A Colored Petri Net Simulation Model 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, LRU-min, LFU and SLRU. 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 items along with their related content. It is generally known that the use of Colored Petri Nets (CPNs) for modeling the simulation of new idea is becoming very popular throughout software development projects. Moreover it could be very handy in describing the overall logic of the system. The CPN Tools is one of the most robust software which supports all necessary tools and functions to build and run the simulation model for CPN. Therefore in this paper, we will use CPN Tools software in order to build a simulator for above mentioned innovation. Then we analyze the results derived from the simulation of the model.

1. Introduction

Nowadays while the Internet is spreading all over the world, it is natural that the access to 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 copies 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, LRU-min, LFU and SLRU which are commonly used for caching web content. Further we will focus on FIFO algorithm and apply it for newly proposed concept of caching related content. We will build a web caching simulator using CPN Tools. Through the use of CPN Model we will illustrate the design of a simulator for above proposed technique and compare it with classic caching approach. Generally speaking, the use of CPN in building cache simulator is a new approach for designing and understanding the logic behind web caching. So it will help us to understand the proposed concept much clearly.

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.

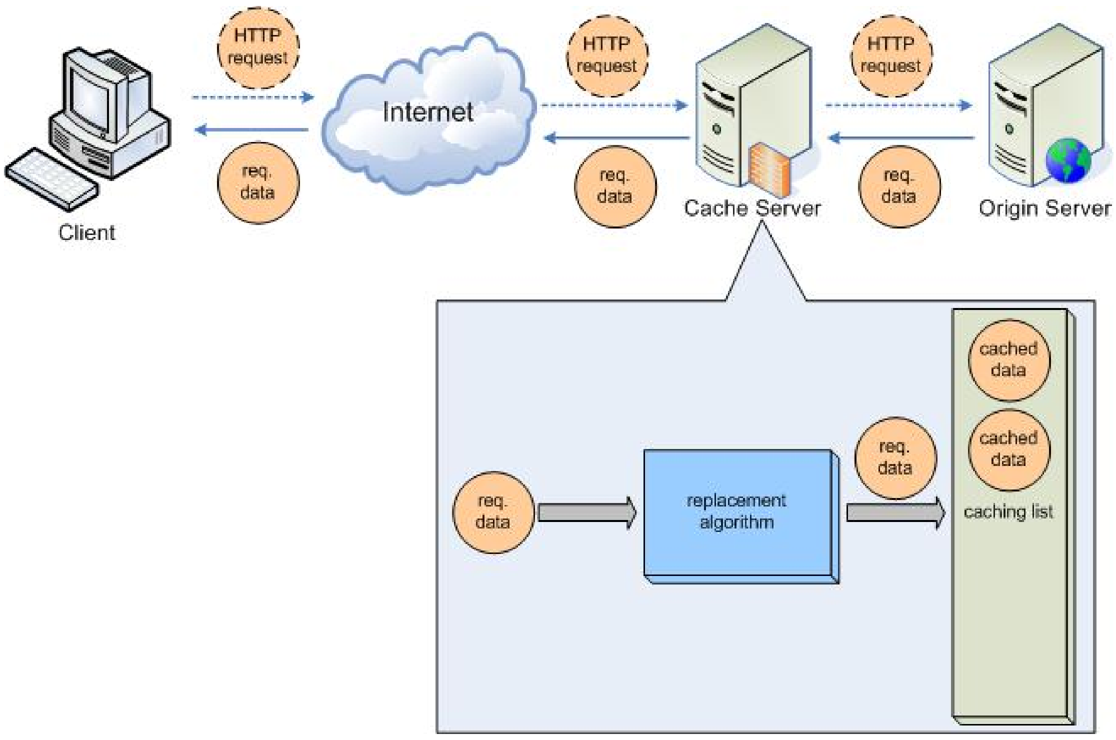

Figure 1. Ordinary caching method

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.

