The Seamless Peer And Cloud Evolution Framework

Guillaume Leclerc, Joshua E. Auerbach, Giovanni Iacca[*] and Dario Floreano
Laboratory of Intelligent Systems
Ecole Polytéchnique Fédérale de Lausanne
Lausanne, Switzerland
{first name}.{last name}@epfl.ch

ABSTRACT

Evolutionary algorithms are increasingly being applied to problems that are too computationally expensive to run on a single personal computer due to costly fitness function evaluations and/or large numbers of fitness evaluations. Here, we introduce the Seamless Peer And Cloud Evolution (SPACE) framework, which leverages bleeding edge web technologies to allow the computational resources necessary for running large scale evolutionary experiments to be made available to amateur and professional researchers alike, in a scalable and cost-effective manner, directly from their web browsers. The SPACE framework accomplishes this by distributing fitness evaluations across a heterogeneous pool of cloud compute nodes and peer computers. As a proof of concept, this framework has been attached to the RoboGen$^{\text{TM}}$ open-source platform for the co-evolution of robot bodies and brains, but importantly the framework has been built in a modular fashion such that it can be easily coupled with other evolutionary computation systems.

CCS Concepts

•**Mathematics of computing** → *Bio-inspired optimization;* •**Computing methodologies** → *Distributed algorithms;* •**Software and its engineering** → *Distributed systems organizing principles;*

Keywords

Evolutionary Computation; Distributed Computing

1. INTRODUCTION

Evolutionary algorithms (EAs) are increasingly being applied to problems that are too computationally expensive to run on a single personal computer due to costly fitness function evaluations and/or large numbers of fitness evaluations. For example, the experiments behind one recent Evolutionary Robotics paper were stated to have taken over 100 CPU-years of compute time, and hence were carried out on a 7.1 teraflop supercomputing cluster [4]. Unfortunately, many people who would like to take advantage of evolutionary computation techniques for these kinds of problems, usually do not have access to such clusters, and therefore it is unfeasible for them to achieve meaningful results in a reasonable amount of time. Next to that, the majority of existing EA systems are either only available for a specific platform and/or require time-consuming and possibly difficult installation procedures.

Being relevant to several application domains, these issues have attracted a certain attention in the Evolutionary Computation research community for over 15 years, and a number of systems which distribute EAs among peers (or to the cloud) have been proposed. From an implementation point of view, we can identify three main ways in which such systems have been previously realized:

1) The first includes volunteer peer-to-peer (P2P) systems, where each peer installs a client application to communicate the data of the evolutionary algorithm over the Internet (typically over TCP/IP). A pioneering work in this area was performed by Chong and Langdon [6], who implemented a P2P Genetic Programming (GP) framework. A P2P Genetic Algorithm (GA) was then proposed in [2, 25, 26]. All of these platforms share a similar architecture: they use a deme-structured population (i.e., an island-model GA with occasional migration of elite individuals) distributed over multiple computing resources. Each peer runs a Java client encapsulating an application layer (made of an evolver and an algorithm library–both distributed over the network) and a P2P core service layer, which handles the network functionalities (secure peer discovery, messaging, and file transferring). In all cases the underlying layer runs over the TCP/IP protocol stack, except for the system proposed in [2] which runs over a DRM (Distributed Resource Machine).

Volunteer-computing was also tested in [8], by gathering the computing resources of up to 27,000 hosts participating in the MilkyWay@home project[1]. In this case, the authors ran a distributed version of Differential Evolution (DE) and Particle Swarm Optimization (PSO), with a parallelization granularity at the level of a single evaluation.

Distributed GP was also contributed, more recently, in [7, 29]: the proposed P2P framework, called FlexGP, is based on the OpenStack[2] open-source cloud computing platform and makes use of a simple island-based system where all

[*]G. Iacca is also affiliated with the Département d'Ecologie et d'Evolution, Université de Lausanne, Keller Group.

GECCO'16, July 20-24, 2016, Denver, Colorado, USA.
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN TBA.
DOI: 10.1145/1235

[1]http://milkyway.cs.rpi.edu
[2]http://www.openstack.org

the peer nodes run independent copies of the GP-based regression algorithm, each one using different parameters and learning from different subsets of the original dataset. The best models obtained in each node are eventually fused offline to provide the user with an optimal meta-model for the regression task at hand (the system is, therefore, particularly suitable for solving large regression problems). Finally, an abstract model for scalable P2P evolutionary computation, named EvAg (Evolvable Agent), was proposed in [17], and a thorough analysis of the scalability of the system with respect to the problem size was provided. This model was further investigated in [16]. However, in both studies, the authors focused mostly on the conceptual aspects of the distributed GA (and how to simulate those) rather than the actual implementation.

2) The second set consists of frameworks that rely on distributed file systems, or existing cloud storage services, for the sharing of data and resources needed by an evolutionary algorithm. Examples of this kind are represented by island-based methods that realize the parallelization through MapReduce, a powerful platform for distributed computation based on the distributed file system Apache Hadoop[3]. These include MapReduce implementations of GP [11], Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [30], and, more recently, a new Swarm Intelligence method called the *Fireworks Algorithm* [18]. Another island-based GA, which was presented in [21], relies instead on two different cloud storage services (Dropbox and SugarSync) for the file synchronization among islands.

