DISTRIBUTER – The Distributed System for Efficient Execution of Parallel Programs

Domagoj Matijević, Petar Taler
Department of Mathematics, J. J. Strossmayer University of Osijek
Trg Lj. Gaja 6, HR-31000 Osijek, Croatia
Phone: ++385 31 224 800 Fax: ++385 31 224 801 E-mails: domagoj@mathos.hr, petar@mathos.hr
Goran Martinović
Faculty of Electrical Engineering, J. J. Strossmayer University of Osijek
Kneza Trpimira 2B, HR-31000 Osijek, Croatia
Phone: ++385 31 224 600 Fax: ++385 31 224 605 E-mail: goran.martinovic@etfos.hr

Abstract – We developed the distributed system called "Distributer" in order to make use of millions of unused CPU cycles inside the LAN. The system is based on server-client architecture and the interaction with the system is implemented via web interface on the server side. Clients connect to server and periodically update their status based on which server distributes and sends jobs and at the end collects results.

Our system is easy-to-use and easy-to-install and has the simple centralized management of applications needed on client side for jobs execution. We keep all applications on one central repository that is shared to all clients inside LAN with the help of network file system protocol (NFS).

I. INTRODUCTION

A typical distributed system consists of multiple autonomous computers that communicate through a computer network [1]. The idea that emerged in mid-1990s, known as public-resource computing, was to make use of a distributed system in order to enhance the number of FLOPS needed for large-scale computational projects.

One of the probably most famous public large-scale project was SETI@home [2], launched in 1999, which has attracted millions of participants worldwide. SETI@home now runs on about 1 million computers, providing a sustained processing rate of over 70 TeraFLOPS, and using the BOINC distributed platform ([3], [4]). BOINC stands for Berkeley Open Infrastructure for Network Computing and is probably the most famous open-source platform for public-resource distributed computing. It is being developed at U.C. Berkeley Spaces Sciences Laboratory by the group that developed and continues to operate SETI@home.

As the contrast to public-resource computing, companies like Google Inc. has developed their own patented programming model MapReduce [5] for processing and generating large data sets in a distributed fashion but within their corporate firewalls. The framework is inspired by *map* and *reduce* functions commonly used in functional programming although their purpose in the MapReduce framework is not the same as their original forms. MapReduce allows for a particular, stylized way of programming that makes it easy to split work among many machines. The basic idea is to divide the job into two parts: a Map, and a Reduce. Map takes the problem, splits it into sub-parts, and sends the sub-parts to different machines such that all the pieces run at the same time. Reduce takes the results from the sub-parts and combines them back together to get a single answer. However, as it seems that MapReduce perfectly fits to specialized needs of the company like Google, others may find their computations more difficult to implement in a MapReduce framework.

As the need for large processing power emerged rapidly from the small research groups at our Department, we developed our own system for distributed computing, called the Distributer. As a result, the system itself is mainly targeting small scale research projects and is expected to run inside the LAN. The running research project that currently extensively uses the Distributer is concerned with the dampers' and viscosity optimization in mechanical systems (see [6], [7]). Their efficient algorithm optimizes simultaneously the dampers' positions and their viscosities and is highly parallelizable. For example with the help of the Distributer we managed to compute in real-time the optimal positions of dampers for several real-world examples that normally would take months of computation time and only with the help of normal desktop computers.

The current implementation of the Distributer accommodates several different applications and it provides flexible and scalable mechanism for distributing data. The clients are expected to frequently turn off or disconnect from the network. Existing applications in common languages (C, C++, FORTRAN, Python) or some commercial products, such as Wolfram's Mathematica and MathWorks' Matlab, can run as Distributer applications, all shared to clients from a single central place, i.e. *Application Central Repository* (ACR). The current implementation supports jobs as submitted in a form of a single input file (e.g. input file for some of supported applications or a coordinating script).

The system is easy-to-install from both server and client side and only asks for more expertise in case of setting up the license agreements in case of commercial software such as Matlab or Mathematica. Once the Distributer is installed, users even with a very limited computer expertise can easily submit jobs via the central web-interface that directly communicate with the server side of the system.