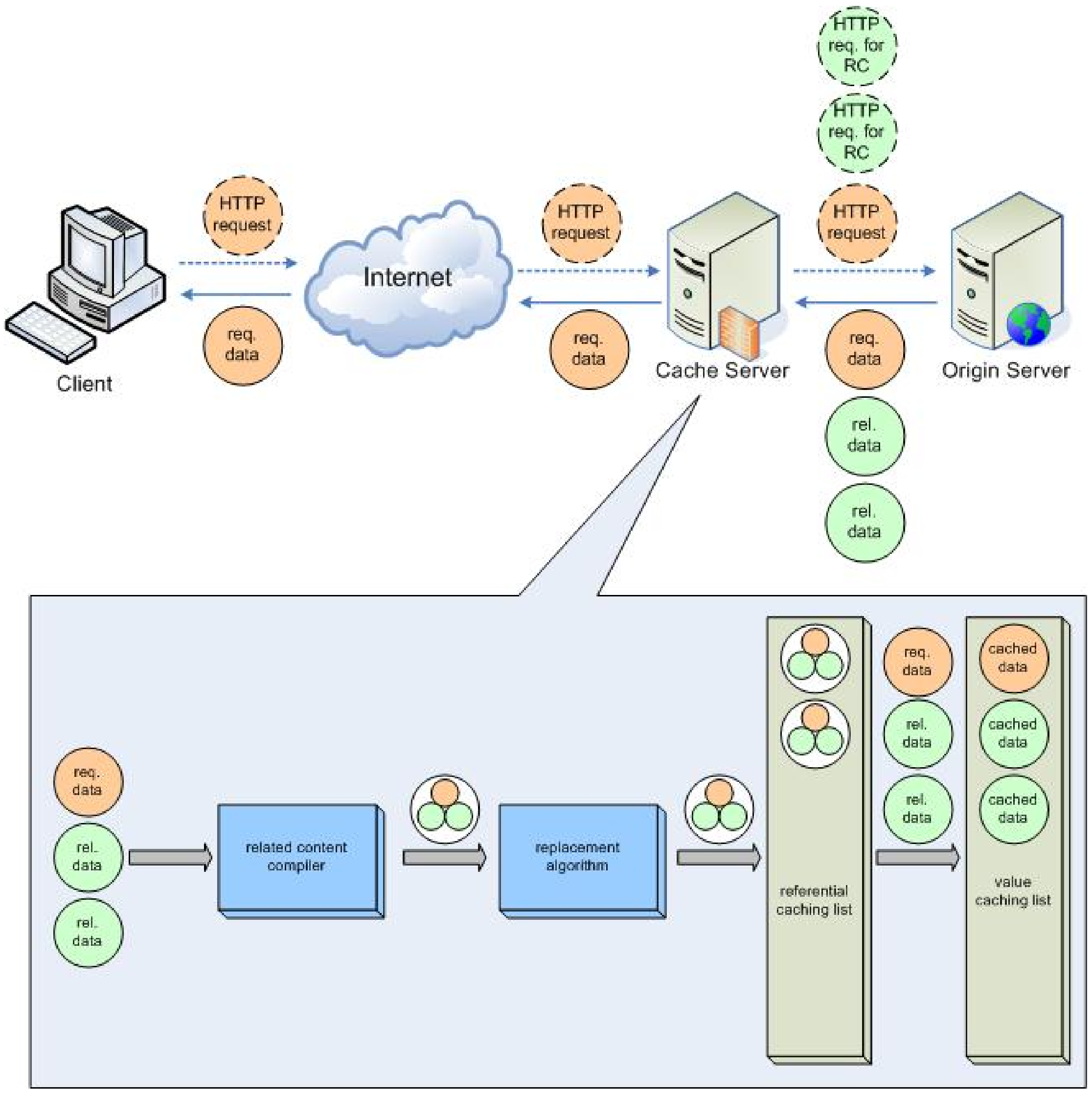

Figure 2. Caching with Related Content

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. CPN Model

We built two CPN models for simulating the web caching both with classical caching approach and with proposed method. For the sake of simplicity, we chose the easiest caching algorithm – FIFO. So we can compare two methods: original FIFO caching and FIFO caching with related content. (Tadao Murata, 1989)

4.1 CPN Model for FIFO Caching

In this model, we have three color set declarations, so it means that we deal with 3 types of objects. First is the actual web data which is represented as token with W color. Second is the list of web data, which refers to a caching list in this model. The list object is predefined in CPN ML Standard Library and stands for extendable container of objects. Thirdly, we have declared colorless token as E, which helps us to control some transitions.

