

# BlinkDB Documentation

Generated by Doxygen 1.9.1



<b>1 Class Index</b>	<b>1</b>
1.1 Class List	1
<b>2 File Index</b>	<b>3</b>
2.1 File List	3
<b>3 Class Documentation</b>	<b>5</b>
3.1 BloomFilter Class Reference	5
3.1.1 Detailed Description	5
3.1.2 Constructor & Destructor Documentation	5
3.1.2.1 BloomFilter()	5
3.1.3 Member Function Documentation	6
3.1.3.1 contains()	6
3.1.3.2 insert()	6
3.1.3.3 remove()	7
3.2 Client Class Reference	7
3.2.1 Detailed Description	8
3.2.2 Constructor & Destructor Documentation	8
3.2.2.1 Client()	8
3.2.2.2 ~Client()	9
3.2.3 Member Function Documentation	9
3.2.3.1 close_server()	9
3.2.3.2 del()	9
3.2.3.3 get()	10
3.2.3.4 server_init()	11
3.2.3.5 set()	11
3.2.4 Member Data Documentation	12
3.2.4.1 buffer	12
3.2.4.2 buffer_size	12
3.2.4.3 ip_addr	12
3.2.4.4 port	13
3.3 Dict Class Reference	13
3.3.1 Detailed Description	13
3.3.2 Constructor & Destructor Documentation	14
3.3.2.1 Dict()	14
3.3.2.2 ~Dict()	14
3.3.3 Member Function Documentation	14
3.3.3.1 add()	14
3.3.3.2 enableResize()	15
3.3.3.3 find()	16
3.3.3.4 get_size_of_dict()	16
3.3.3.5 isRehashing()	17
3.3.3.6 rehash()	17

3.3.3.7 remove()	17
3.3.3.8 replace()	18
3.3.3.9 size()	19
3.4 LoadBalancer Class Reference	19
3.4.1 Detailed Description	19
3.4.2 Constructor & Destructor Documentation	20
3.4.2.1 LoadBalancer()	20
3.4.2.2 ~LoadBalancer()	20
3.4.3 Member Function Documentation	20
3.4.3.1 server_init()	20
3.5 LRUCache Class Reference	23
3.5.1 Detailed Description	24
3.5.2 Constructor & Destructor Documentation	24
3.5.2.1 LRUCache() [1/2]	24
3.5.2.2 LRUCache() [2/2]	24
3.5.2.3 ~LRUCache()	25
3.5.3 Member Function Documentation	25
3.5.3.1 del() [1/2]	25
3.5.3.2 del() [2/2]	25
3.5.3.3 get() [1/2]	26
3.5.3.4 get() [2/2]	27
3.5.3.5 max_memory() [1/2]	28
3.5.3.6 max_memory() [2/2]	28
3.5.3.7 memory_usage() [1/2]	29
3.5.3.8 memory_usage() [2/2]	29
3.5.3.9 printList()	29
3.5.3.10 set() [1/2]	30
3.5.3.11 set() [2/2]	30
3.5.3.12 size() [1/2]	31
3.5.3.13 size() [2/2]	32
3.5.4 Member Data Documentation	32
3.5.4.1 dict	32
3.5.4.2 head	32
3.5.4.3 storage	33
3.5.4.4 tail	33
3.5.4.5 value	33
3.6 LRUCache::Node Struct Reference	33
3.6.1 Detailed Description	34
3.6.2 Constructor & Destructor Documentation	34
3.6.2.1 Node()	34
3.6.3 Member Data Documentation	34
3.6.3.1 key	34

3.6.3.2 next	34
3.6.3.3 prev	34
3.6.3.4 value	35
3.7 PersistenceKVStore Class Reference	35
3.7.1 Detailed Description	35
3.7.2 Constructor & Destructor Documentation	35
3.7.2.1 PersistenceKVStore()	35
3.7.2.2 ~PersistenceKVStore()	36
3.7.3 Member Function Documentation	36
3.7.3.1 get()	36
3.7.3.2 insert()	37
3.7.3.3 remove()	38
3.7.3.4 remove_db()	39
3.8 Server Class Reference	39
3.8.1 Detailed Description	40
3.8.2 Constructor & Destructor Documentation	40
3.8.2.1 Server()	40
3.8.2.2 ~Server()	41
3.8.3 Member Function Documentation	41
3.8.3.1 encode_resp()	41
3.8.3.2 handle_command()	42
3.8.3.3 init()	44
3.8.3.4 parse_resp()	46
3.9 ServerAdd Struct Reference	47
3.9.1 Member Data Documentation	47
3.9.1.1 ip	47
3.9.1.2 port	47
3.10 Trie Class Reference	48
3.10.1 Constructor & Destructor Documentation	48
3.10.1.1 Trie()	48
3.10.1.2 ~Trie()	48
3.10.2 Member Function Documentation	48
3.10.2.1 insert()	49
3.10.2.2 isDeleted()	49
3.10.2.3 remove()	49
3.10.2.4 search()	50
3.11 TrieNode Class Reference	50
3.11.1 Member Data Documentation	51
3.11.1.1 children	51
3.11.1.2 file_offset	51
3.11.1.3 isDeleted	51

<b>4 File Documentation</b>	<b>53</b>
4.1 lib/bloomfilter.h File Reference	53
4.2 lib/client.h File Reference	54
4.3 lib/dict.h File Reference	55
4.3.1 Detailed Description	56
4.3.2 Function Documentation	56
4.3.2.1 freeString()	57
4.3.2.2 stringCompare()	57
4.3.2.3 stringDup()	57
4.3.2.4 stringHash()	58
4.4 lib/load_balancer.h File Reference	58
4.4.1 Detailed Description	59
4.5 lib/lru_cache_v0.h File Reference	59
4.6 lib/lru_cache_v1.h File Reference	60
4.7 lib/persistence_kv_store.h File Reference	61
4.8 lib/server.h File Reference	62
4.9 lib/tire.h File Reference	63
4.10 lib/types.h File Reference	64
4.11 src/blink_cli.cpp File Reference	65
4.11.1 Macro Definition Documentation	66
4.11.1.1 BUFFER_SIZE	66
4.11.1.2 SERVER_IP	66
4.11.1.3 SERVER_PORT	66
4.11.2 Function Documentation	66
4.11.2.1 command_loop()	66
4.11.2.2 main()	68
4.12 src/blink_server.cpp File Reference	68
4.12.1 Function Documentation	69
4.12.1.1 main()	69
4.12.2 Variable Documentation	70
4.12.2.1 BUFFER_SIZE	70
4.12.2.2 MAX_EVENTS	70
4.12.2.3 MAX_MEMORY_BYTES	71
4.12.2.4 SERVER_IP	71
4.12.2.5 SERVER_PORT	71
4.13 src/blink_server_with_lb.cpp File Reference	71
4.13.1 Macro Definition Documentation	72
4.13.1.1 BUFFER_SIZE	72
4.13.1.2 MAX_EVENTS	72
4.13.1.3 MAX_MEMORY_BYTES	72
4.13.1.4 SERVER_IP	72
4.13.1.5 SERVER_PORT	72

---

4.13.2 Function Documentation . . . . .	72
4.13.2.1 main() . . . . .	73
4.13.2.2 parse_key() . . . . .	73
4.13.2.3 serverThread() . . . . .	74
4.14 utils/create_non_locking_socket.h File Reference . . . . .	75
4.14.1 Function Documentation . . . . .	76
4.14.1.1 create_non_locking_socket() . . . . .	77
4.15 utils/set_nonblocking.h File Reference . . . . .	78
4.15.1 Function Documentation . . . . .	79
4.15.1.1 set_nonblocking() . . . . .	79
<b>Index</b>	<b>81</b>





# Chapter 1

## Class Index

### 1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<a href="#">BloomFilter</a>	Implements a simple Bloom filter for fast key existence checks . . . . .	5
<a href="#">Client</a>	A class to interact with a Blink server using RESP (Blink Serialization Protocol) . . . . .	7
<a href="#">Dict</a>	A dictionary (hash table) implementation with dynamic resizing and rehashing . . . . .	13
<a href="#">LoadBalancer</a>	Handles load balancing by distributing client requests to backend servers . . . . .	19
<a href="#">LRUCache</a>	Implements a Least Recently Used (LRU) cache with memory constraints . . . . .	23
<a href="#">LRUCache::Node</a>	Represents a node in the doubly linked list for the LRU cache . . . . .	33
<a href="#">PersistenceKVStore</a>	A persistent key-value store with background rewriting and indexing . . . . .	35
<a href="#">Server</a>	Implements a Blink-compatible server with an LRU-based in-memory database . . . . .	39
<a href="#">ServerAdd</a>	. . . . .	47
<a href="#">Trie</a>	. . . . .	48
<a href="#">TrieNode</a>	. . . . .	50



## Chapter 2

# File Index

### 2.1 File List

Here is a list of all files with brief descriptions:

lib/ <a href="#">bloomfilter.h</a> . . . . .	53
lib/ <a href="#">client.h</a> . . . . .	54
lib/ <a href="#">dict.h</a> . . . . .	
Implementation of a dictionary (hash table) with rehashing support . . . . .	55
lib/ <a href="#">load_balancer.h</a> . . . . .	
Implements a non-blocking load balancer using epoll and consistent hashing . . . . .	58
lib/ <a href="#">lru_cache_v0.h</a> . . . . .	59
lib/ <a href="#">lru_cache_v1.h</a> . . . . .	60
lib/ <a href="#">persistence_kv_store.h</a> . . . . .	61
lib/ <a href="#">server.h</a> . . . . .	62
lib/ <a href="#">tire.h</a> . . . . .	63
lib/ <a href="#">types.h</a> . . . . .	64
src/ <a href="#">blink_cli.cpp</a> . . . . .	65
src/ <a href="#">blink_server.cpp</a> . . . . .	68
src/ <a href="#">blink_server_with_lb.cpp</a> . . . . .	71
utils/ <a href="#">create_non_locking_socket.h</a> . . . . .	75
utils/ <a href="#">set_nonblocking.h</a> . . . . .	78



## Chapter 3

# Class Documentation

### 3.1 BloomFilter Class Reference

Implements a simple Bloom filter for fast key existence checks.

```
#include <bloomfilter.h>
```

#### Public Member Functions

- [BloomFilter](#) (int `_size`=10000)  
*Constructor to initialize the Bloom filter with a given size.*
- void [insert](#) (const std::string &`_key`)  
*Inserts a key into the Bloom filter.*
- bool [contains](#) (const std::string &`_key`)  
*Checks if a key exists in the Bloom filter.*
- void [remove](#) (const std::string &`_key`)  
*Removes a key from the Bloom filter (Note: Bloom filters generally do not support removals correctly).*

#### 3.1.1 Detailed Description

Implements a simple Bloom filter for fast key existence checks.

#### 3.1.2 Constructor & Destructor Documentation

##### 3.1.2.1 BloomFilter()

```
BloomFilter::BloomFilter (  
    int _size = 10000 ) [explicit]
```

Constructor to initialize the Bloom filter with a given size.

**Parameters**

<code>_size</code>	The size of the Bloom filter (default: 10,000).
--------------------	---

```
52 : filter(_size, false), filter_size(_size) {}
```

**3.1.3 Member Function Documentation****3.1.3.1 contains()**

```
bool BloomFilter::contains (
    const std::string & _key )
```

Checks if a key exists in the Bloom filter.

**Parameters**

<code>_key</code>	The key to check.
-------------------	-------------------

**Returns**

True if the key is possibly in the filter, false otherwise.

```
60 {
61     return filter[hashKey(_key)];
62 }
```

Here is the caller graph for this function:

**3.1.3.2 insert()**

```
void BloomFilter::insert (
    const std::string & _key )
```

Inserts a key into the Bloom filter.

**Parameters**

<code>_key</code>	The key to insert.
-------------------	--------------------

```

55 {
56     filter[hashKey(_key)] = true;
57 }

```

Here is the caller graph for this function:



### 3.1.3.3 remove()

```

void BloomFilter::remove (
    const std::string & _key )

```

Removes a key from the Bloom filter (Note: Bloom filters generally do not support removals correctly).

#### Parameters

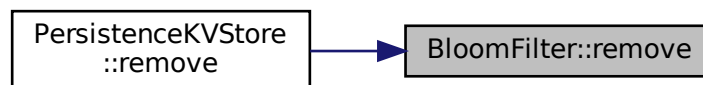
<code>_key</code>	The key to remove.
-------------------	--------------------

```

65 {
66     filter[hashKey(_key)] = false;
67 }

```

Here is the caller graph for this function:



The documentation for this class was generated from the following file:

- [lib/bloomfilter.h](#)

## 3.2 Client Class Reference

A class to interact with a Blink server using RESP (Blink Serialization Protocol).

```
#include <client.h>
```

## Public Member Functions

- [Client](#) (std::string \_ip\_addr, int \_port, int \_buffer\_size)  
*Constructor for [Client](#) class.*
- [~Client](#) ()  
*Destructor to free allocated memory.*
- int [server\\_init](#) ()  
*Initializes the connection to the Blink server.*
- std::string [set](#) (const std::string &\_key, const std::string &\_value)  
*Sends a SET command to store a key-value pair.*
- std::string [get](#) (const std::string &\_key)  
*Sends a GET command to retrieve the value of a key.*
- std::string [del](#) (const std::string &\_key)  
*Sends a DEL command to delete a key.*
- void [close\\_server](#) ()  
*Closes the connection to the Blink server.*

## Public Attributes

- int [buffer\\_size](#)  
*Buffer size for reading responses.*
- std::string [ip\\_addr](#)  
*IP address of the Blink server.*
- char \* [buffer](#) = nullptr  
*Dynamic buffer for receiving data.*
- int [port](#)  
*Port number of the Blink server.*

### 3.2.1 Detailed Description

A class to interact with a Blink server using RESP (Blink Serialization Protocol).

### 3.2.2 Constructor & Destructor Documentation

#### 3.2.2.1 Client()

```
Client::Client (
    std::string _ip_addr,
    int _port,
    int _buffer_size ) [inline]
```

Constructor for [Client](#) class.

#### Parameters

<a href="#">_ip_addr</a>	IP address of the Blink server.
<a href="#">_port</a>	Port number of the Blink server.
<a href="#">_buffer_size</a>	Size of the buffer for reading responses.



```
32         : ip_addr(_ip_addr)
33     {
34         port = _port;
35         buffer_size = _buffer_size;
36         buffer = new char[buffer_size];
37     }
```

### 3.2.2.2 ~Client()

```
Client::~~Client ( ) [inline]
```

Destructor to free allocated memory.

```
43     {
44         delete[] buffer;
45     }
```

## 3.2.3 Member Function Documentation

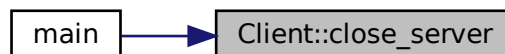
### 3.2.3.1 close\_server()

```
void Client::close_server ( ) [inline]
```

Closes the connection to the Blink server.

```
118     {
119         close(sock);
120     }
```

Here is the caller graph for this function:



### 3.2.3.2 del()

```
std::string Client::del (
    const std::string & _key ) [inline]
```

Sends a DEL command to delete a key.

