Asynchronous Distributed Genetic Algorithms with Javascript and JSON

Juan Julián Merelo-Guervós, Pedro A. Castillo, JLJ Laredo, A. Mora García, A. Prieto

Abstract— In a connected world, spare CPU cycles are up for grabs, if you only make its obtention easy enough. In this paper we present a distributed evolutionary computation system that uses the computational capabilities of the ubiquituous web browser. Asynchronous Javascript and JSON (Javascript Object Notation, a serialization protocol) allows anybody with a web browser (that is, mostly everybody connected to the Internet) to participate in a genetic algorithm experiment with little effort, or none at all. Since, in this case, computing becomes a social activity and is inherently impredictable, in this paper we will explore the performance of this kind of virtual computer by solving simple problems such as the Royal Road function and analyzing how many machines and evaluations it yields. We will also examine possible performance bottlenecks and how to solve them, and, finally, issue some advice on how to set up this kind of experiments to maximize turnout and, thus, performance. The experiments show that we we can obtain high, and to a certain point, reliable performance from volunteer computing based on AJAJ, with speedups of up to several (averaged) machines.

I. INTRODUCTION

Application–level networks (ALNs), are configured as a set of clients/servers (*servents*) that can provide their spare CPU cycles by means of a downloadable application, establishing a distributed computation network which can provide ad hoc computational power. Some ALN like SETI@Home have been quite successful [1], creating a virtual computer that has procceed a high number of teraflops, while other experiments such as Popular Power (and most others, in fact) have not [2].

The key feature of these application–level networks is the simplicity of use: we believe that the best way to obtain the participation of as many users as possible is to make it as simple as possible. In particular, it will be easier if they do not need to download a special application (such as a screen-saver) to participate, as is needed in BOINC, the core of SETI@Home. For this reason, we are exploring the use of applications that are commonly installed in the user's computer, such as the web browser, which is available even in PDAs and some cellular phones[1]. Moreover, most browsers natively include a JavaScript interpreter [3], [4], [5] or virtual machine. JavaScript is an interpreted language[2], initially proposed by Netscape, and later adopted as an ECMA standard [6], [7], [8], [9]. In this way, most browsers are compatible, at least at a language level (not always at

the level of browser objects, where there exists a reasonable compatibility, anyway). Most browsers also include elements such as a Java virtual machine and a Flash plugin, which, with ActionScript, has more or less the same capabilities. However, there are several disadvantages to these: they might or might not be present (they are not *native*), they are *noisy* in the sense that, since they act as plugins, their execution is always noted by the user, their programs are more heavyweight than simple text code, and, finally, its integration with the browser is more awkward than the seamless integration that JavaScript offers.

By itself, an interpreted languaje is not enough for creating a metacomputer if there is no way to convey information back from the client to the server in a seamless way. The ability to use the virtual machine included in browsers for distributed computing appeared with the XmlHttpRequest object, which allows asynchronous petitions to the server, in what has been called AJAX, Asynchronous JavaScript and XML [10]. AJAX is just one of the possible ways to perform asynchronous client-server communication, the others being AJAJ (Asynchronous Javascript and JSON), and *remoting* using applets or embedded objects. However, it is quite popular, and a wide user base and documentation is available for it, using any of these asynchronous client/server communication protocols. The traditional client/server model becomes then more egalitarian, or closer to a peer to peer model, since a bidirectional communication line appears: the browser can make calls to the server, do some computation and later send the results to the server.

AJAX (and AJAJ, which differ only in the way data is serialized) works as follows: the XmlHttpRequest is provided with a request to the server and a pointer to a *callback* function. The request generates an event, which is asynchronously activated when a reply is received making use of the *callback* function. Following this approach the browser is not locked, providing a way to program applications that are similar to the ones used at the desktop, in the sense that they do not have to wait for the application response to be loaded and rendered on the screen every time a request is made. It also means that a user clicking on the Submit button is no longer needed to initiate communication with the server; any JavaScript thread can do so, with the constraint that the only server they can communicate with is the one that hosts the page in which the script is included. On the other side, this provides a way to use the browser for application level networks that create distributed computing systems, since the request-response loop does not need the user participation in a fashion very similar to any

The authors are with the Departamento de Arquitectura y Tecnología de Computadores, University of Granada (Spain), corresponding email jj@merelo.net

[1]Whose computing power is similar to four-year-old desktop machines

[2]Which has nothing to do with Java, other than the name and its syntactic similarity

other distributed computing application; these ALN can be controlled from the server with any programming language. Of course, it can also be combined with other distributed programming frameworks based on OpenGrid [11] or other distributed computing paradigms.

