A PEER DISTRIBUTED WEB CACHING SYSTEM WITH INCREMENTAL UPDATE SCHEME

ZHANG YONG

(M.Eng., PKU)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgements

I would like to thank my supervisor, Associate Professor Tay Teng Tiow, for his sharp insights into my research, as well as to provide continuous and valuable guidance. I have learned a lot on the research topic itself, but more importantly the way to conduct research. I would also like to thank Associate Professor Guan Sheng-Uei, Veeravalli Bharadwaj and Dr. Le Minh Thinh for their advices.

My thanks also goes to the fellow researchers from the Electrical and Computer Engineering department, who greatly enrich my knowledge in research and make my life in NUS fruitful and enjoyable with their friendship.

Finally, I wish to thank my parents and brother, and to them, I dedicate this thesis.

Zhang Yong

April 2004

Contents

Summary

With the rapid expansion of the Internet into a highly distributed information web, the volume of data transferred on each of the links of the inter-network grows exponentially. This leads to congestion and together with processing overhead at network nodes along the data path, adds considerable latency to web request response time. One solution to this problem is to use web caches, in the form of dedicated cache proxies at the edge of networks where the client machines resides. While dedicated cache proxies are effective to some extent, alternative cache sources are the caches on other peer clients in the same or nearby local area network. This thesis proposes a peer distributed web caching system where the client computers utilize their idle time, which in today's computing environment, is a large percentage of total time, to provide a low priority cache service to peers in the vicinity. This service is provided on a best-effort basis, in that locally generated jobs are always scheduled ahead of this service. The result is unreliable service on an individual basis, but collectively in a large network, composing many clients, the service can be satisfactory. Another issue addressed in this thesis is cache consistency. The trend towards dynamic information update shortens the life expectancy of cached documents. However, these documents may still hold valuable information as many updates are minor. In this thesis, an incremental update scheme is proposed to utilize the still useful information in the stale cache. Under the scheme, the original web server generates patches whenever there are updates of web objects by coding the differences between the stale and the fresh web objects. This scheme together with the peer distributed web caching system forms the complete caching infrastructure proposed and studied in this thesis. The proposed protocol allows a client requesting an object to retrieve a small patch from the original server over bandwidth limited inter-cluster network links and the patchable stale file from its local or peer cache storage. The stale file together with the patch then generates the up-to-date file for the client. A key highlight of the proposed scheme is that it can co-exist with the current web infrastructure. This backward compatibility is important for the success of such a proposal as it is virtually impossible to require all computers, servers, clients, routers, gateways and others to change to a new system overnight, regardless of the merits of the proposed system. The generation of the patch is a key issue in our proposed scheme. The second half of the thesis is devoted to the development and evaluation of algorithms for the effective generation of patches. The key criterion is the size of the patches. It has to be significantly smaller than the original objects for the benefits of the scheme to be felt. Secondly the time complexity for the generation of the patch must be reasonable. Thirdly the coding format of patches must be such that update can be concurrent with the reception of the stale file and corresponding patch. This is important to ensure that connection time of request, roughly defined as the time from request to the time the user sees the first result of that request is minimal. In this thesis, various patch generation algorithms are developed and evaluation experiment are conducted. More than 20,000 URL were checked for update regularly and patches were generated once updates were detected. Results show that most updates are minor and most patches are much smaller than the original files. We are able to show that our peer-distributed web caching with incremental update scheme is efficient in terms of reduced inter-cluster traffic and improved response time.

List of Tables

List of Figures

Chapter 1

Introduction

In 1980, Tim Berners-Lee wrote a program, ENQUIRE [5], to run on Norsk Data machines. ENQUIRE is a method of documenting systems. It describes the parts of a system and specifies how they are interrelated. ENQUIRE allows linking between arbitrary parts of a system. This was one of the main inspirations behind the currently well known World Wide Web (WWW). In 1990, Tim Berners-Lee continued on to develop the first hypertext GUI browser and editor and named it "WorldWideWeb". In the same year, the first web server `info.cern.ch` was set up. Since then, World Wide Web has expanded rapidly. The amount of information available on the Internet, as well as the number of Internet users, grow exponentially. A key feature of the WWW is a client's accessibility of information from a server. Both the client and server can be located anywhere in the world. This makes WWW one of the most successful applications of the Internet.

The WWW can be viewed as an information mesh, where any information consumer can reach directly any information producer without knowledge of its physical location. At the information content layer, the WWW has a highly distributed structure resembling a fully interconnected model. However the model of the physical layer is very different. Ref [6] models the Internet as a hierarchy of Internet service providers (ISPs) (Fig. 1.1). There are three tiers of ISPs in this hierarchy; institutional networks, regional networks and national backbones. Clients are connected to institutional networks; institutional networks are connected to regional networks; regional networks are connected to national networks. National networks are also connected by transoceanic links to national networks on other continents. At the physical layer, the client-to-server path may traverse many networks, and through a series of intermediate sites consisting of routers, switches, proxy servers and others, connected by network links. The processing overhead at each intermediate site and the transmission time on each link sum together to give the overall latency experienced by the client. The transmission time on each link in the path may vary greatly due to the hierarchical network structure. In the hierarchical structure, a link connects the network below it to the Internet. As the network underneath grows, the bandwidth competition on the link becomes heavier. This situation is worse on higher level links, as they serve a larger number of clients in the bigger network below. Compared to a low level link, a high level link is more inclined to become the network bottleneck.

With the information content layer of WWW moving towards a more global source-destination distribution, there is a mismatch between the information content layer and the physical layer. The result is the emergence of a bottleneck in the inter-cluster data flow. This gives rise to an increased request-to-delivery latency. However, given a web link, the client usually cannot perceive the client-to-server distance or the resulting request-to-delivery latency. The client always expects a short response time as if the requested server is just next door, even though the server could actually be thousands of miles away. The resulting long waiting time is a

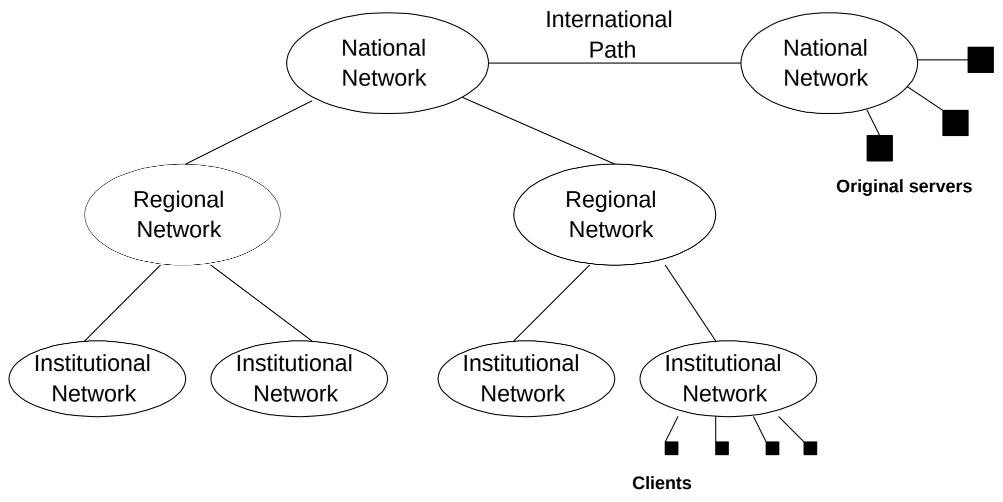


Figure 1.1: Network topology (Page 406. IEEE/ACM Transactions On Networking, Vol. 9, No. 4, Aug 2001)

cause of much frustration, leading to the cheeky "World Wide Wait" recast of WWW. To keep WWW attractive, the latency experienced by the client must be maintained under a tolerable limit.

1.1 Web caching

One method to decrease the request-to-delivery latency is to implement web caching. Caching has a long history and is a well-studied topic in the design of computer memory systems (e.g. [7, 8]), in virtual memory management in operating systems (e.g. [9]), in file systems (e.g. [10]), and in databases (e.g. [11]). In the WWW, web caching is an attempt to re-align the demand of the upper information content layer with the capability of the lower physical layer [12, 13]. The principle of web caching is to keep frequently requested items close to where they are needed [14, 15, 16]. The cached copies of web objects can be stored in the client computers, dedicated cache proxies or even web servers.

1.1.1 Client local caching and web server caching

Most modern web browsers store recently accessed pages as temporary files in the disk or memory. These pages are then quickly displayed when the user revisits the pages. Usually, the web browser allows the user to set aside a section of the computer's hard disk to store objects that the user has seen. This is a simple client local caching scheme.

Caching can also be deployed on a web server [17]. In this case, the web server contains pointers to other web servers, and it uses a local copy that is fetched in advance to fulfill a client's request. One example is the CERN HTTPD (also known as W3C HTTPD) [18], a widely used web server software that was published by CERN Lab, Switzerland in 1993. Besides acting as a web server, W3C HTTPD can also perform caching of the documents retrieved from remote hosts.

The client local caching can provide a cached copy fast, but it only serves one single client. It also has limited cache storage, which becomes more obvious when the user accesses more web pages. Web server caching on the other hand can only avoid forwarding requests further, but does nothing to reduce the latency along the path from client to the server [19].

1.1.2 Cache proxy

To mitigate the problems discussed above, the cache proxy (Fig. 1.2) was developed to provide bigger cache storage compared to client local caching, and shorter cache retrieval distance compared to the web server caching, as well as applicable access control.

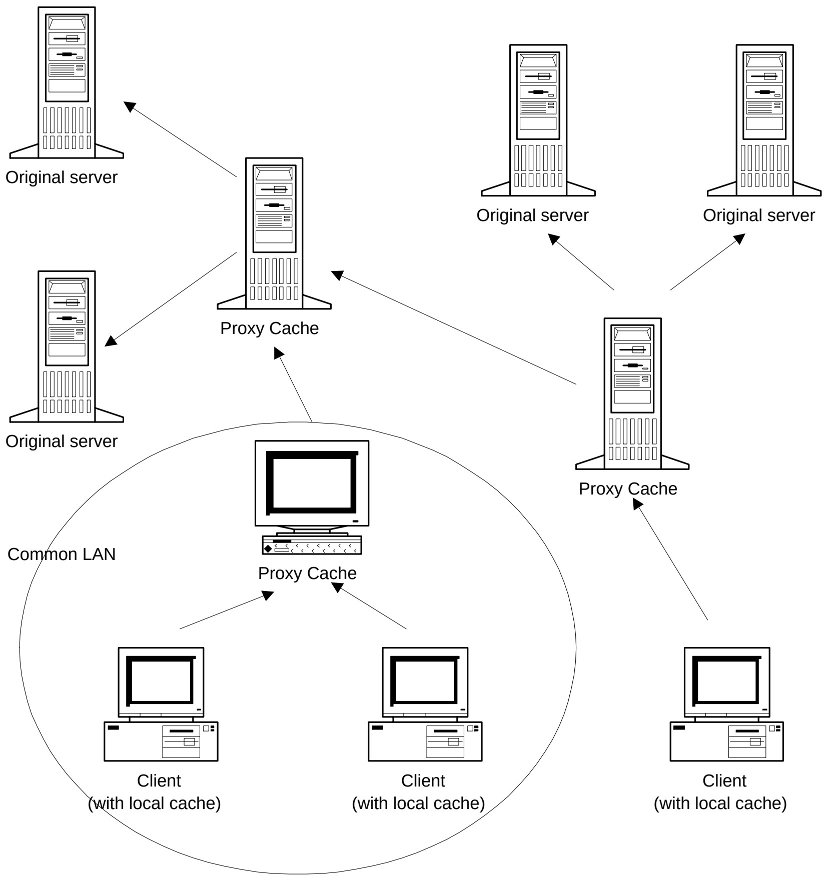
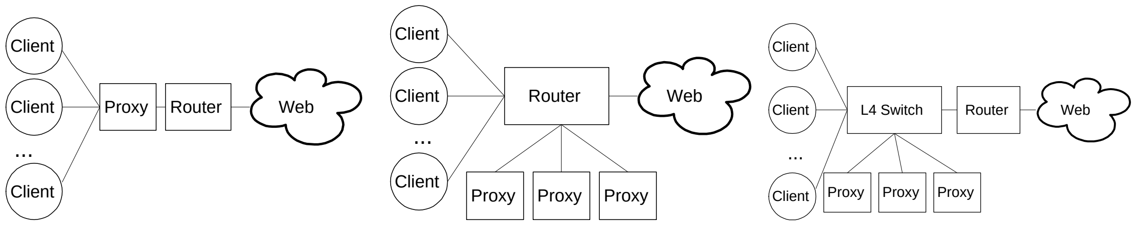


Figure 1.2: Web caching with proxies (page 171. IEEE Communication Magazine June, 1997)

Cache proxies are usually deployed at the edges of networks such as gateways or firewall hosts. They reside between original web servers and a group of clients. A cache proxy watches the HTTP requests from the clients. If the requested document is in its cache, it returns it to the client. Otherwise, it fetches the web document from the original server, saves it in its local cache and relays it to the requesting client.



(a) stand alone (b) Router transparent (c) L4 switch transparent

Figure 1.3: Cache proxy deployed at network edge (Figure 1 in [2])

A cache proxy can be configured in a standalone manner as shown in Fig. 1.3(a). In this configuration, the proxy acts as a bridge connecting its clients to the outside network. One drawback of this configuration is that when the proxy is unavailable, the network also appears unavailable. This is the so called “one point failure” problem. This configuration also requires that all web browsers be manually configured to use the appropriate cache proxy [2]. This requirement is eliminated in transparent proxy caching. In transparent caching [20, 21, 22, 23], the clients are served by multiple cache proxies, and a point is established where administrative control (e.g. load balance across multiple proxies) is possible. At such a node, the HTTP requests are intercepted and redirected to appropriate proxies. Thus, there is no need for the browser to be configured manually to use a certain proxy. The administrative node can be set up at the router (Fig. 1.3(b)) or at the switch (Fig. 1.3(c)).

Web proxies that are deployed at edges of different networks may work cooperatively to form a proxy infrastructure [24, 25, 26, 27]. One of the first documented approaches to build a coherent large caching infrastructure, HENSA UNIX service, started in late 1993 at the University of Kent [28] [14]. The goal was to provide an efficient national cache infrastructure for the United Kingdoms. Since then, many cooperative architectures have been proposed and they can be divided into three major categories. They are hierarchical, distributed and hybrid [29].

Hierarchical caching architecture

Harvest research project [30] at the University of Southern California pioneered the hierarchical caching architecture. In the hierarchical caching architecture, a group of cache proxies is arranged hierarchically in a tree like structure (Fig. 1.4). The root cache is the top of the tree. Below the root are child proxies. A child proxy again is parent to proxies linked below it. Proxies with the same parent are siblings. End clients are at the bottom of the tree. In the hierarchical tree, if a proxy cannot fulfill a request, it can query sibling and parent proxies [2]. Sibling proxies only inform a querying sibling if they have the requested document, but will not fetch the requested document. Parent proxy on the other hand will fetch the requested document for querying children [30]. The unfulfilled request will travel upwards to the root until the requested document is found. Once found, the requested document will be sent back to the requesting client through the reverse path, and each intermediate proxy on the path keeps a copy of the document [31]. The communication among cache proxies can be conducted using the Internet Cache Protocol (ICP) [32]. ICP was initially developed in the Harvest research project. It is an application layer protocol running on top of the User Datagram Protocol (UDP). It is used to exchange information on the existence of web objects among proxies. By exchanging ICP queries and replies, proxies can select an appropriate location to retrieve a document. One real life example of hierarchical caching is NLANR in the U.S. [33].

In the hierarchical architecture, each layer introduces additional delays in processing requests. Moreover, since unfulfilled requests are sent upwards, higher layer proxies may become bottlenecks during child query processing. This is solved by distributed caching architecture, which is not layered.

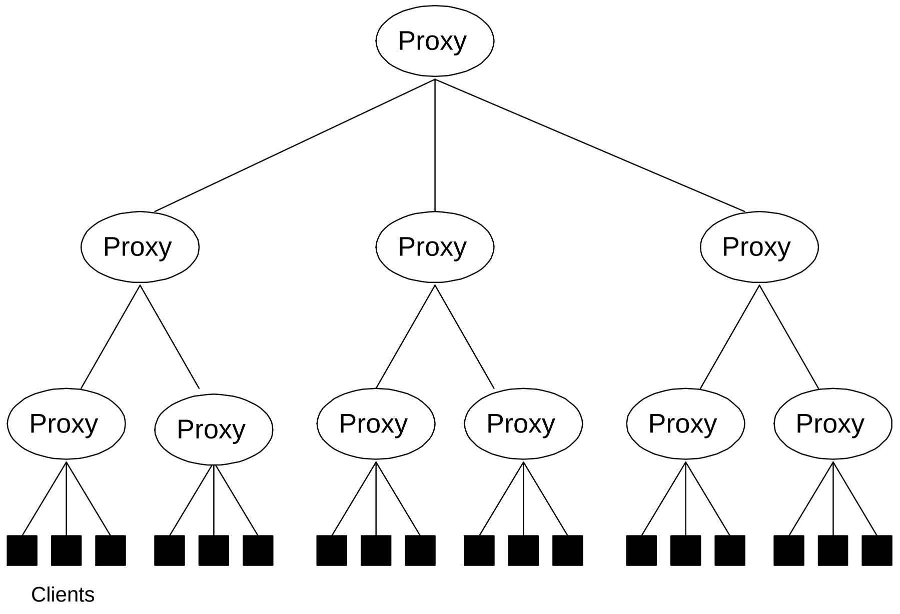


Figure 1.4: An illustration of hierarchical web caching structure

Distributed caching architecture

In conventional distributed caching architectures [34, 35, 36], institutional proxies at network edges cooperate, with equal importance, to serve each other's cache-misses [37]. Since there is no intermediate proxy to collect or centralize the requests from other proxies, as in hierarchical caching architecture, distributed caching architecture needs other mechanisms to share cache storage [6].

The sharing mechanisms can be query-based, digest summary based or hash based.

ICP is a popular method used to construct a query-based distributed caching architecture. With such a mechanism, proxies can query other cooperating proxies for documents that result in local misses. The location of the requested cache can be discovered through ICP query/reply exchanges. Moreover, ICP reply messages may include information that assists selection of the most appropriate cache source. Query-based mechanism tries to achieve a high hit rate, and response time is good when the cache proxies are near to each other.

[38] and [39] propose summary-based and digest-based mechanisms respectively. With these mechanisms, proxies keep and update periodically the compressed directory of other proxies' cache content in the form of digest or summary. The cache location can be decided locally and fast by checking the digest or summary. The summary mechanism proposed in [38] and the digest mechanism proposed in [39] are similar. The major difference is that the summary mechanism uses ICP to update the directory, while the digest mechanism uses HTTP to transfer the directory [40]. Distributed proxies can also cooperate using a hash function [41, 42]. The hash function maps a cache request into a certain cache proxy. With this approach, there is no need for proxies to know about each other's cache content. However, there is only one single copy of a document among all cooperative proxies. This drawback limits the approach to a local environment with well-interconnected proxies.

Hybrid caching architecture

If in a hierarchical caching architecture, a cache proxy cooperates with other proxies (not necessarily at the same level) using a distributed caching mechanism, it becomes a hybrid caching architecture.

In the hybrid architecture, a proxy that fails to fulfill a request first checks if the requested document resides in any of the proxies that cooperate with it in a distributed manner. If no such proxy has the requested document, the request will be forwarded upwards as in a hierarchical architecture.

Pablo and Christian [6] modeled the above three caching architectures and compared their performance. They found that hierarchical caching systems have lower connection time, the time lapse from the client requests of a document to the reception of the first data byte, while distributed caching systems have lower transmission time, the time to complete transmitting a document. They also found that hierarchical caching has lower bandwidth usage, while distributed caching distributes the traffic better as it uses more bandwidth in the lower network level. Their analysis also shows that in a hybrid caching system, the latency depends very much on the number of proxies that cooperate in a distributed manner. Well-configured hybrid scheme can reduce both connection time and transmission time.

1.1.3 Dynamic caching architecture

The conventional caching architectures discussed above such as Harvest [30] and Squid [43] are deemed static as they have limited flexibility in forwarding unfulfilled requests [44]. To address this issue, some dynamic caching architectures have been proposed in recent years to provide flexibility in the communication path among cache proxies.

One example is adaptive caching proposed in [3, 45]. In adaptive caching, all web servers and cache proxies are organized into multiple local multicasting groups as shown in Fig. 1.5. A cache proxy may join more than one group, so that the groups heavily overlap each other. An unfulfilled request at a proxy will be multicasted within the group. If the group cannot resolve the request, the request will be forwarded to a nearby group that is closer to the original server through the joint proxy. In this way, an unfulfilled request will travel through a chain of overlapped cache groups between the client and the original server, until it reaches a group with the requested page or the group that includes the original server. When the requested document is found at a proxy in a group, the proxy will multicast the document within the group. Thus all the neighboring proxies in the same group are loaded with this document. This document will then be relayed back to the requesting client via unicasting by traversing those proxies that forwarded the request earlier. In adaptive caching, popular web objects will quickly propagate themselves into more proxies, while pages with infrequent request will be seen only by a few proxies near the original server. Cache Group Management Protocol (CGMP) is developed to make the group creation and maintenance self-configuring. The ongoing negotiation of mesh formation and membership result in a virtual and dynamic topology.

Caching Neighborhood Protocol (CNP) [4, 46] is another dynamic caching system. In CNP, an original server builds its own "caching representative" neighborhood (see Fig. 1.6). A caching representative is a cache proxy that represents multiple original servers and is devoted to distributing loads for them. An original server collects certain information from its representatives, invites proxy servers to join its neighborhood or drops a representative off the list at its own discretion. Compared with adaptive caching, CNP allows the original server to take a more active role in neighborhood maintenance. The CNP approach is regarded dynamic because the set of cache proxies that collaboratively handle the requests may change for every single request.

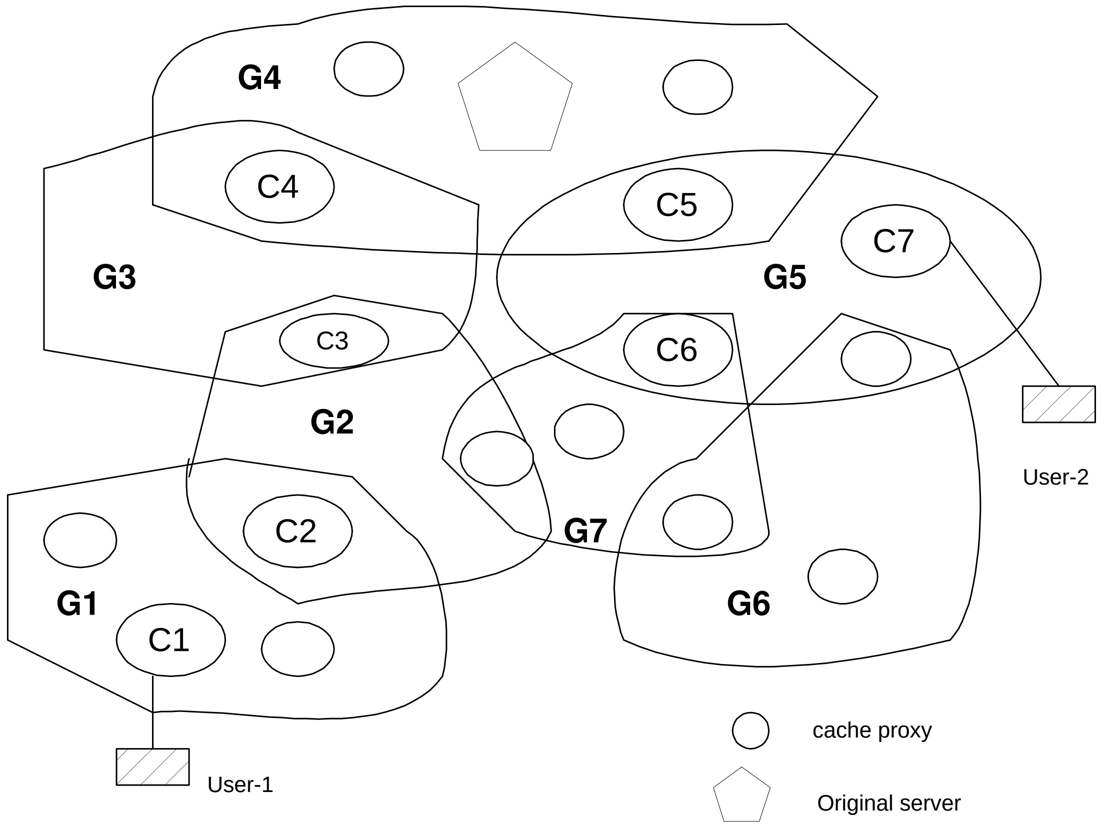


Figure 1.5: An illustrative example of adaptive caching design (Figure 1 in [3])

1.1.4 Local cache sharing

Cache proxies were developed initially to provide bigger cache storage and to serve more clients than the client local caching. Different cache proxy architectures are created with a variation of components, dedicated cache server hardware, and protocols. The goal is to achieve a balance between performance improvement and implementation cost. As cache proxy architectures evolve, the computation power and the storage space of the client personal computer also increase as a result of advancement in fabrication and storage technologies. Client personal computers are now grouped into clusters in a LAN. Compared with the cache on a dedicated cache proxy deployed outside of the cluster, the cache on a peer client computer

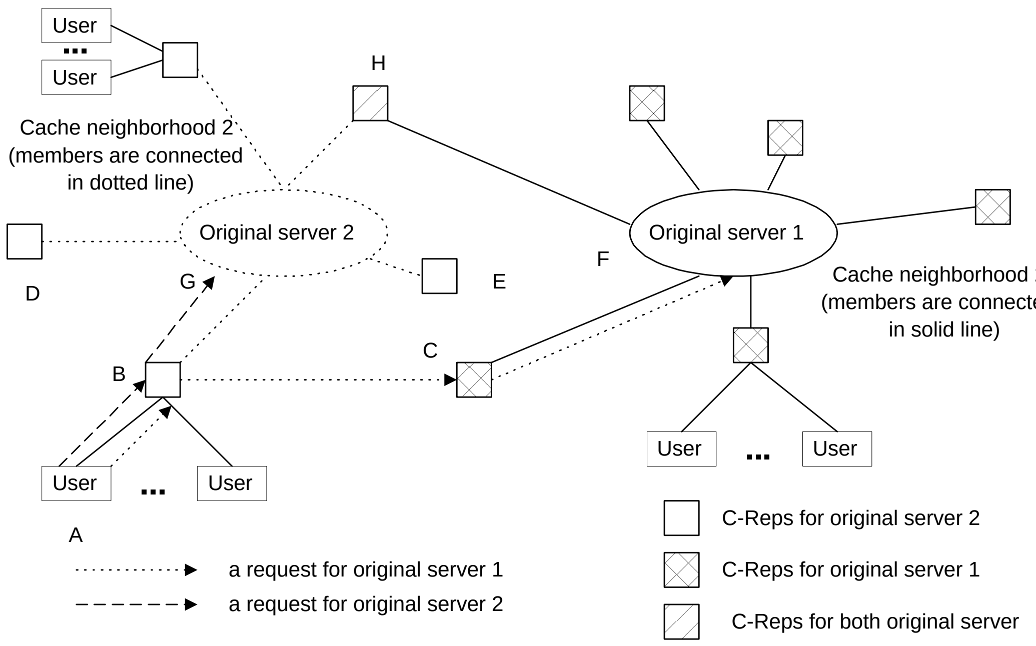


Figure 1.6: An illustrative scenario consisting of two caching neighborhoods (Figure 1 in [4])

within the same cluster is nearer. Moreover, the client users in a cluster usually belong to the same organization, so very likely they have similar web interest. Therefore, the contents of the local cache on a client computer may be appropriate to peers in the same cluster. Thus, it is natural to consider that client personal computers share their local cache storage with peers. Any unutilized computation power of the client computer, which is a perishable resource, can be utilized for this server service.

The sharing of cache among peer clients can be conducted in a centralization manner as in the peer-to-peer sharing scheme proposed by [47]. In this scheme, a dedicated proxy connecting to a LAN is the centralized control point. The dedicated proxy maintains an index of web objects on all the clients in the LAN. It searches the index for a “hit” on a client when there is a miss in its own cache. Once such a hit is found, the proxy instructs the client to send the data to the requesting node. In an alternative implementation, the proxy fetches the data from the source node and sends it to the requesting client. In the peer-to-peer sharing scheme, cache location discovery is fast. However, the maintenance cost is high. To maintain an up-to-date client cache index, the dedicated proxy needs to record traces of web object communications, and the client needs to report all its cache manipulations to the dedicated cache server. Moreover, the peer-to-peer sharing does not consider the case where more than one dedicated cache proxy servers are connected to a LAN to distribute load.

Unlike the centralized method, we propose in this thesis a distributed way to share client cache. We refer to it as peer distributed web caching. A query-based mechanism is used to share cache in this proposal. Peer clients in the same cluster take on an additional cache server function on an on-demand basis. A requesting client queries its peers when a miss occurs in its local cache and waits for a hit reply from a peer client holding the requested document. The request that cannot be fulfilled within the cluster will be forwarded outside. This peer distributed caching system can be deployed without any change to intermediate dedicated cache proxies on the path from the client to the original server.

1.2 Caching consistency

In this thesis, we also introduce an “incremental update scheme” to address the cache consistency problem [48, 49, 50].

For web caches to be useful, cache consistency must be maintained by the cache proxy, that is, cached copies should be updated when the originals change. A cache proxy can provide weak cache consistency or strong cache consistency. Weak consistency is defined as one in which a stale web object might be returned to the client under certain unusual circumstances, and strong consistency is defined as one in which after an update on the original web object completes, no stale copy of the modified web object would be returned to the client [51, 52, 53, 54]. A widely-used weak consistency mechanism is Time-To-Live (TTL). In this mechanism, a TTL value is assigned to each web object. The TTL value is an estimate of the object's life time. When the TTL elapses, the web object is considered invalid, and the next request for the object will cause the object to be requested from its original server. TTL mechanism is implemented in HTTP using the optional "expires" header field [55].