3) Finally, the newest frameworks distribute the calculations via the browser, either to simply submit resources, configure and run the evolutionary tasks through a web interface, or by performing computations directly in a user's web browser by leveraging cutting-edge web technologies. This idea falls within the broad new trend of leveraging bleeding edge web technologies to make user friendly platforms that can be run directly from a web browser without the need to install any additional software [27].

Such a possibility was first tested in [20]: in this work, JavaScript clients (running in the browser) communicate with a Perl server through JSON and `XmlHttpRequest`[4]. Each client evolves a separate population (with a predetermined JavaScript fitness function) and asynchronously exchanges the local elites with the global elites taken from a master population stored on the server. The same framework was recently re-engineered in [24] by making use of cutting edge technologies (and a reimplementation of the server in PHP), and was extensively tested on several browsers. Due to the new design, the resulting system, (named jsEO) is characterized by a higher level of flexibility and configurability. Yet another improved version of this system, named NodIO [19], replaces the original server with a REST server that handles the routing to the JavaScript clients.

Duda and Dłubacz have proposed [9] a similar island-model system composed of JavaScript clients and a Node.js server. The main difference from jsEO and NodIO is that instead of using AJAX/AJAJ, it makes use of the AJAX "long polling" technique to push data from the server to the clients in the absence of an explicit request.

[3]http://hadoop.apache.org/core

[4]http://www.w3.org/TR/XMLHttpRequest

In the present contribution, we further push the research in the area of browser-based evolutionary computation, asking whether it is possible to make the computational resources necessary for running large scale evolutionary experiments, with expensive fitness evaluations, available for amateur and professional researchers alike, in a web-based, user-friendly, accessible, scalable and cost-effective manner. To this aim, we introduce the Seamless Peer And Cloud Evolution (SPACE) framework[5], a novel web-based distributed framework that allows for any visitor to a website to initiate an evolutionary experiment, which is then distributed to other computing resources. An unprecedented feature of our system is that it combines the possibility of distributing the computation at the same time to other peers visiting the same website, as well as to "cloud" based servers running optimized versions of the fitness evaluation code (written in the developer's language of choice). Both kinds of nodes (browser clients and cloud servers) can seamlessly join the system in a way that is completely transparent to the users. This allows us to leverage on-demand, cloud-based service providers, such as Amazon Elastic Comput Cloud (EC2)[6], at the same time as encouraging the general populous to donate CPU cycles from their personal computers, such as has been done in popular citizen science projects such as SETI@home [1]. Both of these sources of computing resources are then made available to the users directly from their web browsers.

The remainder of this paper is structured as follows. The next section describes the system architecture and the main technology choices behind that. Section 3 describes an example application of the SPACE framework based on RoboGen^{TM}, an open-source platform for the co-evolution of robot bodies and brains [3]. Section 4 discusses the experimental results and highlights the key elements of the SPACE framework in comparison with alternative solutions from the literature. Finally, Section 5 concludes this work.

2. METHODS

This section first defines the system requirements. Then, a description of the specific technology choices that were made is provided along with the reasoning behind those choices, and how those technologies were applied. Finally there is a discussion of how the SPACE framework was designed and implemented.

2.1 Requirements

A major goal of this work is to build a distributed evolutionary computation system that is usable by the largest possible number of users. To fulfill this goal, the system should be:

1. *cross-platform*, i.e. it should be available on as many computer platforms as possible (Linux/Windows/Mac OS systems, as well as tablets or smart phones);
2. *usable off-the-shelf*, i.e. it should not require the installation of any third party components (such as plug-ins, addons, etc.), which would make the user experience less immediate.

While the first requirement could be met by using interpreted or cross-platform languages, such as Python or Java,

[5]https://github.com/lis-epfl/space-framework

[6]https://aws.amazon.com/ec2/

this would still require users to install an interpreter (as in the case of Python) or a Virtual Machine (as in the case of Java). Furthermore, some of these components may not be available on all platforms, may require continuous updates, or may even contain security flaws (this is the case, for instance, of the Adobe Flash player and some Java applets).

A more appealing alternative is to use browser technologies. Nowadays, web browsers are bundled with almost every operating system, so by offering a solution that runs inside of a web browser, we would provide compatibility with the vast majority of personal computers, tablets, and smart-phones, without requiring the user to install any software. Therefore, we concentrate our choices on technologies already available in most web browsers.

2.2 Networking protocols

An important aspect with browser technologies is the choice of the protocol to allow bi-directional communication between multiple users of the system and (as we will see below) with the cloud. While `HTTP` would be the simplest choice in this case (since it is the default protocol for web pages), making raw `HTTP` requests is equivalent to repeatedly loading a new web page. This would introduce a latency that would be detrimental to the user experience and the whole system throughput.

However, modern browsers now offer efficient alternatives to `HTTP`, namely: `XMLHttpRequests`, `WebSockets` and `Socket.io`. Below, the main features of these three protocols are analyzed, highlighting the pros and cons of each, and finally motivating our choice.

XMLHttpRequest

`XMLHttpRequest` is an API that allows JavaScript code to send `HTTP` and `HTTPS` requests in the background to the same domain name as the one from which a web page was downloaded from[7]. This protocol has many advantages: it is mature, it is supported by every modern browser, and it is simple to use. However, it also has disadvantages: it is uni-directional, which means that the client can initiate a connection to the server, but the server is passive and has to wait for user requests. The server can respond with data only after receiving a user request. In order to use this protocol for bi-directional communication it would be necessary to find a mechanism for the server to send evaluation requests to the clients. This could be done with a polling protocol [5], however this requires creating a new connection for each polling request, which incurs a large latency.

