韓國 컴퓨터情報學會 論文誌
第10卷 第6號, 2005. 12.

2005-10-6-1-16

데이터 검색의 적중률 향상을 위한 이중 캐시의 푸시 에이전트 모델 설계

김 광 종*, 고 현**, 김 영 자***, 이 연 식****

Design of Push Agent Model Using Dual Cache for Increasing Hit-Ratio of Data Search

Kwang-jong Kim*, Hyun Ko **, Young-ja Kim***, Yon-sik Lee ****

요 약

기존 단일 캐시 구조는 각기 사용되는 교체 전략에 따라 적중률의 차이를 보임으로써 보다 향상된 적중률을 제공하며 서버 및 네트워크 트래픽을 감소시키는 새로운 캐시 구조가 필요하다. 그러므로 본 논문에서는 지속적인 정보 제공 및 동일한 정보의 중복 요청으로 인한 서버 과부하 및 네트워크 트래픽을 감소시키며 적중률을 향상시키는 이중 캐시를 이용한 푸시 에이전트 모델을 설계한다. 제안된 푸시 에이전트 모델에서는 서버 및 네트워크 부하를 감소시키기 위해 두 개의 캐시 저장소를 이용하여 단계적인 캐시 교체를 수행하는 이중 캐시 구조를 제안하며, 또한 캐시 내 데이터들의 효용성을 증가시키기 위해 Log(Size)+LRU, LFU, PLC의 교체 정책을 기반으로 데이터 갱신과 삭제를 수행하는 새로운 캐시 교체 기법 및 알고리즘을 제시한다. 그리고 실험을 통해 이중 캐시 푸시 에이전트 모델에 대한 성능을 평가한다.

Abstract

Existing single cache structure has shown difference of hit-ratio according to individually replacement strategy. However, It needs new improved cache structure for reducing network traffic and providing advanced hit-ratio. Therefore, this paper design push agent model using dual cache for increasing hit-ratio by reducing server overload and network traffic by repetition request of persistent and identical information. In this model proposes dual cache structure to do achievement replace gradual cache using by two caches storage space for reducing server overload and network traffic. Also, we show new cache replace techniques and algorithms which executes data update and delete based on replace strategy of Log(Size)+LRU, LFU and PLC for effectiveness of data search in cache. And through an experiment, it evaluates performance of dual cache push agent model.

▸ Keyword : 이중 캐시(Dual Cache), 서버 푸시 에이전트(SPA), 캐시 관리 에이전트(CMA), 네이밍 에이전트(NA)

• 제1저자 : 김광종
• 접수일 : 2005.10.26, 심사완료일 : 2005.12.15
* 군산대학교 ※ 이 논문은 한국과학재단 특정기초연구 (R01-2004-000-10946-0)지원으로 수행되었음

Ⅰ. 서 론

푸시 기술은 분산 컴퓨팅 환경에서 사용자가 원하는 작업 목표만을 명시하면 해당하는 정보들을 사용자가 원하는 시간에 지속적으로 제공하는 서비스 기술이다[1,2,3,4]. 사용자들은 이러한 새로운 인터넷 정보 서비스 방식인 푸시 기술을 통해 웹상에 존재하는 수많은 정보들을 지속적으로 제공받을 수 있으며, 사용자가 원하는 정보들을 찾기 위해 소요되는 시간과 노력을 감소시킬 수 있다. 그러나 기존의 푸시 소프트웨어들은 제각기 그 구조와 구현이 너무도 다양하고, 개발 시 사용되고 있는 언어가 매우 상이하여 서로 간의 상호 호환성을 제공하지 못하고 있다. 또한, 푸시 방식은 사용자의 정보 제공 요구가 발생한 시점부터 정보 서비스의 중단 요청 때까지 지속적으로 정보를 제공함으로써 네트워크 부하를 가중시키게 된다. 그러므로 동시에 많은 정보 서비스 요청이 발생하였을 경우 이를 처리하는 서버 시스템에서는 과부하가 발생하여[2,3,4,5], 다수의 사용자로부터 동일한 정보의 중복된 요청에 대한 정보 서비스 제공이 어려워진다.

따라서 기존의 푸시 소프트웨어들이 가지는 상호 호환성 결여 문제와 푸시 방식을 통한 정보 서비스 제공 시 발생하는 서버 과부하 및 네트워크 부하를 감소시키기 위한 새로운 푸시 시스템의 개발이 요구되며, 정보 서비스의 질을 향상시키기 위해 사용자에 대한 정보 제공 시간의 단축과 동일한 정보에 대한 사용자들의 중복된 요청을 좀더 효율적으로 처리하기 위한 방안이 필요하다.

본 논문에서는 기존의 푸시 소프트웨어들이 가지는 상호 호환성 결여 문제를 해결하기 위해 이기종의 시스템과 이종의 네트워크 환경에서 상호운용성을 제공하는 CORBA 기반의 이중 캐시 푸시 에이전트 모델을 설계한다. 제안한 모델은 서버의 정보를 능동적으로 제공하기 위해 필요한 클라이언트 및 서버 푸시 에이전트(Client Push Agent : CPA, Server Push Agent : SPA)와 서버 및 네트워크 부하를 감소시키기 위해 필요한 캐시 관리 에이전트(Cache Management Agent : CMA), 에이전트 객체에 대한 위치 투명성을 보장하는 네이밍 에이전트(Naming Agent)를 포함한다. 각 에이전트들의 객체 참조자(Object Reference)를 관리하는 네이밍 에이전트는 에이전트 간의 통신 시 원격에 위치한 에이전트의 객체 참조자를 제공하여 서로 간의 접근을 허용함으로써 분산 환경에서의 에이전트 객체들에 대한 위치 투명성을 보장한다. 또한, 각 에이전트들에게 독립적인 작업 수행 환경을 제공하는 에이전트 플랫폼을 통해 상호운용성을 지원함으로써 기존 푸시 소프트웨어들 간의 상호 호환성 문제를 해결한다.

한편, 제안 모델에서는 사용자에게 능동적인 정보 서비스를 제공하면서 발생하는 푸시 서버의 과부하 및 서버와 연결된 네트워크 선로의 부하를 감소시키기 위해 이중 캐시 구조를 가진 캐시 서버를 설계한다. 그리고 푸시 서버로부터 정보 전송을 수용할 때마다 발생하게 되는 캐시 데이터의 재정립을 위해 캐시 저장소 내의 데이터 교체 기법 및 알고리즘을 제시한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련연구로서 캐시 형태 및 캐시 교체 기법에 대하여 설명한다. 3장에서는 기존 캐시 교체 기법들의 문제점을 해결하기 위해 이중 캐시 구조를 설계하고 이를 위한 이중 캐시 교체 기법 및 알고리즘과 제안된 이중 캐시 구조와 교체 전략을 통해 서버 과부하 및 네트워크 부하를 해결하는 이중 캐시의 푸시 에이전트 모델의 구조를 제시하고, 4장에서는 푸시 에이전트 모델을 설계하고, 5장에서는 기존의 단일 캐시 기법을 적용한 푸시 에이전트 모델과 이중 캐싱 기법을 적용한 푸시 에이전트 모델에서의 적중률을 통해 제안된 모델의 성능을 평가한다. 마지막 6장에서는 결론 및 향후 연구 과제를 제시한다.

Ⅱ. 관련연구

캐시란 데이터들을 임시로 저장해두는 저장소로 보통 사용자에게 자주 서비스되는 데이터들을 저장해둠으로써 사용자 요청에 대해 하드웨어나 소프트웨어의 성능 향상을 유도한다. 이러한 캐시는 설치할 위치와 구성 방법에 따라 클라이언트 캐시, 서버 캐시, 프락시 캐시, 그리고 계층적 캐시 네 가지로 구분할 수 있으며[6,7,8,9], 공통적인 관심사는 한정된 저장 영역의 효율적인 운용이다. 일반적으로 사용빈도가 높은 데이터를 캐시의 저장 영역에 저장해야 캐시의 성능을 높일 수 있다. 〈표 1〉은 캐시 공간이 필요할 경우 캐시 내의 파일들을 갱신하거나 제거하기 위한 주요 캐시 교체 기법들이다.