Strong consistency could be achieved by a polling-every-time mechanism or by using an invalidation callback protocol. In the polling-every-time mechanism, every time the cache proxy receives a web request and a cached copy is available, the cache proxy contacts the original server to check the validity of the cache copy. If it is still valid, the cache proxy returns it to the requesting client; otherwise a new copy is fetched from the original server and returned to the requesting client. Invalidation callback protocol involves the original server keeping track of all the cache proxies where the web object is cached and then sending an invalidation command to the cache proxies once the web object is updated. The problem with invalidation protocols is the expensive implementation cost.

Whatever method is used to validate cached copies, once a cached copy is found stale, it is flushed off. The new version of the web object is then fetched from the original server. With the trend towards up-to-date dynamic information delivery, the life expectancy of the web object at cache servers is shortened, thus, cache misses occur more frequently and the advantage of caching is decreased. If update on web object at the original server is incremental, with small changes at each update, then the stale object still holds valuable information. In such a situation, we can visualize the original server delivering a patch or a "delta" between two versions, instead of the whole fresh file, to update the stale file at the client.

The idea of "delta" encoding for HTTP is not new. The WebExpress project [56] appears to be the first published description on delta encoding for HTTP. However, it is applied only in wireless environments. [57] suggested the use of optimistic deltas, where a server-end proxy and a client-end proxy deployed at the two ends of a slow link collaborate to reduce latency. Both of these two projects assume that the existing clients and original servers are not aware of the "delta" encoding and rely on proxies situated at the ends of slow links. [58] proposed to extend HTTP protocol to make end-to-end delta encoding possible. The delta algorithms used in [58] are "diff-e", "compressed diff-e" and "vdelta" [59]. These algorithms take web objects as plain text or strings. The hierarchical structure that web objects may have is not utilized. Moreover, compression used in the latter two algorithms makes the delta not usable until it is received completely. [58] estimated and confirmed the benefits based on "live" proxy level and packet level traces. Although the estimation is conservative, the delta querying and requesting time, which is dependent on the caching scheme and protocol used, is not taken into account.

In this thesis, we extend the delta delivery and decoding technology and refer to it as an "incremental update and delivery scheme". This scheme is incorporated into the peer distributed web caching to construct an integrated caching system, referred to as peer distributed web caching with incremental update scheme (PDW-CIUS).

1.3 Contribution of the thesis

In this thesis, we propose a novel peer distributed web caching system with an incremental update scheme to improve caching effectiveness. In the proposed caching system, every client is assigned a cache server service to share its local cache with peers in the same cluster, and the original server computes and provides patches.

We developed a comprehensive set of protocol for cache querying, cache retrieving, patch querying and patch retrieving to fulfill web requests. We introduce new HTTP header fields in the proposed protocol for patch communication and ensure end-to-end delivery of patches.

Our peer distributed web caching system with incremental update scheme tries to resolve a web request within the local cluster by sharing client local cache and utilizing up-to-date content in the cache. It reduces inter-cluster traffic and request-to-delivery latency and improves cache hit rate. It is shown to be an effective alternative to the objective of increasing caching effectiveness.

We also proposed methods to solve the patch generation problem, the key issue in the incremental update and delivery scheme. The patch is expected to be small to achieve a short delivery time, and the patch generation is expected to be applicable to all kinds of web objects such as HTML files, XML files, plain text, image, audio, and video files. In this thesis, we unify web object file types and transform web objects into tree structures. Web patch is then generated as a tree-to-tree correction. To achieve the minimal patch size, this thesis recasts the tree-to-tree correction problem into a minimal set cover problem and solves it under some simplifying assumptions. We also developed suboptimal patch generation algorithms using fixed instruction set to achieve a lesser time complexity. Experiments and analytical methods were conducted to evaluate the proposed algorithms.

This thesis also shows how the proposed system supports dynamic documents that change very frequently.

1.4 Organization of the thesis

The thesis is organized as follows.

In Chapter 2, the proposed peer distributed web caching system with incremental update scheme is described. The benefits derived as a result of implementing the proposed system are also analyzed. It is shown that the caching effectiveness is related to the patch size. A minimal patch size is desirable. This leads us to the next four chapters of this thesis which address the patch generation problem.

Chapter 3 sets up the environment required to generate the minimal patch used in our proposed protocol. We first unify web file types so that a web object file can be viewed as a combination of structured data and unstructured data. A method is also proposed to add structure to unstructured data and to transform a web object file into a tree structure. Once it is recasted into a tree structure, the patch generation problem then becomes a tree correction problem. The patch structure is also defined in this chapter.

Chapter 4 discusses the general solution to the minimal patch generation problem. To achieve the minimum patch size, a dynamic instruction set is used. The minimal patch problem is recasted into a minimal set cover problem (MSCP) with dynamic weight. Appropriate simplifying assumptions are made and solutions to the dynamic weight MSCP under these assumptions are proposed using available approximate solutions of MSCP.

To achieve a lesser time complexity than the algorithm proposed in Chapter 4, we propose in Chapter 5 the use of a fixed instruction set to generate a web patch. Algorithms are proposed and evaluation experiments are conducted.

Chapter 4 and Chapter 5 address changes on web objects that are random or general in structure. However, in many real time, customized web applications, there exist some unchanged structures or nodes between consecutive versions of a page. By exploiting knowledge of the structure or node, web patch can be generated online to make support of the dynamic document possible in the proposed caching system. This is discussed in Chapter 6.

Conclusions are drawn in Chapter 7.

Chapter 2

System Description and Analysis

In this chapter, we first discuss the network structure for which our proposed caching system is intended. We then describe the proposed protocol, discuss the implementation issues, analyze the benefits and finally discuss system scalability and service reliability.

2.1 Introduction

The Internet is a network of computer networks allowing computers connected to any part of the network to exchange information. Nodes on a computer network are typically designated to handle certain types of data and perform certain types of functions. From the perspective of web application, there are three major types of nodes, namely routers, servers and clients. A router is a layer three device that forwards data packets along networks. It uses IP packet headers and a forwarding table to determine the best path for forwarding the packets. A router is located at the network gateway and is connected to at least two networks. The router directs a data packet towards its destination, which may travel across multiple networks. A server is a computer on a network that provides services such as web files, file storing, printing, chatting, database, and many others. Servers are often dedicated, meaning that they perform no other tasks besides their server tasks. A client computer, on the contrary, is a requester of services provided by servers. Client computers are usually accessed by end users.

Fig. 2.1 illustrates how routers at different levels connect clients and servers in different networks to form the Internet. The routers at the highest level are scattered around the world and connected to national networks. In each national network, there are regional networks interconnected by second level routers. Some of these second level routers are also gateways communicating with the higher level routers. Within each regional network, there are institutional networks. These networks are interconnected to each other by another layer of routers. Within the institutional network, computers are organized into groups, usually by their locations. Each of these groups forms a local area network (LAN). These local area networks are again interconnected by the lowest level routers. Within a local area network, there are computers that act as servers or clients.

Typically server nodes on the network are powerful computers with a high degree of reliability while client nodes are usually served by less powerful personal computers. However, in recent years, the personal computer's capability in terms of processing power and disk storage increases rapidly. Personal computers running clients applications are now becoming as powerful as low-end server computers. A server computer's workload is very much dependent on the number of clients it serves. It does vary through the day, but in general it tends to be much more uniform compared to the workload on a client personal computer. The system demand on a client personal computer tends to vary a lot more and in fact, for a

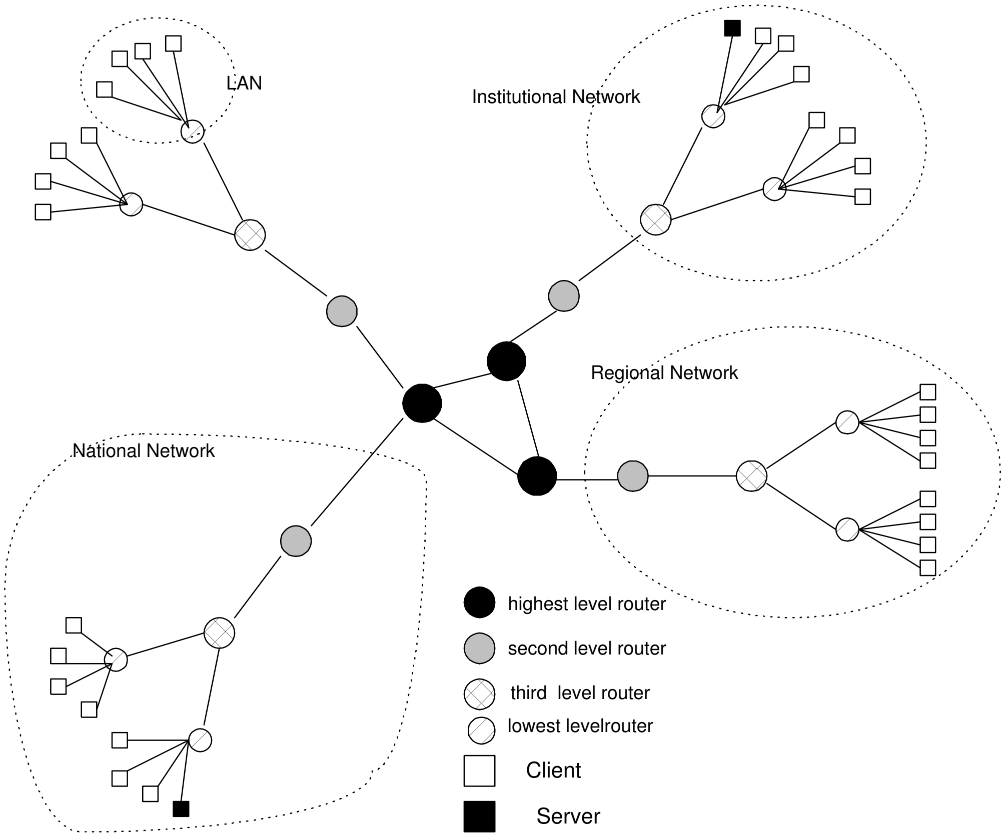


Figure 2.1: General network structure

good part of the time the client computer is in an idle state.

The free computation power on the client personal computer is a perishable resource. It can be utilized to perform some server function on an on-demand basis, when the demand for local computing power is low. A proposal in this thesis is the cache server function, which provides local cache to peer computers. Under this proposal, a client computer performs locally initiated tasks as its first priority. When there is no locally initiated task listed in the queue, the operating system (OS) can assign the computing resources to perform caching function. If a locally initiated task is created, the client computer would abandon the caching server task. In this way, users should not experience a degradation of performance of their computers. One drawback of making the server functions a second priority task is that the service reliability on any one single computer cannot be guaranteed. However, if multiple client computers collaborate to provide the caching service, then the overall service reliability can be improved as shown later in Section 2.6. In our proposal, namely peer distributed web caching, all the client computers in the same computer cluster share their local cache with peers in a distributed manner with equal importance. This proposal aims to provide a near cache source to peer computers in a cluster. In fact, typically the peer distributed web caching system is deployed within a cluster, say a LAN, where the computers are geographically close together.

An additional feature in our proposal is an incremental update and delivery scheme. This scheme works in the context of web caching. It aims to improve cache usage by relaxing the cache consistency criteria. A cached copy of a web object becomes stale or inconsistent after the original copy is updated at the original server. However, parts of the content of the web object may not be changed. In this case, the stale cached copy is still of value. The incremental update and delivery scheme proposes to utilize the stale cached copy. Under the incremental update and delivery scheme, the original server provides a patch, which is a sequence of edit operations that will transform a stale web object into a fresh version. If the original server can provide the corresponding patch for a stale version, the cached copies of that version are considered patchable. With the incremental update and delivery scheme, a client has a new way to get its web request fulfilled, that is to retrieve a patchable cached copy within the cluster and a patch from the original server, and then to regenerate the up-to-date version with the two files using a patch-decoding routine. This is depicted in Fig. 2.2.

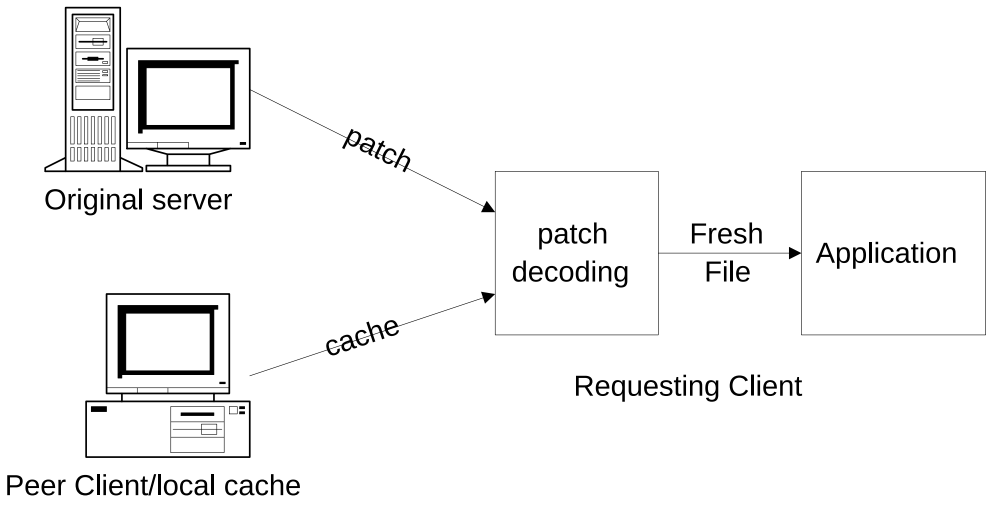


Figure 2.2: A general scenario of the incremental update scheme

To perform the additional cache server function, the OS on the client personal computer is extended with a new module, client-end module for distributed web caching (C-DWEBC). The C-DWEBC module enables the client computer to store the web objects that it receives. The storage space allocated for this caching purpose is a parameter to be set. C-DWEBC can serve the cached web object to the client itself or to peer clients in the same cluster when the same object is requested later. A set of protocol is proposed in this chapter for C-DWEBC module to support serving the locally cached objects to peers in the vicinity. Fig. 2.3 shows the sequence of communication among C-DWEBC modules. The C-DWEBC module intercepts the web request from the local client application. It first attempts to use a local cached copy to resolve the web request. If a local cached copy is not available, it sends cache query to peer C-DWEBC modules within the cluster. A peer holding the requested cached copy replies with a cache hit notification. Once a cache hit notification is received, the requesting C-DWEBC establishes a connection with the responding peer to retrieve the cached document.

To provide patches to clients, the OS on the original web server is extended with a new module, server-end module for distributed web caching (S-DWEBC). Fig. 2.3 shows the communication between S-DWEBC and C-DWEBC. To fulfill a web request, the requesting C-DWEBC module asks for the cached copy from peer C-DWEBC modules as described in last paragraph. At the same time, it queries the S-DWEBC module for the range of the patchable cache version. If the cached copy is patchable, the requesting C-DWEBC opens a connection with S-DWEBC to fetch the corresponding patch. From the patch and the cached copy, a fresh version is regenerated.

2.2 Protocol

In this section, we describe the proposed protocol. It specifies how C-DWEBC modules and S-DWEBC modules communicate to solve web requests.

In the proposed protocol, the C-DWEBC modules and S-DWEBC modules exchange information on top of the TCP/UDP layer. As shown in Fig. 2.3, the communication among peer C-DWEBC modules includes cache query, cache hit and cache transfer. The cache query data and the cache hit data should be received by all C-DWEBC modules in the same peer distributed caching system. It is a one-to-many data communication implemented using a multicast protocol on top of the UDP layer. All the C-DWEBC modules in a peer distributed caching system are configured to belong to the same multicast group by assigning the same

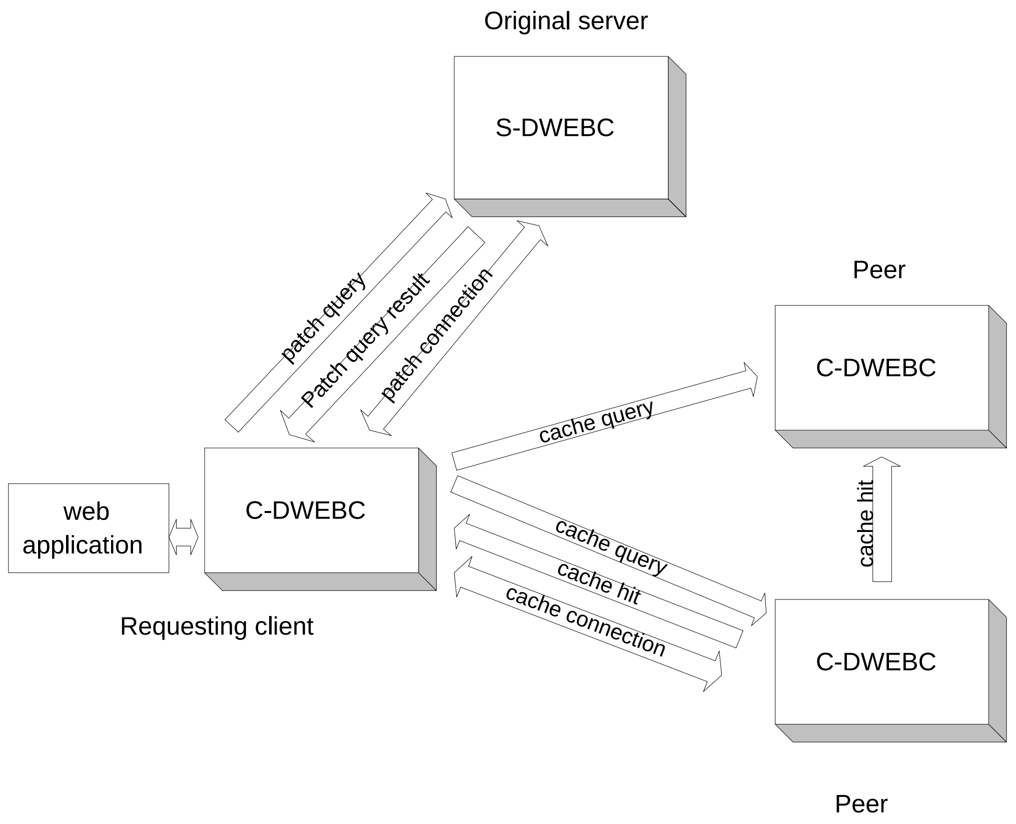


Figure 2.3: Modules in peer distributed web caching system

multicast address to them. As for the cache transfer connection between two C-DWEBC modules, it uses the HTTP protocol on top of the TCP layer. The two modules form a simple client-server architecture. The communication between S-DWEBC module and C-DWEBC module includes patch query/reply and patch transfer connections. They follow the HTTP protocol with some newly introduced header fields (see Section 2.3.1).

2.2.1 Description of C-DWEBC - local request

A C-DWEBC module deals with service requests from both the local applications and peer client computers. The processing of local web requests is described as follows, while the processing of service requests from peers will be presented in the next subsection.

1. On receiving a request for a web object, say O, C-DWEBC checks if it has a local cached copy. Simultaneously, it sets a timer and requests the original server for the patch information on O. The patch information includes the time stamp of the updated O (V_{Latest}), the time stamp of the oldest patchable stale file (V_{Oldest}), the size of the updated O (S_o) and whether the original server maintains the patches for O.

2. If a local cached copy is not available, it proceeds to Step 4. If a local cached copy is available, it waits for the arrival of the patch information. If it does not arrive before the pre-defined deadline, the original server is considered unreachable, and the local cached copy is delivered to the client application with a “no-verification” indication. The procedure ends. If the patch information arrives before the deadline, it goes to Step 3.

3. C-DWEBC checks the local cache’s freshness and patchability. If V_{Cache}, the time stamp of cached copy, is the same with V_{Latest}, the fresh cache is accepted, delivered to the application and put into the local cache storage. The procedure ends. If V_{Cache} is unavailable or older than V_{Oldest}, the cached copy is inconsistent and flushed off. It proceeds to Step 4. If it falls between V_{Latest} and V_{Oldest}, the cached copy is considered patchable and it proceeds to Step 7.

4. C-DWEBC on the requesting client multicasts a cache-query and starts a timer to implement a deadline for the responses from potential peer cache servers. Note that the deadline may be updated later to achieve a proper waiting time. This is discussed later in this chapter.

5. On receiving a "cache-hit" response, C-DWEBC immediately opens a unicast channel with the responder to request the cache header, including V_{Cache}. For the current implementation, a first-come-first-accept algorithm is adopted for the selection of cache servers. If there is no response from any computers by the deadline for response, the requesting client may go back to Step 4 with perhaps a larger hop value multicast. Alternatively, it directly fetch O from the original server.

6. Upon receiving V_{Cache}, C-DWEBC checks the cache's freshness and patchability as in Step 3. If the cached copy is patchable, it proceeds to Step 7. The fresh cache is accepted, delivered to the application and put into the local cache storage. The procedure ends. The inconsistent cache connection is aborted. C-DWEBC module returns to Step 4 if the repetition has not exceeded a pre-defined limit, MAX_REPEAT. Note that returning to Step 4 after this point would allow the multicast to be performed with the received patch information. The repeat also serves to invalidate the inconsistent cached copies in peers. If patch information does not arrive before the deadline, the original server is considered unreachable, and the cached copy is delivered to the client application with a no-verification indication.

7. C-DWEBC opens a connection with the original web server to request a patch of a particular version.

8. If the patch response header indicates that the satisfying patch follows, the reception of the cached copy and the patch continues. The fresh web object is

computed, delivered to the application and put into the local cache storage. The procedure ends.

9. If the patch header shows that the desirable patch is not available, the cache session is aborted. If the patch header shows that the updated object follows, the connection continues, the updated object is delivered to the application and put into its local cache. The procedure ends. If the patch header shows that no response body follows, the connection is aborted, and C-DWEBC fetches the updated O directly from the original server.

In Step 4, a deadline is implemented for the receipt of a cached copy. When the requested O is huge, it makes sense to wait a longer time to increase the chance of a peer responding since the inter-cluster bandwidth requirement for a huge object is high. However, if the requested O is small, waiting for a long time is unwise.

Let P_{cc} be the probability that the requesting client gets a patchable cached copy locally or from peers. Such a cached copy has a matching patch at the original server. Let S_o be the size of O, S_p be the patch size, and S_c be the size of the cached copy. Let r_o be the transmission rate of the original web object, and r_c be the transmission rate of the cached copy. Let T_c be the longest acceptable waiting time for peers' response. The requesting client has two ways to get the web object. The first way is to fetch the file directly from the original web server, and the time cost is S_o/r_o. The second way is to fetch the cached copy from peers and the patch from the original server and then compute the fresh one. The time cost is $\max(S_c/r_c, S_p/r_o)$ plus the extra waiting time for peers' response. This extra waiting time is expected to be less than $\frac{S_o}{r_o} - \max(\frac{S_c}{r_c}, \frac{S_p}{r_o})$. Thus, it is desirable that

$$T_c = P_{cc}(\frac{S_o}{r_o} - \max(\frac{S_c}{r_c}, \frac{S_p}{r_o})). \quad (2.1)$$

The parameters r_o, r_c and P_{cc} are known values. r_o and r_c are the average values based on original file transmission and peer cache transmission within a period of time. P_{cc} is the average based on the ratio of patchable caches received within a period of time. At the onset, S_o is not known. The initial time-out period, T_{co}, is selected based on some assumed probability that a peer has a cached copy. T_{co} may be chosen from a preset range. A more accurate deadline, T_c, is subsequently calculated when the patch information is received. Since we do not know which patch version is needed and consequently S_p until we get the response from peers, we assume that $\max(\frac{S_c}{r_c}, \frac{S_p}{r_o})$ equals $\frac{S_c}{r_c}$ since the patch is very small. Our experiment also shows that on average the original file and the cache file are almost the same in size, and an assumption is that S_c is the same as S_o. Thus T_c can be calculated as follows:

$$T_c = P_{cc}(\frac{S_o}{r_o} - \frac{S_o}{r_c}) \tag{2.2}$$

Once the patch information is received, the requesting client extracts S_o from it, calculates the longest acceptable waiting time using Eq. 2.2 and updates the previous T_{c0}.

2.2.2 Description of C-DWEBC - peer request

In this subsection, we describe how C-DWEBC processes the service request from peers. The service requests from peers can be of two types, namely cache query and cache hit messages. Cache query messages are multicast messages originating from peer computers enquiring on the availability of certain documents in the cache storage of the said computer. Cache hit messages are also multicast messages, but originating from computers that had earlier received cache query messages for certain documents, and have those documents in their cache storage.

C-DWEBC module uses a foreground thread and a background task to deal with the service requests from peers. It also maintains a queue for the multicast messages. The queue in general is filled by the foreground thread and consumed by the background task. Specifically, the foreground thread is invoked each time a cache query or a cache hit message is received. The foreground thread upon receiving the messages will pre-process them in the following manner to determine if the message should be queued.

```
Case 1: The multicast packet is a cache-query message.
    The packet is added at the rear of the queue together with the time, $t_{receive}$,
    when the packet is received.
Case 2: The multicast packet is a cache-hit message.
  If (a web request for the object described in the cache-hit packet is in process
    locally)
      This cache-hit packet is forwarded to the function dealing with the local
      web request (see Step 5 in Section 2.2.1).
  else
      It is added at the rear of the service request queue.
```

As shown, the foreground thread of C-DWEBC only processes incoming "cache-hit" messages that happen to match its local web request. Otherwise it is queued. Thus no incoming requests from peer computers will be processed ahead of local requests, and the response time of local requests should not be adversely affected.

The background task of C-DWEBC is typically allocated the lowest scheduling priority. Therefore only when there is no higher priority task in the system, will C-DWEBC process the service request messages. The messages are processed and removed from the queue in a random fashion. The rationale for the random selection of the message will be further discussed in Section 2.5. The processing of each of the messages removed from the queue is done as follows.

Case 1: It is a cache-hit packet.

```
  Search the queue for the cache query that matches this cache hit.
  If (there is a matching cache query)
      Remove the query packet.
  else
      Ignore the cache-hit packet.
Case 2: It is a cache-query.
  If (a copy of the web object is available and consistent)
      Estimate the time out period at the requesting peer, $T_c$, with $T'_c$, using the
      equation, $T'_c = P'_{cc}(\frac{S_c}{r_o} - \frac{S_c}{r_c})$, where $r_o$, $r_c$ and $P_{cc}$ are values from earlier
      transactions.
      If ($t_{receive} + T'_c < t_{now}$, indicating the requesting client has given up)
         The "cache-query" is ignored.
      else
         Multicast a cache-hit on the same multicast address as in the cache-query
         packet.
  If (a copy of the web object is available but inconsistent)
      The cached copy is flushed and the cache-query message is ignored.
  If (the web object identified in the cache-query is not available)
      The cache-query message is ignored.
```

A parameter to be determined is the length of the queue. Ideally, the length should be set such that $t_{receive}$ plus the processing time for responding to queries in the queue is still within the waiting period of the sender. In this thesis, we model the multicast "cache-query" and "cache-hit" packets at a node as a Poisson process. The multicast packet comes randomly at a rate of $\lambda(t)$ per second. The probability that k "cache-query" packets arrive from time t to $t + \triangle t$, $P(k \quad in \quad [t, t + \triangle t])$, is calculated as

$$P(k \quad in \quad [t, t + \triangle t]) = \frac{1}{k!}[\int_t^{t+\triangle t} \lambda(\varsigma)d\varsigma]^k exp[-\int_t^{t+\triangle t} \lambda(\varsigma)d\varsigma]. \tag{2.3}$$

If $\triangle t$ is very small, Equation 2.3 can be simplified to

$$P(k \;\; in \;\; [t, t + \triangle t]) = \frac{1}{k!}[\lambda(t)\triangle t]^k exp[-\lambda(t)\triangle t]. \tag{2.4}$$

The average number of packets arriving from time t to $t + \triangle t$, $\eta(t)$, is then

$$\eta(t) = \lambda(t)\triangle t. \tag{2.5}$$

Since C-DWEBC only processes packets received in the past T_c' time, the queue length can be simply set to

$$\overline{\eta}(t) = \lambda(t)\overline{T_c'}. \tag{2.6}$$

$\lambda(t)$ is estimated based on the observation of previous incoming rate. $\overline{T_c'}$ is calculated by averaging previously known T_c'.

2.2.3 Description of S-DWEBC

S-DWEBC is deployed on the original server. It answers patch enquires and patch requests from requesting clients (Fig.2.3). An original sever can generate patches off-line and keep them for later requests. If a requested document is "dynamic", a patch may be generated online during request processing. Dynamic documents support will be discussed in Section 2.3.3.

Upon receiving a patch query, a "HEAD" request containing a "Request-Patch" (Sect. 2.3.1), S-DWEBC is to respond with the version information on the patch and web object. To do this, S-DWEBC first checks if such a patch is available. If the patch is maintained offline or can be generated online, it adds a Patch-held field (Sect. 2.3.1) into the header. The patch information, V_{Oldest} and V_{Latest}, are put into the header as the “Old-Version” and “New-Version” fields respectively. If no patch for the web object is available, a non-Patch field is put into the header. The header is then sent to the requesting client.

Upon receiving the patch request, a “GET” request containing a “Request-Patch”, S-DWEBC extracts V_{Cache} from the request header, and checks if a patch is available for this particular version of the stale file. If the original server can offer the required patch, and the patch is smaller than the value of “Max-Length” in the request header, a “Response-Patch” is filed. The patch length and the description of instruction set used in the patch are put into the response header. After the header, the patch follows in the response body. If the original server can not provide the satisfying patch, S-DWEBC checks if the “Accept-Newversion” field is present in the request header. If it is, S-DWEBC puts the “Response-Newversion” header field and $V_{Lastest}$ in the header, followed by the updated web object. Otherwise, S-DWEBC puts the “Non-Patch” header field in the header and only the response header is sent.

2.3 Implementation issues

As shown in Section 2.2, clients and original servers exchange information on patches. This section introduces new HTTP header fields for this purpose. To guarantee that the patch is always up-to-date, patches are expected to be available only at the publishing server and the patch communication is expected to be transparent to intermediate sites. This section proposes methods to achieve this. The patch maintenance and replacement [60] issues are covered in this section. This section also shows how dynamic web documents are supported by the proposed system.

