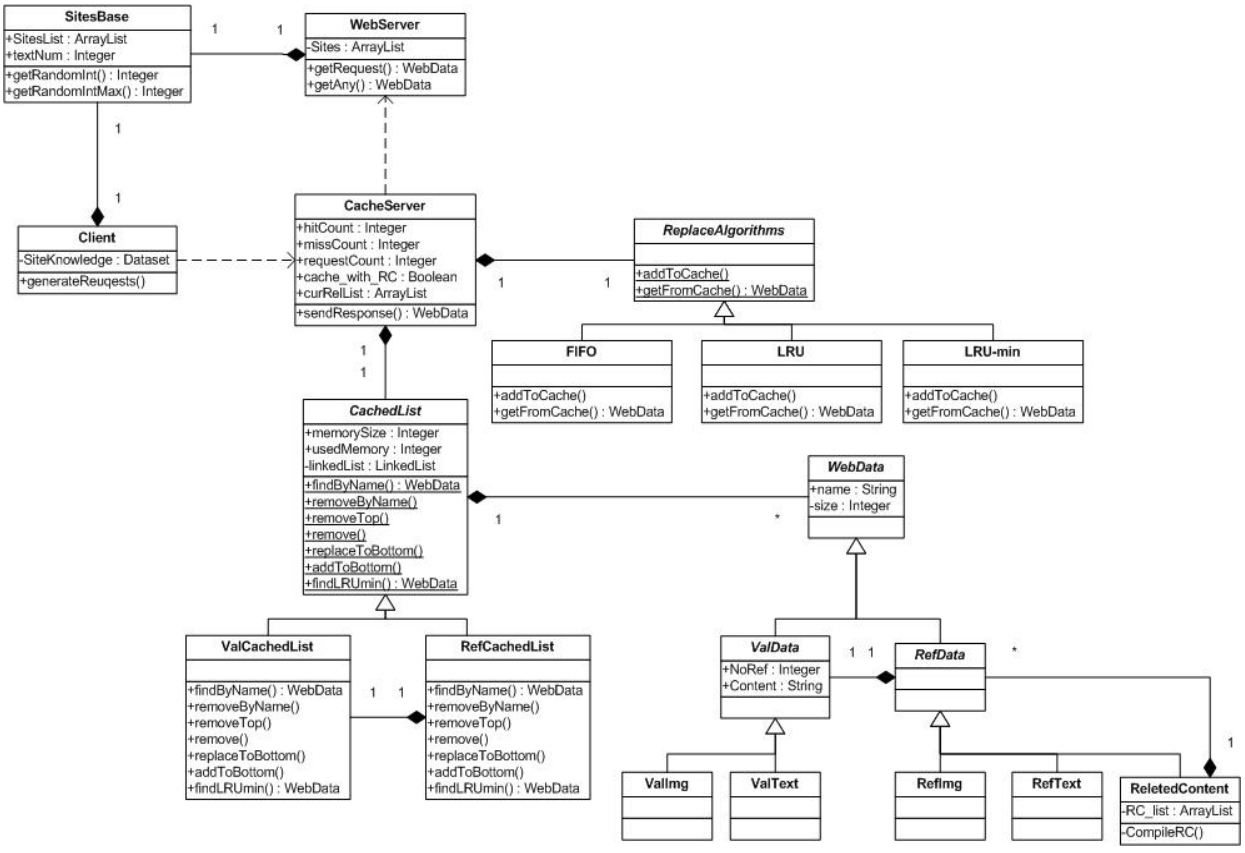

Figure 3: Class Diagram for Caching Simulator using FIFO, LRU and LRU with related content

4.2 Process Flow Diagram

In order to describe the dynamic interaction of objects we illustrate two Process Flow Diagrams. One is ordinary FIFO caching and the other one is FIFO caching with related content. Web used only one - FIFO caching algorithm in order to show the dynamics of our system. The other algorithms such as LRU and LRU-min might have same procedures, but they differ on how they replace the data.

In Figure 4, we can observe how objects communicate with each other when they cache the data with ordinary FIFO caching algorithm. The Client object invokes generateRequests method and sends the request to CacheServer object. Then CacheServer object asks the FIFO_Algorithm object to retrieve the requested data from ValCachedList. After invoking findByName method, ValCachedList object returns its response. If it is not null, the data will be passed to client. In case, if the response of ValCachedList object is null, then CacheServer requests the data from WebServer and invokes the addToCache method after receiving the requested data. If the memory for caching new data is available, the FIFO_Algorithm simply adds the new data to the bottom of the list. In case if the memory is not enough, the FIFO_Algorithm object recursively remove the topmost data from the ValCashedList object and check if enough size is freed. Meanwhile caching the data to cache memory, the cache server sends the requested data to client and starts to process next request.

In case of FIFO caching with related content, the cache server acts a bit different. The Figure 5 shows the Process Flow Diagram for FIFO caching with related content.

