Map Reducer SW Design Document

Phase 4

Group 5: Abi Parekh, Emmanuel Sevieux, Stephen O’Connor
Date: 12/14/22

Design of Map Reducer software

TABLE OF CONTENTS

1 Introduction

All the detailed design information will be captured within this Software Design Document. The design information includes but is not limited to Software Architecture, Process Flow, and Algorithm Details.

1.1 Document overview

This document describes the design of the Phase-3 Map Reducer Software Component, part of the Object Oriented Design Class.

2 Software Architecture overview

The Phase-3 Map Reducer Software is a Microsoft Visual Studios based Program that is executed from the Command Line. Below Figure 1 (UML Diagram) describes how the classes fit together while Figures 2 and 3 indicate the responsibilities of the different Software Components. Note that the Software can be broken into 4 different components:

1. **MapReducer WorkFlow**: Top Level Class which validates the input configuration, access DLLs and creates the threads for Map and Reduce.
2. **Mapper DLL**: Extracts individual words (tokens) from files and exports results to intermediary directory
3. **Reducer DLL**: Arranges tokenized words into alphanumeric sorted order and sums the total occurrence of each word.
4. **File Utility**: Set of Common Functions for Reading and Writing from File IO that is used by all of the software components.

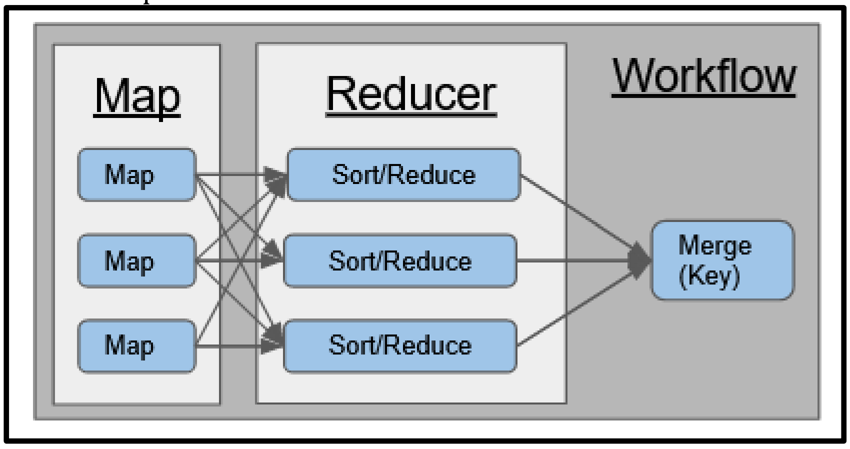


Fig. 1. Thread Breakdown

3 Software design description

3.1 Workflow

3.1.1 Client-Server Socket Design

This project phase introduces a Client-Server architecture comprised of 3 components:

- **Controller**: This server provides the general workflow directing when Mapper and Reducer threads are to be generated. It creates a client connect to each stub to be able to send requests to createThreads. Each thread can be designated to run a mapper or reducer process, and when complete, it sends a message back to the controller, to notify that a mapped or reduced file has been generated. The controller can then decide to send the mapped files by signaling the reducer stub or call the finalize class if all files have been reduced for purpose of merging all results into one single output file.
- **Stub(listener):** the stub servers sit and listen for request messages from the controller (using sockets)to either create a handler for a Mapper or Reducer request. Once a message is received the stubs will spawn a thread handler passing the socket connection and message request to the handler. Upon successful handoff the stub will wait and listen for a new request. for this program the stub servers will be initiated by Main and presumed to be always running
- **Map/Reduce Processors (Handler)**: Depending on the controller message request the stub will generate either a Mapper or Reducer thread passing the current socket connection as as callable object. The Thread clients will handle the controllers requests and will send a synchronous heartbeat every 3 secs to maintain connection to the controller while processing the request. Once the Map or Reduce is complete the processor will send “Done” Message and close thread.