We previously used Ruby on Rails for making this kind of distributed AJAX application [12]; however, performance and scaling behavior were not too satisfactory mainly for two reasons: the nature of the Ruby on Rails server, which required the setup of a load-balancing server, and the (probably determinant) fact that part of the genetic algorithm was done on the server, with a low degree of paralellism and thus a high impact on performance. Latest experiments (after publication of that paper) yielded a maximum of 40 chromosome evaluation per second[3].

In this paper, following the same concept of distributed evolutionary computation on the browser via AJAX, we have redesigned the application using Perl and PostgreSQL (on the server) and Javascript, as before, on the client, and renamed it AGAJAJ (pronounce it A-gah-yai), which stands for *Asynchronous Genetic Algorithm with Javascript and JSON*; in this case, the genetic algorithm (solving the well known Royal Road problem) is carried out only on the clients, with the server used just for interchange of information among them. We will perform several experiments in which clients donate computing power by just loading a web page to find out what kind of performance we can expect from this kind of setup, from the number of machines that will be made available by their users to the number of evaluations each one of them can perform; in these experiments, we have improved two orders of magnitude the performance achieved in the previous experiments which used Ruby on Rails, and also the number of concurrent machines available to perform them, showing that this kind of setup is ready to take more computing-intensive experiments without the need of an expensive server setup.

This paper follows our group's line of work on distributing evolutionary computation applications, which has already been adapted to several parallel and distributed computing paradigms (for example, Jini [13], JavaSpaces [14], Java with applets [15], service oriented architectures [16] and P2P systems [17], [18]).

Evolutionary computation is quite adequate for this kind of distributed environment for several reasons: it is a population based method, so computation can be distributed among nodes (via distribution of population) in many different ways; besides, some works suggest that there are synergies among evolutionary algorithms and parallelization: isolated populations that are connected only eventually avoid the loss of diversity and produce better solutions in fewer time obtaining, in some cases, superlinear accelerations [19].

Of course, with a suitable work division method, many other algorithms could be adapted to browser-based distributed computation; however, in this paper will solve only genetic algorithms, and concentrate on raw performance,

rather than algorithmic behavior.

The rest of the paper is organized as follows: next section concentrates on the application of volunteer/involuntary computing to evolutionary computation; the setup is described in Section III. Experiments and results are shown in Section IV and discussed in V, along with future lines of work.

II. STATE OF THE ART

So called *volunteer computing* [20], [21] systems are application-level networks set up so that different people can donate CPU cycles for a joint computing effort. The best known project is SETI@home[4], which, from the user's point of view, is a screen-saver which has to be downloaded and installed; when the user's CPU is not busy it performs several signal analysis operations. Some companies related to volunteer computing, such as Popular Power (and others; they are referenced, for example, in [22]) did some experimentation with Java based clients, but none has had commercial success; on the other hand, the SETI@Home program has been open-sourced and extended as the BOINC (Berkeley Open Infrastructure for Network Computing) framework [23]. This kind of volunteer computing has been adapted to evolutionary computation in several ocasions, using frameworks such as DREAM [24], which includes a Java-based virtual machine, GOLEM@Home, *Electric Sheep* [25] and G2-P2P [26]. Both approaches acknowledge that to achieve massive scalability, a peer to peer (P2P) approach is advisable, since it eliminates bottlenecks and single points of failure.

There are mainly two problems in this kind of volunteer networks: first of all, it is important not to abuse the CPU resources of volunteers; secondly, a sufficient number of users is needed in order to be able to do the required computation, which can be a problem on its own if there are too many of them, bringing the network, or at least the solution-collecting node, to its knees. A third problem is that performance prediction is difficult when neither the number of participants nor their individual node performances are known in advance. Finally, fault-tolerance [27] and cheating [28] are also important issues; if the environment is competitive, or any single computation is important, they will have to be taken into account.

In any case, we believe that the best way to obtain a good amount of users is to make it easy for them to participate, using technologies available in their computers, as the browser is. In fact, some suggestions were published (for example, the one from Jim Culbert in his weblog [29], and in some mailing lists), and, besides our own [12], there have been some recent papers and reports on similar setups. For instance, W. Langdon has been running for some time an interactive evolution experiment using Javascript in the browser [30], which was mainly intended for achieving high diversity in a fractal snowflake design than high performance. Even more recently, Klein and Spector [31] present a system based on the Push3 language, which is compiled to JavaScript

---

[3]The published figure was even lower.

[4]See `http://setiathome.berkeley.edu/` for downloading the software and some reports.

Fig. 1. Experiment running on two different browsers (Firefox and Epiphany) in the same machine. User interface is written in Spanish, since in this initial experiment was addressed to audience speaking that language. The colored (or gray-scale) horizontal bar is a graphical representation of the chromosome, with green (darker shade) standing for 1 and red for 0. The inset windows, which shows the experiment running on the Epiphany web browser started in second place, and thus, the number of generations run so far (*Generacion*) is lower, although the state of the best solution is almost the same.

in the browser. This system would be the closest to what we are presenting in this paper.