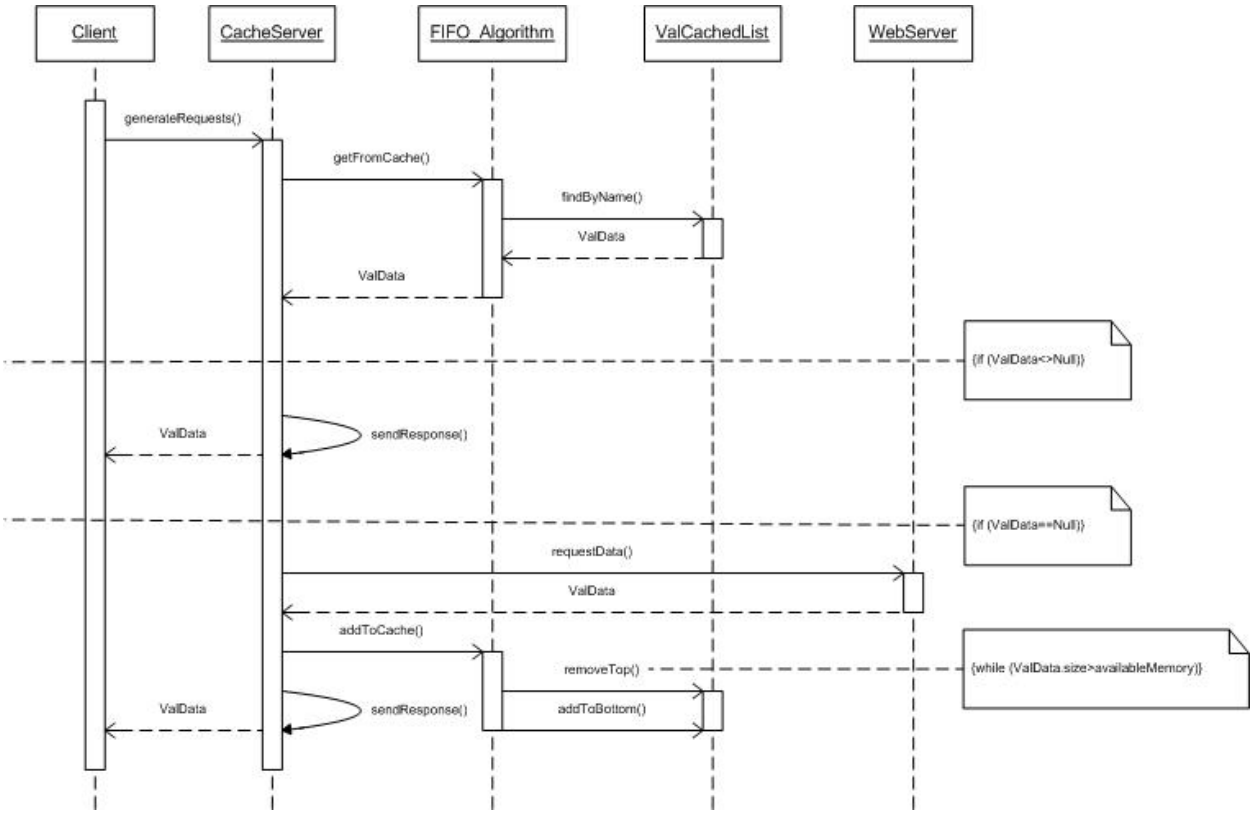

Figure 4: Process Flow Diagram for Caching Simulator using ordinary FIFO Caching Method

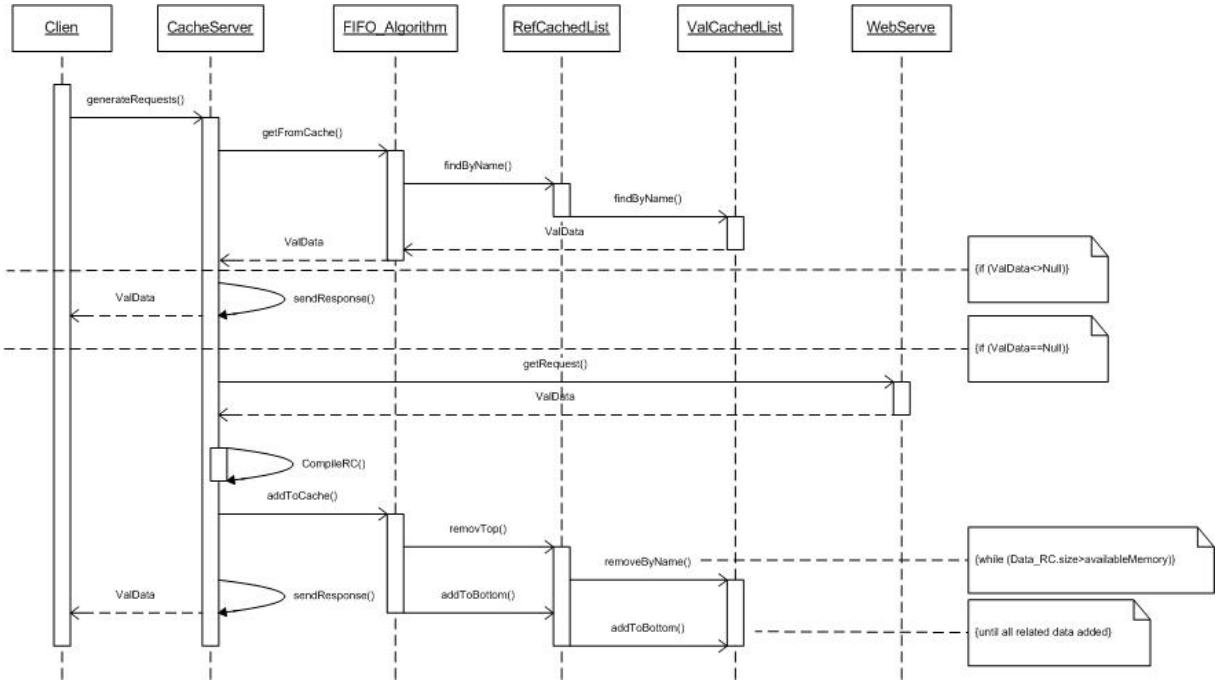

Figure 5: Process Flow Diagram for Caching Simulator using FIFO Caching with Related Content