2.3.1 Patch-Control header fields

To exchange information on patches between servers and clients, new Patch-Control header fields are introduced. In this thesis, we define the new fields under the "extension-header" mechanism provided by HTTP protocol [55]. This extension allows additional entity-header fields to be defined without changing the protocol. According to RFC2616.7.1 [61], unaware recipients, on receiving these new definitions should ignore the fields and forward the packets without modification. The proposed scheme could thus co-exist with legacy packets in the current network infrastructure. The patch-control header fields are defined below.

```
Patch-Control="Patch-Control"
   ":"|#Patch-directive
Patch-directive = patch-general-directives | patch-request-directive|
   patch-response-directive
patch-general-directives =
   "Reference-URL" "=" URL
Patch-request-directive =
   "Request-Patch"|
   "Accept-Newversion"|
   "Old-Version" "=" Http-date|
   "Max-Length" "=" 1*DIGIT
patch-response-directive =
   "Response-Patch"|
   "Response-Newversion"|
   "Non-Patch"|
   "Zero-Patch"|
   "Patch-Held"|
   "Old-Version" "=" Http-date|
```

"New-Version" "=" Http-date|

"Patch-Length" "="1*DIGIT|

"Instruction-Set" "=" Instruction Set Description

The general header filed, "Reference-URL", indicates the URL of the cache on which the patch is based. It is designed for dynamic document caching and will be discussed in Section 2.3.3.

"Request-Patch" field, when present, indicates that the patch on the web object is requested. "Accept-Newversion" field, when present, indicates that the original web server can respond with the latest web object when the original server does not hold the requested patch. The value of the "Max-Length" field is the maximum patch length accepted by the client. The values of the "Old-Version" field and the "New-Version" filed are the modification dates of the patchable stale files and the new web objects respectively.

"Response-Patch" field, when present, indicates that the response message is a patch on the web object. "Response-Newversion" field, when present, indicates that the response body is the latest version of the web object. "Non-Patch" field, when present, indicates that the original web server does not hold the desired patch. "Patch-Held" field, when present, indicates that the original server maintains the patch for the web object. "Zero-Patch" field, when present, indicates that there is no difference between the up-to-date file and the old version identified by "Old-Version". The value of the "Patch-Length" filed is the length of the response patch. "Instruction-Set" fields describe the instruction set used in the patch.

The patch exchange information is included in header fields only, there is no need to use new HTTP commands besides "GET" and "HEAD". A "HEAD" request is considered as a patch query, while a "GET" request is a patch request.

2.3.2 Transparent patch communication to intermediate cache proxies

In this proposal, the original server is the only source of the patches for the web objects that originate from it. Thus the “patch consistency” problem would not arise. To achieve this, it is necessary to make patch communication transparent to intermediate cache proxies.

- The intermediate cache proxy, if any on the path, should forward the patch request to the original server.
- The response of a patch request is non cacheable at the intermediate cache server.

In the proposed system, we utilize the “cache-control: no-cache” header fields supported by HTTP/1.1 [55]. Since HTTP/1.0 may not recognize and obey the directive of “cache-control: no-cache”, we also use a program directive, “Pragma: no-cache” header field, which has the same semantics as the no-cache cache-directive and is defined to be backwards compatible with HTTP/1.0 [55]. According to [55], the intermediate dedicated cache server, if any, would not use a cached copy to response to such a request, and it should forward the patch request to the original server. A response with “no cache” header is considered as uncacheable at the intermediate cache proxy. The incremental update and delivery scheme can thus be implemented without modifying the intermediate cache servers.

The following is an example of a patch request header.

```
GET webfiles/index.html http1.1 CRLF
...
Cache-control: no-cache CRLF
Pragma: no-nache
Patch-control: Request-Patch CRLF
Patch-control: Accept-Newversion CRLF
Patch-control: Old-Version = Thu, 01 Dec 1994 16:00:00 GMT CRLF
Patch-control: Max-Length = 2000 CRLF
...
CRLF
```

Due to the presence of "no-cache" header fields, intermediate HTTP/1.0 and HTTP/1.1 cache proxies will forward such a patch request to the original server.

Similarly, S-DWEBC at the original server includes the "no-cache" header fields in the patch response header. The intermediate cache proxy, if any, would not save the patch and make the original server the only source of the patch. The following is an example of such a response header:

```
HTTP/1.1 200 OK CRLF
...
Cache-control: no-cache CRLF
Pragma: no-cache
Patch-control: Response-Patch CRLF
Patch-control: Old-Version = Thu, 01 Dec 1994 16:00:00 GMT CRLF
Patch-control: New-Version = Thu, 28 Feb 2002 19:07:23 GMT CRLF
Patch-Length: 100 CRLF
...
CRLF
```

When the original web server does not have any patch for a file, it puts the latest version of the file in the response body. One example of such a response is:

```
HTTP/1.1 200 OK CRLF
...
Patch-control: Response-Newversion CRLF
Patch-control: Non-Patch CRLF
Patch-control: New-Version = Thu, 28 Feb 2002 19:07:23 GMT CRLF
...
CRLF
"Latest Version File"
```

Note that this response is cacheable at the cache servers. This allows the cache server to refresh its cache storage.

2.3.3 Dynamic document support

As web servers become more sophisticated and customizable, the number of "dynamic documents" on the web increases steadily. Although all web documents can be considered as dynamic since they may change from time to time, we specifically treat the following cases.

- Dynamic document in time domain. The information data on an original server changes so fast in time that the response may change upon every access. Examples are weather forecast pages and stock price pages.
- Dynamic document in requesting client domain. The response is personalized based on user inputs. The same server process may deliver different documents for different users. One example is the result of a search engine [62].

Usually, cache proxies do not cache a dynamic document as it is likely that the original server delivers different documents for later requests. However caching a dynamic document still makes sense if parts of it are static and not changed across many versions and the dynamic part is only a small part of the whole document. One example is the New York stock exchange page, `http://nyse.com/marketinfo/marketinfor.html`. Its layout, which corresponds to the information rendering specification data in HTML format, has not changed for many months and the changed portion, which is the price information data, is only about 462 bytes out of the 17,610 bytes.

[63] proposed a dynamic document caching scheme, referred to as the active caching scheme. In the active caching scheme, cache proxies fetch the corresponding applets from the original servers as well as the requested documents. A subsequent cache hit will invoke the applet and the applet in turn instructs the cache proxy to request the original server for a new document or to send back the cached copy. This effectively migrates parts of server processing to cache proxies. The active caching scheme enables customization in caching. However, it only addresses the

case that documents are dynamic in the requesting client domain. In the time domain, applets at cache proxies still have the consistency problem as other cached documents.

In this thesis, we do not use applets to support dynamic document caching. We resolve the problem under the incremental update scheme in a unified way regardless of the domain where the document is dynamic. We assign the original server a key role in dynamic document caching. The original server decides the cacheability of a dynamic document and provides the patch for the cached pages.

In the following cases, the original server may mark a dynamic document as uncacheable by utilizing "cache control" directives [64].

- The transparency of the document semantic is deemed necessary by the service author.
- The dynamic document contains confidential or private information.
- Only very small part of the dynamic document is static across versions.

From the perspective of the client, there is no difference between requesting a static document and requesting a dynamic document. The key question is if a cached copy and the corresponding patch are available.

The original server on the other hand treats static and dynamic documents differently. For static documents, the patches are generated offline, and the original server maintains patches for a certain number of old versions (see Section 2.3.4). For dynamic documents, which have a high variability, it is not practical for the original server to maintain patches for different versions. It is expected that a patch is generated online upon each request. To generate a patch for a dynamic document online, the original server needs to determine the format of the cached dynamic document. This information should be provided by the requesting client. This is possible if the old version of the document is already available at the requesting client. If the document is dynamic in time domain, the time stamp of the cached dynamic document is required. If the document is dynamic in the requesting client domain, the user inputs that generate the set of results in the cached document is required. Usually nonconfidential user inputs appear in the URL forwarded to the server. In fact, the information in this URL is sufficient for the original server to reconstruct the cached copy of the document. In such a case, the URL consists of two parts. The first part is the address of the pointer to the main object that the user wants to retrieve. The second part is the customization parameters that allow the server to customize the actual information sent. Here we refer to the first part as the prefix of the URL. From Section 2.3.1, the "Reference-URL" header field is designed to exchange the user inputs information among the original server and clients.

In a patch response header, the "Reference-URL" header field contains information for a receiving C-DWEBC module to determine the kind of request that can be mapped into this response as a cache hit. In a patch request header, the "Reference-URL" header field contains information for the original server, on the complete and original URL of the cached document that the requesting client has. Thus the original server can determine the cached dynamic document based on the user inputs in the reference URL.

The following procedures depicted in Fig. 2.4 give an illustration on how a dynamic document is cached and served to peer clients in the incremental update

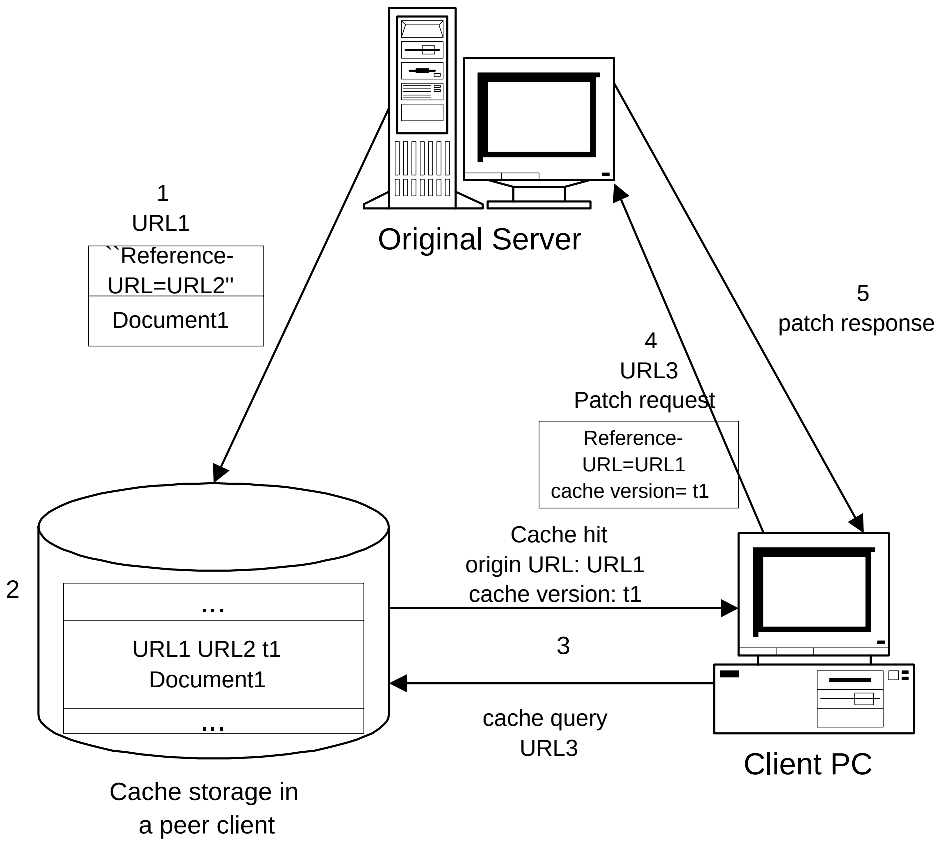


Figure 2.4: Cache and patch of dynamic document in PDWCIUS

and delivery scheme. Each of the Steps 1 to 5 is described as follows.

Step 1. When the original server delivers a cacheable dynamic document with a URL, say URL_1, to a client, it constructs a reference URL, say URL_2. URL_2 can simply be URL_1 or the prefix of URL_1. It is included in the response header as "Reference-URL = URL_2". It tells the receiving client that this response can be considered the cache of the document requested with a URL = URL_2 in the case that URL_2 contains user inputs, or a URL whose prefix is URL_2 in the case that URL_2 has no use inputs.

Step 2. A C-DWEBC module caches the response with two matching URLs, URL_1 and URL_2, together with the response's time stamp, t_1.

Step 3. A cache-query at a URL of URL_3 arrives. If URL_3 is the same as URL_2, or in the case that URL_2 has no use inputs, URL_3's prefix is URL_2, it is considered a cache hit. The original URL, that is URL_1, and time stamp t_1, are included in the cache-hit packet.

Step 4. A patch request is sent to the original server. In the header, "Reference-URL = URL_1" header field and time stamp t_1 are included.

Step 5. With the reference URL_1 and the time stamp, t_1, the original server determines what the stale cached copy is and computes the corresponding patch for URL_3.

The real-time patch generation for dynamic documents will be discussed in Chapter 6.

2.3.4 Patch version maintenance and cache replacement

For static documents that have a low update frequency, patches can be generated off-line. Original server needs to maintain patches for web objects.

A simple and commonly used Least Recently Used (LRU) algorithm can be used in both the cache and patch replacement scheme [65, 66, 67]. We present here methods to further enhance the performance of the LRU algorithm.

The original server maintains various versions of patches for one web object to serve clients with various versions of stale files. The inconsistent caches at the clients and the inconsistent patches at the server use up valuable resources. It is thus important to have efficient patch version maintenance and cache replacement

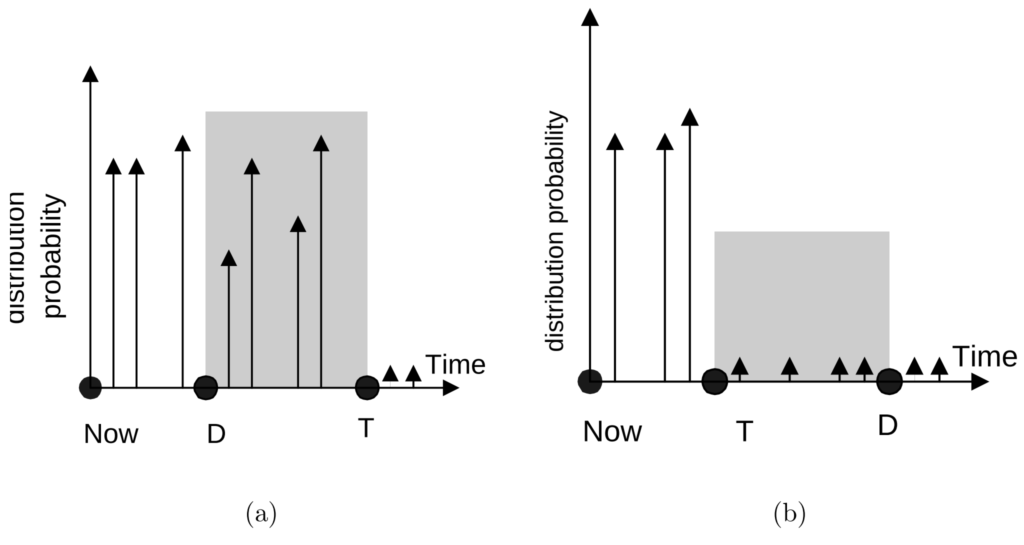


Figure 2.5: Cache distribution over time

policy to ensure that the patchable cache version bound and the in-request patch version bound are synchronized.

Let D, a date value, be the patch maintenance depth threshold for web object O. Let T, a date value, be the cache's LRU threshold. Good consistency between D and T saves resources and improves performance. Fig. 2.5 illustrates the distribution of caches and situations where a gap between D and T is present.

In Fig. 2.5(a), the patch maintenance depth is shorter than the cache replacement threshold. The cached documents with version between D and T (shadowed) are unpatchable but kept at clients. It wastes storage resources and results in patch misses. Since the cache replacement threshold, T, applies to all local caches from various sources with different Ds, it is not practical to change T to cope with a particular D. One possible method to flush off the unpatchable cached documents is for the client to uses a "group cache replacement protocol". In this protocol, after a request for O is fulfilled, the requesting client sends out at an appropriate time a multicast cache replacement request with D included. The peers then utilize D to invalidate their cache storage.

In the case illustrated in Fig. 2.5(b), the cache replacement threshold is shorter, and some patches in the shadowed region will never be requested but nevertheless kept at the server. Since the patch maintenance depth, D applies to all patches provided to various clusters with various T, it is not practical to change D to cope with a particular T. One possible method to flush off the non-required patches, is for the original server to employ a Least Frequently Used (LFU) method besides LRU [68, 69]. Those non-required patches will have a requesting rate that gets lower with time. The patches with request rate lower than a relative or absolute threshold are flushed off.

2.4 Benefits

In [58], the potential benefits from HTTP delta encoding was concluded. However, [58] did not consider the particular caching scheme where the HTTP delta encoding is embedded. This section analyzes the benefits arising from the peer distributed caching together with incremental update scheme.

2.4.1 Hit rate of the web-caching System

In a web caching system, a cache server receives the request for a web object from the client that it serves. If there is a valid cached copy in the server's cache storage, a “cache hit” occurs; otherwise, a “cache miss” occurs. There are four sources of cache miss [35, 70]. They are compulsory misses, which occur for a first access to an object; capacity misses, which occur when the client requests an object that has been discarded due to space constraints; consistency misses, which occur when the cache server holds a stale copy of the file; and uncacheable/error misses. The incremental update scheme allows the stale cached copies to be served to clients

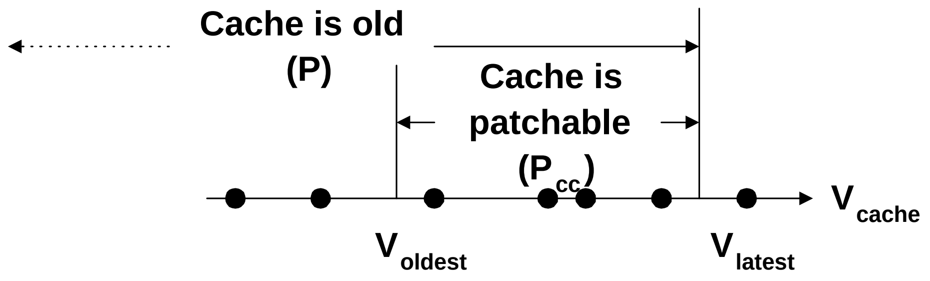


Figure 2.6: Probability of stale cached copy and patchable cached copy

as long as a corresponding patch is available at the server. In this case, no consistency cache miss occurs. Incremental and update scheme relaxes the criteria of an inconsistent cache. Thus the usage of cache storage and the overall hit rate of the web caching system increase.

The next two sections examine the proposed system in terms of traffic load and response time under the following three cases.

1. Cached copy is available locally.
2. Peers hold the cache.
3. Cached copy is not available in the cluster.

We first define some notations that will be used in the next two sections.

Let P depicts the probability that a request encounters an update, S_p depicts the size of the patch for a request and S_o depicts the size of the up-to-date original file. In Section 2.2.1, P_{cc} is defined as the probability that a cached copy is consistent in the proposed system. The proposed system relaxes the consistency criteria. An old but patchable cache is still consistent. Thus $P \geq P_{cc}$. Figure 2.6 illustrates the relationship between P and P_{cc}.

2.4.2 Traffic on the inter-cluster and intra-cluster networks

In this thesis, we view each network in Fig. 2.7 [6] as a cluster in Fig. 2.8. Peer clients are assumed to be in the same cluster. The communication with the original web server is through the inter-cluster network. Usually, a dedicated cache proxy is deployed at the edge of a network as shown in Fig. 2.8. It is assumed that the dedicated cache proxy and the client browser in the cluster use the "client-polling-every-time" consistency policy as the proposed system does. The size of the query and response data is depicted as S_{qr}.

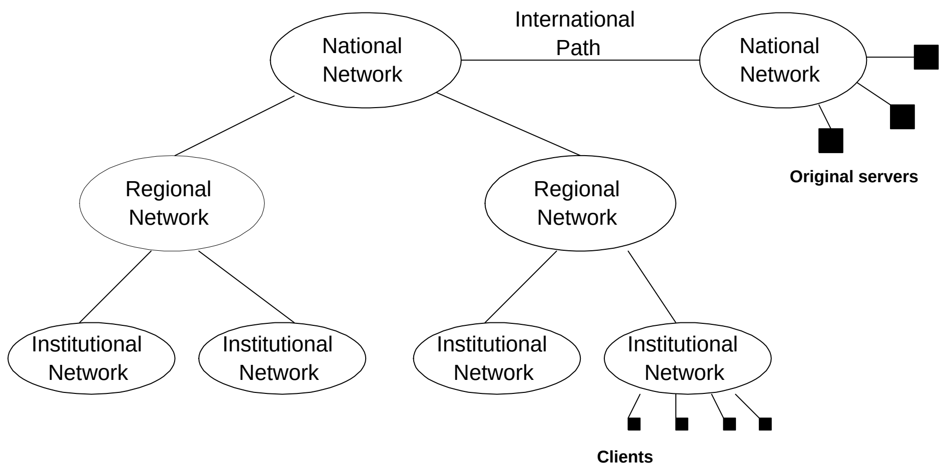


Figure 2.7: Network topology

The traffic load is directly related to the size of the data transmitting over the network. Let $S_{InterDWEBC}$ and $S_{IntraDWEBC}$ be the expected size of the data on the inter-cluster network and intra-cluster network respectively for a request when the proposed system is deployed. Let S_{Inter} and S_{Intra} be the expected size of the data on the inter-cluster network and intra-cluster network for a request respectively when the proposed system is not deployed. Let S_{pi} be the size of the data for patch information exchange for a request, and S_{ci} be the size of a multicast

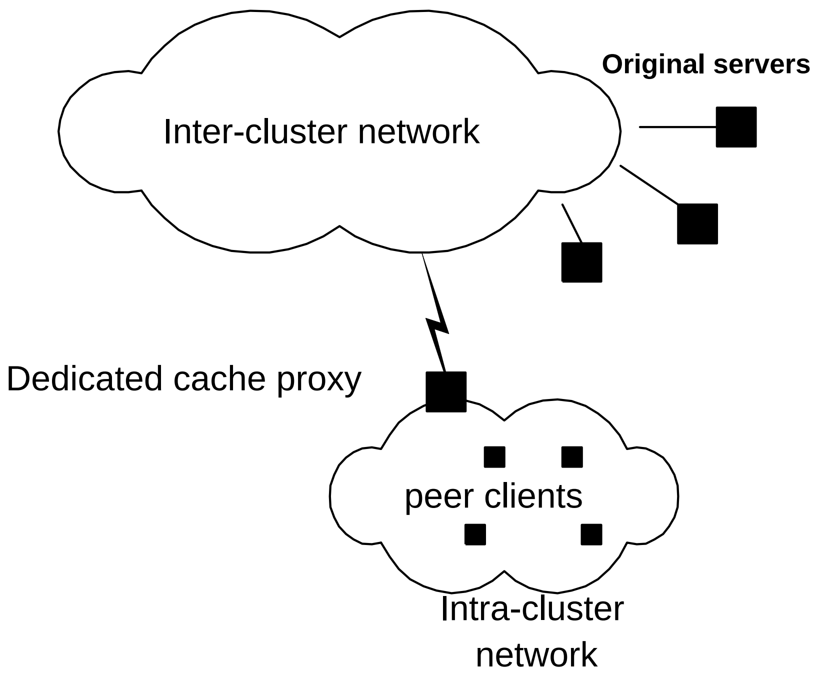


Figure 2.8: A simple network topology

packet for cache information exchange. We shall now compare the inter- and intra-cluster traffic for the situation where the proposed system is not deployed, and the situation where the proposed system is deployed, under the three cases in Section 2.4.1

Case 1: local cache is available

If the proposed system is not deployed, the existing web browser caching scheme works as follows. The browser queries the original server to check the local cache's freshness. If the web object has been updated, the request is fulfilled by an updated file from the original server. Otherwise, the local cache is used. Under this situation, the expected sizes of S_{Intra} and S_{Inter} are the same, given by

$$S_{Inter} = S_{Intra} = S_{qr} + P \cdot S_o. \tag{2.7}$$

In the situation where peer distributed web caching system is deployed, the requesting client would ask for patch information from the original server, to check the consistency of the local cached copy. If the local cache is patchable, the request is fulfilled by a patch from the original server. If the local cache is not patchable, the updated version is fetched from the original server. Under this situation, the expected sizes of $S_{IntraDWEBC}$ and $S_{InterDWEBC}$ are the same too, given by

$$S_{InterDWEBC} = S_{IntraDWEBC} = S_{pi} + P_{cc}S_p + (P - P_{cc})S_o. \tag{2.8}$$

In our proposed protocol, the patch information exchange data is included in the HTTP header, in the form of several header fields, and the cache query and reply messages are multicast packets. All of them are small in size compared with the cache or the original file. If we simply let S_{qr} equal S_{pi}, then since $P \geq P_{cc}$, we have

$$\begin{aligned} S_{InterDWEBC} &\leq S_{Inter} \\ S_{IntraDWEBC} &\leq S_{Intra}. \end{aligned} \tag{2.9}$$

The reduced intra-cluster and inter-cluster traffic are achieved in peer distributed web caching system under Case 1.

Case 2: peers hold the cache

If the proposed system is not deployed, the caches in peer clients are not accessible. It is assumed that the dedicated cache proxy has a cached copy. The dedicated cache proxy queries the original server to check the cache's freshness. If it is fresh, the cached copy is sent to the requesting client. Otherwise, a new file fetched from the original server is sent to the requesting client.

$$\begin{aligned} S_{Intra} &= S_{qr} + P \cdot S_o + (1-P)S_c \\ S_{Inter} &= S_{qr} + P \cdot S_o \end{aligned} \tag{2.10}$$

In the peer distributed web caching system, the requesting client asks for patch information from the original server and the cache information from the peers to check the cache's consistency. The cache information exchanges involve N_m multicast packets within the cluster. If the cached copy is still patchable, the request is fulfilled by a patch from the original server and the cached copy from the peer. If the cached copy is not patchable, the updated version is fetched from the original server.

$$\begin{aligned} S_{IntraDWEBC} &= S_{pi} + N_m S_{ci} + P_{cc}(S_p + S_c) + (P - P_{cc})S_o + (1-P)S_c \\ S_{InterDWEBC} &= S_{pi} + P_{cc}S_p + (P - P_{cc})S_o \end{aligned} \tag{2.11}$$

As in Case 1, if $S_{qr} = S_{pi}$, the proposed system achieves the reduced inter-cluster traffic. However, this reduction is achieved at the cost of the increased intra-cluster traffic. This increase is due to the multicasted "cache-query"/"cache-hit" and the patch. Fortunately, compared with the inter-cluster connection, the intra-cluster connection is less likely to be the network bottleneck.

Case 3: cache is not available in the cluster

When the proposed caching system is not deployed, it is assumed the dedicated cache proxy serving this cluster has a cached copy, and we have the same traffic load as in Case 2.

$$\begin{aligned} S_{Intra} &= S_{qr} + P \cdot S_o + (1-P)S_c \\ S_{Inter} &= S_{qr} + P \cdot S_o \end{aligned} \tag{2.12}$$

In the peer distributed web caching system, the requesting client multicasts a "cache-query", and received no "cache-hit" by the time the timer expires. It thus asks for the updated web object from original server.

$$\begin{aligned} S_{IntraDWEBC} &= S_{pi} + S_{ci} + S_o \\ S_{InterDWEBC} &= S_{pi} + S_o \end{aligned} \tag{2.13}$$

From Equation 2.12 and 2.13, we can see that the proposed system leads to increased traffic load when no cached copy is available in the system. The reason is that the updated file request from the proposed system is not served by the outside cache. It goes to the original server directly. Moreover, this case can be avoided once a fresh copy is fetched and shared within the system.

2.4.3 Response time that the client experiences

Figure 2.8 models the Internet as a combination of the intra-cluster and inter-cluster networks from the perspective of clients. A dedicated cache server serves the cluster where the clients reside. It is assumed that the dedicated cache proxy and the client browsers in the cluster use the "client-polling-every-time" consistency policy.

The patch transmission and the original file request from the peer distributed caching system are designed to be transparent to any external caching system. In this thesis, we use the transmission rate r_o to evaluate the communication between the client within the cluster and the original server situated outside of the cluster. Let r_c depict the transmission rate within the cluster. Let T_{DWEBC} be the expected response time for a request when the incremental update scheme is deployed, and $T_{NoDWEBC}$ be the expected response time when the proposed peer distributed caching system is not deployed. Since patch decoding can be done concurrently with the reception of the files, decoding time is ignored. Let us examine the following three cases.

Case 1: local cache is available

If the proposed system is not deployed, upon a request for a web object, and upon determining that the object is in the local cache, the web browser would query the original server to validate the cache's freshness. The validation time at the client is $\frac{S_{qr}}{r_o}$. If the web object had been updated, the fresh version of the object is fetched from the original server. Otherwise, the version in the local cache is used. Retrieving a local cached copy is fast and the local access time can be ignored. Therefore the expected response time is given by

$$T_{NoDWEBC} = \frac{S_{qr}}{r_o} + P\frac{S_o}{r_o}. \tag{2.14}$$

On the other hand, with the proposed system in place, the requesting client requests patch information to validate the consistency of the local cached copy. The transmission time for receiving information on the patch is $\frac{S_{pi}}{r_o}$. If the local cache is still patchable, a patch is then needed. In this case, we then have

$$T_{DWEBC} = \frac{S_{pi}}{r_o} + P_{cc}\frac{S_p}{r_o} + (P - P_{cc})\frac{S_o}{r_o}. \tag{2.15}$$

Suppose $S_{qr} = S_{pi}$, the response time improvement in Case 1 is given by

$$T_{NoDWEBC} - T_{DWEBC} = P_{cc}(\frac{S_o - S_p}{r_o}). \tag{2.16}$$

Since the patch is smaller than the original file, $(T_{NoDWEBC} - T_{DWEBC})$ is positive and thus a shorter response time can be claimed in Case 1.