표 1. 주요 캐시 교체 기법(Cache Replacement Strategy)

※ [] 참고문헌	캐시 교체 기법 특성
FIFO (First In First Out)	캐시에 들어온 시간 순으로 교체하는 정책 파일의 중요도나 다른 기중치를 설정하지 않고 오래된 순서대로 교체 사용 빈도가 높은 파일도 시간에 따라 교체하는 문제 발생
LRU (Last Recently Used)[10,11]	최근 참조 시간이 가장 오래된 파일을 삭제하는 교체정책 최근 사용 파일이 가까운 시간 내 재사용될 가능성이 높다는 시간적 지역성에 기반 서버 캐시나 다른 캐시 시스템에서는 성능이 우수한 반면 프록시 서버에서는 다른 교체 정책에 비해 성능 저하
LFU (Last Frequency Used)[13,15]	참조 빈도수(Hit Count)가 가장 낮은 파일을 삭제하는 교체정책 많이 사용된 파일은 계속해서 사용될 가능성 높다는 점에 기반 초기에는 많이 사용되지만 나중에는 사용되지 않는 파일들을 제대로 교체하지 못함 프록시 캐시에서는 참조시간에 기반한 LRU보다 성능 우수
SIZE[14]	크기가 가장 큰 파일을 교체하는 정책으로 프록시 캐시에서 성능 우수 크기가 큰 하나의 파일이 작은 여러 개의 파일을 교체하는 경우 발생, 크기가 가장 큰 파일을 먼저 삭제함으로써 문제 해결
Log(size) + LRU[16]	LRU의 변형으로 가장 큰 같은 log(size) 크기를 가진 데이터 중에서 가장 오래 전에 사용되었던 데이터를 제거하는 정책 LRU 적용 시 여러 개의 작은 파일 교체 방지
LRU-THRESHOLD [12,15]	LRU의 또다른 변형으로 크기가 큰 파일들을 캐싱 대상에서 제거, 캐시 내에 작은 크기의 데이터들만을 유지하는 교체 정책 캐시 공간이 여유가 있더라도 크기가 임계값(threshold) 이상인 파일은 캐시의 대상에서 제외시키고, 교체 시 LRU 적용
LAT (Latency)[13]	지연시간을 줄이기 위한 정책으로 전송시간이 많이 걸리는 파일에 높은 우선순위를 주어 전송시간이 적은 파일을 먼저 교체하는 방식 서버로부터 파일을 가져오는 시간은 파일의 크기, 파일의 위치, 망 상태에 따라 다르게 나타날 수 있으므로 정확한 지연시간 측정이 어려움
PLC (Popularity based Lazy Caching)[15,16]	일정기간 서비스된 파일들의 선호도를 조사하여 선호도가 큰 파일들을 일괄적으로 캐싱하는 교체정책으로 캐시의 교체주기가 짧은 경우 LRU와 동일, 교체주기가 긴 경우 캐싱을 하지 않는 효과 초래 캐싱이 발생할 때 마다 데이터 교체전략을 수행함으로써 캐시 메모리에서 발생하는 과부하 감소

이와 같이 현재까지 연구된 교체 기법들은 캐싱 데이터의 다양한 특성을 골고루 반영하지 못하고 특정한 한 가지 성격만을 편중되게 반영하여 캐시 내 데이터들을 최적화시키지 못한다[10,11,12,16]. 이로 인해 클라이언트의 정보 요청에 대한 서버의 정보 전송률이 증가하게 되어 서버 과부하를 발생시킨다. 또한 최적의 캐시 상태를 유지하기 위해서 파일의 크기, 참조 빈도수, 참조시간, 전송시간 등의 모든 형태를 캐시 교체 전략에 반영하려 할 경우에는 캐시 교체 시 시간 복잡도의 증가와 캐시 서버의 과부하를 발생시킬 위험이 있다[13,14,15]. 따라서 새로운 캐시 교체 전략은 데이터의 특성 및 필요한 교체 기준 요소를 분석하여 적절한 캐시 교체 기법을 적용시킴으로써 캐시 데이터의 효용성 및 데이터 적중률을 향상시킬 수 있어야 한다.

III. 이중 캐시를 적용한 캐싱 기법

3.1 이중 캐시 구조

일반적인 캐시 시스템들은 캐시 저장소에 대한 캐시 교체 전략으로 FIFO, LRU, LFU, SIZE, LAT 등의 기법들을 사용한다. 그러나 이러한 각 캐시 기법들은 각기 나름대로의 특성을 가짐으로써 캐싱된 데이터의 다양한 특성을 제대로 반영하지 못하여 캐시 내 데이터들을 최적화시키지 못한다. 그렇다고 최적의 캐시 상태를 유지하기 위해 파일 크기, 참조 빈도수, 참조 시간, 전송 시간 등의 모든

교체 기준을 캐시 교체 전략에 반영하려고 하면, 데이터의 캐싱 및 교체 시간이 증가할 수 있고 캐시 서버의 과부하를 발생시킬 수도 있다.

이러한 문제를 해결하기 위해 이중 캐시 구조는 캐시 저장소를 각각 메모리 영역에 존재하는 First Cache와 디스크 영역에 존재하는 Second Cache로 구성하여 단계적인 데이터 캐싱 및 교체 방법을 통해 교체 시의 시간 복잡도를 감소시키기 위한 구조이다. 즉, 여러 교체 기준을 사용하여 캐시 교체를 수행하되 각 저장소마다 다른 요소를 기준으로 수행함으로써 단계적인 데이터 여과를 통해 First Cache에 적중률이 높은 데이터를 유지하기 위한 형태이다. (그림 1)은 여러 캐시 교체 기준을 사용할 수 있도록 두 개의 캐시 저장소로 구성된 이중 캐시 구조이다.

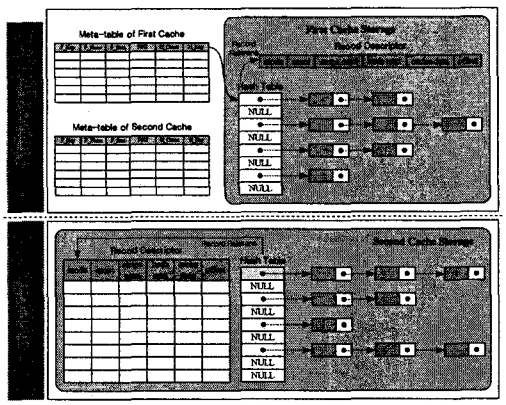


그림 1. 이중 캐시 구조(Dual Cache Architecture)

이중 캐시 구조는 (그림 1)과 같이 두 캐시 저장소의 관리 및 참조를 위해 First Cache 및 Second Cache 저장 데이터의 메타 정보를 기록하기 위한 각각의 메타 테이블과 각 데이터의 참조 정보를 기록하기 위한 해시 테이블을 가진다. 이 때, 해시 테이블을 통해 데이터를 저장하는 이유는 데이터 검색 시 비교 연산 없이 검색하고자 하는 레코드를 한번에 직접 검색할 수 있고, 데이터 검색 및 삽입, 삭제 연산 시의 계산량이 데이터 개수에 좌우되지 않아 매우 좋은 성능을 보이기 때문이다. 한편, 메타 테이블은 캐시 교체 기준 요소인 파일 크기(File Size), 파일에 대한 참조 빈도수(Hit Count), 최근 참조 시간(Recently Reference Time) 및 각 파일을 식별하기 위한 파일 식별키(File Key)와 파일명(File Name) 그리고 해쉬 테이블에서 사용되는 식별키(Hash Key)로 구성된다. 이러한 메타 테이블을 생성하는 목적은 캐시에 저장된 데이터의 메타 정보를 관리하여 캐시 교체 시 기준이 되는 요소들로 사용하기 위함이고, 해시 테이블이 정렬에 관련된 연산에는 성능이 좋지 않은 단점을 가짐으로써 캐시 교체 시 메타 정보에 대한 정렬 연산을 수행하기 위해서이다. (그림 2)는 해시 레코드 기술자 및 캐시 교체 기준 요소들로 구성된 메타 테이블의 구조이다.