In this method, the Client as usual generates the requests through invoking genrateRequests method. Then CacheServer object asks the FIFO_Algorithm object to retrieve the data from RefCachedList object. The RefCachedList object in turn forwards the request to ValCachedList object and it sends back the ValData object to CacheServer. If the sent object is not equal to null, it is forwarded to client. However if the object is null, then the CacheServer object retrieves requested object from web server along with its related content and compiles it into one object. Then it invokes addToCache method to RefCachedList object. The RefCachedList in turn checks weather there is a space for new data. If the space is enough, it adds the data to the bottom and passes it to ValCachedList which also adds the data to its bottom, but not as a whole, rather separately. If the space is not enough, then the FIFO_Algorithm removes the top of the list until sufficient space is freed. While removing referential types, the RefCachingList maintains that also the data in ValCachingList is being removed if it is not referenced by other objects. Thus the memory in cache will be freed for new data. Then the new data is added to a cache, as it is explained above. Finally the data will be sent to client, who will generate the next request then.

5. Overall Structure of the Simulator

In this section, we will discuss the overall architecture of our simulator which is going to simulate the caching methods mentioned above. We will illustrate first the overall module architecture of our caching simulator and describe how it works. Then we will go GUI of our simulator and provide some explanation on how it is implemented.

5.1 Module Architecture

In Figure 6, the module architecture of our simulator, we can see that it consists of three main actors. First one is client, which generates requests to cache server. The client is directly connected to cache server and sends its requests to cache server and receives requested data from it. Client can have two options before it starts generating requests. First option is the number of requests to generate. While launching our client, we have to set the number of requests to be generated, so it can generate that number of requests to cache server. Second option is the probability of choosing related content. Here we set with what likelihood client decides to access prefetched content. If the client only requests prefetched data, the hit rate for cache server will be nearly 100%. If client never decides to access related data, the hit rate will be almost the same with ordinary caching method.

Therefore it is crucial to know the probability of client's next choice. This option could be adjusted after analyzing the log files in cache servers and finding out how often users access hyperlinks within requested page.

The second component of the simulator's architecture is the Web Server which retrieves the requested data for cache server. So alike to Client component, it is directly linked to cache server. The main role of Web Server component is to find and send the requested data to cache sever. While initializing, the Web Server component generates different files with different sizes and stores them for user requests. The Web Server component can have 3 options. The first option is the average file size to generate, so it will generate files with the size among given average size. The second and third options are the number of text and image files to generate. The purpose of separating files into texts and images is done because image files cannot contain information about related sites. Therefore compiler can process only text files and collect the related content for them. Also the client can request text files directly, so the cache server will cache all images related to text file and the client can also view the images within requested text file.

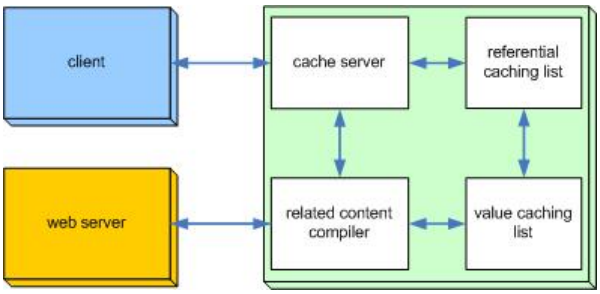

Figure 6. Module Architecture of a Simulator

The final important component is the cache server itself. It consists of 4 subcomponents. The first subcomponent – cache server is responsible for handling client requests and placing the data to caching list using some caching algorithm. The next component – compiler is responsible for compiling related content for requested data and putting all into a single encapsulated package. At the same time compiler checks if the related content for requested data exists in value caching list. In case if it exists, it will create a reference object to it, otherwise it will request that data from web server. Compiler subcomponent directly interacts with Web Server and downloads all related data. After completing the compilation, it passes the encapsulated package to cache server. Further cache server places the package to next important subcomponent - Referential Caching List. This subcomponent is responsible for keeping the reference objects and maintaining the value caching list according to operations underdone over it. The cache server applies all replacement operations over referential list, and consequently referential list maintains the Value Caching List which is the fourth subcomponent of cache server module. Value Caching List subcomponent is responsible for storing the actual data and providing it when it is requested by cache server. It also interacts with Compiler subcomponent for preventing the duplication while compiling related content.

All subcomponents of cache server work together in order to provide faster response to client, but involve heavy internal processing. Therefore this architecture involves high-speed servers with sufficient memory size.

5.2 User Interface Design

The GUI for simulator is built with very simple interface. As it was described in module architecture section, the interface consists of 3 main settings: Client settings, Web Server settings and Cache Server settings. In Client settings, we can define the number of requests to be generated and the probability of clicking related content. The maximum value the field for number of requests can take is 1000 and the probability defined in terms of percentage. Web Server settings include 3 options: average file size, number of text and number of image files. The maximum value for each control is 1000. Cache Server has 5 options: with Related Content check box which switches between ordinary caching and caching with related content, cache memory which could take minimum 1000 and maximum 5000, the choice of replacement algorithm and logging interval which refers to snapshotting the state of Cache Server every nth request.