The Distributer is originally meant for solving problems that can be fragmented into independent parts. The current architecture of the Distributer expects no real

communication between clients. Due to the inability of clients' intercommunication it is necessary that jobs are mutually independent, i.e. each job has to be self-contained. In that way, each client can independently process assigned job, regardless of the state of other jobs.

It is worth mentioning that the Distributer could be easily integrated with any existing software solution that supports multi-user interaction (such as most of the web-based applications). In that way users could distribute millions of user-jobs away from the centralized machine that can continue to operate smoothly running the interface application to the users.

II. The Distributer

The system can be described through the three main components that put together make the system as one:

- Web Interface and the Server component;
- Client component;
- Application Central Repository (ACR).

As shown in Fig.1, the web-interface is to provide the anywhere and anytime user-friendly access to the system. Users can easily submit their jobs via web-interface as a simple upload of the input files for some of supported applications or the upload of the coordinating script inside which the user defines the different input parameters for functions that can run in parallel.

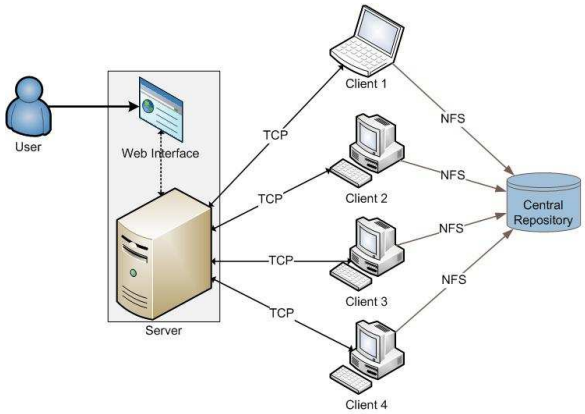

Fig. 1. Components of Distributer

For each job that is being submitted user must specify hard margin for the time they expect the job to finish (*time-to-live TTL*), as well as the expected memory requirement. The idea behind TTL is to protect our clients from getting stuck due to badly written jobs (never ending jobs, infinite loops etc.). User has an ability to update the TTL value, in case he realizes that job will not finish within the expected given time limit. During the running time of the job user can always terminate the execution explicitly by a simple mouse click.

Once the jobs are submitted there are classified into three different categories; i.e. jobs that are in the state of execution, finished or jobs that are still waiting to be executed and they all will appear in a single queue marked with the appropriate flag identifying clearly the category they momentarily belong to (see Fig.2.). The queue of jobs is shown to the user and user has full control over it, such as aborting the execution of the job at any time, removing the job completely from the queue, or resubmitting the job to the queue.

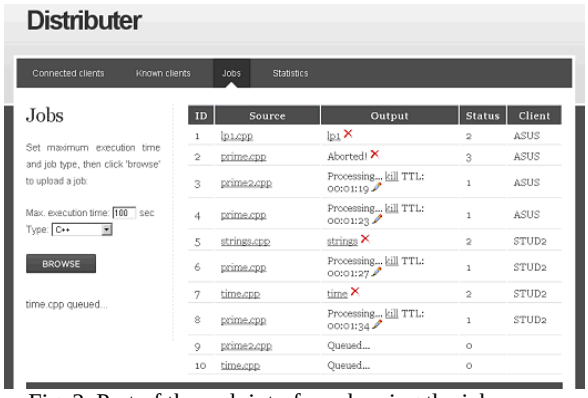

Fig. 2. Part of the web interface showing the job queue.

The efficient *scheduling policy* on the server side takes care that all jobs find their ways to the most suitable clients. Note that the quality of the scheduling is closely related with the monitoring process. The task of the monitoring is to collect and keep updated the data about the client computers that are at disposal for job execution.

Every client that connects to the server for the first time will execute benchmark program that produces its FLOPS (Floating point Operations per Second) value. Once the client made itself available to the system, its current status of available resources is being periodically sent and maintained on the server side, based on which server chooses the most appropriate client for the new job.