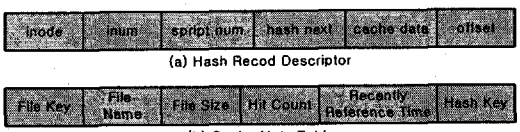


그림 2. 해시 레코드 기술자 및 캐시 메타 테이블 구조(Hash Record Descriptor & Cache Meta Table)

이중 캐시 구조 형태의 캐시 서버는 사용자 정보 요청이 발생하면 First Cache의 메타 테이블(Meta Table)을 검색하여 해당 파일이 존재하는지 확인한 후, 존재하지 않을 경우 Second Cache의 메타 테이블을 검색한다. First 혹은 Second Cache에 사용자 요구 파일이 존재할 경우, 해당 Cache의 Hash Table을 이용해 파일에 접근하여 이를 참조한다. 만약, 두 Cache에 요구 파일이 존재하지 않으면 캐시 서버는 사용자 요구를 푸시 서버에 전달하여 푸시 서버로 하여금 요청 파일을 제공하도록 한다. 이중 캐시 구조는 이와 같은 과정을 통해 사용자 요구에 대한 보다 효율적인 정보 서비스를 제공한다.

3.2 이중 캐시 교체 전략

3.2.1 Log(size)+LRU를 적용한 1CS(first cache) 교체 기법

이중 캐시 구조 중 사용자 정보 요청을 가장 먼저 수용하게 되는 1CS(first cache)는 최근 참조 시간과 데이터 크기를 통해 캐시 교체를 수행한다. 1CS는 2CS(second cache)로부터 참조 빈도수가 높고 크기가 작은 데이터를 전송받게 되는데, 2CS로부터의 데이터 전송 시 캐시 저장소에 여유 공간이 없을 경우 Log(size)+LRU 기반의 교체 전략을 수행한다. Log(size)+LRU를 이용한 1CS에서의 캐시 교체 과정은 다음과 같다.

〈표 2〉의 과정들 중 (2)~(3) 과정은 1CS에서 교체될 파일을 탐색하는 과정으로, 파일 참조 시간이 가장 오래되고 파일의 크기가 가장 큰 파일 순으로 정렬하여 최상위에 위치한 레코드의 파일을 제거 대상으로 결정한다. 이 때, 1CS에서 참조 빈도수가 아닌 참조 시간을 교체 기준으로

삼는 이유는 이미 2CS로부터 일정 기준 이상의 참조 빈도수가 넘는 데이터를 전송받아 저장하고 있기 때문이다. 즉, 1CS 내 데이터들의 참조 빈도수는 정보 요청이 발생할 때마다 1씩 증가하므로 대부분 참조 빈도수가 높기 때문에 이를 사용한 교체 대상의 분별은 적절하지 못하다. 따라서 과거에 요청되었던 데이터가 또다시 이용되어질 확률이 낮음으로 데이터의 참조 시간에 교체 기준을 두게 된다. (그림 3)은 1CS에서 Log(size)+LRU를 이용하여 캐시를 교체하는 과정에 대한 예이다.

따라서 Log(size)+LRU 교체 기법을 적용한 1CS는 참조 시간이 가장 낮으며 크기가 가장 큰 파일들을 제거하여 적은 데이터를 여러 번 삭제하는 데서 오는 시간 지연을 감소시킨다. 또한, 1CS는 2CS로부터 파일 참조 빈도수가 높고 가능하면 파일 크기가 작은 데이터 파일을 받아 저장함으로써 사용자가 자주 요청하는 파일에 대한 이용률을 높인다.

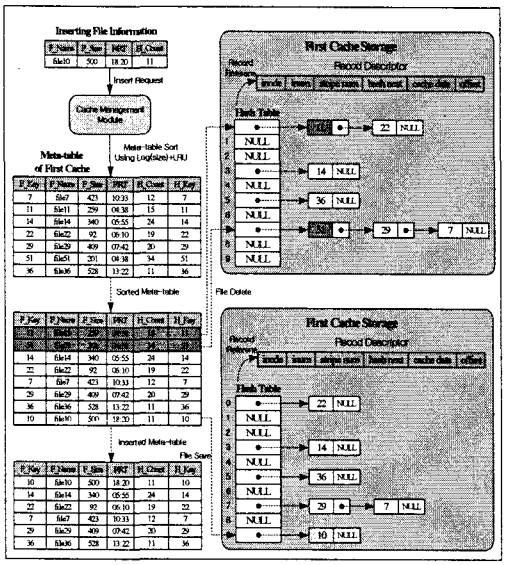


그림 3. Log(size)+LRU를 이용한 캐시 교체 과정 예(Cache Replacement Stage Using Log(size)+LRU)

3.2.2 1CS의 교체 전략 알고리즘

1CS에 저장된 N개의 파일들에 대한 메타 정보는 CachedFileInfo[n] 객체 배열에 저장하고, 2CS에서 전달된 교체할 파일들에 대한 메타 정보는 CachingFile에 저장한다. 먼저, 2CS로부터 데이터 전송이 발생하였을 경우 1CS에 여유 공간이 없다면 파일의 최근 참조 시간(RRT : Recently Reference Time)이 가장 작은 순서대로 정렬한다. 이때, 동일한 참조 시간을 갖는 파일이 검색되면 이들은 파일의 크기를 기준으로 하여 내림차순 정렬한다(모든 파일들이 최근 참조 시간이 같을 확률은 거의 0에 가깝다). 즉, CachedFileInfo[n].RRT가 작고 CachedFileInfo[n].size 값이 가장 큰 순서로 정렬한다. 또한, 정렬이 끝나면 첫 번째 교체 대상이 되는 파일, 즉 i=0 일 때의 CachedFileInfo[i].size를 ReplaceFileSize에 저장한다. 저장된 ReplaceFileSize는 CachingFile의 크기와 비교하여 ReplaceFileSize가 충분히 클 경우, CachedFileInfo[i]의 파일을 제거한 후 2CS로 전송하여 저장하도록 하고, 1CS에는 CachingFile을 저장한다. 만약 ReplaceFileSize가 CachingFile의 크기보다 작을 경우, ReplaceFileSize에 CachedFileInfo[i+1]까지의 크기를 재저장하는 과정을 반복적으로 수행하여 ReplaceFileSize 값이 CachingFile의 크기를 충분히 만족시키면 해당된 파일들을 차례로 제거한다. <표 3>은 Log(size)+LRU를 이용한 1CS에서의 교체 전략 알고리즘이다.

2. Log(size)+LRU를 이용한 1CS 캐시 교체 과정(1CS Cache Replacement Stage Using Log(size)+LRU)

(1) 2CS로부터 1CS에 캐싱 해야 할 파일이 추출되면 CMA는 해당 파일을 저장가능한지 1CS를 검사한다. 즉, 캐싱해야할 파일의 크기에 맞는 빈 공간이 있는지를 검사한다.

(2) 만약 캐싱 해야 할 파일의 크기에 맞는 빈 공간이 없으면, 먼저 1CS 메타 테이블에서 각 파일의 최근 참조 시간을 이용하여 참조 시간에 대한 오름차순 정렬을 수행한다.

(3) 1CS 메타 테이블을 오름차순으로 정렬하는 중 파일의 참조 시간이 같을 경우에는 파일의 크기를 참조하여 가장 크기가 큰 파일이 상위에 오도록 재정렬한다. (파일 참조 시간이 같을 경우는 존재하지 않을 수 있음)

(4) 정렬된 메타 테이블로부터 최상위 레코드의 해시 식별키를 이용, 실제 파일이 저장된 위치를 찾은 후 이를 메모리 영역에서 제거하고, 메타 테이블의 레코드 정보도 삭제한다. 이 때, 캐싱할 파일의 크기와 제거될 파일의 크기를 비교하여 제거될 파일의 크기가 작으면 캐싱할 파일의 크기보다 클 때까지 반복적으로 바로 하위 레코드에 해당하는 파일들을 제거한다.

