

# Peer-to-Peer Proxy Cache Server

Version 0.01, September 21, 2007

Document Author(s):

Project Sponsor:

<b>1</b>	<b>Revision History .....</b>	<b>1</b>
<b>2</b>	<b>Introduction.....</b>	<b>2</b>
<b>3</b>	<b>Overview .....</b>	<b>2</b>
<b>4</b>	<b>Detailed Design.....</b>	<b>4</b>
4.1	<i>P2P application message format.....</i>	4
	HTTP Client Request Headers .....	6
4.2	<i>Software Modules .....</i>	7
4.2.1	Connection Manager.....	8
4.2.2	Parser .....	8
4.2.3	Cache Manager .....	9
4.2.4	Cache .....	9
4.2.5	Disk Manager .....	10
4.2.6	Util, Error Handler and, Configure .....	10
<b>5</b>	<b>Todos .....</b>	<b>10</b>
<b>6</b>	<b>Reference .....</b>	<b>10</b>

## 1 Revision History

Ver.	Date	Author	Change Description
0.01	9/21/2007	behroozn	Initial draft

## 2 Introduction

Peer-to-peer (P2P) systems currently generate a major fraction of the total Internet traffic accounting for as much as 60-70% of the traffic in some Internet Service Providers (ISPs). Furthermore, it is expected that the amount of P2P traffic will even increase in the future. We explore the potential of deploying proxy caches in different Autonomous Systems (ASs) with the goal of reducing the cost incurred by Internet service providers and alleviating the load on the Internet backbone. We conducted an eight-month measurement study to analyze the P2P characteristics that are relevant to caching, such as object popularity, popularity dynamics, and object size. Our study shows that the popularity of P2P objects can be modeled by a Mandelbrot-Zipf distribution, and that several workloads exist in P2P traffic. Guided by our findings, we develop a novel caching algorithm for P2P traffic that is based on object segmentation, and proportional partial admission and eviction of objects. Our trace-based simulations show that with a relatively small cache size, less than 10% of the total traffic, a byte hit rate of up to 35% can be achieved by our algorithm, which is close to the byte hit rate achieved by an off-line optimal algorithm with complete knowledge of future requests. Our results also show that our algorithm achieves a byte hit rate that is at least 40% more, and at most 200%, the byte hit rate of the common web caching algorithms. Furthermore, our algorithm is robust in the face of aborted downloads, which is a common case in P2P systems.

Here, we include the usecase diagram for the p2p proxy cache system, the class diagram for the software architecture and finally descriptions on modules and data structures. In the rest of this document, we first examine give an overview on the system and its responsibilities. Then, it is followed by the design details.

## 3 Overview

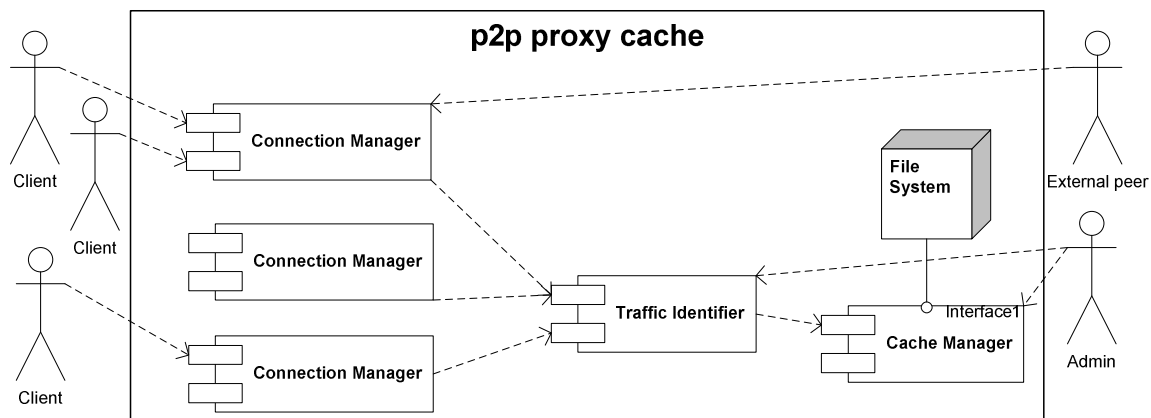
The usecase diagram which is shown in Fig. 1 described the behavior of the system. In general, the system consists of three functional level, (1) *connection manager*, (2) *traffic identifier*, and (3) *cache manager*. In the following paragraphs, these parts are described in detail.