The results pane shows a table with 5 fields. The first field is the number of requests generated, the second and third are the hit and miss counts for cache server. Fourth field stands for the total memory of cache server and 5^{th} field shows the how much memory is consumed at given snapshot. Using this results we can make our analysis upon the performance of each caching method and conclude which of them is better. The simulation could be started with the click of Start button, so the progress bar on status strip will indicate how much of work has been done while simulating. The screenshot for simulator is presented in Figure 7.

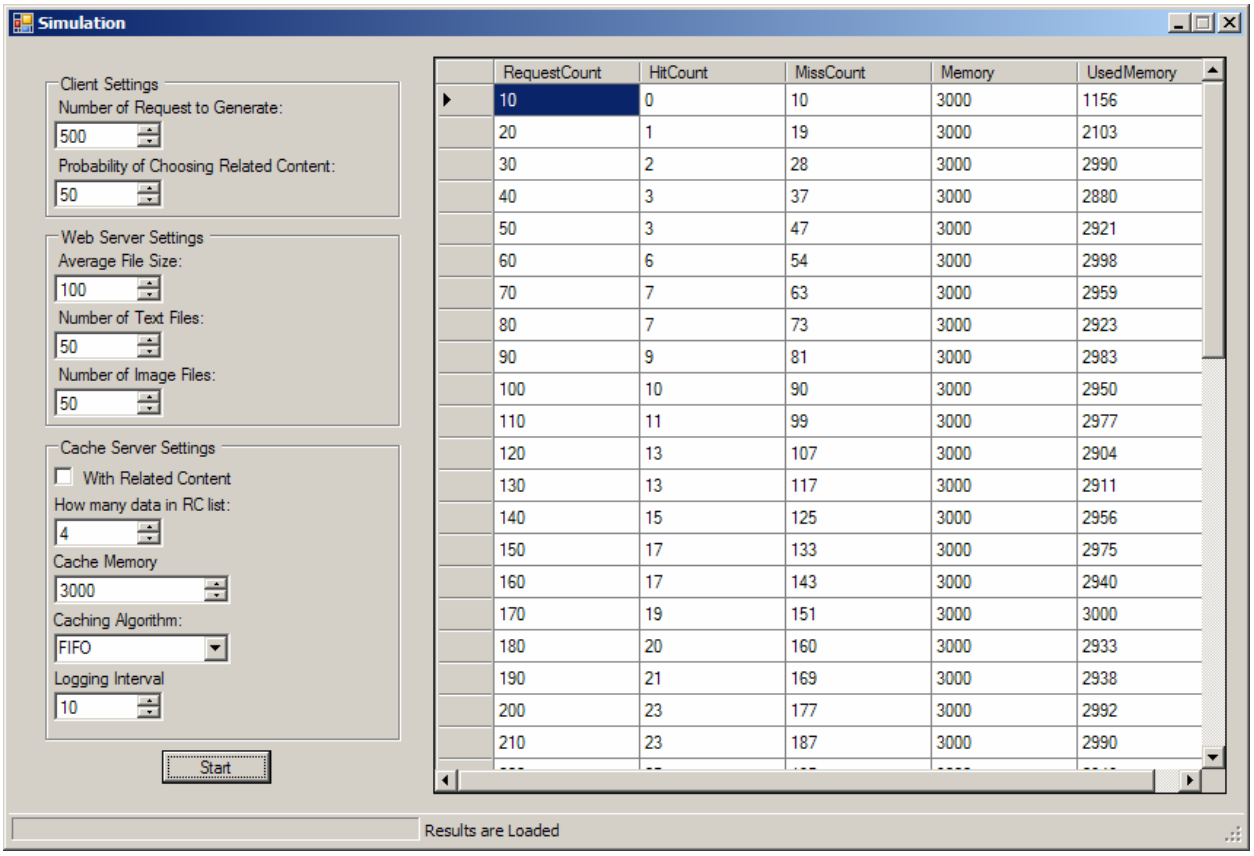

Figure 7. The Screenshot from the Simulator's Work Process

6. Experimental Results

In this section, we will discuss the results derived from the simulation of a simulator program. We will test each replacement algorithm separately and analyze its performance. At final stage we will collate all results into one graph and examine how efficient they are. As an initial setup for simulator we chose following settings:

- ***Caching Memory:*** *3000 Bytes*
- ***Web Content:*** *5000 – 25000 Bytes (10000 Bytes by default)*
- ***Caching Algorithms:*** *FIFO, LRU, LRU-min (FIFO by default)*
- ***Number of text files:*** *50*
- ***Number of image files:*** *50*
- ***Number of requests:*** *200*
- ***Average file size:*** *100 Bytes*
- ***Probability of choosing RC:*** *0-100% (50% by default)*

The following experiment shows us the comparative performance of all available caching algorithms with and without caching related content. The key performance metric used to evaluate the efficiency of caching method is the hit count. The hit count refers to the requests immediately handled by cache server without requesting them from web server. Therefore in our experiments, we use hit count as a major performance measurement.

6.1 Experimental Results with FIFO algorithm

In Figure 8, we can see how caching with the same FIFO algorithm depending on weather it is cached using related content or not could differ. It is obviously seen that caching with related content is taking advantage since the beginning of the simulation and growing dramatically whereas the ordinary FIFO caching method is showing almost zero performance in the beginning and getting increased gradually after 50 user requests. We can see that caching with related content handles the user requests without taking care about which request is it, so it fills the cache memory with related content very rapidly and utilizes it much more efficiently than ordinary FIFO method.