The CPN Model described in Figure 3 stands for ordinary caching server which caches with the use of FIFO algorithm. There 5 basic actions within this model, or we can refer them as transitions. Firstly, we initiate the genReq transition which refers to the action of generating requests. In this model, genReq transition randomly generates requests for web objects within the domain of W color. While generating request, it adds E token to the ReqCnt place for the sake of counting the requests produced. Here E token does not have any real meaning but used for counting the requests. Similarly, the E token consumed from Switch place limits the genReq transition from continuously generating requests till the recently requested object processed and sent to the user. Finally the genReq transition inputs reqw token to reqData place. This place stores the request sent to cache server for processing.

After coming to reqData place, the reqw token has two directions. One case is when the getData transition is enabled and it can consume the token from reqData place. The getData transition is guarded with condition checking if the requested data held in caching list. In order to do so, it takes the cacheList token from CData place and checks the above mentioned condition with *mem cacheList reqw* expression. The mem command is predefined function for List structure in CPN Tools' standard library. If it gives us “true”, then the getData transition is enabled and the data could be passed to client requesting it. So the reqw object takes the dataSent state. On the other hand, getData transition put E token to Switch place enabling the genReq transition to generate next request.

In contrast to getData transition’s guard, we have reverse condition for addMiss transition which checks if the data is absent in caching list. So in case of the getData transition is not enabled, the addMiss transition will be surely active. This transition adds E token to Misses place which counts the number of missed data. If we know the total number of requests and misses, we can derive the hits through finding the difference between total requests and misses. Also the addMiss transition transfers the reqw object from reqData place to toCache place noting that this data has to be cached.