(5) 제거된 파일은 2CS에 전달하여 저장하도록 한다.

(6) 마지막으로, 새로 추가될 파일의 메타 정보를 추출하여 메타 테이블에 새로운 파일 레코드를 생성한 후 캐시 저장소의 해시 레코드 기술자를 이용하여 적절한 장소에 추가한다.

3.2.3 Log(size)+LFU와 PCL을 적용한 2CS 교체 기법

2CS 교체 전략의 가장 큰 특징은 지연성에 있다. 기존의 교체 기법들에서와 같이 매번 데이터를 서비스하는 순간에 캐시를 수정하는 것이 아니라 일정 시간 동안 2CS에 도착하는 요청들을 살펴본 후에 결정한다. 다만 향후에 2CS로부터 삭제될 파일과 1CS로 전송되어야 할 데이터에 대한 참조 빈도수 값만을 계산한다.

Log(size)+LFU와 PLC를 적용한 2CS 교체 전략은 사용자 요구가 빈번히 변화하지 않고 일정기간 동안 동일한 정보 요청이 발생하는 환경에서 최고의 캐시 적중률을 보이는 교체 기법이다. 2CS에서는 푸시 서버 및 1CS로부터의 데이터 캐싱 발생 시 2CS에 여유 공간이 없을 경우 Log(size)+LFU와 PLC에 따른 캐시 교체 전략을 수행한다. 다음은 Log(size)+LFU와 PLC를 적용한 2CS에서의 교체 전략 과정이다.

위의 과정 중 (2)~(3) 과정은 2CS에서 저장된 파일들을 교체하기 위해 Log(size)+LFU를 기준으로 메타 테이블을 정렬하는 과정이고, (4)와 (5) 과정은 각각 파일의 참조 빈도수가 10 이상일 때와 10 미만일 때 파일을 제거하는 과정이다. 위의 (2) 과정에서 파일 참조수를 교체 기준으로 사용하는 이유는 어느 시점에서 참조 빈도수가 큰 데이터들의 모임이 그 다음 시점에서도 매우 높은 참조 빈도수를 가지기 때문이다. 일반적으로 사용자의 정보 요청이 시간의 흐름에 따라 꾸준히 변하기는 하지만 캐시 저장소를 즉시 조작해야할 정도로 급격히 변하지는 않기 때문에 특별한 경우를 제외하고는 데이터들이 캐싱되는 순간마다 데이터를 캐싱할 것인지 삭제할 것인지를 판단할 필요는 없다. 이처럼 2CS는 참조 빈도수를 통해 사용자 정보 요청의 흐름을 파악하고 참조 빈도수가 증가하는 데이터에 대해서는 사용자 정보 요청이 앞으로도 빈번하게 일어날 것이라는 가정하에 1CS로 데이터를 전송한다. (그림 4)는 2CS에서 Log(size)+LFU와 PLC를 이용한 캐시 교체 과정에 대한 예이다.

표 3. 1CS 교체 알고리즘(1CS Replacement Algorithm)

```
// CachedFileInfo[n] : 1CS에 저장된 N개의 파일들에 대한 메타 정보
// CachingFile : 2CS에서 전달된 교체할 파일

IF (1CS == FULL) THEN
  LOOP (i=0 에서부터 i=n-1 까지)
    LOOP (j=1 에서부터 j=n-i 까지)
      IF (CachedFileInfo[j-1].RRT > CachedFileInfo[j].RRT) THEN
        // 캐쉬된 두 파일에 대한 메타 정보 교환
        CachedFileInfo[j-1]와 CachedFileInfo[j] 교환;
      ELSE IF (CachedFileInfo[j-1].RRT == CachedFileInfo[j].RRT) THEN
        IF (CachedFileInfo[j-1].size < CachedFileInfo[j].size) THEN
          // 캐쉬된 두 파일에 대한 메타 정보 교환
          CachedFileInfo[j-1]와 CachedFileInfo[j] 교환;
        END IF
      END IF
    END LOOP
  END LOOP

  i = 0;
  ReplaceFileSize = CachedFileInfo[i].size;

  IF (ReplaceFileSize >= CachingFile.size) THEN
    CachedFileInfo[i]를 1CS에서 제거하고 2CS에 CachedFileInfo[i] 저장;
  ELSE
    LOOP (ReplaceFileSize >= CachingFile.size)
      ReplaceFileSize = ReplaceFileSize + CachedFileInfo[i+1].size;
      CachedFileInfo[i+1]까지의 메타 정보를 1CS에서 제거;
      2CS에 CachedFileInfo[i+1]까지의 저장;
    END LOOP
  END IF
  CachingFile을 1CS에 저장;
END IF
```

표 4. Log(size)+LFU와 PCL을 이용한 2CS 교체 과정(2CS Replacement Stage Using Log(size)+LFU, PCL)

(1) 1CS나 푸시 서버로부터 2CS에 캐싱해야 할 파일이 전달되면 CMA는 해당 파일을 저장 가능한지 2CS를 검사한다.
(2) 만약 캐싱할 파일의 크기에 맞는 빈 공간이 없으면, 먼저 2CS 메타 테이블에서 각 파일의 참조 빈도수를 이용하여 참조 빈도수에 대한 내림차순 정렬을 수행한다.
(3) 2CS 메타 테이블을 내림차순으로 정렬하는 중 파일의 참조 빈도수가 같을 경우에는 파일의 크기를 참조하여 가장 크기가 작은 파일이 상위에 오도록 재정렬한다.
(4) 정렬된 메타 테이블에서 최상위의 파일 참조 빈도수를 추출하여 빈도수가 10 이상인지를 검사한 후 최상위 레코드의 해시 식별키를 이용, 실제 파일이 저장된 디스크 영역으로부터 파일을 제거하고, 메타 테이블의 레코드 정보도 삭제한다. 이 때, 캐싱할 파일의 크기와 교체될 파일의 크기를 비교하여 교체 파일의 크기가 작으면 캐싱할 파일의 크기보다 클 때까지 반복적으로 바로 하위 레코드에 해당하는 파일들을 제거한다.
(5) 만약, 최상위의 파일 참조 빈도수가 10 이하일 경우, 캐싱할 파일의 크기와 메타 테이블에서 n번째 레코드에 해당하는 파일의 크기를 비교하여 교체 대상 파일의 크기가 클 경우에만 파일과 메타 테이블의 레코드 정보를 제거한다. 만약, 교체 대상 파일의 크기가 작으면 캐싱할 파일의 크기에 만족할 때까지 (n-1), (n-2), … 레코드의 파일들을 반복적으로 제거한다.
(6) 새로 추가될 파일의 메타 정보를 추출하여 메타 테이블에 새로운 파일 레코드를 생성한 후 캐시 저장소의 해시 레코드 기술자를 이용하여 적절한 장소에 추가한다.