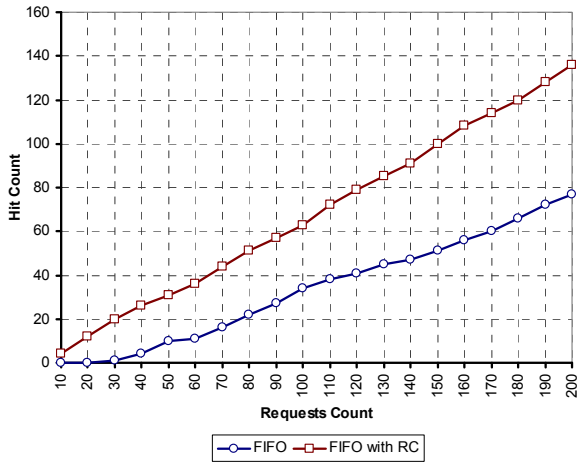

Figure 8. Performance of FIFO caching algorithms with and without Related Content

The Figure 9 shows the relationship between the cache memory and web content size ratio and hit count for FIFO caching method. It is obvious that if the domain of requestable objects increases, we more cache memory to store them all. But the cache memory is scarce and we have to use it efficiently. So in order to test how the cache memory is managed, we change the ration of cache memory size and web content size and see how the cache server acts if the web content is 7-8 times bigger than cache memory. We will check how caching algorithms manage the memory in this situation and also observe how it gets changed with the introduction of caching with related content.

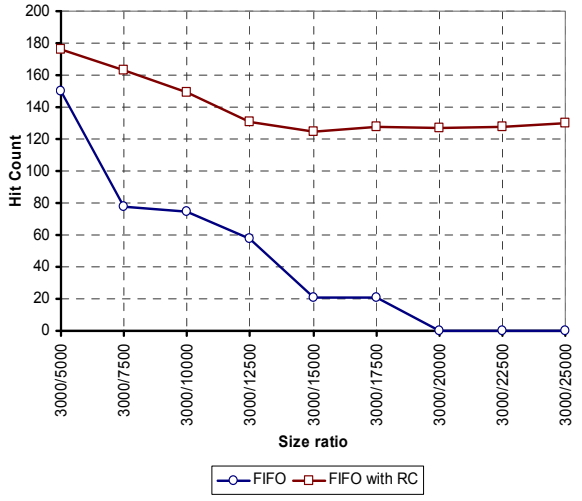

Figure 9. Dependency of FIFO caching with and without Related Content on the Size of Web Objects' Domain

The above graph is derived from the simulator with following setup, so the number of requests is 200 and the probability of client to choose related content is 50%. With this settings, the ratio of cache memory and web content is changed gradually and as it is obvious from graph, the ordinary FIFO caching start to fail with 0 cache hit when the ratio reaches 3000/20000. So it could be seen that FIFO caching will not perform well if the ratio is too small. On the other hand, FIFO caching with Related Content shows us stable result, despite how the ratio changes. At the first time, the performance is a bit decreasing, but starting from 3000/15000 ratio, it is leveling off and the performance is staying stable.

The Figure 10 shows us how hardly the caching with related content depends on user's next choice for request. If the user chooses the related content with 50% probability, the performance of cache server dramatically increases and the caching with related content will work for profit. Let's see what happens if the user's probability for choosing related content is different. The figure below shows this dependency with FIFO caching algorithm.

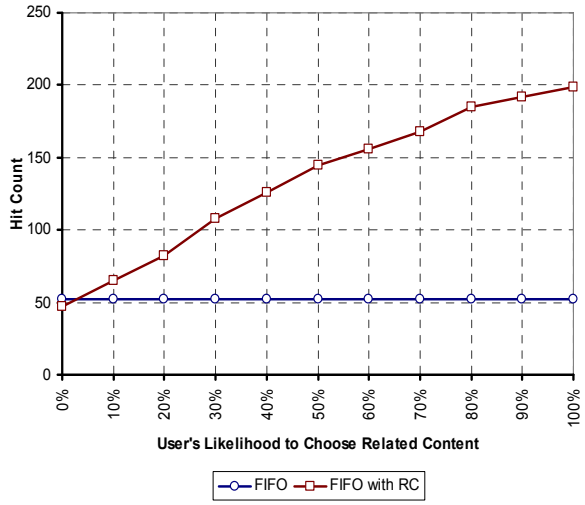

Figure 10. Dependency of FIFO caching with Related Content on user's probability of choosing related links

The initial setup for above graph is 3000/10000 ratio on cache memory and web content and 200 requests have been generated. The graph shows that when user has 0% probability to choose related content, the performance of caching will not be affected. If the users start clicking the links within opened web pages and in that way increase the probability of choosing related content, the caching performance will increase dramatically. For example, only 30% probability for choosing the related content could increase the hit count twice. Naturally, if the client only clicks the related links and nothing else, the hit count will be almost equal to request count.

6.2 Experimental Results with LRU algorithm

The Figure 10 shows the performance of LRU caching with and without Related Content. We have to note that caching with related content is performing similarly to a previous result. Since the beginning of the simulation, the LRU caching with related content is showing considerably better performance against ordinary caching. The ordinary LRU caching is acting similar to previous caching algorithm. So at the beginning phase, it is failing to handle user requests, but starting from 30th request it is starting to rise.