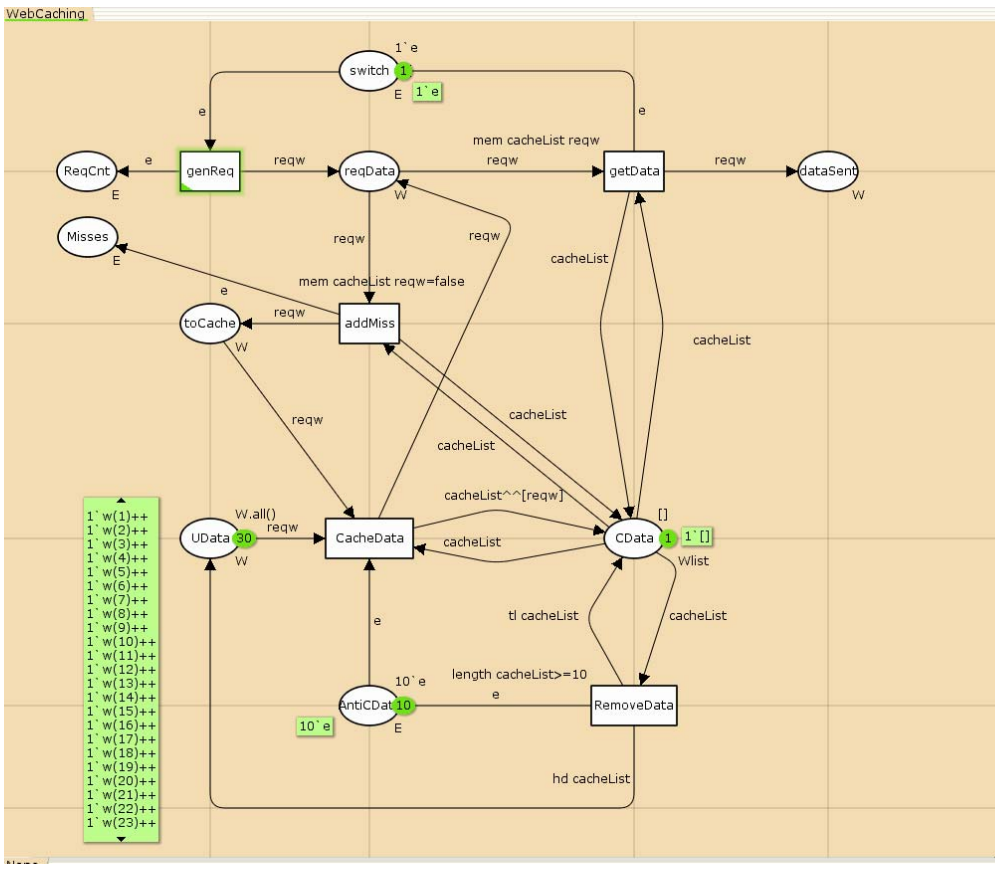

Figure 3. CPN Model for FIFO web caching

Further the cacheData transition will be enabled. This transition takes the requested data from UData place and appends it to caching list. UData place stands for data which is not cached yet. In order to append the requested data to caching list, cacheData transition takes the cacheList object from

CData place and through using *cacheList^^[reqw]* expression puts back it with appended newly cached data. In this manner, the caching list is filled with data. In real cases, the caching memory is restricted, and such resource limitation is very vital in caching. In order to represent the resource limitation, we used a technique called Anti-places. This technique refers to adding reverse arcs and an anti-place against some place in order to bound some transition such it can be fired only n times. As the CPN Tools software lack such property like defining the limitation for places, we walked around it by adding anti-place for CData place. We called it antiCData. So depending on the initial number of tokens in antiCData, the CachedData can be fired n times. If there is no tokens left in antiCData place, this represents the caching memory is full and some objects should be removed from caching list. For this purpose, we have RemoveData transition which is enabled in case if and only if the caching memory is full. In order to check this, we use guard for RemoveData transition with *length cacheList>=n* expression. It checks the length of cacheList object and if it is equal or more than designated number, the transition will be enabled. The RemoveData transition also takes the cacheList object from CData place and after beheading it, returns back its tail which refers to the rest of elements of the list except first. The head of the cacheList is pushed back to UData place which stores the web objects which are not cached. This is the essential concept for FIFO, so we take the old data from top and append the new one to the bottom. Moreover RemoveData transition gives one token to antiCData enabling the cacheData transition to be fired one more.

After the data is successfully cached to caching list, the getData transition will be enabled, as its guard gives true result. Through firing getData, we send the requested web object to the client and wait for next request.

4.2 CPN Model for FIFO Caching with Related Content

In this model, we make some changes to previously discussed CPN Model. Firstly we will describe the generation of requests. It is a bit sophisticated then just randomly selecting the web objects from domain as a request. In this module, we use some stochastic techniques to define the probability of the client’s decision. Normally, the users after opening the web page, they click on the links located within that link. We assume that the user has 50% likelihood for clicking the hyperlink within opened page.

The 50% likelihood is when the user accesses the other web page which may be not within the opened web page. This mechanism is implemented in CPN Tools in such way as follows.

