



University of Pittsburgh

Department of Computer Science

Project report

CS 2510 Computer Operating Systems

December 2015

Mini Google

Mohammad Hasanazadeh Mofrad

hasanzadeh@cs.pitt.edu

Abstract

This report presents mini Google which is a full stack implementation of a search engine. Mini Google is built on top of an inverted index which is the central data structure of a search engine indexing algorithm. It allows fast full-text searches and retrieves by following the in-memory computing paradigm. Mini Google follows the client/server architecture and keeps the computation in helper nodes while serving trivial index/retrieve operations in master node. It is designed around master/slave architecture to provide high availability feature by failure detection and data replication. Furthermore, it is a multithread implementation of a system software in C which allows concurrent and resilient index/retrieve operations. Finally, mini Google accepts index/retrieve operations via both SSH session and web browser.

Keywords: Information retrieval; Inverted index; Master/slave architecture; Client/server architecture; High availability; Failover; Replication;

1 Introduction

Information Retrieval is the process of gathering relevant information from a collection of resources. This process has been widely adopted in search engines that searches can be done based on content-based indexing. The whole process initiates when a user enters a query into the search engine. User queries are matched against the system information. Depending on the application, data objects are ranked according to one or multiple numeric criteria. Finally, the top ranking objects are shown to the user. If the user wishes to see the content of an object, search engine will retrieve this data for him.

Mini Google is a system software that is designated to search for information on a set of uploaded documents. A search query is initiated by entering a keyword and the search results are generally presented as a ranking order of documents. Furthermore, the user could request for the content of the ranked documents. As shown in Figure 1, the user could connect to Mini Google via a SSH session/web browser to initiate an index/retrieve operation through client shell.

The *index operation* uploads a directory/URL from client shell into the master node. After receiving the new documents, the master will distribute them to helper nodes and eventually gather the indexing results to build an inverted matrix. The *inverted index* is a list of unique words that appear in each document, and for each word, a list of documents with their corresponding frequencies in which the term is appeared.

The *retrieve operation* submits a query for a specific word from the client shell into the master node. The master node checks the inverted matrix and returns an ordered list of documents along with their corresponding word frequencies to the client shell. The client shell presents these results to the user via the SSH session/web browser. Then, the user may submit an additional query for one of the resultant documents by entering its document identifier. Next, the client shell passes the document identifier to the master node and master locates the exact location of the requested document and retrieves its content from the helper that hosts it. Finally, the client shell receives the document content and sends it to the user SSH session/Web browser.

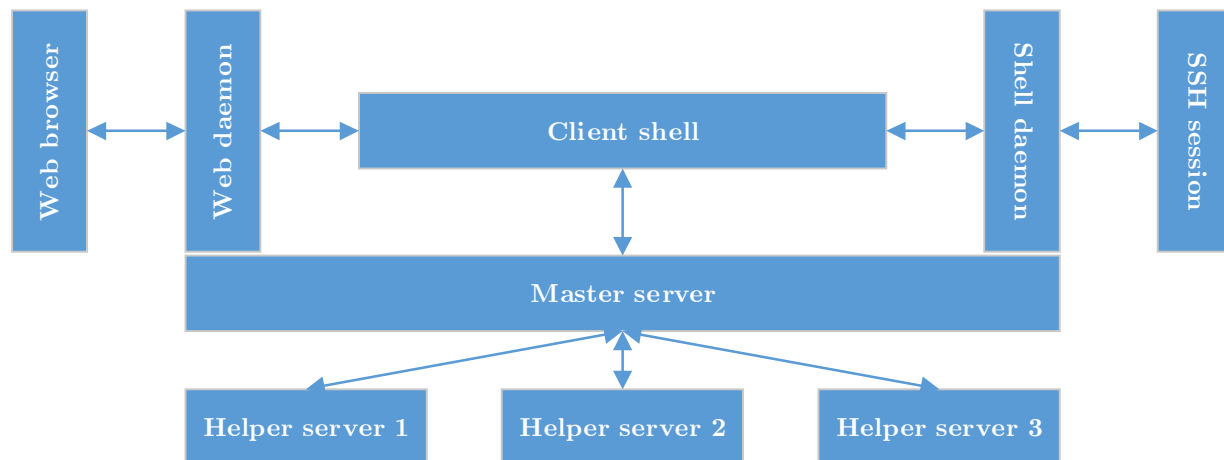


Figure 1 Mini Google Framework.

The rest of this report is organized as follows: Section 2 introduces the mini Google system software, and Section 3 concludes this report.

2 Mini Google

From the system centric view, mini Google consists of four tiers which are shown in Figure 2.

1. Client shell which is the single point of entry of users to submit their index/retrieve queries from SSH session/web browser.
2. Master server which distributes the incoming documents into helper servers and stores the results in a centralized inverted index.
3. Slave server which monitors master server socket connection by sending heartbeat messages. In case of detecting a failure, slave server tries to listen to the IP address and port of master server, accepts replication messages from helper servers, and resumes the mini Google functionalities.
4. Helper servers which process the given documents (one at a time) and send the (unique words, frequency) tuple to the master server. Helpers store the content of incoming documents in memory for future document retrieval. Also, they store a local copy of the computed inverted matrix for plausible high availability scenario.