3.1.2 Sequence Diagram

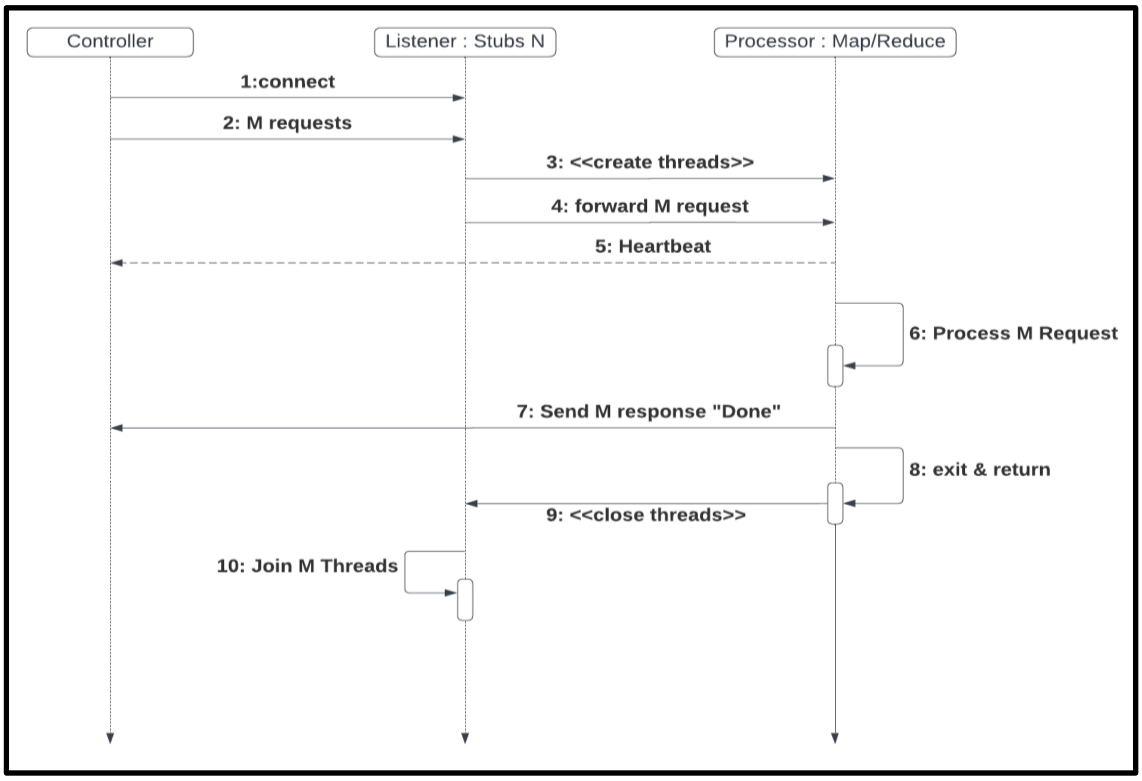


Fig 2: Sockets and Messaging flow of data

3.1.3 Messaging

The Controller will communicate to the stubs and map/reduce processors via TCP/IP messaging. Below reflects the expected behavior for each message type communicated within system:

Message	memberData	methods
BaseMessage: abstract class for all message objects listed below. Pure virtual member functions must be defined by each msg class to support buffer and calculating size	● ● char* buffer	Pure Virtual ● bool createBuffer(Buffersize) ● bool reduceBuffer(char*, Buffersize) ● uint32_t calculateSize()
CreateThreadMsg: This message is sent from Controller -> Stub and used to tell the stub to create a Mapper or Reducer Thread process. users must specify thread type(Map, Reduce)	● uint32_t messageType ● THREAD_TYPE threadType ● string inputFileName ● string outputFolderName	baseMethods + ● void printMessage()
JoinThreadMessage: This message is sent from Controller -> Stub and used to block a stub from accepting new requests until the requested thread is joined. primary purpose is to ensure proper memory deallocation to prevent memory leaks.	● bool join	baseMethods
MappedFileMessage: This message is sent from Processor -> Controller and used to update the controller on the completion status of a given mapper thread. Sent only when the mapper has written the intermediate files to directory and completed processing. This is the final message from the mapper after which no more heartbeat messages are sent to the controller.	● uint32_t messageType ● string outputFileName	baseMethods
ReducedFileMessage: This message is sent from Processor -> Controller and similar to Mapper it updates the controller on the completion status of a given reducer thread. Sent only when the mapper has written files to	● uint32_t messageType ● string outputFileName	baseMethods

the output directory and completed processing. Controller will use this signal on conjunction with the remaining reducers to join threads on completion after which it will run it's own finalizer to consolidate final results.		
HeartbeatMessage: This message is sent from Processor -> Controller and used to maintain updates regarding the state of each Mapper and Reducer thread. by default the heartbeat will be sent **every 5 seconds** and will share one of following Health_Status' (INIT, INPROGRESS, COMPLETE)	● uint32_t messageType ● HEALTH_STATUS healthStatus; ● string threadName;	baseMethods

3.1.4 Component interfaces

The Top Level Map Reducer Workflow Class provides 2 public methods that allow an external user to create and execute the Reduce Functionality.

- **MapReducer(std::string configFileLocation):** A Constructor Function that requires the location of a configuration file. The Configuration File is a simple Text File with the Name of the Variable and its value on each line. Below is the Configuration Key Words:
  - **Input_Directory**: Location of Input Directory
  - **Map_DLL_Location**: Location of Map DLL File
  - **Reduce_DLL_Location**: Location of Reduce DLL File
  - **Number_Of_Map_Threads**: Number of Map Threads that will be created
  - **Number_Of_Reduce_Threads**: Number of Reduce Threads will be created
- **bool reduce(std::string& outputFileName)**: Call to execute Reduce functionality on Files identified via the Configuration File passed in during creation.

The UML Diagram below describes the relationship between the Software Components.

