

# Distributed Processing Over Loosely Coupled Networks

**A. M. Amin** <ahmedamin@ieee.org>, **A. M. Azab** <aazab@eg.ibm.com>,  
**H. Zakaria** <hammamz@lycos.com>, **A. M. Nabawy**, **A. M. Darwish** <darwish@ieee.org>

## ABSTRACT

Several systems exist that distribute data streams to a single task installed at client machines, while others migrate the whole job to a machine with predetermined processing capabilities. In this paper we propose a hybrid system, DISTPRO, that allows the distribution of any task and/or data stream to participating clients. An experimental client-server approach is discussed and implemented. A centralized management server performs the tasks of dispatching processes to idle clients, receiving results of completed jobs, and maintaining the status of subscribed clients. Satisfactory results have been obtained for tasks such as genetic algorithms and public-key cracking.

*Keywords: Distributed Processing, client-server processing, grid computing*

## 1 INTRODUCTION

The idea of finding a faster solution for problems with massive processing requirements seems very promising for many researchers and enterprises. There has been a constant need for computing power, but the current advancement of processing technology cannot keep pace in handling such large processing requirements efficiently. Distributed processing of such jobs is the field that achieves such required processing power in reasonable time.

The system is designed to distribute heavy-load tasks on idle computers connected to a loosely-coupled network infrastructure (e.g. LAN, the Internet). Its basic structure consists of server and client modules. The server module controls and manages the system and exists on the server machine. It is responsible for dispatching the jobs and receiving the results. The client module, installed on client machines, is responsible for detecting the idle times of the machine. Whenever the machine is idle, the client module receives the job to execute and returns the results to the server. The system is generic. It distributes the task specified and coded by the user (adhering to some rules) regardless of its nature. Another important aspect of the system is its modularity; the task is separate from the server main code. There is no need to rebuild the server module on changing the task. The system uses different class for each purpose, making it easy to modify a certain module of the system. Furthermore, the system works in a multi-threaded fashion by creating different threads for handling different categories of messages.

The rest of this paper is structured in the following way. The next section discusses similar projects implemented previously in the same field, after which we describe the design of the system. In section 4, we present the results of running the system. A conclusion is then presented, followed by suggestions for the future work in this system.

## 2 CURRENTLY IMPLEMENTED SYSTEMS

Currently implemented systems may be categorized into two broad categories: job migration and data distribution. The job migration category [1, 2, 7] aims to migrate the complete task, code and data to a reachable workstation that is capable of executing this job in reasonable time. Efforts in this area target job dispatching and scheduling as well as resource allocation techniques. An example is the Condor project [3] which sends the whole process to a free target machine which possesses the required processing capabilities. The Condor provides a resource management framework which maintains and schedules jobs to resources. Data distribution follows a client server approach where all clients execute the same code on data sent from the server, and results are returned for post-processing on the central server. SETI@Home [4, 5] is a project that allows ordinary Internet users to download the client software, and utilizes their PCs for running specific subsets of the data stream. Another classification may be regarded as distributed processing, where different jobs and corresponding data streams are distributed from a single central server. COSM [6] is an example of such a system which consists of central servers that maintain job specifications and client status. It provides a framework for distribution. Pipelined implementations also exist where the data flow presumes from client to client until the final processing stage has been reached, after which the results are sent for post-processing at the server.

## 3 DISTPRO ARCHITECTURE

### 3.1 Overview

The proposed system provides a framework and protocol for distributing jobs to subscribed client machines. A central server is responsible for maintaining client status and handling new job requests. The application programmer is required to provide external interfaces to aid in the distribution process, rather than having to alter the existing