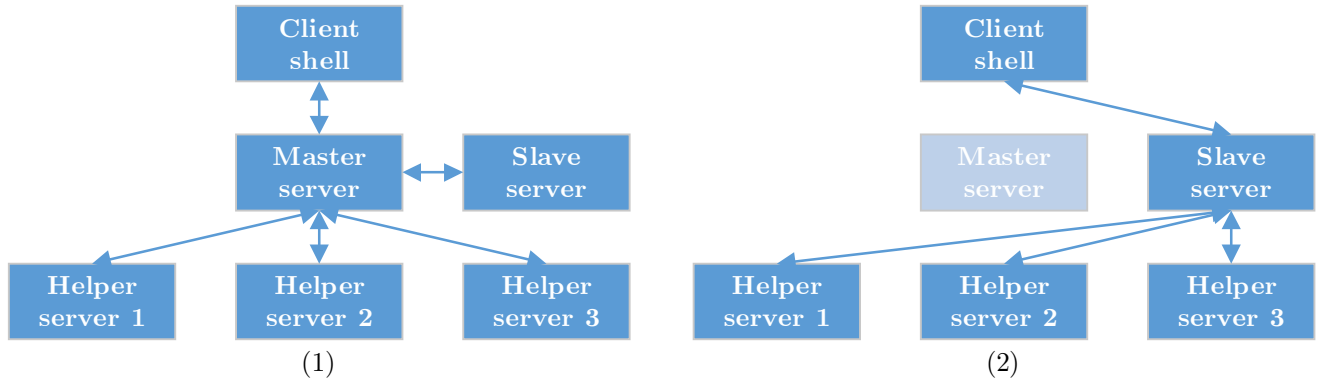


Figure 2 Mini Google four tier architecture. (1) Master server handling all incoming communications, and (2) failover happened and slave server taking the control of incoming communications and accepts replicas from helper nodes.

2.1 Message Passing Protocol

Building a concrete information retrieval platform, mini Google follows a robust and resilient message passing protocol. As shown in Figure 1, there is a sequence of requests/responses for completing the given index/retrieve task. Note that for the sake of simplicity, we only show the corresponding messages of client shell, master server, and one helper server and keep the discussion about the failover and replication messages for Section 2.6.

Figure 3 shows the message codes of mini Google along with their calling components. These message codes are used to establish communication between different tiers of mini Google. Also, Figure 4 sketches the flow of the proposed message passing protocol.

Message	Description
100	1xx: <i>Index</i> : It means an index request is received from a client.
200	2xx: <i>Retrieve</i> : It means a retrieve request is received from a client.
300	3xx: <i>Register</i> : It means a register request is received from a helper server.
400	4xx: <i>Heartbeat</i> : It means a heartbeat request is received from a slave or helper server.
500	5xx: <i>Replication</i> : It means a replication request is received from a helper sever.

Figure 3 Message codes.

Figure 4 – 1 shows the indexing part of the proposed message passing protocol. This part includes the following steps:

1. Master server starts listening to incoming requests.
2. Message code 300 triggers a new helper registration which stores helper's endpoint address.
3. Message code 100 triggers an index operation which is followed by sending a directory/URL to the master.
4. New index jobs are enqueued in the master.
5. Master distributes index jobs into its available helper servers and dequeues them upon successful indexing operation.
6. Helper server sends the indexing results to master as tuples of (term, frequency).

