

Applying Design Patterns in Distributing a Genetic Algorithm Application

Nick Burns Mike Bradley Mei-Ling L. Liu
California Polytechnic State University
Computer Science Department
San Luis Obispo, CA 93407

Abstract --This paper describes an experiment that employs an incremental approach to distribute a genetic algorithm application. The incremental approach allows the application to be initially developed for a standalone system, and then extended to run on multiple systems concurrently. For the extension, design patterns are employed wherever appropriate. The use of design patterns in the experiment is described, and the outcome of the experiment, including quantitative measurements, is presented.

Keywords: design patterns, distributed system, genetic algorithms, Robocode, GALib.

1. Background

A genetic algorithm (GA) [1] is a problem solving method inspired by Darwin's theory of evolution: a problem is solved by an evolutionary process resulting in a best (fittest) solution (survivor). In a GA application, many individuals derive, independently and concurrently, competing solutions to a problem. These solutions are then evaluated for fitness and individuals survive and reproduce based upon their fitness. Eventually, the best solutions emerge after generations of evolution.

The flow of a typical GA simulation is as follows: First, a GA server creates many individuals randomly. Each of these individuals is tested for fitness. Based on their fitness, measured by a fitness function that quantifies the optimality of a solution, the server selects a percentage of the individuals that are allowed to crossover with each other, analogous to gene sharing through reproduction in biological organisms. The crossover between two parents produces offspring, which have a chance of being randomly mutated. A child thus produced is then placed into the population for the next generation, in which it will be evaluated for fitness. The process of selection, crossover, and mutation repeats until the new population is full and the new generation repeats the behavior of the previous generation. After many generations, the individuals are expected to become more adept at solving the problem to which the GA is being applied.

In order for a GA simulation to work well, there needs to be a significant number of individuals within a population, and the simulation needs to be allowed to run for many generations. Furthermore, the simulation will typically need to be run repeatedly while parameters -- such as mutation rate, population size, crossover functions -- are tuned. Thus, a successful GA simulation requires the calculation of the fitness function thousands of times or more. It is therefore critical that the function that performs the calculation of the fitness, called the fitness function, can be executed as speedily as possible.

When a GA simulation is limited to run on a single computer, the GA server is responsible for spawning the individuals as well as calculating the fitness functions of the myriad individuals,

resulting in typically very lengthy runs. Our project, *DistributedGALibRobocode*, is an attempt to distribute the computations of the fitness functions, so that the GA server parcels out the computations to clients running on other computers, thereby accelerating the simulation run.

2. The *DistributedGALibRobocode* Project

Java *GALib* is a Genetic Algorithms Library for Java [4]. *Robocode* is an easy-to-use robotics battle simulator developed by IBM [5]. *Robocode* provides an environment in which automated robot tanks, or bots, created by programmers, can compete against each other.

The *GALibRobocode*, originally developed at Cal Poly as a class project, is a system that combines the Java *GALib* and *Robocode* to use genetic algorithm to eliminate inferior bots. A standalone version of *GALibRobocode*, running on a single computer, required 24 hours to complete a relatively simple run for 30 generations with 100 individuals.

Our experiment extends the *GALibRobocode* to a distributed version, termed *DistributedGALibRobocode*. In the experiment, we applied an incremental approach to create the distributed system in these steps:

1. Build a working *GALibRobocode* application that runs on a single computer.
2. Partition the *GALibRobocode* application into a server and a client where the server is responsible for managing the generations of individuals and the client is responsible for computing the fitness functions.
3. Extend the application to distribute multiple clients, so that the clients run independently on separate systems. Each client interacts with the server to compute fitness functions at the request of the servers.
4. Add provisions for fault tolerance to the application.

3. Challenges for the Project

Although the underlying concept is simple, *DistributedGALibRobocode* raises a number of software engineering challenges, described below.

Synchronization and Concurrency control: *GALib* keeps track of all individuals of each generation and their fitness values in a set of arrays. In the standalone version, the computations of the fitness values are performed serially by the server alone. In the distributed version, the computations are performed and returned to the server by the clients concurrently. Synchronization and concurrency control are required whenever a single data object is updated by multiple sources concurrently.

Performance: The main objective of distributing the computations of the fitness functions is to improve the performance of the GA simulation. Ideally, the runtime would improve linearly, so that the runtime of the standalone *GALib* will be shortened by n times when distributed to n systems. However, the performance gain is not expected to be linear, due to factors such as network latency and load balancing. The design of *DistributedGALibRobocode* should optimize the performance gained through concurrency.