The connection manager is responsible for listening to the incoming connections, pass the data to the traffic identifier module, get the response from traffic identifier module, and finally encapsulate the data within network packets and forward them to the appropriate destination. It is also responsible for the connections on the opposite direction –data received from external network. Ideally, the connection manager should work transparently. A transparent proxy (or correctly naming “intercepting proxy”), is a combination of a proxy server and a gateway. All the connections by the clients should pass through the gateway and gateway automatically forwards these connections to the proxy server. In this case all the connections would be forwarded to the proxy cache without any configuration or knowledge on the client side. However, in this release (v.01)

the cache server is not intercepting. Thus, the p2p client should be configured in order to use our proxy server. Most of the p2p client applications can be easily configured. In our experiments we used Limewire [??], an open source p2p application written in Java which works in Gnutella network.

The traffic identifier receives the data from the connection manager and first decides if this data is known p2p traffic or not. The p2p traffic recognition can be done in three ways. The earliest approach to this problem is mapping of known ip-addresses or port numbers. Second approach use application signature analysis. Third approach which is introduced recently is by considering the p2p traffic behavior pattern. For example, the p2p sessions are usually established for a long time interval, they transfer large amount of data and, they start with a TCP handshaking following by a several UDP connections. In this release, our traffic identifier module uses the application signatures in order to recognize the p2p traffic. This approach shows more promising results in compare with the first approach and it can be implemented more easily compare to the third approach which is fairly new and need more research.

At last, the cache manager gets a unique identifier (UID) for the requested file range and searches its database to find out if it already stores the whole or any part of the file on the disk. In case of a hit the file will be served to the traffic identifier. The traffic identifier is also responsible to let the cache manager knows about the downloaded segments. In this case if the cache manager decides, based on some replacement policies, to store the file, it can get the file from the traffic identifier module.



**Figure 1: p2p proxy cache component diagram**

## 4 Detailed Design

We present detailed design that includes data structure, functions and, operations in this section.

### 4.1 P2P application message format

The Gnutella protocol [??] use several messages in order to make communication among the peers. We choose the Gnutella protocol first because its one of the top three most popular p2p networks and second, it is an open-source protocol. After *bootstrapping*, *handshaking* and, sending *ping and pong* messages, a client can send a *query* message in order to search for one or more keyword(s). This query message will propagate through the Gnutella overlay network and if any of the peers have a file which its properties match part of the query, a *query hit* message will be generated and will be sent back to the requested client.

Payload of query and query hit messages are shown in tables below. Query and query hit messages usually does not contain enough information for the caching purpose, because they only have a few keywords and does not address a file precisely. The requested file can be identified later on by the requester. The requester received several query hit messages and will decide to download some of these files by sending an HTTP download message. This download message contains enough data for the cache server. A typical client request message can be seen in the following.

A typical **query** message payload:

Bytes	Field name	Description
0-1	Minimum Speed (Flags)	The minimum speed (in kb/second) of servants that should respond to this message. A servant receiving a Query message with a Minimum Speed field of n kb/s <b>SHOULD</b> only respond with a Query Hit if it is able to communicate at a speed $\geq$ n kb/s.
2-	Search Criteria	This field is terminated by a NUL (0x00). See section 2.2.7.3 for rules and information on how to Interpret the Search Criteria
Rest	Extensions Block	OPTIONAL. The rest of the query message is used for extensions to the original query format. The allowed extension types are GGEP, HUGE and XML (see Section 2.3 and Appendixes 1 and 2).  If two or more of these extension types exist together, they are separated by a 0x1C (file separator) byte. Since GGEP blocks can contain 0x1C bytes, the GGEP block, if present, <b>MUST</b> be located after any HUGE and XML blocks.

A typical **query hit** message payload:

Bytes	Field name	Description
0	Number of Hits	The number of query hits in the result set.
1-2	Port	The port number on which the responding host can accept incoming HTTP file requests. This is usually the same port as is used for Gnutella network traffic, but any port <b>MAY</b> be used.
3-6	IP Address	The IP address of the responding host. Note: This field is in big-endian format.
7-10	Speed	The speed (in kb/second) of the responding host.
11-	Result Set	A set of responses to the corresponding Query. This set contains <i>Number_of_Hits</i> elements.
x	Extended QHD	This block is not strictly required, but strongly recommended. It is sometimes called EQHD, or (incorrectly) just QHD.
x	Private Data	Undocumented vendor-specific data. This field continues till the servant Identifier, which uses the last 16 bytes of the message.
Last 16	Servent Identifier	A 16-byte string uniquely identifying the responding servant on the network. This <b>SHOULD</b> be constant for all Query Hit messages emitted by a servant and is typically some function of the servant's network address. The servant Identifier is mainly used for routing the Push Message.