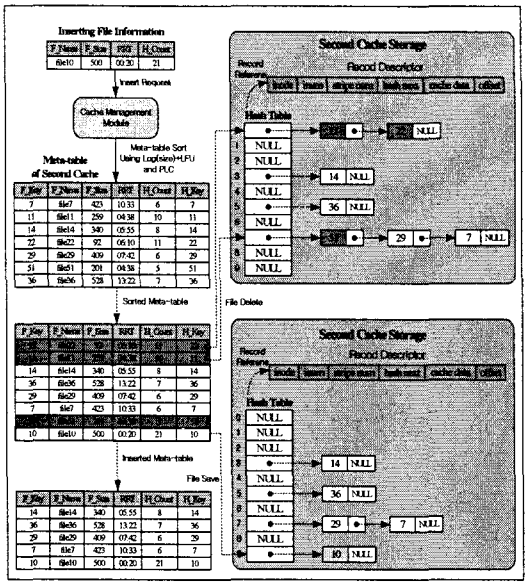


그림 4. Log(size)+LFU와 PLC를 이용한 캐시 교체 과정 예(2CS Replacement Stage Using Log(size)+LFU, PCL)

3.2.4 2CS의 교체 전략 알고리즘

2CS에 저장된 N개의 파일들에 대한 메타 정보는 CachedFileInfo[n] 객체 배열에 저장하고, 하나의 파일에 대한 메타 정보는 CachedFileInfo 객체에 저장한다. 먼저, 2CS에 저장된 데이터들을 파일 빈도수가 가장 높고 파일의 크기가 가장 작은 순으로 정렬한다. 다음 과정은 i=0 일 때, CachedFileInfo[i].hit가 10 이상인 경우에만 ReplaceFileSize에 CachedFileInfo[i].size 값을 저장하고, ReplaceFileSize와 CachingFileSize를 비교한다. 만약 ReplaceFileSize가 크거나 같으면 i 번째의 레코드와 디스크 영역에 저장된 실제 파일을 제거하고, ReplaceFileSize가 작으면 CachingFileSize보다 크거나 같을 때까지 i-1, i-2, … 번째의 레코드들과 실제 파일을 제거한다(단, i-1, i-2, … 번째의 레코드 정보 중 참조 빈도수가 10 이상인 경우). 여기서 제거된 파일들은 참조 빈도수가 높고 파일의 크기가 작기 때문에 1CS에 저장하기 위해 전달된다.

또한, CachedFileInfo[i].hit가 10 미만인 경우에는 ReplaceFileSize에CachedFileInfo[n-(i+1)].size 값을 저장하여 ReplaceFileSize와 CachingFileSize를 비교한 후 ReplaceFileSize가 클 경우에만 n-(i+1) 번째의 레코드와 디스크 영역에 저장된 실제 파일을 제거한다. 만약, ReplaceFileSize가 작을 경우에는 위의 방식과 마찬가지로 CachingFileSize보다 크거나 같을 때까지 i를 1씩 증가시켜 n-(i+1) 번째들의 레코드와 파일들을 제거한다. 이 때 제거되는 파일들은 참조 빈도수가 낮고 파일의 크기가 크기 때문에 1CS에 전달되지 않고 영구히 삭제된다. <표 5>는 Log(size)+LFU와 PLC를 이용하여 2CS에서 캐시 교체를 수행하는 알고리즘이다.

Ⅳ. 이중 캐시 푸시 에이전트 모델 설계

4.1 전체 모델 구조

제안한 모델은 분산 컴퓨팅 환경에서 사용자가 원하는 작업 목표만을 명시하면 푸시 에이전트가 작업을 수행하여 검색된 정보들을 사용자가 원하는 시간에 지속적으로 제공하는 시스템 모델이다. (그림 5)와 같이 클라이언트 브라우저(Client Browser) 및 클라이언트 푸시 에이전트(Client Push Agent)로 구성된 클라이언트와 각 에이전트 객체들에 대한 위치 투명성을 제공하는 네이밍 서버(Naming Server), 서버 푸시 에이전트를 포함하는 푸시 에이전트 서버(Push Agent Server) 그리고 캐시 저장소를 접근하고 관리하는 캐시 서버(Cache Server)로 구성된다. 푸시 에이전트 모델에서의 캐시 서버는 푸시 에이전트 서버와 클라이언트 사이에 위치하는 것으로 사용자 요청에 대해 캐시 저장소를 검색하여 해당 결과를 반환해줌으로써 푸시 에이전트 서버와 동일한 역할을 수행한다. 이는 푸시 에이전트 서버의 과부하를 해결하고 사용자에게 좀더 빠르고 안정적인 정보 서비스를 제공해준다.

4.2 모델의 통신 과정

모델에서의 통신 과정은 각 에이전트 시스템 관련 정보 등록 과정과 각 에이전트 간의 작업 처리 과정으로 구분된다. (그림 5)에서 (i)~(iii) 과정은 각 에이전트 시스템의 관련 정보들을 등록하는 과정으로 푸시 에이전트 모델을 구성하는 각 시스템들의 초기 기동 시 에이전트, 플레이스, 에이전트 시스템에 대한 객체들을 네이밍 서버에 등록하는 과정이다. 이는 등록된 객체 정보를 통해 에이전트들 간의 통신이 가능하도록 위치 투명성을 제공한다. 또한, 푸시 에이전트 모델에서의 각 에이전트 간 작업 처리 과정은 다시 캐시 서버에 해당 정보가 존재하는 경우와 그렇지 않은 경우로 나뉜다.

클라이언트와 캐시 서버 간의 통신은 캐시 저장소에 사용자가 요청한 정보가 존재하는 경우에만 이루어진다. (그림 5)에서 [1]~[4-3] 과정은 캐시 서버에 사용자 요구 정보가 존재하는 경우로 클라이언트와 캐시 서버 간의 작업 처리 과정은 <표 6>과 같다.

(그림 6)은 위의 과정을 통해 사용자가 요청한 정보를 캐시 서버로부터 검색하여 클라이언트 제공해주는 과정에 대한 시퀀스 다이어그램(Sequence Diagram)이다.

4.3 푸시 에이전트

제안된 푸시 에이전트는 클라이언트 푸시 에이전트와 서버 푸시 에이전트로 구성되며, 클라이언트 푸시 에이전트는 사용자 정보 요청을 수용하고, 서버 푸시 에이전트는 해당 정보 검색 결과를 사용자에게 푸시해주는 역할을 수행한다.

먼저, 서버 푸시 에이전트는 캐시 저장소에 사용자의 요구에 대한 정보가 존재하지 않았을 경우 구동되는 에이전트로, 데이터베이스 연결 및 질의 처리 기능을 수행하는 DB 핸들러(DB Handler)와 이를 포함하는 SPA 실행 모듈(SPA Execution Module) 그리고 푸싱 모듈(Pushing Module)로 구성된다. SPA 실행 모듈은 캐시 관리 에이전트로부터 전송된 클라이언트 푸시 에이전트의 객체 참조자 및 사용자 프로파일 정보를 이용하여 데이터베이스에 질의를 수행하고, 푸싱 모듈은 검색된 결과를 캐시 관리 에이전트와 클라이언트 푸시 에이전트로 전송한다.

클라이언트 푸시 에이전트는 모니터링 모듈을 포함하는 CPA 실행 모듈(CPA Execution Module)과 사용자 프로파일 생성 모듈(User Profile Generate Module), 필터링 모듈(Filtering)로 구성된다. 먼저, CPA 실행 모듈은 전반적인 작업을 관리하는 모듈로서 사용자로부터 서비스 요청을 받고 프로파일 생성 모듈을 이용하여 해당 요구에 따른 사용자 프로파일을 생성한다. 또한, CMA나 SPA로부터 사용자 요구에 대한 검색 결과가 전송되면 모니터링 모듈(Monitoring)을 통해 이를 감지하여 CPA 스토리지(CPA Storage)에 검색 결과를 저장한 후 필터링 모듈을 호출한다. 필터링 모듈은 각각의 푸시 모듈로부터 전송되어진 데이터들의 중복성을 제거하여 사용자에게 질적인 정보를 서비스하도록 한다. (그림 7)은 이러한 모듈들로 구성된 푸시 에이전트 구조이다.