Reliability: Whereas the distribution of the application is expected to result in performance gain, the tradeoff is that *DistributedGALibRobocode* will be less reliable than the standalone

version, due to the possibility of partial failures. Therefore, fault tolerance measures must be provided so that a GA simulation run can carry on in spite of client failures.

4. Employing Design Patterns

One of the objectives of the experiment is to apply design patterns in the incremental development of *DistributedGALibRobocode*, specifically to the task of extending the standalone *GALibRobocode* to its distributed version.

“A design pattern is a particular form of recording design information such that designs which have worked well in particular situations can be applied again in similar situations in the future by others. [2]” Design patterns are widely accepted in object-oriented software design, and collections of such patterns can be found in numerous publications [3, 4]. Some of the patterns employed in the design of *DistributedGALibRobocode* are described below.

A key design pattern used for extending *GALibRobocode* to its distributed version is the *Half-Sync / Half-Async* pattern [4], a pattern for handling client/server connectivity and job delegation/scheduling. The pattern is employed to decouple asynchronous and synchronous service processing by introducing two intercommunicating layers, one for asynchronous processing, and the other one for synchronous processing. This pattern works well with *GALib*'s client-server design. The client/server portion is the asynchronous part, while the *GALib* processing is the synchronous part: *GALib* processing cannot be asynchronous because a generation must be complete before the algorithm can proceed to the next generation.

The pattern also supports the fault tolerance needed for the application: whenever a client is detected to have failed, the server returns the job assigned to the failed client to the *JobQueue*, thereby allowing another client to take up the work and keep the server running. The adoption of the pattern allows for more flexibility in the rendezvous of the client and server, and results in efficient and fast client-server interaction.

Another design pattern, the *Thread per Connection* model [4], is applied to the interaction between the clients and the server. For each client, the server spawns a separate *ServerThread* dedicated to interacting with the client.

The class diagram presented in Figure 1 illustrates the design of the client and the server.

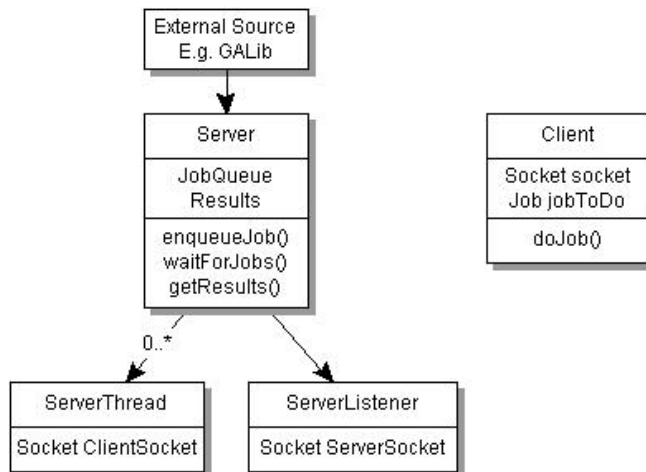


Figure 1: Class Diagram of Server related classes

The *Composite* design pattern [3] was employed to batch the jobs sent from the server to clients. Jobs for computing fitness functions are grouped in a vector in an object sent to each client; the number of jobs placed in each request is based on the number of clients connected to the server. This clumping of jobs helps to reduce the communication overhead between the server and the clients. The composite design also allows load balancing among the clients. For example, jobs could be distributed by the server based on client speeds in addition to the count of clients. Figure 2 presents the class diagram of the classes *Job* and *Jobs*.

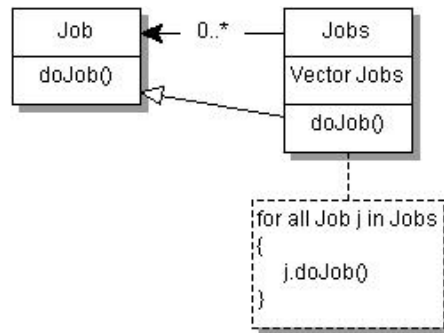


Figure 2: Class Diagram of *Job* and *Jobs*

Another design pattern, the *Singleton*[3], was employed in the design of the client to guarantee that a single instance of the *Robocode* will be created. The creation of a *Robocode* instance, which loads all of the bots and initializes the environment, takes a considerable amount of time.