WebSockets

An alternative is offered by `WebSockets`[8], which were designed to address the main limitations of `XMLHttpRequests`. They are similar to TCP sockets, and have the convenient characteristic that they can run alongside a standard `HTTP` server on the same port, thus avoiding any issues related to firewall configurations. However, they do have some limitations: connections can only be opened between the server and a client, but it is not possible to have peer-to-peer communication. Moreover, `WebSockets` are not interoperable with classic TCP sockets, and currently they are not implemented in all web browsers.

[7]There are also ways to reach other domains name using CORS (http://www.w3.org/TR/cors/) policies.

[8]http://www.w3.org/TR/websockets/

Socket.io

Finally, `Socket.io`[9], is a bleeding edge technology which solves the few shortcomings of `WebSockets`, while also providing compatibility with a diverse set of programming languages. At the time being, it supports JavaScript (on both server and client side), C/C++, Python, Java and Swift. Another major advantage of `Socket.io` is that is supports several underlying protocols, including `XMLHttpRequest`, `WebSockets`, `Flash Sockets`, `AJAX long-polling`, `AJAX multipart streaming`, `IFrames` (standard HttpRequests), and `JSON Polling`. `Socket.io` automatically chooses the best protocol depending on the capabilities of the client and server, by allowing old browsers to fall back to `XMLHttpRequest` (JSON) polling if required. The only drawback of `Socket.io` is that, since it uses a “client/server” architecture, all messages destined to another client must go through the server, which impedes true P2P communication. This might be overcome by using `WebRTC` (Web Real Time Communication), a lightweight UDP-based open-source protocol developed by Google in 2011[10]. However, `WebRTC` is currently supported only by 58.09% of Internet users[11] and, unlike `WebSockets`, there are no libraries that provide a fall-back protocol for browsers that do not support `WebRTC`. This is unacceptable for our purposes, since it would mean that a large percentage of potential users could not use the system. Therefore, our framework makes use of a client/server architecture.

To summarize, although `XMLHttpRequest` is robust and used on almost every website, it does not offer bi-directional communication. It would require introducing a polling procedure, which would unduly stress the network, and provide inferior performance. Therefore it was decided to use `Socket.io`, as it can leverage `WebSockets` when available, without adding any measurable overhead. We should note that this choice differentiates the SPACE framework from most of the previous works in the literature [19, 20, 24].

Furthermore, the multi-language support provided by `Socket.io` is important because it allows for a simple means of communication between heterogeneous processes. In fact, to make the system as scalable as possible (as will become clear in the next section) it is desirable to distribute fitness evaluations among peers running a JavaScript implementation in their web browser, as well as cloud computing nodes running an optimized C++ implementation of the same fitness evaluation code.

2.3 System architecture

In order to support `Socket.io`, the system must be designed around its client/server architecture, and hence a central server is needed to route the messages between clients. This server will do very little computation, but it will need to handle many clients connected at the same time, and since we also provide computation on the cloud, it will be up to this server to interact with cloud providers in order to start up and shutdown virtual machines on demand. This server will also require sufficient memory to maintain many concurrent, open connections.

`Socket.io` is a library originally designed for the browser

[9]http://socket.io/

[10]http://www.w3.org/TR/webrtc/

[11]http://caniuse.com/#feat=rtcpeerconnection

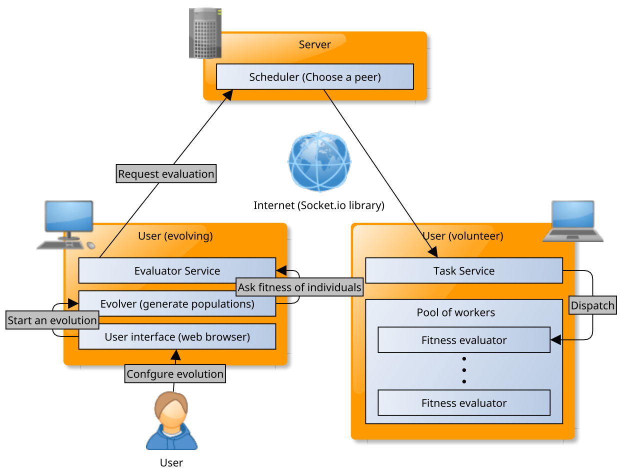

Figure 1: Global architecture

and the Node.js platform. This is why it uses JSON (JavaScript Object Notation) to serialize data across the network. Implementing the scheduling server in Node.js would be a natural choice. It offers many advantages for the current application: its concurrency model avoids data races and makes it capable of handling many request in a short amount of time. While Node.js is typically slower than compiled languages such as C++, this is not a big issue for our purposes, because the server will not do any heavy computation. The main problem with Node.js is that it can only run on a single thread. Under heavy traffic, this might present a problem, however, this can be solved by using multiple Node.js servers and a load balancer.

In order to keep the code clean and modular, a set of interfaces and specialized implementations are required. The problem with JavaScript is that it is an untyped language and does not have any concept of interfaces. When dealing with network packets, it is always safer to have a strongly typed language able to detect missing/misspelled fields so that users do not received malformed packets. For these reasons, the decision was made to use TypeScript[12], an open-source superset of JavaScript. As described in [23], TypeScript is useful for avoiding mistakes while writing JavaScript code.

Distributing Evaluations

As mentioned above, the aim of the SPACE framework is to distribute complex fitness evaluations. Here, we explain in detail how the CPU cycles of volunteers are used to speed up the evolutionary process.