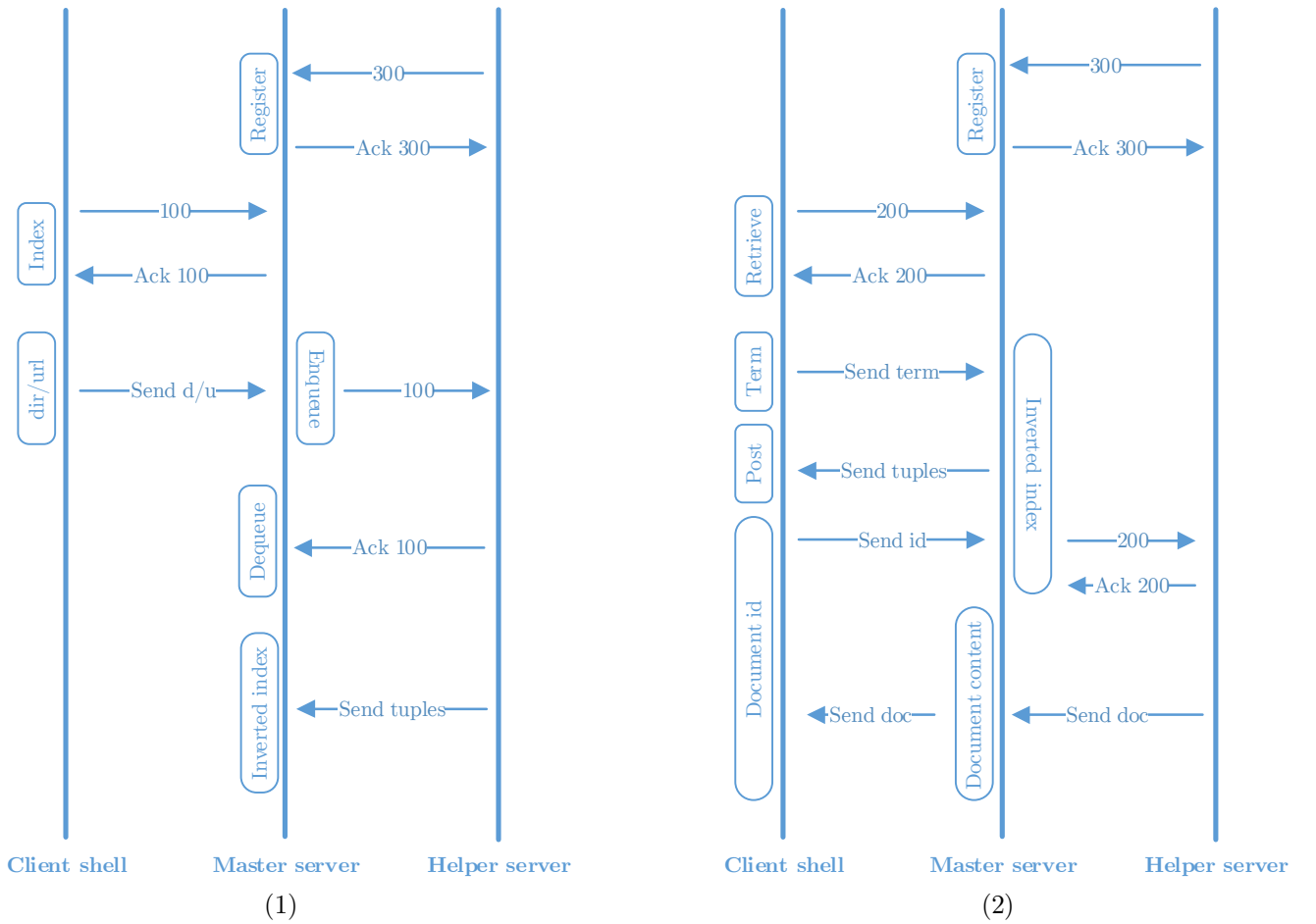


Figure 4 Mini Google request/response flow. (1) Indexing operation, and (2) retrieval operation.

Figure 4 – 2 shows the retrieval part of the proposed message passing protocol. This part includes the following steps:

1. Master server starts listening to incoming requests.
2. Message code 300 triggers a new helper registration which stores helper's endpoint address.
3. Message code 200 triggers a retrieve operation which is followed by sending the corresponding term to the master.
4. Master retrieves the term from the master's inverted index and sends tuples of (frequency, document id) to the client.
5. The client may continue its query by entering a specific document id from the received post list.
6. Master server lookups the document id from the post list which is retrieved previously.
7. Message code 200 triggers a retrieve operation from master server which is followed by sending the corresponding document id to the helper server which hosts it.
8. Helper server lookups the requested document id in its document list and sends back its content to the master.
9. The master sends the document content to the client shell.
10. The client shell presents the retrieved document's content to the user.

2.2 Data Structures

2.2.1 Inverted Index

The inverted index is the key data structure of mini Google. It stores a mapping from unique terms to their document id and number of occurrences in each document. There are three data structures that are using to build the inverted index:

1. The *post data structure* which is a linked list that stores the term name and its corresponding frequency and document id for a group of similar terms.
2. The *term data structure* which stores *post data structure* into a hash map function for efficient access.

Term	Doc. name	char*
	Doc. id	char*
	Doc. host name	char*
	Doc. host port	char*
	Term	char*
	Frequency	int
	Post	post*
(1)		
post	Doc. name	char*
	Doc. id	char*
	Doc. host name	char*
	Doc. host port	char*
	Term	char*
	Frequency	int
	Next	post*
(2)		

Figure 5 Inverted index key data structures. (1) Term list, and (2) post list.

2.2.2 Job Queue

Mini Google equipped with a queue for handling incoming indexing jobs. This queue consist of a linked list of incoming documents for index operation (Figure 6 – 2), where each document is stored in the *document data structure* before inserting to the job queue (Figure 6 - 1).

Document	Doc. name	char*
	Doc. id	char*
	Doc. host name	char*
	Doc. host port	char*
	Doc. source	char*
	Doc. path	char*
	Doc. directory	char*
(1)		
Queue	Job id	char*
	Job document	document*
	Queue	queue*
(2)		

Figure 6 Job queue key data structures. (1) Document list, and (2) queue list.

2.2.3 Service Type

Service data structure (Figure 7) is a unified data structure that is used for storing the server and helper endpoint information such as their IP address, port number and type ({Type: master, slave, and helper}). The *create()* procedure is responsible for creating a new service e.g. adding a new helper server. This function accepts a registration buffer which consists of a hostname, port number and type for a server and stores it for further use.

Service	IP address	char*
	Port number	char*
	Service type	char*

Figure 7 Service data structure for server and directory service.

2.3 Master/Slave Server Tier

Master server is the key tier of mini Google implementation. It handles all communication and distributes all computation of mini Google. So, the concurrency is one of the challenges that should be addressed while implanting this tier. Thus, the master server tier follows the multithread framework of Figure 8. As shown in Figure 8, these threads collectively fulfill the index/retrieve tasks. In the following of this section, the components of master server tier will be introduced (Figure 8).