The *Asynchronous Completion Token* [4] design pattern was incorporated in the *GAJob* class: a completion token is used by a client to return the result of each job. Each *GAJob* object contains the location in the fitness array (on the server side) where the result of a fitness function is to be placed. Thus, when a job is completed, the server receives from the client a completion token and knows exactly where to put the fitness value in the fitness array. Figure 3 illustrates the class diagram of *GAJob*.

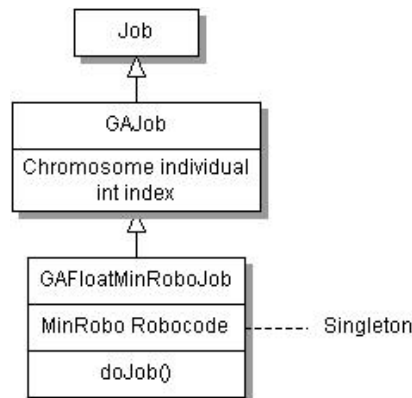


Figure 3: Class Diagram of *GAJob* and *GAFloatMinRoboJob*

The basic interactions of these classes are illustrated in Figure 4. An external source, i.e., *GALib*, fills the *JobQueue* with *Jobs*. Then *ServerThreads* dequeue jobs from the *JobQueue*

and dispatch them to the clients. When completed, a job is placed in a *Result* object and sent back by the client to a *ServerThread*. Each client connects to the *ServerListener* and a *ServerThread* is created to handle the interaction with each client.

Concurrent access to the *JobQueue* and *Results* are handled with Java synchronized blocks: access to *GALib*'s fitness array is serialized. *GALib*'s requirement of a full complete fitness array to start a new generation is handled by simply checking when the *JobQueue* has become empty. When the *JobQueue* is empty, all jobs for fitness computations have been completed and it is then time for the next generation to start. The detection of completion is done by using Java *wait* and *notify* on *JobQueue*.

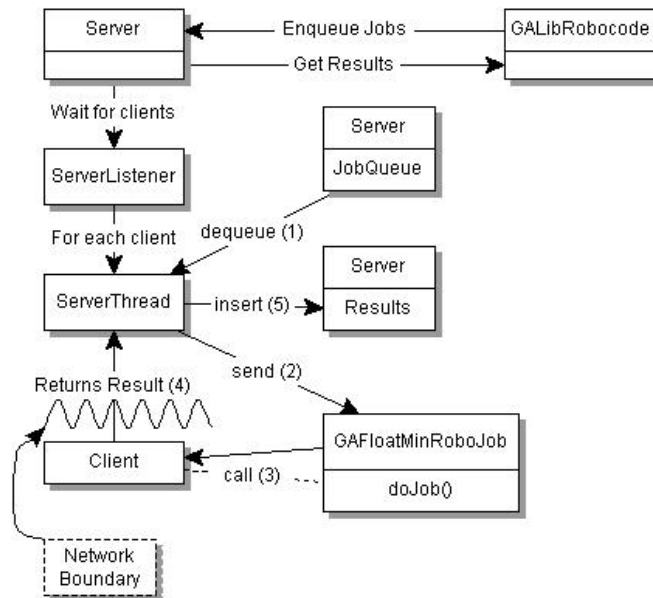


Figure 4: Interaction Diagram

5. Incremental implementation

A main objective of the experiment is to put to test the applicability of the incremental approach whereby a distributed system is developed by (i) building a standalone version of the system, then (ii) modifying the system so that the functionalities are distributed.

First, we developed the fitness function for *Robocode* for use with *GALib*. Then we developed a standalone version of *GALibRobocode*. After the standalone version had been tested to our satisfaction, we focused on making *GALibRobocode* distributed. A generic client/server framework based on the Half-Sync / Half-Async pattern was developed. For testing purpose, stub jobs consisting of a simple time delay were employed for testing.

We then produced a prototype of the distributed *GALibRobocode* system by refactoring the standalone version into a client and a server. The server performs all the *GALib* related functionalities and the clients perform *getFitness* calls at the request of the server. We tested the distributed version by running the server on one computer and the client on several other computers in a laboratory.

6. Experiment Results

We found the use of an incremental design for developing a distributed system satisfactory. The incremental approach allowed us to focus on the functionality of the genetic algorithm simulation in the standalone version, separate from problems related to distributed systems, which were subsequently addressed in the distributed version. The incremental approach also allowed us to develop code that is more generalized and easier to debug.