**Parameters**

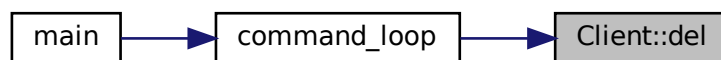
<code>_key</code>	The key.
-------------------	----------

**Returns**

Response from the Blink server.

```
110     {  
111         return decode_resp(send_req(encode_command("DEL " + _key)));  
112     }
```

Here is the caller graph for this function:

**3.2.3.3 get()**

```
std::string Client::get (  
    const std::string & _key ) [inline]
```

Sends a GET command to retrieve the value of a key.

**Parameters**

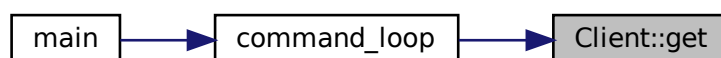
<code>_key</code>	The key.
-------------------	----------

**Returns**

Response from the Blink server.

```
100     {  
101         return decode_resp(send_req(encode_command("GET " + _key)));  
102     }
```

Here is the caller graph for this function:



### 3.2.3.4 server\_init()

```
int Client::server_init ( ) [inline]
```

Initializes the connection to the Blink server.

#### Returns

Socket descriptor or -1 on failure.

```
52     {
53         sock = 0;
54         struct sockaddr_in serv_addr;
55
56         // Create socket
57         if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
58         {
59             std::cerr << "Socket creation error" << std::endl;
60             return -1;
61         }
62
63         serv_addr.sin_family = AF_INET;
64         serv_addr.sin_port = htons(port);
65
66         // Convert IPv4 address from text to binary
67         if (inet_pton(AF_INET, ip_addr.c_str(), &serv_addr.sin_addr) <= 0)
68         {
69             std::cerr << "Invalid address / Address not supported" << std::endl;
70             return -1;
71         }
72
73         // Connect to server
74         if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
75         {
76             std::cerr << "Connection failed" << std::endl;
77             return -1;
78         }
79
80         return 1;
81     }
```

Here is the caller graph for this function:



### 3.2.3.5 set()

```
std::string Client::set (
    const std::string & _key,
    const std::string & _value ) [inline]
```

Sends a SET command to store a key-value pair.

**Parameters**

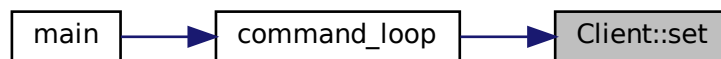
<code>_key</code>	The key.
<code>_value</code>	The value.

**Returns**

Response from the Blink server.

```
90     {  
91         return decode_resp(send_req(encode_command("SET " + _key + " " + _value)));  
92     }
```

Here is the caller graph for this function:



## 3.2.4 Member Data Documentation

### 3.2.4.1 buffer

```
char* Client::buffer = nullptr
```

Dynamic buffer for receiving data.

### 3.2.4.2 buffer\_size

```
int Client::buffer_size
```

Buffer size for reading responses.

### 3.2.4.3 ip\_addr

```
std::string Client::ip_addr
```

IP address of the Blink server.

### 3.2.4.4 port

```
int Client::port
```

Port number of the Blink server.

The documentation for this class was generated from the following file:

- lib/[client.h](#)

## 3.3 Dict Class Reference

A dictionary (hash table) implementation with dynamic resizing and rehashing.

```
#include <dict.h>
```

### Public Member Functions

- [Dict](#) (std::function< unsigned int(const void \*)> hashFunc, std::function< void \*(const void \*)> keyDupFunc, std::function< void \*(const void \*)> valDupFunc, std::function< int(const void \*, const void \*)> keyCompareFunc, std::function< void(void \*)> keyDestructorFunc, std::function< void(void \*)> valDestructorFunc)  
*Constructor for the dictionary.*
- [~Dict](#) ()  
*Destructor for the dictionary.*
- void [enableResize](#) (bool enable)  
*Enables or disables automatic resizing.*
- int [add](#) (void \*key, void \*val)  
*Adds a key-value pair to the dictionary with automatic rehashing.*
- int [replace](#) (void \*key, void \*val)  
*Replaces a key's value in the dictionary.*
- int [remove](#) (const void \*key)  
*Removes a key from the dictionary with incremental rehashing.*
- void \* [find](#) (const void \*key)  
*Finds a key in the dictionary.*
- int [rehash](#) (int n)  
*Performs a rehash operation.*
- bool [isRehashing](#) ()  
*Checks if rehashing is in progress.*
- size\_t [get\\_size\\_of\\_dict](#) ()  
*Retrieves the total memory usage of the dictionary for keys, values.*
- int [size](#) ()  
*Retrieves the total no of keys in the dictionary.*

### 3.3.1 Detailed Description

A dictionary (hash table) implementation with dynamic resizing and rehashing.

### 3.3.2 Constructor & Destructor Documentation

#### 3.3.2.1 Dict()

```
Dict::Dict (
    std::function< unsigned int(const void *)> hashFunc,
    std::function< void *(const void *)> keyDupFunc,
    std::function< void *(const void *)> valDupFunc,
    std::function< int(const void *, const void *)> keyCompareFunc,
    std::function< void(void *)> keyDestructorFunc,
    std::function< void(void *)> valDestructorFunc ) [inline]
```

Constructor for the dictionary.

##### Parameters

<i>hashFunc</i>	Hash function.
<i>keyDupFunc</i>	Key duplication function.
<i>valDupFunc</i>	Value duplication function.
<i>keyCompareFunc</i>	Key comparison function.
<i>keyDestructorFunc</i>	Key destructor function.
<i>valDestructorFunc</i>	Value destructor function.

```
123         : hashFunction(hashFunc), keyDup(keyDupFunc), valDup(valDupFunc),
124           keyCompare(keyCompareFunc), keyDestructor(keyDestructorFunc),
125           valDestructor(valDestructorFunc)
126     {
127         _dictInit(&d);
128     }
```

#### 3.3.2.2 ~Dict()

```
Dict::~Dict ( ) [inline]
```

Destructor for the dictionary.

```
134     {
135         _dictClear(&d);
136     }
```

### 3.3.3 Member Function Documentation

#### 3.3.3.1 add()

```
int Dict::add (
    void * key,
    void * val ) [inline]
```

Adds a key-value pair to the dictionary with automatic rehashing.

## Parameters

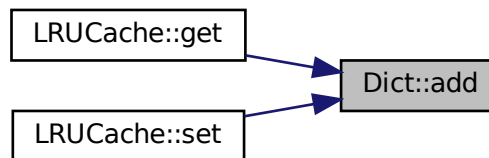
<i>key</i>	Key pointer.
<i>val</i>	Value pointer.

## Returns

0 on success, 1 on failure.

```
154     {
155         // Check if resize is needed before adding
156         if (_dictShouldResize())
157         {
158             // Calculate new size based on current usage
159             unsigned long newSize = d.ht[0].used * 2;
160             _dictExpand(&d, newSize);
161         }
162
163         // Perform a rehash step if rehashing is in progress
164         if (dictIsRehashing(&d))
165             _dictRehashStep();
166
167         return dictAdd(&d, key, val);
168     }
```

Here is the caller graph for this function:



### 3.3.3.2 enableResize()

```
void Dict::enableResize (
    bool enable ) [inline]
```

Enables or disables automatic resizing.

## Parameters

<i>enable</i>	True to enable resizing, false to disable.
---------------	--

```
143     {
144         dict_can_resize = enable;
145     }
```

### 3.3.3.3 find()

```
void* Dict::find (
    const void * key ) [inline]
```

Finds a key in the dictionary.

#### Parameters

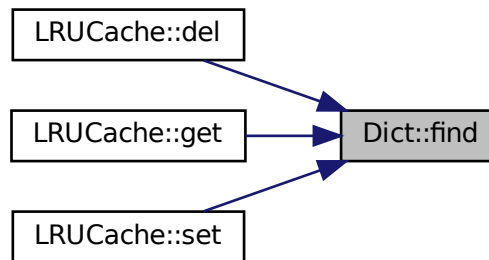
<i>key</i>	Key pointer.
------------	--------------

#### Returns

Pointer to the value if found, nullptr otherwise.

```
201 {
202     // Perform a rehash step if rehashing is in progress
203     if (dictIsRehashing(&d))
204         _dictRehashStep();
205
206     dictEntry *he = dictFind(&d, key);
207     return he ? he->v.val : nullptr;
208 }
```

Here is the caller graph for this function:



### 3.3.3.4 get\_size\_of\_dict()

```
size_t Dict::get_size_of_dict ( ) [inline]
```

Retrieves the total memory usage of the dictionary for keys, values.

#### Returns

**size\_t** The total size of the dictionary in bytes.

```
234 {
235     return total_size_of_dict;
236 }
```



### 3.3.3.5 isRehashing()

```
bool Dict::isRehashing ( ) [inline]
```

Checks if rehashing is in progress.

#### Returns

True if rehashing, false otherwise.

```
225     {  
226         return dictIsRehashing(&d);  
227     }
```

### 3.3.3.6 rehash()

```
int Dict::rehash (  
    int n ) [inline]
```

Performs a rehash operation.

#### Parameters

<i>n</i>	Number of steps to rehash.
----------	----------------------------

#### Returns

0 on completion, 1 if rehashing is ongoing.

```
216     {  
217         return dictRehash(&d, n);  
218     }
```

### 3.3.3.7 remove()

```
int Dict::remove (  
    const void * key ) [inline]
```

Removes a key from the dictionary with incremental rehashing.

#### Parameters

<i>key</i>	Key pointer.
------------	--------------

#### Returns

0 on success, 1 if key not found.

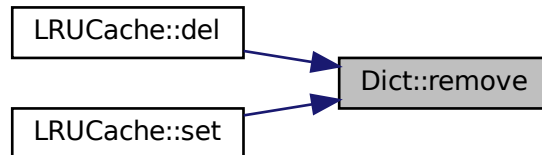
```
187     {  
188         // Perform a rehash step if rehashing is in progress
```

```

189         if (dictIsRehashing(&d))
190             _dictRehashStep();
191
192         return dictDelete(&d, key);
193     }

```

Here is the caller graph for this function:



### 3.3.3.8 replace()

```

int Dict::replace (
    void * key,
    void * val ) [inline]

```

Replaces a key's value in the dictionary.

#### Parameters

<i>key</i>	Key pointer.
<i>val</i>	Value pointer.

#### Returns

0 if key already exists and value is replaced, 1 if key is newly added.

```

177     {
178         return dictReplace(&d, key, val);
179     }

```

Here is the caller graph for this function:



### 3.3.3.9 size()

```
int Dict::size ( ) [inline]
```

Retrieves the total no of keys in the dictionary.

#### Returns

int To total no of keys in the dict

```
244     {
245         return d.ht[0].used + d.ht[1].used;
246     }
```

Here is the caller graph for this function:



The documentation for this class was generated from the following file:

- [lib/dict.h](#)

## 3.4 LoadBalancer Class Reference

Handles load balancing by distributing client requests to backend servers.

```
#include <load_balancer.h>
```

### Public Member Functions

- [LoadBalancer](#) (std::string lb\_ip, int lb\_port, std::vector< [ServerAdd](#) > &servers\_add, int buffer\_size, int max\_events)  
*Constructs a [LoadBalancer](#) instance.*
- [~LoadBalancer](#) ()  
*Destructor to clean up resources.*
- void [server\\_init](#) (parsingKeyFuncPtr func)  
*Initializes the server and starts handling client requests.*

### 3.4.1 Detailed Description

Handles load balancing by distributing client requests to backend servers.

### 3.4.2 Constructor & Destructor Documentation

#### 3.4.2.1 LoadBalancer()

```
LoadBalancer::LoadBalancer (
    std::string lb_ip,
    int lb_port,
    std::vector< ServerAdd > & servers_add,
    int buffer_size,
    int max_events )
```

Constructs a [LoadBalancer](#) instance.

##### Parameters

<i>lb_port</i>	The port the load balancer listens on.
<i>serves_add</i>	The list of server addresses and ports.
<i>buffer_size</i>	The size of the buffer for client messages.
<i>max_events</i>	The maximum number of events handled by epoll.

```
81                                     : servers_add_(servers_add)
82 {
83     lb_ip_ = lb_ip;
84     lb_port_ = lb_port;
85     buffer_size_ = buffer_size;
86     max_events_ = max_events;
87
88     buffer_ = new char[buffer_size];
89     for (auto &server_add : servers_add)
90     {
91         int key = hashKey(server_add.ip + std::to_string(server_add.port));
92         hash_ring_.insert(key);
93         server_map_[key] = server_add;
94     }
95 }
```

#### 3.4.2.2 ~LoadBalancer()

```
LoadBalancer::~~LoadBalancer ( )
```

Destructor to clean up resources.

```
98 {
99     delete[] buffer_;
100 }
```

### 3.4.3 Member Function Documentation

#### 3.4.3.1 server\_init()

```
void LoadBalancer::server_init (
    parsingKeyFuncPtr func )
```

Initializes the server and starts handling client requests.