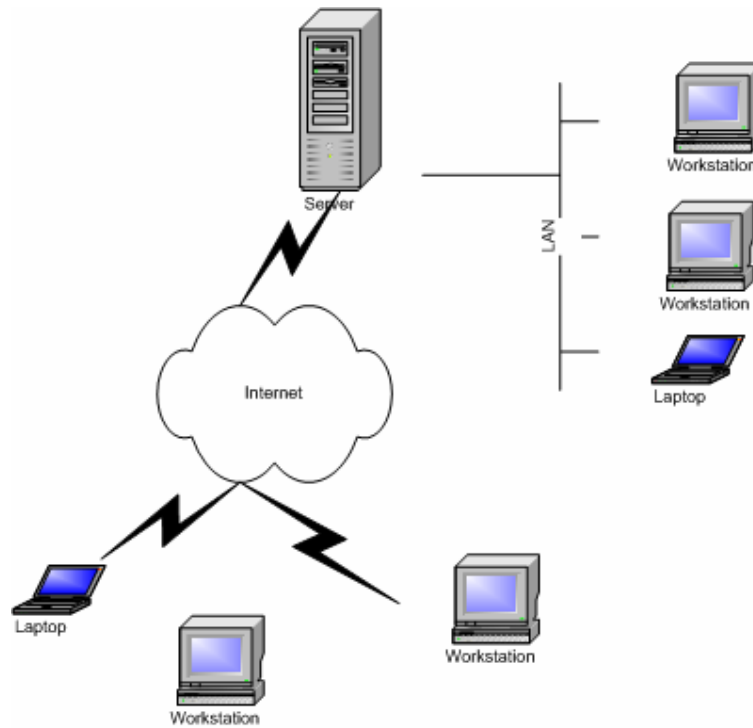


Fig. 1. Architecture Overview

code. The system is designed to work over any loosely coupled network, and several additions to the protocol guarantee robustness to network failures or misbehaving clients. The server is designed in a multi-threaded fashion to increase the performance and speed gained from distribution.

### 3.2 Job Description

The system may run one job up to completion at any one time. Each job may be distributed into any number of *job-parts*. A job-part is code that may be executed independently of other job-parts. Each job-part is given a unique sequence identifier within the job, and they are distributed in sequence order. Our system also defines a *data-part*, which is a stream of data a job-part consumes to produce results. Each data-part is also given a unique identifier. The job hierarchy may be seen in figure 2. For example, if we have the job of matrix multiplication, there is only one job part which is multiplying two arrays together and adding their results. A data-part in this case would consist of two arrays of numbers: a row in the first matrix and a column in the second. Another example is finding the private key in a PK security system. The first job-part would be searching for the prime factors of the public key. A data-part would be a sequence of possible candidate numbers. The second job-part would be to substitute each of the resulting primes into the PK algorithm. The data-parts would be the prime numbers resulting from the first job-part.

### 3.3 Job Distribution

When a machine reports that it is idle, the server checks to see if there are any incomplete jobs. The server maintains a count of job-parts and their status (complete/ incomplete). If there are any incomplete parts, it calls a function provided by the application programmer that retrieves the next logical data-part. If all data-parts are complete, the server checks if the user requires any duplicates, and starts sending duplicates of the current data-parts. When all data-parts and their duplicates have been processed and results have been received, the server increments the job-part counter. When all job-parts are complete, the server marks the job as finished.

When a client completes processing, it establishes a connection with the server and transfers the results file. The server calls a method implemented by the application programmer that handles these results. Users may choose to simply store these files or add extra post processing. They may also handle duplicate results returned by different clients by performing binary matching or thresholding values. The actual technique is left to the application programmer to maintain its generic property.

### 3.4 Client

The system is built on a client-server model. Client status is always maintained in the server database. When a job is initiated, the server polls the idle machines and sends them

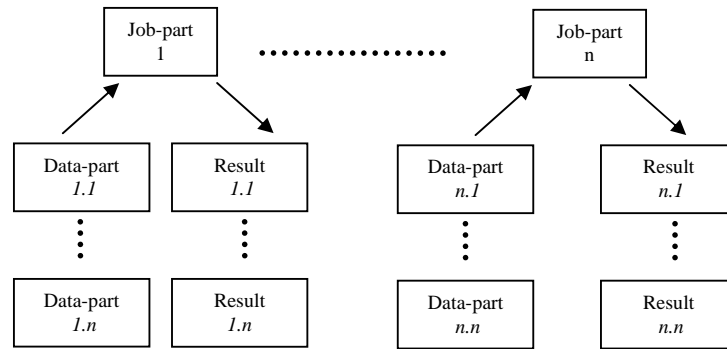


Fig. 2. Job Hierarchy

the job part code and data part. A protocol has been developed that handles synchronization and transfer between the server and participating clients in all situations, such as subscription to the system, reporting idle, delivering code and data to the client, and receiving results.

An overview of the client is given in figure 3. The client consists mainly of three main modules, each running in its own thread: check-idle, main controller, and report-alive reporter. The check idle module monitors the client machine processor activity. Once processor activity drops below a certain threshold for a specified period of time, the idle module spawns and runs the main controller thread and the check-alive reporter thread. If processor activity due to external activities increases above the threshold (the client machine is in a busy state), the check idle thread pauses the main controller and monitors the processor activity again.

