FOUNDATIONS OF COMPUTING AND DECISION SCIENCES
Vol. 42 (2017) No. 1
DE GRUYTER OPEN
DOI: 10.1515/fcds-2017-0001
ISSN 0867-6356
e-ISSN 2300-3405

BROWSER-BASED HARNESSING OF VOLUNTARY COMPUTATIONAL POWER

Tomasz FABISIAK, Arkadiusz DANILECKI*

Abstract. Computers connected to internet represent an immense computing power, mostly unused by their owners. One way to utilize this public resource is via world wide web, where users can share their resources using nothing more except their browsers. We survey the techniques employing the idea of browser-based voluntary computing (BBVC), discuss their commonalities, recognize recurring problems and their solutions and finally we describe a prototype implementation aiming at efficient mining of voluntary-contributed computing power.

Keywords: voluntary computing, desktop grids, browser-based computing, internet computing, survey, crowdsourced supercomputer, peer-to-peer computing

1. Introduction

With regard to a computing power, modern personal computers are comparable to supercomputers from two decades ago. In 1996, the last supercomputers in the list achieved between 4.6 to 6.5 Gflops (depending on the measure used)[1] – a performance easily matched by laptops such as Lenovo Y50, available for less than 1000$. By even the most conservative estimates there are more than one billion personal computers currently in use[2],

* Institute of Computing Science, Poznań University of Technology

1 http://www.top500.org/list/1996/11/?page=5

2 http://www.worldometers.info/computers/. Some estimates give a number of more than two billions personal computers in use worldwide.

from which we can conclude global total computing power is in range of several exaflops. In Poland alone about ten million households have at least one computer, with total computing power in range of tens of petaflops. Most of the time, this power is left unused.

The scientific community long have wanted to harness this tremendous computing power [30]. Nowadays, there are at least two popular frameworks aiming at that goal: BOINC (Berkeley Open Infrastructure for Network Computing) [3] and XtremWeb [28]. In these and similar approaches a user is often required to register at a web page. Later the user must download and install a specialized software. The software may be then executed by the user or scheduled to run in a background, usually when the user's machine is idle (for example, it may run as a screensaver), downloading the jobs to be computed from some centralized server and later uploading back the results.

The undeniable achievements of BOINC- and XtremWeb- based projects notwithstanding, we note that they look less impressive when comparing the combined computing power of all their projects (e.g. 10.5 PetaFLOPS average computing power available in case of BOINC) to the computing power theoretically available on the web. Several factors limit their success. The required software installation step may be prohibitively difficult to many users, despite the efforts to make the task less daunting [15]. Many users are afraid of installing unfamiliar software, fearing malware and spyware [18]. Many may be simply lazy. Finally, projects' outreach is limited by lack of awareness of projects' existence and lack of appeal to the average user.

There are estimated 3.6 billion internet users in 2016[3] (some using mobile appliances or sharing the same machines). Those users spent a lot of time on WWW, an activity they enjoy and which does not require any technical prowess. When browsing web sites, especially those providing a lot of content without requiring user's reaction ("high screen time and low user interaction" websites [50]) most of the users' machine computing power is unused, since even frequent human interactions introduce large latencies.

The appealing idea is browser-based voluntary computing (BBVC): to create web applications solving scientific problems, automatically executed by a browser during the user's visit to a particular website. In this approach, users are not required to install any software and may even not realize they contribute their computing power (resulting ethical problems are briefly discussed in Section 4.4). The impact of additional work on browsing experience is usually considered negligible and not discernible by users [80].

Unfortunately, in contrast to voluntary computing in general, we are aware of no up-to-date complete survey concentrating on BBVC only. The main goal of our paper is to fill this gap by reviewing more than 40 articles dedicated to browser-based voluntary computing which appeared in the literature during the last two decades. Despite its huge potential, BBVC is still not as popular as we feel it should be, and we try to provide answers why. The second goal is to describe our ongoing work, a new voluntary computing platform, implementing our ideas addressing the main obstacles preventing BBVC from wider acceptance.

The rest of paper is structured as follows: we introduce the basic issues, concepts and vocabulary in the Section 2. We continue in section 3 with our main contribution: a survey of three generations of browser-based voluntary computing. We identify common problems

3 http://www.internetworldstats.com/stats.htm

and their solutions in Section 4. Our secondary contribution is contained in Section 5 where we describe our work-in-progress: a new BBVC platform PovoCop, its proof-of-concept prototype implementation and some initial experimental results. We discuss non-browser-based voluntary computing in Related Work (Section 6). Section 7 concludes the paper.