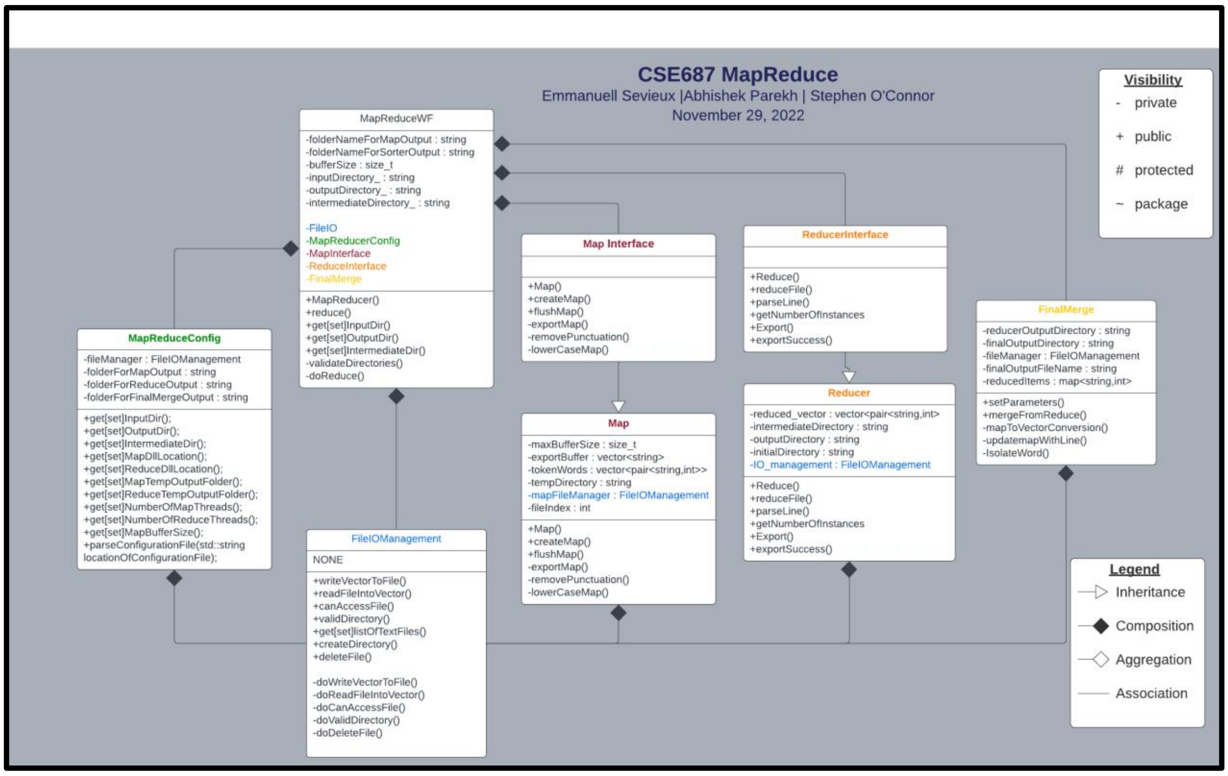


Fig. 2. UML diagram of MapReduce

3.1.5 Component design description

The Figure below shows the End to End Flow Diagram for the program. Additionally, Figure 3 identicats how data is passed between the Threads. The Workflow is broken into 4 separate stages with the 1st stage deal with validating the input information. The 2nd Stage launches X Number of Map Threads which creates mapped files. The 3rd stage takes the output of the Map Files and creates R Reduced Files. Lastly the final creates a single output file based on the R files in the previous step.

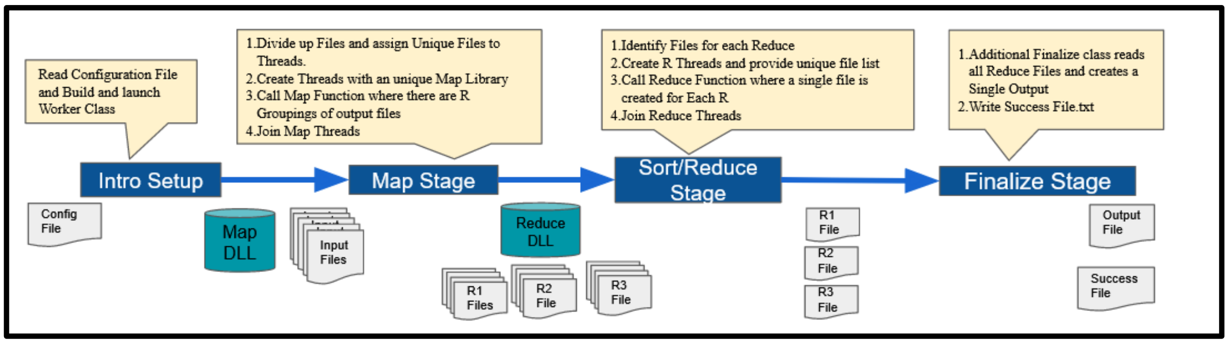

Read Configuration File and Build and launch Worker Class
1.Divide up Files and assign Unique Files to Threads.
2.Create Threads with an unique Map Library
3.Call Map Function where there are R Groupings of output files
4.Join Map Threads
1.Identify Files for each Reduce
2.Create R Threads and provide unique file list
3.Call Reduce Function where a single file is created for Each R
4.Join Reduce Threads
1.Additional Finalize class reads all Reduce Files and creates a Single Output
2.Write Success File.txt
Intro Setup
Map Stage
Sort/Reduce Stage
Finalize Stage
Config File
Map DLL
Input Files
Reduce DLL
R1 Files
R2 File
R3 File
R1 File
R2 File
R3 File
Output File
Success File

