# Genetic Daemon − A genetic algorithm server

Giancarlo Niccolai

Know How 7

Viale Ranzoni 17 − 20100 Milano, Italy.

Comments at giancarlo@niccolai.ws

**Genetic Daemon is a TCP−IP based server that allows to manage remotely genetic algorithms in a dedicated environment. It's client−server architectures allows data sharing through remote calculation facilities, effectively allowing massive parallel genetic algorithm development. This document describes version 0.1.**

## I. BASIC FEATURES

Genetic Daemon is a server developed to provide research environment (research centers, study groups, universities, firms etc.) with a calculation facilities specialized in evolving genetic algorithms.

An authenticated user can log in to the server, create a genetic agent space (called genetic engine), program and/or modify learning sets, set several control variables and, finally, start the evolution of the genetic algorithms. Genetic engines can then be stopped, managed (adding, saving, restoring or removing agents) and started again. The engine will continue to evolve it's population until a reasonable result is found; the evolution is not bound to any limit, and partial results can be queried and used while the engine is running. Different algorithms can be programmed through a plug−in mechanism in plain ANSI C++. A most interesting feature is the standardization of the genetic agent storage, that allows to share different agents between different Genetic Daemons, allowing a massive parallel processing on the same population.

## II. OPERATIONAL MODEL

### *A. The client−server architecture*

Genetic Daemon operates through a standard TELNET connection as a TCP−IP bound server. A protocol similar to RFC based protocols (SMTP, FTP, POP3 etc.) has been developed to allow both advanced software clients and human users to connect with the server, and interact with it. An administrator (root user) can set up the server, add new user accounts and manage user rights, that range between the "dump" right (being able to see other user's engines status and populations) and the "create" right (being able to create new engines). The administrator can also hot−plug new engine plug−ins, or unplug old ones. This allows Genetic Daemon to be expanded with new genetic algorithms while other algorithms are running.

The server runs in a protected multi−threading environment. Separated threads are provided for each engine run and user connection. An hard error in a single engine, as a division by zero due to untested plug−ins, or an error originating in a user connection, won't affect the main server thread, nor other engines.

This architecture has been specifically chosen to allow genetic algorithm results to be shared across academic or research environments. Engine creators can access their genetic agent populations, retrieve partial or final results, and share results with other researches from any point of their organization, or from outside the organization, if organizational policies allow external connections. Since all engines share the same binary format, a side−effect of this architecture is that users can transfer their engines, or part of a population, in different servers on different organizations.

Future development will allow remote servers to set up a master−slave hierarchy to be set up. This allows to automatically allocate population slices between different servers, alowing an automatic allocation based on resource availability to take place without human intervention. At this moment, this step is in final development stage.

### *B. The evolution model*

Since Genetic Daemon is meant to manage very different genetic algorithms, its basic evolution model is very simple. Evolution is structured in turns. Each turns, the genetic population is handled to an evaluator algorithm that must be provided by the plug−in managing a particular genetic algorithm. The evaluator is responsible to give a premium to each genetic, based upon it fitness in solving the problem at stake. To simplify new genetic algorithms programming, a learning set structure is provided and can be used as is by plug−ins, or reprogrammed on algorithm needs. The premium that the evaluator can give to each single agent is a real number limited in range as a "double" ANSI C++ value.

When the evaluator has given a premium to each agent, the control is handled back to Genetic Daemon, that starts a tournament to decide which agent will survive up to the next turns.

Each agent has an energy value; each turn, a genetic agent consumes exactly one unit of energy. Then, each agent is feed with a quantity of energy in a range between 0 and a valued controlled by the engine creator, called competitiveness. Energy is assigned to the agents in an order determined by the premium (the agent with the higer prmium get energy for first) until the available energy pool is exhausted. The quantity of energy given to an agent is a linear function of it's premium; the maximum energy given is equal to the competitiveness, and is given to the agent or agents that had the higher premium. If an agent depletes all its energy, it will be eliminated.

Next, reproduction is started. New born agents can have a genetic code derived from a single surviving agent (singleton reproduction) or from two separate surviving agents (coupled reproduction). The newly created agent can have a unit of its genetic code randomly added, removed or mutated. The probability that a mutation will occur is determined by variables controlled by the engine creator; the creator can also control the coupled reproduction rate.

Reproduction will create a number of new agents equal to the survived ones, up to a maximum agent count determined by the engine creator.

This turn sequence is repeated until the evaluator declares itself to be satisfied, or until the user stops the evolution.

A more sophisticated competitive algorithm, that will include cooperative co−evolution, is on the project schedule.

### *C. Sharing agents across engines*

One of the most interesting feature of the Genetic Daemon is it's capability to save and restore agent genetic code; this code can be reused in a later stage in the same engine, to perturb the population; it can be sent to another engine in the same server, so to start a parallel evolution, or it can be sent to other servers. This method allows to share the results of a research with other researches, or to fork an evolution so to test different evolution strategies, parameter settings, or to have various chances to reach different optimums in the multidimensional environment.

## III.THE PLUG−IN API

One of the most interesting features of Genetic Daemon is its expandability through a plug−in API, that can be programmed in plain ANSI C++. The genetic algorithm implementer needs only to write some code, ranging from fifty to one thousand program lines, accordingly with the complexity of the algorithm to be implemented.

The API has been kept simple so to allow any programmer with low to medium C++ knowledge to be able to use it. Future versions of Genetic Daemon will provide an XML interpreter that will allow any user to implement basic genetic algorithms, without a prior knowledge of a programming language. Visual front ends to help users in writing their algorithms are also on the schedule of the project.

The plug−in programmer needs to re−implement just the evaluator function, that has the duty to give premiums to genetic agents on the base of a custom fitness function. There is also the possibility to provide new learning set types, that can be dynamic and extract data from a changin environment, and specialized genotypes, i.e. strings, virtual machine instructions etc.

## IV.INCLUDED ALGORITHMS

Genetic Daemon v. 0.1 comes with two plug−ins: one is a simple demonstration and tutorial code, the other is a powerful genetic algorithm capable of analyze complex statistical series.

The tutorial plug−in is called *intseq*, and is a basic genetic algorithm. It evolves agents with integer sequence based genetic code. The aim of the population is to develop an agent that has a genetic code so that the sum of each integer in it's code is equal to a target value chosen by the user.

The second algorithm is called *gfunc*, or *Genetic Function Finder*. This advanced algorithm analyses a given set of statistical data in the search of a non linear, multidimensional function describing the phenomenon that originated the observations. This algorithm not only gives a coherent output on a given set of input using its experience: it outputs a suitable mathematical function, or a model, describing real world phenomena. This is a task commonly delegated to researches, that try to interpret observations and, using common sense, to extrapolate a suitable model: once a model is proposed, it is tested against observations to verify its suitability. The gfunc algorithm automatizes this process through genetic evolution.

The details of this algorithms are found in a separate document [1].

## V. REFERENCES

**[1] "Function Finder" − Giancarlo Niccolai − 2001 − paper submit ted for approval at WCCI2002. Can be retrieved at "http://niccolai.ws/works.php"**