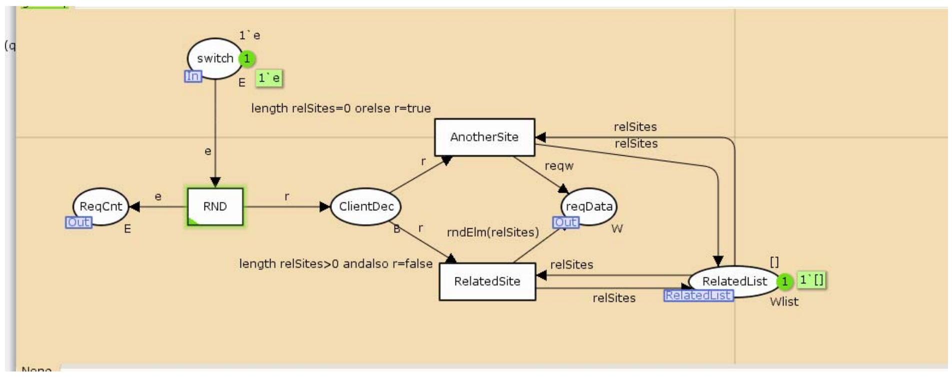

Figure 4. CPN Model for Generation of Requests

The above shown Figure 4 is a substitution transition for genReq transition mentioned in Figure 3, a previous CPN model for ordinary FIFO caching. It has two output ports and one input port. One output port pushes token to ReqCnt place for counting the requests generated. Another output port gives us the requested web object. The input port takes the E token from Switch place. The RND transition generates a Boolean value, either true or false. So the probability of getting true or false value is 50 to 50. Then two transitions check the value inputted to ClientDec place. ClientDec place refers to the client’s decision and if it is equal to true, the client will access the page which is not related to currently opened one. If the client’s decision is not to open the related site, the AnotherSite transition will randomly generate a request from the domain of web objects and passes it as request to cache server. If it is false, then client’s decision is to click the hyperlink within currently opened web page. Also to smooth down the case when the client’s decision is to open related link when the web browser is empty, we check if the RelatedList is not empty. The important thing to note is when the client decides to related site, he knows what sites are related from the place which keeps the related links of currently browsed page. And this place is exactly the RelatedList place. The client randomly chooses any link from this place and sends that as a request. We will discuss more about RelatedList fusion place later in this section.