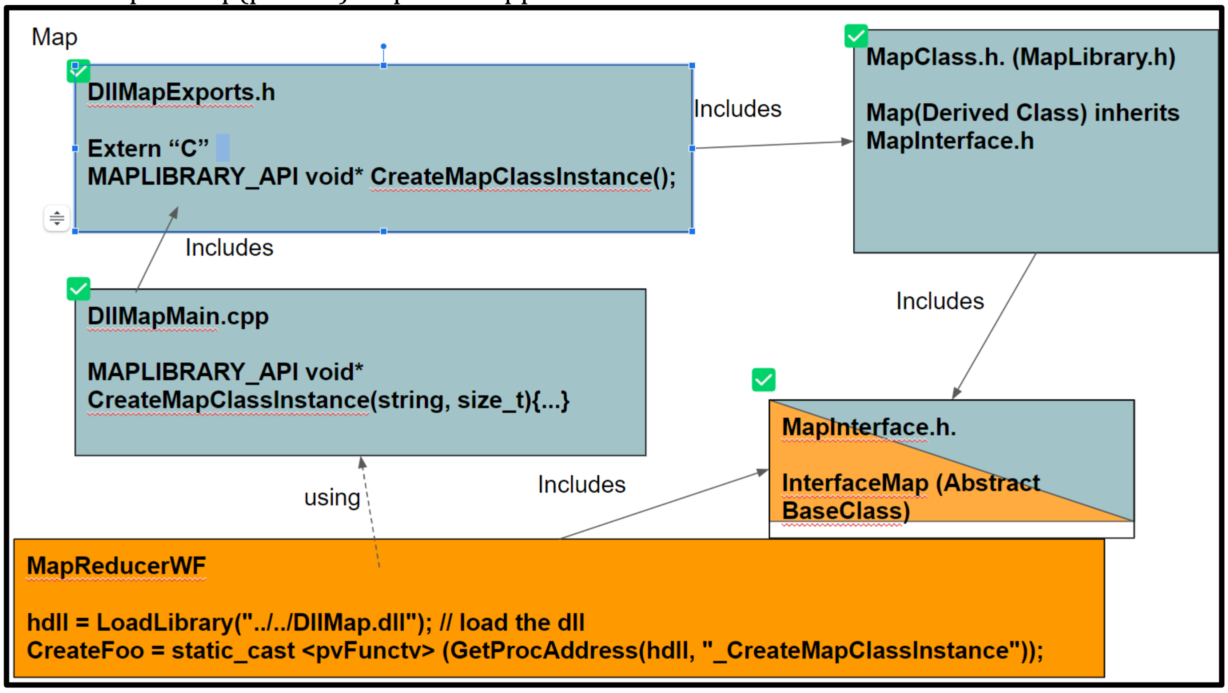
Fig. 2: End to End Flow Diagram

3.2 Map Class

3.2.1 Component interfaces

MapInterface provides pure virtual functions that allow for data abstraction and member function declaration. The map Libraries provide method definitions allow the program to seamlessly integrate with various library versions all Managed via comfier file. Primary methods include:

- CreateMap: reads 1 line at a time from file and tokenize each word into a vector of strings
- FlushMap: Exports the contents of Map onto .txt file on disc and clears cache
- lowerCaseMap: reads line from file and converts letters to lower case
- setParameters: Sets critical data member values
- exportMap(private): exports Mapped tokens to file


Map
DllMapExports.h
Includes
MapClass.h. (MapLibrary.h)
Map(Derived Class) inherits MapInterface.h
Includes
DllMapMain.cpp
Includes
MapInterface.h.
InterfaceMap (Abstract BaseClass)
using
Includes
MapReducerWF

```
Extern "C"
MAPLIBRARY_API void* CreateMapClassInstance();
```

```
MAPLIBRARY_API void*
CreateMapClassInstance(string, size_t){...}
```

```
hdll = LoadLibrary("../../DllMap.dll"); // load the dll
CreateFoo = static_cast <pvFunctv> (GetProcAddress(hdll, "_CreateMapClassInstance"));
```

Fig. 4. Workflow of Map DLL

Component design description

Map Thread Configuration:

To improve efficiency, speed and maximize throughput of the program we've enabled multithreading for the Map processor. The following parameters will determine the number of threads processed and may be adjusted as needed:

- Number_Of_Map_Threads 5

- Number_Of_Reduce_Threads 4
- Map_Buffer_Size 3000

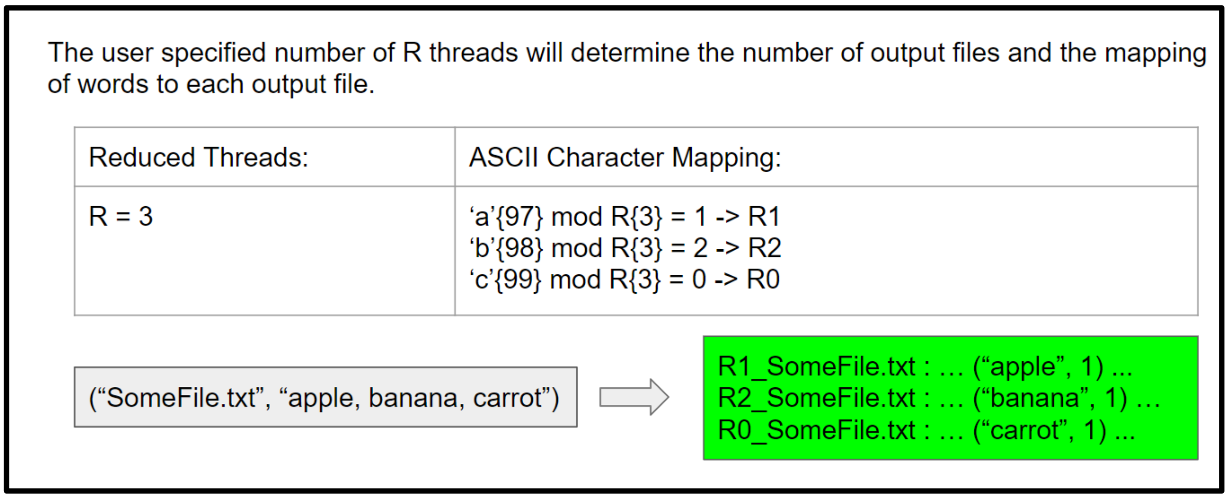
Methods

- *MapThreadFunction* is used to instantiate eachMap object which takes in the Input/Output Directories, File Lists and all configuration settings as parameters. This function is not part of Map Library but is instead part of the general Map Reduce Workflow. It creates a handle to loadlibrary and instantiates a Map object with CreateMap. mapThreadFunction will block the main Workflow while it waits for all map threads to join.
- *SetParameters*: In addition to default Map parameters this method was extended to take as argument an R value indicating number of reducer threads to be created. Map does not create reducer threads however Map will partition the Key (tokenWords) into each R thread.
- *lowerCaseMap*: Since ascii manipulation is used to map words to reducer files we need to ensure duplicate tokens are not created. Therefore converting all string to lower case ensures that two words such as "Hello" and "hello" are counted as the same word.
- *flushMap*: By default the createMap method exports words to intermediate file whenever the user specified buffer limit is reached, however at the end of file it is likely that the buffer will not be reached therefore the remaining text needs to be flushed from cache. This method calls export Map passing an empty string to indicate end-of-file.
- *exportMap*: This method takes the buffer of tokenized words and uses FileIO.h to write to file. It also handles adding the Reducer thread Prefix (i.e. R0_, R1_,etc) to filename utilize to tell reducer threads which files to process. This method uses a helper function emptyCache() that both structures the tokens into tokenPairs (string, int) and frees any used memory from cache to prepare for the next file to be read.
- *createMap*: this method scans each character in a string to determine if it is alphanumeric and splits the characters into a substring representing a token word whenever punctuation is detected. Each token is then passed to exportMap to be stored in temporary buffer before writing to file.

3.2.2 Workflows and algorithms

Map Algorithm

The Map class provides an algorithm to tokenize and partition words from files in a directory and group the words into unique files based on the first character in the words. This partitioning is determined by taking the First character in the word % the Number of Reduce Threads.



The user specified number of R threads will determine the number of output files and the mapping of words to each output file.

Reduced Threads:	ASCII Character Mapping:
R = 3	'a'{97} mod R{3} = 1 -> R1 'b'{98} mod R{3} = 2 -> R2 'c'{99} mod R{3} = 0 -> R0

("SomeFile.txt", "apple, banana, carrot")

R1_SomeFile.txt : … ("apple", 1) ...
R2_SomeFile.txt : … ("banana", 1) …
R0_SomeFile.txt : … ("carrot", 1) ...

Fig. 5. Threads and its outcome

3.3 Reduce Class

3.3.1 Component interfaces

The Reducer class is a derived class of the abstract reduce interface class. This enables the Workflow Class to create and call the Reducer Class's Functions without knowing the implementation at compile time. Instead the Workflow class compiles in with the base class then gets hooks to the implementation via the DLL.

Additionally, the Reducer Class takes in a Folder path and an empty file path name. The class walks through all files within the Output of Map Class Directory and loads them into a standard Map by keyword. After all the files have been read the Reducer Class writes the output Keyword, and value to a file with its thread name appended to the front.

