Online Compiler as a Cloud Service

Arjun Datta[1], Arnab Kumar Paul[2]
[1]Research and Development
[2]M.Tech. in Software Engineering
[1]Lexmark International India Pvt. Ltd.Kolkata, India
[2]National Institute of Technology, Rourkela,Odisha, India
[1]arjundat07@gmail.com,[2]arnabkrpaul@gmail.com

***Abstract*: Often there is a need to have many compilers in the same machine to compile programs in different languages at the same time. This paper focuses on solving the problem of storage and portability of compilers. The user without having to install any compiler needs to submit the program into the user interface provided. The controller will then decide which compiler server the program should be assigned to compile, depending on the load of backend compilers. The compiler server will compile and run the program. The output is then given back to the user. The distribution of load by the controller is also tested by calculating the total response time of the programs in both serial and parallel program allocation to compilation tier.**

Keywords—Online Compiler, Cloud Computing, Load Balancing, Multithreaded Programming

I. INTRODUCTION

Cloud Computing is computing that involves a large number of computers connected through a communication network such as the internet, similar to utility computing. [1]

The International Telecommunication Union (ITU) defines 'cloud service' as 'a service that is delivered and consumed on demand at any time, through any access network, using any connected devices using cloud computing technologies.' Cloud Service is further classified into Cloud Software as a Service (SaaS), Communications as a Service (CaaS), Cloud Platform as a Service (PaaS), Cloud infrastructure as a service (IaaS) and Network as a service (NaaS).

In this paper, we propose Online Compiler as a Software as a Service (SaaS). A compiler transforms source code from a higher level language to a lower, machine level language. This is mainly done in order to create executable files which can then be run in order to execute the program and its instructions. [2]

Section II shows the Compiler Architecture. In Section III, the algorithm is explained through pseudo code. The experimental results are presented in Section IV. Finally, the conclusion is drawn in Section V.

II. COMPILER SYSTEM ARCHITECTURE

The online compiler provides service for compiling programs written in either C, C++ or Java. The user need not have a compiler installed in his system. He has to just submit the program to the user interface provided by either typing the code in the text box provided or uploading the text file. The user will get the output after compilation. If compilation is not successful, the errors are shown else the output is given.

The architecture of the online compiler is divided into 3 tiers – (a) User Interface Tier, (b) Controller Tier and (c) Compilation Tier.

A. USER INTERFACE TIER

The User Interface Tier contains the user interface and the database which is implemented using Microsoft SQL Server. The user interface is a web application hosted on the IIS Server which provides the user an interface to submit their programs. The user can submit by typing the code in the area provided or by uploading the code as a file having the required extension (.c for C, .cpp for C++ and .java for Java program).

There are two types of users of the system–

1. *Guest Users*- Guest users are those users who do not register with the system. They are provided with the functionality of writing their programs using any mechanism and receiving their output after a certain amount of time.
2. *Registered Users*- Registered users are those who register with the system. These users are provided with certain added features which are not provided to the regular guest. These features are:

(a) *Viewing of program history*: The registered users are able to view their past activity in the system.

(b) *Viewing of program details*: The registered users are able to view each and every detail of every program they have submitted. This includes codes and outputs as well as compile and run status.

(c) *Longer program execution time*: The registered users have a longer time for which the system waits in order to get the output.

B. CONTROLLER TIER

The Controller Tier manages the interactions between User Interface Tier and Compilation Tier. The Compiler Control Centre is the central part of this tier.

The Compiler Control Centre has 3 parts-

1. Compiler Server Management- It enables the addition of new compilation tier servers, viewing the status of existing compiler servers and removal of compiler servers. It also pings the compiler servers in certain fixed intervals to ensure that all of them are running and active. In case a compiler server fails to respond, it is marked as faulty so that no programs are