The main concern of this project was performance. Ideally, the performance should be a one-to-one linear scale-up: by distributing the problem to n processors, the performance should be accelerated n times. Figure 5 summarizes the performance of the distributed system compared to the standalone system when the application was run on a group of 2.4 GHz Dell microcomputers with Java version 1.4.2, Windows XP, and Service Pack 2. On a simulation of five generations, the runtime speedup is almost linear to the computers used in the experiment. The distribution speedup, computed by dividing the runtime of a distributed version over the runtime of the standalone version, is presented in the rightmost column in Figure 5, and illustrated in Figure 6.

Run Type	Computers used	Robocode Rounds	Population size	Generations	Time (seconds)	Distribution Speedup
Stand Alone	1	50	10	5	2247	1.00
Distributed	2	50	10	5	1034	2.17
Distributed	5	50	10	5	451	4.98
Distributed	10	50	10	5	279	8.05

Figure 5: Performance of *GALibRobocode*

7. Conclusions

Our experiment indicates that the performance of genetic algorithm simulation runs can be significantly improved by distributing the computations of the fitness functions over multiple computers. The framework we developed for *DistributeGALibRobocode* can readily be adapted to any *GALib* application: Only the *GAJob* class needs to be customized for the specific fitness function. Indeed a separate experiment at Cal Poly applied a similar distributed framework that uses genetic algorithms to train the agents for a game of Tetris.

In terms of software engineering, our experiment indicates that the incremental approach for developing a distributed system by starting with a standalone version is a workable and satisfactory approach, especially when design patterns are applied to the task of extending the standalone version.

GALibRobocode Distribution Speedup

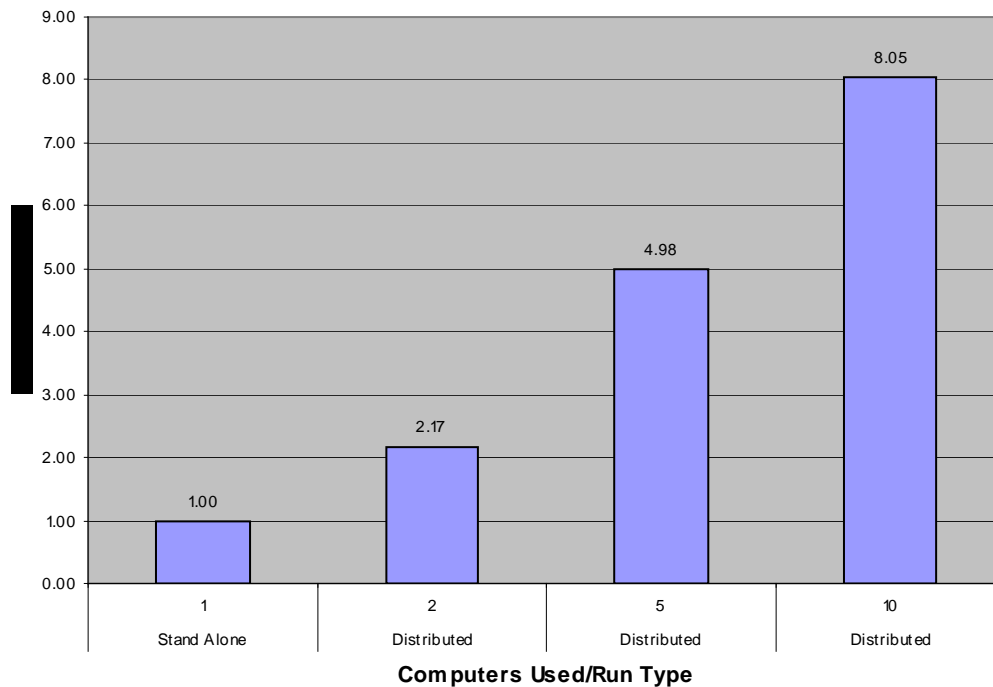


Figure 6: Normalized *GALibRobocode* Distribution Speedup

8. References

1. Marek Obitko. Genetic Algorithms. <http://cs.felk.cvut.cz/~xobitko/ga/>, 1998
2. Kent Beck, et al. Industrial Experience with Design Patterns. Proceedings of the 18th international conference on Software engineering. 1996.
3. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns. Addison-Wesley, Reading, Mass. 1994.
4. Douglas Schmidt, et al. Pattern-Oriented Software Architecture. Volume 2. John Wiley & Sons, Ltd., England. 2000.
5. IBM alphaWorks. Robocode. <http://Robocode.alphaworks.ibm.com/home/home.html>. 2002.
6. Sourceforge.net. Java GALib. <http://sourceforge.net/projects/java-GALib>. 2003.