**DS Design**

The doc explains the framework and applications built around it for the search engine project. The framework written is general purpose and can be used to build any application where task parallelism is needed.

**Requirements**

**Y: Implemented**

**N: Not implemented**

**P: Partially implemented**

|  |  |  |
| --- | --- | --- |
| **R#** | **Description** | **Done** |
| R1 | A simple, easy to use framework to build applications which can | **Y** |
| Communicate among themselves. | **Y** |
| Are easy to manage. | **Y** |
| Communicate without the hassle of providing conn detail each  time. | **Y** |
| R2 | A simple API to probe the state of these applications. | **Y** |
| R3 | An application to index the data crawled by the crawler sub-system | **P** |
| Data structures to efficiently index the pages | **N** |
| Interface to reverse index the docs using the query words. | **Y** |
| R4 | An application to provide the interface to the user to query the doc-links using text query | **Y** |
| A HTTP interface/server for processing the queries and providing  the results. | **Y** |
| A feedback management system to record the feedback provided  by the user. | **Y** |
| R5 | An application (proxyserver) which can map the document IDs to the URLs | **P** |
| A HTTP interface to this proxyserver. | **Y** |
| Mapping of document IDs to URLs | **Y** |
| Tracking and improving the search results based on the hits. | **N** |

**Architecture**

The design principle of the framework is master slave. There are basically two kind of applications which run in the framework. Slave applications are also of two types.

1. Master application
2. Slave applications
   1. Node manager
   2. User applications

Here are some of the common terminologies used in the document.

1. **Node:** Node is the logical representation of a machine. It has a *Node manager* *(Fireup)* running on it which allows the master application to create the user processes on this node. You can create multiple nodes in the single physical machine as nodes are logical entities.
2. **Master:** Master is an application running on a node in the domain/cluster and provides the following functionalities.
   1. Creation of the processes
   2. Managing of the processes
   3. Killing the processes

You can extend the functionalities of the master by extending the *MasterProcess* class.

1. **Slaves Processes:** There are two kinds of slaves running in the domain.
   1. **Node manager:** These help in managing processes in the nodes. The functionalities of these processes involve
      1. Creation of process
      2. Termination of process
      3. Log management
   2. **User applications** These are the actual processes which are of the use. The framework allows you to write these slave applications by extending the *SlaveProcess* class.

**Let’s look at the design in terms of Class Diagrams**

1. **Framework**

**A close up of a map

Description generated with very high confidence**

1. **Node manager**

**A screenshot of a social media post

Description generated with very high confidence**

Let’s look at the design in terms of scenarios (sequence diagrams).

***Scenario 1. Creating and Registering a node.***

1. Start a node manager (fireup) instance.

$: java -jar \

fireup.jar \

<master-ip> <master-ipc-port> \

<configuration-file-path>

1. Master creates a ticket for the node and assigns it to the node.
2. Node uses this ticket each further communication. The processes created in the node must inherit the same the ticket.
3. Master also sends the minimum JAR version of the applications it has (timestamp when jar was created).
4. Master uses the given scheduler to schedule the jobs on the node.

![A close up of text on a white background

Description generated with very high confidence]()

***Figure. Node startup sequence diagram.***

***Scenario 2. Process creation.***

1. The master sends the process creation arguments such as jar name and command line arguments to node manager.
2. The node manager checks whether the JAR specified is latest and if not, it downloads it from the master using HTTP protocol.
3. Node manager creates a directory with ***app id*** for the logs at **NODEMANAGER\_HOME/logs/**
4. The ***app id*** has following format **“proc\_DDMMYYYYHHMMSS\_appCount”**. Here appCount is an incremental counter (Starts from 1 for each instance of nodemanager).
5. The node manager creates the process with below command line arguments.
   1. **Ticket** (assigned by master)
   2. **App id**
   3. **Host** and **ipc port** of the master.
   4. Other arguments sent by the master.
6. The process after start, reports to master about its start by sending ***ConnectPDU.***
7. The master introduces this new application/guest to all the other applications.

![A close up of text on a white background

Description generated with very high confidence]()

***Figure. Sequence diagram of process creation***

***Scenario 3. Introducing a guest app.***

1. All new applications in the domain are considered as guest and are introduced to other apps after the guest reports to master.
2. A ***IntroPDU***is sent to all other applications
3. After receiving the ***IntroPDU*** the applications say *“Hi”* to this new application by sending a ***HiPDU****.* This helps the guest in knowing the ports on which other applications run.
4. After receiving a ***HiPDU***the applications send back a “Hi” (similar ***HiPDU***).