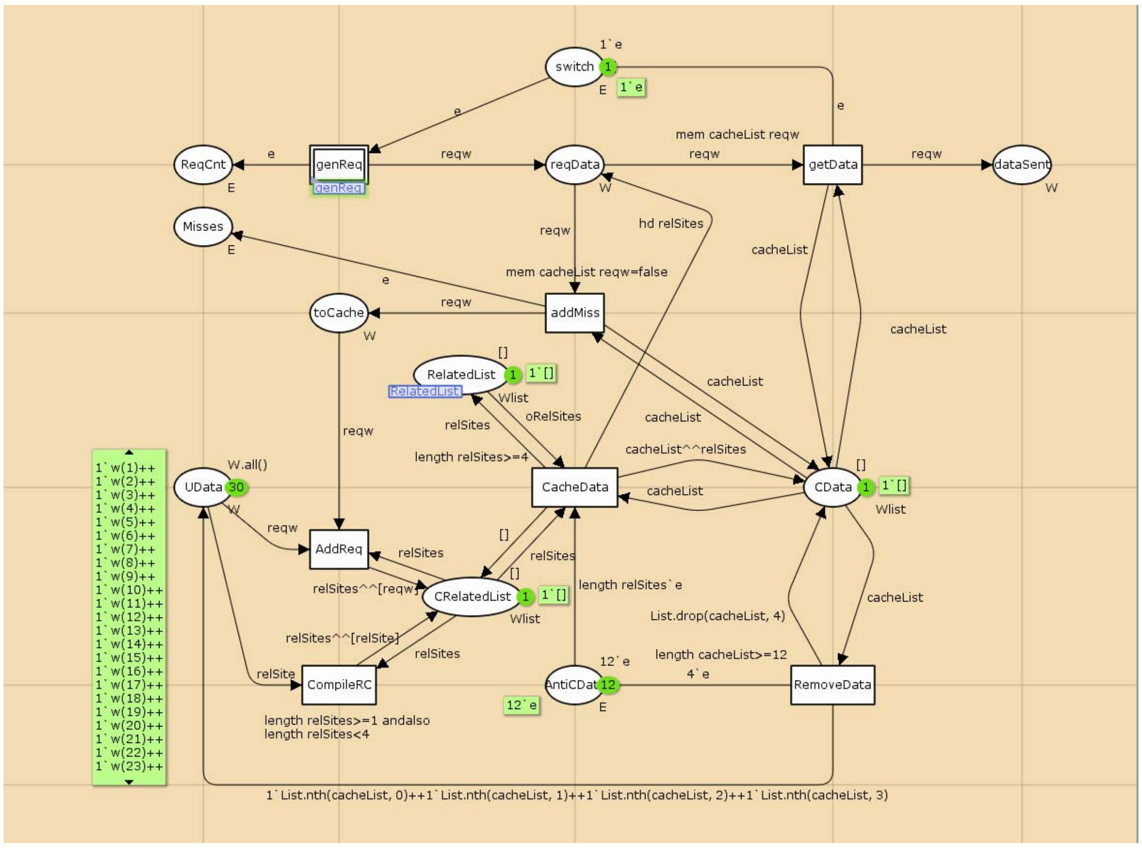

Figure 5. CPN Model for FIFO caching with Related Content

The overall model for FIFO caching with related content described in Figure 5 also takes some changes. We have a new transition called as CompileRC, which defines the related links for requested web object. In our case, for the sake of simplicity, this transition randomly assigns the related link to requested object. We also strictly defined that we find 3 related links for each requested object. Therefore after adding the requested object to CRelatedList place as a head of the list through firing the AddReq transition, we enable the CompileRC transition to retrieve 3 related links for requested object. After retrieving the related objects, they are appended to the relSites object stored in CRelatedList place. The fulfillment of CRelatedList place enables the cacheData transition and then it can be fired. The cacheData transition appends the relSites list to caching list and also returns back the head of relSites list to reqData place. As we mentioned previously, the head of relSites list is requested object. So now the requested object is placed reqData and CData and consequently it enables the getData transition.

Another important effect of cacheData transition is that it transfers the relSites object to RelatedList fusion place. We discussed about this place above while generating requests in case if the client decides to click the hyperlink within the opened web site. So the RelatedList place stands for a place which stores a list object which keeps the related content of currently opened web site. So harnessing this information kept in RelatedList place, we will be able to generate meaningful requests based on related content of currently browsed web page.

As we are adding data in the form of batch and we have already predefined that the batch consists of 4 objects where 1st is the requested object and the other 3 are the related data. We have mentioned about antiCData place in our previous example. In this model, every time when we append relSites list to caching list, we take 4 tokens from antiCData. It means that we are using more memory than in our previous example. As measurement of memory, we are counting the number of tokens stored in caching list, so respectively if the batch consists of 4 objects, we have to take 4 tokens from antiCData place. On the other hand, while removing the data from caching list, we take 4 topmost objects from caching list and push them back to UData. We also input 4 E tokens to antiCData place, in order to enable cacheData transition.

5. Initial Setup and Experimental Results

In this section we will compare the results derived from simulation of both models and prove the efficiency of proposed caching method. As an initial set up we declare 30 web objects as a domain for W token. So these tokens will be requested by client and consequently cached in caching list. The limitation for caching is 10 tokens for ordinary FIFO caching model. So the caching ratio is one third to available data. The summary for initial setup is shown below:

Total data: 30 web objects

Max. data for caching: 10 web objects

Caching method: FIFO

Number of requests generated: 200

Thus after generating 200 requests for a cache server using ordinary FIFO method, we have got the following picture. Since the beginning of simulation, the caching server showed poor results in terms of cache hits. As the number of requests grew, the difference between cache misses and hits became broader and broader. So observing the Figure 6, we can see how the cache hits getting leveled, while the cache misses grow. We can conclude that ordinary FIFO approach is not enough to be a good solution for web caching, as it shows low performance.