The system is best described through an example use-case. Say that a user (researcher/hobbyist) wants to run a new experiment. They visit a specified website where they start their experiment directly in their web browser. This user's computer will henceforth be referred to as the the initiator, but importantly multiple initiator computers may exist in the system at the same time. To evaluate an individual, the initiator prepares a simple JSON packet containing a GUID[13] and the description of the individual. This packet is given to the SPACE framework API. It is forwarded to the scheduling server using `Socket.io`. The scheduler is the central component of the system, its roles are to maintain an up-to-date set of connected users, assign fitness evaluations to nodes and route packets to their recipients.

The fitness evaluation task will then be assigned to a node based on the scheduling policy (see Section 2.4). Once the task has been assigned to a node, the packet is sent to that machine. Every node runs a pool of Web workers. The task is thereafter saved in a local queue where it waits for an available worker, and then evaluates the individual as soon as possible. Once the fitness has been computed, a new JSON packet, which contains the same GUID as the request along with the computed fitness value, is created and travels back to the initiator using the same path. The scheduler maintains a map from each GUID to the initiator to be sure the packet is sent back to the correct node. This data flow is depicted graphically in Fig. 1.

2.4 Scheduling Policy

A policy is needed for deciding which peer machine will be assigned each requested fitness evaluation. This policy should aim to spread the load across peers as uniformly as possible. To accomplish this goal the following scheduling policy was implemented. When a new evaluation task t arrives the scheduler computes the load of all connected peers and chooses the peer m with minimal load. If m has at least one core available (load < 1) t will be assigned to m, otherwise t is placed into a first-in-first-out (FIFO) queue. When a peer finishes an evaluation, it must have at least one core available (the one which computed the finished evaluation), and so an evaluation is popped from the queue and assigned to this peer. This scheduler is simple, works well to balance the evaluations across users, has very little overhead and good throughput. However, it is not perfect, and some of its shortcomings are discussed below (see Section 5).

2.5 Leveraging the "Cloud"

The SPACE framework supports connecting, disconnecting and crashing peers during an evaluation[14]. However, a sufficient number of peers volunteering their CPU cycles is needed to achieve a meaningful pool of compute resources. On the other hand, modern "cloud" platforms offer access to flexible virtual machines. They can be started in minutes, use templates, and can be killed when no longer needed, all while being billed only for the hours consumed. For these reasons, we have incorporated the ability to add virtual machines to our worker pool. Specifically, we have included virtual machines provided by Amazon EC2 (as also done, for instance, in [11]), but other virtual machine providers, such as Microsoft Azure[15] or the Google cloud platform[16] would also be straightforward to incorporate. Another possibility would be to set up a private cloud system using OpenStack, as done in [7, 29].

To handle the decisions of how many virtual machines are desired[17], a virtual machine manager (VMM) has been implemented. The VMM watches the state of the sched-

[12]https://github.com/Microsoft/TypeScript
[13]Globally Unique Identifier

[14]When a user is no longer connected all of their assigned tasks are rescheduled to other peers.
[15]https://azure.microsoft.com/en-us/
[16]https://cloud.google.com/
[17]We leverage the Amazon `Auto Scaling Group` API to automatically scale the number of VMs based on this quantity.

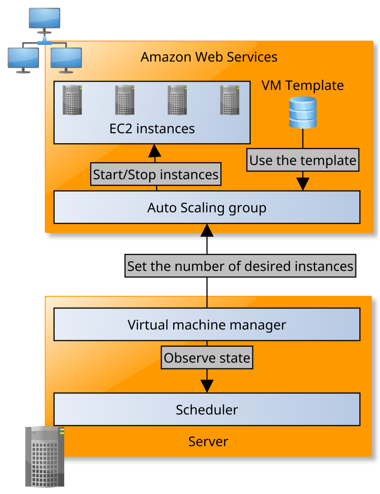

Figure 2: Cloud management architecture

uler (contents of the queue, number of connected users, and current throughput), and decides the number of desired instances based on a specified criteria. The architecture is detailed in Fig. 2.

Like the scheduler, the VMM is swappable by another manager. In this way different scaling policies can be implemented. The currently implemented policy works as follows: every fixed interval[18], the VMM computes the expected time to empty (ETE) the queue of the scheduler. This ETE is smoothed using an exponential moving average. If the smoothed ETE is bigger than a threshold, then a new instance is added; if it is lower than a different, smaller threshold, then the number of desired instances is decreased. After the number of desired instances is modified the systems waits a given amount of time for new machines to be started, and then resets the moving average to zero.

The ETE is estimated as follows: first the VMM asks the scheduler for the throughput (in tasks per second), as well as the number of queued tasks. If this throughput is below a given threshold, then the VMM asks for the average waiting time of tasks before being processed and multiplies this by the number of queued tasks to arrive at an ETE. However, this estimate may be too heavily influenced by outdated information. So, if the throughput is sufficiently large, then the ETE is computed as the number of tasks divided by the throughput. Since the throughput is always a fresh measurement (< 10s old), this provides a more accurate estimate.

3. RoboGen™ CASE-STUDY

Since the SPACE framework has been designed to speed up evolutionary experiments that have costly fitness functions, we chose to demonstrate its effectiveness in the domain of Evolutionary Robotics [22] where in most cases the computational bottleneck is created by the large cost of physically simulating robots. In particular we chose the RoboGen™ Evolutionary Robotics platform [3] for experimentation, because it is open-source and has recently been ported to run inside a web browser.