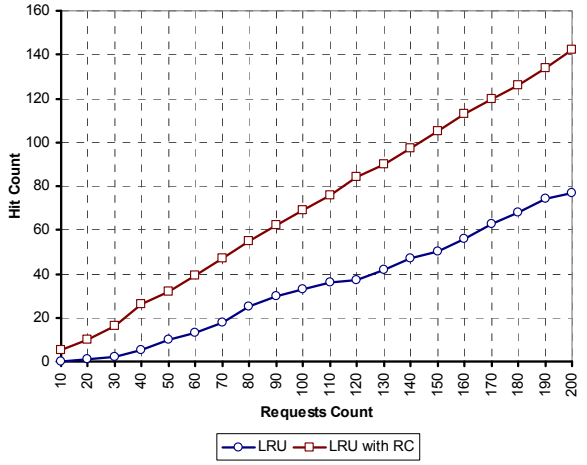

Figure 11. Performance of LRU caching algorithm with and without caching Related Content

The next Figure 12 shows us how the LRU caching algorithm reacts the when the domain of web objects become more big and the users can request more objects.

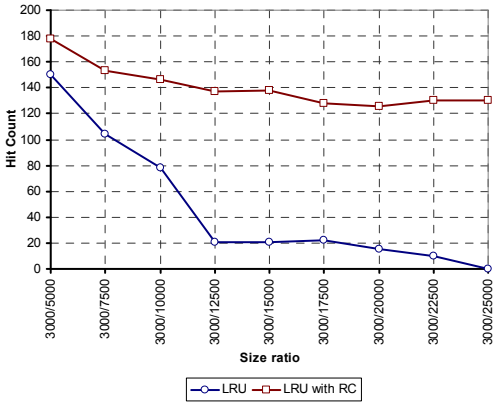

Figure 12. Dependency of LRU caching with and without Related Content on the Size of Web Objects' Domain

We can see in above graph that LRU caching with related content keeps more stable and gets leveled off and keeps the high performance even if the domain of web objects become almost 10 times larger. On the other hand, we can observe that the performance of LRU caching dramatically decreases if the domain of web objects gets larger and larger. It ends up with 0 hits per 200 requests when the domain is 10 times bigger than cache memory.

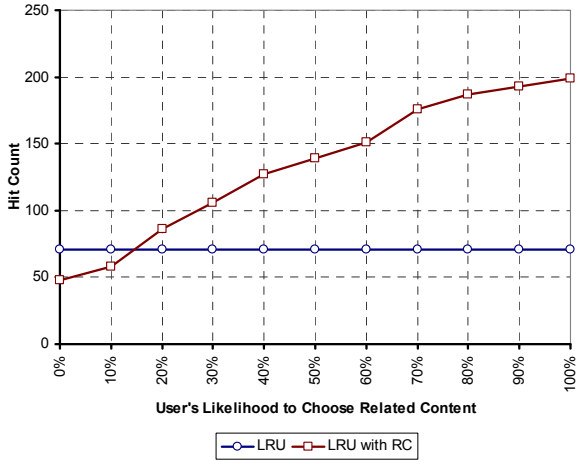

Figure 13. Dependency of LRU caching with Related Content on user's probability of choosing related links

The Figure 13 shows the tricky point of our approach were all the performance of caching method depends on the probability of client's decision to access related links. If the client never accesses the links within just opened web page, the results for LRU caching with related content will be a bit less than ordinary LRU caching method. But coming up to probability about 15% to click the related links, the performance chases up the ordinary caching method and continues growing if the probability increases. For example, it needs only 50% probability for choosing related links in order to improve the performance twice better.

6.3 Experimental Results with LRU-min algorithm

The Figure 14 below shows us the comparative performance of LRU-min caching algorithm with and without related content. We can see that the caching with related content ultimately takes advantage starting from the beginning of the simulation whereas the ordinary caching moving up very slowly. The ordinary LRU-min needs some time to get adapted with the user requests. On the other hand, the LRU-min caching with related content rapidly gets adapted to the requests by downloading possibly the next request of the client.

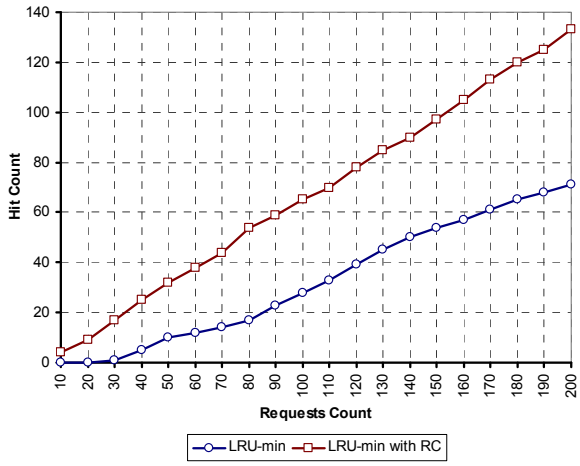

Figure 14. Performance of LRU-min caching algorithm with and without caching Related Content

The next Figure 15 shows us how the memory is efficiently used, if the domain of web objects is dynamic. Actually in real life examples, it is true. It could be seen that the standard LRU-min performs rather bad when the domain gets larger. For example it hits nearly zero requests if the size of domain reaches 22500 Bytes. On the other hand, the LRU-min caching with related content lowers a bit, but keeps stable after the domain size reaches 12500 Bytes. We have to note that the cache memory size is 3000 Bytes by default and it stays unchanged while the domain of web objects expands.