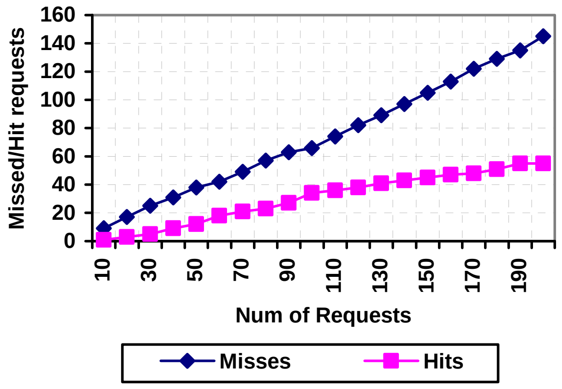

Figure 6. Simulation results for standard FIFO caching

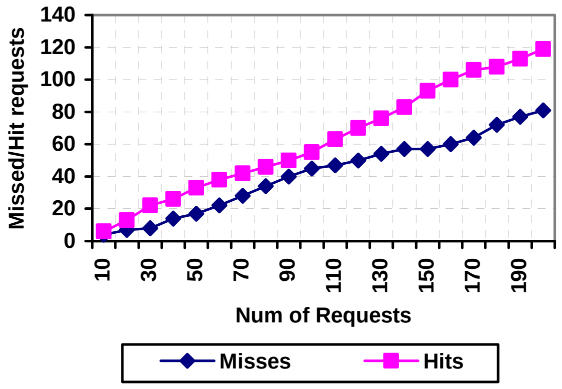

Figure 7. Simulation results for FIFO caching with Related Content

Above we can see if our approach enhances the performance of FIFO caching method. We can observe from Figure 7 that from the initiation of the simulation the cache hits are gaining large spread against cache misses. So the difference between cache hits and misses is gradually growing. This means that the efficiency of our proposed method is continuously increasing. So we have seen that the use of FIFO caching with related content makes dramatic improvements to the performance of caching server. So we can conclude that our method paid off our efforts.

6. Conclusion and Future Perspectives

Our study focused on building the CPN simulation model for web caching with related content using CPN Tools software and testing the efficiency of proposed caching algorithm. We derived the results from simulation and proved that the proposed approach will dramatically improve the caching performance of standard caching algorithms. Of course, the proposed algorithm might require more CPU power for internal processing, but in comparison with the time spent for downloading the objects from internet and delivering it to the client, it might be very insignificant period. On the other hand, the use of CPN modeling techniques could reveal the logic of our system in much comprehensive way. Therefore we used this technique in order to describe the application's core logic.

The proposed method 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 on proposed topic. For example, as it was mention above in this paper, the "Compiler" component of simulator has to be revised and improved. Despite finding the links within web text, the "Compiler" should be intelligent enough in order identify valid links and add them to the batch of related resources. Moreover there should be an intelligent module making decision about weather to download related resource of some requested page to the cache based on how busy the link is. Generally speaking, all those issues mentioned above could contribute to the efficiency of web caching along with related content in future.

References

[1] 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

[2] Sarvar Abdullaev, Franz I.S. Ko, "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

[3] Franz I.S. Ko, “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

[4] Tadao Murata, Petri Nets: Properties, Analysis and Applications, Proceedings of the IEEE, Vol. 77 No. 4, April 1989

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

[10] 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, 1996, pp.293-304.

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

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

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

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

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

[16] 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

[17] “Web Caching”, http://www.mnot.net/cache_docs/ (last accessed: 11.07.07)

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

Franz I. S. Ko (Il Seok Ko), Professor of Department of Computer Engineering, Dongguk University, Korea and Honorable Director General, IBC (International Biographical Centre), Cambridge, UK**.** Major research interests are Convergence IT, User’s characteristics analysis, Networked Services, Media Delivery, Security Engineering and Ubiquitous Computing.

Sarvar Abdullaev is currently pursuing Masters Degree in Computer Science at the Division of Computer Science and Multimedia of Dongguk University. He received his BSc in Business Computing from Westminster International University in Tashkent, 2006. Also he obtained the international qualification of Certified Accounting Practitioner (CAP) from International Institute of Accounting Standards. His major research interests are Business Computing, Computerized Accounting and Finance, Decision Sciences, Multi-agent systems, Business Intelligence systems and Convergence IT.