2. Characteristic features of browser-based voluntary computing

The idea of utilizing idle cycles of volunteering users is associated with many labels, denoting same or at least heavily overlapping concepts. We encountered names such as peer-to-peer computing [28][52], global computing [28][62], public resource computing [3], internet computing [75][52], crowdsourced supercomputer [79], desktop grid [75] and cycle stealing [28]. The name most commonly used seems to be "volunteer computing" or "voluntary computing" [13][75][4][26]. As mentioned earlier, in this paper we concentrate only on browser-based solutions, i.e. on a subset of voluntary computing, Henceforth we will use the name "browser-based voluntary computing" (BBVC).

Browser-based voluntary computing is a subset of distributed computing, differentiated by several characteristics of the environment.

First, there are two kind of the nodes in the system: the generally long-living servers and the machines controlled by volunteers – providing resources we want to use. Obviously, in BBVC we assume that every volunteer has a browser installed. Browsers provide execution environment common for the applications running in the system.

The second characteristics is that, in contrast to e.g. grids and clusters where work is "pushed" from server to slave nodes, in voluntary computing the work must be actively pulled from server by workers [75]. The workers must find the server, inquire about the existing tasks, choose some of them for download and execution, and have to voluntarily upload the results to the server at time suitable for the worker. The fact that the worker volunteered for some task does not guarantee that it will actually complete the task.

The third distinguishing characteristic is that resources used in voluntary computing are ephemeral and unreliable in nature, and are not managed nor authenticated [26][75]. Especially in BBVC, users may connect and disconnect at will, and may (or might not) reappear later. The interaction between users and the web sites may last as short as ten seconds and is rarely longer than few minutes [14]. Resource characteristics often are unknown (i.e. it is not known in advance what computing power will be available for use) and, even if they were known, they could quickly become obsolete (as computers are in use and may occasionally experience bursts of activitity).

The distinguishing characteristics of computing environment pose a set of challenges inherent in voluntary computing in general [28][69], and those are applicable also for BBVC: accessibility, programmability, fault tolerance, security, scalability, usability, availability, heterogenity, adaptability/dynamicity.

Accessibility means that the proposed solution must be easily accessible to users irrespective of their machines' architectures and capabilities. Users should require no technical knowledge to share their resources. Projects should be attractive to the users irrespective of their demographic characteristics.

Programmability means that developers should be able to quickly create applications destined for BBVC. No specialized language should be required. *Fault tolerance* means that system should be tolerant of crashes and failures. Machines may disconnect at any time, and there may be network connectivity problems between server and volunteer's machines (see Section 4.6). *Security* means security of servers harvesting for public resources, and volunteer's machines should not be compromised (see Section 4.4). *Usability* means the ease to deploy and use. *Availability* means the solution should strive to provide service, regardless the failures, changes in load or users' capabilities.

Scalability means that the architecture should accommodate increasing number of nodes. System should be able to quickly distribute the work, using available computing power. In turn, that means that applications created for the system should be scalable as well. In practice, scalability demand limits the kind of possible problems which can be solved by BBVC.

Heterogenity means machines owned by volunteers may differ by hardware architecture and operating system. It would be tempting to detect the user's architecture and then to map compatible hosts and binaries. However, the disadvantage of downloading specific standalone application is the additional complexity in maintenance. This challenge is less relevant to the BBVC, as the languages of choice (Java or JavaScript) is generally machine-independent. Browser, however, may differ in their support for different standards, requiring developer to decide whether they want to limit their audience by introducing features supported only by some subset of available browsers, or constrain themselves to the lowest common denominator, forfeiting the possible advantages from using some more modern or advanced features.

Finally, *adaptability* (or dynamicity) means the solution should be aware that environment is ever-changing, as the number of nodes and their capabilities may vary over the course of execution.

In addition to the challenges and characteristics shared to some extent with other voluntary computing solutions, BBVC impose additional constraints resulting from the fact that the tasks are executed by the browsers. Browsers are generally not used for computations, but rather for searching for information or participating in social activities. This means tasks shipped for workers for execution must not disturb the users during their normal activities. The browser may in fact terminate an activity deemed unresponsive. Moreover, task execution time should be short, because computation executed by a given user is terminated as soon as the user leaves the website, which – on average – happens after at most several minutes.