[18]All intervals/thresholds can be configured through a JSON file on the server.

3.1 Architectural Setup

There are currently two versions of the RoboGen software platform[19]: A C++ version, and a JavaScript version called `RoboGen.Js`. Most of the JavaScript code is `ASM.js`[20] that is compiled from the C++ code base using `emscripten`[21]. Only the non-portable code differs between the two versions.

Even if `ASM.js` offers performance benefits over vanilla JavaScript, `RoboGen.js` is still about 4-5x slower than the C++ version, mostly due to the complex matrix operations inherent in simulating robot physics[22]. In order to get the best performance from the system, the compute grid is comprised of heterogeneous nodes. The EC2 instances (see above) run the more efficient C++ version, while peer users run the in-browser version that does not require installing any software. Finally, to avoid blocking the user interface while computations are going on (as explained by [19]), the SPACE framework uses a dynamic pool of workers[23].

3.2 Experimental Setup

The goal of the first experiment (Experiment 1) is to evaluate how the system is capable of making use of a heterogeneous pool comprized of both peer-nodes and cloud servers. In order to show how peer volunteers offering their CPU cycles through the browser based software impact the performance, a number of peers gradually join the system over the duration of the experiment. To make the measurements more consistent, all the peers run on a machine with the same hardware and software configuration. They all run Ubuntu 12.04, Firefox 43 on an Intel™ i7 with 4/8 multi-threaded 3.2GHz cores. The worker pool of each peer contains eight threads, which means eight fitness evaluations can run concurrently on each peer machine.

Additionally, a dynamic pool of EC2 instances starts empty and is capped at 4 instances. All instances are of type ‘T2.micro’[24] with 1GB of main memory, and run the C++ version of RoboGen™. In order to show that it is also possible to add additional virtual machines to the grid while it is operating, five more virtual machines are added after the dynamic pool has reached its maximum capacity.

The task we want to tackle is to evolve the brain (neural network) of a quadruped robot. The fitness function is the distance traveled by the robot in a 60 second simulation. In order to measure the speed of the evaluations, we evaluate a population of $10,000$ individuals for a single generation. This configuration is used in order to study this scaling without polluting the data with brief intervals of zero throughput that would be present while new generations are being generated (as that interval is independent of the number of nodes in the system, and should generally be short relative to the time needed for expensive fitness evaluations). In real world applications, it is likely that multiple experiments are running in the system concurrently, which would keep all nodes busy at all times.

Two additional experiments (Experiments 2 and 3) aim

[19]http://robogen.org
[20]http://asmjs.org/spec/latest/
[21]https://kripken.github.io/emscripten-site/
[22]In RoboGen all physics simulations are conducted using the Open Dynamics Engine (http://www.ode.org/), which has also been compiled to `ASM.js` with `emscripten`.)
[23]https://github.com/GuillaumeLeclerc/WebWorker
[24]More information about EC2 instances: https://aws.amazon.com/ec2/instance-types/

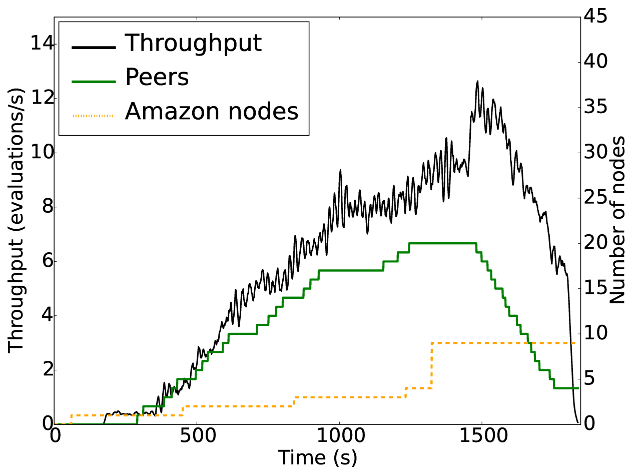

Figure 3: Experiment 1: throughput and number of nodes as a function of time. This experiment demonstrates the ability of the system to effectively utilize a heterogeneous pool of peer and cloud compute nodes.

to further demonstrate the strength of the SPACE framework: fault tolerance and scalability, respectively. In order to demonstrate how the SPACE framework seamlessly handles nodes either failing or leaving the pool, an experimental protocol is defined that reflects the behavior of real users from previous volunteer-computing experiments: the length of time that each node spends in the pool is sampled from a Weibull distribution [28] with shape parameter 0.38 and scale 3.1 as was presented as the best fit of user behavior in [15]. In Experiment 2, up to 98 workers[25] join the pool over the course of a 3000s time window during which up to four evolutionary runs are optimizing neural network controllers to maximize the displacement of an articulated robot[26]. Each node joins the pool at a time chosen uniformly at random in the first $[0, 1800]$s and then remains active for a number of seconds sampled from the Weibull distribution just described.

Since many of the nodes in Experiment 2 leave the pool prior to the end of the experiment, a final experiment, Experiment 3, replicates Experiment 2, but once a node joins the pool it remains there for the duration of the experiment. This allows us to observe the performance of the system as it scales beyond what is seen in Experiments 1 and 2.

3.3 Results and interpretation

The results of Experiment 1 are shown in Fig. 3. This plot shows the throughput of the system over time along with the number of peers and EC2 instances connected to the grid. These measurements have been smoothed with a simple moving average with a 10s window in order to make the results easier to understand.