Case 2: peers hold the cache

In Case 2, without the proposed system, the requesting client would ask for the updated web object from outside of the cluster. It is assumed that the dedicated cache proxy serving this cluster has a cached copy. It queries the original server to check the cache's freshness. The validation time at the dedicated cache proxy is $\frac{S_{qr}}{r_o}$, giving

$$T_{NoDWEBC} = \frac{S_{qr}}{r_o} + P\frac{S_o}{r_o} + (1-P)\frac{S_c}{r_c}. \tag{2.17}$$

For Case 2, with the proposed system in place, the requesting client requests for patch information and simultaneously multicasts a "cache-query" request. It also sets a timer of T_c for "cache-hit" response. The actual wait time is depicted as T_w, and $T_w \leq T_c$. If the cache is patchable, the patch connection and the cached connection are performed simultaneously. The longer of the two times, the cache response time and the patch response time, contributes to the response time. If the cache is not patchable, a new fresh file is fetched from outside of the cluster. If the cache is fresh, it is accepted and the patch connection is aborted. We then have

$$T_{DWEBC} = T_w + P_{cc}\max\{\frac{S_c}{r_c}, \frac{S_p}{r_o}\} + (P - P_{cc})\frac{S_o}{r_o} + (1-P)\frac{S_c}{r_c}. \tag{2.18}$$

Since timer T_c is set not to wait longer than the direct file fetching time (Equation 2.1), we then have $T_{DWEBC} \leq T_{NoDWEBC}$.

Suppose $T_w = \frac{S_{qr}}{r_o}$, the response time improvement in Case 2 is

$$T_{NoDWEBC} - T_{DWEBC} = P_{cc}(\frac{S_o}{r_o} - \max(\frac{S_c}{r_c}, \frac{S_p}{r_o})). \tag{2.19}$$

Equation 2.16 and 2.19 show that in Case 1 and 2, the three factors, namely the availability of more cached copies or patchable versions (P_{cc}), the smaller patch and the wider intra-cluster bandwidth lead to better improvement in response time.

Case 3: cache is not available in the cluster

In Case 3, when the proposed caching system is not deployed, we have the same response time as in Case 2, given by

$$T_{NoDWEBC} = \frac{S_{qr}}{r_o} + P\frac{S_o}{r_o} + (1-P)\frac{S_c}{r_c}. \tag{2.20}$$

On the other hand, with the peer distributed web caching system, the requesting client needs to wait for T_c to make sure that no cache within the cluster is available, before asking for the updated web object from outside. We then have

$$T_{DWEBC} = T_c + \frac{S_o}{r_o}. \tag{2.21}$$

From Equation 2.20 and 2.21, we can see that the proposed system takes more time to get an updated file when there is no cache available within the system. However, once an updated version is fetched and shared within the cluster, Case 3 can be avoided subsequently.

2.4.4 Real-time independent patch decoding

During the reception of the requested objects, the client may receive the patch and an old cached copy via two connections simultaneously. It is desirable that the patch decoding routine can function while the files are in transmission, and not only after the files are received completely (Fig. 2.9). This is achieved in our proposed system with an appropriately designed encoding format.

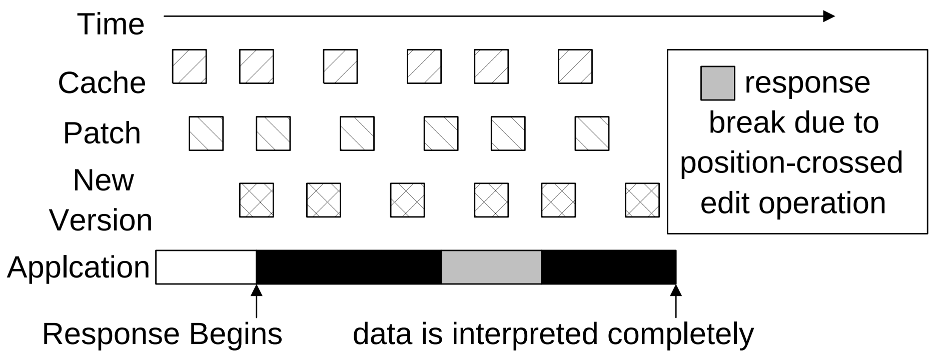

Time
Cache
Patch
New Version
Applcation
response break due to position-crossed edit operation
Response Begins
data is interpreted completely

Figure 2.9: Data converting delay

In the next three chapters of this thesis, we will discuss patch generation in tree space in detail. Briefly, we transform a web object to a tree in pre-order (the root first, then sub trees in pre-order from left to right) format. The tag data indicating the beginning of a hierarchical structure is transformed into the root of a subtree. The data embedded in the structure is transformed into nodes in the subtree in a way such that the forth-back relationship is transformed into the left-right sibling relationship (Fig. 2.10).

The nodes in the tree are indexed in ascending pre-order format. Consequently, the edit operations in patches are sent in the ascending order of the index of the involved nodes. In a tree, a parent node has a smaller pre-order index than its descendants, and the left sibling has a smaller pre-order index than right siblings. The nodes in the ascending pre-order format correspond to data blocks in the forth-back order in the web object file (Fig. 2.11). Thus, the transmission order of stale file data is consistent with the transmission order of the edit operations. Concurrent reception and decoding can thus be achieved.

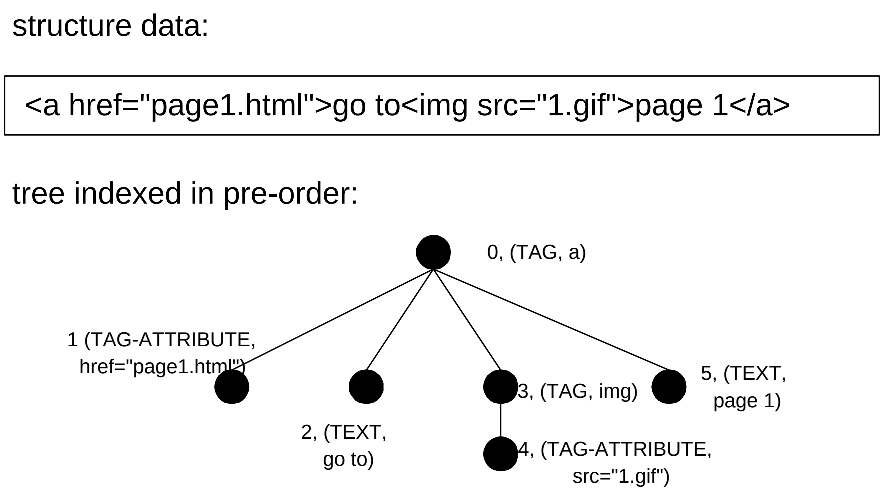
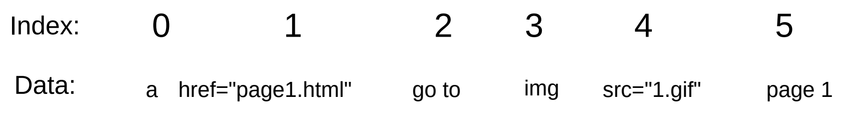

structure data:
<a href="page1.html">go to<img src="1.gif">page 1</a>
tree indexed in pre-order:
0, (TAG, a)
1 (TAG-ATTRIBUTE, href="page1.html")
2, (TEXT, go to)
3, (TAG, img)
4, (TAG-ATTRIBUTE, src="1.gif")
5, (TEXT, page 1)

Figure 2.10: Example of transforming hierarchical data into tree



Index:	0	1	2	3	4	5
Data:	a	href="page1.html"	go to	img	src="1.gif"	page 1

Figure 2.11: Nodes in ascending pre-order index and data corresponding to them

2.5 Cache-hit flood and scalability

The benefits discussed above are achieved at the cost of the time to generate patches and the increased intra-cluster traffic. The increased intra-cluster traffic may affect the system effectiveness.

The multicasted “cache-query” and “cache-hit” packets among peer C-DWEBC modules are the newly introduced traffic cost. A flood of multicast packets is not desirable. C-DWEBC is designed under the following two guidelines.

1. The probability of multiple “cache-hit” packets responding to one “cache-query” is minimized. Note that since a cache-query reaches every peer client,

it is possible that each client responds with a cache-hit for each cache query.

2. The probability that a responding peer computer issues a "cache-hit" while the requesting client has given up is minimized.

As shown in Section 2.2.2, C-DWEBC uses a "cache-hit" packet to find and remove from the queue those cache queries that have been responded to. This helps to reduce the unnecessary multiple "cache-hit" for one single "cache-query" (Guideline 1). But if the cache-hit that is supposed to invalidate local "cache-queries" comes during or after the processing of the "cache-query", more than one cache-hit may appear for one cache-query. Let q_c be a "cache-query" in the queue on a peer computer. Since it is a multicast message, we assume that it exists in the queues of other cooperative peers. Let t_0 be the time when q_c is processed first by one of those peers and T_{p0} be the corresponding process time. A peer will issue another "cache-hit" if it picks q_c to process from time t_0 to $t_0 + T_{p0}$. Let p_0 be the probability that a peer is free to process the service queue, p_1 be the probability that a peer holds a cached copy matching q_c, and p_2 be the probability that a peer picks q_c to process next. Under the assumption that the processing time of a "cache-query" on all peers are the same, then in a system with N peer computers, the probability of a burst of m "cache-hit" addressing the same cache-query, q_c, is

$$P\{m\,burst\,for\,q_c/t_0\} = C_{N-1}^{m-1}(p_0p_1p_2)^{m-1}(1-p_0p_1p_2)^{N-m} \tag{2.22}$$

Let N_m be the number of multicast packets including "cache-query" and "cache-hit" incurred by a request. Its average, $\overline{N_m}$ is

$$\begin{aligned}\overline{N_m} &= 1 + \sum_{m=1}^{N} mC_{N-1}^{m-1}(p_0p_1p_2)^{m-1}(1-p_0p_1p_2)^{N-m} \\ &= 2 + (N-1)(p_0p_1p_2) \end{aligned} \tag{2.23}$$

Obviously, a smaller $p_0p_1p_2$ results in a smaller $P\{m\,burst\}$ (Guideline 1). However, a high p_0 and p_1 is desirable to improve caching performance. As for p_2, if we pick packets in the queue in some regular order (e.g. FIFO and LIFO), then there is a high probability that the same packet is selected by two or more peer computers at the same time, leading to a high probability of a hit flood. To address this issue, we select the next packet in a random fashion, therefore reducing the probability that two or more computers select the same packet at the same time.

To achieve Guideline 2, C-DWEBC uses T_c' to estimate T_c. If the cache query is processed before time instant, $t_{receive} + min(T_c, T_c')$, the accuracy of the estimation has no impact. The responding peers issue the cache-hits that are expected and the requesting peer will not wait for a cache-hit that anyway will not be issued. The same is in the ideal case that T_c' is equal to T_c. If the cache query is processed after the time instant, $t_{receive} + max(T_c, T_c')$, the accuracy of estimation does not matter either, since the requesting peer would have given up and the responding peers would have ignored the cache query. In the case that $T_c' > T_c$ and the cache query is processed after time instant $t_{receive} + T_c$ but before time instant, $t_{receive} + T_c'$, a cache-hit will be sent. However, the requesting client has timed out and switched to the original server. In this case, the cache-hit is simply ignored at the requesting client. It will also serve to flush off matching cache queries on other peers. In the case that $T_c' < T_c$ and the cache query is processed after time instant, $t_{receive} + T_c'$ but before time instant, $t_{receive} + T_c$, no cache-hit will be sent, although the requesting client would still be waiting. However, it is possible that another peer may have a larger T_c' and will subsequently issue a cache-hit message.

Equation 2.23 shows that the probability of a cache-hit burst increases when the system grows. Thus when the peer distributed system is deployed in a huge cluster or across many hops, the multicasting "cache-hit" burst may result in the intra-cluster congestion and impair the response time improvement as shown in Equation 2.16 and 2.19.

As discussed above, the C-DWEBC module in the responding peer estimates T_c to avoid issuing unwanted "cache-hit" messages, and processes the "cache-query" queue randomly to reduce the probability of a "cache-hit" burst. These measures decrease the number of the multicasting "cache-hit" within the cluster and improve the system's scalability.

2.6 Service reliability

As shown in Section 2.2, the cache server function provided by C-DWEBC is assigned the lowest scheduling priority at the client computer. It would be scheduled out when a higher priority local task is created. Thus the cache server service on one single client computer cannot be guaranteed. However, if multiple client computers cooperate to provide cache service, reliability is improved.

When a client issues a local web request in the cluster, the cooperating peer clients can be viewed as special "cache servers" providing two kinds of service. One is to answer cache query and the second is to answer cache request. The "server" has fulfilled a cache server task if it is accessible during both service requests. We thus define that for a web request, the "server" is considered accessible only if the following two requirements are met.

1. During the requesting client's waiting period, at least one peer client has its C-DWEBC module on. Thus, the cache query message can be processed.

2. Among those clients meeting the above requirement, at least one has its C-DWEBC module on during the subsequent cache transmission period. Thus, the cache, if any, can be sent to the requesting client.

The above two requirements do not guarantee service performance, but if they are not satisfied, then it is not possible for the "server" to provide a cached document and the "server" is considered down.

Let $P_{ServerOn}(n)$ be the probability that the "server", which consists of n peers, is accessible when a web request arrives. Let p_{d1} be the probability that the C-DWEBC is down at a peer client during the requesting client's waiting period, and p_{d2} be the probability that the C-DWEBC is down at a peer client during the subsequent supposed cache transmission period. Since a C-DWEBC module can be shut down anytime, we simply assume p_{d1} and p_{d2} to be independent. Based on the above two requirements, we have

$$P_{ServerOn}(n) = \sum_{i=1}^{n} C_n^i (1-p_{d1})^i {p_{d1}}^{n-i} (1-{p_{d2}}^i). \tag{2.24}$$

Because $C_n^i = C_{n-1}^i + C_{n-1}^{i-1}$ $(1 \le i \le n-1)$, we have

$$\begin{aligned} P_{ServerOn}(n) &= \sum_{i=1}^{n-1} C_{n-1}^i (1-p_{d1})^i {p_{d1}}^{n-i} (1-{p_{d2}}^i) + \\ &\quad \sum_{i=1}^{n-1} C_{n-1}^{i-1} (1-p_{d1})^i {p_{d1}}^{n-i} (1-{p_{d2}}^i) + \\ &\quad (1-p_{d1})^n (1-{p_{d2}}^n) \\ &= p_{d1} \cdot P_{ServerOn}(n-1) + (1-p_{d1}){p_{d1}}^{n-1}(1-p_{d2}) + \\ &\quad \sum_{i=1}^{n-1} C_{n-1}^i (1-p_{d1})^{i+1} {p_{d1}}^{n-1-i} (1-{p_{d2}}^{i+1}). \end{aligned} \tag{2.25}$$

Because $p_{d2} \leq 1$, so $1 - {p_{d2}}^{i+1} \geq 1 - {p_{d2}}^{i}$, and we have the following inequality.

$$\begin{aligned}
P_{ServerOn}(n) &\geq p_{d1} \cdot P_{ServerOn}(n-1) + (1-p_{d1}){p_{d1}}^{n-1}(1-p_{d2}) + \\
&\quad \sum_{i=1}^{n-1} C_{n-1}^{i}(1-p_{d1})^{i+1}{p_{d1}}^{n-1-i}(1-{p_{d2}}^{i}) \\
&= p_{d1} \cdot P_{ServerOn}(n-1) + (1-p_{d1}){p_{d1}}^{n-1}(1-p_{d2}) + \\
&\quad (1-p_{d1})P_{ServerOn}(n-1) \\
&\geq P_{ServerOn}(n-1) \qquad (2.26)
\end{aligned}$$

Inequality 2.26 shows that the server reliability can be improved by redundancy.

2.7 Chapter summary

This chapter proposes a peer distributed web caching system with incremental update and delivery scheme. This proposal utilizes the perishable computational power and cache storage on nearby peer clients to achieve a pooled large cache storage at a close range. It also utilizes the fact that the majority of web page updates are minor, so that small patches may be generated to update stale cached objects, thereby improving the "cache hit" rate. Such benefits are achieved at the cost of patch computation and increased intra-cluster traffic. This chapter suggests measures to limit the volume of newly introduced intra-cluster traffic to improve the system's scalability. The service reliability is also analyzed in this chapter.

Chapter 3

Web object to tree conversion

In this chapter, the preliminary background information required for the understanding of generation of patches is described. The approach adopted in this thesis, towards the solution of the patch generation problem, is to first convert a web object into a tree. Once it is recast into a tree, the patch generation problem is then equivalent to a tree correction problem.

3.1 Unification of web object files

There are many common file types available on the WWW. Examples are HTML file [71], XML file [72], plain text file, image file and video file. According to [73][74][75], files can be classified into three categories; structured, semi-structured and unstructured.

- **Structured file:** Structured files have strict inner structure. An example is the representation of the content of a database table such as BibTex [76].
- **Semi-structured file:** Semi-structured files have specific data format. The structure is not as rigid, regular or complete as that of the structured file.

An example is the HTML file [77, 78].

- **Unstructured file:** Unstructured files have no specific semantics of the data stored. Examples are plain text files.

The above classification is made based on the links among the semantics of the data, the syntax of the file and its regularity, completeness and volatility of the structure. If we are not concerned about the semantics of the data and do not distinguish the regularity of the structure, then in general, one file can be thought of containing both structured data and unstructured data.

For structured data, the ancestor-descent and sibling relationship can be used to transform it into a tree. For unstructured data, structure can be added by simply treating it as an ordered queue of bytes with one virtual root. Thus unstructured data can be transformed into a tree with a depth of two.

3.2 Transforming web object to ordered labelled tree

In this thesis, we use an ordered labelled tree to represent a web object. An ordered labelled tree, T, is defined as follows [79, 80, 81].

T, is a finite nonempty set of vertices with a labeling function such that:

1. T has a distinct vertex, the root of the tree.
2. The remaining vertices (excluding the root) are partitioned into disjoint sets, and each of these sets is a tree, or sub-trees of T.
3. Associated with each vertex, is a label.

4. The ancestor (parent-child) relationship and the left-to-right ordering among siblings are significant.

Every node in the ordered labelled tree has a pre-order index and a postorder index shown below. The two indexes are used for different purposes as shown in Section 3.3.

```
PreOrderIndex($T$)
{
    Index($T$'s root)
    PreOrderIndex($T$'s left most subtree)
    ... // PreOrderIndex $T$'s subtrees from left to right one by one
    PreOrderIndex($T$'s right most subtree)
}
```

```
PostOrderIndex($T$)
{
    PostOrderIndex($T$'s left most subtree)
    ... // PostOrderIndex $T$'s subtrees from left to right one by one
    PostOrderIndex($T$'s right most subtree)
    Index($T$'s root)
}
```

In a tree, every node is again the root of a subtree unless it is a leaf, and the nodes beneath in the subtree are the descendants of the root.

The node in an ordered labelled tree is also assigned a label with two elements, object-type and object-content. Four object types are defined, namely TAG, TAG-ATTRIBUTE, VIRTUAL-ROOT and BYTE. The object content can be the tag name, the attribute of a tag, the octet value of a byte queue or NONE.

The hierarchical structure of a web object can be transformed into a tree as follows. A tag indicates the beginning of a hierarchical structure, and it is transformed to the root of a tree labelled as (TAG, tag name). This is followed by the tag attribute which is the left most child of the root. Other tags or plain text in the enclosed content are the right siblings of the TAG-ATTRIBUTE object. The enclosed content is further transformed into sub-trees in the same way and the order of the tag contents is preserved as the order of corresponding nodes or sub trees.

Fig. 3.1 illustrates an HTML link, which is a piece of data of hierarchical structure, and the corresponding tree constructed from it.

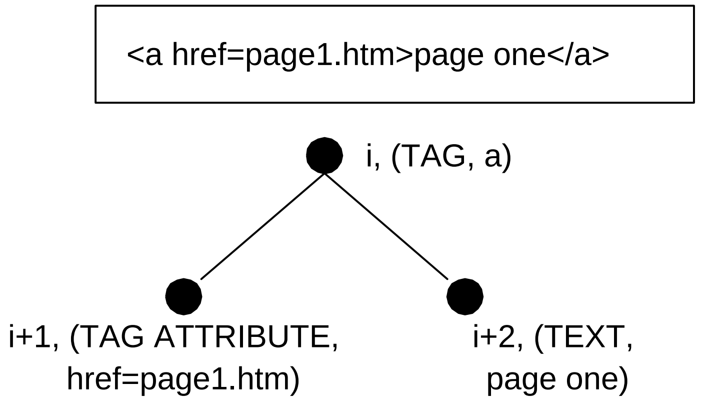

<a href=page1.htm>page one</a>
i, (TAG, a)
i+1, (TAG ATTRIBUTE, href=page1.htm)
i+2, (TEXT, page one)

Figure 3.1: A link in html and 3 nodes with consecutive pre-order index

Hierarchical data is transformed into a tree in pre-order. Unlike the well known “pre-order traversal”, this pre-order tree construction is not strictly for binary trees only. Pre-order construction constructs the root of a tree first, and then the sub-trees from left to right, each also in pre-order.

In this thesis, the root of an ordered labelled tree is always constructed as a

VIRTUAL-ROOT, below which a web object file is transformed into nodes. In this way, the trees derived from web object files have the same root. This conforms with Assumption 5.1 used in the patch generation algorithms in Chapter 5.

We use dynamic programming with recursion to transform a block of data or a string of bytes into a tree. The recursion function is called "ConvertStringToTree". It takes three inputs: a string to convert, the address of the last constructed node and the down or right relative position of the next node. The whole tree is constructed in a recursive way by giving the whole file, a root node and "down" as the initial input. The function, "ConvertStringToTree" is described below.

```
ConvertStringToTree
Input:
  1. a string to convert
  2. the address of the last constructed node
  3. the relative position of the next node (down or right)
Output:
  a labelled ordered tree
Implementation:
Call "LookForOneNodeContent" (see below) on the input string to get:
  1. the content for new node(s)
  2. the strings for the next search
If (the type of the new node is PURETEXT)
  Allocate memory for the node.
  If (the relative position is down)
    Construct a parent-child relationship between the new node and the last node.
  else
```

```
        Construct a sibling relationship between the new node and the last node.
    Call "ConvertStringToTree" with the following inputs:
        1. the next searching string
        2. the new node's address
        3. relative position of right
If (the type of the new node is a tag without end tag)
    Construct a new node from the tag name and allocate memory.
    If (the relative position is down)
        Construct a parent-child relationship between the new node and the last node.
    else
        Construct sibling relationship between the new node and the last node.
    If (tag attribute is available)
        Construct a new node from it and allocate memory.
        Construct a parent-child relationship between the tag name node and the tag
        attribute node.
    Call "ConvertStringToTree" with the following inputs:
        1. the next searching string
        2. address of the tag attribute node
        3. relative position of right
If (the type of the new node is a tag with end tag)
    Construct a node from the tag name and allocate memory.
    If (the relative position is down)
        Construct a parent-child relationship between the new node and the last node.
    else
        Construct a sibling relationship between the new node and the last node.
    If (tag attribute is available)
        Construct a node from it and allocate memory.
    Construct a parent-child relationship between the tag name node and the tag
    attribute node.
    Call “ConvertStringToTree” with the following inputs:
        1. nextString1 returned by “LookForOneNodeContent”
        2. address of the tag attribute node
        3. relative position of right
    Call “ConvertStringToTree” with the following inputs:
        1. nextstring2 returned by “LookForOneNodeContent”
        2. address of the tag attribute node
        3. relative position of right
  else
    Call “ConvertStringToTree” with the following inputs:
        1. nextString1 returned by “LookForOneNodeContent”
        2. address of the tag node
        3. relative position of down
    Call “ConvertStringToTree” with the following inputs:
        1. nextString2 returned by “LookForOneNodeContent”
        2. address of the tag node
        3. relative position of right
End ConvertStringToTree
```

The function, “LookForOneNodeContent” returns the string content for the next possible nodes. This function is described as below.

```
LookForOneNodeContent
Input:
  The string from which to look for a node.
```

Outputs:

1. The node found
2. the strings (up to 2) for next search
   (nextstring1 and nextstring2)

Implementation:

```
If (the searching range is invalid)
  Return.
Look for a valid tag by searching for and checking the indicating character and tag name.
If (no valid tag is found)
  Take the whole string as a PURETEXT node.
  Make the next search range invalid.
  Return.
If (a valid tag is found)
  Take the tag name and attribute as 2 nodes
  If (the tag has an ending tag)
    Two ranges are for next search:
        1. the substring within the tag
        2. the substring after the ending tag
  else
    Only one string for next search: the substring after the tag.
  Return.
End LookForOneNodeContent
```

The web object is recovered from the ordered labelled tree as follows. The content of the data piece in the web object is exactly the object-content label of the node in the tree. The data piece corresponding to the child node is embedded in the data piece corresponding to the parent node. The data piece corresponding to the left sibling node is located before the data piece corresponding to the right sibling.

Theorem 3.1. *The web object file and its image in the tree space have a one-to-one mapping relationship following the conversion methods described in Section 3.2.*

Proof. From the definition of the ordered labelled tree in Section 3.2, we can see that an ordered labelled tree is completely determined by the nodes' labels, the ancestor relationship among nodes and the left-right order of sibling nodes. A web object file is completely determined by the content of data pieces, the embedding of data pieces and the front-back order of sibling data pieces. In the conversion between the web object and the ordered labelled tree described in Section 3.2, the node label of the tree and the content of data piece of the web object file, the ancestor relationship among nodes and the embedding of data pieces, and the left-right order of sibling nodes and the front-back order of data pieces determine each other completely and uniquely. □

By converting a web object to an ordered labelled tree, the web patch problem is cast into a tree-to-tree correction problem.

Selkow [79] resolved the tree-to-tree correction problem by extending the string-to-string correction problem solution proposed by [82]. Three instructions ("change a node", "insert a node" and "delete a node") were used. Tai's algorithm [83] computes the globally minimal tree correction under the same three instructions by finding the minimal cost "mapping". Zhang and Shasha [1] improves Tai's algorithm by using post order numbering and achieves a smaller time complexity and space complexity that are $O(V \times V' \times min(l,p) \times min(l',p'))$ and $O(V \times V')$ respectively, where p, p' are the leaves number of the trees, V and V' are the number of nodes in each tree, L and L' are maximum depths of each tree.

The instructions used in the above algorithms are fixed. The algorithms proposed in the next two chapters use a dynamic or a fixed but richer instruction set to achieve a smaller patch size.

3.3 Constructing post-order index

A node in a tree has a pre-order index and a postorder index. In the released patch, the indexes of the nodes are in pre-order, and the editing operations in the patch are in the ascending order of the involved nodes' indexes. Thus the "real-time independent patch decoding" discussed in Section 2.4.4 is possible. The postorder index will be used in the patch generation algorithm in Chapter 5.

Since the tree is constructed in pre-order as described in Section 3.2, the pre-order index of a node is just the order that the node is constructed. When a node is constructed, the addresses of its parent, first left child, left and right siblings (if any) and its pre-order index are recorded. These will be used to construct post order index for nodes after the whole tree is constructed completely.

The dynamic function, "ConstructPostOrderIndex", constructs the post order index for each node, and also determines the pre-order index of the rightmost leaf descendant for each node. The pre-order index of the rightmost leaf descendant for each node will be used in patch generation in Chapter 5. The implementation of this function is described as follows.

<u>ConstructPostOrderIndex</u>

Inputs:

1. pRootNode, the address of the root of the tree

2. nIndexInArray, the starting post order index for the subtree rooted at “pRootNode”.

Output:

nDesc, number of node in this tree.

Global variable:

1. PostOrderIndex[], an array of pointers, whose *i*th element is the address of the node with post order index of i.
2. RightMostDescPreOrderIndex[], an array of integers, whose *i*th element is the pre-order index of the rightmost leaf descendant of node with post order index of i.

Implementation:

```
Declare a node variable pNode=pRootNode's first left child.
Let nDesc be 0.
Loop while (pNode is not a NULL pointer)
{
   nDesc+=ConstructPostOrderIndex(pNode,nIndexInArray).
   Let pNode point to pNode's right sibling
}
PostOrderIndex[nIndexInArray]= pRootNode
RightMostDescPreOrderIndex[pRoot's pre-order index]=
   pRoot's pre-order index+nDesc
nIndexInArray++;
nDesc++;
Return nDesc;
End ConstructPostOrderIndex
```

Calling “ConstructPostOrderIndex” with the inputs of tree root and starting index 0 will construct the postorder index and find the pre-order index of the rightmost leaf descendant for each node in the tree.

3.4 Definitions and assumptions

In this thesis, we compute web patches in tree space. This subsection gives some definitions and assumptions that will be used in the patch generation.

Assumption 3.1. *The minimal editable object is a node in a tree.*

A patch is a file published by an original web server to update the stale cached copy. Its structure is defined as follows.

Definition 3.1. *Patch Structure*

A patch consists of a patch header and a patch body. The patch header is a series of items that describe an instruction used in the patch. The patch body is a series of edit operations that make changes on the tree. An edit operation consists of the encoded instruction, referred to as Op-code, and the operands.

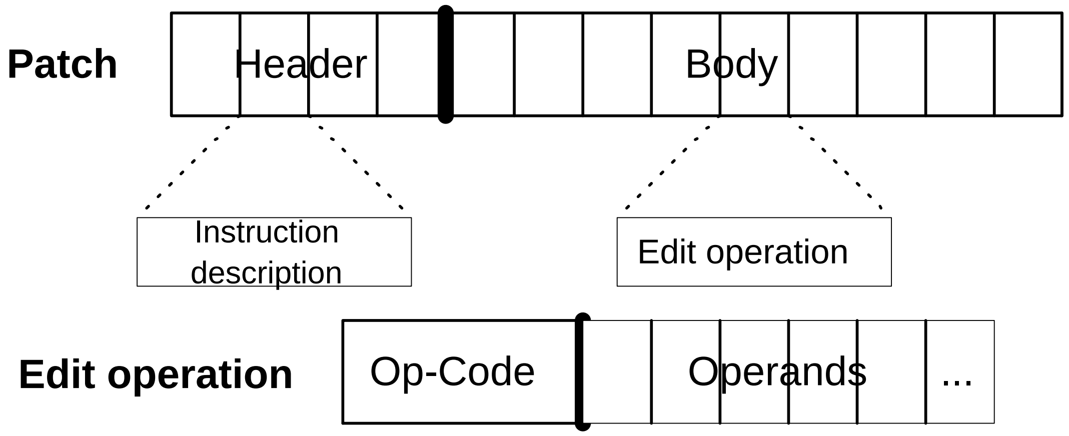


Figure 3.2: Patch structure

For instruction, it is assumed that

Assumption 3.2. *The instruction is independent of the node data.*

Thus, the node data will not appear in the header, and makes Assumption 4.2 in Section 4.3 reasonable.