## Parameters

<i>func</i>	Function pointer for parsing the key from client messages.
-------------	--

```

120 {
121     struct sockaddr_in address;
122     int lb_sockfd, epoll_fd;
123     struct epoll_event event, events[max_events_];
124     int addrlen = sizeof(address);
125
126     lb_sockfd = create_non_locking_socket(lb_ip_, lb_port_, address);
127
128     epoll_fd = epoll_create1(0);
129     if (epoll_fd == -1)
130     {
131         perror("[LB]: Epoll creation failed");
132         exit(EXIT_FAILURE);
133     }
134
135     event.events = EPOLLIN;
136     event.data.fd = lb_sockfd;
137
138     if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, lb_sockfd, &event) == -1)
139     {
140         perror("[LB]: Epoll_ctl failed");
141         exit(EXIT_FAILURE);
142     }
143
144     std::cout << "[LB]: Load Balancer listening on port " << lb_port_ << std::endl;
145
146     while (true)
147     {
148         int num_events = epoll_wait(epoll_fd, events, max_events_, -1);
149         if (num_events == -1)
150         {
151             perror("[LB]: Epoll wait failed");
152             break;
153         }
154
155         for (int i = 0; i < num_events; ++i)
156         {
157             int sock_fd = events[i].data.fd;
158
159             if (sock_fd == lb_sockfd)
160             {
161                 // New client connection
162                 int client_fd = accept(lb_sockfd, (struct sockaddr *)&address, (socklen_t *)&addrlen);
163
164                 // char client_ip[INET_ADDRSTRLEN];
165                 // inet_ntop(AF_INET, &address.sin_addr, client_ip, INET_ADDRSTRLEN);
166                 // int client_port = ntohs(address.sin_port);
167
168                 if (client_fd == -1)
169                 {
170                     perror("[LB]: Accept failed");
171                     continue;
172                 }
173
174                 set_nonblocking(client_fd);
175
176                 event.events = EPOLLIN | EPOLLET;
177                 event.data.fd = client_fd;
178
179                 if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, client_fd, &event) == -1)
180                 {
181                     perror("Epoll_ctl client add failed");
182                     close(client_fd);
183                     continue;
184                 }
185
186                 // std::cout << "[LB]: New client connected: " << client_ip << ":" << client_port <<
std::endl;
187             }
188             else
189             {
190                 // Handle client request
191                 memset(buffer_, 0, buffer_size_);
192                 int bytes_read = recv(sock_fd, buffer_, buffer_size_, 0);
193
194                 if (bytes_read > 0)
195                 {
196                     buffer_[bytes_read] = '\0';
197                     std::string key = parse_key(buffer_, bytes_read);
198                     ServerAdd server_add = getServer(key);
199
200                     // Create a new connection to the backend server

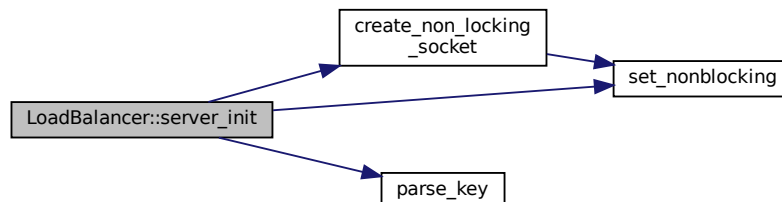
```

```

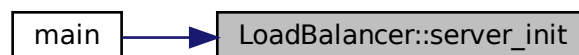
201         int server_fd = socket(AF_INET, SOCK_STREAM, 0);
202         struct sockaddr_in server_addr;
203         server_addr.sin_family = AF_INET;
204         server_addr.sin_port = htons(server_add.port);
205
206         if (inet_pton(AF_INET, server_add.ip.c_str(), &server_addr.sin_addr) <= 0)
207         {
208             perror("[LB]: Invalid IP address");
209             exit(EXIT_FAILURE);
210         }
211
212         if (connect(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == 0)
213         {
214             send(server_fd, buffer_, bytes_read, 0);
215             // char response[buffer_size_] = {0};
216             int resp_bytes = recv(server_fd, buffer_, buffer_size_, 0);
217             if (resp_bytes > 0)
218             {
219                 send(sock_fd, buffer_, resp_bytes, 0);
220             }
221             close(server_fd); // Close backend server connection after response
222         }
223         else
224         {
225             perror("[LB]:server connection failed");
226             close(server_fd);
227         }
228     }
229     else
230     {
231         // Client disconnected
232         // std::cout << "[LB]: Client " << sock_fd << " disconnected." << std::endl;
233         epoll_ctl(epoll_fd, EPOLL_CTL_DEL, sock_fd, nullptr);
234         close(sock_fd);
235     }
236 }
237 }
238 }
239
240 close(lb_sockfd);
241 close(epoll_fd);
242 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



The documentation for this class was generated from the following file:

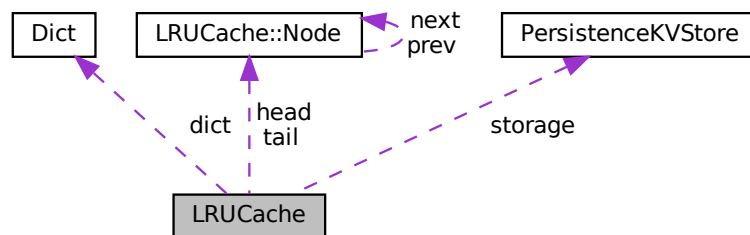
- lib/[load\\_balancer.h](#)

## 3.5 LRUCache Class Reference

Implements a Least Recently Used (LRU) cache with memory constraints.

```
#include <lru_cache_v0.h>
```

Collaboration diagram for LRUCache:



### Classes

- struct [Node](#)  
*Represents a node in the doubly linked list for the LRU cache.*

### Public Member Functions

- [LRUCache](#) (size\_t max\_mem)
- void [set](#) (const std::string &key, const std::string &value)
- bool [get](#) (const std::string &key, std::string &value)
- bool [del](#) (const std::string &key)
- size\_t [memory\\_usage](#) () const
- size\_t [max\\_memory](#) () const
- size\_t [size](#) () const
- [LRUCache](#) (size\_t max\_mem)  
*Constructs a new LRU Cache with the specified memory limit.*
- [~LRUCache](#) ()  
*Destroys the LRU Cache and frees all allocated memory.*
- std::string [get](#) (const void \*key)  
*Retrieves the value for a given key.*
- void [printList](#) ()  
*Prints the current state of the cache for debugging.*
- void [set](#) (void \*key, void \*value)  
*Adds or updates a key-value pair in the cache.*
- int [del](#) (const void \*key)

- *Deletes a key-value pair from the cache.*
- `size_t memory_usage ()`  
*Gets the current memory usage of the cache.*
- `size_t max_memory ()`  
*Gets the maximum memory limit of the cache.*
- `size_t size ()`  
*Gets the number of items in the cache.*

## Public Attributes

- `Dict dict`
- `Node * head`
- `Node * tail`
- `PersistenceKVStore storage`
- `std::string value`

### 3.5.1 Detailed Description

Implements a Least Recently Used (LRU) cache with memory constraints.

This class provides a memory-constrained LRU cache implementation that evicts least recently used items when memory limits are exceeded. It uses a doubly linked list for tracking usage order and a dictionary for O(1) lookups.

### 3.5.2 Constructor & Destructor Documentation

#### 3.5.2.1 LRUCache() [1/2]

```
LRUCache::LRUCache (
    size_t max_mem ) [inline]
37 : current_memory_usage(0), max_memory_bytes(max_mem) {}
```

#### 3.5.2.2 LRUCache() [2/2]

```
LRUCache::LRUCache (
    size_t max_mem ) [inline]
```

Constructs a new LRU Cache with the specified memory limit.

#### Parameters

<code>max_mem</code>	Maximum memory limit in bytes
----------------------	-------------------------------

```
61                                     :dict(stringHash, nullptr, nullptr, stringCompare, freeKey, freeValue),
    current_memory_usage(0), storage("./blink")
```



```

62     {
63         max_memory_bytes = max_mem;
64         head = new Node(strdup("-1"), strdup("-1"));
65         tail = new Node(strdup("-1"), strdup("-1"));
66         head->next = tail;
67         tail->prev = head;
68     }

```

### 3.5.2.3 ~LRUCache()

```
LRUCache::~~LRUCache ( ) [inline]
```

Destroys the LRU Cache and frees all allocated memory.

```

74     {
75         freeNode(head);
76         freeNode(tail);
77     }

```

## 3.5.3 Member Function Documentation

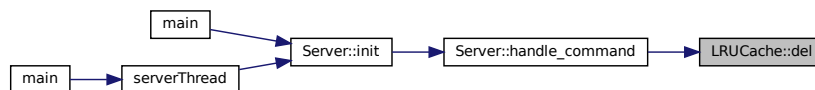
### 3.5.3.1 del() [1/2]

```

bool LRUCache::del (
    const std::string & key ) [inline]
{
    auto it = cache_map.find(key);
    if (it == cache_map.end()) {
        return false; // Key not found
    }
    // Update memory usage
    current_memory_usage -= it->second->key.size() + it->second->value.size();
    // Remove from list and map
    cache_list.erase(it->second);
    cache_map.erase(it);
    return true;
}

```

Here is the caller graph for this function:



### 3.5.3.2 del() [2/2]

```

int LRUCache::del (
    const void * key ) [inline]

```

Deletes a key-value pair from the cache.

**Parameters**

<i>key</i>	The key to delete
------------	-------------------

**Returns**

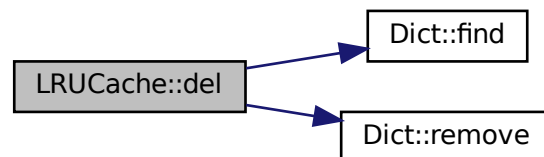
int 0 if successful, 1 if the key was not found

```

174     {
175         Node *retrievedValue = static_cast<Node *>(dict.find(key));
176
177         if (retrievedValue)
178         {
179             remove(retrievedValue);
180             current_memory_usage -= getSize((char *)retrievedValue->key) + getNodeSize(retrievedValue);
181             dict.remove(retrievedValue->key);
182             return 0;
183         }
184         return 1;
185     }

```

Here is the call graph for this function:

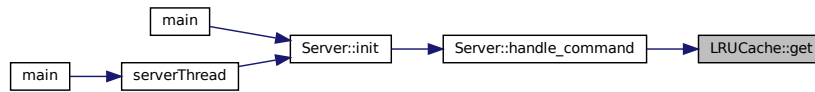
**3.5.3.3 get() [1/2]**

```

bool LRUCache::get (
    const std::string & key,
    std::string & value ) [inline]
{
79
80     auto it = cache_map.find(key);
81     if (it == cache_map.end()) {
82         return false; // Key not found
83     }
84
85     // Update access time and move to front of list
86     it->second->last_accessed = std::chrono::steady_clock::now();
87
88     // Move to front (most recently used)
89     if (it->second != cache_list.begin()) {
90         cache_list.splice(cache_list.begin(), cache_list, it->second);
91     }
92
93     value = it->second->value;
94     return true;
95 }

```

Here is the caller graph for this function:



### 3.5.3.4 get() [2/2]

```
std::string LRUCache::get (
    const void * key ) [inline]
```

Retrieves the value for a given key.

If the key exists, it moves the corresponding node to the front of the list to mark it as most recently used.

#### Parameters

<i>key</i>	The key to look up
------------	--------------------

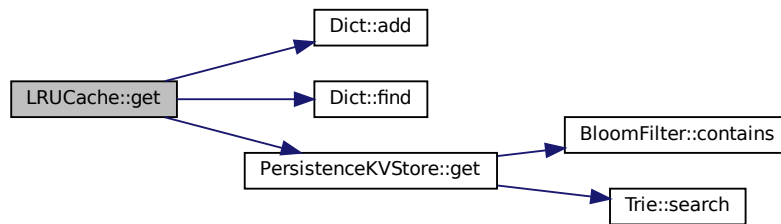
#### Returns

std::string The value associated with the key, or "-1" if not found

```

89     {
90         Node *retrievedValue = static_cast<Node *>(dict.find(key));
91         if (!retrievedValue) {
92             std::string value;
93             if (storage.get(std::string((char *)key), value)) {
94                 void *key1 = (void *)key;
95                 void* value1 = (void *) strdup(value.c_str());
96                 Node *node = new Node(key1, value1);
97                 dict.add(key1, node);
98                 add(node);
99                 current_memory_usage += getSize((char *)key) + getNodeSize(node);
100                 return value;
101             };
102             return "-1";
103         }
104
105         remove(retrievedValue);
106         add(retrievedValue);
107         return static_cast<char *>(retrievedValue->value);
108     }
109 }
```

Here is the call graph for this function:



### 3.5.3.5 max\_memory() [1/2]

```
size_t LRUCache::max_memory ( ) [inline]
```

Gets the maximum memory limit of the cache.

#### Returns

size\_t Maximum memory limit in bytes

```

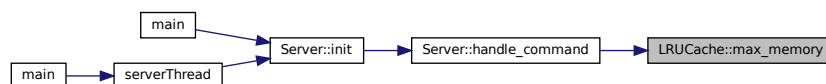
203 {
204     return max_memory_bytes;
205 }
```

### 3.5.3.6 max\_memory() [2/2]

```

size_t LRUCache::max_memory ( ) const [inline]
120 {
121     return max_memory_bytes;
122 }
```

Here is the caller graph for this function:



### 3.5.3.7 memory\_usage() [1/2]

```
size_t LRUCache::memory_usage ( ) [inline]
```

Gets the current memory usage of the cache.

#### Returns

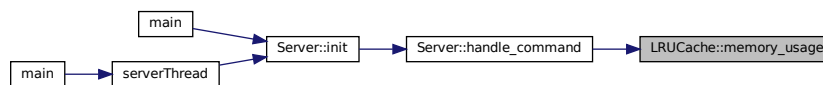
size\_t Current memory usage in bytes

```
193     {
194         return current_memory_usage;
195     }
```

### 3.5.3.8 memory\_usage() [2/2]

```
size_t LRUCache::memory_usage ( ) const [inline]
115     {
116         return current_memory_usage;
117     }
```

Here is the caller graph for this function:



### 3.5.3.9 printList()

```
void LRUCache::printList ( ) [inline]
```

Prints the current state of the cache for debugging.

```
115     {
116         Node *curr = head->next;
117         std::cout << "Cache state: ";
118         while (curr != tail)
119         {
120             std::cout << "[" << static_cast<char *>(curr->key) << ":" << static_cast<char *>(curr->value) <<
121             "]" ";
122             curr = curr->next;
123         }
124         std::cout << std::endl;
125     }
```

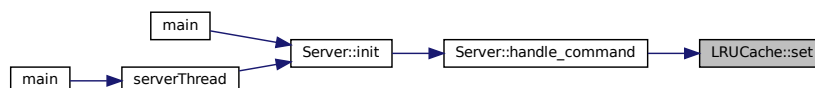
### 3.5.3.10 set() [1/2]

```

void LRUCache::set (
    const std::string & key,
    const std::string & value ) [inline]
{
40
41    // Calculate memory for this entry (key + value + overhead)
42    size_t entry_size = key.size() + value.size() + sizeof(CacheEntry);
43
44    // Check if key already exists
45    auto it = cache_map.find(key);
46    if (it != cache_map.end()) {
47        // Update existing entry
48        current_memory_usage -= it->second->key.size() + it->second->value.size();
49        cache_list.erase(it->second);
50        cache_map.erase(it);
51    }
52
53    // Check if we need to evict entries to make space
54    while (current_memory_usage + entry_size > max_memory_bytes && !cache_list.empty()) {
55        // Evict least recently used item
56        auto last = cache_list.back();
57        std::cout << "LRU Eviction: Removing key '" << last.key << "'" << std::endl;
58        current_memory_usage -= last.key.size() + last.value.size();
59        cache_map.erase(last.key);
60        cache_list.pop_back();
61    }
62
63    // If we still can't fit the new entry, don't add it
64    if (current_memory_usage + entry_size > max_memory_bytes) {
65        std::cerr << "Warning: Entry too large to fit in cache" << std::endl;
66        return;
67    }
68
69    // Add new entry to front of list (most recently used)
70    cache_list.emplace_front(key, value);
71    cache_map[key] = cache_list.begin();
72    current_memory_usage += entry_size;
73
74    // std::cout << "Memory usage: " << current_memory_usage << "/" << max_memory_bytes
75    //          << " bytes (" << (current_memory_usage * 100.0 / max_memory_bytes) << "%)" <<
    std::endl;
76 }

```

Here is the caller graph for this function:



### 3.5.3.11 set() [2/2]