[25]Here all workers use a single core of an Amazon c4.large instance. Only Amazon nodes are included in this experiment to allow for defining the start times and durations programmatically, and because the ability to employ a heterogeneous pool is studied separately in Experiment 1.

[26]Each experimental run consists of a generational EA with a population size of 500 where each fitness evaluation involves simulating a robot for 50s of simulated time.

The first thing that can be noticed is the delay between the time when an EC2 instance starts up and when it starts contributing to the throughput. It typically takes about 45 seconds for a virtual machine to start. This can be observed in the plot when the first instances start as well as when the five additional instances start. This is clearly not the case for peer volunteers, because their machines are already up and running when they open the website.

The first four EC2 instances start every 30 seconds (approximately), which means the virtual machine manager is working properly. Indeed, it was configured not to start virtual machines too fast to reduce the cost of this small experiment (because it is necessary to pay for every hour of computation we start). The five other EC2 instances start at the same time because they were started manually. We can also remark that the throughput reaches 0 at the end of the experiment even though there are still machines in the grid. This is the case because we reached the end of the experiment (all 10,000 individuals had been evaluated).

The most important takeaway from this figure is that the system scales almost perfectly: the throughput is approximately the sum of the two other curves (if the green curve is shifted in order to take into account that EC2 instances take some time to start). This result demonstrates the effectiveness of the SPACE framework. Moreover the CPU and network usage on the scheduling server remained low (the processor utilization never exceeded 6%), even though we were using the smallest instance available in our geographical region (`T2.micro`)).

The results of Experiments 2 and 3 are shown in Fig. 4. The main figure depicts the number of nodes and the throughput over time. Here, the primary results are that the system (a) functions well while multiple evolutionary runs are active simultaneously and (b) the system is able to seamlessly handle nodes crashing and/or users leaving as would be the case in applied experiments. Moreover the results of Experiment 3 (shown in the inset) demonstrate a near perfect scaling to a large number of nodes.

4. DISCUSSION

Most other solutions in this area (such as [19, 24]) use island based algorithms, therefore peers evolve a population on their own and only occassionally share individuals. The SPACE framework is closer to the classic volunteering solutions such as `SETI@Home`. Indeed, the only task that peers perform is evaluating fitnesses. They receive tasks (i.e. individual evaluation requests) and they return the fitness as soon as they are done. In the example with RoboGen, the fitness depends on the result of a potentially long physics and neural network simulation. As demonstrated in the previous section, this system can successfully be used to speed up these costly fitness evaluations.

Applying island-like algorithms to problems with complex fitness functions could lead to some problems. Since each peer would need to maintain and evaluate its own population, its load could be heavier. Moreover, it is possible that some users would close their browser before the computation has completed. In this situation they would not have sent the locally best individual to the global population (hosted on a central server), and all the CPU cycles from those users would be lost. In addition, multiple islands might needlessly perform the same fitness evaluation, which would be a potentially large waste of resources. All of these

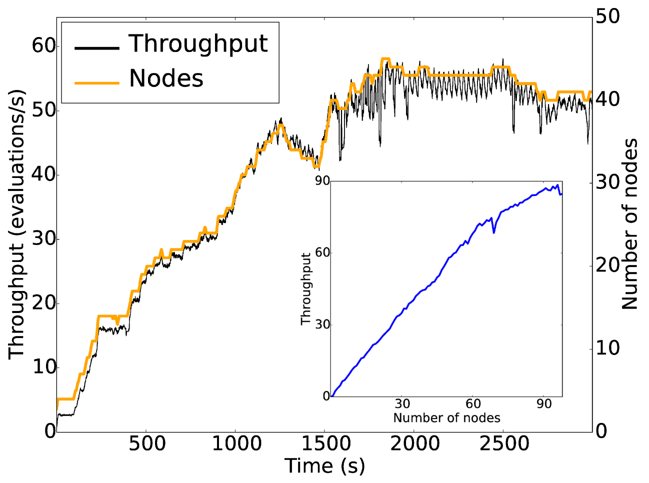

Figure 4: Experiments 2 and 3. Main figure: throughput and number of nodes as a function of time as nodes join and leave the system (see text for details). Inset: throughput as the number of nodes is further increased to test the system's scalability.

problems are amplified when working with slow devices such as smart phones.

In the SPACE framework, there is instead an independent population for each active evolutionary experiment, maintained by the initiator of that experiment. This guarantees that computation is not wasted (it would be easy to avoid that the same fitness evaluation is unduly performed multiple times), and every fitness computed by a peer is actually usable (although partially finished evaluations are still lost and rescheduled to an active peer).

Furthermore, the SPACE framework was designed with the idea that multiple different experiments, initiated by different users, can be performed concurrently on the same network. Other solutions (as they are implemented now) do not offer this opportunity, because they host the population on a central server and therefore would require a separate server for each concurrent experiment, see for instance the EvoSpace framework described in [12].

Finally, because the abstraction provided by the platform is very thin, it allows users to implement the fitness evaluation in any language supported by the `Socket.io` protocol. This allows for the creation of a heterogeneous pool, which mixes nodes running implementations in multiple languages. For example, as we have demonstrated, it is possible to combine user nodes running the in-browser, JavaScript implementation with virtual machines on the "cloud" running an optimized C++ version. This architecture also allows for advanced users to install a more efficient version while other users use the installation-free (but less efficient) solution.