The proposed approach could also be considered as *parasitic computing* since, as stated in Section I, the only participation from the user will be to load a web page and click on a button; in fact, any AJAX-based could use these resources without his consent (and, in any case, it would be desirable to run without causing much trouble). The concept was introduced by Barabási in [32], and followed by others (for instance, Kohring in [33]). In that work they proposed to use the Internet routers to compute a *checksum* by means of a set of specially crafted packets, whose aggregated result would be used to solve the SAT problem. Anyway, although the concept is interesting, there seems not to be a continuation for this work (at least openly), probably due to its inherent dangers (as analyzed in a paper by Lam et al. [34]).

The virtual machine embedded into the browser provides a way to easily do that kind of sneaky/parasitic computing, but JavaScript faces the handicap of being an interpreted language, which means that the efficiency of different implementations varies wildly. Moreover, it is not optimized for numerical computation but for object tree management (the so called DOM, document object model) and strings. Nevertheless its wide availability makes us think about considering it.

III. METHODOLOGY AND EXPERIMENTAL SETUP

For this experiments we have designed and implemented a client-server program written in Perl (server-side) and Javascript (client-side), communicating using JSON via the XMLHttpRequest object. This object requires than the website and the AJAX requests are served from the same host, which is a constraint. Code for both is available, under

the GPL, from `http://rubyforge.org/projects/dconrails/`.

The algorithm runs on the client for a fixed number of generations, as shown in Figure 1; running parameters are set from the server and are downloaded from it along with the webpage from which the experiment is run. After the preset number of generations, a request is made to the server with the best individual in the last generation. The algorithm stops and waits for the answer from the server. The server receives the request, stores it in a database, and sends back the best individual stored in the server. This individual is incorporated in the client population, which starts again to run. Several clients acting at the same time make requests asynchronously, using the facilities of the standard Apache web server. The server is thus used as a clearinghouse for interchange of information among the different clients; however, there's no explicit comunication or topology among the different nodes running the genetic algorithm. Besides, the fact that the server always contain the best individuals generated so far guarantees that the best solution (with a fixed number of evaluations resolution) is always kept. The server also sends back the number of generations the client should run; which is usually the same number as before, but turns to 0, thus stopping the client, when the stopping condition is met. Clients leave the experiment by the expeditive method of surfing away to another page or closing the web browser; in tabbed browsers (most browsers nowadays), a tab (or several) can run the experiment while the browser is available for other tasks. When the experiment has been running for a predeterminad number of evaluations (which were set, for this experiment, to 750000), all clients get a message to stop running, and change their user interface to a message offering them to reload the (new) experiment and start all over again. Besides, there is a *watching* daemon
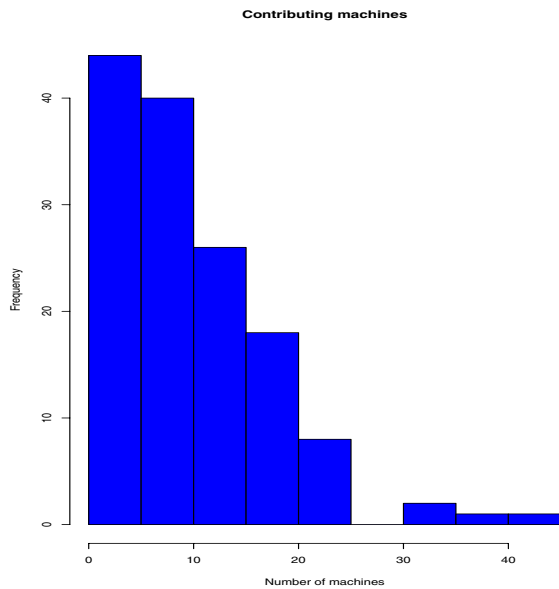
Contributing machines

Fig. 2. Histogram of the number of machines used in each experiment. The median is 8.5 machines, and the 75% quartile is placed at 15 machines, with a peak of 44 machines. A single client using several browsers counts as a single machine. It should be noted that not all clients are simultaneously connected.

running on the server which checks the database for the number of individuals evaluated, and resets the experiment by incrementing the experiment ID by one and eliminating the population. Thus, experiments can run unchecked on a server while this watchgad daemon[o] is running. Several additional utilities are also provided via several webpages, that inform on the state of the experiment, or allow to set the GA parameters. Experimental subjects were gathered by several methods: sending it via email to department and project coworkers, using the URL for the experiment as a Google Talk status line, as a Twitter (`http://twitter.com`) message, as a blog post, and, eventually, it was picked up by a wildly popular Spanish blog [5], which managed to gather the highest number of machines.