```

void LRUCache::set (
    void * key,
    void * value ) [inline]

```

Adds or updates a key-value pair in the cache.

If adding the new item exceeds the memory capacity, the least recently used items will be evicted until the memory usage is within limits.

## Parameters

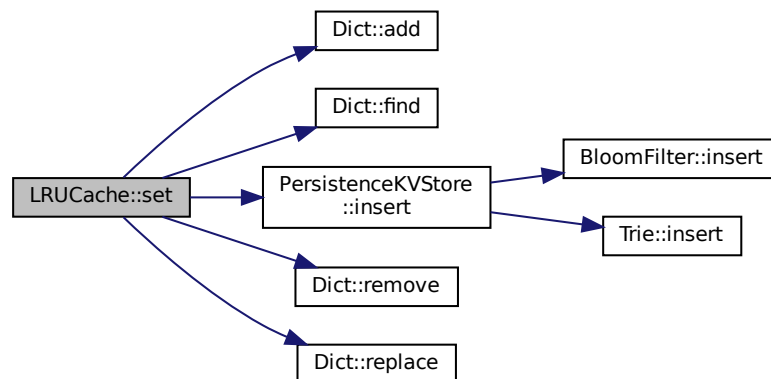
<i>key</i>	Pointer to the key
<i>value</i>	Pointer to the value

```

136     {
137         Node *retrievedValue = static_cast<Node *>(dict.find(key));
138         Node *node = new Node(key, value);
139
140         if (retrievedValue)
141         {
142             current_memory_usage += getNodeSize(node) - getNodeSize(retrievedValue);
143             remove(retrievedValue);
144             dict.replace(key, node);
145             add(node);
146         }
147         else
148         {
149             dict.add(key, node);
150             add(node);
151             current_memory_usage += getSize((char *)key) + getNodeSize(node);
152         }
153
154         if (current_memory_usage >= max_memory_bytes)
155         {
156             Node *nodeToDelete = tail->prev;
157             storage.insert(std::string(strdup((char *)nodeToDelete->key)), std::string(strdup((char
*)nodeToDelete->value)));
158             remove(nodeToDelete);
159             std::cout << current_memory_usage << std::endl;
160             current_memory_usage -= getSize((char *)nodeToDelete->key) + getNodeSize(nodeToDelete);
161             dict.remove(nodeToDelete->key);
162             std::cout << "Eviction happen" << std::endl;
163             std::cout << current_memory_usage << std::endl;
164         }
165     }

```

Here is the call graph for this function:



## 3.5.3.12 size() [1/2]

```
size_t LRUCache::size ( ) [inline]
```

Gets the number of items in the cache.

**Returns**

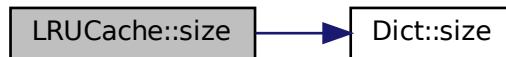
size\_t Number of items in the cache

```

213     {
214         return dict.size();
215     }

```

Here is the call graph for this function:

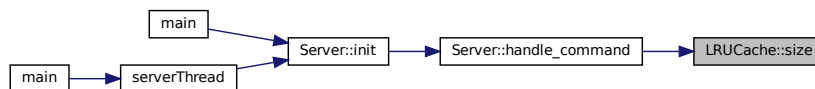
**3.5.3.13 size() [2/2]**

```

size_t LRUCache::size ( ) const [inline]
125     {
126         return cache_map.size();
127     }

```

Here is the caller graph for this function:

**3.5.4 Member Data Documentation****3.5.4.1 dict**

`Dict` `LRUCache::dict`

Dictionary for O(1) lookups

**3.5.4.2 head**

`Node*` `LRUCache::head`

Head of the doubly linked list



### 3.5.4.3 storage

`PersistenceKVStore` LRUCache::storage

### 3.5.4.4 tail

`Node*` LRUCache::tail

Tail of the doubly linked list

### 3.5.4.5 value

`std::string` LRUCache::value

The documentation for this class was generated from the following files:

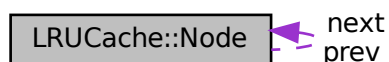
- `lib/lru_cache_v0.h`
- `lib/lru_cache_v1.h`

## 3.6 LRUCache::Node Struct Reference

Represents a node in the doubly linked list for the LRU cache.

```
#include <lru_cache_v1.h>
```

Collaboration diagram for LRUCache::Node:



### Public Member Functions

- `Node` (void \*k, void \*v)  
*Constructs a new `Node` with the given key and value.*

### Public Attributes

- void \* `key`
- void \* `value`
- `Node` \* `next`
- `Node` \* `prev`

### 3.6.1 Detailed Description

Represents a node in the doubly linked list for the LRU cache.

Each node contains a key-value pair and pointers to the next and previous nodes.

### 3.6.2 Constructor & Destructor Documentation

#### 3.6.2.1 Node()

```
LRUCache::Node::Node (
    void * k,
    void * v ) [inline]
```

Constructs a new [Node](#) with the given key and value.

##### Parameters

<i>k</i>	Pointer to the key
<i>v</i>	Pointer to the value

```
40     {
41         key = k;
42         value = v;
43         next = nullptr;
44         prev = nullptr;
45     }
```

### 3.6.3 Member Data Documentation

#### 3.6.3.1 key

```
void* LRUCache::Node::key
```

Pointer to the key

#### 3.6.3.2 next

```
Node* LRUCache::Node::next
```

Pointer to the next node in the list

#### 3.6.3.3 prev

```
Node* LRUCache::Node::prev
```

Pointer to the previous node in the list

### 3.6.3.4 value

```
void* LRUCache::Node::value
```

Pointer to the value

The documentation for this struct was generated from the following file:

- [lib/lru\\_cache\\_v1.h](#)

## 3.7 PersistenceKVStore Class Reference

A persistent key-value store with background rewriting and indexing.

```
#include <persistence_kv_store.h>
```

### Public Member Functions

- [PersistenceKVStore](#) (const std::string \_dbname, int \_bloom\_filter\_size=10000, int rewrite\_interval=5000)  
*Constructor: Initializes the key-value store, loads the index, and starts the rewrite scheduler.*
- [~PersistenceKVStore](#) ()  
*Destructor: Ensures background rewriting stops and closes the file.*
- void [insert](#) (const std::string &\_key, const std::string &\_value)  
*Inserts a key-value pair into the store.*
- bool [get](#) (const std::string &\_key, std::string &\_value)  
*Retrieves the value for a given key.*
- void [remove](#) (const std::string &\_key)  
*Removes a key from the store.*
- void [remove\\_db](#) ()  
*Removes the database file.*

### 3.7.1 Detailed Description

A persistent key-value store with background rewriting and indexing.

### 3.7.2 Constructor & Destructor Documentation

#### 3.7.2.1 PersistenceKVStore()

```
PersistenceKVStore::PersistenceKVStore (
    const std::string _dbname,
    int _bloom_filter_size = 10000,
    int rewrite_interval = 5000 )
```

Constructor: Initializes the key-value store, loads the index, and starts the rewrite scheduler.

## Parameters

<code>_dbname</code>	The database name.
<code>_bloom_filter_size</code>	The size of <a href="#">BloomFilter</a> .
<code>_rewrite_interval</code>	Interval for background rewrite in milliseconds (default: 5000).

```

93     : bloomFilter(_bloom_filter_size)
94 {
95     filename = _dbname + ".txt";
96     tempfilename = _dbname + ".temp.txt";
97     rewrite_interval_ms = _rewrite_interval;
98     stopRewrite.store(false);
99     dataFile.open(filename, std::ios::in | std::ios::out | std::ios::binary);
100     index = new Trie();
101
102     if (!dataFile)
103     {
104         dataFile.open(filename, std::ios::out | std::ios::binary);
105         dataFile.close();
106         dataFile.open(filename, std::ios::in | std::ios::out | std::ios::binary);
107     }
108
109     syncIndex();
110     rewriteThread = std::thread(&PersistenceKVStore::startRewriteScheduler, this);
111 }

```

## 3.7.2.2 ~PersistenceKVStore()

PersistenceKVStore::~PersistenceKVStore ( )

Destructor: Ensures background rewriting stops and closes the file.

```

114 {
115     dataFile.clear();
116     stopRewrite.store(true);
117     if (rewriteThread.joinable())
118     {
119         rewriteThread.join();
120     }
121     dataFile.close();
122 }

```

## 3.7.3 Member Function Documentation

## 3.7.3.1 get()

```

bool PersistenceKVStore::get (
    const std::string & _key,
    std::string & _value )

```

Retrieves the value for a given key.

## Parameters

<code>_key</code>	The key to search for.
<code>_value</code>	Reference to store the retrieved value.

## Returns

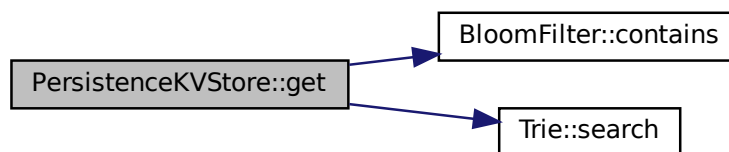
True if the key exists, false otherwise.

```

143 {
144     std::lock_guard<std::mutex> lock(mtx_index);
145     if (!bloomFilter.contains(_key))
146         return false;
147
148     long offset = index->search(_key);
149     if (offset == -1)
150         return false;
151
152     dataFile.clear();
153     dataFile.seekg(offset, std::ios::beg);
154
155     if (!dataFile)
156         return false;
157
158     std::string storedKey;
159     dataFile » storedKey;
160
161     if (storedKey != _key)
162         return false;
163
164     std::getline(dataFile » std::ws, _value);
165
166     return !_value.empty();
167 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



### 3.7.3.2 insert()

```

void PersistenceKVStore::insert (
    const std::string & _key,
    const std::string & _value )

```

Inserts a key-value pair into the store.

## Parameters

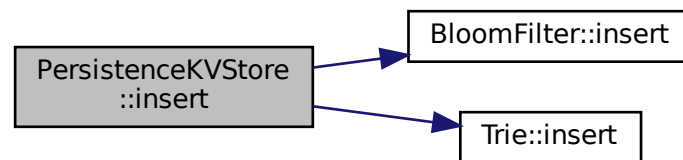
<code>_key</code>	The key to insert.
<code>_value</code>	The corresponding value.

```

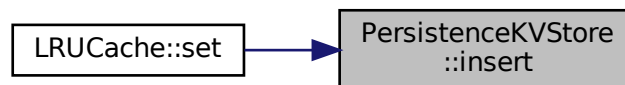
125 {
126     dataFile.clear();
127     dataFile.seekp(0, std::ios::end);
128     long offset = dataFile.tellp();
129     if (offset == -1)
130     {
131         std::cerr << "Error: tellp() returned -1" << std::endl;
132         return;
133     }
134
135     dataFile << _key << " " << _value << std::endl;
136     dataFile.flush();
137     std::lock_guard<std::mutex> lock(mtx_index);
138     index->insert(_key, offset);
139     bloomFilter.insert(_key);
140 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



### 3.7.3.3 remove()

```

void PersistenceKVStore::remove (
    const std::string & _key )

```

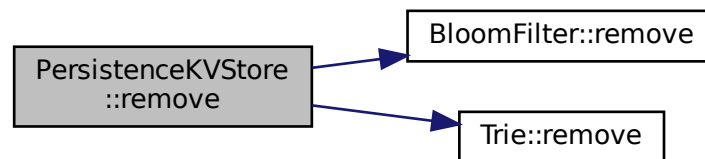
Removes a key from the store.

## Parameters

<code>_key</code>	The key to remove.
-------------------	--------------------

```
170 {  
171     std::lock_guard<std::mutex> lock(mtx_index);  
172     index->remove(_key);  
173     bloomFilter.remove(_key);  
174 }
```

Here is the call graph for this function:



### 3.7.3.4 remove\_db()

```
void PersistenceKVStore::remove_db ( )
```

Removes the database file.

```
177 {  
178     std::remove(filename.c_str());  
179 }
```

The documentation for this class was generated from the following file:

- [lib/persistence\\_kv\\_store.h](#)

## 3.8 Server Class Reference

Implements a Blink-compatible server with an LRU-based in-memory database.

```
#include <server.h>
```

## Public Member Functions

- void [parse\\_resp](#) (const std::string &input, std::vector< std::string > &result)  
*Parses a RESP (Redis Serialization Protocol) formatted string.*
- void [encode\\_resp](#) (std::string &response, bool is\_error)  
*Encodes a response string into RESP format.*
- void [handle\\_command](#) (const std::vector< std::string > &command, std::string &response)  
*Handles client commands and generates appropriate responses.*
- [Server](#) (std::string ip, int port, int buffer\_size, int max\_events, int max\_mem\_bytes)  
*Constructs a [Server](#) object.*
- [~Server](#) ()  
*Destructor to release allocated resources.*
- void [init](#) ()  
*Initializes the server, sets up epoll, and starts listening for connections.*

### 3.8.1 Detailed Description

Implements a Blink-compatible server with an LRU-based in-memory database.

### 3.8.2 Constructor & Destructor Documentation

#### 3.8.2.1 Server()

```
Server::Server (
    std::string ip,
    int port,
    int buffer_size = 2048,
    int max_events = 4096,
    int max_mem_bytes = 1024 * 1024 * 1024 )
```

Constructs a [Server](#) object.

#### Parameters

<i>ip</i>	<a href="#">Server</a> IP address.
<i>port</i>	<a href="#">Server</a> port number.
<i>buffer_size</i>	Buffer size for receiving data.
<i>max_events</i>	Maximum epoll events.
<i>max_mem_bytes</i>	Maximum memory allocation for caching.

```
83     : ip(_ip),
84       database(_max_mem_bytes)
85 {
86     port = _port;
87     buffer_size = _buffer_size;
88     max_mem_bytes = _max_mem_bytes;
89     max_events = _max_events;
90     tag = "[" + ip + ":" + std::to_string(_port) + "] ";
91     buffer = new char[_buffer_size];
92 }
```



### 3.8.2.2 ~Server()

```
Server::~~Server ( )
```

Destructor to release allocated resources.

```
95 {
96     delete[] buffer;
97 }
```

## 3.8.3 Member Function Documentation

### 3.8.3.1 encode\_resp()

```
void Server::encode_resp (
    std::string & response,
    bool is_error )
```

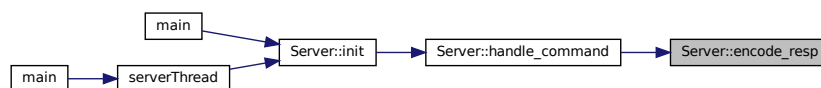
Encodes a response string into RESP format.

#### Parameters

<i>response</i>	Response string to encode.
<i>is_error</i>	Whether the response is an error message.