The current architecture is not without its drawbacks, however. Indeed, there are two potential points of failure: the scheduling server and the initiator of a given experiment. Even if it is very unlikely the former would fail, it might occur quite frequently for the latter (if the internet connection breaks for a long time for example). This could be mitigated by several measures [13], for instance by the initiator frequently saving the current state of the program to disk, but this capability has not been implemented yet.

Additionally, the SPACE framework, as it is now, would not be well suited for applications with inexpensive fitness evaluations. The reason is simple: if it is faster to compute the fitness than to send a packet containing the individual and wait for the result, then there is no point in distributing the fitness evaluations. This problem is amplified by the latency between users and the scheduling server. To minimize the latency in our experiments, the server and the EC2 instances were both situated in the same AWS region, but it may be desirable to use EC2 instances in other regions where computation is less expensive.

5. CONCLUSION

In this work, we have introduced the SPACE framework for distributing expensive fitness evaluations across an elastic, heterogeneous pool of compute nodes that includes both the personal computers of users/volunteers running JavaScript software in their web browsers as well as a variable sized pool of cloud compute nodes running an optimized C++ version of the software. The utility of the system has been demonstrated in the context of Evolutionary Robotics using the RoboGen$^{\mathrm{TM}}$ open-source platform and the use of virtual machines from Amazon EC2.

While successful speed-up of complex evolutionary robotics experiments has been demonstrated, there are still possible improvements that could be made.

In particular, the scheduler, as described above (see Subsection 2.4) is not perfect. If a user a starts a new evolutionary experiment while there are many pending evaluations from other users then user a's experiment will take some time before it begins to obtain results. Even if user a shares some computing power, it will not be used for that experiment because it will be assigned the tasks from other users first. Additionally, a user a who may want to run an experiment rapidly could be willing to start many machines and connect them to the network. But if there are many other users running experiments, even if they are not contributing computing power to the system (by disabling the sharing of their computing power for example), user a's machines will be used to evaluate other users' individuals, and hence the scheduling would not be fair.

This unfairness could be remedied by attaching each machine/user to an account, and each account to one or more specific collaborative groups. This way the scheduler knows which user started a given machine, as well as how much each user has contributed to the overall computing power of the system. Also, it would be possible to limit the sharing of computing resources only to the groups the user belongs to. With this knowledge, a more fair algorithm could be implemented. For example, a Round-Robin approach could be used. Time would be cut into small slices where each slice belongs to a single user. During a given time slice, all peers would compute tasks from the user the slice belongs to, and the duration of a slice would be proportional to the computing power provided by the user. It would also be possible to implement more complex fair share schedulers such as those given in [10, 14].

Besides reputation and credit mechanisms, account authentication might also be used to provide a layer of security and privacy: in some sensitive applications, users might be reluctant to share their data with unknown peers. Group policies would allow a complete access control. Another layer of security might be implemented by adding encryption to the communication channels used in the system.

Finally, to fully take advantage of the capabilities of the SPACE framework, it is desirable to attract a large pool of users willing to volunteer their compute resources. Due to the installation free web-interface, this should be fairly easy to accomplish, but it will be necessary to (a) offer problems that are of interest to users, (b) advertise experiments on popular web platforms such as `reddit`[27], and (c) offer specific rewards and/or incentives that are tied to the number of computations that a user's computers perform.

6. ACKNOWLEDGMENTS

This work was supported by the Swiss National Science Foundation under grant no 141063 and the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 308943 (FET-OPEN Insight project).

7. REFERENCES

[1] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[2] M. Arenas, P. Collet, A. Eiben, M. Jelasity, J. Merelo, B. Paechter, M. Preuß, and M. Schoenauer. A framework for distributed evolutionary algorithms. In *Parallel Problem Solving from Nature - PPSN VII*, volume 2439 of *Lecture Notes in Computer Science*, pages 665–675. Springer Berlin Heidelberg, 2002.

[3] J. E. Auerbach, D. Aydin, A. Maesani, P. Kornatowski, T. Cieslewski, G. Heitz, P. Fernando, I. Loshchilov, L. Daler, and D. Floreano. RoboGen: Robot Generation through Artificial Evolution. In *Artificial Life 14: Proceedings of the Fourteenth International Conference on the Synthesis and Simulation of Living Systems*, pages 136–137. The MIT Press, 2014.

[4] J. E. Auerbach and J. C. Bongard. Environmental influence on the evolution of morphological complexity in machines. *PLoS computational biology*, 10(1):e1003399, 2014.

[5] E. Bozdag, A. Mesbah, and A. Van Deursen. A comparison of push and pull techniques for AJAX. In *Web Site Evolution, 2007. WSE 2007. 9th IEEE International Workshop on*, pages 15–22. IEEE, 2007.

[6] F. S. Chong and W. B. Langdon. Java based Distributed Genetic Programming on the Internet. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1229, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann. Full text in technical report CSRP-99-7.

[7] O. Derby, K. Veeramachaneni, and U.-M. O'Reilly. Cloud driven design of a distributed genetic programming platform. In *Applications of Evolutionary Computation*, volume 7835 of *Lecture Notes in Computer Science*, pages 509–518. Springer-Verlag, Berlin, Heidelberg, 2013.

[8] T. Desell, M. Magdon-Ismail, B. Szymanski, C. Varela, H. Newberg, and D. Anderson. Validating evolutionary algorithms on volunteer computing grids. In *Distributed Applications and Interoperable Systems*, volume 6115 of *Lecture Notes in Computer Science*, pages 29–41. Springer Berlin Heidelberg, 2010.