The experiment consisted in optimizing the 256-bits Royal Road function, and each instance consisted in a maximum of 750000 evaluations (which were barely enough to find the solution). The algorithm was steady state (with incorporation of the inmigrant every 20 generations), with rank-based selection and substitution; every generation, 50% of the population was generated, substituting the worst 50% individuals. Crossover priority was set to 80%, and mutation to 20%, changing 1% of the bits. However, these settings will have no influence on performance, other than the fact that, if the solution is found before reaching the maximum

number of evaluations, the users will get bored and change to a new page[6], instead of staying to see if the solution is found.

Data was gathered from two different sources: the watcher-daemon logs, which mainly gave data about the number of individuals evaluated and the time needed for each experiment, and the Apache daemon log; the relevant lines were extracted just by using grep. It should be noted that the server was not running exclusively the experiment, but doing it along with the usual tasks. The server was a 700-MHz, 1 Gigabyte-RAM machine, with the database running in a different, dual processor, 450-MHz machine. Both machines were running obsolete RedHat 7.x and 9.x Linux operating systems[7].

Results of the set of experiments that have been performed will be commented in the following section.

IV. EXPERIMENTAL RESULTS

Eventually, the experiment was running for several days, with different degrees of intensity. Several hundred machines participated in different instances, coming from all over the world, although mainly from Spain. The first question we wanted to answer was, how many machines should we expect in this kind of experiment? BOINC and SETI@home have gathered thousands of machines in its 10+ year run, but our experiments were limited in time (several hours, at most, if no machine was available, a few minutes if it was), so a limited number of machines should also be expected. The distribution of the number of machines is shown in Figure 2.

The exact number of machines and generations will vary in every experiment, but it seems clear that the median number of machines will hover around 1/4 of the maximum number. Besides, it is quite easy to obtain 5 clients for a certain number of evaluations; most experiments have less than 10 clients.

On the other hand, the number of evaluations each client contributes are quite different, as is shown in Figure 3. This figure reveals that most clients contribute just a few generations, be it because the browser stops running the program if it takes more than a predetermined number of seconds (which can happen in IE or in Firefox if clients are too slow; usually the predetermined number of generations will be less than this number), because the user gets bored and moves on, or because statistically most people join when the experiment has been running for some time and show up only when there are a few evaluations left. Each one of these problems would have to be tackled separately, but the baseline is that, even when a good number of machines joins an experiment, they will do so only for a limited

---

[5]Who posted it at `http://www.microsiervos.com/archivo/ordenadores/experimento-computacion-distribuida.html`.

[6]And this is just an example of how social factors in this kind of experiments affect performance.

[7]Both machines host our group web server and home pages; we thought it was better to run the experiment in our standard setup instead of a dedicated one, since the whole point of the experiment is to prove that no big modifications have to be made to run it, be it from the client or the experimenter point of view
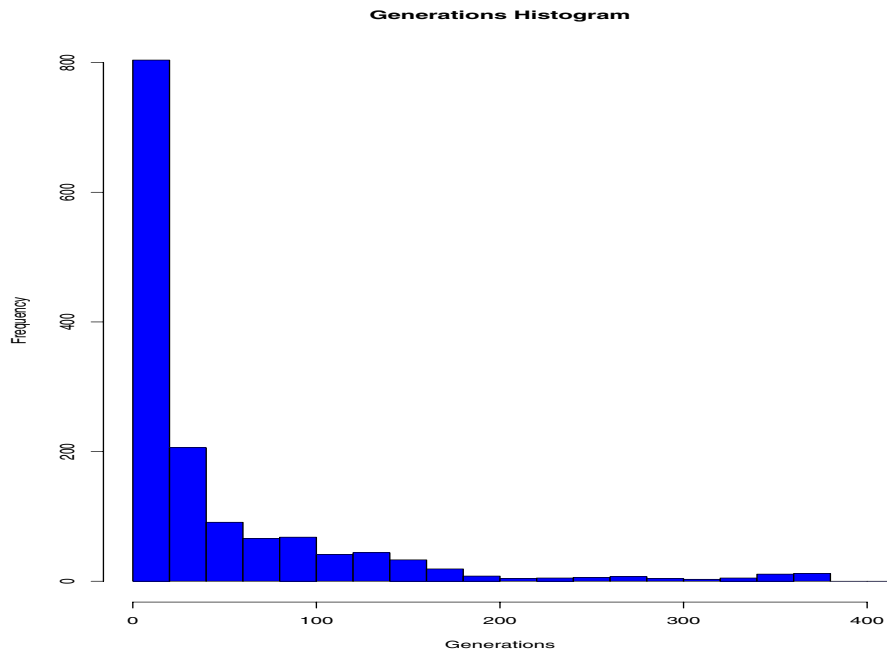
Fig. 3. Histogram of the number of generations all machines participating in the experiment have contributed, cut off at 400 generations. Each generation corresponds to 50 new individuals. Most clients contribute 100 generations or less, with a few contributing more than 200. The median is 16 generations, with the 3rd quartile placed at 55 generations.