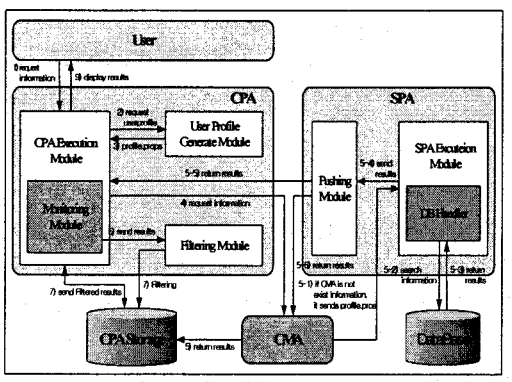


그림 7. 푸시 에이전트(Push Agent)

4.4 캐시 관리 에이전트

캐시 관리 에이전트는 클라이언트 푸시 에이전트와 서버 푸시 에이전트 사이에 위치하여 사용자 정보 요청이 발생하였을 경우, 이를 캐시 관리 에이전트가 수행함으로써 서버 접근을 제한하도록 한다. 캐시 관리 에이전트는 정보 검색 모듈(Information Search Module)과 캐시 관리 모듈(Cache Management Module), 푸싱 모듈(Pushing Module)로 구성되어 진다. 먼저, 캐시 관리 에이전트의 정보 검색 모듈은 클라이언트 푸시 에이전트로부터 정보 검색 요청이 발생하면 이를 분석하여 1CS를

표 5. 2CS 교체 알고리즘(2CS Replacement Algorithm)

```
// CachedFileInfo[n] : 2CS에 저장된 N개의 파일들에 대한 메타 정보
// CachingFile : 2CS에서 전달된 교체할 파일들에 대한 메타 정보

IF (2CS == FULL) THEN
  LOOP (i=0 에서부터 i=n-1 까지)
    LOOP (j=1 에서부터 j=n-i 까지)
      IF (CachedFileInfo[j-1].hit > CachedFileInfo[j].hit) THEN
        // 캐쉬된 두 파일에 대한 메타 정보 교환
        CachedFileInfo[j-1]와 CachedFileInfo[j] 교환;
      ELSE IF (CachedFileInfo[j-1].hit == CachedFileInfo[j].hit) THEN
        IF (CachedFileInfo[j-1].size < CachedFileInfo[j].size) THEN
          // 캐쉬된 두 파일에 대한 메타 정보 교환
          CachedFileInfo[j-1]와 CachedFileInfo[j] 교환;
        END IF
      END IF
    END LOOP
  END LOOP

  i = 0;
  IF (CachedFileInfo[i].hit >= 10) THEN
    ReplaceFileSize = CachedFileInfo[i].size;
    IF (ReplaceFileSize >= CachingFile.size) THEN
      CachedFileInfo[i]를 2CS에서 제거하고 1CS에 저장;
    ELSE
      LOOP (ReplaceFileSize >= CachingFile.size)
        IF (CachedFileInfo[i+1].hit >= 10) THEN
          ReplaceFileSize = ReplaceFileSize + CachedFileInfo[i+1].size;
          CachedFileInfo[i+1]를 2CS에서 제거하고 1CS에 저장;
        ELSE
          ReplaceFileSize = ReplaceFileSize + CachedFileInfo[n-(i+1)].size;
          CachedFileInfo[n-(i+1)]를 2CS에서 제거하고 1CS에 저장;
        END IF
      END LOOP
    END IF
    CachingFile을 2CS에 저장;
  ELSE
    ReplaceFileSize = CachedFileInfo[n].size;
    IF (ReplaceFileSize >= CachingFile.size) THEN
      CachedFileInfo[n]을 2CS에서 제거하고 1CS에 저장;
    ELSE
      LOOP (ReplaceFileSize >= CachingFile.size)
        ReplaceFileSize = ReplaceFileSize + CachedFileInfo[n-(i+1)].size;
        CachedFileInfo[n-(i+1)]를 2CS에서 제거하고 1CS에 저장;
      END LOOP
    END IF
    CachingFile을 2CS에 저장;
  END IF
END IF
```

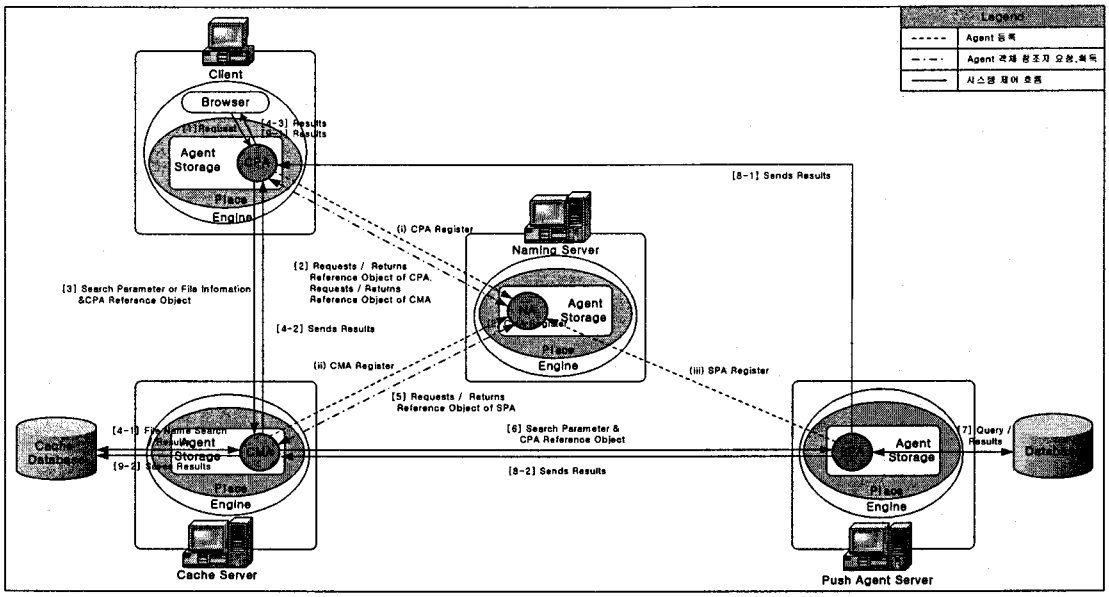


그림 5. 이중 캐시를 이용한 푸시 에이전트 모델 구조(Push Agent Model Using Dual Caches)

표 6. 클라이언트와 캐시 서버 간의 작업 처리 과정(Work Flow Stages between Client and Server)

(1) 클라이언트 브라우저를 통해 사용자가 요구하는 정보, 즉 파일 관련 정보를 입력받는다.
(2) 브라우저를 통해 입력된 파일 관련 정보가 CPA에 전달되면, CPA는 NA로부터 CMA의 객체 참조(Object Reference)와 자기 자신의 객체 참조자를 획득한다.
(3) 과정에서 획득한 CMA 객체 참조자를 이용하여 캐시 서버와 클라이언트 내의 CPA를 연결한 후, 캐시 서버에 파일 관련 정보와 CPA 객체 참조자를 전달한다.
(4) CMA는 CPA로부터 획득한 파일 관련 정보를 이용하여 캐시 저장소를 검색한다.
(5) CMA는 캐시 저장소로부터 추출된 파일을 CPA 객체 참조자를 이용하여 클라이언트의 CPA에게 푸싱한다.
(6) CPA는 캐시 서버로부터 전달받은 검색 결과를 사용자 브라우저에 디스플레이(Display) 한다.