## HTTP Client Request Headers

Header	Status	Usage	Example
Host:	Recommended (mandatory in HTTP/1.1)	Public address of the HTTP server, as used by the client	Host : 192.0.2.1:6346
User-Agent:	Mandatory in Gnutella (recommended in HTTP)	The user agent (for the client only)	User-Agent : LimeWire/4.8.2 (Pro) User-Agent : BearShare/2.9.1
Connection: keep-alive	Optional	Indicates to a HTTP/1.0 server to keep the connection persistent. In absence of this request header, the client MUST close itself the connection after receiving the response, if the server is HTTP/1.0, and returns Content-Length and/or Request-Range headers, and has not already closed the connection. Recommended only for HTTP/1.0-only clients that may perform multiple successive requests to the same server.	Connection: keep-alive
Connection: close	Optional	No effect on most HTTP/1.0 servers. Indicates to a HTTP/1.1 server that a persistent connection is not needed. A HTTP/1.1 server will not close the connection itself, but will indicate to the client to close the connection after parsing the server response.	Connection: close
Range:	Mandatory with some Gnutella-based servers (optional in HTTP)	Requested range	Range : bytes=4932766-5066083
X-Gnutella-Alternate-Location:	Recommended	Known alternate locations for the file (HUGE extension)	(see HTTP Server Response Headers)
X-Alt:	Recommended	Replacement for X-Gnutella-Alternate-Location.	(see HTTP Server Response Headers)
X-Features:	Optional	Indicates support of the listed features.	(see HTTP Server Response Headers)

A request captured by our cache server listening on a Limewire client is like this:

```
GET /uri-res/N2R?urn:sha1:I63A6CQTDPN2GDYY3HLTEHZIJIO6WOUU HTTP/1.1
HOST: 75.28.136.191:26265
User-Agent: LimeWire/4.12.6
X-Queue: 0.1
X-Gnutella-Content-URN: urn:sha1:I63A6CQTDPN2GDYY3HLTEHZIJIO6WOUU
Range: bytes=262144-393215
X-Features: queue/0.1
X-Downloaded: 225352
```

ADD SOME DETAIL ON THIS.

## 4.2 Software Modules

In this section we describe the software modules and major data structures. Fig. 2 gives a high level on the main classes and the relations between them. The diagram will have more detail as we proceed in the coding. In the following subsections we described each of these classes in detail.