![A close up of text on a white background

Description generated with very high confidence]()

***Figure. Introducing the guest application to others.***

***Scenario 4. Heartbeat***

1. The apps running in the framework send heartbeat signals to the master about its activeness after every HEARTBEAT\_INTERVAL milliseconds.
2. The master waits 3 \* HEARTBEAT\_INTERVAL time for the heartbeat of an app and decides that the app is hung or dead after this time.

![A close up of text on a white background

Description generated with very high confidence]()

***Figure. Heartbeat sequence diagram***

***Scenario 5. Process termination***

1. The master can terminate the processes running on nodes if user wants to take down the domain/node/process. This can be done by REST API provided by the framework.
2. The graceful termination of a process can provide options such as shutdown hook. (cleanup call)

![A black and silver text on a white surface

Description generated with high confidence]()

***Figure. Termination of a process***

**Inter-Process-Communication Protocol**

The IPC is kept completely independent module so that we can change the IPC method with minimal changes. The two ways of passing the PDUs (Protocol Data Units) are

1. JSON data
2. Serialized Java Objects

The first one is helpful in debugging the applications while second one can lessen the bandwidth usage. The impressive part here is we can switch over the modes by just changing handful (4-5) lines of code.

The protocol data units are explained below.

|  |  |  |
| --- | --- | --- |
| **PDU Type** | **Field** | **Field Detail** |
| PDU | who | Process type of the sender application |
| method | Integer value indicating the type of message (refer methods) |
| data | Additional data (for extension) |
| AckPDU | jarRevision | The minimum version of the JAR file the server has |
| restPort | Http port of the application |
| ticket | The ticket assigned to the recipient node manager. |
| ConnectPDU | ipcPort | The port on which the sender process listens for IPC |
| pid | Process ID of the sender process |
| sysInfo | System’s hardware details |
| sysInfo.cores | Available cores in the system |
| sysInfo.availableMemory | Memory available for JVM |
| sysInfo.freeMemory | Free memory available in JVM’s space |
| sysInfo.startTime | The timestamp when the system started |
| sysInfo.osType | The name and version of the OS |
| sysInfo.javaVersion | The installed JAVA version |
| CreatePDU | executable | The JAR which master wants to execute |
| arguments | Command line arguments for the process being created |
| DiePDU |  | This is sent the process which must be killed |
| HiPDU | ipcPort | The port on which the process saying **Hi** listens for IPC |
| httpPort | The port on which the process saying **Hi** listens for HTTP |
| IntroPDU | host | The IP of the machine on which guest process is running |
| port | The IPC port on which guest process listens |
| type | The type of the guest process |
| KillPDU | pid | The PID of the process which must be killed |
| SearchPDU | query | A string which must be reverse-indexed |
| StatusPDU |  | Sent as a heartbeat |

**Configuration of processes.**

All the processes running in the cluster need a configuration object to instantiate. The user processes may extend the configuration object provided by the framework to add their own attributes. The generic configuration object looks as follows. The configuration objects are specified in JSON format (one example is given below)

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Datatype** | **Comment** |
| ticket |  |  |
| pid | String | PID assigned to this process by node manager |
| processrole | String | Type of the process (Just an identifier for type) |
| masterHost | String : IP | IP address of master |
| masterPort | Integer | IPC port on which master listens |
| log-level | Integer | 0: Protocol level  1: Debug level  2: Low level  3: Medium level  4: High level |
| doc-root | String : Path | Path to the document root of the http server. |

{

    "ticket": "",

    "masterHost": "",

    "pid": "",

    "masterPort": "",

   "debug-level": "1",

    "processrole": "dmgr",

    "doc-root": "."

}

**User processes**

There are totally 3 user processes in the cluster for the search engine. Each of them is explained below

1. **datamaster:** Master process of the cluster

This is application which manages all the nodes and processes on the cluster. It extends the ***MasterProcess*** class and adds its own scheduler to schedule jobs on the cluster.

Below is the class diagram of the datamaster application

A screenshot of a cell phone

Description generated with high confidence

***Figure. Class diagram of datamaster.***

1. **dmgr** : Data Manager

This is the application which processes the WebPages downloaded and index them. The way it works is simple. It polls the *content-dir* directory after every *scan-interval* time and indexes the docs found in it. The application has the following configuration attributes defined.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Datatype** | **Comment** |
| scan-thread-count | Integer | Number of threads running in parallel processing docs |
| dictionary-file | String: Path | The path where the trie is stored |
| content-dir | Path | The path to directory where the indexing data is stored |
| trie-file | String: Path | The path to file where serialized trie object is stored |
| temp-dir | String: Path | The path to directory where the downloaded web-content is stored |
| scan-interval | Integer | Time in milli-seconds after which scan job is triggered |

**Indexing:**

Indexing and reverse indexing is the process of storing and retrieving the documents based on some key parameter. The key used here for indexing are words itself.