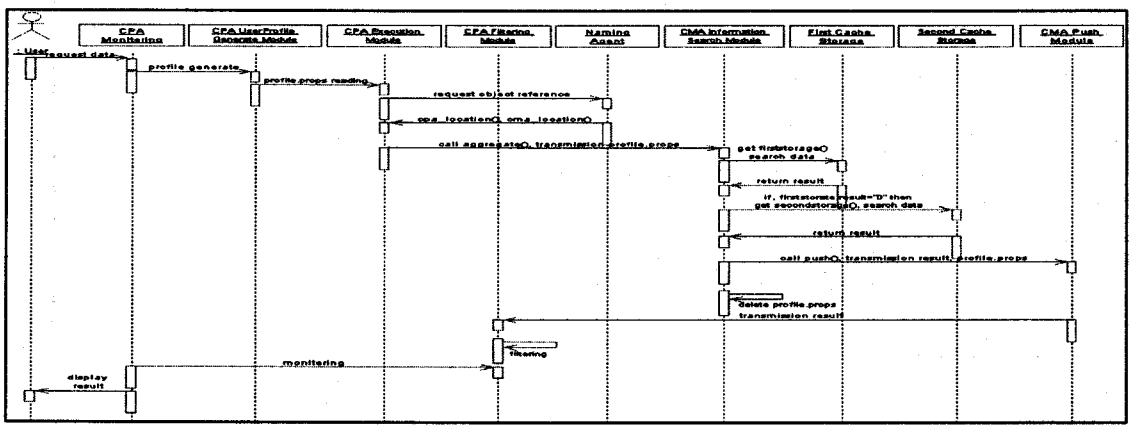


그림 6. 클라이언트와 서버의 통신(Communication between Client and Server)

질의한다. 만약, 1CS에 해당 결과 값이 존재할 경우에는 이를 푸싱 모듈로 전송하고, 해당 정보를 검색하지 못했을 경우에는 2CS를 질의한다. 또한, 2CS에서 해당 정보를 검색하였을 경우에는 이를 푸싱 모듈로 전달하고, 적절한 정보를 검색하지 못하였을 경우에는 서버 푸시 에이전트로 해당 정보 검색을 요청한다.

한편, 푸싱 모듈은 1CS 또는 2CS로부터 전달받은 데이터를 클라이언트 푸시 에이전트로 전송하며, 전송이 완전히 이루어지면 정보 검색 모듈은 클라이언트 푸시 에이전트의 정보를 삭제함으로써 정보 전송을 마친다. 캐시 관리 모듈은 3.2절에서 소개한 캐시 교체 정책을 통하여 1CS와 2CS 내의 데이터를 삭제 및 갱신함으로써 데이터 적중률을 향상시킨다. (그림 8)은 두 개의 캐시 저장소를 관리하고 검색된 정보를 클라이언트에 푸싱하는 캐시 관리 에이전트의 구조이다.

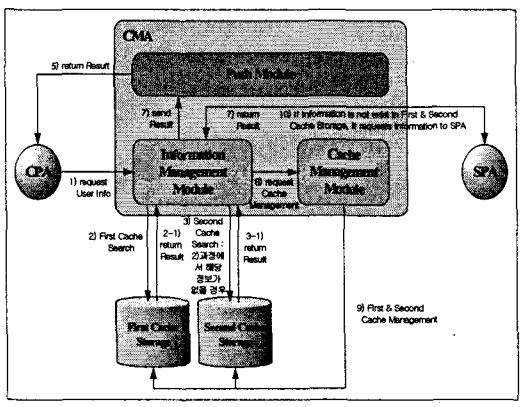


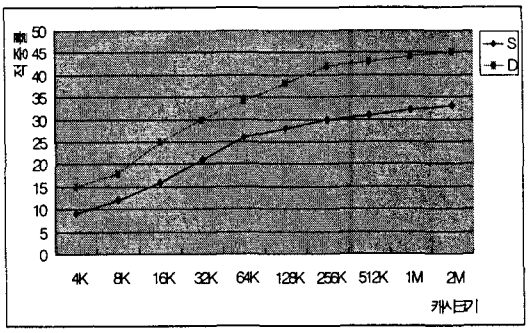
그림 8. 캐시 관리 에이전트(Cache Management Agent)

V. 실험 및 분석

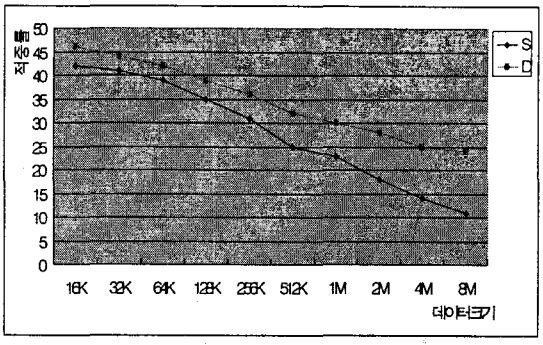
제안된 이중 캐시 푸시 에이전트 모델의 성능을 분석하기 위하여 본 논문에서는 (그림 5)와 같은 구조의 프로토타입 시스템을 구현하였다. 프로토타입 시스템은 네트워크에 연동된 캐시 서버를 대상으로 사용자가 정보를 요청하면 해당 캐시 서버내 First 캐시와 Second 캐시를 검색하여 검색된 결과를 사용자에게 푸시 에이전트가 전달하는 것이며, 구현 환경은 에이전트 시스템들 간의 상호운용성 지원을 위한 OMG의 MAF 명세와 CORBA 3.0 Spec.에 기반한 OrbixWeb 3.2를 사용하였다. 또한, 분산 환경에서의 성능 평가는 서버의 부하, 네트워크 트래픽, 사용자 요청에 대한 응답시간 등의 문제를 모두 고려해야 하지만 실험 환경 여건상 균등 네트워크 상이라 가정하고 사용자 요구에 대한 단일 캐시와 이중 캐시의 적중률만을 평가하였다.

실험은 데이터 크기와 캐시 크기에 따라 실험하였다. 즉, 데이터를 단일 캐시에 적재한 경우와 이중 캐시를 이용하여 분할하여 저장한 경우의 데이터 크기 및 캐시 크기를 증가시킴으로써 적중률을 비교하였다. 교체 기법은 단일 캐시를 사용한 경우에는 Log(size)+LRU 사용하였으며, 이중 캐시를 사용한 경우, 1CS는 Log(size)+LRU와 2CS는 Log(size)+LFU와 PLC를 사용하였다. 그리고 단일 캐시는 S로 표시하였으며 이중 캐시는 D로 표시하였다.

먼저, 실험에서 각 캐시의 크기는 4K ~ 2M 바이트까지 변화시키면서 실험한 것이다. 그 결과 (그림 9)(a)와 같이 이중 캐시를 사용할 경우, 캐시의 적중률이 단일 캐시의 적중률에 비해 높은 것을 알 수 있다. 이는 기존 연구에서도 데이터를 캐시 영역에 많이 저장함으로써 적중률이 높아진다는 실험 결과를 뒷받침하는 결과라 할 수 있으며, 데이터의 재사용성이 매우 높음을 나타냄으로써 전체적인 캐시 적중률이 높아진다는 것으로 판단된다. 그러나 일정 크기 이상에서는 캐시 적중률의 변화가 거의 나타나지 않는 것을 알 수 있다. 단일 캐시의 경우에는 캐시 크기가 64 ~ 128K 이상이면 캐시 적중률의 증가가 현저히 둔화된다. 또한, 이중 캐시의 경우에도 캐시 크기가 512K 이상이면 캐시 적중률의 변화가 거의 없다는 것을 알 수 있다. 이는 캐시 설계 시 캐시 크기를 무조건 크게 하기보다는 비용 대비 효율측면을 고려한다면 각 캐시를 적절한 크기로 구성할 필요가 있음을 시사하는 것으로 볼 수 있다.



(a) 캐시 크기에 따른 적중률 비교



(b) 데이터 크기에 따른 적중률 비교
그림 9. 단일 캐시와 이중 캐시의 적중률
(Hit-ratio of Single Cache and Dual Cache)

데이터 크기에 대한 적중률 실험은 데이터의 크기가 큰 멀티미디어 데이터와 작은 일반 데이터를 일정한 비율로 저장하여 데이터의 크기에 따른 적중률을 비교하였다. 이 실험은 캐시의 크기와 밀접한 연관이 있으므로 단일 캐시와 이중 캐시의 크기를 동일하게 설정하여 실험하였다. (그림 9)(b)와 같이 단일 캐시는 이중 캐시보다 캐시 크기에 따른 실험 결과처럼 적중률이 감소되는 것을 알 수 있다. 이는 데이터의 크기가 큰 경우에는 선인출 시 적은 수의 블록을 읽어옴으로써 이들이 가까운 장래에 사용될 때 캐시의 적중될 가능성이 낮아진다. 반면, 데이터의 크기가 작은 경우에는 선인출 시 많은 수의 블록을 읽어옴으로써 미리 캐시에 적재되어 있던 데이터들이 교체될 가능성이 낮아진다. 따라서 전체적인 적중률은 데이터의 크기에 따라 교체 주기가 빈번히 발생함으로 이중 캐시보다 적중률이 현저히 감소하였다.