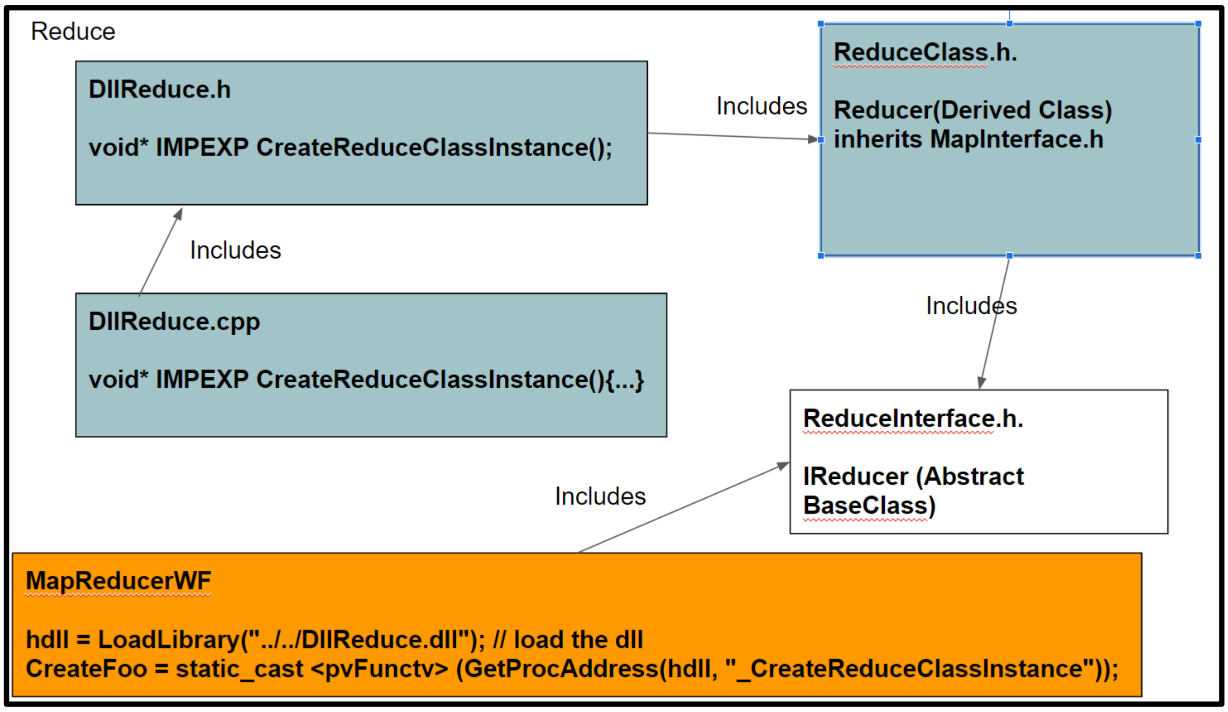

Reduce
DllReduce.h
void* IMPEXP CreateReduceClassInstance();
Includes
ReduceClass.h.
Reducer(Derived Class) inherits MapInterface.h
Includes
DllReduce.cpp
void* IMPEXP CreateReduceClassInstance(){...}
Includes
ReduceInterface.h.
IReducer (Abstract BaseClass)
Includes
MapReducerWF
hdll = LoadLibrary("../../DllReduce.dll"); // load the dll
CreateFoo = static_cast <pvFunctv> (GetProcAddress(hdll, "_CreateReduceClassInstance"));

Fig. 6. Reduce DLL workflow

3.3.2 Component design description

Describe the design of the component, Use package diagrams and/or class diagrams to show the links between sub-components/sub-packages and or classes inside the component.

Methods

- *setParameters:* It is crucial to set up the directory for reducer DLL, since the the output from the reducer will be stored in the ReducerOutputDirectory. In addition, threads need to be defined for the reducer DLL, for the sole purpose of adding multithreading to the program, resulting in much more efficient execution of the console program when reducing and sorting the files.
- *Protected: AddFileContentsToSorter:* This method within the reducer accepts a folder and a filename for the sole objective of accessing the contents exported from the map DLL. It reads the file, iterates through each line, and adds the phrases to a map, and return true, indicating it has appended the contents from the file to the map.

- *Protected: AddPhraseToMap:* Taking the results from isolateWord, that being type string, this method takes the contents from the string and appends it into a map in a sorted format of word and value, the map being type string and uint32.
- *reduceFile:* This method accepts a folderpath and a filename, it basically say if the method AddFileContenetsToSorter does not have have a file being processed in it, then it was unable to add the file into the sorter.
- *proofDLLWork:* Method used to verify the functionality of the DLL at work, to ensure the data is processed dynamically
- *Protected: IsolateWord:* The method isolateWord takes in a string of items, it searches for the first string, and then the second string. Since the first string will be a word, and the second string will be the key for that word. It does this by finding the start index and the end index of the string, for the purpose of finding the word and the value.
- *Protected: exportResults:* This method basically sends out an export call with the formatted output being pushed into the outputVector, and writes that vector to a file in the ReducerOutputDirectory.

3.3.3 Workflows and algorithms

Use collaboration diagrams and/or sequence diagrams to show the workflows of components/packages/classes inside the component.
Describe algorithms, if possible. An algorithm may be described outside this document, in this case, add the reference to that document.

Reducer DLL Algorithm

As can be seen in figure 6 above, it is the workflow and the linkage for the reducer DLL for the purpose of communicating with the Reducer interface. The source code for the reducerDLL calls methods from the header file DLLReduce.h, this header includes the ReduceLib.h & the framework.h files. The ReduceLib.h file contains all the methods used in the source code for the Reducer. Then the header for the ReduceLib.h links to the Reduceinterface.h, this file contains all the inputs and outputs for the Reducer DLL, such the directories, importing files into the sorter and the export. Since this is setup, the MapReducer workflow can include the ReduceInterface.h, since the main functionality for the workflow is to manage the flow of inputs and outputs.

3.4 Final Class

3.4.1 Component interfaces

Essentially, the main functionality of the final class is to fetch the reduced files from the ReducerOutputDirectory, combine all contents into one single output file and export it to the finalOuputDirectory.

- *setParameters:* sets up all the directories needed for the purpose of inputting and outputting the data from the final class

● *ReducerOutputDirectory:* holds the input data, or in other words the output from the reducer DLL.
● *finalOutputDirectory:* Will hold the data exported after the final class executes
● *mergeFromReduce:* This the main function within the final class solely used for inputting the data and processing it so that all items are pushed onto one file.