The whole process of Figure 8, starts with an index/retrieve task from the client shell. Upon receiving an index/retrieve task, master server distributes the new task to helper nodes/searches the inverted index. Also, the presence of heartbeat and replication provide the high available feature for master server. The following of this section is designated to the in-depth discussion of master server's components. Furthermore, Figure 9 is the pseudocode of master server.

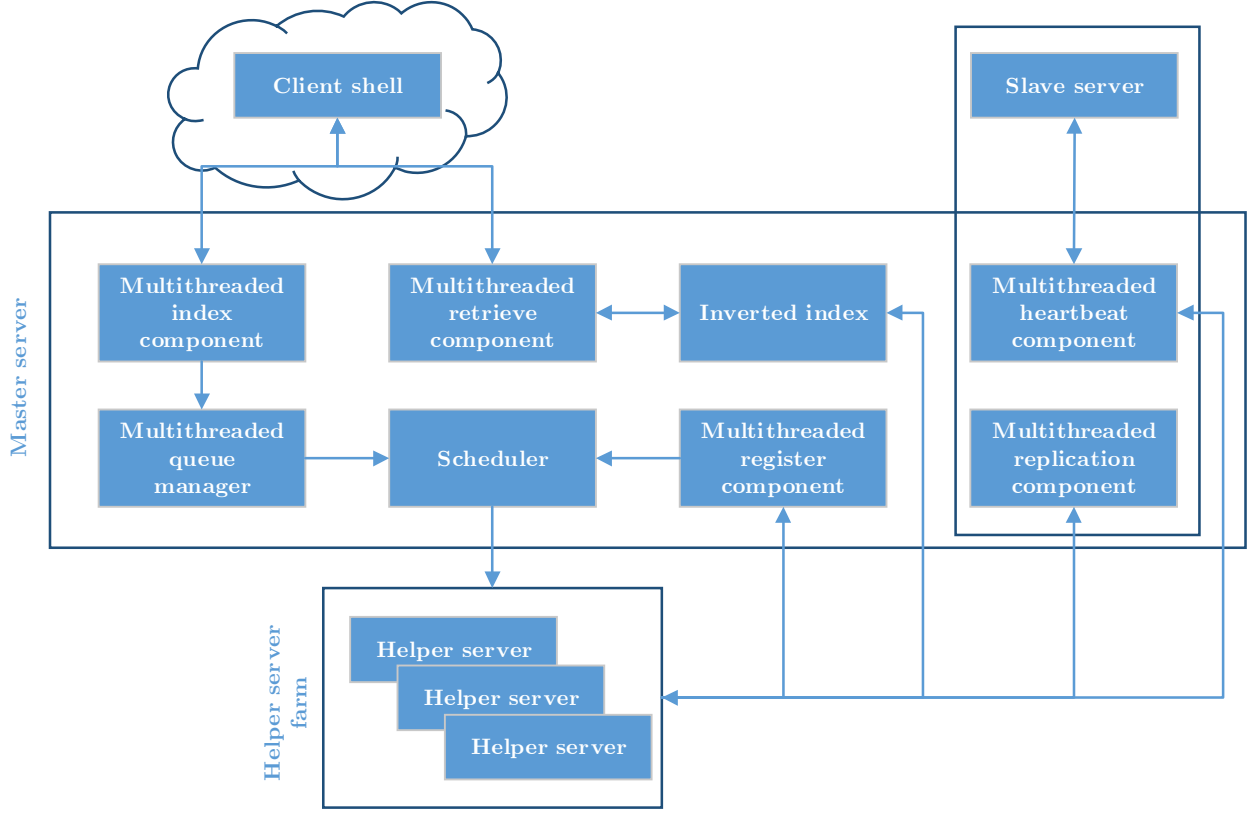


Figure 8 Multithread design of master server.

Master/Slave server pseudocode

```

if(type == slave)
    while(True)
        if(!heartbeat(master)) // Monitor server connection
            break;
    else
        continue;

create(queue manager thread) // Start queue manager
while(True)
    if(100)
        create(index thread) // Indexing job
    if(200)
        create(retrieve thread) // Retrieval job
    if(300)
        create(register thread) // Helper registration
    if(400)
        create(heartbeat thread) // Keep alive message
    if(500)
        create(replication thread) // Replication message

```

Figure 9 Master server pseudocode.

2.3.1 Index Component

This component receives the incoming documents from the client shell. At first, this component receives the number of documents incorporated to a batch indexing job and then receives and enqueue them into the job queue.

2.3.2 Queue Manager and Scheduler Components

The queue manager component iteratively checks the job queue of master server. Upon inserting a new indexing job to the queue, queue manager fetches it and passes the new job to the scheduler. The scheduler then selects a helper node with respect to the round robin policy and dispatches the job to it. Upon successful completion of indexing job, the inverted index will be updated and the next job will be started. Figure 10 shows the process of index job submission into the index queue and dispatching it to a helper node, and finally updating the master's inverted index.