ISBN No. 978-1-4799-3914-5/14/$31.00 ©2014 IEEE

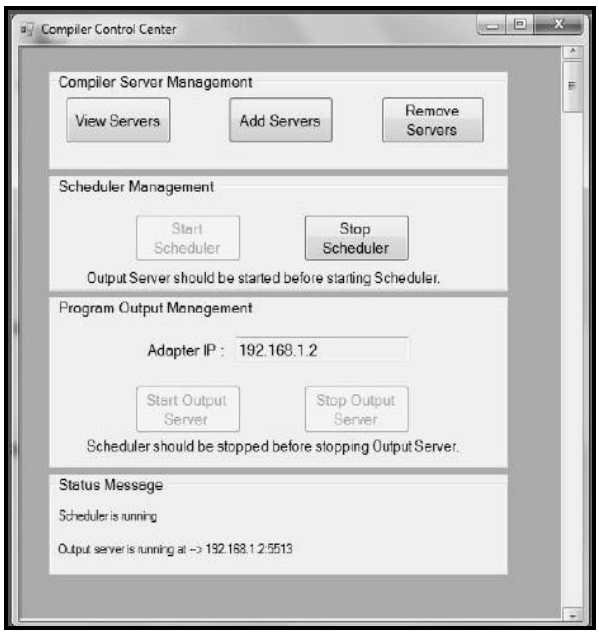

Figure 1 – Compiler Control Centre

assigned to it in future, and existing programs that are routed to that compiler server can be re-routed to other active servers.

2. Scheduler Management- The scheduler fetches the un-compiled program from the database and sends the program data packet to the compilation tier servers.

3. Program Output Management- The "receive output server" receives the compiled program packets sent by the compilation tier servers and stores them in the database. The IIS server in the User Interface Tier fetches the output from the database and sends them to the web-client.

Figure 1 shows the Compiler Control Centre interface.

C. COMPILATION TIER

The compilation tier consists of "n" number of compiler servers which are used to compile and execute the programs. Each compiler server checks its CPU usage and available RAM before accepting a program to compile it, run it and generate the output. If the CPU usage and available RAM are above a pre-defined threshold value then it rejects the program. On successful compilation and execution of a program, the generated output is sent back to an Output Server in the Controller Tier.

Figure 2 shows the complete compiler architecture.

III. ALGORITHM

```
/* Constants representing program states */
  WAITING = 0;
  THREAD_ALLOCATED = 1;
  OUTPUT_RECEIVED = 2;
```