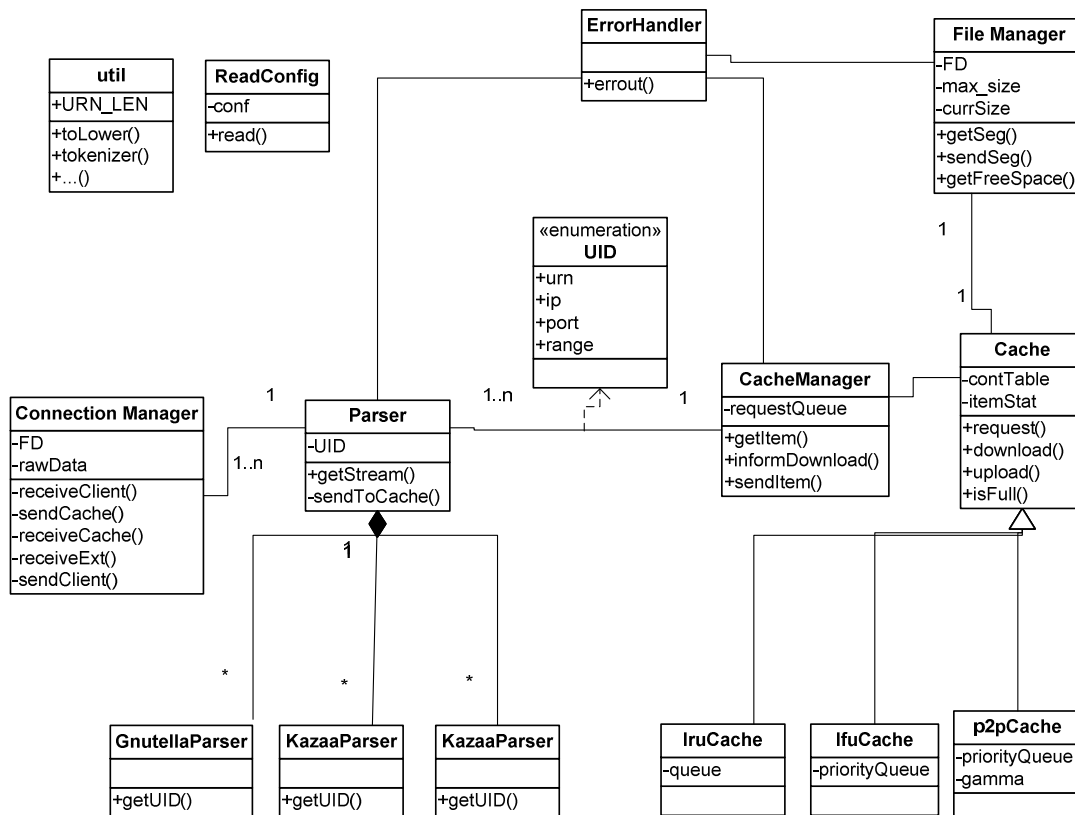


Figure 2: p2p proxy cache class diagram

## 4.2.1 Connection Manager

ADD MORE DETAIL.

```
Connection *getNewConnection(struct Server *this, Connection *creator);
void closeConnection(Connection *);
void appendWriteBuf(Connection *this, const char *data, int len);

SOCKETFD connectToNextAddress(Connection *this);
SOCKETFD connectToAddress(Connection *this);

void closeMarkedConnection(struct Server *this, Connection *conn);
int readConnection(Connection *this);
int writeConnection(Connection *this);
void nonBlockSocket(SOCKETFD webConn, unsigned long on);
void closePipeFds(Connection *this);
```

## 4.2.2 Parser

Parser is our traffic identifier module. Parser receives the raw packets from the connection manager. A raw packet contains the HTTP data like what is shown in Section 4.1. After parsing the packet all the required information will be put in a UID data structure. A UID contains the following fields:

```
typedef struct UID {
    std::string urn;
    std::string ip_s;
    int port;
    long lRange;
    long rRange;
}UID;
```

Parser has a few subclasses for each p2p protocol. These subclasses are GnutellaParser, KazaaParser and, etc. Other fields of parser class are as follows:

```
public:
    std::string type;           // which protocols it should parse
                                // eg. Gnutella, KaZaA and, etc.

    parser(std::string);       // constructor get the "type" as input
    ~parser();
    int getStream(std::string, int, CacheManager*);

private:
    // put the UID in cache manager queue
    int sendToCache(UID*, CacheManager*);
    UID *uid;
```



### 4.2.3 Cache Manager

Cache manager is responsible for getting the requests from the parser and ask them from the cache. The main data structure in cache manager is a queue which contains the requests passed by the parser. This queue has a synchronized access because the parser write in it and in the same time cache manager reads the data from it. Cache manager has a “run” method which upon running, forks a “threadGetReq” process. “threadGetReq” read the items from the request queue as long as the queue is not empty. If the queue does not have any more items a semaphore wait will be called. The process would be locked until it receives a signal from the parser, which means a new item has been pushed into the queue.

Cache manager major methods and data types are:

```
public:
    std::queue<UID*> q;
    CacheManager(int, int);
    int setReq(UID*);
    int run();
    UID* getReq();
    pthread_mutex_t mut;
    sem_t sem;

private:
    Cache *cache;
```

### 4.2.4 Cache

Cache is an abstract class which contains different subclasses. Every cache object would be polymorph to either LRUCache, LFUCache, P2PCache or, etc. A simple data type defined in Cache is “Item”:

```
typedef struct {
    std::string name; // 32 char
    long piece_size; // in byte
    long item_size; // in byte
} Item;
```

Items are the objects should be pushed into the cache queue. This queue should be defined as a “std::deque” because we need to search within the contents of the queue to find the requested objects. Cache functions with a few details are:

```
public:
Cache(std::string, int); // constructor take the path on the disk
                          // and size of the cache in MB
int request(UID*); // receiving a request for an item
int download(UID*); // receiving a request for a file range
                          // (the request response should be positive)
int upload(UID, int); // send the cache a downloaded file to store
int isFull(); // check if the cache is full
```

```
private:
std::deque<*Item> q;           // contains the Item objects
int size;                     // size of the cache in MB
```

#### **4.2.4.1 LRU Cache**

It is a cache with LRU replacement policy. It should be implemented with a simple FIFO queue.

#### **4.2.4.2 LFU Cache**

It is a cache with LFU replacement policy. It should be implemented with a priority queue (or a heap) in order to keep the most requested object on the top. In this way the less popular items at the bottom of queue will be replaced by the new items.

#### **4.2.4.3 P2P Cache**

It used the P2P algorithm proposed in [4]. The implementation needs a heap to prioritize the items.

#### **4.2.5 Disk Manager**

UNDER DEVELOPMENT.

#### **4.2.6 Util, Error Handler and, Configure**

All the utilities are put in the util.cc. config.cc would configure the parse, cache manager and all the other module which need configuration. Error handler would deal with the exceptions.

### **5 Todos**

ADD THE ITEMS.

### **6 Reference**