The main controller thread is responsible for the majority of the activities of the client side. Once activated, it sends a MACHINE\_IDLE message to the server. It also sends the latest code version that the client has. If it is the currently dispatched job at the server, then data transfer is initiated between the client and server, otherwise both the code and the data are transferred. This avoids resending the same code to the client more than once, thus reducing overhead and increasing throughput. The code is executed\* in its own process. Upon completion, the controller initiates a result transfer session with the server, and resends the MACHINE\_IDLE request.

Periodically, the server sends a CHECK\_ALIVE beacon message to all clients that are currently in active mode. The client check-alive thread responds with a REPORT\_ALIVE message indicating that it is still running the code and that it is still connected to the server.

\* The client detects whether the code is an executable ready to be run, or in intermediate language form, where it is compiled before execution. This allows the system to work on multiple platforms since the code is translated to machine code at the end platform.

### 3.5 Server

An overview of the server module is shown in figure 4. The server design is based upon a relational database and a message queue. The database stores data such as the subscriber client list and addresses, their specifications, as well as all data regarding the jobs. The message queue is composed of two concurrent threads. The message listener thread listens on a specific port for any messages from clients, and simply enqueues them in a shared message queue. The message handler thread polls the queue for any new messages. The message is identified by a unique number in its header defined by our protocol. Depending on its type, a handler is called to process the request. Another concurrent thread is the check-alive thread, which sends a beacon CHECK\_ALIVE message to all clients and waits for their response. If a client fails to respond then it is rendered as busy, and the job-part/data-part it is assigned is passed to another client.

### 3.6 Application Interface

As described before, the system may handle all types of applications. However, the user must provide certain interfaces to the system. In the current implementation, a simple dynamic link library with the required classes and methods is a flexible and generic approach, since the user may replace the library with another one for a different job. The library should consist of three main classes: Job, Data and Result. The JOB class should implement an interface that returns the next job-part id, and returns -1 when all job-parts are done. This method is called by the server when the results for all data parts for the current job part have been received successfully. The class should also provide a method that takes a job-part id as an argument, and returns a file handler to the code or executable file.

The Data class should also implement a method that returns the next data-part id, given a job-part id as an argument. It should also implement a method that returns a file handler to the data stream given a job part id and a data part id.

The result class is invoked by the server when it receives results from a client machine. A single method is required