Another constraint imposed by the environment are substantial communication costs, conspicuously large when compared to more tightly-coupled computing environment such as clusters. The choice of problems is further constrained by problems with direct task-to-task communication mentioned in Section 4.2. Ideally, single task execution time should be short (at most several minutes), because of average short time spent by users at websites.

Moreover, browsers may limit resources available for the computation, especially when executed on mobile devices. In one solution [24] targeting the mobile devices, the browser on one of the tested smartphones hung for larger problem instances. In another [57] authors find out clients could not handle input data chunks larger than 100MB because of memory management issues.

Therefore, the ideal jobs for BBVC should be easily divisible into small, computationally independent tasks – or, at least, if the computation runs in phases, tasks should be computationally autonomous within each phase. The general problem categories considered by researchers working on BBVC were called piecework computations [16], master-worker [26], coarse-grained [22], malleable applications [19] or bags of tasks [23].

During our survey we encountered the following problems identified as well-suited for BBVC: evolutionary/genetic algorithms [23][25][44][58][64], simulated annealing [23][64], colony optimization [23], particle swarm optimization [23], artificial bee colonies [23], brute force search [64], cryptography [9][64][80], game-tree evaluation [64], map-reduce problems with computation-intensive map phase [22][47][50][57][67][79], monte-carlo simulations [19][54][55] branch-and-bound [16] and its limited subset, bulk-synchronous-parallel (BSP) [68], and, finally, the algorithms based on directed acyclic graphs [19].

BBVC could be used in social media to facilitate image tagging for improving users' browsing experience or reducing load on servers. Hence, the following problems were suggested: face recognition [80][83], logo recognition [83], image scaling and sentiment analysis [80]. Similarly, it was suggested that near similarity detection can be used for detection of copyrighted media [83], and images can be classified by neural networks [57].

During experimental evaluation of the surveyed proposals few problems were regularly reappearing. Raytracing was very popular, starting with one of the first applet-based approach (distributed applet-based massive parallel processing, DAMPP) from 1997 [77]. Other work which suggested raytracing as example problem are [16][23][71], though the author of [71] admitted his test results were not encouraging.

Other problems which in reviewed works were used as benchmarks ranged from the very simple, like searching for prime numbers [22] and large internet number 1234567890123456789 factorization [68], to genome analysis [79] and protein sequence alignments using Needleman-Wunsch algorithm. [19]. Other problems used in experiments were: DES code breaking [10], matrix multiplication [16][68], Traveling Salesman Problem [16][24][67], Mandelbrot generation [69][70], searching for Mersenne primes using Lucas-Lehmer primality test [56][29], Collatz hypothesis verification [56], the factorization of large integer into two primes and Pearson correlation evaluation on set of genetic samples [9].

For those problems linear or even superlinear speedup [16][57] was often achieved, with experiments employing from 80 [67] through 200 [47] up to 400 clients [22]. A number of researchers [23][47][54][55][57][71] noticed the problem of "speedup decay": adding more workers caused speedup to level off or even drop. With more workers, each of them received less work, meaning communication costs imposed by work distribution overshadowed the gains. In solutions using centralized architecture, the central server could become overloaded with the increased number of workers [57][67]. Finally, the bandwidth depletion was suggested as another explanation of the phenomenon [47].

After presenting the problems well-suited for BBVC and discussing the characteristics of the environment and the challenges associated with voluntary computing in general, we are now ready to review the browser-based proposals which appeared during the last two decades.

3. Three generations of browser-based voluntary computing

All BBVC solutions we have reviewed are similarly structured, with several recurring roles in their architectures. We start by identifying common roles and by specifying vocabulary to be used hereafter.

The fundamental concepts are *jobs* and *tasks*. By a job we understand a running application solving one particular problem instance (with particular set of parameters). One problem may concurrently be solved by several different jobs. Jobs are divided into many tasks. A job is finished only after all tasks belonging to the job are finished.

First role deals with job creation and submission, and inspection of the results. It's called *client* in Javelin [16], *owner* in Alchemi [52] and *buyer* in POPCORN [64]. We will call this role *job owner*.