```
229 {
230     if (is_error)
231     {
232         response = "-ERR " + response + "\r\n";
233     }
234     else if (response.empty())
235     {
236         response = "$-1\r\n";
237     }
238     else
239     {
240         response = "+" + response + "\r\n";
241     }
242 }
```

Here is the caller graph for this function:



### 3.8.3.2 handle\_command()

```
void Server::handle_command (
    const std::vector< std::string > & command,
    std::string & response )
```

Handles client commands and generates appropriate responses.

#### Parameters

<i>command</i>	Parsed command tokens.
<i>response</i>	String to store the response.

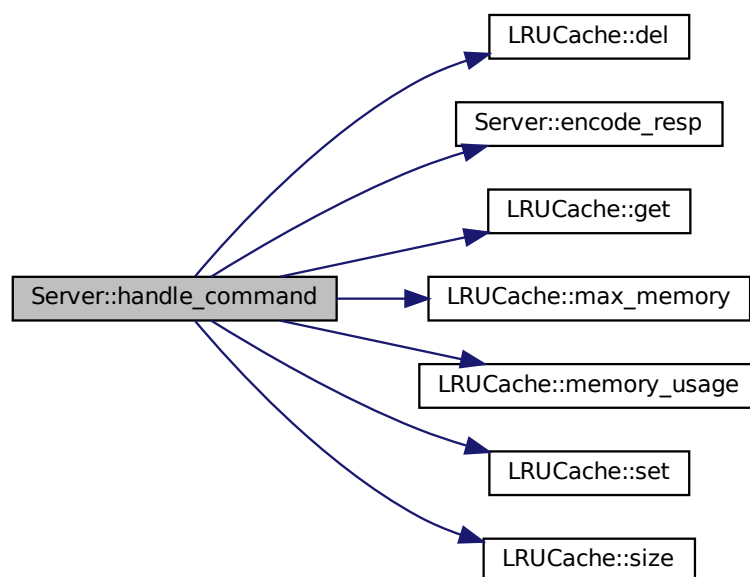
```
245 {
246     if (command.empty())
247     {
248         response = "Invalid command";
249         encode_resp(response, true);
250         return;
251     }
252
253     std::string cmd = command[0];
254     std::transform(cmd.begin(), cmd.end(), cmd.begin(), ::toupper);
255
256     if (cmd == "SET")
257     {
258         if (command.size() < 3)
259         {
260             response = "SET command requires key and value";
261             encode_resp(response, true);
262             return;
263         }
264
265         database.set(strdup(command[1].c_str()), strdup(command[2].c_str()));
266         response = "OK";
267         encode_resp(response, false);
268     }
269     else if (cmd == "GET")
270     {
271         if (command.size() < 2)
272         {
273             response = "GET command requires key";
274             encode_resp(response, true);
275             return;
276         }
277
278         std::string value = database.get(command[1].c_str());
279         if (value != "-1")
280         {
281             response = "$" + std::to_string(value.length()) + "\r\n" + value + "\r\n";
282         }
283         else
284         {
285             response = "$-1\r\n";
286         }
287     }
288     else if (cmd == "DEL")
289     {
290         if (command.size() < 2)
291         {
292             response = "DEL command requires key";
293             encode_resp(response, true);
294             return;
295         }
296
297         int count = 0;
298         for (size_t i = 1; i < command.size(); i++)
299         {
300             count += database.del(command[i].c_str()) ? 1 : 0;
301         }
302
303         response = ":" + std::to_string(count) + "\r\n";
304     }
305
306     else if (cmd == "INFO")
307     {
308         // Add INFO command to get memory usage statistics
309         std::string info = "# Memory\r\n";
310         info += "used_memory:" + std::to_string(database.memory_usage()) + "\r\n";
311         info += "maxmemory:" + std::to_string(database.max_memory()) + "\r\n";
```

```

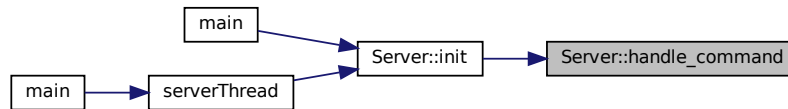
312     info += "maxmemory_policy:allkeys-lru\r\n";
313     info += "# Stats\r\n";
314     info += "keyspace_hits:" + std::to_string(database.size()) + "\r\n";
315
316     response = "$" + std::to_string(info.length()) + "\r\n" + info + "\r\n";
317 }
318 else if (cmd == "CONFIG")
319 {
320     // Basic CONFIG command implementation
321     if (command.size() < 2)
322     {
323         response = "CONFIG command requires subcommand";
324         encode_resp(response, true);
325         return;
326     }
327
328     std::string subcmd = command[1];
329     std::transform(subcmd.begin(), subcmd.end(), subcmd.begin(), ::toupper);
330
331     if (subcmd == "GET" && command.size() >= 3)
332     {
333         std::string param = command[2];
334         std::transform(param.begin(), param.end(), param.begin(), ::tolower);
335
336         if (param == "maxmemory")
337         {
338             response = "*2\r\n$9\r\nmaxmemory\r\n$" +
339             std::to_string(std::to_string(database.max_memory()).length()) +
340             "\r\n" + std::to_string(database.max_memory()) + "\r\n";
341             return;
342         }
343         else if (param == "maxmemory-policy")
344         {
345             response = "*2\r\n$16\r\nmaxmemory-policy\r\n$11\r\nallkeys-lru\r\n";
346             return;
347         }
348         response = "Supported CONFIG commands: GET maxmemory, GET maxmemory-policy";
349         encode_resp(response, false);
350     }
351 }
352 else
353 {
354     response = "Unknown command";
355     encode_resp(response, true);
356 }
357 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



### 3.8.3.3 init()

```
void Server::init ( )
```

Initializes the server, sets up epoll, and starts listening for connections.

```

100 {
101     int server_fd, epoll_fd;
102     struct sockaddr_in address;
103     socklen_t addrlen = sizeof(address);
104     struct epoll_event event, events[max_events];
105
106     server_fd = create_non_locking_socket(ip, port, address);
107
108     epoll_fd = epoll_create1(0);
109     if (epoll_fd == -1)
110     {
111         perror("[Server]: Epoll creation failed");
112         exit(EXIT_FAILURE);
113     }
114
115     event.events = EPOLLIN;
116     event.data.fd = server_fd;
117     if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, server_fd, &event) == -1)
118     {
119         perror("[Server] Epoll_ctl failed");
120         exit(EXIT_FAILURE);
121     }
122
123     std::cout << tag << "Blink-compatible server listening on port " << port << std::endl;
124     std::cout << tag << "Memory limit set to " << (max_mem_bytes / (1024 * 1024)) << " MB with LRU eviction
policy" << std::endl;
125
126     while (true)
127     {
128         int ready_fds = epoll_wait(epoll_fd, events, max_events, -1);
129         if (ready_fds == -1)
130         {
131             perror("Epoll wait failed");
132             break;
133         }
134
135         for (int i = 0; i < ready_fds; i++)
136         {
137             int sock_fd = events[i].data.fd;
138
139             if (sock_fd == server_fd)
140             {
141                 int client_fd = accept(server_fd, (struct sockaddr *)&address, &addrlen);
142
143                 // char client_ip[INET_ADDRSTRLEN];
144                 // inet_ntop(AF_INET, &address.sin_addr, client_ip, INET_ADDRSTRLEN);
145                 // int client_port = ntohs(address.sin_port);
146
147                 if (client_fd == -1)
148                 {
149                     perror("Accept failed");
150                     continue;
151                 }
152
153                 set_nonblocking(client_fd);

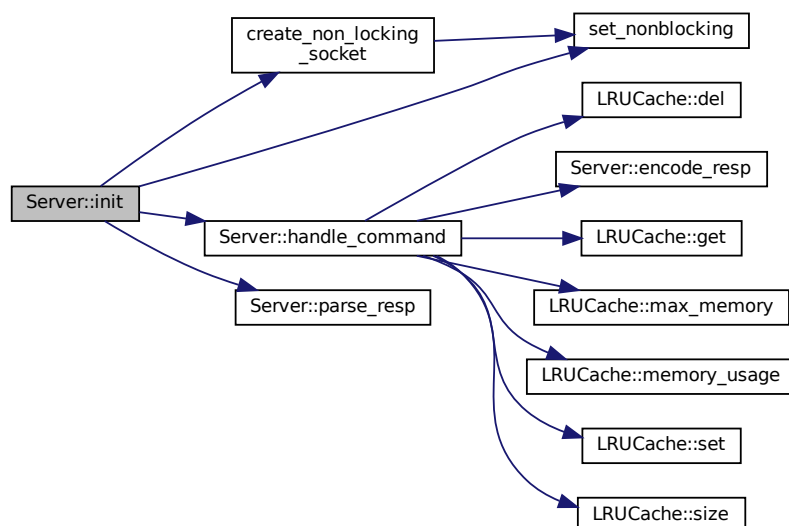
```

```

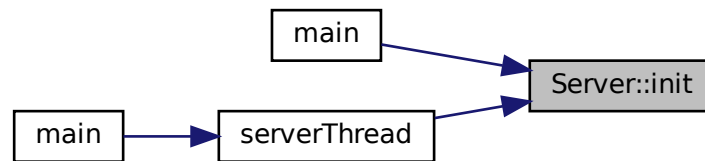
154
155     event.events = EPOLLIN | EPOLLET;
156     event.data.fd = client_fd;
157     if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, client_fd, &event) == -1)
158     {
159         perror("Epoll_ctl client add failed");
160         close(client_fd);
161         continue;
162     }
163
164     // std::cout << tag << "New client connected: " << client_ip << ":" << client_port <<
    std::endl;
165 }
166 else
167 {
168     // memset(buffer, 0, buffer_size);
169     int bytes_read = recv(sock_fd, buffer, buffer_size, 0);
170
171     if (bytes_read > 0)
172     {
173         buffer[bytes_read] = '\0';
174         std::string input(buffer, bytes_read);
175         std::vector<std::string> result;
176         std::string response;
177
178         parse_resp(input, result);
179         handle_command(result, response);
180
181         send(sock_fd, response.c_str(), response.length(), 0);
182     }
183     else
184     {
185         // std::cout << tag << "Client " << sock_fd << " disconnected." << std::endl;
186         epoll_ctl(epoll_fd, EPOLL_CTL_DEL, sock_fd, nullptr);
187         close(sock_fd);
188     }
189 }
190 }
191 }
192
193 close(server_fd);
194 close(epoll_fd);
195 }

```

Here is the call graph for this function:



Here is the caller graph for this function:



### 3.8.3.4 parse\_resp()

```
void Server::parse_resp (
    const std::string & input,
    std::vector< std::string > & result )
```

Parses a RESP (Redis Serialization Protocol) formatted string.

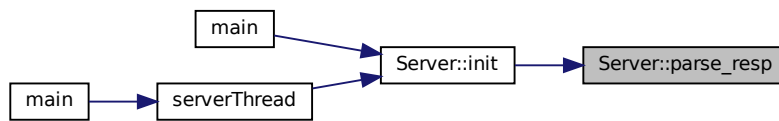
#### Parameters

<i>input</i>	Input string in RESP format.
<i>result</i>	Vector to store parsed tokens.

```

198 {
199     if (input.empty())
200         return;
201
202     if (input[0] == '*')
203     {
204         size_t pos = 1;
205         size_t newline = input.find("\r\n", pos);
206         int array_len = std::stoi(input.substr(pos, newline - pos));
207
208         pos = newline + 2;
209         for (int i = 0; i < array_len; i++)
210         {
211             if (pos >= input.length())
212                 break;
213
214             if (input[pos] == '$')
215             {
216                 pos++;
217                 newline = input.find("\r\n", pos);
218                 int str_len = std::stoi(input.substr(pos, newline - pos));
219
220                 pos = newline + 2;
221                 result.push_back(input.substr(pos, str_len));
222                 pos += str_len + 2;
223             }
224         }
225     }
226 }
```

Here is the caller graph for this function:



The documentation for this class was generated from the following file:

- [lib/server.h](#)

## 3.9 ServerAdd Struct Reference

```
#include <types.h>
```

### Public Attributes

- `int` [port](#)
- `std::string` [ip](#)

### 3.9.1 Member Data Documentation

#### 3.9.1.1 ip

```
std::string ServerAdd::ip
```

#### 3.9.1.2 port

```
int ServerAdd::port
```

The documentation for this struct was generated from the following file:

- [lib/types.h](#)

## 3.10 Trie Class Reference

```
#include <tire.h>
```

### Public Member Functions

- [Trie](#) ()
- [~Trie](#) ()
- void [insert](#) (const std::string &key, long offset)
- long [search](#) (const std::string &key)
- void [remove](#) (const std::string &key)
- bool [isDeleted](#) (const std::string &key)

### 3.10.1 Constructor & Destructor Documentation

#### 3.10.1.1 Trie()

```
Trie::Trie ( ) [inline]  
22 { root = new TrieNode(); }
```

#### 3.10.1.2 ~Trie()

```
Trie::~Trie ( ) [inline]  
23 { deleteTrie(root); }
```

### 3.10.2 Member Function Documentation



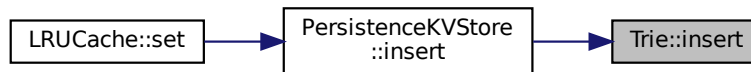
### 3.10.2.1 insert()

```

void Trie::insert (
    const std::string & key,
    long offset ) [inline]
26 {
27     TrieNode *node = root;
28     for (char ch : key)
29     {
30         if (!node->children.count(ch))
31             node->children[ch] = new TrieNode();
32         node = node->children[ch];
33     }
34     node->file_offset = offset;
35     node->isDeleted = false;
36 }

```

Here is the caller graph for this function:



### 3.10.2.2 isDeleted()

```

bool Trie::isDeleted (
    const std::string & key ) [inline]
63 {
64     TrieNode *node = root;
65     for (char ch : key)
66     {
67         if (!node->children.count(ch))
68             return false;
69         node = node->children[ch];
70     }
71     return node->isDeleted;
72 }

```

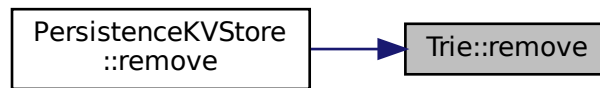
### 3.10.2.3 remove()

```

void Trie::remove (
    const std::string & key ) [inline]
51 {
52     TrieNode *node = root;
53     for (char ch : key)
54     {
55         if (!node->children.count(ch))
56             return;
57         node = node->children[ch];
58     }
59     node->isDeleted = true;
60 }

```

Here is the caller graph for this function:

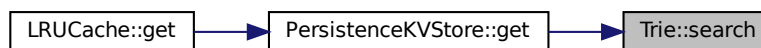


### 3.10.2.4 search()

```

long Trie::search (
    const std::string & key ) [inline]
39     {
40         TrieNode *node = root;
41         for (char ch : key)
42         {
43             if (!node->children.count(ch))
44                 return -1;
45             node = node->children[ch];
46         }
47         return node->isDeleted ? -1 : node->file_offset;
48     }
  
```

Here is the caller graph for this function:



The documentation for this class was generated from the following file:

- [lib/tire.h](#)

## 3.11 TrieNode Class Reference

```
#include <tire.h>
```

### Public Attributes

- `std::unordered_map< char, TrieNode * >` `children`
- `long` `file_offset` = -1
- `bool` `isDeleted` = false

### 3.11.1 Member Data Documentation

#### 3.11.1.1 children

```
std::unordered_map<char, TrieNode *> TrieNode::children
```

#### 3.11.1.2 file\_offset

```
long TrieNode::file_offset = -1
```

#### 3.11.1.3 isDeleted

```
bool TrieNode::isDeleted = false
```

The documentation for this class was generated from the following file:

- [lib/tire.h](#)