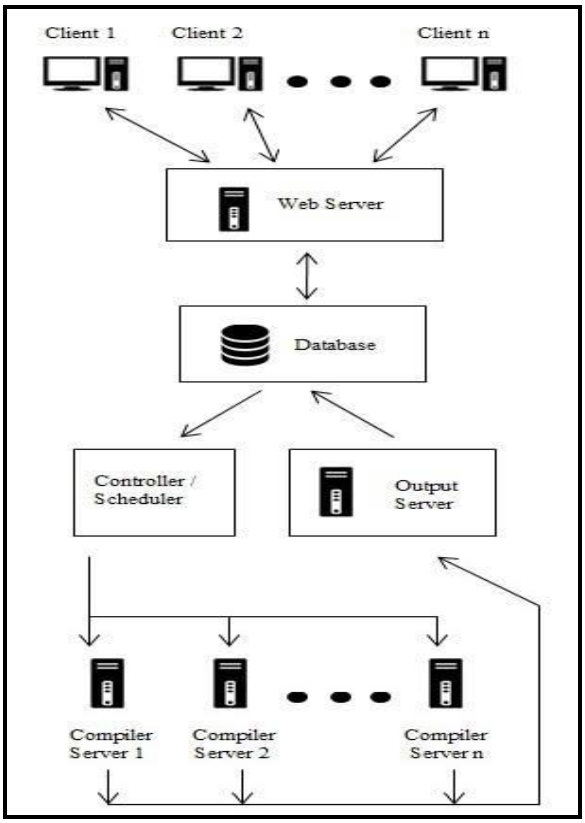

Figure 2 – Online Compiler Architecture

```
/* Constants representing compiler server states */
  STOPPED = 0;
  RUNNNING = 1;

/* Constant representing a dummy server that does not exist */
  DUMMY = "dummy";

/* Constant representing maximum waiting time in minutes for
receiving output of a program after it is scheduled */
    TIME_THRESHOLD = 3;

/* Constant representing maximum number of active threads */
    THREAD_LIMIT = 25;

Start new thread "startProgramMonitor()";
/* Purpose: monitoring programs in the database */

Start new thread "receiveOutput()";
/* Purpose: monitoring output of programs scheduled to a
compiler server */

    startProgramMonitor()
    {
        threadCount = 1;
        while (true){
            programList = getListOfProgramsFromDatabase();
```

```
    for each ( program in programList ){
        if (program.serverAllocated  != DUMMY
            &&currentTime – program.submissionTime>
            TIME_THRESHOLD  && program.state !=
            OUTPUT_RECEIVED){
                program.serverAllocated.pendingPrograms -=1;
                program.serverAllocated = DUMMY;
                program.state = WAITING; }
       else if (program.serverAllocated  == DUMMY
          &&program.state == WAITING){
          while (threadCount>= THREAD_LIMIT){
              /* Do nothing and wait */
          }
          Start new thread "allocateServer(program)";
          /* Purpose: Allocate a back-end compiler server to
           the program */
          threadCount +=1;
          program.state = THREAD_ALLOCATED;
       }
     }
   }
}

allocateServer ( program )
{
    compilerServerList = getListOfCompilerServers();
    Sort compilerServerList in ascending order of priority;
    for each ( compilerServer in compilerServerList ){
       if (compilerServer.state == RUNNING
          && compilerServer.supportsLanguage
          (program.language)&&program.state !=
          OUTPUT_RECEIVED){
             if (compilerServer.
                 readyToAcceptNewProgram()){
                program.allocatedServer = compilerServer;
                compilerServer.pendingPrograms += 1;
                compilerServer.priority += 1;
                break; // Stop this thread execution
             }
             else /* compilerServer rejects the program due to
                    server overload */
             {
                 compilerServer.priority += 2;
              }
                    }
       }
}

receiveOutput (program)
{
    Update program.output in the Database;
    program.state = OUTPUT_RECEIVED;
    allocatedCompilerServer = program.serverAllocated;
    allocatedCompilerServer.pendingPrograms -= 1;
    allocatedCompilerServer.priority -= 1;
}
```

Explanation

The algorithm is based upon a multithreaded approach. Initially two new threads *startProgramMonitor()* and *receiveOutput(program)* are started. The *startProgramMonitor* thread fetches the list of all programs from the database and checks for various conditions to determine whether a program needs to be sent to a back end compiler server for compilation. There are two possible cases when a program needs to be allocated for compilation.

1. A program which has not yet been allocated to a backend compiler server
2. A program which has been allocated to a backend compiler server but the output for that program has not been generated within a stipulated time. In this case, it is assumed that the previously allocated server is either unreachable or out of order. And therefore, the program needs to be reallocated to a new backend compiler server.

Once a program is identified for allocation to a backend server, a new thread *allocateServer(program)* is started if the current number of threads is within the THREAD_LIMIT constant. If the number of threads has already reached the predefined limit then the execution is suspended till the number of threads falls below the limit.

In the *allocateServer(program)* thread, the list of currently running backend compiler servers is fetched from the database. The servers are then sorted in ascending order of priority using any suitable sorting algorithm. The thread then tries to send the program to the compiler server with the lowest priority. If the compiler server accepts the program then the priority of that server is increased by one. If the server is overloaded and rejects the program, then the priority for that server is increased