The operands in an edit operation can be the applying position, associated raw data, the positions of the other involved nodes, and the auxiliary structure description. The cost of operands in an edit operation, say the ith operation is denoted as β_i. It is assumed that

Assumption 3.3. *The instruction and operands in an edit operation completely and exclusively describe the desirable change on a tree.*

Under Assumption 3.3, an edit operation can work alone regardless of others, and β_i is independent of the other operations.

In this thesis, we define two preliminary edit instructions, "insert a node" and "delete a node". Any change in a tree can be described as a certain combination of deletion and insertion. The preliminary instructions are sufficient to transform an old tree to the new one.

"Insert a node" is to construct a labelled node. It takes three types of operands. One type of operand is the raw data of the constructed node. The second type of operand indicates the applying location or the index of an existing node. The third type of operand indicates the structural relationship between the existing node and the constructed node. The structural information includes "before" or "after", "child", "parent" or "sibling", the number of descendants of the inserted node, and the number of other inserted nodes in-between. The explicit structure description makes the insert operations independent of each other even for those applying on the same location, thus the order of insertions does not matter.

Deleting a node means making the children of this node, if any, become the children of the deleted node's parent. With the view that the ordered labelled tree will be transformed into a web object file finally, "delete a node" is defined not to physically remove the node from the tree, but just to mark the node, and indicate that the node will not be involved in reconstructing the web object file from the tree subsequently. Thus the "deleted" node is still there, and the insertion applying on it is still applicable. The order of insertion and deletion does not matter. Moreover, after applying an edit operation, the nodes in the tree are not re-indexed. Thus, given a group of edit operations, the way that a tree is changed is fixed regardless of the operation order.

Fig. 3.3 gives an example of a delete operation. In Fig. 3.3, the post order index and the label are given beside the node. In this example, node 4 is deleted, thus its descendants, nodes 0, 1, 2, 3, become the descendants of its parent, node 5. The structure of node 4's descendants persists in the new tree.

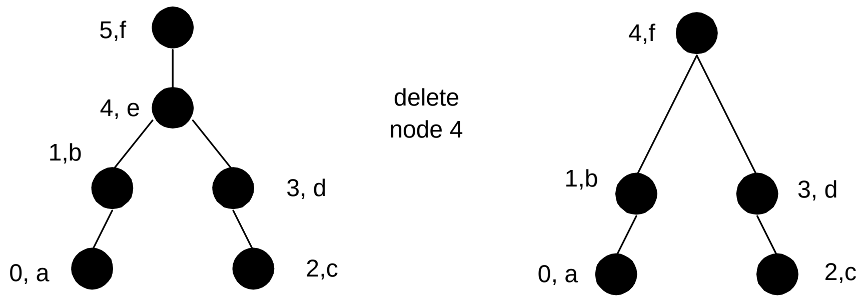


Figure 3.3: An example of delete operation

Fig. 3.4 gives an example of an insert operation, which is the complement of the dele operation in Fig. 3.3. In this example, node 'e' is to be inserted back to its original position. The structure description in the insert operation indicates that the node ‘e’ is inserted **after** node 3 and that node 3 is its **child**. By “after” here, we mean that the inserted node will have a larger postorder index than node 3, and if we list the nodes in the ascending order of postorder index, node ‘e’ will be after node 3. The structure description also indicates that the inserted node has **4 descendant** nodes and there is **no** other nodes inserted between it and node 3. According to the structure description, nodes 0, 1, 2, 3 become the descendants of the inserted node ‘e’, and node 3’s parent, node 4, becomes its parent. The node ‘e’ is thus put back to its original position before deletion in Fig. 3.3.

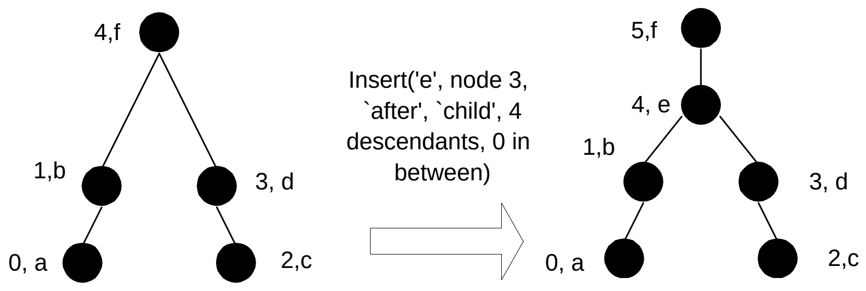

4,f
1,b
3, d
0, a
2,c
Insert('e', node 3, `after', `child', 4 descendants, 0 in between)
5,f
4, e
1,b
3, d
0, a
2,c

Figure 3.4: An example of insert operation

3.5 Chapter summary

In this chapter, we unify web object file types, propose how to transform a web object into an ordered labelled tree and define the web patch structure used in this thesis. In the next three chapters, we will show how to compute web patches in tree space.

Chapter 4

Minimal web patch with dynamic instruction set

In this chapter, we use a dynamic instruction set to compute the minimal web patch. We model web patch generation as a minimal set cover problem (MSCP) with dynamic weight. This chapter presents appropriate simplifying assumptions and solves the dynamic weight MSCP using available approximation solutions of MSCP.

4.1 Dynamic instruction set and patch size

As shown in Section 3.4, two preliminary instructions are sufficient to correct a tree. However, a patch that contains only preliminary edit operations may be large. A certain combination of preliminary edit operations can be defined as a smart or advanced operation to make the patch smaller. An example is the "Replace" edit operation, which is a substitution of "Delete a node" and "Insert a node" at the same location. An advanced edit operation is derived from a certain combination of preliminary edit operations. It contains an advanced instruction and the corresponding operands. Note that not all combinations of preliminary edit operations make a change to the underlying web object. Some combinations of preliminary edit operations do nothing to correct a tree. This kind of advanced instruction is called a null instruction.

As defined in Section 3.4, a patch has a patch header and a patch body. Patch header describes the instructions used in the patch, and patch body contains a series of edit operations. Using more advanced instructions may decrease the number of edit operations in the patch body, but at the same time, higher cost is introduced into the patch header in the form of more instruction definitions. A balance is desirable to achieve the minimal patch size. The advanced and preliminary instructions used in the patch constitute a dynamic instruction set.

The patch size is the sum of the header size and the body size. Let N_I depict the number of non-null instructions used in the patch, and let the cost of describing the kth type of instruction be α_k. Thus, the patch header size, S_H, is $\sum_{k=1}^{N_I} \alpha_k$.

The patch body, whose size is depicted as S_B, is a series of edit operations. Let N_E be the number of non-null edit operations in the patch body. An edit operation has its own operands. The size of the operands of an edit operation, say the ith operation, is denoted as β_i. Under Assumption 3.3, an edit operation can work alone, and β_i is independent of the other operations. Let $\zeta(N_I)$, a function of N_I, represent the size of the encoded instruction in an edit operation. The size of the ith edit operation, denoted as S_{Ei}, is then

$$S_{Ei} = \zeta(N_I) + \beta_i. \tag{4.1}$$

The patch size, S_P is then

$$\begin{aligned} S_P &= S_H + S_B \\ &= \sum_{k=1}^{N_I} \alpha_k + \sum_{i=1}^{N_E} (\zeta(N_I) + \beta_i). \end{aligned} \tag{4.2}$$

Given this formulation for the patch size, we shall now proceed to solve the web patch problem as a set cover problem to achieve the minimal patch size.

4.2 Formulating the minimal web patch problem as a set cover problem

Given two web objects, two ordered labelled trees, T_1 and T_2 can be generated. Suppose they have n_1 and n_2 nodes respectively. The trivial way to transform one tree, say T_1, to the other, say T_2, is to delete all nodes in T_1 and insert all nodes of T_2 into T_1. These preliminary edit operations are defined as the ground elements, depicted as t_i ($i \leq n$, $n = n_1 + n_2$), and they together form a set, depicted as $T = \{t_1, t_2, ..., t_n\}$.

To reduce the preliminary patch, advanced edit operations are desirable to substitute certain combinations of preliminary edit operations. As defined in Section 4.1, an edit operation is equivalent to one or a group of preliminary operations, and forms a subset of T. In this chapter, we use the corresponding subset to depict the edit operation. The edit operations are depicted as $S_1, S_2, ..., S_m$, where $m = \sum_{j=1}^{n} C_n^j = 2^n - 1$ and $S_i \subseteq T$.

The subsets are further classified into groups. Edit operations performing similar kinds of actions are classified into the same group. They then share the same instruction code. Let $G_1, G_2, ..., G_l$ $(l \leq m)$ depict these groups, corresponding to non-null actions. A $m \times l$ matrix B is used to indicate the subsets' belonging to each non-null action. The element $b_{ij} \in \{0, 1\}$. $b_{ij} = 1$ indicates that subset S_i is of the jth type. Null instructions are grouped into a separate and special "null group", G^*. A vector C is used to indicate the subsets' belonging to G^*. The element $c_i \in \{0, 1\}$. $c_i = 0$ indicates that subset S_i is a null instruction, while $c_i = 1$ indicates that subset S_i is a non-null instruction.

Let a vector $X = [x_1, x_2, ..., x_m]$ $(x_i \in \{0, 1\})$ represent a valid web patch that transforms T_1 to T_2. If edit operation S_i is in this patch, $x_i = 1$; otherwise, $x_i = 0$. Let a vector $Y = [y_1, y_2, ..., y_l]$ $(y_j \in \{0, 1\})$ indicate the types of non-null instructions used in this patch. If the jth type of instruction is used in the patch, $y_j = 1$; otherwise, $y_j = 0$. Y is calculated from X and B as follows.

$$y_j = \{ \begin{array}{l} 1, if \ \sum_{i=1}^{m} b_{ij} \cdot x_i \geq 1 \\ \\ 0, if \ \sum_{i=1}^{m} b_{ij} \cdot x_i = 0 \end{array} \quad (4.3)$$

N_E, the number of non-null operations and N_I, the number of non-null instructions are calculated as follows.

$$\begin{aligned} N_E &= \sum_{i=1}^{m} c_i \cdot x_i \\ N_I &= \sum_{j=1}^{l} y_j \end{aligned} \quad (4.4)$$

The size of a patch (Eq. 4.2) represented by a vector X can now be recast as follows.

$$S_p = \sum_{i=1}^{m} (\beta_i + \zeta(N_I)) \cdot c_i \cdot x_i + \sum_{j=1}^{l} \alpha_j \cdot y_j \quad (4.5)$$

Theorem 4.1. *Given a valid patch represented by X, all the items in the subsets selected by the patch and those in the null subsets unselected by the patch cover the set of T completely. It is depicted as*

$$[\bigcup_{i=1}^{m} x_i \cdot S_i] \cup [\bigcup_{i=1}^{m} (1 - c_i) \cdot (1 - x_i) \cdot S_i] = T. \tag{4.6}$$

Proof. If

$[\bigcup_{i=1}^{m} x_i \cdot S_i] \cup [\bigcup_{i=1}^{m} (1 - c_i) \cdot (1 - x_i) \cdot S_i] \neq T$,

there must be some preliminary edit operations that can fill the gap, but cannot comprise a null instruction. Let $G^{'}$ be the collection of these edit operations, and we have

$[\bigcup_{i=1}^{m} x_i \cdot S_i] \cup [\bigcup_{i=1}^{m} (1 - c_i) \cdot (1 - x_i) \cdot S_i] \cup G^{'} = T$.

The set of T transforms T_1 to T_2. The above equation implies that the edit operations in the patch, $G^{'}$ and those unselected null instructions $(\bigcup_{i=1}^{m}(1-c_i)\cdot(1-x_i)\cdot S_i)$ together transform T_1 to T_2. Since the null instruction does not change the tree, thus only X and $G^{'}$ transform T_1 to T_2. If we use X and G' as the functions to correct a tree as well, the above equation can be recast as

$$G^{'}(X(T_1)) = T_2. \tag{4.7}$$

Let $X(T_1) = T_2^{'}$, thus we have

$$G^{'}(T_2^{'}) = T_2. \tag{4.8}$$

Since $G^{'}$ is a group of edit operations that cannot comprise a null operation, it makes change on the applied object. So from Eq. 4.8, we have

$$T_2^{'} \neq T_2. \tag{4.9}$$

However, patch X is given as a valid correction on T_1 to get T_2, we then have

$$T_2^{'} = T_2. \tag{4.10}$$

This is a contradiction with Eq. 4.9. □

Generating the minimal web patch becomes finding a particular vector X to minimize S_p in Eq. 4.5 under the constraint given in Eq. 4.6. The minimal web patch problem is thus formulated as a set problem as follows:

Formulation 1. *The minimal web patch problem in tree space.*

Inputs:

- *Ground elements* $T = \{t_1, t_2, ..., t_n\}$, $n = n_1 + n_2$.
- *Subsets* $S_1, S_2, ..., S_m \subseteq T$, $m = \sum_{j=1}^{n} C_n^j = 2^n - 1$.
- *Subsets' belonging to* $l (l \leq m)$ *non-null instruction groups, depicted as a matrix of* B.
- *Subsets' belonging to null instruction group, depicted as a vector of* C.
- *The method to encode instruction, and the cost of instruction code, depicted as* ζ*, a function of number of instruction used in the patch.*
- *Each subset's operand cost in the patch body, depicted as* β_i.
- *The cost of each type of subset in patch header, depicted as* α_j.

Goal:

- *Select subsets (represented by a vector* $X(x_1, x_2, ..., x_m)$, $x_i \in \{0, 1\}$*, where if* S_i *is selected, then* $x_i = 1$*, otherwise* $x_i = 0$*) to minimize*
  $S_p = \sum_{i=1}^{m} (\beta_i + \zeta(N_I)) \cdot c_i \cdot x_i + \sum_{j=1}^{l} \alpha_j \cdot y_j$ *where* N_I, Y *are calculated as in Eq. 4.3 and Eq. 4.4 respectively.*

  subject to

  $[\bigcup_{i=1}^{m} x_i \cdot S_i] \cup [\bigcup_{i=1}^{m} (1 - c_i) \cdot (1 - x_i) \cdot S_i] = T$.

The minimal web patch problem is formulated as a set problem achieving a minimal index S_p. It is further modeled as a set cover problem.

Since the null instruction does nothing on tree correction and contributes nothing to patch size (see Eq. 4.5), it can be assumed that

Assumption 4.1. *Null subsets are always selected in the feasible web patch problem solution.*

Theorem 4.2. *Under Assumption 4.1, the minimal web patch problem is a set cover problem.*

Proof. The goal of the web patch problem is to select subsets to minimize S_P. The selected subsets and the unselected null subsets are required to cover the whole ground element set. Given Assumption 4.1, the constraint (Eq. 4.6) becomes
$[\bigcup_{i=1}^{m} x_i \cdot S_i] \cup [\bigcup_{i=1}^{m} (1 - c_i) \cdot x_i \cdot S_i] = T$.
Obviously, $\bigcup_{i=1}^{m} (1 - c_i) \cdot x_i \cdot S_i \subseteq \bigcup_{i=1}^{m} x_i \cdot S_i$. Thus the constraint (Eq. 4.6) just becomes

$$\bigcup_{i=1}^{m} x_i \cdot S_i = T, \tag{4.11}$$

and that is a complete set cover. □

The minimal web patch problem is thus formulated as a set cover problem achieving the minimal S_P.

4.3 Solving the minimal web patch problem using WMSCP's solutions

4.3.1 The weighted minimal set cover problem (WMSCP)

This section introduces the minimal set cover problem and some of its solutions that are relevant to the formulation of the web patch in Section 4.2.

[84] formulates the weighted minimal set cover problem (WMSCP) as follows.

Inputs:

- Ground elements $T = \{t_1, t_2, ..., t_n\}$.
- Subsets $S_1, S_2, ..., S_m \subseteq T$.
- Weights $w_1, w_2, ..., w_m$ for the subsets.

Goal:

- Find a set $I \subseteq \{1, 2, ..., m\}$ that minimizes
  $\sum_{i \in I} w_i$,
  subject to
  $\bigcup_{i \in I} S_i = T$.

WMSCP is a NP optimization problem [85, 84, 86]. To address the complexity issues, Hochbaum [87] presents a LP rounding WMSCP approximation algorithm following the general 3-step approach for constructing approximation algorithms. The algorithm is as follows.

1. Formulate the WMSCP problem as an Integer Problem (IP) [88]. A variable x_i is assigned for each subset S_i. If $i \in I$, then $x_i = 1$, otherwise $x_i = 0$. The

goal becomes finding a vector $X(x_1, x_2, ..., x_m\}$ to

$$\begin{aligned} &\textbf{minimize} \\ &\qquad \textstyle\sum_{i=1}^{m} w_i \times x_i \\ &\textbf{subject to} \\ &\qquad \textstyle\sum_{j:t_i \in S_j} x_j \geq 1 \forall t_i \in T \\ &\qquad x_i \in \{0, 1\}. \end{aligned} \tag{4.12}$$

2. Relax it to a Linear Program (LP) [89] by changing the last constraint to $x_i \geq 0$.

3. Solve the LP / dual-LP, and obtain a suboptimal solution to IP by rounding the LP / dual-LP solution.

Hochbaum's algorithms have the time complexity of $O(n^3 \log n)$. Hochbaum also shows that the approximation ratio, f, is the maximum number of sets that contain any given element, [90]. This is given as Lemma 4.1 in this thesis.

Lemma 4.1. *[87] : Hochbaum's bounding LP algorithm is a f-approximation algorithm for WMSCP, and*

$$f = \max_i |\{j : t_i \in S_j\}|. \tag{4.13}$$

Two other well known algorithms that achieve smaller time complexity are the primal-dual algorithm [91] and greedy algorithm [92]. Primal-dual algorithm proposed by Bar-Yehuda and Even attains the same approximation ratio as [87], but in a reduced time complexity of $O(n^2)$. *Chvátal* [92] studies a natural greedy algorithm. The greedy algorithm selects a subset with minimal weight to cardinality ratio in each step. It leads to an approximation ratio of $\ln(\max_j |S_j|)$ and a time complexity of $O(mn)$.

4.3.2 Solve the minimal web patch problem using WMSCP solutions

In this subsection, we show how to use WMSCP's approximation solutions discussed in the last subsection to solve the minimal web patch problem under some assumptions. In Section 4.2, we formulate the minimal web patch problem as a weighted set cover problem. The edit operation in the patch corresponds to the selected subset in the set cover problem. It has its own cost in the patch header and patch body. The cost can be thought of as the associated weight. Eq. 4.5 in Section 4.2 shows that the weight associated with the subset is dependent on the solution, thus the minimal web patch problem is a minimal set cover problem with dynamic weight. The weight's dependence on the solution makes the minimal web patch problem more difficult than WMSCP.

We first examine the dynamic weight. In Eq. 4.5, β_i is the cost of operands in an operation. As discussed in Section 3.4, the operands of an edit operation are independent of other operations, so, β_i is independent of the solution and fixed. For α_j, since the instruction is independent of node data (Assumption 3.2), it can be assumed that:

Assumption 4.2. *The instructions used in a patch have the same description cost in the header.*

Under this assumption, all α_j are the same and is depicted as α. In this thesis, the dynamic component of the dynamic weight is not determined by a particular value but the statistical value of the solution (vector X). Eq. 4.5 becomes

$$S_p = \sum_{i=1}^{m} (\beta_i \cdot c_i) \cdot x_i + \zeta(N_I) \cdot N_E + \alpha \cdot N_I, \tag{4.14}$$

where N_E, N_I, Y can be calculated from X, B and C as in Eq. 4.4 and 4.3.

If we focus on those Xs that have the same N_I and N_E, the goal is equivalent to finding a X to minimize $\sum_{i=1}^{m}(\beta_i \cdot c_i) \cdot x_i$. It then becomes a MCSP with additional conditions of fixed N_I and N_E. Let us examine the WMSCP solutions mentioned in Section 4.3.1 under the additional (N_E, N_I) constraints.

The primal-dual algorithm of [91] and the greedy algorithm of [92] while achieving smaller time complexity, are not applicable to solve WMSCP with the additional (N_E, N_I) constraints. They use certain indexes to select one subset globally at each step. However, the additional N_E and N_I constraints on X make the selection possible only locally, and consequently conflict with the index model.

Let us examine the LP rounding algorithm [87]. Its first step is to formulate WMSCP as an Integer Problem (IP). The additional (N_E, N_I) constraints then become two more equation constraints in IP, given by

$$\begin{aligned} \sum_{i=1}^{m} c_i \cdot x_i &= N_E \\ \sum_{j=1}^{l} y_j &= N_I, \end{aligned} \tag{4.15}$$

where y_i is a function of X as given in Eq. 4.3.

Obviously, the two additional constraints in IP do not hinder its relaxation to a Linear Program in Step 2 and solving the LP in Step 3. Algorithm [87] can thus be used to solve WMSCP with (N_E, N_I) constraints.

Thus, under Assumption 4.2, an approximate minimal web patch problem algorithm utilizing WMSCP solution can be as follows:

Algorithm 1. *An approximate minimal web patch algorithm.*

For $N_E = 1 \, to \, m$

```
  **For** $N_I = 1$ *to* $N_E$
    *Solve WMCSP with ($N_E$, $N_I$) constraints using LP rounding algorithm [87].*
    **If** *(it is solvable)*
      *Let* $\hat{X}(N_E, N_I)$ *be the result and* $\hat{S_P}(N_E, N_I)$ *be the corresponding patch size.*
  **Next** $N_I$
**Next** $N_E$
$\hat{X}(N_E, N_I)$ *with the minimal* $\hat{S_P}(N_E, N_I)$ *is the final solution.*
```

Theorem 4.3. *Under Assumption 4.2, Algorithm 1 is an f-approximation algorithm for web patch problem, where*

$$f = \max_i |\{j : t_i \in S_j\}|.$$

Proof. Let OPT be the optimal solution for web patch problem under Assumption 4.2. Let $\hat{S_P}$ be the patch size achieved by Algorithm 1.

Let $OPT(N_E, N_I)$ be the optimal patch size under (N_E, N_I) constraint. From Lemma 4.1, we have

$$\hat{S_P}(N_E, N_I) - (\zeta(N_I) \cdot N_E + \alpha \cdot N_I) \leq$$
$$f \cdot [OPT(N_E, N_I) - (\zeta(N_I) \cdot N_E + \alpha \cdot N_I)].$$

Since $f \geq 1$, the above inequality becomes

$$\hat{S_P}(N_E, N_I) \leq f \cdot OPT(N_E, N_I).$$

OPT must be one of $OPT(N_E, N_I)$. Let (N_{E0}, N_{I0}) be that corresponding constraints, then we have

$$\hat{S_P}(N_{E0}, N_{I0}) \leq f \cdot OPT.$$

According to Algorithm 1, $\hat{S_P}$ is the smallest one among $\hat{S_P}(N_E, N_I)$, then

$$\hat{S_P} \leq \hat{S_P}(N_{E0}, N_{I0}).$$

With the above two inequalities, we have

$$\hat{S_P} \leq f \cdot OPT.$$

□

4.4 Chapter summary

This chapter models the patch generation with dynamic instruction set as a dynamic weighted MSCP. Under some simplifying assumptions, solutions to MSCP can be used to solve this dynamic weighted MSCP. This is given as Algorithm 1. The worst case in Algorithm 1 is that every WMSCP with (N_E, N_I) constraint is solvable. If that is the case, solutions to $n \times (m-1)/2$ MCSP s are needed to find the approximate minimal web patch solution, where $m = 2^n - 1$ and $n = n_1 + n_2$, the total number of nodes of the two trees. This non-polynomial time complexity makes this algorithm not scalable. In the next chapter, we will use a fixed instruction set and design algorithms with polynomial time complexity.

Chapter 5

Web patch with fixed instruction set

Chapter 4 uses a dynamic instruction set to compute minimal size web patches. The computation, however, has a high time complexity. To achieve a better time complexity, this chapter defines a fixed instruction set. This fixed instruction set is richer than that used in [79, 83, 1]. This chapter describes patch generation algorithms under this fixed instruction set. To evaluate them, more than 200,000 URLs were checked regularly for updates and the proposed algorithms were applied on them.

Since the instruction set used in this chapter is fixed, the patch header size is also fixed. In this chapter, patch size refers to the patch body size.

5.1 Fixed instruction set

Given the stale version and the fresh version web objects, two trees, a stale tree and a fresh tree can be constructed using the method given in Chapter 3. We first classify the tree nodes. A node whose label appears in both trees is called a common node. In the stale tree, a node whose label does not appear in the new tree is an old node. In the fresh tree, a node whose label does not appear in the stale tree is a new node.

The node-oriented operation can be classified into five job scopes. They are construction, destruction, substitution, duplication and shuffling. Corresponding to these job scopes, five instructions are defined. They are “insert”, “delete”, “replace”, “copy” and “move”.

“Insert” and “delete” are preliminary instructions, and they have been defined in Section 3.4. This chapter constrains that only new nodes can be inserted. This constraint is also imposed on the “replace” instruction, which uses a new label to replace the existing label of a node. Such a constraint will eliminate the possibility that common node data appearing in the patch. To avoid confusion, the insert and replace instruction that do not have such constraint are denoted as insert* and replace*, respectively.

With this constraint, “insert”, “delete” and “replace” may not be sufficient to correct a tree, since they cannot handle node duplication and shuffling. These two kinds of node manipulation are covered by “copy” and “move” respectively.

Fig. 5.1 gives the relationship between node classes and instructions.

5.2 Algorithms

This section proposes three patch generation algorithms using the instruction set defined above. The first two algorithms are suboptimal. They borrow ideas in [83] and [1], and have the following assumption as in [83] and [1].

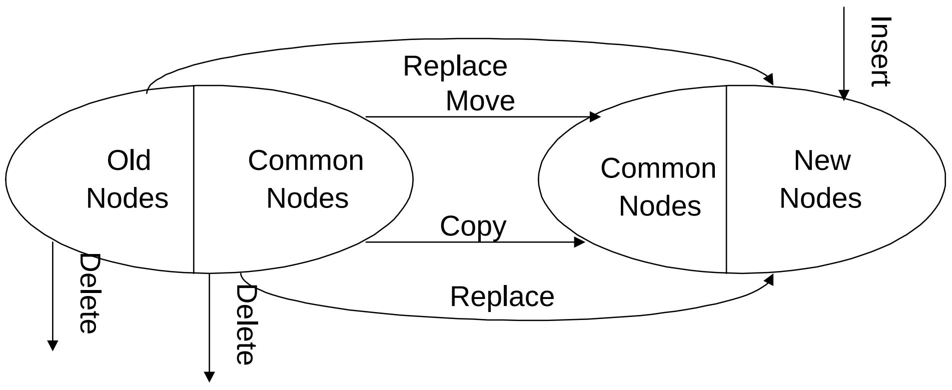


Figure 5.1: Instructions and node types

Assumption 5.1. *The roots of all trees have the same label and the root of each tree remains unchanged during editing.*

5.2.1 Maximum number in-order mapping method: a sub-optimal algorithm to compute patch by dividing the problem in the node domain

Given two trees, we first classify the nodes in both trees into old and common or new and common types according to the above definition. We then compute edit operations first on common nodes, followed by the rest of the nodes.

A common node in an old tree and a common node in a new tree with the same label form a mapping. In this thesis, we define two mappings, in-order mappings and out-of-order mappings.

Convention 1. *Let T[i] be the ith node in tree T following the postorder numbering. The mapping* $(T_1[i_1], T_2[j_1])$ *and* $(T_1[i_2], T_2[j_2])$ *are in-order if and only if the following are satisfied.*

1. *$T_1[i_1]$ and $T_2[j_1]$ have the same label (denoted as $T_1[i_1] = T_2[j_1]$);*

2. *$T_1[i_2] = T_2[j_2]$;*

3. *$j_1 = j_2$, if and only if $i_1 = i_2$;*

4. *$T_1[i_1]$ is to the left of $T_1[i_2]$ if and only if $T_2[j_1]$ is to the left of $T_2[j_2]$;*

5. *$T_1[i_1]$ is an ancestor of $T_1[i_2]$ if and only if $T_2[j_1]$ is an ancestor of $T_2[j_2]$.*

Mappings that do not meet the above requirements are out-of-order. A group of mappings are in-order when each two of them are in order. A mapping is out-of-order with a group of mappings when it is out-of-order with at least one mapping in the group. Notice that the "mapping" defined in [83] and [1] refer to a set of node pairs that meets Requirements 3, 4 and 5 only. It is called mapping* here to avoid confusion.

[83] states that if the nodes that are not attached by "mapping*" are removed from the two trees, the rest of the nodes in both trees are then of the same structure. Since "in-order mapping" is "stricter" than "mapping*", it can be claimed similarly that after removing the nodes unattached by in-order mapping, the two trees are exactly the same. In other words, the nodes attached by in-order mappings need not be considered in edit operations. The common nodes that are attached by out-of-order mappings need position shifting operations. They lead to "copy" and "move" edit operations.

The first proposed algorithm is performed in two steps as follows.

Step 1

We first determine all mappings between common nodes of the two trees. We then find the maximum number of in-order mappings among them. Finally, we examine the out-of-order mappings to generate copy and move edit operations.

In order to find the maximum number of in-order mappings, the algorithm uses a function MappingCost(i,j) to represent the cost of a mapping* associated with mapping $(T_1[i], T_2[j])$.

Definition 5.1. *The cost of a mapping* between node* $T_1[i]$ *in tree* T_1 *and node* $T_2[j]$ *in tree* T_2 *is defined as*

$$MappingCost(i,j) = \{ \begin{array}{l} -\infty, when T_1[i] \neq T_2[j]; \\ 1, when T_1[i] = T_2[j], \end{array}$$

The maximum number of in-order mapping can then be found by finding the maximum cost of mapping* as follows.