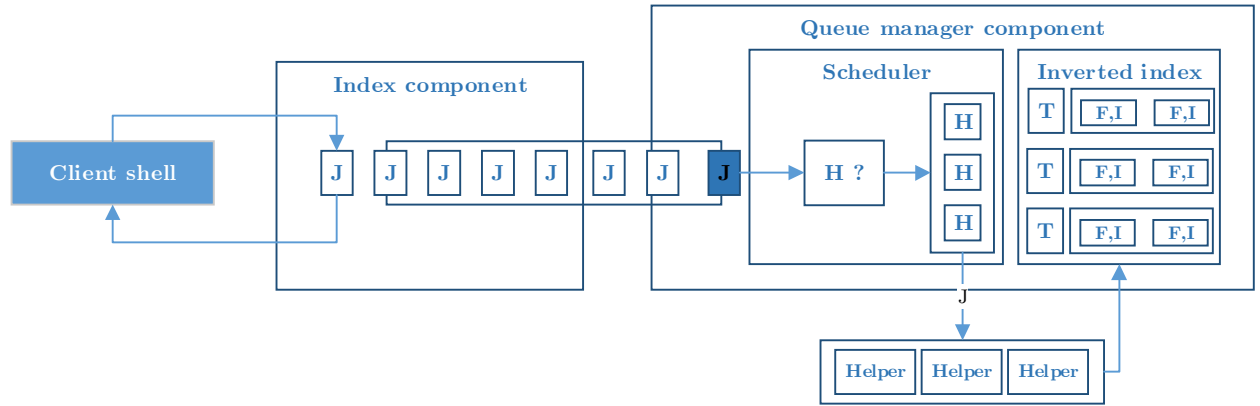


Figure 10 Indexing flow. J, H, T, F, and I denotes an indexing job, helper node, term, term frequency, and document id, respectively.

2.3.3 Retrieve Component

As shown in Figure 11, the retrieve component receives the requested term from the client shell and process the inverted index to extract the corresponding post information. It then sends back the extracted post information to the client which consists of the number of occurrences of the requested term in documents in which contain the term. The user then would ask for the content of a specific document by passing its corresponding id. Next, master locates the location of the requested document, and asks its corresponding helper to retrieve the document content. Finally, the master pass the requested document content to the client shell to show it to user.

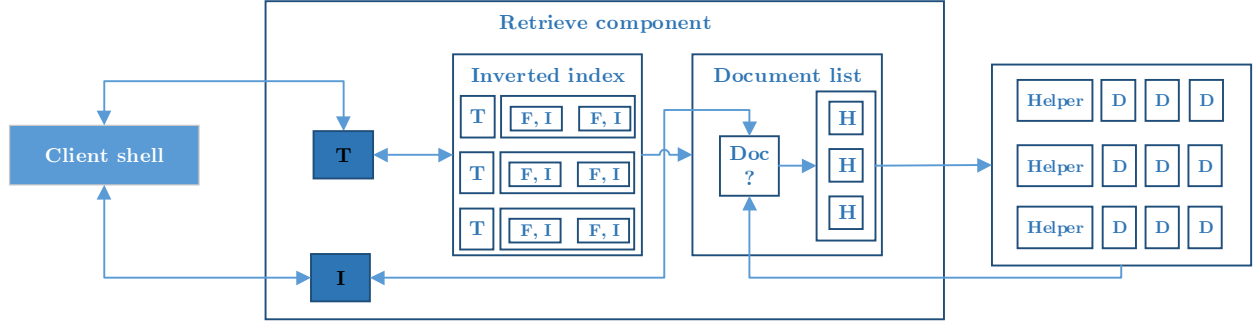


Figure 11 Retrieval flow. J, H, T, F, and I denotes an indexing job, helper node, term, term Frequency, and document id, respectively.

2.3.4 Register Component

The master server utilizes the register component in order to manage the registered helpers. This component allows master to store the endpoint information of new helpers and shares this information with the scheduler for efficient distribution of incoming jobs between helper. Figure 12 shows the process of registering a new helper server.

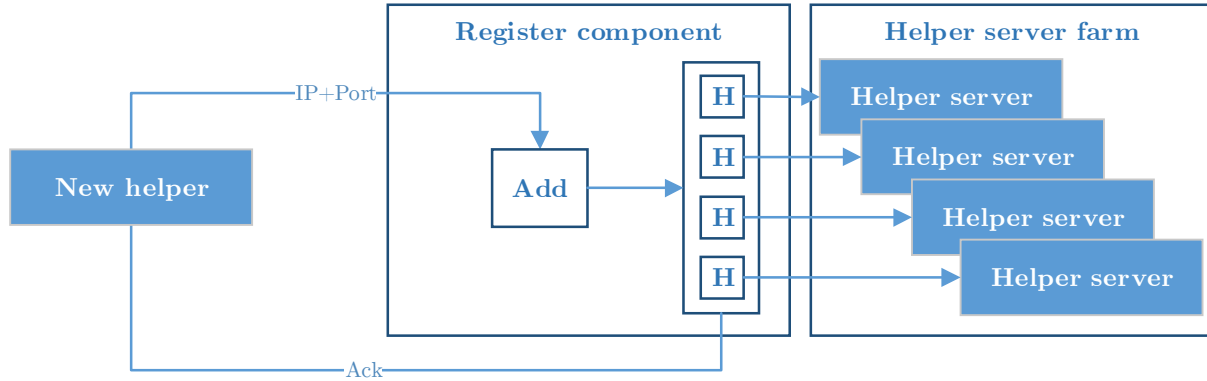


Figure 12 Helper server registration process where H denotes a helper node.