amount of time on average. Besides, these limitations in number of generations translate also to limitations in time, so that experiments will have to be designed in a way that sufficient information is transmitted from clients during the expected time they are going to spend in the experiment. However, different clients have different performance, so it is also interesting to measure what is the average time (and thus average performance) it takes the clients between two communications to the server (that is, 20 * 50 evaluations + waiting time). This is interesting for two main reasons: server performance will have to be tuned to be able to serve this level of requests, and second, the generation gap will also have to be fine-tuned so that waiting time and the possibility that the script is blocked due to overtime is minimized. The performance figures obtained in the experiment are shown in Figure 4.

The graph represented in Figure 4 shows that, for this kind of problem, the vast majority of clients will have a gap smaller than two seconds. This quantity will vary for different problems, but the conclusion is that most clients will have high or median performance, with few clients having lower performance. This measure also gives us an estimate of the average performance (2.906 seconds/20 generations).

However, at the end of the day the setup is intended to achieve high performance when running an evolutionary computation experiment. This data is presented in Figure 5.

This figure is quite similar to Figure 4. Median is at

1000 seconds, with a minimum at 292 and 3rd quartile at 2323; 75% of runs will last less than 2323 seconds. Taking into account that the average 20-generation step is 2.906 seconds, and a single experiment needs 375 such steps, the average single machine run would take 1089.75 seconds; this means that the maximum speedup achieved is 1089.75/292 = 3.73 clients running simultaneously, and the median is approximately a single *average* machine. This will probably vary for experiments of different duration, but, on average, we could say that significative (albeit small) speedups can be achieved using spontaneous volunteer computing. In general, however, several machines will sequentially provide CPU cycles to an experiment, adding up to a single machine doing all the work. In general also, the fact that there are up to 44 machines working in a single experiment, or that the range of running times can vary in a factor of up to one hundred, indicates that, for this experiment, no bottleneck has been found. Of course, more simultaneous machines will have to be tested to find the limit. Finally, the fact that all contributions are volunteer means that the evaluation rate is not constant, yielding figures like Figure 6, where the steepness of each line is roughly equivalent to the evaluation speed, since the x axis corresponds to time, and the y axis number of individuals evaluated.

V. CONCLUSIONS, DISCUSSION AND FUTURE WORK

While in previous papers [12] we proved that this kind of AJAX based, volunteer, and potentially sneaky, computation
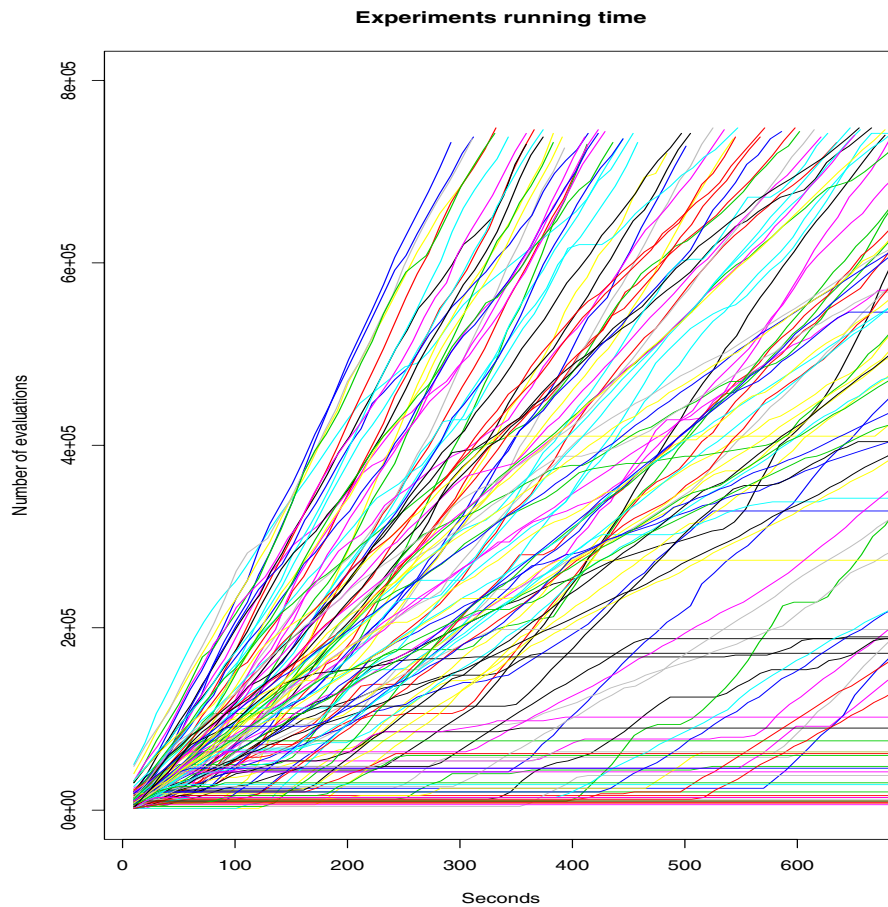
Experiments running time