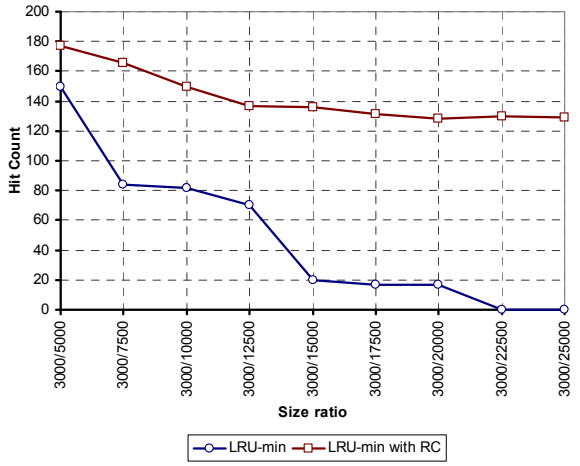

Figure 15. Dependency of LRU caching with and without Related Content on the Size of Web Objects' Domain

In the Figure 16, we will see how hardly the LRU-min caching with related content depends on the user's likelihood of choosing related content. It is seen from the graph that LRU-min caching with related content shows lower performance when the probability of user choosing related links is less than 20%. But if the probability increases, the caching with related content becomes an optimal solution and shows better performance.

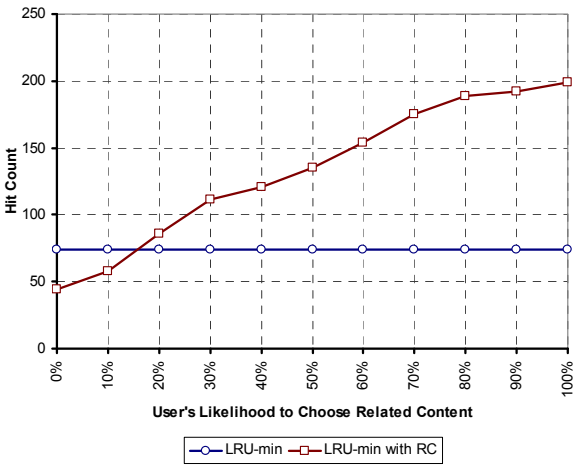

Figure 16. Dependency of LRU-min caching with Related Content on user's probability of choosing related links

It is obviously seen from the graphs above that despite which algorithm we use, the caching with related content is showing outstanding results. The optimal case is when the client's probability to choose related site is 50% and the size ration of cache server and web server is 3000/10000. We can see that standard algorithms act similarly and do not show any considerable advantage of one over another. Also after applying the caching with related content method, the results are improved evenly for each algorithm. So we can conclude that caching with related content could be used with any caching algorithm, and surely it will improve the performance of cache server in terms of cache hits. The Figure 17 summarizes the performance of each caching algorithm with or without related content.

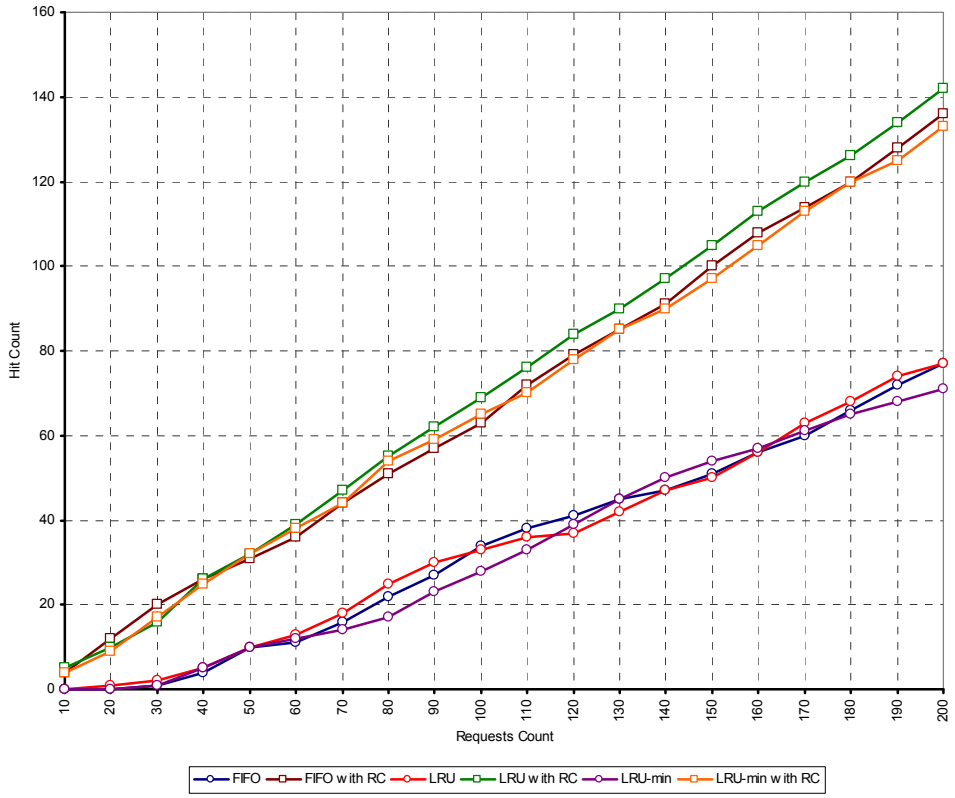

Figure 17. Hit counts for all available caching algorithms with and without Related Content

7. Conclusion

Our study focused on building the simulator using object-oriented paradigm and design patterns in order to implement new caching method. We discussed about the related researches and cited the prefetching topic. Then the new caching approach – caching with related content method has been introduced. We highlighted the originality of our idea from previous researches. Unlike other caching approach, our caching method treats requested data and its related content as one encapsulated object and applies to it some replacement algorithm. In design part of paper, first we talked about the overall architecture of the simulation. We defined the components of our system and clear explained the role of each. Then we turned to design patterns and UML diagrams that made our software easy to build, and much scalable and reusable. We used Strategy, Composite, Façade, Proxy and Flyweight design patterns in our design. We also proposed the use of other design patterns like Command and Visitor in future works. Afterwards, we launched the simulator and derived some results from it. We used those results to analyze the efficiency of proposed algorithm and proved that our caching method presents better performance and more reliable than other algorithms in case of low ratio between cache memory and web content. We showed how hardly our caching method sticks to the probability of user's next request.