Once the job is assigned to the client, the adequate application is loaded from the ACR and the computation starts. During the time of the computation, user can see which client is doing the computation and monitor the remaining TTL.

When the execution of the job successfully finishes, results of the computation are returned to the server that pass the result back to the web-interface. The results can be seen by user directly in the browser or downloaded to local disk as a file. When user reviews or downloads the results, he can delete the job and all its associated data from the server.

In the following sections we present the implementation details of our system in more depth.

A. Server

Server is the key part of the Distributer system. It is logically divided into two components: monitoring and scheduling. Monitoring holds the list of currently connected clients and detailed data about them at any time. On the other hand, the scheduling component of the server maintains the list of user jobs that is waiting for the execution, currently being executed, or whose execution is completed. The main role of the scheduling component is to decide which client gets which job, to send the job to the client along with the instructions for its execution, and to

receive job results and present it back to the user. Hence, server's tasks altogether can be grouped as:

- Maintaining list of currently connected clients and their resources,
- Checking job queue, choosing the best client, sending the job and instructions for its execution,
- Acquiring and processing user commands from the web interface,
- Processing the newly arrived jobs,
- Collecting the results of finished jobs from the clients, and
- Controlling of job execution (aborting jobs, changing *time-to-live*)

DATA STRUCTURES

Key data structures of the server are job queues and the list of connected clients.

Three **job queues** are maintained, one for waiting, active and finished jobs. Every submitted job passes through these three queues and stays in the last one until user deletes it. Using these queues server keeps track of all relevant data for every submitted job.

List of connected clients contains data regarding every client currently registered on the server (state of resources, description of client's capabilities etc.).

Since the server application and web interface run as a two different independent processes, server's variables can't be seen from the web interface and vice versa. Therefore, list of connected clients and job queues are simultaneously being written into database on a hard drive, so user can see them and access them from the web interface. There are three tables in the database: *current_clients*, which is the copy of server's list of connected clients, *job_queue***,** which combines three job queues and *known_clients*. Server holds information about every client that ever registered. Since that data needs to be stored even if server quits (due to computer shutdown, power loss, etc.), it is written in the *known_clients* database table.

SERVER THREADS

In order to keep server run things in parallel, multithreaded architecture is used in its design. Server simultaneously runs several program threads that share access to the common data structures. Fig.3. shows threads of server having *n* clients registered.

PipeListener, *Scheduler* and *ResultCollector* threads are continuously active.

New instance of **Monitoring** thread is created whenever a new client registers. Main purpose of this thread is to periodically receive client's status reports (CPU utilization and amount of free memory). Based on these reports, client's data in the list of connected clients, as well as the associated database table, is kept updated. If the connection between server and client breaks, *Monitoring* thread triggers deregistration procedure for that client and its *Monitoring* thread is destroyed.

Scenario shown on Fig.3. implies server with *n* registered clients, of which *Client_n* is chosen for the job execution. Server's *Scheduler* thread sends the job to the client, while the *ResultCollector* thread retrieves the result.

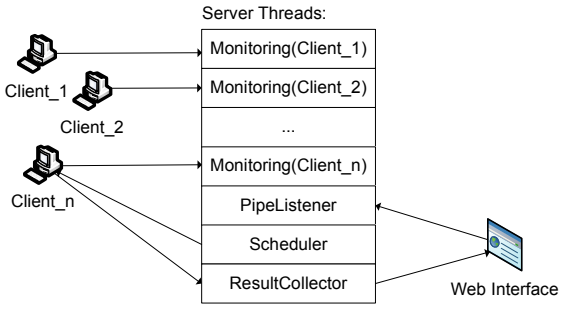

Fig. 3. Server's threads

PipeListener thread serves for the communication between web interface (i.e. user) and server. This communication is implemented via the *pipe* file on server's hard drive. Pipe file, sometimes called *anonymous pipe*, is a simplex FIFO communication channel that may be used for one-way interprocess communication. An implementation is often integrated into the operating system's file IO subsystem.