Fig. 6. Plot showing the number of individuals evaluated vs. time for a few dozens experiments; time is plotted up to 600 seconds only. As is seen, some experiments have a more or less constant evaluation rate (constant inclination), while other are more step-like with clients leaving and joining the experiment all the time.

could be used profitably for performing genetic algorithm experiments, in this paper we have proved that, without an expensive or far-fetched setup, it can achieve high performance, equivalent, at most, to several computers of average performance. The code used to perform the experiment is publicly available and is modular so that creating different experiments is just a matter of writing a new JavaScript fitness function and tuning the GA parameters accordingly.

The experiments have proved that there is a good amount of computational power that can be easily tapped and used for evolutionary computation experiments, however, the nature of AGAJAJ constrains also the way users donate computing power, as well as the number of clients available for an experiment. In this paper we have found some figures, which will undoubtedly vary for other experiments; however, the general shape of the curves will probably be the same, following a very steep decrease from the maximum values obtained.

The GA, being asynchronous, faces some problems that have not been tackled in this paper. What is the best approach to preserve diversity? To generate a new population in each client, and receive inmigrants as soon as possible, which are incorporated into the population? Or is it better to create new client populations based on existing populations? What is really the algorithmic contribution of new clients? These issues will be explored as future work. We will also try to measure the limits of this technology, and test the impact of servers of varying performance and workload on overall performance. Eventually, we will also try to perform a *sneaky* experiment, to check what kind of performance can be expected in that kind of setups.

Another venue of work will be to examine the algorithmic performance of AGAJAJ; even as new clients are added to an experiment, what's the improvement obtained from them? In order to check that, a controlled experiment using known computers will be used, adding them one at a time, so that the

Generation gap

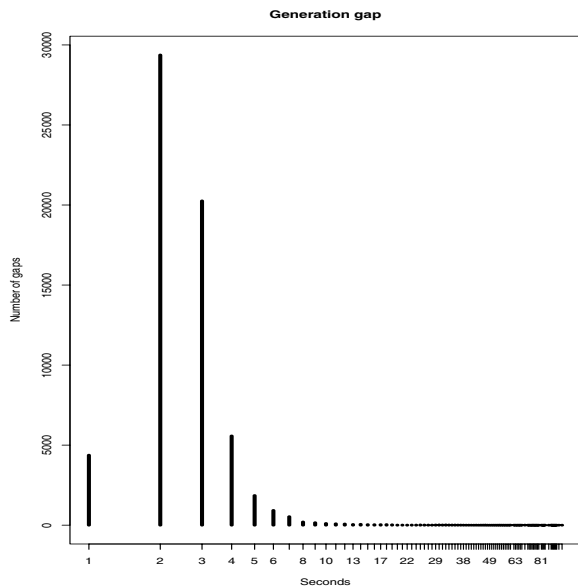
Experiment running time – Histogram

Fig. 4. Distribution of *gap* time, that is, time between two requests from a particular client; it has been rounded to the closest number of seconds (since the webserver logs it has been extracted from have that resolution). Most clients take 3 second or less to process 20 generations, with a majority taking 2 or less. Gaps bigger than 10-15 seconds are probably non-significative, in fact, negative gaps and those bigger than 100 have been eliminated from the data set. For this set of experiments, the median is at 2. x axis is logarithmic, to emphasize the fact that the distribution of client performance falls very fast, although a different median value should be expected for a different problem.
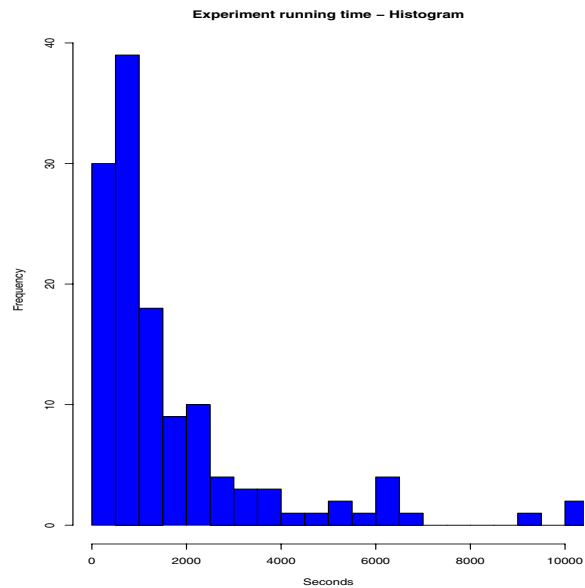
Fig. 5. Distribution of running times for a fixed amount of evaluations. Some outliers have been cut off; approximately 10% of runs took more than 10000 seconds. The mode is between 500 and 1000 seconds, and most runs end before 2000 seconds. Time starts to count from the moment the first client connects to the server.