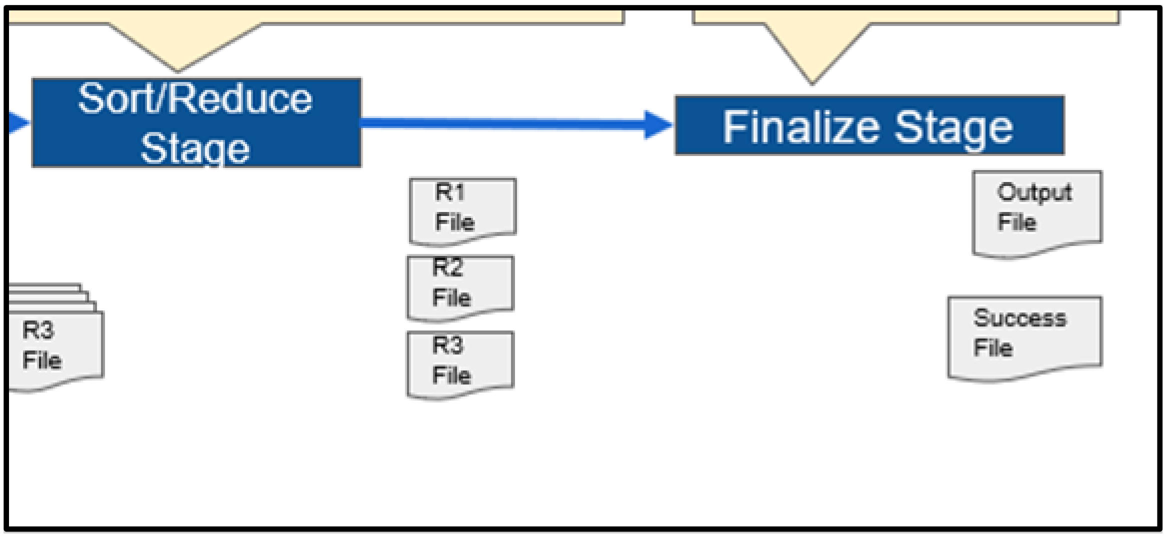


Fig. 7. Final exports will include a single output file & a success file

3.4.2 Component design description

No threads were used in the final class, and since the final class is not a DLL it had to be placed inside the workflow directory instead of having a directory of its own; the true intention is to handle the data and threads dynamically.

● # Of reducer thread: 0
● # Of mapper threads: 0
● ReducerOutputDirectory: stores all reduced and sorted files
● finalOutputDirectory: stores one single merged file

Methods

● *setParameters:* It is crucial to set up the directories for reducer DLL and the final class, since the the output from the reducer will be stored in the ReducerOutputDirectory for it to be fetched by the final class, for the purpose of merging and packaging the contents into one file. The output from the final class will be exported to finalOutputDirectory.
● *mergeFromReduce:* This the main function within the final class since it gets the list of reduced files, stores the files inside filelist vector. Then it reads the files one by one to get content from the file and stores it inside a stringList vector. Else if no files exist in the reducer directory or the fileList vector, then it cannot read any file presently. A final output vector is declared and assigned to write the vector to a file into the finalOutputDirectory.
● *mapToVectorConversion:* A map of string is formatted to separate the outputpairs, it stores it into a string and pushes the formatted results into the finalOutputVector.

- *IsolateWord:* The method isolateWord takes in a string of items, it searches for the first string, and then the second string. Since the first string will be a word, and the second string will be the key for that word. It does this by finding the start index and the end index of the string, for the purpose of finding the word and the value.
- *updateMapWithLine*: Using the results from IsolateWord method, it stores the result into a map of type string and uint32.
- *ReducerOutputDirectory:* The ReducerOutputDirectory will hold the output files exported from the reducer DLL, the contents of these files have been sorted and reduced, and they will be fetched by the final class with the intention to merge/join all content into one single output file.
- *finalOutputDirectory:* The finalOuputDirectory will hold the final merged file exported from the final class