Whenever user submits a new job, changes the TTL of a running job, terminates it, or deletes a finished job from the queue, appropriate command is written to the pipe file. *PipeListener* thread constantly monitors that file and takes adequate action – triggers corresponding function (*JobAccepter(), UpdateTTL(), JobKiller()* or *JobDeleter()*).

Scheduler thread is the thread implementing the scheduling policy, i.e. it is in charge of choosing the best client for execution of a job that is waiting. It periodically checks the queue of waiting jobs and, if it finds one or more jobs there, chooses the one with the highest priority. In the current implementation, priority is equivalent to the time of job submission such that the job that is waiting longer has higher priority. When choosing the most appropriate client, *scheduler* is guided by clients' FLOPS value. Using this value server maintains the sorted list of connected clients. The most powerful client is chosen and checked if it meets the conditions required (e.g. enough free RAM or unused CPU cycles). If those conditions are met, that client is chosen. Otherwise, next client is tested. That procedure gets repeated until the suitable client is found. If there is not such client currently, job stays in the queue of waiting jobs until some of the existing clients with the suitable needs gets free, or the new adequate client registers to the system. After the *scheduler* sends a job to client, it moves it from the queue of waiting jobs to the queue of active jobs.

ResultCollector thread keeps all the time the TCP connection waiting for the connections from the clients that finished their computations and would like to send job results back to the user. Client which has initiated such connection sends the identifier of the finished job and its results. Server creates a file on the disk and writes received job results into it. Finally, respective job gets transferred to the queue of finished jobs.

Fig.4. shows architecture of the server with marked communication with client and web interface.

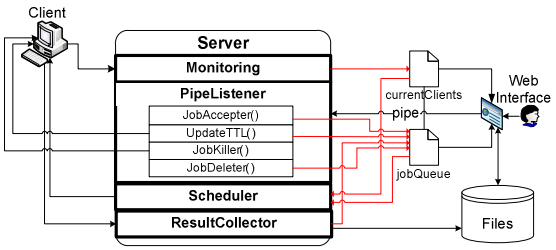

Fig. 4. Architecture of the server

B. Client

Computers connect to the Distributer system using the client application. Its primary purpose is receiving user jobs, running it and sending the results back to the server. It is important to mention here that all applications needed for the job execution are located on the ACR (Application Central Repository), which is shared to all clients. Clients execute those applications via network file system, and as a result there is no need to have custom applications installed on the clients.

Tasks of the client application are:

- Registering to the server and periodically reporting current state of client's resources,
- Acquiring and processing server's commands,
- Receiving new jobs from the server,
- Executing jobs and sending the results to the server,
- Aborting job execution upon expiration of *time-to-live*,
- Changing *time-to-live* period in case of user's command.

DATA STRUCTURES

Design of the client application allows each client to simultaneously execute several jobs (depending on a number of its CPU cores). Therefore, client needs to maintain the **list of active jobs**. Every job currently being executed on the client has a record in that list. When the job execution is completed (successfully, due to *time-to-live* expiration, or due to user's termination) job's record gets deleted.

CLIENT THREADS

As well as the server, the client is designed as multithreaded application. Fig.5. shows client's threads and their communication with the server.

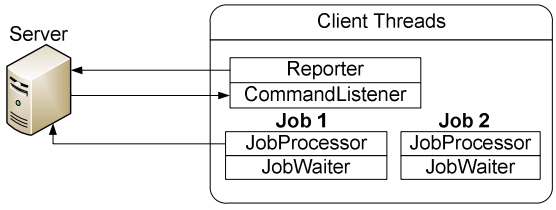

Fig. 5. Client's threads

Reporter and *CommandListener* threads are active all the time, while one instance of *JobProcessor* and *JobWaiter* threads exists per each job client is currently executing. Fig.5. illustrates a client executing two jobs.