The nodes in the tree are numbered in the post order convention. Let *T[i..j]* be the sub-forest of T induced by the nodes numbered from i to j inclusive. Let ancestor(i) be the collection of the indexes of *T[i]*'s ancestors. An empty tree or forest is denoted as φ. Let $l(i)$ be the post order index of the leftmost leaf descendant of the sub-tree rooted at T[i]. The maximum number of in-order mapping between forests $T_1[i'..i]$ and $T_2[j'..j]$ is denoted as forestmap$(T_1[i'..i], T_2[j'..j])$ or simply forestmap$(i'..i, j'..j)$. The maximum number of in-order mapping between two subtrees $T_1[l(i)..i]$ and $T_2[l(j)..j]$ is denoted as treemap(i,j). Initially, we have:

$$\begin{aligned} forestmap(\varphi, \varphi) &= 0 \\ forestmap(T_1[i'..i], \varphi) &= 0 \\ forestmap(\varphi, T_2[j'..j]) &= 0 \end{aligned} \tag{5.1}$$

Lemma 5.1. *Let $i_1 \in ancestor(i)$ in tree T_1 and $j_1 \in ancestor(j)$ in tree T_2, then*

$$\begin{aligned} forestmap(l(i_1)..i; l(j_1)..j) = Max\{ \\ forestmap(l(i_1)..i-1, l(j_1)..j), \\ forestmap(l(i_1)..i, l(j_1)..j-1), \\ forestmap(l(i_1)..l(i)-1, l(j_1)..l(j)-1) + \\ forestmap(l(i)..i-1, l(j)..j-1) + MappingCost(i,j)\}. \end{aligned} \tag{5.2}$$

Proof. The mapping can be extended to $T_1[i]$ and $T_2[j]$ in three ways.
Case 1. $T_1[i]$ is not attached by an in-order mapping, so

$$forestmap(l(i_1)..i; l(j_1)..j) = forestmap(l(i_1)..i-1; l(j_1)..j).$$

Case 2. $T_2[j]$ is not attached by an in-order mapping, so

$$forestmap(l(i_1)..i; l(j_1)..j) = forestmap(l(i_1)..i; l(j_1)..j-1).$$

Case 3. $T_1[i]$ and $T_2[j]$ are both attached by an in-order mappings. Then the mapping must be *(i,j)* argued as follows. Suppose *(i,k)* and *(h,j)* are in-order mapping. If $l(i_1) \leq h \leq l(i)-1$, then $T_1[i]$ is to the right of $T_1[h]$. From the sibling order condition on mappings (Condition 4), $T_2[k]$ is to the right of $T_2[j]$ too. But this is impossible in $T_2[l(j_1)..j]$. Similarly, if $T_1[i]$ is an ancestor of $T_1[h]$, $T_2[k]$ must be an ancestor of $T_2[j]$ from the ancestor condition on mapping. But it is also impossible in $T_2[l(j_1)..j]$. So *h=i*, and by symmetry, *k=j*. From the ancestor condition on mapping (Condition 5), any node in the subtree rooted at $T_1[i]$ can only be mapped to a node in the subtree rooted at $T_2[j]$. So $T_1[i] = T_2[j]$ and

$$\begin{aligned} forestmap(l(i_1)..i, l(j_1)..j) &= MappingCost(i,j) + \\ &\quad forestmap(l(i_1)..l(i)-1, l(j_1)..l(j)-1) + \\ &\quad forestmap(l(i)..i-1, l(j)..j-1). \end{aligned}$$

□

Note that Definition 5.1 makes Case 3 in Lemma 5.1 possible only when $T_1[i] = T_2[j]$ (See Fig. 5.2).

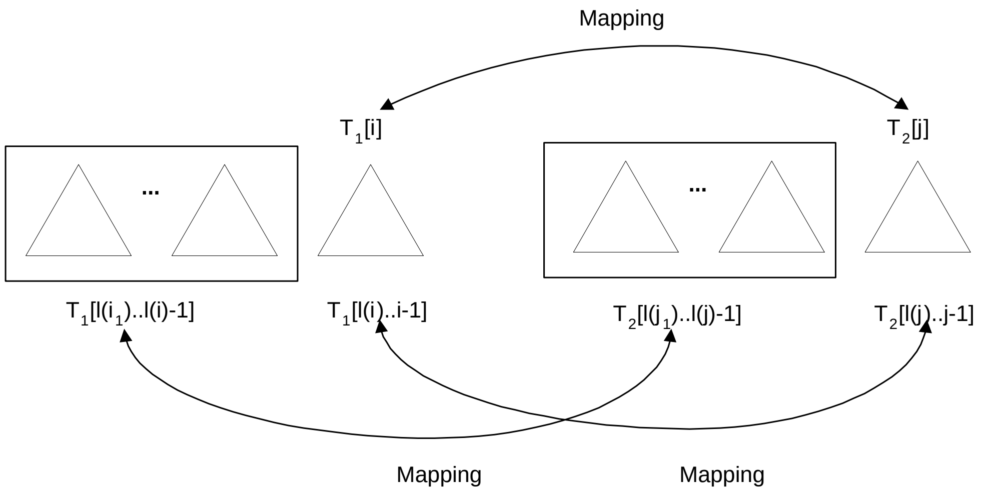


Figure 5.2: Case 3 in Lemma 5.1

From Lemma 5.1, we have the following theorem.

Theorem 5.1. *Let $i_1 \in ancestor(i)$ and $j_1 \in ancestor(j)$.*

Consider Lemma 5.1 in the following two situations (see Fig. 5.3).

If $l(i) = l(i_1)$ and $l(j) = l(j_1)$,

$$\begin{aligned} forestmap(l(i_1)..i, l(j_1)..j) &= treemap(i,j) \\ &= Max\{forestmap(l(i_1)..i-1, l(j_1)..j), \\ &\quad forestmap(l(i_1)..i, l(j_1)..j-1), \\ &\quad forestmap(l(i_1)..i-1, l(j_1)..j-1) \\ &\quad +MappingCost(i,j)\}. \end{aligned} \tag{5.3}$$

If $l(i) \neq l(i_1)$ or $l(j) \neq l(j_1)$,

$$\begin{aligned} forestmap(l(i_1)..i, l(j_1)..j) &= Max\{forestmap(l(i_1)..i-1, l(j_1)..j), \\ &\quad forestmap(l(i_1)..i, l(j_1)..j-1), \\ &\quad forestmap(l(i_1)..l(i)-1, l(j_1)..l(j)-1) \\ &\quad +treemap(i,j)\}. \qquad (5.4) \end{aligned}$$

Proof. Following from Lemma 5.1, if $l(i) = l(i_1)$ and $l(j) = l(j_1)$, then in Case 3, $forestmap(l(i_1)..l(i)-1, l(j_1)..l(j)-1) = forestmap(\varphi, \varphi) = 0$. Equation 5.2 becomes

$$\begin{aligned} forestmap(l(i_1)..i, l(j_1)..j) &= treemap(i,j) \\ &= Max\{forestmap(l(i_1)..i-1, l(j_1)..j), \\ &\quad forestmap(l(i_1)..i, l(j_1)..j-1), \\ &\quad forestmap(l(i_1)..i-1, l(j_1)..j-1) + \\ &\quad MappingCost(i,j)\}. \end{aligned}$$

If $l(i) \neq l(i_1)$ or $l(j) \neq l(j_1)$, then in Case 3, since we are looking for the maximum cost of in-order mapping, $forestmap(l(i_1)..i, l(j_1)..j) \geq forestmap(l(i_1)..i-1, l(j_1)..j-1) + treemap(i,j)$. Similarly, $treemap(i,j) \geq forestmap(l(i)..i-1, l(j)..j-1) + MappingCost(i,j)$. These two inequalities imply that in Case 3, $forestmap(l(i_1)..i, l(j_1)..j) = forestmap(l(i_1)..l(i)-1, l(j_1)..l(j)-1)+treemap(i,j)$ Equation 5.2 becomes

$$\begin{aligned} forestmap(l(i_1)..i, l(j_1)..j) &= Max\{forestmap(l(i_1)..i-1, l(j_1)..j), \\ &\quad forestmap(l(i_1)..i, l(j_1)..j-1), \\ &\quad forestmap(l(i_1)..l(i)-1, l(j_1)..l(j)-1) + \\ &\quad treemap(i,j)\}. \end{aligned}$$

□

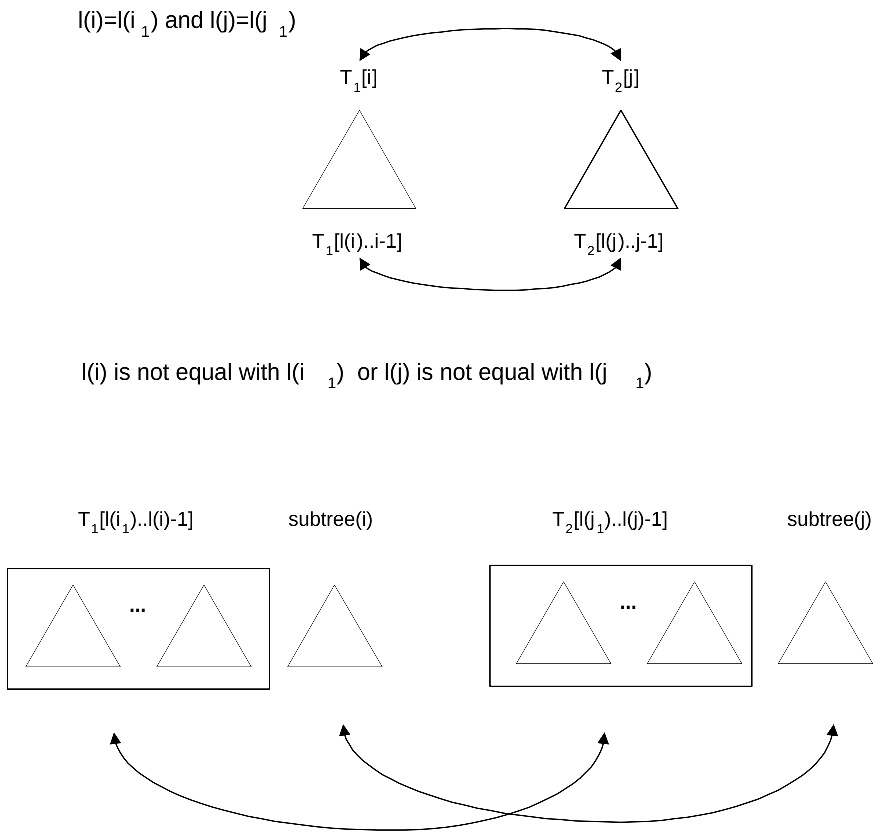


Figure 5.3: Two situations in Theorem 5.1

The basic idea in the proof of Lemma 5.1 is the same as that in the proof of Lemma 4 [1], although in Lemma 4 [1] the minimum "forest distance" is computed instead. The basic idea in deducing Theorem 5.1 from Lemma 5.1 is also the same as that in deducing Lemma 5 [1] from Lemma 4 [1]. Lemma 5 [1] uses a dynamic programming algorithm [93] to compute the minimum cost "editing distance" between the two trees. Similarly, in this chapter, we use a dynamic programming algorithm to compute the maximum number of mapping between the two trees from bottom upwards to reduce time complexity at the cost of space complexity.

To avoid computing a subtree mapping repeatedly, as in [1], $keyNodes$, which is a set of node index for a tree T is defined as follows.

$keyNodes = \{k|T[k]$is the tree root or $T[k]$ has no left sibling in the tree$\}$.

In the course of computing subtree mapping, $treemap(i_1, j_1)$, $treemap(i, j)$ is computed as a byproduct, where $i_1 \in keyNodes_1, i_2 \in keyNodes_2$, $T_1[i]$ is on the path from $T_1[l(i_1)]$ to $T_1[i_1]$ and $T_2[j]$ is on the path from $T_2[l(j_1)]$ to $T_2[j_1]$.

The algorithm below shows a method to find the maximum number of in-order mappings.

Inputs: trees T_1 and T_2

Output: The maximum cost of in-order mapping, and the corresponding in-order mapping node pairs.

Two global arrays:

1. treemap[][] with SizeOf(T_1)×SizeOf(T_2) elements. The element *treemap*[i][j] is the maximum mapping cost between the subtree rooted at $T_1[i]$ and the subtree rooted at $T_2[j]$.
2. treemapNodes[][] with SizeOf(T_1)×SizeOf(T_2) elements. The element *treemapNodes*[i][j] is the maximum in-order mapping group between the subtree rooted at $T_1[i]$ and the subtree rooted at $T_2[j]$.

```
Main loop:
For each $i'$ from 1 to SizeOf($keyNodes_1$) do
    For each $j'$ from 1 to SizeOf($keyNodes_2$) do
        $i = keyNodes_1(i')$,$j = keyNodes_2(j')$.
        Compute treemap[i][j] and Compute treemapNodes[i][j].
End Main loop
```

The maximum in-order mapping cost is treemap[SizeOf(T_1)][SizeOf(T_2)].

The corresponding mapping node pairs are in treemapNodes[SizeOf(T_1)][SizeOf(T_2)].

```
Compute treemap[i][j]:
Temporary array $forestmap$ is needed.
Use Equation 5.1 to initialize
    1. $forestmap[\varphi, \varphi]$.
    2. $forestmap(T_1[l(i)..i_1], \varphi)$ (where $i_1 = l(i)...i$).
    3. $forestmap(\varphi, T_2[l(j)..j_1])$ (where $j_1 = l(j)...j$).
For each $i_1$ from l(i) to i do
    For each $j_1$ from l(j) to j do
        Use Theorem 5.1 to compute $forestmap(l(i)..i_1, l(j)..j_1)$.
End Computing treemap[i][j]
```

The corresponding mappings are found by retracing the computation of treemap[i][j] backwards as follows.

```
Compute treemapNodes[i][j]:
Define $i_1 = i$; $j_1 = j$.
While ($i_1 \geq l(i)$ AND $j_1 \geq l(j)$) do
    If ($T_1[i_1] == T_2[j_1]$)
        If ($l(i_1) == l(i)$ AND $l(j_1) == l(j)$)
            If ($forestmap(l(i)..i_1, l(j)..j_1) == forestmap(l(i)..i_1 - 1, l(j)..j_1 - 1)$
              $+MappingCost(i_1, j_1)$)
              Append $(i_1, j_1)$ to treemapNodes[i][j].
              $i_1 = i_1 - 1$.
              $j_1 = j_1 - 1$.
              continue.
        else
            If ($forestmap(l(i)..i_1, l(j)..j_1) == forestmap(l(i)..i_1 - 1, l(j)..j_1 - 1)$
              $+treemap(i_1, j_1))$
              Append $(i_1, j_1)$ to treemapNodes[i][j].
              $i_1 = l(i_1) - 1$.
              $j_1 = l(j_1) - 1$.
              **continue**.
    **else**
        **If** $(forestmap(l(i)..i_1, l(j)..j_1) == forestmap(l(i)..i_1 - 1, l(j)..j_1)$ AND
              $forestmap(l(i)..i_1, l(j)..j_1) == forestmap(l(i)..i_1, l(j)..j_1 - 1))$
            **If** $(i_1 > j_1)$
              $i_1 = i_1 - 1$.
            **else**
              $j_1 = j_1 - 1$.
            **continue**.
        **If** $(forestmap(l(i)..i_1, l(j)..j_1) == forestmap(l(i)..i_1 - 1, l(j)..j_1))$
            $i_1 = i_1 - 1$.
            **continue**.
        **If** $(forestmap(l(i)..i_1, l(j)..j_1) == forestmap(l(i)..i_1, l(j)..j_1 - 1))$
            $j_1 = j_1 - 1$.
            **continue**
End computing treemapNodes[i][j]
```

Following from Lemma 6, 7 and Theorem 2 in [1], the time complexity of finding the maximum number of in-order mapping is

$$O(|T_1| \times |T_2| \times min(depth(T_1), leaves(T_1)) \times min(depth(T_2), leaves(T_2))),$$

where the function $depth(T)$ and $leaves(T)$ give the number of levels and leaves in the tree T respectively.

As shown above, we retrace the computation of *treemap*[][] backwards a step at a time through the feasible paths. At each step, each pair of corresponding nodes in the two trees are checked to construct the in-order mappings. In the situation that there are two paths with the same retrace cost, we select the path in which the next two nodes tied by its mapping have the closer index. By finding the maximum number of in-order mapping such that the relative index of a pair of corresponding nodes in any computing situations is smallest, we minimize any structure mismatch between the two trees.

After finding the maximum in-order mapping group, we generate "copy" and "move" operations from the rest of the mappings. We examine the common nodes in the new tree one at a time. If a common node in the new tree is not involved in any in-order mapping, then there is an out-of-order mapping with this common node. If its counterpart in the old tree is attached by an in-order mapping, a "copy" edit operation is generated, otherwise, a "move" edit operation is generated. After all the common nodes are examined, Step 1 ends.

Step 2

This step works on the tree nodes that are not involved in the in-order mappings, and the copy and move edit operations generated in Step 1. From the two incomplete trees, only insert, delete and replace edit operations are constructed in Step 2.

In this step, we use [1]'s algorithm to generate the correction on the two incomplete trees. [1]'s algorithm uses three instructions: insert*, delete, and replace* to compute the minimum tree correction. Although insert* and replace* used in [1] do not have the constraint introduced in Section 5.1, they would not lead to edit operations that conflict with the constraint. This is because all the common nodes in the new tree are not involved in Step 2. When applying [1]'s algorithm on the two trees, those nodes involved in in-order mappings, "copy" and "move" are skipped. A "replace" between two nodes is possible only when the link of the two nodes is in order with the in-order mapping group found in Step 1. This is to avoid conflict of "mapping*" in Step 2 with the in-order mapping in Step 1.

A node in a tree that is attached by an in-order mapping is referred to as a "in-order node" here. We use the in-order nodes to divide the tree into "forests" as follows.

The nodes in a tree are indexed in postorder. The nodes that are before the first in-order node, after the last in-order node or between two successive in-order nodes form one or more subtrees, and generally, we refer to it as a forest. In this way, multiple forests are isolated by the in-order nodes in a tree. If a forest in the old tree and a forest in the new tree are isolated by the in-order nodes in the same in-order mappings, they form an "in-order forest pair". Fig. 5.4 gives an example of an in-order forest pair. In Fig. 5.4, T_1 and T_2 have two in-order mappings, $(T_1[3], T_2[3])$ and $(T_1[6], T_2[5])$. Both T_1 and T_2 are divided into two forests. In this case, $T_1[0..2]$ and $T_2[0..2]$ form an in-order forest pair, and $T_1[4..5]$ and $T_2[4..4]$ form another in-order forest pair.

Because the "replace" operation must be in-order with the in-order mapping group generated in Step 1, "replace" is possible only between two forests in the same in-order forest pair. Nodes that are not attached by a "replace" are deleted if they are in the old tree. If they are in the new tree, such nodes are inserted. We note that the operation generated on a node in an in-order forest pair is independent of

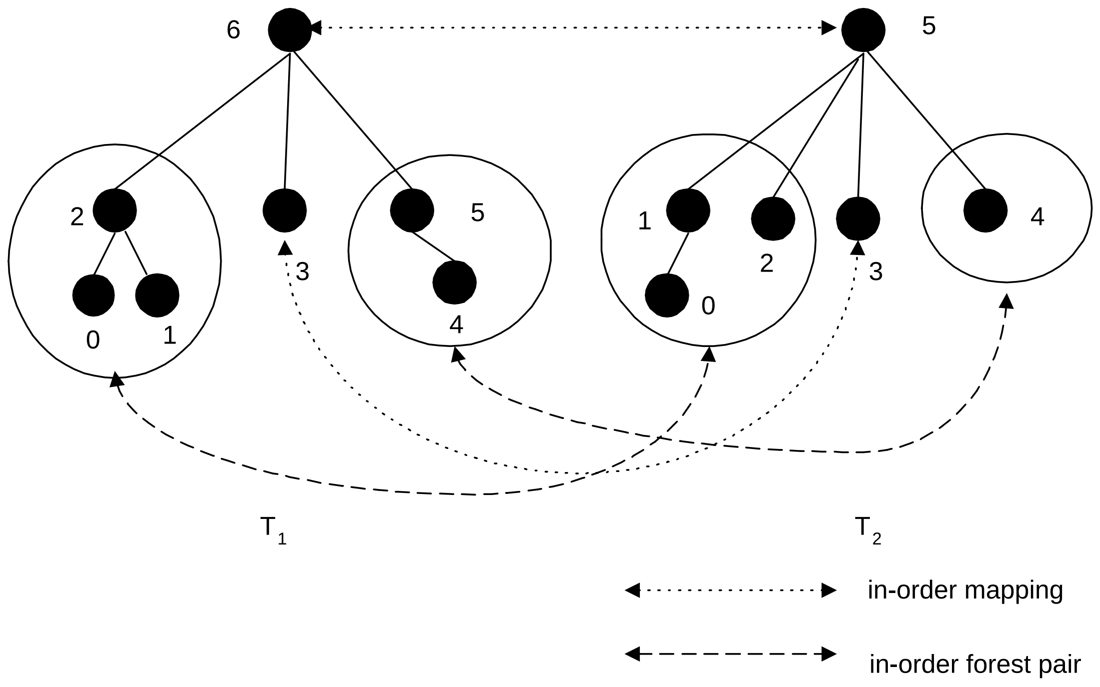


Figure 5.4: An example of in-order forest pair

other in-order forest pairs. Thus computing the correction for two incomplete trees can be done by computing the correction for each in-order forest pair. In our case, [1]'s algorithm is applied on each in-order forest pair. To comply with Assumption 5.1, we add a virtual root with an unique label to each forest.

In Step 2, when we compute the correction for each in-order forest pair, we simultaneously check if breaking an in-order mapping can result in a smaller correction. For an in-order mapping situated between two in-order forest pairs, say $Pair_1$ and $Pair_2$, we examine two cases. In the first case, we do not break the in-order mapping. The corrections for $Pair_1$ and $Pair_2$ are computed independently. In the second case, the in-order mapping is transformed to a "copy" operation, so that "replace" is possible between the forest in $Pair_1$ and the forest in $Pair_2$. We combine $Pair_1$ and $Pair_2$ into a new in-order forest pair and compute the correction for it. We compare the cost of the corrections in the above two cases and take the smaller one of the two.

Step 2 is described as follows.

```
Let $M$ be the number of in-order mapping generated in Step 1.
Let $N_1$ and $N_2$ be the number of nodes in $T_1$ and $T_2$ respectively.
Let two arrays, $InOrderNodeIndex_1[]$ and $InOrderNodeIndex_2[]$, hold the post
order index of the in-order node in $T_1$ and $T_2$ respectively in the ascending order.
Define $OpCollection$. It holds the correction on the two incomplete trees.
Define array $OpPair[]$. It holds the correction on the in-order forest pair.
Define array $CostPair[]$. It holds the cost of the correction on the in-order forest pair.
Define $OpTwoPair$. It holds the correction on an in-order forest pair that is
constructed from two.
Define $CostTwoPair$. It holds the cost of the correction on an in-order forest pair
that is constructed from two.
Define $i, j, m, n, k, l$. They hold the index of the node.
Define $I = 0$. It is the index of the in-order mapping to check.
If ($M == 0$)
    Apply [1]'s algorithm on the two incomplete trees.
    The generated correction is put in $OpCollection$.
else
{
  While ($I < M$) do
  {
```

```
$m = InOrderNodeIndex_1[I]$.
$n = InOrderNodeIndex_2[I]$.
**If** $(I == 0)$
        $i = 0$.
        $j = 0$.
**else**
        $i = InOrderNodeIndex_1[I-1]+1$.
        $j = InOrderNodeIndex_2[I-1]+1$.
**If** $(I == M-1)$
        $k = N_1 - 1$.
        $l = N_2 - 1$.
**else**
        $k = InOrderNodeIndex_1[I+1]-1$.
        $l = InOrderNodeIndex_2[I+1]-1$.
**If** ($OpPair[I]$ has not been computed)
        Apply [1]'s algorithm on the in-order forest pair, $(T_1[i..m-1], T_2[j..n-1])$.
        The correction is put in $OpPair[I]$, and the cost is put in $CostPair[I]$.
Apply [1]'s algorithm on the in-order forest pair, $(T_1[m+1..k], T_2[n+1..l])$.
The correction is put in $OpPair[I+1]$, and the cost is put in $CostPair[I+1]$.
Apply [1]'s algorithm on the in-order forest pair, $(T_1[i..k], T_2[j..l])$. (Note
that $T_2[n]$ is skipped in the computation.)
The generated correction plus a "copy $T_1[m]$ to $T_2[n]$" is put in $OpTwoPair$.
**If** ($OpTwoPair$ contains "delete $T_1[m]$")
        Combine the "delete" and the "copy" into a "move".
The cost of $OpTwoPair$ is put in $CostTwoPair$.
**If** $(CostTwoPair < CostPair[I] + CostPair[I+1])$
        Append $OpTwoPair$ to $OpCollection$.
```

```
            $I = I + 2$.
            **continue**.
    **else**
            Append $OpPair[I]$ to $OpCollection$.
            Append $OpPair[I + 1]$ to $OpCollection$.
            $I = I + 1$.
            **continue**.
  }
}
```

After using [1]'s algorithm to compute the tree correction on the two incomplete trees as above, we then search all the edit operations generated so far (including those from Step 1) for the pair of "delete a common node" and "copy this common node". Such a pair of operation is combined into a "move" operation. Finally, we iterate through the "move" operations to discover and remove unnecessary ones. This is done by evaluating the node pair in a "move" operation. If it is in order with the in-order mapping group constructed in Step 1 as well as all node pairs in "replace" operations, this node pair is actually an in-order mapping. The corresponding "move" operation is therefore unnecessary and removed. Step 2 concludes upon removing all unnecessary "move" operations.

Fig. 5.5 shows an example of a situation where the further evaluation of the mappings of Step 1 could lead to an improvement in the patch size. In this example, the two trees have two in-order mappings, $(T_1[7], T_2[7])$ and $(T_1[0], T_2[6])$.

Consider the mapping $(T_1[0], T_2[6])$. As shown in Table 5.1, if $(T_1[0], T_2[6])$ is considered an in-order mapping, "replace" edit operation is not possible between

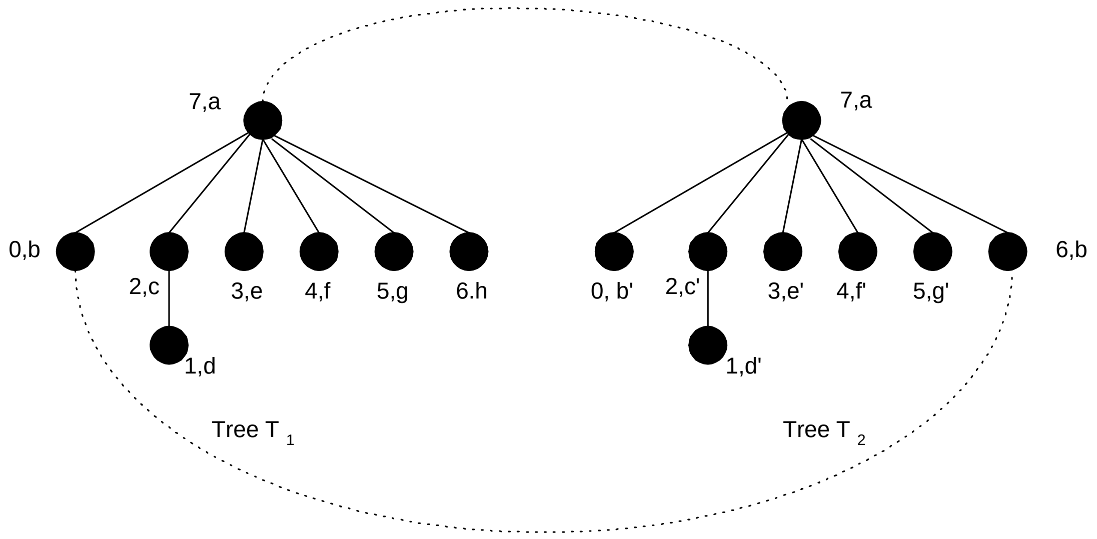


Figure 5.5: Two trees in comparison

$T_1[1,6]$ and $T_2[0,5]$. Consequently, six pairs of “insert” and “delete” operations are needed to transform the old tree to the new tree. However, if $(T_1[0], T_2[6])$ is considered an out-of-order mapping, we can have six “replace” edit operations to perform the correction, instead of the six pairs of “insert” and “delete”. The out of order mapping conversion leads to an additional cost of one “copy” edit operation and one “delete” operation. In this example, the method of Section 5.3.1 is used to encode the edit operations. The cost of each operation is given in Table 5.1. The result in Table 5.1 shows that transforming $(T_1[0], T_2[6])$ to a “copy” operation leads to a patch which is eight bytes smaller than the one where $(T_1[0], T_2[6])$ is considered an in-order mapping.

Let us now describe [1]'s algorithm used in this step.

[1] uses post order index. In [1], deleting a node, say $T[i]$, is depicted as $T[i]- > \wedge$, inserting a node, say $T[i]$, is depicted as $\wedge- > T[i]$, and replacing $T[i]$ with $T'[j]$ is

Table 5.1: Checking $(T_1[0], T_2[6])$ in the example in Fig. 5.5

	Case 1	Case 2
in-order mapping group	$(T_1[7], T_2[7])$,$(T_1[0], T_2[6])$	$(T_1[7], T_2[7])$
Edit operation and cost	insert b' before node 0 (4 bytes) insert c' before node 0(4 bytes) insert d' before node 0(4 bytes) insert e' before node 0(4 bytes) insert f' before node 0(4 bytes) insert g' before node 0(4 bytes) delete node 1 (1 bytes) delete node 2 (1 bytes) delete node 3 (1 bytes) delete node 4 (1 bytes) delete node 5 (1 bytes) delete node 6 (1 bytes)	copy node 0 after node 5 (3 bytes) replace node 0 with 'b' (3 bytes) replace node 1 with d'(3 bytes) replace node 2 with c'(3 bytes) replace node 3 with e'(3 bytes) replace node 4 with f'(3 bytes) replace node 5 with g'(3 bytes) delete node 6 (1 bytes)
Patch size	30 bytes	22 bytes

depicted as $T[i]->T'[j]$. [1] uses a function γ to assign cost to an edit operation. [1] uses $forestdis[i'..i, j'..j]$ to denote the minimum distance or correction cost from forest $T[i'..i]$ to forest $T'[j'..j]$. [1] uses $treedis[i, j]$ to denote the minimum cost to transform the subtree rooted at $T_1[i]$ to the subtree rooted at $T_2[j]$. Obviously $treedis[i, j] = forestdis[l(i)..i, l(j)..j]$. The computing of $forestdis[i'..i, j'..j]$ is given as Lemma 5 in [1], as shown below.