The proposed algorithm is specifically designed for web caching, so it could be implemented in future proxy cache servers. There are still many issues that should be researched in proposed topic. For example, as it was mention above in this paper, the compilation of batch of related content is another issue for designing the system. The batch compiler component should not just find the related content according to the hyperlinks within requested page, but it should be also intelligent enough in order identify valid and mostly requested links while disregarding passively requested links. So there should be an intelligent module making decisions about weather to download related resource of some requested page to the cache according to how busy the link is. Generally speaking, there are lot more research topics on proposed method and we hope that we could reveal those issues and find optimal solutions for them in our further researches.

References

[1] Erich Gamma et al, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, USA, 2005

[2] Sarvar Abdullaev, Franz I.S. Ko, "An Object-oriented Design of a Simulator for Caching with Related Content using LRU", Advances in Information Sciences and Services, Vol 2, AICIT, S. Korea, November 2007, pp. 96-103

[3] Franz I.S. Ko, Sarvar Abdullaev,"RADS: Web Caching Algorithm with Size Heterogeneity of Web Objects", Advances in Information Sciences and Services, Vol 2, AICIT, S. Korea, November 2007, pp. 212-218

[4] Ko Il Seok, "ACASH: An Adaptive Web Caching method based on the Heterogeneity of Web Object and Reference Characteristics", Information Sciences(ISSN:0020-0255), ELSEVIER SCIENCE INC., Vol.176, Issue12., 1695-1711, 2006.6.22

[5] Michael Rabinovich, Oliver Spatscheck, Web Caching and Replication, Addison-Wesley, USA, 2003

[6] Crovella, M., and Barford, P. The network effects of prefetching. In Proceedings of INFOCOM, pp. 1232-1239, 1998

[7] Kroeger, T. M., Long, D. D. E., and Mogul, J. C.. Exploring the bounds of Web latency reduction from caching and prefetching. In Proceedings of the USENIX Symposium on Internet Technologies and Systems, pp. 13-22, 1997

[8] G. Barish, K. Obraczka, World Wide Web Caching: Trends and Algorithms. IEEE Communications, Internet Technology Series, May 2000.

[9] H. Bahn, S. Noh, S. L. Min, and K. Koh, "Efficient Replacement of Nonuniform Objects in Web Caches," IEEE Computer, Vol.35, No.6, pp.65-73, June 2002.

[10] L. Rizzo, L. Vicisano, "Replacement Polices for a Proxy Cache," IEEE/ACM Trans. Networking, vol.8, no.2, pp.158-170, 2000.

[11] S. Williams, M. Abrams, C. R. Standridge, G. Abhulla and E. A. Fox, "Removal Policies in Network Caches for World Wide Web Objects," Proc. 1996 ACM Sigcomm, pp.293-304, 1996.

[12] V. Almcida, A. Bestavros, M. Crovella, and A. Oliveira, "Characterizing Reference Locality in the WWW," In Proc. of the 4th Int'l Conf. on Parallel and Distributed Information Systems, 1996.

[13] J. C. Bolot and P. Hoschka, "Performance engineering of the World-Wide Web: Application to dimensioning and cache design," Proc. of the 5th Int'1 WWW Conf., 1996.

[14] M. Abrams, C. Standridge, G. Abdulla, S. Williams and E. Fox, "Caching Proxies: Limitations and Potentials," Proc. 4th Int'l World Wide Conf., 1995.

[15] N. Niclausse, Z. Liu, P. Nain, "A New and Efficient Caching Policy for the World Wide Web," Proc. Workshop on Internet Server Performance(WISP 98), pp.94-107, 1998.

[16] C. Aggarwal, J. Wolf and P. Yu, "Caching on the World Wide Web," IEEE Trans. Knowledge and Data Engineering, vol 11, no.1, pp.94-107, 1999.

[17] S. Sahu, P. Shenoy, D. Towsley, "Design Considerations for Integrated Proxy Servers," Proc. of IEEE NOSSDAV'99, pp.247-250, June, 1999.

[18] J. Wang, "A Survey of Web Caching Schemes for the Internet," ACM Computer Communication Review, 29, pp.36-46, October, 1999.

[19] E. Cohen and H. Kaplan, "Exploiting Regularities in Web Traffic Patterns for Cache Replacement," In Proceedings of the 31st Annual ACM Symposium on Theory of Computing, ACM, 1999.

[20] C. D. Murta, Virgilio A. F. Almeida, Jr. W. Meira, "Analyzing Performance of Partitioned Caches for the WWW," In Proceedings of the Third International WWW Caching Workshop, Manchester, England, June, 1998

[21] "Web Caching", http://www.mnot.net/cache_docs/ (last accessed: 11.07.07)

[22] "Web Caching: Making the Most of Your Internet Connection", http://www.web-cache.com (last accessed: 11.07.07)