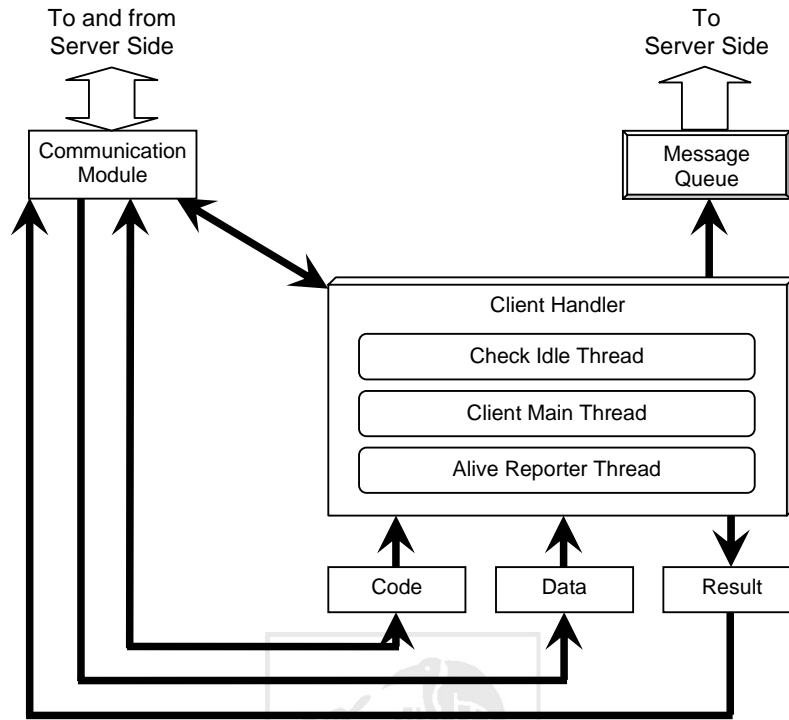


Fig. 3. Client Module

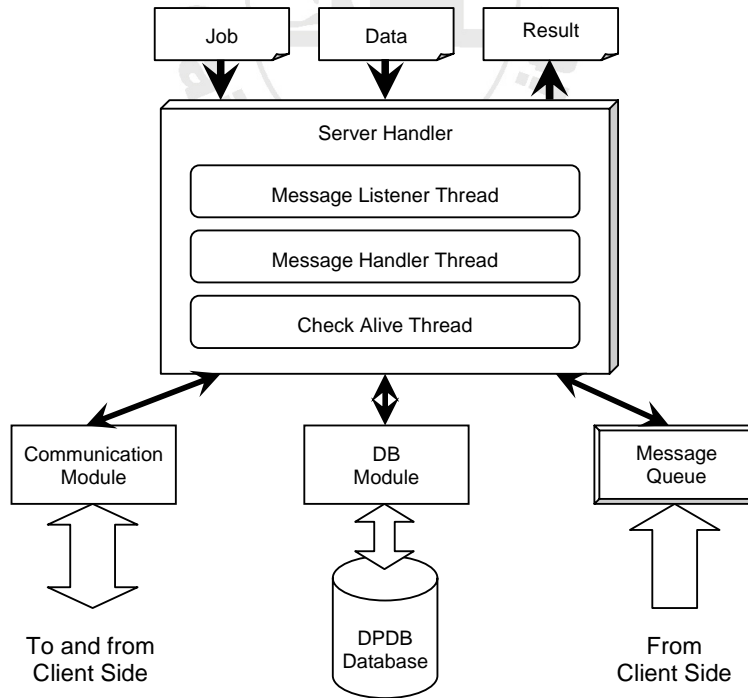


Fig. 4. Server Module

that takes the job-part id, data-part id, duplicate id and a file handler to the results file. What the function does is up to the user. The user may concatenate these files or apply any post processing on them. The user should also take into account that duplicates may be received.

### 3.7 Threaded Server Approach

The server currently polls the message queue in a FIFO manner, and processes each message in turn. Several operations such as transferring large code segments or data streams will block the rest of the queue, thus clients are left idle for longer periods of time. A different approach has been implemented that spawns a thread for each message, thus allowing more clients to be handled concurrently, and blocking is only limited by the resources of the server. System overhead is also reduced, and scalability may be enhanced. This implementation may have the drawback of introducing inconsistencies in the database, so database operations must be protected by assigning each thread a unique id and attaching the id to the transaction.

## 4 RESULTS

The system was tested on a normal LAN with different numbers of subscribed clients. Client machines were chosen to have similar processing capabilities. The job was a computationally intensive application which analyzes the evolving cellular automata problem using genetic algorithms. The job consists of two job parts and each job part is run on nine different data-parts, giving a total of eighteen data-parts. Only one duplicate was required per data-part. The nature of the problem is heuristic, thus there is no predetermined execution time for the problem, and therefore our evaluation was primarily based on average times for each data-part. Total execution time was also recorded to give an estimate of the performance gain by using our system. Table I shows the results and figure 5

provides a graphical representation. The first line shows the duration taken for running the job on a single machine without using our system.

The system's overhead for this job is approximately 87 seconds (1.02%). This may vary due to the heuristic nature of the job. Overhead includes job-part code transfer, data and result stream transfer and connection establishment with client machines. Larger code size and data streams will lead to increased overhead. Network congestion may also lead to delays. However these will be insignificant in computationally intensive applications. Usually the number of duplicated processed is larger than required. This is due to machines remaining idle waiting for unfinished data-parts to be completed for the current job-part. This increase will not be so significant if the number of duplicated per data-part was initially large. The results shown are for an ideal environment. A real life situation may include non identical machines, network congestion, and machines that produce faulty results. This will lead to longer execution times.

## 5 CONCLUSIONS AND FUTURE WORK

In this project we managed to: build the core of the system (both server and client modules), implement a protocol for the job distribution, construct the needed database, and handle most of the expected errors and unexpected faults due to user application runtime errors and communication failures. Results verification is done by dispatching duplicate data-parts to more than one client. Three problems were tested using our system. This lead us to: noticeably reducing the time taken in processing heavy-load tasks, utilizing the not well-used processing power available on the network machines, and providing results verification service. The system may further be enhanced by supporting a pipelined approach where clients send results to other clients representing processing stages.