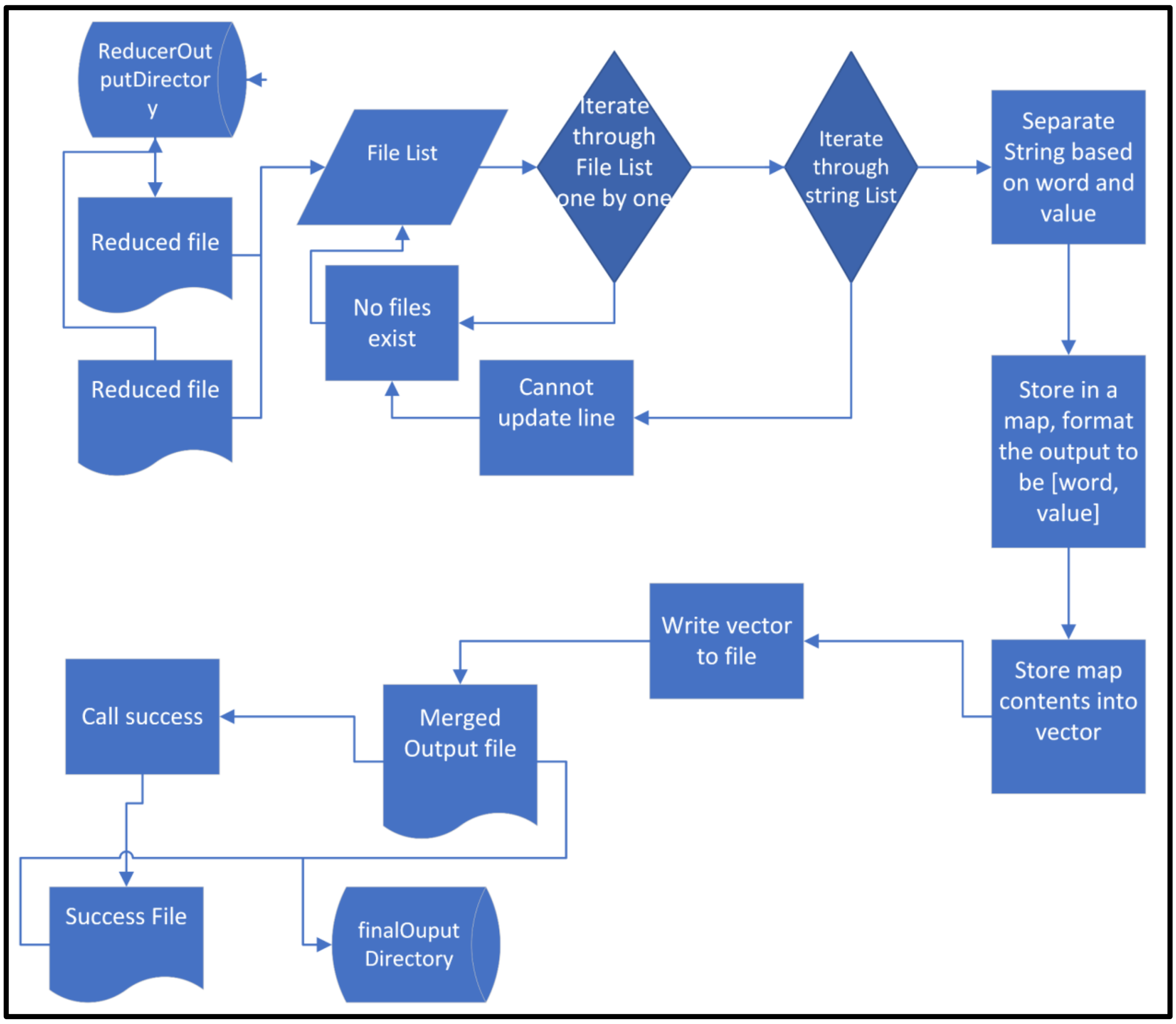

ReducerOutputDirectory
Reduced file
Reduced file
File List
Iterate through File List one by one
Iterate through string List
Separate String based on word and value
No files exist
Cannot update line
Store in a map, format the output to be [word, value]
Write vector to file
Store map contents into vector
Call success
Merged Output file
Success File
finalOuput Directory

Fig. 8. Internal components of the final class

3.4.3 Workflows and algorithms

3.4.3.1 Finalize Algorithm

The final class initializes two directories, the ReducerOutputDirectory for the input data, and the finalOuputDirectory, for exporting the output from the final class. The main method called merge FromReduce get a list of files from the ReducerOutputDirectory to iterate through it to grab lines of string. Later, the string is separated into the word and the value pairs and stores the isolated word and isolated value into a map of type string and uint32. Then, the contents from the map are transferred into a vector, formatting it according to a key value pair order. In the main method call the finalOutputVector is written out to a file onto the finalOutputDirectory.

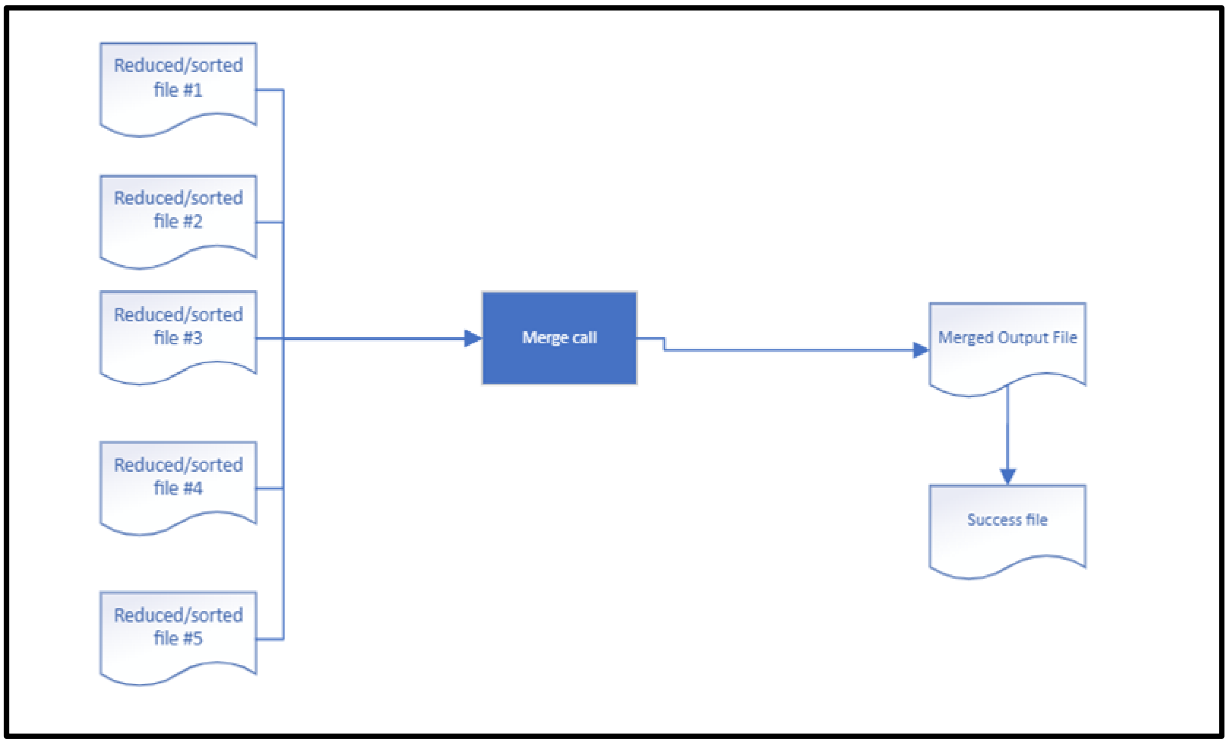


Fig. 9. Inputs and outputs of the Final class