## Chapter 4

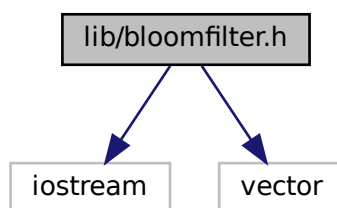
# File Documentation

### 4.1 lib/bloomfilter.h File Reference

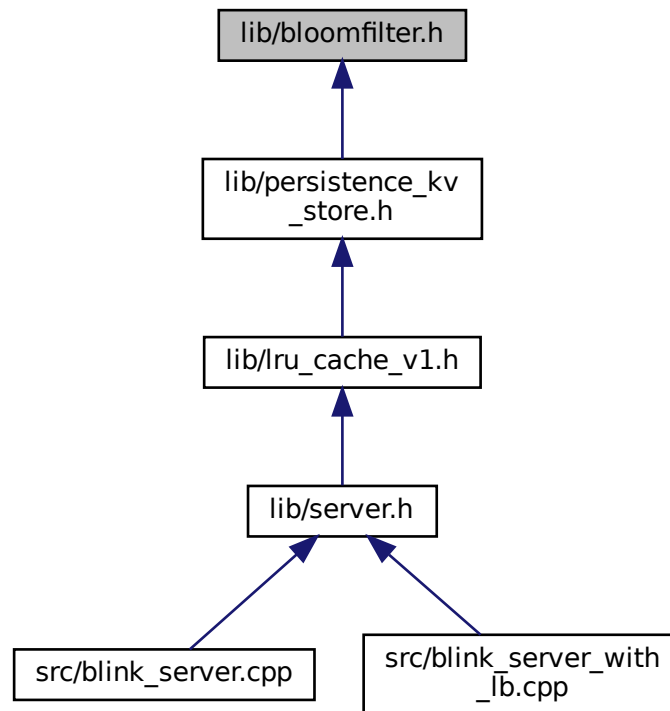
```
#include <iostream>
```

```
#include <vector>
```

Include dependency graph for bloomfilter.h:



This graph shows which files directly or indirectly include this file:



## Classes

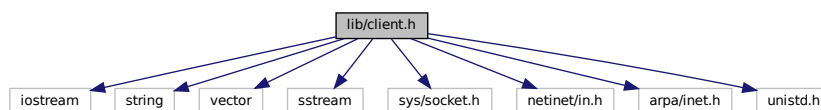
- class [BloomFilter](#)

*Implements a simple Bloom filter for fast key existence checks.*

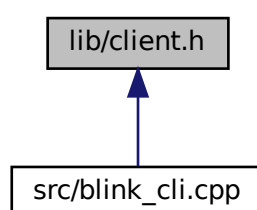
## 4.2 lib/client.h File Reference

```
#include <iostream>
#include <string>
#include <vector>
#include <sstream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
```

Include dependency graph for client.h:



This graph shows which files directly or indirectly include this file:



## Classes

- class [Client](#)

*A class to interact with a Blink server using RESP (Blink Serialization Protocol).*

## 4.3 lib/dict.h File Reference

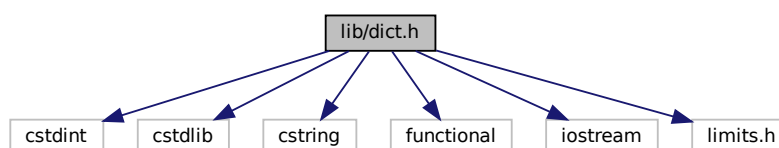
Implementation of a dictionary (hash table) with rehashing support.

```

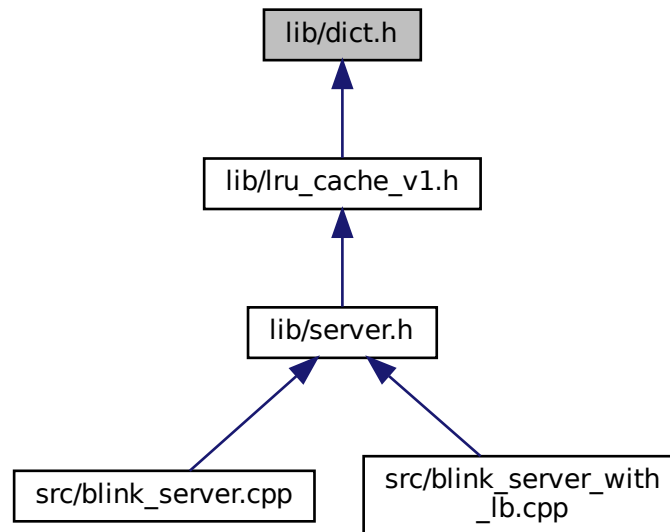
#include <cstdint>
#include <cstdlib>
#include <cstring>
#include <functional>
#include <iostream>
#include <limits.h>

```

Include dependency graph for dict.h:



This graph shows which files directly or indirectly include this file:



## Classes

- class [Dict](#)  
*A dictionary (hash table) implementation with dynamic resizing and rehashing.*

## Functions

- unsigned int [stringHash](#) (const void \*key)  
*Hash function for C-style strings.*
- int [stringCompare](#) (const void \*key1, const void \*key2)  
*Compares two C-style string keys.*
- void \* [stringDup](#) (const void \*key)  
*Duplicates a C-style string key.*
- void [freeString](#) (void \*ptr)  
*Frees a dynamically allocated C-style string.*

### 4.3.1 Detailed Description

Implementation of a dictionary (hash table) with rehashing support.

### 4.3.2 Function Documentation



#### 4.3.2.1 freeString()

```
void freeString (
    void * ptr )
```

Frees a dynamically allocated C-style string.

This function releases the memory allocated for a string key or value.

##### Parameters

<i>ptr</i>	Pointer to the string to be freed.
------------	------------------------------------

```
648                                     {
649     free(ptr);
650 }
```

#### 4.3.2.2 stringCompare()

```
int stringCompare (
    const void * key1,
    const void * key2 )
```

Compares two C-style string keys.

This function compares two string keys using `strcmp` and returns whether they are equal.

##### Parameters

<i>key1</i>	Pointer to the first string key.
<i>key2</i>	Pointer to the second string key.

##### Returns

int Returns 1 if keys are equal, 0 otherwise.

```
623 {
624     return strcmp(static_cast<const char *>(key1), static_cast<const char *>(key2)) == 0;
625 }
```

#### 4.3.2.3 stringDup()

```
void* stringDup (
    const void * key )
```

Duplicates a C-style string key.

This function creates a copy of the given string key using `strdup`. The caller is responsible for freeing the allocated memory.

**Parameters**

<i>key</i>	Pointer to the original string key.
------------	-------------------------------------

**Returns**

`void*` Pointer to the duplicated string.

```

637 {
638     return strdup(static_cast<const char *>(key));
639 }
```

**4.3.2.4 stringHash()**

```

unsigned int stringHash (
    const void * key )
```

Hash function for C-style strings.

This function computes a hash value for a given string using the DJB2 algorithm. It iterates through the characters and accumulates a hash value.

**Parameters**

<i>key</i>	Pointer to the C-style string key.
------------	------------------------------------

**Returns**

`unsigned int` The computed hash value.

```

603 {
604     const char *str = static_cast<const char *>(key);
605     unsigned int hash = 5381;
606     int c;
607     while ((c = *str++))
608         hash = ((hash << 5) + hash) + c; // hash * 33 + c
609     return hash;
610 }
```

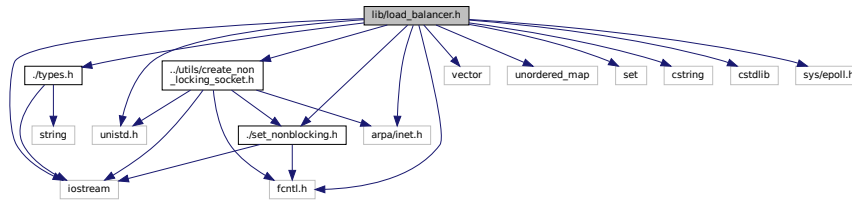
**4.4 lib/load\_balancer.h File Reference**

Implements a non-blocking load balancer using epoll and consistent hashing.

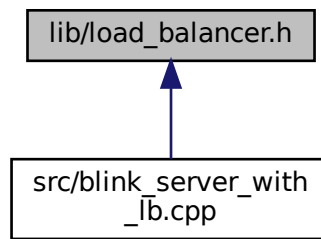
```

#include <iostream>
#include <vector>
#include <unordered_map>
#include <set>
#include <cstring>
#include <cstdlib>
#include <unistd.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <sys/epoll.h>
```

```
#include "../utils/create_non_locking_socket.h"
#include "../utils/set_nonblocking.h"
#include "../types.h"
Include dependency graph for load_balancer.h:
```



This graph shows which files directly or indirectly include this file:



## Classes

- class [LoadBalancer](#)  
*Handles load balancing by distributing client requests to backend servers.*

### 4.4.1 Detailed Description

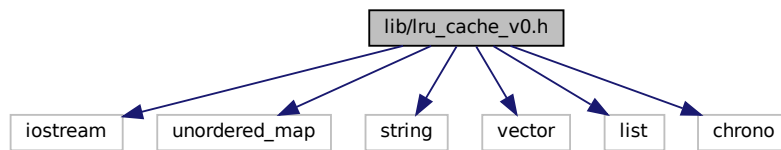
Implements a non-blocking load balancer using epoll and consistent hashing.

## 4.5 lib/lru\_cache\_v0.h File Reference

```
#include <iostream>
#include <unordered_map>
#include <string>
#include <vector>
#include <list>
```

```
#include <chrono>
```

Include dependency graph for lru\_cache\_v0.h:



## Classes

- class [LRUCache](#)

*Implements a Least Recently Used (LRU) cache with memory constraints.*

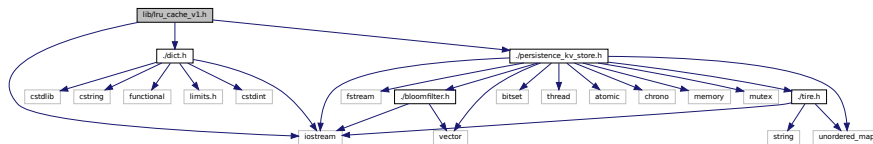
## 4.6 lib/lru\_cache\_v1.h File Reference

```
#include <iostream>
```

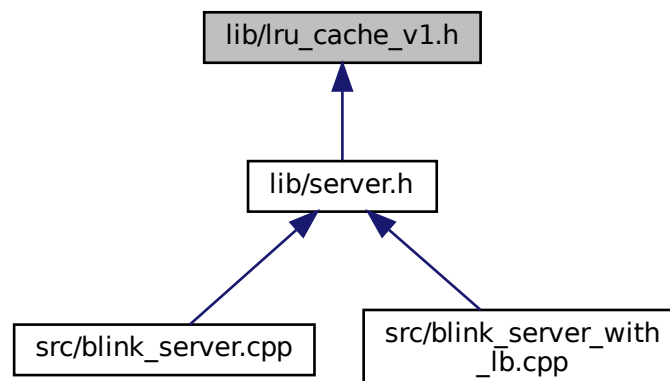
```
#include "../dict.h"
```

```
#include "../persistence_kv_store.h"
```

Include dependency graph for lru\_cache\_v1.h:



This graph shows which files directly or indirectly include this file:

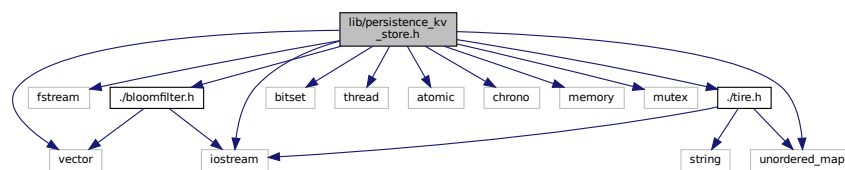


## Classes

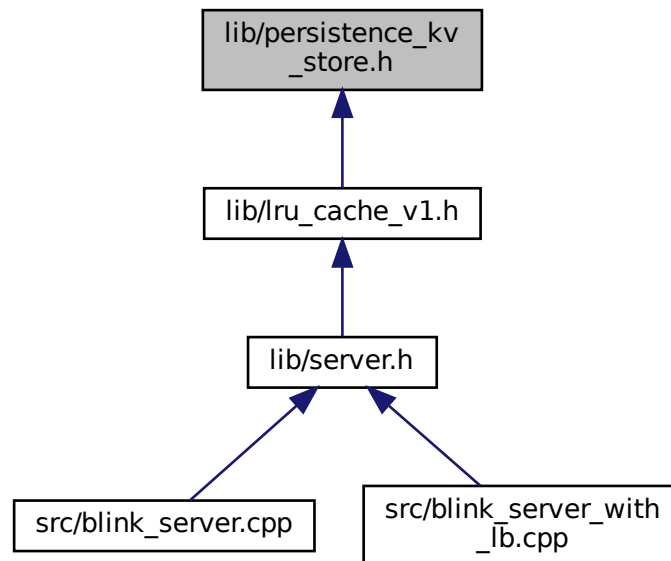
- class [LRUCache](#)  
*Implements a Least Recently Used (LRU) cache with memory constraints.*
- struct [LRUCache::Node](#)  
*Represents a node in the doubly linked list for the LRU cache.*

## 4.7 lib/persistence\_kv\_store.h File Reference

```
#include <iostream>
#include <fstream>
#include <unordered_map>
#include <vector>
#include <bitset>
#include <thread>
#include <atomic>
#include <chrono>
#include <memory>
#include <mutex>
#include "../bloomfilter.h"
#include "../tire.h"
Include dependency graph for persistence_kv_store.h:
```



This graph shows which files directly or indirectly include this file:



## Classes

- class [PersistenceKVStore](#)

*A persistent key-value store with background rewriting and indexing.*

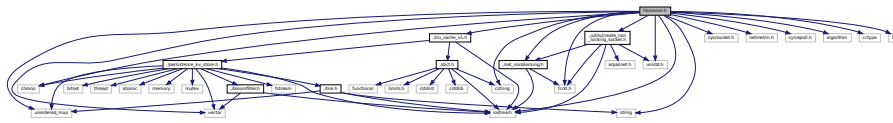
## 4.8 lib/server.h File Reference

```

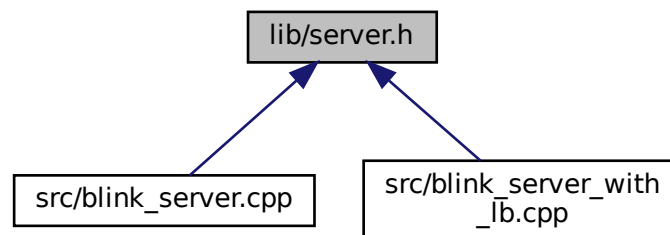
#include <iostream>
#include <cstring>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/epoll.h>
#include <fcntl.h>
#include <unordered_map>
#include <string>
#include <vector>
#include <algorithm>
#include <cctype>
#include <list>
#include <chrono>
#include "../lru_cache_v1.h"
#include "../../utils/create_non_locking_socket.h"

```

```
#include "../utils/set_nonblocking.h"
Include dependency graph for server.h:
```



This graph shows which files directly or indirectly include this file:



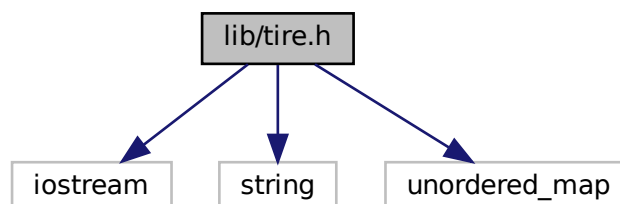
## Classes

- class **Server**

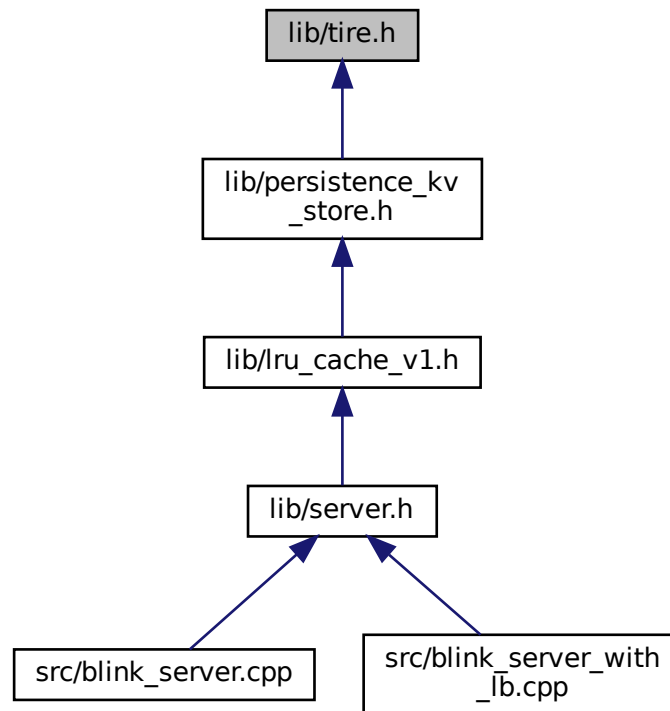
*Implements a Blink-compatible server with an LRU-based in-memory database.*

## 4.9 lib/tire.h File Reference

```
#include <iostream>
#include <string>
#include <unordered_map>
Include dependency graph for tire.h:
```



This graph shows which files directly or indirectly include this file:



## Classes

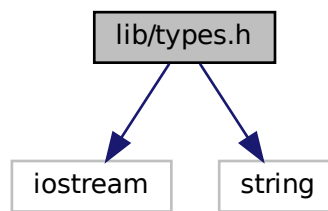
- class [TrieNode](#)
- class [Trie](#)

## 4.10 lib/types.h File Reference

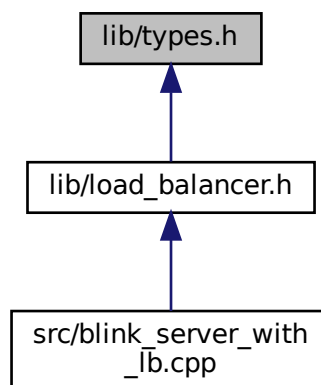
```
#include <iostream>
#include <string>
```



Include dependency graph for types.h:



This graph shows which files directly or indirectly include this file:



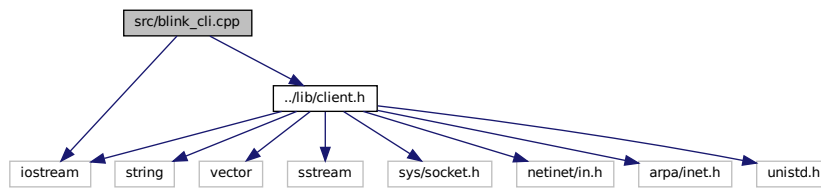
## Classes

- struct [ServerAdd](#)

## 4.11 src/blink\_cli.cpp File Reference

```
#include <iostream>
#include "../lib/client.h"
```

Include dependency graph for blink\_cli.cpp:



## Macros

- `#define BUFFER_SIZE 1024`
- `#define SERVER_PORT 9001`
- `#define SERVER_IP "127.0.0.1"`

## Functions

- `void command_loop (Client &client)`  
*Reads user input, parses commands, and interacts with the [Client](#) class.*
- `int main ()`

### 4.11.1 Macro Definition Documentation

#### 4.11.1.1 BUFFER\_SIZE

```
#define BUFFER_SIZE 1024
```

#### 4.11.1.2 SERVER\_IP

```
#define SERVER_IP "127.0.0.1"
```

#### 4.11.1.3 SERVER\_PORT

```
#define SERVER_PORT 9001
```

### 4.11.2 Function Documentation

#### 4.11.2.1 command\_loop()

```
void command_loop (
    Client & client )
```

Reads user input, parses commands, and interacts with the [Client](#) class.

## Parameters

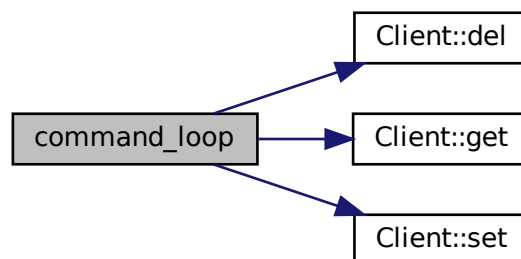
<i>client</i>	A reference to the <a href="#">Client</a> object.
---------------	---

```

13 {
14     std::cout << "Server is connected at " << client.ip_addr << ":" << client.port << std::endl;
15     std::cout << "Enter command (SET key value, GET key, or EXIT to quit):" << std::endl;
16
17     std::string input, command, key, value;
18     while (true)
19     {
20         std::cout << "> ";
21         std::getline(std::cin, input);
22
23         std::istringstream iss(input);
24         iss >> command;
25         if (command == "SET")
26         {
27             iss >> key >> value;
28             if (key.empty() || value.empty())
29             {
30                 std::cout << "Invalid SET command. Usage: SET <key> <value>" << std::endl;
31                 continue;
32             }
33             std::cout << client.set(key, value) << std::endl;
34         }
35         else if (command == "GET")
36         {
37             iss >> key;
38             if (key.empty())
39             {
40                 std::cout << "Invalid GET command. Usage: GET <key>" << std::endl;
41                 continue;
42             }
43             std::cout << client.get(key) << std::endl;
44         }
45         else if (command == "DEL")
46         {
47             iss >> key;
48             if (key.empty())
49             {
50                 std::cout << "Invalid DEL command. Usage: DEL <key>" << std::endl;
51                 continue;
52             }
53             std::cout << client.del(key) << std::endl;
54         }
55         else if (command == "EXIT")
56         {
57             break;
58         }
59         else
60         {
61             std::cout << "Unknown command. Use SET, GET, DEL, or EXIT." << std::endl;
62         }
63     }
64 }

```

Here is the call graph for this function:



Here is the caller graph for this function:

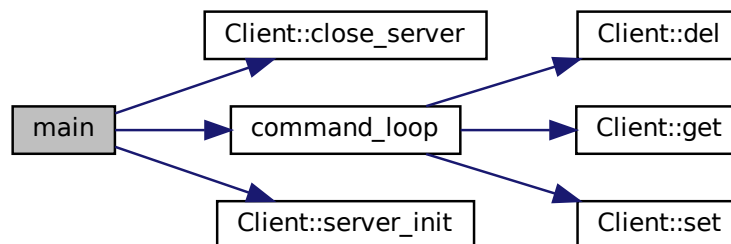


#### 4.11.2.2 main()

```

int main ( )
67 {
68     Client client(SERVER_IP, SERVER_PORT, BUFFER_SIZE);
69
70     if (client.server_init() == -1)
71     {
72         return -1;
73     }
74
75     command_loop(client);
76
77     client.close_server();
78
79     return 0;
80 }
  
```

Here is the call graph for this function:



## 4.12 src/blink\_server.cpp File Reference

```

#include <iostream>
#include <string>
  
```



**Returns**

int Returns 0 on successful execution.

Create a [Server](#) instance with configuration parameters

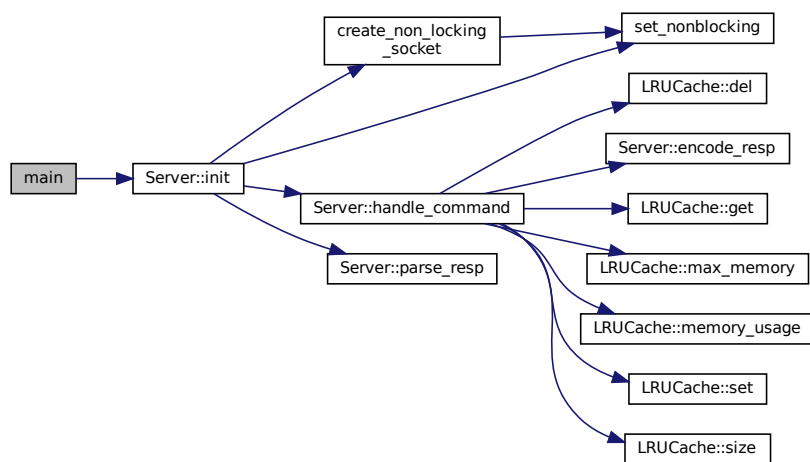
**Initialize the server**

```

23  {
25      Server server(SERVER_IP, SERVER_PORT, BUFFER_SIZE, MAX_EVENTS, MAX_MEMORY_BYTES);
26
28      server.init();
29
30      return 0;
31  }

```

Here is the call graph for this function:

**4.12.2 Variable Documentation****4.12.2.1 BUFFER\_SIZE**

```
const int BUFFER_SIZE = 2048
```

Buffer size for reading data.

**4.12.2.2 MAX\_EVENTS**

```
const int MAX_EVENTS = 4096
```

Maximum number of events for epoll.



## Functions

- void `serverThread` (std::string ip, int port)
- std::string `parse_key` (char \*buffer, int bytes\_read)
- int `main` ()

## 4.13.1 Macro Definition Documentation

### 4.13.1.1 BUFFER\_SIZE

```
#define BUFFER_SIZE 1024
```

### 4.13.1.2 MAX\_EVENTS

```
#define MAX_EVENTS 100
```

### 4.13.1.3 MAX\_MEMORY\_BYTES

```
#define MAX_MEMORY_BYTES 1024 * 1024 * 10
```

### 4.13.1.4 SERVER\_IP

```
#define SERVER_IP "127.0.0.1"
```

### 4.13.1.5 SERVER\_PORT

```
#define SERVER_PORT 9001
```

## 4.13.2 Function Documentation



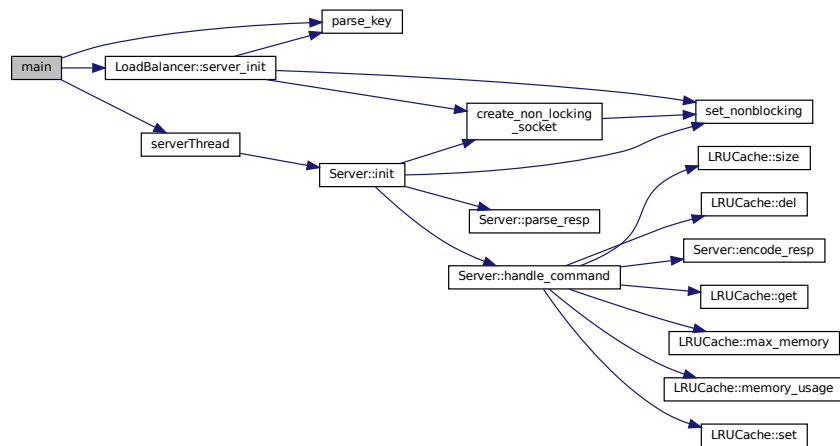
## 4.13.2.1 main()

```

int main ( )
64 {
65
66     int num_servers;
67     std::cout << "Enter number of servers: ";
68     std::cin >> num_servers;
69
70     std::vector<ServerAdd> servers_addr;
71     for (int i = 0; i < num_servers; ++i) {
72         ServerAdd server_add;
73         server_add.port = 5000 + i;
74         server_add.ip = SERVER_IP;
75         servers_addr.push_back(server_add);
76     }
77
78     std::vector<std::thread> server_threads;
79     for (auto &server_addr : servers_addr) {
80         server_threads.emplace_back(serverThread, server_addr.ip, server_addr.port);
81     }
82
83     LoadBalancer loadBalancer(SERVER_IP, SERVER_PORT, servers_addr, BUFFER_SIZE, MAX_EVENTS);
84
85     loadBalancer.server_init(&parse_key);
86
87     for (auto& t : server_threads) t.join();
88
89     return 0;
90 }

```

Here is the call graph for this function:



## 4.13.2.2 parse\_key()

```

std::string parse_key (
    char * buffer,
    int bytes_read )
28 {
29
30     std::string input(buffer, bytes_read);
31     std::vector<std::string> result;
32
33     if (input.empty())
34         return "";
35

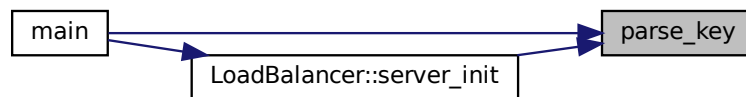
```

```

36     if (input[0] == '*')
37     {
38         size_t pos = 1;
39         size_t newline = input.find("\r\n", pos);
40         int array_len = std::stoi(input.substr(pos, newline - pos));
41
42         pos = newline + 2;
43         for (int i = 0; i < array_len; i++)
44         {
45             if (pos >= input.length())
46                 break;
47
48             if (input[pos] == '$')
49             {
50                 pos++;
51                 newline = input.find("\r\n", pos);
52                 int str_len = std::stoi(input.substr(pos, newline - pos));
53
54                 pos = newline + 2;
55                 result.push_back(input.substr(pos, str_len));
56                 pos += str_len + 2;
57             }
58         }
59     }
60
61     return result[1];
62 }

```

Here is the caller graph for this function:



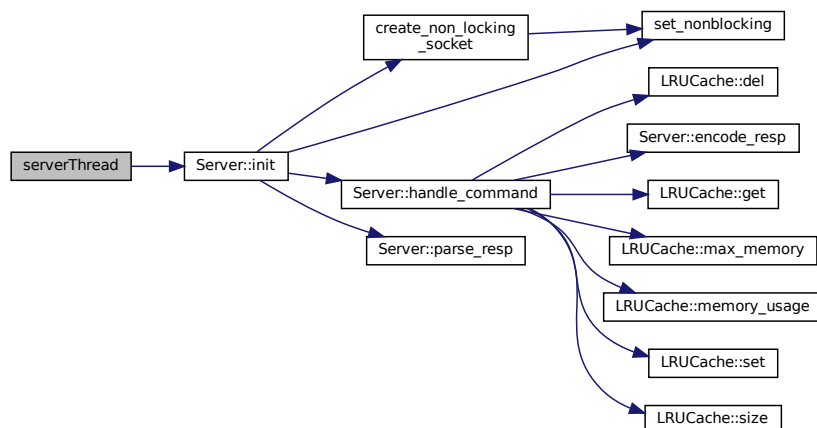
#### 4.13.2.3 serverThread()

```

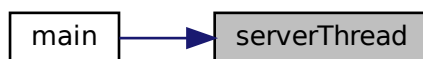
void serverThread (
    std::string ip,
    int port )
23     {
24         Server server(ip, port, BUFFER_SIZE, MAX_EVENTS, MAX_MEMORY_BYTES);
25         server.init();
26     }