real impact on the genetic algorithm is evaluated. Once that is know, it would be interesting to experiment with adaptive client parameters, instead of the one-size-fits-all parameter settings used so far.

REFERENCES

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: an experiment in public-resource computing," *Commun. ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[2] R. Steinmetz and K. Wehrle, Eds., *Peer-to-Peer Systems and Applications*, ser. Lecture Notes in Computer Science, vol. 3485.   Springer, 2005.

[3] Gilorien, *DHTML and JavaScript*.   Upper Saddle River, NJ 07458, USA: Prentice-Hall PTR, 2000.

[8]For instance, this article in my own blog `http://atalaya. blogalia.com/historias/53480`

[4] R. Shah, "A beginner's guide to JavaScript," *JavaWorld: IDG's magazine for the Java community*, vol. 1, no. 1, pp. ??–??, Mar. 1996, http://www.javaworld.com/javaworld/jw-03-1996/jw-03-javascript. intro.htm.

[5] D. Flanagan, *JavaScript Pocket Reference (2nd Edition)*.   O'Reilly, October 2002. [Online]. Available: http://www.amazon.fr/exec/obidos/ ASIN/0596004117/citeulike04-21

[6] ECMA, *ECMA-262: ECMAScript Language Specification*, 3rd ed. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), Dec. 1999. [Online]. Available: http://www.ecma-international.org/publications/standards/ Ecma-262.HTM

[7] ——, *ECMA-290: ECMAScript Components Specification*.   Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), Jun. 1999. [Online]. Available: http://www.ecma-international.org/publications/standards/ Ecma-290.HTM

[8] ——, *ECMA-327: ECMAScript 3: Compact Profile*, 3rd ed.   Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), Jun. 2001. [Online]. Available: http://www.ecma-international.org/publications/standards/ Ecma-327.htm

[9] ——, *ECMA-357: ECMAScript for XML (E4X) Specification*. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), Jun. 2004. [Online]. Available: http://www.ecma-international.org/publications/standards/ Ecma-357.htm

[10] Wikipedia, "Ajax (programming) — wikipedia, the free encyclopedia," 2008, [Online; accessed 12-February-2008]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Ajax_\ %28programming\%29&oldid=190788850

[11] "Towards Open Grid Services Architecture," http://www.globus.org/ ogsa/.

[12] J. J. Merelo, A. M. García, J. L. J. Laredo, J. Lupión, and F. Tricas, "Browser-based distributed evolutionary computation: performance and scaling behavior," in *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary compu-*

tation. New York, NY, USA: ACM Press, 2007, pp. 2851–2858.

[13] M. García-Arenas, J. G. Castellano, P. A. Castillo, J. Carpio, M. Cillero, J.-J. Merelo-Guervós, A. Prieto, V. Rivas, and G. Romero, "Speedup measurements for a distributed evolutionary algorithm that uses Jini," in *XI Jornadas de Paralelismo*, U. d. G. Depto. ATC, Ed., 2000, pp. 241–246.

[14] C. Setzkorn and R. C. Paton, "Javaspaces - an affordable technology for the simple implementation of reusable parallel evolutionary algorithms," in *Proceedings of the International Symposium on Knowledge Exploration in Life Science Informatics, KELSI 2004*, ser. Lecture Notes in Artificial Inteligence, J. A. López, E. Benfenati, and W. Dubitzky, Eds., vol. 3303. Milan, Italy: Springer, 25-26 Nov. 2004, pp. 151–160. [Online]. Available: http://springerlink.metapress.com/openurl. asp?genre=article&issn=0302-9743&volume=3303&spage=151

[15] F. S. Chong and W. B. Langdon, "Java based distributed genetic programming on the internet," in *Proceedings of the Genetic and Evolutionary Computation Conference*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., vol. 2. Orlando, Florida, USA: Morgan Kaufmann, 13-17 Jul. 1999, p. 1229, full text in technical report CSRP-99-7. [Online]. Available: http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/p.chong/DGPposter.pdf

[16] D. S. Myers and M. P. Cummings, "Necessity is the mother of invention: a simple grid computing system using commodity tools," *J. Parallel Distrib. Comput.*, vol. 63, no. 5, pp. 578–589, 2003.