[9] J. Duda and W. Dłubacz. Distributed Evolutionary Computing System Based on Web Browsers with JavaScript. In *Applied Parallel and Scientific Computing: 11th International Conference - PARA*, pages 183–191. Springer Berlin Heidelberg, 2012.

[10] C. Dumitrescu and I. Foster. Usage policy-based cpu sharing in virtual organizations. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 53–60, Nov 2004.

[11] P. Fazenda, J. McDermott, and U.-M. O'Reilly. A Library to Run Evolutionary Algorithms in the Cloud Using Mapreduce. In *Applications of Evolutionary Computation*, volume 7248 of *Lecture Notes in Computer Science*, pages 416–425. Springer-Verlag, Berlin, Heidelberg, 2012.

[12] M. García-Valdez, L. Trujillo, J.-J. Merelo, F. Fernández de Vega, and G. Olague. The EvoSpace Model for Pool-Based Evolutionary Algorithms. *Journal of Grid Computing*, 13(3):329–349, 2014.

[13] D. L. González, F. F. de Vega, and H. Casanova. Characterizing fault tolerance in genetic programming. *Future Generation Computer Systems*, 26(6):847 – 856, 2010.

[14] G. Henry. The unix system: The fair share scheduler. *AT T Bell Laboratories Technical Journal*, 63(8):1845–1857, Oct 1984.

[15] B. Javadi, D. Kondo, J.-M. Vincent, and D. P. Anderson. Mining for statistical models of availability in large-scale distributed systems: An empirical study of SETI@home. In *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS'09)*, pages 1–10, 2009.

[16] J. L. J. Laredo, P. Bouvry, S. Mostaghim, and J. J. Merelo. Validating a Peer-to-Peer Evolutionary Algorithm. In *Applications of Evolutionary Computation*, volume 7248 of *Lecture Notes in Computer Science*, pages 436–445. Springer Berlin Heidelberg, 2012.

[17] J. L. J. Laredo, A. E. Eiben, M. Steen, and J. J. Merelo. EvAg: a scalable peer-to-peer evolutionary algorithm. *Genetic Programming and Evolvable Machines*, 11(2):227–246, 2009.

[18] S. A. Ludwig and D. Dawar. Parallelization of Enhanced Firework Algorithm Using MapReduce. *International Journal of Swarm Intelligence Research*, 6(2):32–51, Apr. 2015.

[19] J.-J. Merelo, M. García-Valdez, P. A. Castillo, P. García-Sánchez, P. de las Cuevas, and N. Rico. NodIO, a JavaScript framework for volunteer-based evolutionary algorithms: first results. *ArXiv e-prints*, Jan. 2016.

[20] J. Merelo-Guervos, P. Castillo, J. Laredo, A. Mora Garcia, and A. Prieto. Asynchronous distributed genetic algorithms with Javascript and JSON. In *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 1372–1379, June 2008.

[21] K. Meri, M. G. Arenas, A. M. Mora, J. J. Merelo, P. A. Castillo, P. García-Sánchez, and J. L. J. Laredo. Cloud-based evolutionary algorithms: An algorithmic study. *Natural Computing*, 12(2):135–147, June 2013.

[22] S. Nolfi and D. Floreano. *Evolutionary Robotics: The Biology,Intelligence,and Technology.* MIT Press, Cambridge, MA, USA, 2000.

[23] J. Park. JavaScript API Misuse Detection by Using Typescript. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity*, MODULARITY '14, pages 11–12, New York, NY, USA, 2014. ACM.

[24] V. M. Rivas, J. J. Merelo, G. R. López, M. G. Arenas, and A. M. Mora. An Object-Oriented Library in JavaScript to Build Modular and Flexible Cross-Platform Evolutionary Algorithms. In *Applications of Evolutionary Computation*, volume 8602 of *Lecture Notes in Computer Science*, pages 853–862. Springer-Verlag, Berlin, Heidelberg, 2014.

[25] K. Tan, A. Tay, and J. Cai. Design and implementation of a distributed evolutionary computing software. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 33(3):325–338, Aug 2003.

[26] K. C. Tan, M. L. Wang, and W. Peng. A P2P Genetic Algorithm Environment for the Internet. *Commun. ACM*, 48(4):113–116, Apr. 2005.

[27] T. Taylor, J. E. Auerbach, J. Bongard, J. Clune, S. Hickinbotham, C. Ofria, M. Oka, S. Risi, K. O. Stanley, and J. Yosinski. WebAL Comes of Age: A review of the first 21 years of Artificial Life on the Web. *Artificial Life*, 2016. Under review.

[28] D. R. Thoman, L. J. Bain, and C. E. Antle. Inferences on the parameters of the Weibull distribution. *Technometrics*, 11(3):445–460, 1969.

[29] K. Veeramachaneni, I. Arnaldo, O. Derby, and U.-M. O'Reilly. FlexGP - Cloud-Based Ensemble Learning with Genetic Programming for Large Regression Problems. *Journal of Grid Computing*, 13(3):391–407, 2014.

[30] D. Wilson, K. Veeramachaneni, and U.-M. O'Reilly. Cloud scale distributed evolutionary strategies for high dimensional problems. In *Applications of Evolutionary Computation*, volume 7835 of *Lecture Notes in Computer Science*, pages 519–528. Springer Berlin Heidelberg, 2013.

[27]https://www.reddit.com/