그러므로 이중 캐시를 이용한 캐싱 능력 향상은 네트워크 트래픽을 감소시켜 사용자 응답 시간과 서버의 부하를 감소시킬 수 있기 때문에 결과적으로 단일 캐시에 비해 제안 기법이 시스템의 처리 성능을 개선할 수 있음을 기대할 수 있다.

Ⅵ. 결론 및 향후 연구

본 논문에서는 이종의 언어로 구현된 푸시 어플리케이션들 간 상호운용성을 제공하는 CORBA를 기반으로 이중 캐시의 푸시 에이전트 모델을 설계하였다. 제안된 모델에서는 서로 다른 상이한 언어로 구현된 기존의 푸시 어플리케이션들이 다른 업체들의 어플리케이션과 호환되어 사용될 수 없는 문제를 해결하기 위해 푸시 에이전트 객체를 CORBA의 분산 객체에 적용하여 생성하고 상호운용성을 제공하는 에이전트 플랫폼을 구현함으로써 어플리케이션 간의 상호 호환성을 지원하도록 하였다. 또한, 서버 과부하 및 네트워크 부하를 감소시킬 수 있도록 두 개의 캐시 저장소를 효율적으로 관리하는 캐시 관리 에이전트를 개발하였고, 각 에이전트들의 객체 참조자를 관리하는 네이밍 에이전트를 통해 원격 에이전트 객체 접근 시 객체 참조를 제공받음으로서 에이전트 객체에 대한 위치 투명성을 보장하도록 하였다.

한편, 기존 푸시 기술의 문제인 지속적인 정보 제공 및 동일한 정보의 중복 요청으로 인한 서버 과부하와 네트워크 부하를 감소시키기 위해 클라이언트와 서버 간에 캐시 서버를 추가하였고, 캐시에서의 데이터 적중률을 향상시키기 위한 이중 캐시 구조와 캐시 교체 기법을 제시하였다. 이중 캐시 구조는 1CS와 2CS를 가진 구조로, 캐시 교체 기법을 두 저장소에 단계적으로 적용시켜 가장 많은 사용자 요청이 발생할 데이터들을 여과하여 관리함으로써 데이터 적중률을 향상시킬 수 있도록 하였다. 또한, 캐시 내 데이터들의 효용성을 증대시키기 위해 Log(size)+LRU와 Log(size)+LFU, PLC 등의 교체 정책을 기반으로 데이터 갱신과 삭제를 수행하는 새로운 캐시 교체 전략과 알고리즘을 제시하였다. 이는 여러 교체 기법을 사용하였으나 교체 시의 시간복잡도와 캐시 서버의 과부하, 데이터 적중률과의 상관관계를 분석하여 캐시 교체가 최적화될 수 있도록 하는 교체 전략이다. 따라서 제안 기법에 대한 실험 결과 이중 캐시를 적용한 제안 기법의 적중률 향상을 확인할 수 있었고, 이에 따라 제안 알고리즘이 사용자의 응답 속도를 개선함을 기대할 수 있다.

향후 연구 과제로는 에이전트들 간의 위치 투명성과 보안 관리를 위한 시스템 모니터링의 확장이 요구되고, 사용자에게 제공되는 정보들의 연관성을 추가함으로써 사용자 정보 요구와 연관된 데이터들을 제공해줄 수 있는 서비스 방안이 연구되어져야 할 것이다.

참고문헌

[1] M.Hansson, "Push Technology-The Next BigThing?", "http://www.tcm.hut.fi/Opinnot/Tik-1.350/Tehtavat/essays/push.html".

[2] M.Bhide, "Adaptive push-pull : dissseminating dynamic web data", IEEE Trans on Computer, Vol.51, No.6, pp.652-668, 2002. 6.

[3] 이윤정, "Push 기술과 정보배포 기술의 분석 및 동향 연구", 한국정보처리학회 추계학술발표논문집, 제7권, 제2호, 2000.

[4] 정혜영, "자바를 이용한 CORBA 형정의 이벤트 서비스의 설계 및 구현", 한국컴퓨터산업교육학회, Vol.1, No.1, 2000.

[5] Marc Wennink, "The push tree problem", SPAA, Vol.1, No.1, pp.318-319, 2001.

[6] Pablo Rodriguez, "Analysis of Web Caching Architectures : Hierarchical and Distributed Caching", ACM Transctions on Networking, Vol.9, No.4, pp.404-418, 2001. 8.

[7] Mohammad Raunak, "Implications of Proxy Caching for Provisioning Networks and Servers", IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, Vol.20, No.7, 2002. 9.

[8] Jian Yin, "Engineering web cache consistency", ACM Trans on Internet Technology, Vol.2, No.3, pp.224-259, 2002. 8.

[9] T.Liao, "Global Information Broadcast : an architecture for internet push channels", IEEE Internet Computing, Vol.4, No.4, pp.16-25, 2000. 7.

[10] Prasenjit Sarkar, "Hint-based cooperative caching", ACM Trans on Computer Systems, Vol.18, No.4, pp.387-419, 2000. 11.

[11] Hal Berghel, "Responsible Web caching", Communications of the ACM, Vol.45, No.9, pp.15-20, 2002. 9.

[12] P.Krishnan, "The cache location problem", IEEE Trans on Networking, Vol.8, No.5, pp.568-582, 2000. 10.

[13] Sandra G. Dykes, "Limitations and Benefits of Cooperative Proxy Caching", IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, Vol.20, No.7, 2002. 9.

[14] JooYong Kim, "A Proxy Server Structure and its Cache Consistency Mechanism at the Network Bottleneck", Journal of Natural Sciences, Sookmyoung Women's Univ., Vol.10, pp.121-127, 1999.

[15] Roland Peter Wooster, "Optimizing Response Time Rather than Hit Rates of WWW Proxy Caches", Master Thesis, Virgina Polytechnic Institute and State Unversity, Dec. 1996.

[16] D.Keen, "Cache coherence in intelligent memory systems", IEEE Trans on Computer, Vol.52, No.7, pp.960-966, 2003. 7.

저 자 소 개

김 광 종

1993년 군산대학교 컴퓨터정보과학과(학사)
1999년 군산대학교 대학원 컴퓨터정보과학과(이학석사)
2004년 군산대학교 대학원 컴퓨터정보과학과(이학박사)
〈관심분야〉 이동 에이전트, 분산객체시스템, 능동 데이터베이스

고 현

2001년 군산대학교 컴퓨터정보과학과(학사)
2003년 군산대학교 대학원 컴퓨터정보과학과(이학석사)
2005년 군산대학교 대학원 컴퓨터정보과학과(박사과정)
〈관심분야〉 에이전트, 분산객체시스템, 멀티미디어 데이터베이스

김 영 자

1993년 군산대학교 컴퓨터정보과학과(학사)
2000년 순천대학교 대학원 컴퓨터정보과학과(이학석사)
2005년 군산대학교 대학원 컴퓨터정보과학과(박사수료)
2003년~현재 인천기능대학 교수
〈관심분야〉 에이전트, 분산객체시스템, CDN

이 연 식

1982년 전남대학교 전자계산학과(학사)
1984년 전남대학교 대학원 전자계산학과(이학석사)
1994년 전북대학교대학원 전산응용공학과(공학박사)
1995년~1997년 군산대학교 교무부처장
1997년~1998년 University of Missouri 교환교수
1999년~2001년 군산대학교 전자계산소 소장
1998년~현재 군산대학교 컴퓨터정보과학과 교수
〈관심분야〉 번역기 이론, 객체지향시스템, 능동시스템, 지능형에이전트