2.3.5 Heartbeat and Replication Components

The heartbeat component receives the heartbeat messages from slave server and helpers, and send acknowledgment messages back to them. In case of failure, the slave initiates the high availability feature of mini Google which will be discussed in Section 2.6.

In case of plausible failure, slave server triggers the high availability scenario and the mini Google resumes its regular functionality, while the replication component waits for replication messages from the helper nodes to recover the inverted index by collecting and combining the sub inverted indexes of helpers (Figure 13).

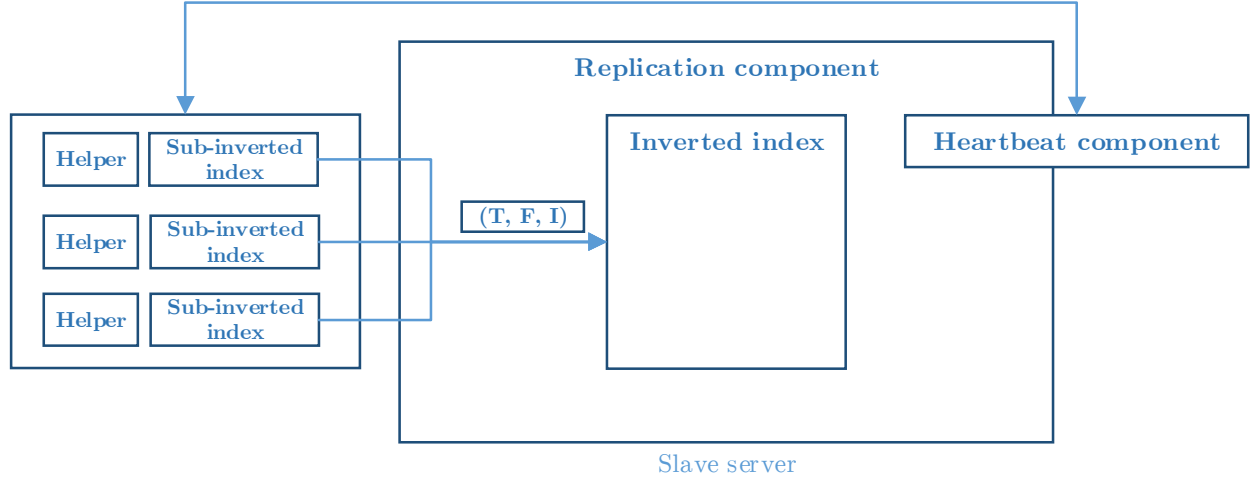


Figure 13 Slave server replication process where T, F, and I denotes the term, term frequency, and document id, respectively.

2.4 Helper Server Tier

The helper server tier handles the computation part of mini Google. This tier consists of a pool of helper nodes that the scheduler assigns an index/retrieve task to them. Each helper implemented in a multithreaded fashion to handle concurrent index/retrieve tasks. Figure 14 shows the logical components of a helper server. The keepalive component monitors the server connection by sending heartbeat messages. Moreover, the index/retrieve components handle the incoming index/retrieve tasks. Also, there is a local inverted index in helper node for plausible server failure and a list of indexed document list for locating document content.