Let $i_1 \in ancestor(i)$ and $j_1 \in ancestor(j)$.

If $l(i) = l(i_1)$ and $l(j) = l(j_1)$,

$$\begin{aligned} forestdis(l(i_1)..i, l(j_1)..j) &= treedis(i, j) \\ &= Min\{forestdis(l(i_1)..i-1, l(j_1)..j) + \gamma(T_1[i_1] -> \wedge), \\ &\quad forestdis(l(i_1)..i, l(j_1)..j-1) + \gamma(\wedge -> T_2[j_1]), \\ &\quad forestdis(l(i_1)..i-1, l(j_1)..j-1) + \gamma(T_1[i_1] -> T_2[j_1]) \\ &\quad \}. \end{aligned} \tag{5.5}$$

If $l(i) \neq l(i_1)$ or $l(j) \neq l(j_1)$,

$$\begin{aligned} forestdis(l(i_1)..i, l(j_1)..j) &= Min\{forestdis(l(i_1)..i-1, l(j_1)..j) + \gamma(T_1[i_1] -> \wedge), \\ &\quad forestdis(l(i_1)..i, l(j_1)..j-1) + \gamma(\wedge -> T_2[j_1]), \\ &\quad forestdis(l(i_1)..l(i)-1, l(j_1)..l(j)-1) + treedis(i, j) \\ &\quad \}. \end{aligned} \tag{5.6}$$

[1] shows only the computation of the minimum tree distance. In our case, we record all paths traversed in the minimum tree distance computation, so that at the end of the computation we can determine the path corresponding to the minimum tree distance. We then construct the corresponding patch by retracing the minimum cost path. The algorithm is given as follows.

Inputs: tree T_1 and T_2

Output: The minimum correction cost and the corresponding patch.

Two global arrays:

1. treedis[][], with SizeOf(T_1)×SizeOf(T_2)elements. The element $treedis[i][j]$ is the minimum cost to transform subtree rooted at $T_1[i]$ to the subtree rooted at $T_2[j]$.
2. treeEditOperation[][], with SizeOf(T_1)×SizeOf(T_2) elements. The element

treeEditOperation[i][j] is a series of edit operations that transform subtree rooted at $T_1[i]$ to the subtree rooted at $T_2[j]$ with the cost of $treedis[i][j]$.

```
Main loop:
For each $i'$ from 1 to SizeOf($keyNodes_1$) do
    For each $j'$ from 1 to SizeOf($keyNodes_2$) do
        $i = keyNodes_1(i')$.
        $j = keyNodes_2(j')$.
        Compute treedis[i][j] and Compute treeEditOperation[i][j].
End Main loop
```

The minimum patch size cost is treedis[SizeOf(T_1)][SizeOf(T_2)], and the corresponding patch has edit operations in the element of treeEditOperation[SizeOf(T_1)][SizeOf(T_2)].

Compute treedis[i][j]:

The method is to compute the distance between forests in the two trees from bottom upwards. The forest distance is put into temporary memory.

```
**Initialization**:
$forestdis[\varphi, \varphi]$=0,
For each $i_1$ from l(i) to i do
    $forestdis(T_1[l(i)..i_1], \varphi) = forestdis(T_1[l(i)..i_1 - 1], \varphi) + \gamma(T_1[i_1] -> \wedge)$.
For each $j_1$ from l(j) to j do
    $forestdis(\varphi, T_2[l(j)..j_1]) = forestdis(\varphi, T_2[l(j)..j_1 - 1]) + \gamma(\wedge -> T_2[j_1])$.
For each $i_1$ from l(i) to i do
    For each $j_1$ from l(j) to j do
        Use Lemma 5 in [1] to compute $forestdis(l(i)..i_1, l(j)..j_1)$.
    $treedis[i][j] = forestdis(l(i)..i, l(j)..j)$.
End Computing treedis[i][j]
```

Compute treeEditOperation[i][j]:

```
Define $i_1 = i$; $j_1 = j$.
While ($i_1 \geq l(i)$and $j_1 \geq l(j)$) do
    If ($l(i_1) == l(i)$ AND $l(j_1) == l(j)$)
        If ($forestdis(l(i)..i_1, l(j)..j_1) == forestdis(l(i)..i_1 - 1, l(j)..j_1 - 1) +$
        $\gamma(T_1[i_1] -> T_2[j_1])$)
        Append edit operation $T_1[i_1] -> T_2[j_1]$ to treeEditOperation[i][j].
        $i_1 = i_1 - 1$.
        $j_1 = j_1 - 1$.
        continue.
        If ($forestdis(l(i)..i_1, l(j)..j_1) == forestdis(l(i)..i_1 - 1, l(j)..j_1) +$
        $\gamma(T_1[i_1] -> \wedge)$)
        Append edit operation $T_1[i_1] -> \wedge$ to treeEditOperation[i][j].
      If ($forestdis(l(i)..i_1, l(j)..j_1) == forestdis(l(i)..i_1, l(j)..j_1 - 1) +$
        $\gamma(\wedge -> T_2[j_1])$)
        Append edit operation $\wedge -> T_2[j_1]$ to treeEditOperation[i][j].
        $i_1 = l(i_1) - 1$.
        $j_1 = l(j_1) - 1$.
        continue.
    else
        If ($forestdis(l(i)..i_1, l(j)..j_1) == forestdis(l(i)..i_1 - 1, l(j)..j_1) +$
        $\gamma(T_1[i_1] -> \wedge)$)
        Append edit operation $T_1[i_1] -> \wedge$ to treeEditOperation[i][j].
      If ($forestdis(l(i)..i_1, l(j)..j_1) == forestdis(l(i)..i_1, l(j)..j_1 - 1) +$
        $\gamma(\wedge -> T_2[j_1])$)
        Append edit operation $\wedge -> T_2[j_1]$ to treeEditOperation[i][j].
        $i_1 = l(i_1) - 1$.
        $j_1 = l(j_1) - 1$.
        **continue**.
        **If** $(forestdis(l(i)..i_1, l(j)..j_1) == forestdis(l(i)..l(i_1) - 1, l(j)..l(j_1) - 1) + treedis(i_1, j_1))$
        Append $treeEditOperation[i_1][j_1]$ to $treeEditOperation[i][j]$.
        $i_1 = l(i_1) - 1$.
        $j_1 = l(j_1) - 1$.
        **continue**.
End computing treeEditOperation[i][j]
```

5.2.2 Combination method: a suboptimal algorithm to compute patch by dividing the problem in the instruction domain

The algorithm presented in the last subsection classifies nodes into groups and computes edit operations for them respectively. In this subsection, we present the second proposal, which approaches the patch generation problem in the instruction domain. This again is done in two steps.

In Step 1, we first relax the constraint that only new nodes can replace other nodes or be inserted, and replace it by loose versions; *insert** and *replace**. The "*insert**", "*replace**" instructions together with "delete" are used in [1]'s algorithm to generate a minimum "loose version patch". The nodes in the two trees that are not involved in the loose version correction construct an in-order mapping group. When we use [1]'s algorithm to compute the minimum tree correction, we assign cost to each edit operation. For the "*insert** a common node" and "*replace**

with a common node" operations, the cost is computed differently compared with the "*insert* a new node" and "*replace* with a new node" operations. This will be discussed later in this subsection.

In Step 2, we re-instate the constraints. Note that "*insert**" and "*replace**" edit operations bring some existing data (common node) into the loose version patch. The existing data are eliminated from the loose version patch by converting the "*insert**" and "*replace**" operations into "copy" and "move" operations. "Copy" and "move" operations are constructed from certain cases of the combination of "*insert** a common node" or "*replace** with a common node" and other operations with the same common node involved. The cases of combination are given as follows.

Case 1.

(Delete a common node)

(Insert* this common node)

Combined to

(Move the common node)

Case 2.

(Insert* a common node)

Transformed to

(Copy the common node)

Case 3.

(Delete common node A)

(Replace* old node B with common node A)

Transformed to

(Delete node B)

(Move common node A)

Case 4.

(Replace* old node A with common node B)

Transformed to

(Delete old node A)

(Copy common node B)

Case 5.

(Delete common node A)

(Replace* common node B with common node A)

Transformed to

(Delete common node B)

(Move common node A)

Case 6.

(Replace* common node A with common node B)

Transformed to

(Delete common node A)

(Copy common node B)

Each "insert*" and "replace*"edit operation in the loose version patch is checked against the above six cases. If one of the above six cases of combination is found, a "copy" or "move" operation is generated. Let N_e be the number of edit operations in the loose version patch generated with *insert**, *replace** and delete. The generation of copy and move by combination requires at most N_e^2 iterations. Since $N_e \leq |T_1| + |T_2|$, the time complex of the combination is $O((|T_1| + |T_2|)^2)$.

After generating "copy" and "move" operations by searching for the above six cases, we next search the result for pairs involving "delete a common node" and "copy this common node". If such a pair is found, it is combined into a "move" operation. Finally, we iterate through the "move" operations to discover and remove unnecessary ones. This is done by evaluating the node pair in a "move" operation. If it is in order with the in-order mapping group constructed in Step 1 as well as all node pairs in "replace" operations, this node pair is actually an in-order mapping. The corresponding "move" operation is therefore unnecessary and removed. Step 2 concludes upon removing all unnecessary "move" operations.

As mentioned before, we assign cost to each edit operation in Step 1. For "*insert* a new node" and "*replace* with a new node" edit operations, the size of the new node data affects the cost. However, for "*insert** a common node" and "*replace** with a common node" operations, the size of the common node would not affect the operation cost. This is because such an operation would eventually be converted to a "copy" or "move" operation in Step 2 and the common node would not appear in the final patch. Ideally, the cost assigned to such an operation is its actual contribution in the final patch after conversion. However, the actual contribution is dependent on the combination cases, and we do not know the results of the combination cases until Step 1 is completed. In this algorithm, we use the average contribution to estimate the actual cost and assign it to an "*insert** a common node" or "*replace** with a common node" edit operation. The assumption here is that the combination cases occur with the same probability.

Let C_C, C_M and C_D be the cost of a "copy", "move" and "delete" operation respectively.

In Cases 1 and 2 above, "copy" or "move" edit operations are constructed from the combination with "*insert** a common node". In Case 1, the "*insert**" operation's contribution in the final patch is $(C_M - C_D)$. In Case 2, the contribution is C_C. The average,

$$\frac{C_C + C_M - C_D}{2},$$

is thus assigned to an "*insert** a common node" operation as its cost in Step 1.

In Cases 3, 4, 5 and 6 above, "copy" or "move" edit operation is constructed from the combination with "*replace** with a common node". In Cases 3 and 5, the "*replace**" operation's contribution in the final patch is C_M. In Cases 4 and 6, the contribution is $(C_C + C_D)$. The average,

$$\frac{C_M + C_C + C_D}{2},$$

is thus assigned to an "*replace** with a common node" operation as its cost in Step 1.

5.2.3 Branch&bound method: an optimal algorithm to compute patch by searching the solution space

The optimal patch can be achieved by a brute force search. It searches all feasible solutions and picks the best one. Branch and bound [94] method is one way to reduce the searching workload. In the branch and bound method, every node in the search tree is assigned a weight value. At every node, the weight value is compared with some bounds to be determined. If the bound is exceeded, there is no need to go on to the branches of this node, thus reducing the search space. Branch and bound method is the same as the brute force search in the worst case, in which, the whole tree is traversed through.

In this subsection, a search tree is constructed while searching for the optimal patch. Each layer in the search tree corresponds to a node in the old web object tree. Each branch from a node in the search tree corresponds to an edit action applied to the corresponding node in the old web object tree. The search tree is constructed in pre-order. In the course of construction, two global variables, GMinimumCostCorrection and GMinimumCost, that represent the minimum cost correction so far and the minimum cost so far respectively are maintained. For a layer, say the ith layer, the following local variables are maintained:

1. The cost of edit operations generated so far (depicted as CurrentCost[i]).
2. Nodes in the new tree that are not involved in any edit operation yet (depicted as AvailableNodes[i]).
3. The possible edit operations applied on the node represented by the layer (depicted as AvailableEditOperation[i]).
4. The node pairs that are in order so far (depicted as MappingNodePairs[i]).

If the old web tree and the new web tree have N_1 and N_2 nodes respectively, the search tree has up to N_1 layers. Whenever traversing to a layer, say the ith layer, CurrentCost[i+1] is calculated, and compared with GMinimumCost. If CurrentCost[i+1] is larger, there is no need to go to the next layer, and the search goes back to the upper layer. Otherwise, AvailableNodes[i+1], MappingNodePairs[i+1] and AvailableEditOperation[i+1] are computed, and the search goes on to the next layer following the possible edit operations on the node. A node in the old tree can be deleted, replaced by a new node in the new tree, or form an in-order and out-of-order mapping with a common node in the new tree. When a leaf is reached in the search tree, the new nodes in the new tree that have not been involved in any edit operations so far, lead to "insert" operations, and the common nodes in the new tree that have not been involved in any edit operation so far, lead to "copy" and "move" operations. Tree-to-tree corrections are achieved at leaves. If the cost of the newly achieved correction is smaller than GMinimumCost, two global variables are updated.

The proposed algorithm is described as follows.

```
**Inputs**: tree $T_1$ and $T_2$
**Output**: The minimum tree correction.
Main loop:
**Initialization**:
  CurrentCost[0]=0.
  MappingNodePairs[0]=none.
  AvailableNodes[0]=All nodes in the new tree.
  Construct AvailableEditOperation[0].
Define nLayer=0.
**While** (nLayer≥0) **do**
  **If** (nLayer==$SizeOf(T_1)$, indicating a leaf is reached)
    Compute the corrections based on AvailableEditOperation[nLayer] and the rest
    of nodes in AvailableNodes[nLayer].
    **If** (a smaller correction is found)
      Update the global variables.
    $nLayer = nLayer - 1$.
    **continue**.
Label "check":
  **If** (AvailableEditOperation[nLayer] is empty)
    $nLayer = nLayer - 1$.
    **continue**.
  Take and remove one edit operation, $EditOp$, from AvailableEditOperation[nLayer].
  CurrentCost[nLayer+1]=CurrentCost[nLayer]+ the cost of $EditOp$.
  If ($CurrentCost[nLayer+1] \geq GMinimumCost$)
    go to label "check".
  AvailableNodes[nLayer+1]=AvailableNodes[nLayer] without the nodes in the
  new tree involved in $EditOp$.
  If ($EditOp$ is Replace OR in-order mapping)
    MappingNodePairs[nLayer+1]=MappingNodePairs[nLayer].
  else
    MappingNodePairs[nLayer+1]=MappingNodePairs[nLayer]+
      the node pair in $EditOp$;
  Construct AvailableEditOperation[nLayer+1] from $T_1[nLayer+1]$ and
    AvailableNodes[nLayer+1].
  $nLayer = nLayer + 1$.
  continue.
End Main loop

Construct AvailableEditOperation[nLayer]:
Take one node from AvailableNodes[nLayer] and denote it as $T_2[j]$.
  Define $bInOrder = FALSE$.
  If (($T_1[nLayer], T_2[j]$) is in order with all the node pairs in MappingNodePairs[nLayer])
    $bInOrder = TRUE$.
  else
    $bInOrder = FALSE$.
  If ($T_2[j]$ is a not common node AND $bInOrder == TRUE$)
    AvailableEditOperation[nLayer]+=a replace operation.
  If ($T_1[nLayer] == T_2[j]$)
    AvailableEditOperation[nLayer]+= a Copy/Move operation.
    If ($bInOrder == TRUE$)
      AvailableEditOperation[nLayer]+=a in-order mapping.
Take next available node and repeat above process.
End Constructing AvailableEditOperation[nLayer]
```

Let $P(m,n)$ be the maximum number of leaves in a sub search tree whose root corresponds to a situation where m and n nodes in the old and new web object trees are available respectively. Suppose one node of the m old tree nodes is to be edited. There are three possible cases when editing the node. Case 1 is to delete the node. Then on the next layer we have $m-1$ and n nodes available in the old and new trees respectively. Case 2 is that the old tree node and a common node from the new tree form a mapping. If this node pair is in-order with all existing in-order ones, an in-order mapping is constructed. At the same time, an out of order mapping is constructed. This is to avoid the possible blocking of in-order mappings subsequently. In this case , on the next layer, we have m-1 and n-1 available nodes in the two trees. Case 3 is that this node is replaced by a "new" node from among those n nodes of the new tree under the condition that the pair is in-order with all existing ones. Then, on the next layer we have $m-1$ and $n-1$ available nodes in the two trees. In other words, from the sub search tree root we have up to $(1 + 2 \times n)$ branches. The first branch is a delete operation. The rest of the $(2 \times n)$ branches are constructed together with the available new tree nodes at this layer. We thus have

$$\begin{aligned} p(m,n) &= p(m-1,n) + 2n \times p(m-1,n-1) \\ &= p(0,n) + 2n \times \sum_{i=0}^{m-1} p(i,n-1), if\, m \times n \neq 0. \\ p(m,n) &= 1, if\, m \times n = 0. \end{aligned} \tag{5.7}$$

With Equation 5.7, we have

$$p(m,1) = 1 + 2m$$
$$p(m,2) = 1 + 4m + 4m^2$$
$$p(m,3) = 1 + 10m + 8m^3.$$

Theorem 5.2. *p(m,n) is an m's n-degree polynomial and the coefficient of m^n is* 2^n.

Proof. It can be proven by induction:
Suppose $p(m,k) = 2^k m^k + ...$, then we have

$$\begin{aligned} p(m,k+1) &= 1 + 2(k+1) \times \sum_{i=0}^{m-1} p(i,k) \\ &= 1 + 2(k+1) \times \sum_{i=0}^{m-1} (2^k i^k + ...). \end{aligned}$$

From Bernoulli [95, 96], we have $\sum_{i=1}^{n} i^k$ is n's k+1 degree polynomial and the coefficient of n^{k+1} is 1/(k+1). The above equation then becomes

$$\begin{aligned} p(m,k+1) &= 1 + 2(k+1) \times 2^k \times \frac{1}{k+1}(m-1)^{k+1} + ... \\ &= 1 + (2m-2)^{k+1} + ..., \end{aligned}$$

leading to, $p(m,k+1)$ is m's k+1 degree polynomial and the coefficient of m^{k+1} is 2^{k+1}. □

To reach a leaf in the search tree, at most $|T_1|$ layers are traversed, so the time complexity of the branch and bound method is $O(p(|T_1|, |T_2|) \times |T_1|)$. The high time-complexity makes the branch and bound method impractical when the tree size is large. For example $p(20, 20) = 1.94211772044854 \times 10^{26}$. If a computer can traverse 1,000,000,000 leaves in one second, about 6,158,414,892 years are needed to finish a full search.

As shown in Table 5.2, the complexity of the branch and bound method grows at the rate of the order of m^n. This makes the branch and bound method impractical for large trees. In comparison, the two suboptimal algorithms compute patch in polynomial time in two steps.

Table 5.2: Patch algorithms' complexity

Algorithms	Complexity
Step 1 of maximum number in-order mapping method	$O(\|T_1\| \times \|T_2\| \times min(depth(T_1), leaves(T_1)) \times min(depth(T_2), leaves(T_2)))$
Step 2 of combination method	$O((\|T_1\| + \|T_2\|)^2)$
branch&bound method	$O(p(\|T_1\|, \|T_2\|) \times \|T_1\|)$
[1]'s algorithm	$min(depth(T_1), leaves(T_1)) \times min(depth(T_2), leaves(T_2)))$

5.3 Evaluation experiments

5.3.1 Methodology

227,802 URLs pointing to html files were retrieved from squid access log files and IRcache access log files available online [97]. 130,078 of the retrieved 227,802 URLs are accessible. The first version of each available URL was fetched and then in the following 87 days, "If-Modified-Since" conditional request were sent to them. Once updates were found, the updated files were fetched. The updating history is saved in a database. Those valid URLs were updated a different number of times during the 87 days (Table 5.3).

Table 5.3: % of URL VS. update times

update times	0	1	2	3	4
% of URL	66.0%	12.2%	5.0%	2.9%	2.2%
update times	5	6	7	8	9
% of URL	1.4%	1.0%	0.9%	0.8%	0.8%
update times	10	11	12	13	14
% of URL	1.1%	1.0%	1.6%	1.5%	1.5%

After fetching the original web files, the patches for each subsequent update were computed using the two suboptimal methods described. For a URL that was updated n times, we have version 1 (oldest), version2, ..., and version n+1 (newest). A patch with i version span was computed on the files of version n+1-i and n+1. Using each suboptimal patch algorithm and the algorithm of [1], 162,053 patches with 1 to 14 version span were computed. For the files that have less than 12 nodes, the branch and bound method was also used and correspondingly 4,486 patches were computed. Note that not all updates have corresponding patches due to virus infection or failed file transmission.

Table 5.4 gives the elements needed to encode an edit operation.

Table 5.4: Elements in an edit operation

Type	Elements in the edit operation
Delete	Instruction, applying position
Insert	Instruction, applying position, Object-type label of new node, length of object-content label, object-content label, number of the descendants of a node, number of other constructed nodes in between.
Replace	Instruction, applying position, Object-type label of new node, length of object-content label, object-content label,
Copy/ Move	Instruction, applying position, position of the common node, number of the descendent of a node, number of other constructed nodes in between.

This experiment uses the node position in the old tree as the applying position of an edit operation. For “insert”, “copy” and “move”, it is needed to indicate if the applying position is before or after a node in the old tree. We have eight instructions, namely ‘delete”, “replace”, “move before ”,“insert before”, “copy before”, “move after ”, “insert after” and “copy after”. Three bits are used to encode them. $ceil(log_2|T_1|)$ bits are used to encode the applying position. The function ceil(x) returns the smallest integer that is greater than or equal to x. These two elements were put together as $part_1$ of a coded edit operation since they are shared by each type of operation. Since the minimum unit of data to save and transfer online is byte, the length of $part_1$ in byte is $ceil(3 + ceil(log_2|T_1|)/8)$.

This experiment uses two bits to encode four types of Object-type label, $ceil(log_2|File_2|)$ bits to encode the length of object-content label ($File_2$ is the new version web object). These two elements appear in an insert or a "replace" operation, which are put together as $part_2$. The length of $part_2$ in byte is $ceil(2 + ceil(log_2|File_2|)/8)$.

"Copy" and "move" operations need the position of the existing node to copy or move. $ceil(log_2|T_1|)$ bits are used to encode this element and this becomes $part_3$ of a coded operation. The length of $part_3$ is $ceil(ceil(log_2|T_1|)/8)$.

"Insert", "copy" and "move" edit operations put a node in a tree. Some structure description information is needed in these edit operations to construct the node's relationship with the existing ones. As defined in Section 3.4, such information includes "before" or "after", "child", "parent" or "sibling", the number of descendent of the node, and the number of other constructed nodes in between. In our experiment, the "before" or "after" information is already encoded in $part_1$. Since we use post order index and compare the two trees from bottom upwards, we do not need the "child", "parent" or "sibling" information. A node put before an existing node cannot be the parent of the existing node since it will have a smaller post order index in the new tree. If we know the number of the descendent of the next node after this newly constructed node in the new tree and the number of other constructed nodes between this constructed node and the existing node, we can construct the relationship between this node and the existing ones. In the case of putting a node after an existing node, the node cannot be the child of the existing one because it will have a bigger post order index in the new tree. Similarly, if we know the number of the descendent of this newly constructed node in the new tree and the number of other constructed nodes between it and the existing node, we can put the node into the new tree structure.

As shown in the last paragraph, the number of the descendent of a node and the number of constructed nodes in between are necessary for "insert", "copy" and "move" edit operations to make the construction to be the complement of destruction. $ceil(2 \times log_2|T_2|)$ bits are used to encode them and this gives $part_4$ of a coded operation.

The object-content label, which is plain text, will not be encoded.

Below is an example of tree-to-tree correction. The old tree T_1 and the new tree T_2 are depicted in Fig. 5.6. Both of them have 6 nodes. The post order index and label are given beside the node. The text above the tree is the web content from which the tree is constructed.

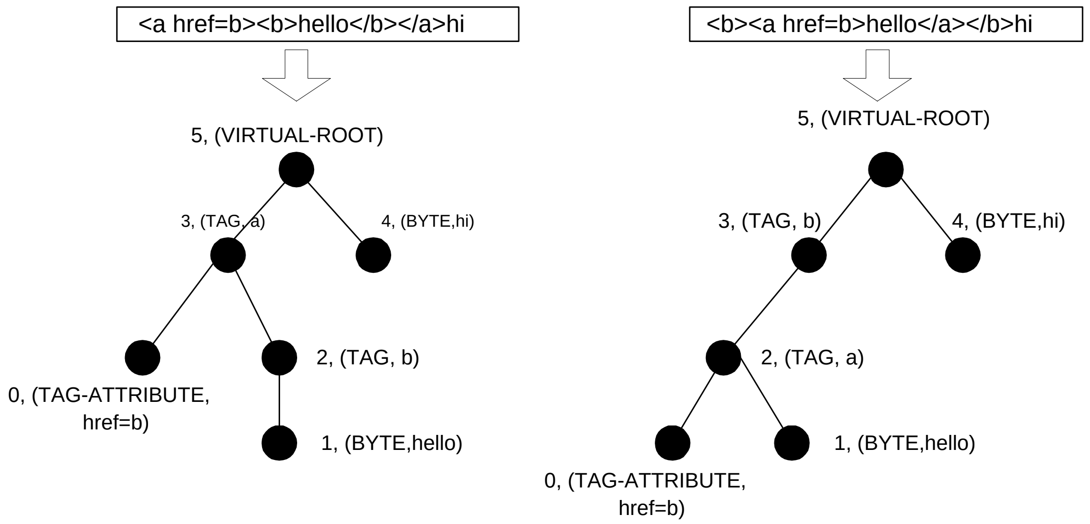


Figure 5.6: An example of tree to tree correction

Algorithm [1] gives the following edit operations: *delete node 2 in T_1 and insert* (loose version) a TAG $< a >$ after node 3 in T_1.* The insert* operation costs 4 bytes and the delete operation costs 1 byte. Total cost of the tree correction using [1] is 5 bytes.

In Step 1 of maximum number in-order mapping method, five in-order mappings were found. They are $(0,0)(1,1)(3,2)(4,4)$ and $(5,5)$. The mapping $(2,3)$ is out of order and leads to a move edit operation which costs 3 bytes. No node is left for Step 2, and this method gives a 3 bytes tree correction.

Combination method works on the result of [1]'s algorithm. The delete and insert* operation are combined into a move edit operation which costs 3 bytes.

The branch&bound method constructs a solution tree of 108 leaves and gives the optimal result consisting of a move operation with a cost of 3 bytes.

In this example, both suboptimal patch algorithms gave an optimal patch. The patch is smaller than the patch generated using algorithm [1].

5.3.2 Patch size VS. original new version file size

The average size ratio of patch to original new version file is given in Table 5.5.

Table 5.5 shows that the average patch size ratios by the two suboptimal algorithms have at most 0.05% difference at each version span. Overall, maximum number in-order mapping method and combination method give the average patch size ratio of 20.50% and 20.51% respectively.

Let us look at the distribution of patch over the size ratio. We divide the size ratio into 4 ranges: [0-25%], (25%-50%], (50%-100%) and 100%(inclusive) onwards. We counted the patches falling into these 4 ranges for each suboptimal algorithm. As shown in Fig. 5.7, only about 2% of patches are larger than their original files; over

Table 5.5: Average patch size ratio

version span	1	2	3	4	5
Algorithm 1	19.30%	20.84%	21.79%	21.35%	21.13%
Algorithm 2	19.28%	20.83%	21.79%	21.36%	21.15%
SimpleFast Algorithm	19.62%	21.66%	22.84%	22.52%	22.39%
version span	6	7	8	9	10
Algorithm 1	20.86%	20.89%	20.71%	20.82%	20.58%
Algorithm 2	20.88%	20.91%	20.74%	20.85%	20.61%
SimpleFast Algorithm	22.15%	22.15%	21.93%	22.02%	21.73%
version span	11	12	13	14	
Algorithm 1	20.27%	18.55%	18.27%	18.20%	
Algorithm 2	20.30%	18.59%	18.32%	18.20%	
SimpleFast Algorithm	21.44%	19.67%	19.39%	19.26%	
Overall patch size ratio					
Algorithm 1	20.50%				
Algorithm 2	20.51%				
SimpleFast Algorithm	21.44%				

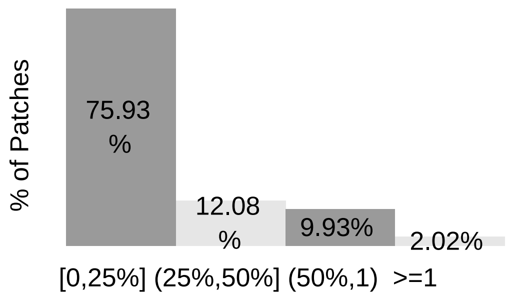
Algorithm1: Maximum number in-order mapping method(suboptimal)

Algorithm2: Combination method(suboptimal)

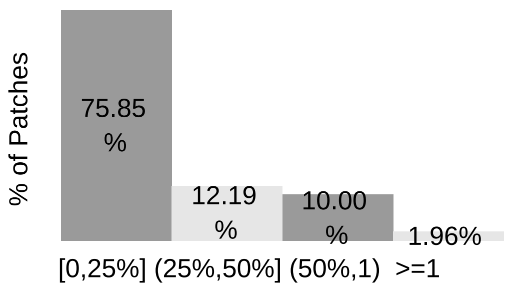
88% of patches are smaller than half of their original files; about 76% of patches are smaller than one quarter of their original files.

5.3.3 Suboptimal algorithms VS. optimal algorithm

Table 5.2 compares the two suboptimal algorithms and the optimal algorithm in terms of time complexity. This section examines how they perform in terms of patch size.



(a) Max.No.in-order method



(b) Combination method

Figure 5.7: Patch distribution over size ratio

To illustrate the performance of suboptimal algorithms compared with the branch and bound method, the patches generated on trees with less than 12 nodes are examined.