Reporter thread creates the new connection to the server and registers on it. Later on, *Reporter* thread will periodically gather state of client's resources (CPU utilization and amount of free memory) and report it to the server.

CommandListener thread has a purpose of receiving commands from the server and taking adequate actions. Using this mechanism server may send a new job to the client, order the client to change the TTL value for a running job, or terminate its execution.

Execution of each job is separated into a distinct **JobProcessor** thread. This thread takes care of running a job until it is finished (successfully, because of TTL expiration, or by user termination). Finally, *JobProcessor* contacts server's *ResultCollector* thread and sends the results to it.

Upon starting each job, a new **JobWaiter** thread is started. This thread has a task to decrement TTL value for that job every second, as long it is active. If job's TTL descends to zero and its execution is still not finished, *JobWaiter* terminates that job.

Architecture of the client application as well as communication with the server is shown on Fig.6.

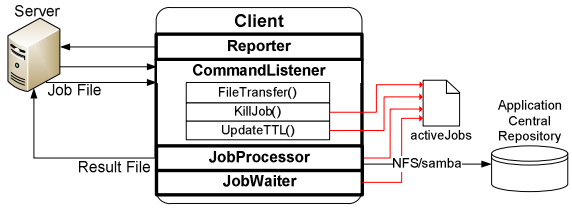

Fig. 6. Architecture of the client

C. Implementation

Client, as well as server application is programmed in Python 2.6 high level programming language. In current implementation, server runs under Linux operating system, but it is possible to port the application to Windows OS with minimum effort, if needed. Client can be started under Linux and Windows operating systems. Clients running Linux access Application Central Repository using Network File System, therefore NFS share should be configured at each client. Windows clients use SMB/CIFS protocol for that purpose, which is enabled by default in operating system.

Web interface is created using PHP5 script language, with extensive usage of JavaScript's jQuery library and AJAX. These techniques allowed us to build a dynamic web application.

Both server and web interface needs to access the database. We have chosen MySQL database engine for that purpose, because of its reliability and good integration with PHP, as well as Python.

III. CONCLUSION

We demonstrated the easy-to-use and easy-to-install implementation of the distributed system for efficient execution of parallel programs.

In near future, we plan to integrate the Distributer into the existing web-based e-learning system Scriptrunner [8]. Scriptrunner is a web application that enables editing and running programs written in various programming languages, as well as collaborative editing of various types of documents. In the current implementation every program that users of Scriptrunner run executes on the single machine running the Scriptrunner itself. By integrating Scriptrunner and Distributer systems together, we would be able to distribute execution of Scriptunner's programs over large number of connected clients. In that way work load generated by execution of user programs would move away from the machine running Scriptrunner, making it able to efficiently handle more users.

For that purposes, we will develop additional mechanisms to make Distributer more reliable and secure system.

REFERENCES

[1] H. Attiya, J. Welch, Distributed Computing: Fundamentals, Simulations, and Advanced Topics, 2nd Edn., Wiley, 2004.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D.Werthimer. "SETI@home: An experiment in public-resource computing". Communications of the ACM, Vol. 45 No. 11, pp. 56-61, November 2002.

[3] D. P. Anderson, BOINC: A System for Public-Resource Computing and Storage, Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04), pp. 4-10, November 8, 2004.

[4] D. P. Anderson, J. McLeod, Local Scheduling for Volunteer Computing, Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pp. 1-8, March 26-30, 2007.

[5] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December 2004.

[6] N. Truhar, Z. Tomljanović, Estimation of the optimal damping for mechanical vibrating systems, Int. J. of Appl. Math. and Mech. 5(5): pp. 14-26, 2009.

[7] N. Truhar, K. Veselić, An efficient method for estimating the optimal dampers' viscosity for linear vibrating systems using Lyapunov equation, SIAM Journal on Matrix Analysis and Applications. 31 (2009), 1; pp. 18-39

[8] B. Mauser, M. Essert, Scriptrunner3, Proceedings of the 26th International Conference on Information Technology Interfaces, Cavtat / Dubrovnik, Croatia, June 7-10, 2004.