[17] J. L. J. Laredo, P. A. Castillo, B. Paechter, A. M. Mora, E. Alfaro-Cid, A. I. Esparcia-Alcázar, and J. J. Merelo, "Empirical validation of a gossiping communication mechanism for parallel EAs," ser. Lecture Notes in Computer Science, M. G. et al., Ed., vol. 4448. Springer, 2007, pp. 129–136.

[18] E. A. Eiben, M. Schoenauer, J. L. J. Laredo, P. A. Castillo, A. M. Mora, and J. J. Merelo, "Exploring selection mechanisms for an agent-based distributed evolutionary algorithm," in *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*. New York, NY, USA: ACM Press, 2007, pp. 2801–2808.

[19] E. Cantú-Paz, "Migration policies, selection pressure, and parallel evolutionary algorithms," *Journal of Heuristics*, vol. 7, no. 4, pp. 311–334, 2001.

[20] L. F. G. Sarmenta and S. Hirano, "Bayanihan: building and studying Web-based volunteer computing systems using Java," *Future Generation Computer Systems*, vol. 15, no. 5-6, pp. 675–686, 1999. [Online]. Available: citeseer.nj.nec.com/sarmenta99bayanihan.html

[21] D. P. Anderson, E. Korpela, and R. Walton, "High-performance task distribution for volunteer computing," in *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 196–203.

[22] P. Cappello and D. Mourloukos, "A scalable, robust network for parallel computing," in *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*. New York, NY, USA: ACM Press, 2001, pp. 78–86.

[23] D. Anderson, "BOINC: a system for public-resource computing and storage," *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pp. 4–10, 8 Nov. 2004.

[24] M. Arenas, P. Collet, A. Eiben, M. Jelasity, J. J. Merelo, B. Paechter, M. Preuß, and M. Schoenauer, "A framework for distributed evolutionary algorithms," ser. Lecture Notes in Computer Science, LNCS, J.-J. M. Guervós, P. Adamidis, H.-G. Beyer, J.-L. Fernández-Villacañas, and H.-P. Schwefel, Eds., no. 2439. Springer-Verlag, September 2002, pp. 665–675. [Online]. Available: http://www.springerlink.com/link.asp?id=h4n29kbl69jvab4c

[25] Wikipedia, "Electric sheep — wikipedia, the free encyclopedia," 2007, http://en.wikipedia.org/w/index.php?title=Electric_Sheep&oldid=159677937.

[26] R. Mason and W. Kelly, "G2-P2P: a fully decentralised fault-tolerant cycle-stealing framework," in *ACSW Frontiers '05: Proceedings of the 2005 Australasian workshop on Grid computing and e-research*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2005, pp. 33–39.

[27] F. F. D. Vega, "A fault tolerant optimization algorithm based on evolutionary computation," in *DEPCOS-RELCOMEX '06: Proceedings of the International Conference on Dependability of Computer Systems (DEPCOS-RELCOMEX'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 335–342.

[28] L. F. G. Sarmenta, "Sabotage-tolerance mechanisms for volunteer computing systems," *Future Generation Computer Systems*, vol. 18, no. 4, pp. 561–572, 2002. [Online]. Available: citeseer.nj.nec.com/449803.html

[29] J. Culbert, "Ajax and distributed computation thoughts," Published at http://culbert.net/?p=6, March 2006, last Accessed December 2007.

[30] W. B. Langdon, "Pfeiffer – a distributed open-ended evolutionary system," in *AISB'05: Proceedings of the Joint Symposium on Socially Inspired Computing (METAS 2005)*, B. Edmonds, N. Gilbert, S. Gustafson, D. Hales, and N. Krasnogor, Eds., University of Hertfordshire, Hatfield, UK, 12-15 Apr. 2005, pp. 7–13, sSAISB 2005 Convention. [Online]. Available: http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_metas2005.pdf

[31] J. Klein and L. Spector, "Unwitting distributed genetic programming via asynchronous JavaScript and XML," in *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*. New York, NY, USA: ACM, 2007, pp. 1628–1635.

[32] A.-L. Barabási, V. W. Freeh, H. Jeong, and J. B. Brockman, "Parasitic computing," *Nature*, vol. 412, no. 6850, pp. 894–897, August 2001. [Online]. Available: http://www.nature.com/cgi-taf/DynaPage.taf?file=/nature/journal/v412/n6850/abs/412894a0_fs.html

[33] G. A. Kohring, "Implicit simulations using messaging protocols," *COMPUTERS AND PHYSICS*, vol. 14, p. 203, 2003. [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0208021

[34] V. T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis, "Puppetnets: misusing web browsers as a distributed attack infrastructure," in *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2006, pp. 221–234.