Table I  
Results

No. of Clients	Total No. of Duplicates processed	Avg. No. of Duplicates per client	Duration (h:m:s)	Duration (sec)	Avg. Job-part Duration (sec)	Avg. Data-part Duration (sec)	Avg. Duplicate Duration (sec)
<b>1- Single</b>	<b>18</b>	<b>18</b>	<b>2:20:28</b>	<b>8428</b>	<b>4214</b>	<b>468</b>	<b>468</b>
1	18	18	2:21:55	8515	4257.5	473	473
2	21	10.5	1:16:22	4582	2291	254.5	218.2
3	24	8	0:58:45	3525	1762.5	195.8	146.9
4	27	6.75	0:49:13	2953	1476.5	164	109
5	30	6	0:42:20	2540	1270	141.1	84.7
6	33	5.5	0:39:26	2366	1183	131.4	71.7

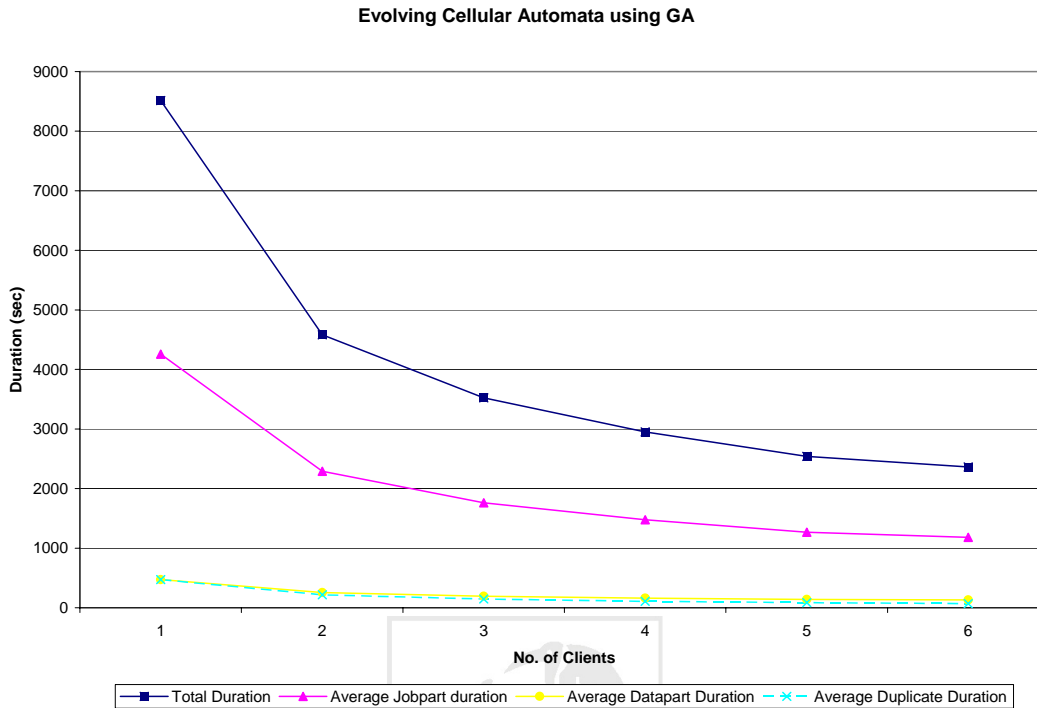


Fig. 5. Results Chart

## 6 REFERENCES

[1] Fields, S., *Hunting for Wasted Computing Power : New Software for Computing Networks Puts Idle PC's to Work*, 1993 Research Sampler, University of Wisconsin-Madison: <http://www.cs.wisc.edu/condor/doc/WiscIdea.html>  
 [2] Basney, J. and Livny, M., *Deploying a High Throughput Computing Cluster*, Department of Computer Sciences, University of Wisconsin Madison, Wisconsin, USA.  
 [3] *Condor Project*: <http://www.cs.wisc.edu/condor/>

[4] *SETI@Home Project*: <http://setiathome.ssl.berkeley.edu/>  
 [5] *SETI@Home*: <http://www.computer.org/cise/articles/seti.htm>  
 [6] *COSM Project*: <http://www.mithral.com/projects/cosm>  
 [7] Stellner G., *CoCheck : Checkpointing and Process Migration for MPI*, Institut für Informatik der Technischen Universität München, Lehrstuhl für Rechnertechnik und Rechnerorganisation, D-80290 München.  
 [8] Forsts, I., Kesselman, C., Tuecke, S., *The Anatomy of the Grid : Enabling Scalable Virtual Organizations*