```

Here is the call graph for this function:



Here is the caller graph for this function:



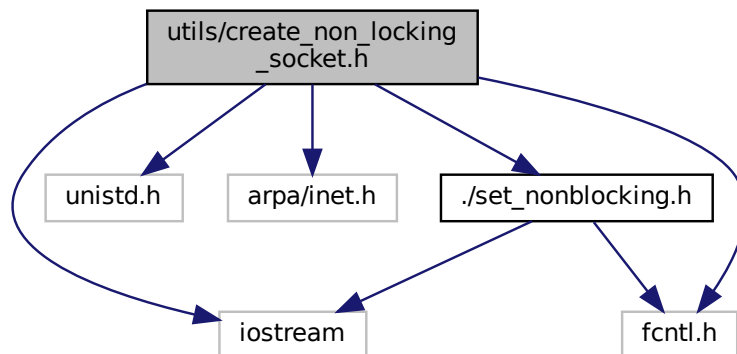
## 4.14 utils/create\_non\_locking\_socket.h File Reference

```

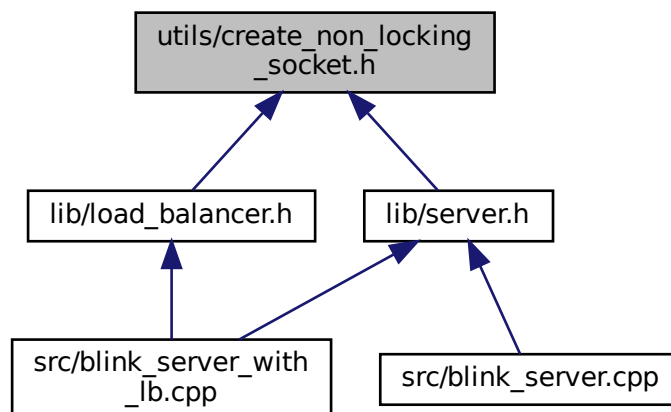
#include <iostream>
#include <unistd.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include "../set_nonblocking.h"

```

Include dependency graph for `create_non_locking_socket.h`:



This graph shows which files directly or indirectly include this file:



## Functions

- `int create_non_locking_socket` (const std::string ip, const int port, struct sockaddr\_in &addr)  
*Creates a non-blocking socket.*

### 4.14.1 Function Documentation

## 4.14.1.1 create\_non\_locking\_socket()

```
int create_non_locking_socket (
    const std::string ip,
    const int port,
    struct sockaddr_in & addr )
```

Creates a non-blocking socket.

## Parameters

<i>ip</i>	The ip to bind the socket to.
<i>port</i>	The port number to bind the socket to.
<i>addr</i>	The addr to save the socket address.

## Returns

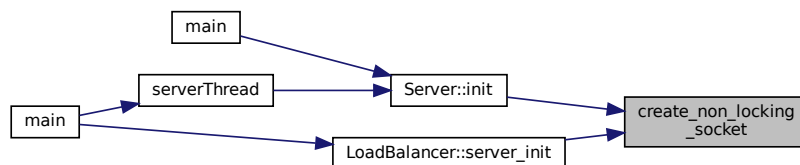
The socket file descriptor.

```
19 {
20     int sockfd = socket(AF_INET, SOCK_STREAM, 0);
21     int opt = 1;
22     setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
23
24     if (sockfd < 0)
25     {
26         perror("Socket creation failed");
27         exit(EXIT_FAILURE);
28     }
29
30     set_nonblocking(sockfd);
31
32     addr.sin_family = AF_INET;
33     addr.sin_port = htons(port);
34
35     if (inet_pton(AF_INET, ip.c_str(), &addr.sin_addr) <= 0) {
36         perror("Invalid IP address");
37         exit(EXIT_FAILURE);
38     }
39
40     if (bind(sockfd, (struct sockaddr *)&addr, sizeof(addr)) < 0)
41     {
42         perror("Bind failed");
43         exit(EXIT_FAILURE);
44     }
45
46     if (listen(sockfd, SOMAXCONN) < 0)
47     {
48         perror("Listen failed");
49         exit(EXIT_FAILURE);
50     }
51
52     return sockfd;
53 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

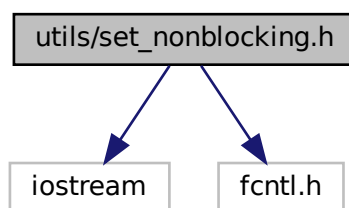


## 4.15 utils/set\_nonblocking.h File Reference

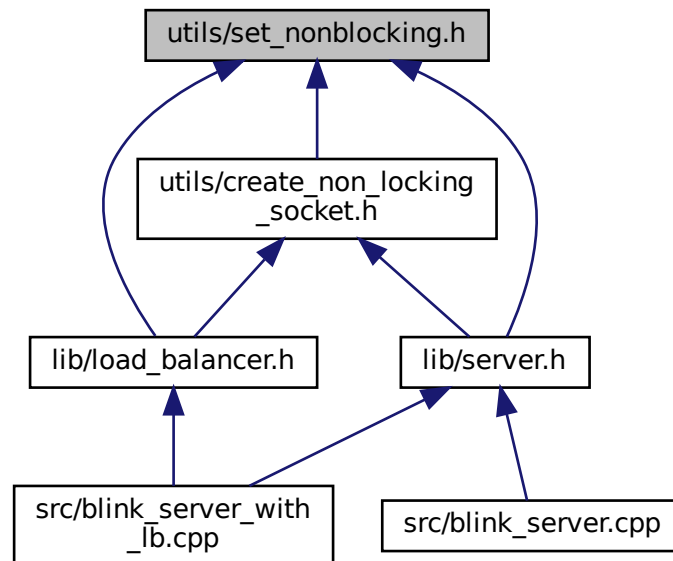
```
#include <iostream>
```

```
#include <fcntl.h>
```

Include dependency graph for `set_nonblocking.h`:



This graph shows which files directly or indirectly include this file:



## Functions

- void [set\\_nonblocking](#) (int sock)

### 4.15.1 Function Documentation

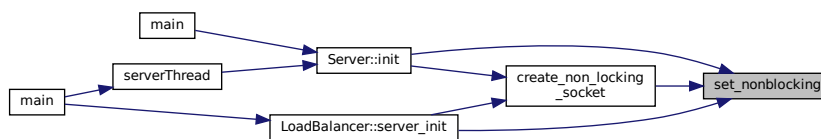
#### 4.15.1.1 set\_nonblocking()

```

void set_nonblocking (
    int sock )
9 {
10     int flags = fcntl(sock, F_GETFL, 0);
11     if (flags == -1)
12     {
13         perror("fcntl F_GETFL failed");
14         exit(EXIT_FAILURE);
15     }
16
17     if (fcntl(sock, F_SETFL, flags | O_NONBLOCK) == -1)
18     {
19         perror("fcntl F_SETFL failed");
20         exit(EXIT_FAILURE);
21     }
22 }

```

Here is the caller graph for this function:





# Index

- ~Client
  - Client, [9](#)
- ~Dict
  - Dict, [14](#)
- ~LRUCache
  - LRUCache, [25](#)
- ~LoadBalancer
  - LoadBalancer, [20](#)
- ~PersistenceKVStore
  - PersistenceKVStore, [36](#)
- ~Server
  - Server, [41](#)
- ~Trie
  - Trie, [48](#)
- add
  - Dict, [14](#)
- blink\_cli.cpp
  - BUFFER\_SIZE, [66](#)
  - command\_loop, [66](#)
  - main, [68](#)
  - SERVER\_IP, [66](#)
  - SERVER\_PORT, [66](#)
- blink\_server.cpp
  - BUFFER\_SIZE, [70](#)
  - main, [69](#)
  - MAX\_EVENTS, [70](#)
  - MAX\_MEMORY\_BYTES, [70](#)
  - SERVER\_IP, [71](#)
  - SERVER\_PORT, [71](#)
- blink\_server\_with\_lb.cpp
  - BUFFER\_SIZE, [72](#)
  - main, [72](#)
  - MAX\_EVENTS, [72](#)
  - MAX\_MEMORY\_BYTES, [72](#)
  - parse\_key, [73](#)
  - SERVER\_IP, [72](#)
  - SERVER\_PORT, [72](#)
  - serverThread, [74](#)
- BloomFilter, [5](#)
  - BloomFilter, [5](#)
  - contains, [6](#)
  - insert, [6](#)
  - remove, [7](#)
- buffer
  - Client, [12](#)
- BUFFER\_SIZE
  - blink\_cli.cpp, [66](#)
  - blink\_server.cpp, [70](#)
  - blink\_server\_with\_lb.cpp, [72](#)
- buffer\_size
  - Client, [12](#)
- children
  - TrieNode, [51](#)
- Client, [7](#)
  - ~Client, [9](#)
  - buffer, [12](#)
  - buffer\_size, [12](#)
  - Client, [8](#)
  - close\_server, [9](#)
  - del, [9](#)
  - get, [10](#)
  - ip\_addr, [12](#)
  - port, [12](#)
  - server\_init, [11](#)
  - set, [11](#)
- close\_server
  - Client, [9](#)
- command\_loop
  - blink\_cli.cpp, [66](#)
- contains
  - BloomFilter, [6](#)
- create\_non\_locking\_socket
  - create\_non\_locking\_socket.h, [76](#)
- create\_non\_locking\_socket.h
  - create\_non\_locking\_socket, [76](#)
- del
  - Client, [9](#)
  - LRUCache, [25](#)
- Dict, [13](#)
  - ~Dict, [14](#)
  - add, [14](#)
  - Dict, [14](#)
  - enableResize, [15](#)
  - find, [15](#)
  - get\_size\_of\_dict, [16](#)
  - isRehashing, [16](#)
  - rehash, [17](#)
  - remove, [17](#)
  - replace, [18](#)
  - size, [18](#)
- dict
  - LRUCache, [32](#)
- dict.h
  - freeString, [56](#)
  - stringCompare, [57](#)
  - stringDup, [57](#)

- stringHash, 58
- enableResize
  - Dict, 15
- encode\_resp
  - Server, 41
- file\_offset
  - TrieNode, 51
- find
  - Dict, 15
- freeString
  - dict.h, 56
- get
  - Client, 10
  - LRUCache, 26, 27
  - PersistenceKVStore, 36
- get\_size\_of\_dict
  - Dict, 16
- handle\_command
  - Server, 41
- head
  - LRUCache, 32
- init
  - Server, 44
- insert
  - BloomFilter, 6
  - PersistenceKVStore, 37
  - Trie, 48
- ip
  - ServerAdd, 47
- ip\_addr
  - Client, 12
- isDeleted
  - Trie, 49
  - TrieNode, 51
- isRehashing
  - Dict, 16
- key
  - LRUCache::Node, 34
- lib/bloomfilter.h, 53
- lib/client.h, 54
- lib/dict.h, 55
- lib/load\_balancer.h, 58
- lib/lru\_cache\_v0.h, 59
- lib/lru\_cache\_v1.h, 60
- lib/persistence\_kv\_store.h, 61
- lib/server.h, 62
- lib/tire.h, 63
- lib/types.h, 64
- LoadBalancer, 19
  - ~LoadBalancer, 20
  - LoadBalancer, 20
  - server\_init, 20
- LRUCache, 23
  - ~LRUCache, 25
  - del, 25
  - dict, 32
  - get, 26, 27
  - head, 32
  - LRUCache, 24
  - max\_memory, 28
  - memory\_usage, 28, 29
  - printList, 29
  - set, 29, 30
  - size, 31, 32
  - storage, 32
  - tail, 33
  - value, 33
- LRUCache::Node, 33
  - key, 34
  - next, 34
  - Node, 34
  - prev, 34
  - value, 34
- main
  - blink\_cli.cpp, 68
  - blink\_server.cpp, 69
  - blink\_server\_with\_lb.cpp, 72
- MAX\_EVENTS
  - blink\_server.cpp, 70
  - blink\_server\_with\_lb.cpp, 72
- max\_memory
  - LRUCache, 28
- MAX\_MEMORY\_BYTES
  - blink\_server.cpp, 70
  - blink\_server\_with\_lb.cpp, 72
- memory\_usage
  - LRUCache, 28, 29
- next
  - LRUCache::Node, 34
- Node
  - LRUCache::Node, 34
- parse\_key
  - blink\_server\_with\_lb.cpp, 73
- parse\_resp
  - Server, 46
- PersistenceKVStore, 35
  - ~PersistenceKVStore, 36
  - get, 36
  - insert, 37
  - PersistenceKVStore, 35
  - remove, 38
  - remove\_db, 39
- port
  - Client, 12
  - ServerAdd, 47
- prev
  - LRUCache::Node, 34
- printList
  - LRUCache, 29

- rehash
  - Dict, [17](#)
- remove
  - BloomFilter, [7](#)
  - Dict, [17](#)
  - PersistenceKVStore, [38](#)
  - Trie, [49](#)
- remove\_db
  - PersistenceKVStore, [39](#)
- replace
  - Dict, [18](#)
- search
  - Trie, [50](#)
- Server, [39](#)
  - ~Server, [41](#)
  - encode\_resp, [41](#)
  - handle\_command, [41](#)
  - init, [44](#)
  - parse\_resp, [46](#)
  - Server, [40](#)
- server\_init
  - Client, [11](#)
  - LoadBalancer, [20](#)
- SERVER\_IP
  - blink\_cli.cpp, [66](#)
  - blink\_server.cpp, [71](#)
  - blink\_server\_with\_lb.cpp, [72](#)
- SERVER\_PORT
  - blink\_cli.cpp, [66](#)
  - blink\_server.cpp, [71](#)
  - blink\_server\_with\_lb.cpp, [72](#)
- ServerAdd, [47](#)
  - ip, [47](#)
  - port, [47](#)
- serverThread
  - blink\_server\_with\_lb.cpp, [74](#)
- set
  - Client, [11](#)
  - LRUCache, [29](#), [30](#)
- set\_nonblocking
  - set\_nonblocking.h, [79](#)
- set\_nonblocking.h
  - set\_nonblocking, [79](#)
- size
  - Dict, [18](#)
  - LRUCache, [31](#), [32](#)
- src/blink\_cli.cpp, [65](#)
- src/blink\_server.cpp, [68](#)
- src/blink\_server\_with\_lb.cpp, [71](#)
- storage
  - LRUCache, [32](#)
- stringCompare
  - dict.h, [57](#)
- stringDup
  - dict.h, [57](#)
- stringHash
  - dict.h, [58](#)
- tail
  - LRUCache, [33](#)
- Trie, [48](#)
  - ~Trie, [48](#)
  - insert, [48](#)
  - isDeleted, [49](#)
  - remove, [49](#)
  - search, [50](#)
  - Trie, [48](#)
- TrieNode, [50](#)
  - children, [51](#)
  - file\_offset, [51](#)
  - isDeleted, [51](#)
- utils/create\_non\_locking\_socket.h, [75](#)
- utils/set\_nonblocking.h, [78](#)
- value
  - LRUCache, [33](#)
  - LRUCache::Node, [34](#)