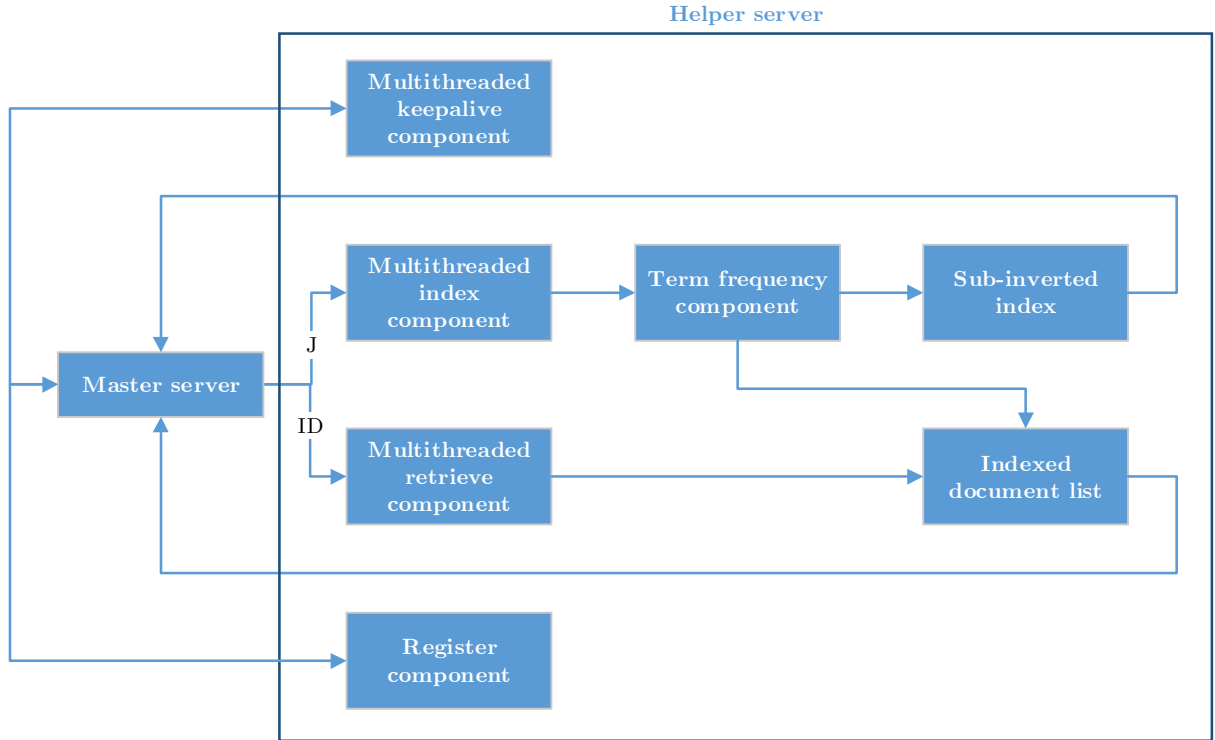


Figure 14 Multithread design of helper server where J, and ID denotes index job, and document id, respectively.

As shown in Figure 15, helper server consists of three multithreaded components. The keepalive thread which actively checks server connection by sending heartbeat messages, and the index/retrieve threads which handles the index/retrieve tasks from the master server.

```

Helper server pseudocode
Register(helper) // Send registration message to master
create(keepalive thread) // Start keep alive operation

while(True)
    if(100)
        create(index thread) // Indexing job
    if(200)
        create(retrieve thread) // Retrieval job

```

Figure 15 Helper server pseudocode.

2.4.1 Index and term frequency Components

Upon receiving an indexing message with message code 100, the index component of helper server passes the incoming document to the term frequency component. The term frequency component starts to parse the received document and compute the word count for each unique word. If the parsed word is a new one, term frequency component creates a new entry for it in the inverted index, otherwise it adds the term along with its document id to its matched post list. Figure 16 shows the term frequency pseudocode for parsing a new document. Note that except for the parsing part, the master server also follows a similar process to the Figure 16 for building the inverted index.

```

Term frequency pseudocode
line = document;
while(line)
    while(term in line)
        if(new term)
            new post = insertSort(new term, document id);
        else
            post = get(term);
            if(new document)
                post = insertSort(term, document id);
            else
                post = update(post, term, document id);
                post = bubbleSort(post);

    inverted index = put(post);
    line = newline;

```

Figure 16 Term frequency pseudocode.

2.4.2 Retrieve Component

The retrieve component is a thread that is invoked upon receiving a 200 message code. As shown in Figure 14, the retrieve component receives the document id from the master server. It then performs a linear search through a list of documents and transfers the content of the requested document back to the master server.

2.4.3 Register Component

As show in Figure 14, before starting the primary service of the helper, it should register itself to the master node. By performing this operation, master will add the endpoint address of helper server into its helper list for future index/retrieve operations.

2.4.4 Keepalive component

Figure 17 shows the overall processing flow of keepalive component. This component is one of the key parts of the high availability feature of mini Google. This component continuously monitors the master server status by sending heartbeat messages. If the connection to the master node has been lost, it starts sending registration messages with the hope that slave server will resume the functionality of mini Google. After successful registration, it sends replication message to the slave node and stars transmitting its local inverted index information to the slave. The process of converting the local inverted index into the network buffer ready for serialization is similar to matrix vectorization. It converts a 2d linked lists of posts to three 1d vectors of term, term frequency, and document id.

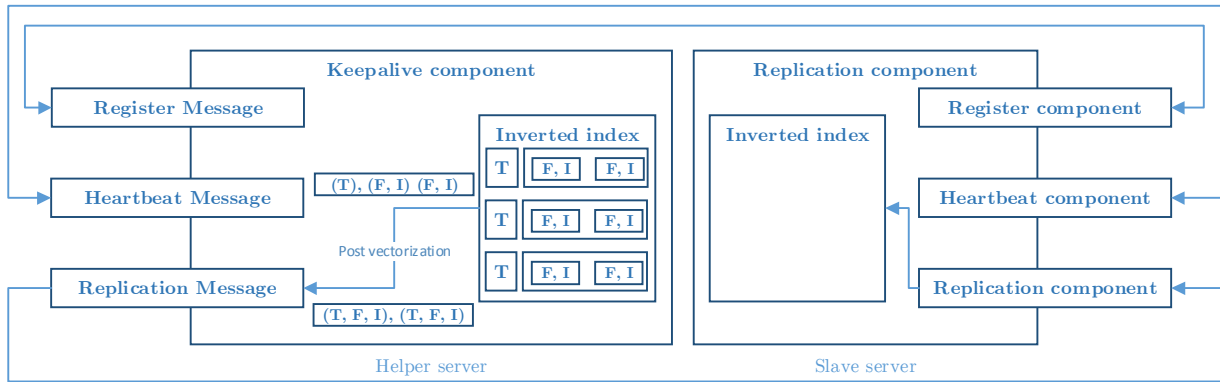


Figure 17 Helper server keepalive component. By losing the server connection, helper register itself to the slave server and starts the replication process. T, F, and I denotes the term, term frequency, and document id, respectively.