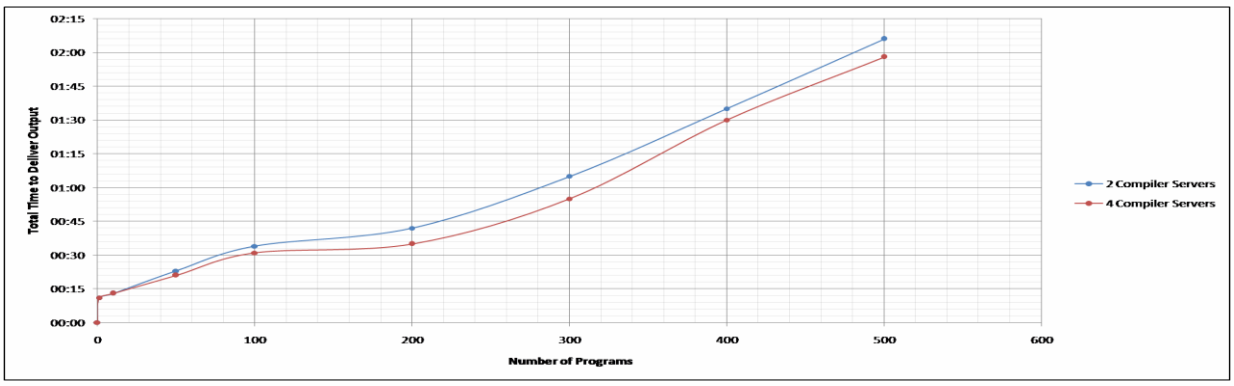

Figure 3 – Total Time to deliver Output using parallel allocation of programs

by two and the thread then tries to allocate the program to the compiler server with the next highest priority. The logic behind increasing the priority of the compiler servers while allocating a program to it, is that during the iteration for the next program, the compiler server which was chosen previously will be having a higher priority and therefore the thread will instead try to allocate to a compiler server with a lower priority which has been allocated lesser number of programs so far and is comparatively less overloaded.

Finally the *receiveOutput(program)* thread is responsible for updating the output of a pre-allocated program in the database. It also reduces the priority of the backend compiler server which sent the output because it's pending jobs is decreased. Therefore, that particular compiler server will have a better rank in the priority list when the next program is processed for allocation.

IV. Experimental Results

The results are computed using a simple Java program of

```
public class DelayExample{
  public static void main(String[] args){
    Thread.sleep(10000);
    System.out.println("Simple Delay Test");
  }
}
```

10seconds delay as follows :-

The result is shown in Figure 3 which plots a graph of Total Time to deliver Output against The number of Programs submitted.

The compilers used to compile the programs are – MinGW for C and C++, jdk 1.6 for Java.

The configurations of systems used during the experiment are –

Server end configuration
Intel Core i7 CPU @ 3GHz
8 GB RAM
500 GB HDD
Windows 7 Home Premium

Compilation tier configuration
Intel Core 2 Duo
2 GB RAM
320 GB HDD
Windows XP-SP3

As shown in Figure 3, the blue line indicates the total time to deliver output with two Compiler Servers and the red line shows the time using four Compiler Servers. In case of parallel allocation of programs, the total time to deliver output for 500 programs is 2.01 minutes or 121 seconds using 4 compiler servers. If serial allocation of programs had been implemented, then assuming that it takes 11 seconds for 1 delay program to generate output, the total time to deliver the output for 500 programs would be 5500 seconds. Therefore the performance is increased by a factor of 5500÷121 or 45.45.

Also, the total time to deliver output of 500 programs using 2 compiler servers is 2.09 minutes or 129 seconds. Therefore the performance is increased by a factor of 5500÷129 or 42.64.

Similarly, we can measure the performance boost for 4 compiler servers over 2 compiler servers. The performance is increased by a factor of 129÷121 or 1.06.

V. Conclusion

The cloud model described in this paper could be implemented in scenarios where a large number of users will need to compile their programs and view the output in minimal time. An example of such a scenario is online coding contests where the contestants need to submit their programs to a central server for evaluation. The number of backend compiler servers could be adjusted according to the expected number of users of the system. As explained in Section IV through the graph in Figure 3, increasing the number of backend compiler servers results in considerable performance improvement.

References

[1] Mariana Carroll, Paula Kotzé, Alta van der Merwe (2012). "Securing Virtual and Cloud Environments". In I. Ivanovet al. Cloud Computing and Services Science, Service Science: Research and Innovations in the Service Economy. Springer Science+Business Media.

[2] AamirNizam Ansari, SiddharthPatil, ArundhatiNavada, AdityaPeshave, VenkateshBorole , "Online C/C++ Compiler using Cloud Computing", Multimedia Technology (ICMT), July 2011 International Conference, pp. 3591-3594.

[3] Software & Information Industry Association, "Software as a Service: Strategic Backgrounder", February 2001

[4] M. Ambrust, A.Fox et al " Above the Clouds: A Berkeley View Of Cloud Computing", EECS Department, University Of California, Berkeley, Technical Report No. UCB/EECB-2009-28, February 10, 2009

[5] European Network and Information Security Agency (ENISA), Cloud Computing: Benefits, Risks and Recommendations for Information Security, Nov. 2009; www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-risk-assessment/at_download/fullReport.

[6] Wikipedia, "Cloud computing," http://en.wikipedia.org/wiki/Cloud_computing.