The indexing is done in a naïve way. Below are the steps

1. The scan thread polls the “***tmp-dir***” (defined in the configuration of ***dmgr*** process) for new documents.
2. If new documents are found in then this thread indexes it.
3. The document has the document assigned as its file name
4. For each word in the doc we create a Set which holds the document ids of the docs which have this word.
5. These sets are written back to the word files present in the ***“content-dir”*** directory (defined in the configuration of ***dmgr*** process).

A close up of a map

Description generated with high confidence

***Figure. Class diagram of the dmgr***

1. **wserver** : Web Server

This is the application which provides the user-interface to the user. It can pass the queries by the user to the dmgrs and get the docids and present them to the user in the browser. The configuration of this process is has nothing more than the generic configuration.

Below is the steps executed to get the search results.

1. User submits the query.
2. Web server creates ***SearchAgent*** threads to search the docs.
3. ***SearchAgent*** connects to the ***dmgrs*** and pass the query.
4. ***dmgrs*** return the docids of the documents
5. ***wserver*** packs all the docids as a ***SearchResponse*** and returns to the user.

Below is the class diagram of the webserver

A screenshot of a social media post

Description generated with very high confidence

***Figure. Class diagram of the web-server***

1. **proxyserver**: Proxy Server

This is the application which provides the mapping from docid to URL. By this we can track the hits for a web-page. The proxy-server listens to the crawler units to get the docid to URL mapping. The configuration of this process has nothing more than the generic configuration

The mapping task is done as follows.

1. User clicks a link in the search results page.
2. A HTTP GET is generated and sent back to the ***proxyserver***
3. The ***proxyserver*** maps docid in GET request to the respective URL and sends back a redirect HTTP header.
4. ***proxyserver*** also increments a counter on the link.
5. The user is redirected to the expected page.

Below is the class diagram of the proxy server process.

A screenshot of a cell phone

Description generated with high confidence

***Figure. Class diagram of the proxyserver***

All the above processes exchange the ports on which they listen by introduce protocol if they have overridden the ***handle\_hello*** function they can use the information sent in the PDU. For example the ***wserver*** gets the ports of ***dmgr*** and ***proxyserver*** from this protocol.

**Logging**

The logs in the frameworks have a defined syntax and are logged by a utility class called ***Logger***. There are 4 different levels in which logs can be logged and the logs can be suppressed using the ***log-level*** property in the configuration of the process. The exceptions generated can also be logged using the ***Logger*** class.

Below are the levels of the logs which are permitted.

1. Protocol: All the IPC PDUs are logged here.
2. Debug: All the debug messages
3. Low: The messages with low priority such as start of a thread.
4. Medium: The messages with medium priority, such as detection of a co-process
5. High: The messages with very high priority, such as crash, shutdown etc.

Logs are also of two types.

1. INFO: Informational logs
2. ERROR: Exceptions or errors

Below is the syntax of a log.

**Timestamp Type Thread Message**

Tue May 29 23:48:36 IST 2018 INFO/ERROR [main] Sample log

Below is the class diagram of the Logger class.

A screenshot of a cell phone

Description generated with very high confidence

**API**

The processes in the framework may choose to expose their status. This can be achieved by implementing the ***ProgressableProcess*** process interface and providing the status object (Any type of object which has fields marked as JsonExposed and methods as RESTExposed) when a callback for ***getProgress*** is done.

Below is the example list of APIs provided by the framework

**Properties**

|  |  |  |
| --- | --- | --- |
| **End point** | **Data type** | **Comment** |
| /status/nodes | java.util.LinkedHashMap | This is a hashmap of slaves. Access the elements using the key [ticket] |
| /status/nodes/${key}/sysInfo | se.dscore.SysInfo | The slave system's resources |
| /status/nodes/${key}/processes | java.util.HashMap | The processes running on this slave |

**Methods**

|  |  |  |
| --- | --- | --- |
| **End point** | **Data to be sent** | **Comment** |
| /exec/shutDownDomain | **NA** | Shuts down the whole domain |
| /exec/shutDownNode | {'node': 'Node-ID'} | This can be used to stop the node. |
| /exec/nodes/${key}/kill | **NA** | Kills a process on this slave whose PID is sent |
| /exec/nodes/${key}/processes/${key}/runMethod | **NA** | Runs a method in given process. Nothing is sent back |

User can take advantage of implementing the UI for managing the tasks in the domain or of the particular process using the above API.