2.5 Client Shell Tier

The client shell tier provides necessary interfaces for users to connect to mini Google. As shown in Figure 18, this tier accepts the index/retrieve tasks form the user and presents the results to the user. Primarily, client tier provides two daemons including shell and web daemon for user connections. The shell daemon accepts connection from SHH session, while the web daemon accepts connection from browser.

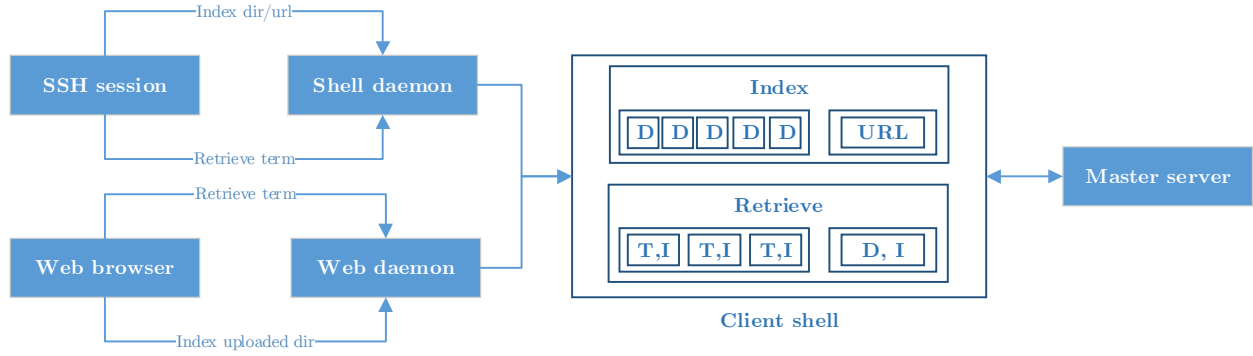


Figure 18 Client shell tier. D, I, T denotes document, document id, and term, respectively.

2.5.1 Shell daemon

The overall process of the shell daemon is as follows:

1. A user connects to the SSH daemon via a SSH session.
2. The user could send an indexing job in the form of a directory or an URL.
3. Also, the user could retrieve a term and consequently the content of one the retrieved documents.
4. Receiving the user's jobs, the shell daemon passes the index/retrieve jobs to the client shell.
5. The client shell communicates with the master server and provide the results of operations to the shell daemon.

2.5.2 Web daemon

The web daemon of mini Google handles request from the mini Google homepage which is written in HTML, CSS, JavaScript, and PHP. The overall process of web daemon is almost similar to the SSH daemon, except that web daemon uses file IO to send/receive the results of index/retrieve operations.

2.6 High Availability

In this section we discuss about the high availability feature of mini Google. If a search engine provides high availability, it then must be able to recover from service failure. The following section describes how mini Google detects a failure and how failover is accomplished by data replication.

2.6.1 Failover Detection

Figure 19 - 1 shows a failover scenario, a slave server is started along with the master server. The master server is responsible for handling all incoming traffics from the clients or helper and the slave server monitors the master's presence by sending heartbeat messages. Also, helper server sends heartbeat messages to master server in case of detecting failure. As shown in Figure 19 – 1 message code 400 is used for failover messages.

3. The slave server initiates a command to listen to the master endpoint address; meanwhile, the helper server switches its current status to unregistered and starts sending register messages again to the server endpoint address.
4. After endpoint assignment and boot up, slave server accepts the message code 300 from helper which triggers a new helper registration and stores helper's endpoint address.
5. Upon successful registration, message code 500 triggers a replication operation from helper to slave server which is followed by sending an acknowledgment message from master to helper.
6. The helper starts sending triplets of (term, frequency, document id) to slave server.
7. The slave server receives triplets and start building the new slave inverted matrix.
8. Upon successful replication, the helper starts sending message code 400 to the slave server.

3 Conclusion

This report presents mini Google which is a system software written in C. It allows a fast and resilient way of full-text search. It supports the basic operations of full-text search including index/retrieve by implementing an inverted index data structure. Receiving an index operation followed by a directory or URL from the user, mini Google updates its inverted index by processing new documents. Moreover, a retrieve operation followed by a document id leads to bring back the content of a previously indexed document to the user.

From the system architecture viewpoint, mini Google consists of 4 key tiers that handles operation specific tasks:

1. Client shell tier which accepts the index/retrieve operations from users
2. Master tier which distributes the index/retrieve operations among available helpers, and also maintains the inverted index which contains the terms' ranking.
3. Slave tier which monitors the master status and initiates the high availability scenario in case of master failure.
4. Helper tier which handle the computation part.

Moreover, from the software architecture viewpoint, each tier of mini Google composes of different logical components. The interconnection of these software components emerges the set of utility features of mini Google:

1. The multithread implementation of mini Google handles concurrent index/retrieve operations.
2. The high availability implementation of mini Google presents the failover detection and replication features.
3. Intuitive implementation of inverted index with a linked list for posting list and a hash map function for terms list provides a fast way of index/retrieve operations.
4. The SSH session/Web browser interface types of mini Google provides an easy to access solution.