Among those 4,486 patches generated with maximum number in-order mapping

Table 5.6: Average patch size ratio on files with less than 12 nodes

	Algorithm1	Algorithm2	Algorithm3
Average Size Ratio:	12.87702%	12.87702%	12.87702%

Algorithm1: Maximum number in-order mapping method

Algorithm2: Combination method

Algorithm3: Branch&Bound method

method on the trees of less than 12 nodes, none is larger than the optimal patch. The same is the case of combination method. The result is given in Table 5.6.

5.3.4 Suboptimal algorithms VS. [1]'s algorithm

As shown in Section 5.2 and Section 5.3.1, the two suboptimal algorithms extend the instruction set used in [1] to avoid existing data appearing in the patch. While this is an improvement, more bits are needed to encode the new instruction. Such trade-off is checked among the generated patches, and the three algorithms' performance in terms of the average patch size ratio is given in Table 5.7.

Table 5.7: Two suboptimal algorithms VS. [1]'s algorithm

Algorithm	Average patch size ratio	Improvement
Algorithm [1]	21.44%	N.A.
Maximum number in order mapping method	20.50%	4.37%
Combination method	20.51%	4.33%

For **49%** of the updates, the two suboptimal methods and [1]'s algorithm yield the patches of the same size. For **46%** of the updates, the maximum number in-order mapping method outperforms [1]'s algorithm with smaller patches. And for **48%** of the updates, the combination method outperforms [1]'s algorithm with smaller patches. The largest improvement on [1]'s algorithm by the two suboptimal algorithms is 99.6%, where [1]'s algorithm generated a patch of 9,060 bytes, while both suboptimal algorithms generated a patch of 32 bytes. In terms of average patch ratio in this experiment, the improvement is not large (Table 5.7). The reason is that most of the updates are minor. Due to the minor changes on the tree structure and correspondingly few node exchanges, [1]'s algorithm can detect most of the in-order mappings and leaves little room for the two suboptimal algorithms to improve the patch size.

5.3.5 substantializing the benefits

In this subsection, the experimental result is utilized to substantialize the benefits from the proposed caching system discussed in Section 2.4.

Based on our experiment, the average web object file size is 12KB, the average transmission ratio on the inter-cluster link is 10KBps, and the average transmission ratio on the intra-cluster link is 100KBps.

If the original web server maintains patches for the latest 5 old versions, $P_{cc} = 23.7\%$. We substantialize the parameters defined in Chapter 2 as follows:

$$\begin{aligned} P &= 34\% \\ P_{cc} &= 23.7\% \\ S_o = S_c &= 12K \\ r_o &= 10KBps \\ r_c &= 100KBps \end{aligned} \tag{5.8}$$

cache hit rate

As discussed in Section 2.4.1, the proposed caching system decreases the cache consistency miss rate by relaxing the cache consistency criteria. Based on the experiment result, the consistency miss rate is reduced by

$$\frac{P_{cc}}{P} = 70\%. \tag{5.9}$$

inter-cluster traffic

As discussed in Section 2.4.2, if a cached copy is provided within the cluster locally (cases 1 and 2 in Section 2.4.2), the inter-cluster traffic is reduced. The inter-cluster traffic in fulfilling a web request is reduced by

$$\begin{aligned} S_{Inter} - S_{InterDWEBC} &= P_{cc} \times (S_o - S_p) \\ &= 0.237 \times (17,000 - 17,000 \times 0.205) \\ &= 3203 bytes \end{aligned} \tag{5.10}$$

response time

As discussed in Section 2.4.3, if a cached copy is provided within the cluster locally (cases 1 and 2 in Section 2.4.3), a shorter response time is achieved. The response time improvement is

$$\begin{aligned} T_{NoDWEBC} - T_{DWEBC} &= P_{cc}(\frac{S_o}{r_o} - \max(\frac{S_c}{r_c}, \frac{S_p}{r_o})) \\ &= 0.237 \times (\frac{12}{10} - \max(\frac{12}{100}, \frac{12 \times 0.205}{10})) \\ &= 0.23 second. \end{aligned} \tag{5.11}$$

5.4 Chapter summary

In this chapter, we compute web patch in tree space with a fixed instruction set. Two suboptimal patch algorithms and one optimal algorithm are proposed. The patch experiment was conducted, and the results show that most updates are minor. The average size ratio of patch to original new version file is about 20% on average under both suboptimal algorithms. For those web pages with less than 12 nodes, experiment results show that both suboptimal algorithms perform nearly optimally. Using patch to update web object in an incremental way is meaningful with respect to the reduced size of data to transmit and correspondingly the reduced response time and network traffic.

Chapter 6

Patch for dynamic document

As shown in Section 2.3.3, dynamic document caching is meaningful under the incremental update and delivery scheme. In the incremental update and delivery scheme, a requesting client does not differentiate if the web object is dynamic or not. It only determines whether a cached copy and a corresponding patch, regardless of how it is generated, are available. The difference is that the original server computes and provides patches in the dynamic situation in real-time upon a request, whereas in the static situation, it is usually computed upon an update of the original document, and often not in real-time.

The patch generation algorithms proposed in Chapters 4 and 5 address general and random changes on a web object. The objective is to achieve the minimal patch size with a considerable time complexity. This makes them not suitable in dynamic applications.

In this chapter we present strategies towards using patches for dynamic documents. In many dynamic documents, some static content or structure may exist between consecutive versions of the documents. This additional information can be utilized to make patch generation possible in real-time. Although the patches generated may not be minimal, the techniques are useful in reducing bandwidth besides unifying support for both static and dynamic documents in the proposed caching system.

As shown earlier, a document can be dynamic in the requesting client domain or the time domain. This chapter discusses patch generation for dynamic documents in these two cases.

6.1 Time domain

In some dynamic information delivery applications, the information data is updated frequently. One example is stock prices of the stock exchange portal. Usually the server runs a server end script to fetch the data from the database, organize them into HTML format, and send the HTML data to the client. If the database is updated in real-time, the same server-end script may deliver different documents within a very short period of time. In such a real-time situation, it is not practical to generate a patch for an old version once the database is updated.

The content of such a dynamic document can be classified into two parts; dynamic information and framework data. The framework data usually refer to the layout description and function description. The layout description defines how the dynamic information is displayed to the client. The function description, for example, java script functions, defines the function triggered by some layout component. The framework data may be a large portion of the whole file.

Although dynamic information changes in a real-time manner, the framework data, which is possibly the bulk of the document, may remain unchanged for a long period of time. In the case that only the dynamic information data is updated across many versions, the original server can check the cache's time stamp which is included in the patch query or patch request header. If the cached copy has up-to-date framework data, the original server sends a patch to refresh the dynamic information data. Such a patch is not a correction on a particular old version of the document, but a supplement to the non-changing framework data. The patch is a complete refresh of the dynamic information data, and it can be a series of replace commands on all the nodes corresponding to the dynamic information data. Since there is no computing, but only retrieving and reorganization of the dynamic information data, patch generation is fast. Moreover, if the dynamic information data is a small portion of the whole document, delivering patch for a dynamic document still makes sense to reduce response time.

6.2 Requesting client domain

Responding differently for different requesting client is a customization issue. The content in such a dynamic document can be classified into the customization data and the universal data. The customization data may be different from client to client, while the universal data, once cached, can be used by any requesting client. One example is that of the search engine. It may use the same layout to present different query results.

An easy method to update the stale cached copy of a dynamic document is to refresh the customization data. To generate a patch on customization data, the original server needs to know the inputs that lead to the cached customization data. As shown in Section 2.3.3, the "Reference-URL" header field is designed for this usage.

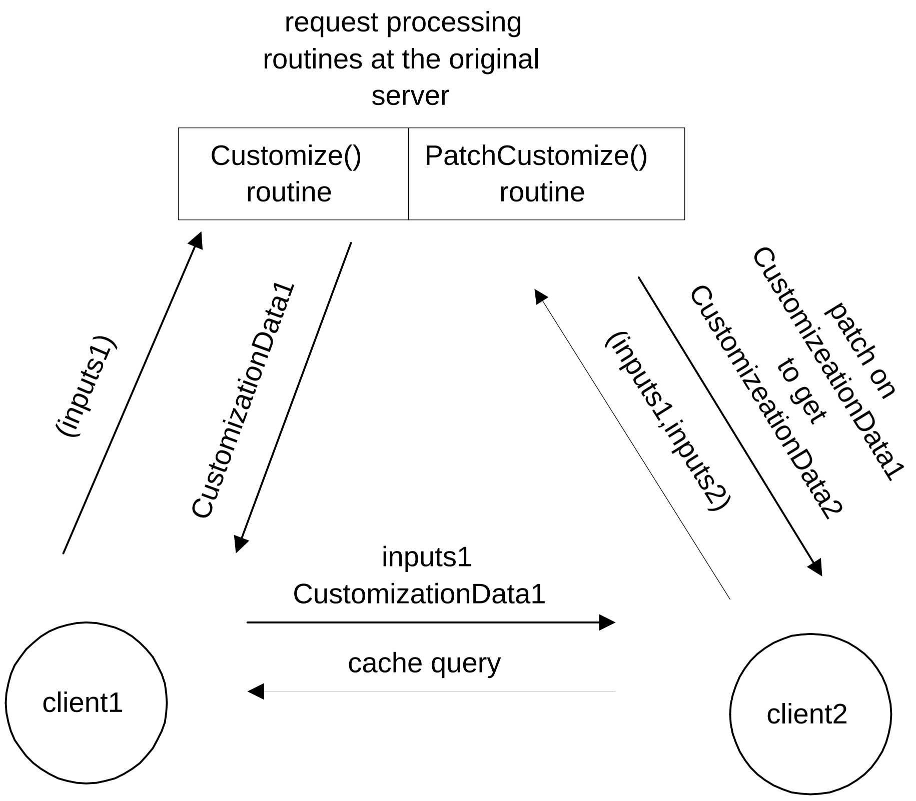


Figure 6.1: An illustration of real-time patch generation for customized document

The request processing program on a server takes user inputs and delivers customization data accordingly. It can be viewed as a function, say, $CustomizationData = Customize(Inputs)$.

Suppose an earlier response, $CustomizationData_1 = Customize(Inputs_1)$, is cached by a C-DWEBC module, and now a new request with new inputs of $Inputs_2$ is made. The patch request will be sent to the server with $Inputs_2$ and $Inputs_1$. $Inputs_1$ is in the "Reference-URL" field and the time stamp of $CustomizationData_1$

is in the requesting header. The original server is expected to deliver

$$Customize(Inputs_2) - Customize(Inputs_1).$$

This can be done by a routine, say $PatchCustomize()$. Once a patch request with $(Inputs_1, Inputs_2)$ is received, this routine is invoked to compute the patch and to return it to the requesting client. The requesting client uses the patch to update the cached $CustomizationData_1$, regenerating $CustomizationData_2$. Such a scenario is illustrated in Fig. 6.1.

6.3 Chapter summary

This chapter presents the possible approaches to support dynamic document under the proposed scheme of this thesis. Given the dependencies of such documents on the underlying application, a general method that minimizes patch size without considering the underlying structure is difficult. However, the framework proposed in this chapter should be useful for many situations.

Chapter 7

Conclusions

In this thesis, we propose a peer distributed web caching system with incremental update and delivery scheme. In the system, clients share their local caches with peers in a distributed manner. This is to utilize the perishable computation power and the cache storage on nearby peer clients to achieve large cache storage and to provide a close cache source. The incremental update and delivery scheme allows an original server to publish a patch to update stale caches. This utilizes the coherence among web page versions to improve cache usage.

Chapter 2 in this thesis describes the proposed system. It presents the protocol, discusses the implementation issues and analyzes its benefits. The proposed protocol runs on top of the TCP/UDP layer. It introduces new HTTP header fields for the original server and the client to exchange patch information. Cache control header field is used to ensure the end-to-end delivery of patches. The proposed caching system increases the cache hit rate by relaxing the cache consistency criteria, it also alleviates the inter-cluster network congestion and improves the response time by reducing the data to transmit across clusters. Moreover, the real time independent patch decoding property allows clients to experience a short data converting delay. These benefits are achieved at the cost of the patch computation and the increased intra-cluster traffic. Although the intra-cluster connection is less inclined to become the network bottleneck, measures are proposed in Chapter 2 to reduce the amount of the newly introduced intra-cluster traffic to improve system's scalability. Chapter 2 also analyzes the service reliability and shows how it can be improved by redundancy.

In this thesis, the patch generation problem is recast as a tree-to-tree correction problem by transforming web objects into ordered labelled trees. The transformation between a web object and a tree is given in Chapter 3. To have a minimal patch size, Chapter 4 models the minimal web patch with dynamic instruction set as the minimal set cover problem with dynamic weights. Under some assumptions, the approximation solutions of the minimal set cover problem with fixed weights are used to solve the minimal web patch problem. To achieve a smaller and applicable time complexity, Chapter 5 proposes algorithms to generate web patch with a fixed instruction set. In the evaluation experiment, over 200,000 URLs were checked for updates periodically in 87 days and 162,053 patches were computed. The results show that most updates are minor. The average size ratio of patch to original fresh version file is about 20% on average. Using patch to update web object in an incremental way is meaningful with respect to the reduced size of data to transmit and correspondingly the reduced response time and network traffic.

The proposed system supports dynamic documents. Chapter 2 shows how a dynamic document is cached and patched in the incremental update scheme. A simple discussion on patch generation for dynamic documents is given in Chapter 6. By exploiting the knowledge of the static structure in a dynamic document, the patch can be generated online as a simple replacement of dynamic content.

Bibliography

[1] Zhang, K., , Shasha, and D., “Simple fast algorithms for the editing distance between trees and related problems,” *SIAM J. Comput. 18 and 6(December 1989)*, pp. 1245–1262, 1989.

[2] G. Barish and K. Obraczka, “World wide web caching and trends and techniques,” *IEEE communication magazine*, May 2000.

[3] L. Zhang, S. Floyd, and V. Jacobson, “Adaptive web caching,” in *2nd Web Caching Workshop*, (Boulder, Colorado), June 1997.

[4] C.-Y. Chiang, M. T. Liu, and M. E. Muller, “Caching neighborhood protocol: A foundation for building dynamic caching hierarchies with www proxy servers,” in *Proceedings of the 29th International Conference on Parallel Processing*, (Aizu, Japan), pp. 516–523, sept 1999.

[5] `http://www.w3.org/History/1980/Enquire/manual`.

[6] P. Rodriguez, C. Spanner, and E. W.Biersack, “Analysis of web caching architectures: Hierarchical and distributed caching,” *ACM transactions on networking and Vol. 9 and No. 4*, August 2001.

[7] F. J. Hill and G. R. Peterson, *Digital Systems: Hardware Organization and Design*. Wiley, 1987.

[8] M. Murdocca and V. P. Heuring, *Principles of Computer Architecture*. Prentice Hall, 1999.

[9] G. Glass and P. Cao, “Adaptive page replacement based on memory reference behavior,” in *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, (Seattle, Washington, United States), pp. 115–126, 1997.

[10] K. L. Pei Cao, Edward W. Felten, Anna R. Karlin, “A study of integrated prefetching and caching strategies,” pp. 188–197.

[11] P. Scheuermann, J. Shim, and R. Vingralek, “Watchman : A data warehouse intelligent cache manager,” in *VLDB’96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pp. 51–62, Morgan Kaufmann, 1996.

[12] A. Mahanti, “Web proxy workload characterisation and modelling,” Sept. 1999. Master’s thesis, Department of Computer Science, University of Saskatchewan.

[13] P. Rodriguez, K. W. Ross, and E. W. Biersack, “Improving the WWW: caching or multicast?,” *Computer Networks and ISDN Systems*, vol. 30, no. 22–23, pp. 2223–2243, 1998.

[14] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, , and P. Sturm., "World wide web caching - the application level view of the internet," *IEEE Communications Magazine*, June 1997.

[15] B. D. Davison, "A survey of proxy cache evaluation techniques," in *Proceedings of the Fourth International Web Caching Workshop (WCW99)*, (San Diego, CA), pp. 67–77, 1999.

[16] Z. Jiang, L. Chang, B. J. Kim, and K. Leung, "Incorporating proxy services into wide area cellular ip networks," *Wireless Communications and Mobile Computing*, vol. Vol 1, No. 3, 2001.

[17] N. Yeager and R. McGrath, *Web Server Technology*. Morgan Kaufman, 3 ed., 1996.

[18] S. Glassman, "A caching relay for world wide web," *Computer Networks and ISDN Systems*, November 1994.

[19] B. Li, X. Deng, M. J. Golin, and K. Sohraby, "Dynamic and distributed web caching in active networks," in *Asia Pacific Web Conference (APWeb98)*, 1998.

[20] Z. Liang, H. Hassanein, and P. Martin, "Transparent distributed web caching," in *Proceedings of the IEEE Conference on Local Computer Networks*, pp. 225–233, Nov. 2001.

[21] B.Williams, "Transparent web caching solutions," in *proceedings of the 3rd international WWW caching workshop*, 1998.

[22] E. Johnson, "Increasing the performance of transparent caching with content-aware cache bypass," in *Proceedings of the 4th International Web Caching Workshop*, (San Diego, California), 1999.

[23] Z. Liang, “Transparent web caching with load balancing,” Master’s thesis, Queen’s University, March 2001.

[24] P. Krishnan and B. Sugla, “Utility of co-operating web proxy caches,” in *Proceedings of the 7th International WWW Conference*, 1998.

[25] P. Rodriguez and E. Biersack, “Bringing the web to the network edge: Large caches and satellite distribution,” 2000.

[26] M. Busari, “Simulation evaluation of web caching hierarchies,” June 2000. M.Sc. Thesis, Department of Computer Science, University of Saskatchewan.

[27] V. Valloppillil and K. W. Ross, “Cache array routing protocol v1.1. internet draft.” `http://ds1.internic.net/internetdrafts/draft-vinod-carp-v1-03.txt`, 1998.

[28] N.G.Smith, “The uk national web caching - the state of art,” in *5th int’l. conf. on World-wide web andcomp. Networks and ISDN sysSystems*, (Paris and France), May 1996.

[29] M. Liu, F.-Y. Wang, Zeng, D., and L. Yang, “An overview of world wide web caching,” in *Systems and Man and and Cybernetics and 2001 IEEE International Conference on and Volume: 5 and 2001*, 2001.

[30] A. C. et al., “A hierarchical internet object cache,” in *1996 USENIX Winter Tec. Conf.*, (an Diego and CA), Jan. 1996.

[31] H. che, Y. Tung, and Z. Wang, “Hierarchical web caching system: Modeling and decision and experimental results,” *IEEE Journal on selected areas in communications*, September 2002.

[32] `http://www.linofee.org/~elkner/proxy/Squid/icp-id.html`.

[33] "National lab of applied network research." `http://ircache.nlanr.net`.

[34] D. S.G., J. C.L., and D. S., "Taxonomy and design analysis for distributed web caching," in *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences, 1999*, vol. Track8, p. 10, Jan 1999.

[35] T. R. Dahlin, M. V. H.M., and K. J.S., "Design consideration for distributed caching on the internet," in *Proceedings 19th IEEE International Conference on Distributed Computing Systems*, pp. 273–284, 1999.

[36] H. A. and M. A., "Webwave: globally load balanced fully distributed caching of hot published documents," in *Proceedings of the 17th International Conference on Distributed Computing Systems, 1997*, pp. 160–168, May 1997.

[37] C. Spanner, "M.s. thesis: Evaluation of web caching strategies: Distributed vs. hierarchical caching," Master's thesis, University of Munich/Institute Eurecom, Sophia Antipolis and France, Nov. 1998.

[38] L.Fan, P.Cao, J.Almedia, and A.Broder, "summary cache: A scalable wide area web cache sharing protocol," in *proc. SIGCOMM'98*, pp. 254–265, Feb. 1998.

[39] A. Rousskov and D. Wessels, "cache digest," in *proc. 3rd int. WWW caching workshop*, pp. 272–273, June 1998.

[40] H. Hassanein, Z. liang, and P. Martin, "Performance comparison of alternative web caching techniques," in *ISCC02*, 2002.

[41] D. Karger, A. Sherman, A. Berkhemier, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, , and Y. Yerushalmi, "Web caching with consistent hashing," in *Proc. 8th Int. World Wide Web Conf.*, pp. 254–265, May 1999.

[42] K. W. Ross, "Hash-routing for collections of shared web caches," *IEEE Network Magazine*, vol. 11, 7, pp. 37–44, Nov-Dec 1997.

[43] `http://squid.nlanr.net/Squid/`.

[44] M. Makpangou, G. Pierre, C. Khoury, and N. Dorta, "Replicated directory service for weakly consistent replicated caches," in *19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, (Austin Texas), May 1999.

[45] A. R. Scott Michel, Khoi Nguyen and L. Zhang, "Adaptive web caching: Towards a new global caching architecture," *Computer Networks and ISDN System*, vol. 30, no. 22-23, pp. 2169–2177, 1998.

[46] C.-Y. Chiang, Y. Li, M. T. Liu, and M. E. Muller, "On request forwarding for dynamic web caching hierarchies," in *The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, (Taipei Taiwan), April 2000.

[47] L. Xiao, X. Zhang, and Z. Xu, "On reliable and scalable peer-to-peer web document sharing," in *Proceedings of 2002 International Parallel and Distributed Processing Symposium (IPDPS'02)*, 2002.

[48] G. J.D. and A. Smith, "Evaluation of cache consistency algorithm performance," *Proceedings of the Fourth International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 236–248, Feb 1996.

[49] S. J., S. P., and V. R., "Proxy cache algorithms: design, implementation, and performance," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 11, pp. 549–562, July-Aug. 1999.

[50] A. Dingle and T. Partl, "Web cache coherence," in *5th int'l. world wide web conf.*, 1996.

[51] C. Liu and P. Cao, "Maintaining strong cache consistency in the world-wide web," in *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems*, May 1997.

[52] V. Duvvuri, P. Shenoy, and R. Tewari, "Adaptive leases: A strong consistency mechanism for the world wide web," in *Proceedings of the IEEE Infocom'00*, (Tel Aviv, Israel), March 2000.

[53] H. Yu, L. Breslau, and S. Shenker, "A scalable web cache consistency architecture," in *SIGCOMM*, pp. 163–174, 1999.

[54] K. B. and W. C.E., "Proxy cache coherency and replacement-towards a more complete picture," in *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*, pp. 332–339, 1999.

[55] `http://info.internet.isi.edu:80/in-notes/rfc/files/rfc2616.txt`.

[56] B. C. Housel and D. B. Lindquist, "Webexpress: A system for optimizing web browsing in a wireless environment," in *Proc. 2nd Annual Intl. Conf. on Mobile Computing and Networking*, (Rye New York), pp. 108–116, November 1996.

[57] G. Banga, F. Douglis, and M. Rabinovich, "Optimistic deltas for www latency reduction," in *Proc. 1997 USENIX Technical Conference*, (Anaheim CA), pp. 289–303, January 1997.

[58] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential benefits of delta-encoding and data compression for http," in *Proceedings of ACM SIGCOMM'97 Conference*, pp. 181–194, Sept. 1997.

[59] J. J. Hunt, K.-P. Vo, and W. F. Tichy, "An empirical study of delta algorithms," in *Proceedings of the 6th Workshop on Software Configuration Management*, March 1996.

[60] M. Busari and C. L. Williamson, "On the sensitivity of web proxy cache performance to workload characteristics," in *INFOCOM*, pp. 1225–1234, 2001.

[61] `http://www.w3.org/Protocols/rfc2616/rfc2616-sec7.html\#sec7.1`.

[62] T. Fagni and F. Silvestri, "Hybrid caching of search engine results," in *Proceedings of The Twelfth International World Wide Web Conference*, (Budapest, HUNGARY), May 2003.

[63] P. Cao, J. Zhang, and K. Beach, "Active cache: Caching dynamic contents on the web," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, (The Lake District, England), September 1998.

[64] `http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html\#sec13.4`.

[65] T. I., R. A., and S. V., "Static caching in web servers," in *Proceedings. Sixth International Conference on Computer Communications and Networks 1997*, pp. 410–417, 1997.

[66] A. Leff, J. Wolf, and S. Yu, "Efficient lru-based buffering ina lan remote caching architecture," *Transactions on Parallel and Distributed Systems*, pp. 191–206, February 1996.

[67] Y. K. N. K.W., "An optimal cache replacement algorithm for internet systems," in *Proceedings of 22nd Annual Conference on Local Computer Networks*, pp. 189–194, Nov 1997.

[68] S. D.N., K. G., and W. W.H., "Effective caching of web objects using zipf's law," in *Proceeding of 2000 IEEE International Conference on Multimedia and Expo*, vol. 2, pp. 727–730, 2000.

[69] D. Lee, J. Choi, H. Choe, S. H. Noh, S. L. Min, and Y. Cho, "Implementation and performance evaluation of the lrfu replacement policy," in *Proceedings of the 23rd Euromicro Conference New Frontiers of Information Technology*, pp. 106–111, Sept 1997.

[70] M. C.D. and A. V.A.F., "Using performance maps to understand the behavior of web caching policies," in *Proceedings. The Second IEEE Workshop on Internet Applications, 2001*, pp. 50–56, July 2001.

[71] R. Darnell, J. Pozadzides, and W. Steel, *HTML 4 unleashed.* Sams, 1997.

[72] `http://www.w3.org/XML/`.

[73] S. Abiteboul, D. Quass, J. McHugh, J. Widom, , and J.Wiener, "The lorel query language for semistructured data," *Journal on Digital Libraries and 1(1) and 1996*, 1996.

[74] S.-J. Lim and Y.-K. Ng., "Extracting structures of html documents," in *13th International Conference on Information Networking (ICOIN '98)*, (Tokyo and JAPAN), Jan. 1998.

[75] D. Konopnicki and O. Shmueli, "W3qs: A query system for the world-wide web," in *Proceedings of the 21st VLDB*, pp. 54–65, Sept. 1995.

[76] `http://www.ecst.csuchico.edu/~jacobsd/bib/formats/bibtex.html`.

[77] C. M., D. H., M. S.S., and M. W., "A new study on using html structures to improve retrieval," in *Proceedings. 11th IEEE International Conference on Tools with Artificial Intelligence, 1999*, pp. 406–409, Nov 1999.

[78] S.-J. Lim and Y.-K. Ng, “Extracting structures of html documents,” in *Proceedings. 13th International Conference on Information Networking (ICOIN ’98)*, (Tokyo, JAPAN), January 1998.

[79] Selkow and S.M., “The tree-to-tree editing problem,” *Inform. Processing Letters 6*, pp. 184–186, December 1977.

[80] W. J.T.L., S. B.A., S. D., Z. K., and C. K.M., “An algorithm for finding the largest approximately common substructures of two trees,” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 20, pp. 889–895, Aug. 1998.

[81] H. X., W. J., B. M., and N. P., “A tool for classifying office documents,” in *Proceedings. Fifth International Conference on Tools with Artificial Intelligence*, pp. 427–434, Nov 1993.

[82] R.A.Wanger and M. Fisher, “The string-to-string correction problem,” *J.Assoc. Comput.*, March 1974.

[83] K.-C. Tai, “The tree-to-tree correction problem,” *JACM 26 and 1997*, pp. 422–433, 1997.

[84] D. P. Williamson. `http://www.almaden.ibm.com/cs/people/dpw\#Notes`.

[85] G. Ausiello, P. Crescenzi, G. Gambosi, A. M.-S. V. Kann, and M. Protasi, *Complexity and Approximation, Combinatorial optimization problems and their approximability properties.* Springer Verlag, 1999.

[86] D. S. Johnson, “Approximation algorithms for combinatorial problems,” in *Proceedings of the fifth annual ACM symposium on Theory of computing*, (Austin, Texas, United States), pp. 38–49, may 1973.

[87] D. Hochbaum, "Approximation algorithms for set covering and vertex cover problems," in *SIAM Journal of Computing*, vol. 11, pp. 555–556, 1982.

[88] `http://www-fp.mcs.anl.gov/otc/Guide/OptWeb/discrete/integerprog/section2\%_1\_1.html`.

[89] B. Korte and J. Vygen, *Combinatorial optimization : theory and algorithms.* New York Springer, 2000.

[90] V. T. Paschos, *survey of approximately optimal solutions to some covering and packing problems ACM Computing Surveys (CSUR) archive*, vol. 29, Issue 2. New York, NY, USA: ACM Press, June 1997.

[91] R., EVEN, and S., "A lineartime approximation algorithm for the weighted vertex cover problem," *J. Alg.*, vol. 2, pp. 198–203, 1981.

[92] *chvátal* V., "A greedy-heuristic for the set covering problem," in *Math. Oper. Res. 4*, pp. 233–235, 1979.

[93] H. Ellis., S. Sahni, and S. Rajasekaran, *Computer algorithms/C++.* New York Computer Science Press, 1997.

[94] R. G. Parker and R. L. Rardin, *Discrete Optimization.* Academic Press, 1988.

[95] P. Borwein and T. Erdelyi, *Polynomials and Polynomial Inequalities.* Springer Verlag, 1994.

[96] D. Bini and V. Y. Pan, *Polynomial and Matrix Computations: Fundamental Algorithms.* Springer Verlag, 2002.

[97] `ftp://ftp.ircache.net/`.

Appendix A

Author's publications

1. "Incremental Update and Delivery of HTML Files". The 3rd International Conference on Information, Communications and Signal processing (ICICS2001)

2. "Distributed Web caching with incremental update", Communication Systems, 2002. ICCS 2002. The 8th International Conference on ,Volume: 2 , 25-28 Nov. 2002 Pages:1147 - 1151 vol.2

3. "Patch on web objects", Communication Technology Proceedings, 2003. ICCT 2003. International Conference on ,Volume: 1 , April 9 - 11, 2003 Pages: 221 - 224

4. "Incremental Update Of Web Objects Using Instruction Patches", submitted to *Computer Communications*, ISSN 0140-3664.

5. "Peer distributed web caching with incremental update scheme", submitted to *IEE Proceeding Communications*, ISSN 1350-2425.

6. "Web Patch Generation for Incremental Web Caching", submitted to *IEE Proceeding Communications*, ISSN 1350-2425.