Second role is called *host* in Javelin, *executor* in Alchemi, *seller* in POPCORN, *helper computer* in [29]. This is a party voluntarily contributing its computing resources to solve the problems submitted by job creators. One node may solve few tasks (possibly from different jobs) at the same time, using *workers* – threads executing one particular task. Because threads only appeared in the third generation, in our survey we will call this entity *worker node*, while in our prototype described in Section 5, we will use the name *plantation* (with many workers).

Third role describes a party mediating between job owners and worker nodes. It's called *broker* in Javelin, *market* in POPCORN, *manager* in Alchemi and *master* in QADPZ. It is sometimes divided in two parts: one plays simply a role of a directory (e.g. *distribution computer* in [29] or Knitting Factory's *directory service* in [6], *broker name service* in Javelin 2.0 [63]) and contains addresses of second part, servers willing to dispatch the jobs (e.g. *computation servers* in [29]). We will use *job scheduler/dispatcher* terms.

The job schedulers/dispatchers are commonly based on HTTP server, but they could implement only part of HTTP server functionality. For example, in [29] computation server implements light-weight web server, which only answers the workers' requests. In [6] a lightweight web server is used to distribute java classes to browsers.

We will now review BBVC platforms, solutions and proposals, grouping them in the three generations, each with distinguishing set of characteristics. The main differences between the generations are presented in the Table 1 below.

Table 1. The comparison of three generations of BBVC

	The 1st generation	The 2nd generation	The 3rd generation
Tasks written in	Java applets	JavaScript	JavaScript
Predominant http server	No dominant	Mostly Node.js	Mostly Node.js
Thread support	Yes (via Java virtual machine)	none	Yes (via WebWorkers)
REST support	none	occasional	standard
Communication protocol	HTTP, UDP, TCP, Java RMI	HTTP, AJAX	HTTP, AJAX, WebSockets

3.1. The first generation: Dead applets society

In the first-generation of browser-based voluntary computing proposals, jobs were implemented as Java applets (called "distriblets" in [29] and "computelets" in [64]). Typically, the applications were listed at a website. User had to choose which computation to run, which usually required him or her to click on the link and download an applet (e.g. [6][7][29]). The applets either contained the necessary input data or connected to a server requesting work (possibly using UDP instead of HTTP as in DAMPP [77]). After finishing the work, the applet would send back the results and ask for another task to execute. Some solutions allowed offline computation: in [29] and [16] a user could download applet, disconnect, and when later he or she would reconnect; the applet would send back the computed results.

To the best of our knowledge, the first implementation of the idea was Charlotte [7], appearing in 1996. The applications had to be written especially for Charlotte and had limited scalability (as its intended use was limited to the authors' home university). Charlotte provided Distributed Shared Memory abstraction, via wrapper classes of common types and explicit "set" and "get" function calls. Programs had to be registered at the project's website. User visiting the site could click one of the listed URLs, loading and executing "worker" applets. In addition, a standalone Java programs, *manager*, resided on a web server's site, one for each distributed application. The web was used only for coordination between clients and job creators. Charlotte programs could indirectly communicate via *managers*. In their followup work [6] they described a KnittingFactory directory service, implemented as JavaScript-enriched HTML page. JavaScript constructed new URL, with search state passed as a part of the created URL, and instructed the browser to follow the URL. Searching took place solely on users' browsers.

We will now describe three first-generation solutions in more detail. Two of them were chosen because they are the most known in the literature and may be considered the early paragons of browser-based solution: Javelin [16] and Bayanihan [68]. In addition, we decided to discuss POPCORN [64] because of its interesting economy-based interaction between job owners and worker nodes.

Javelin is a prototype implementation of the SuperWeb architecture [2]. The architecture consists from brokers, hosts and clients and, in general, does not depend upon Java/browser combination. *Client* create Java programs, applets, which they then register at *broker*. Applets belonging to the same application could communicate via broker. Programs are not sent to broker; only their URL addresses are sent. During registration clients estimate how much resources will be used by their tasks.

Hosts (i.e. worker nodes) were the machines willing to provide their resources (CPU power, storage, memory) to clients. Hosts had to register at brokers. Hosts could specify when their resources may be used and what percentage of a given resource would be available for use. Their SuperWeb architecture demanded the hosts to provide a sandboxed environment in which tasks would be safely executed. This was their primary motivation for choosing applets and browsers, as they obviously provide the required safe environment.

Brokers coordinated between clients and hosts. They ran accounting, giving credits to users. Those credits were envisaged to be later exchanged for "cyber money, access to software licenses, cheap internet access or free cpu time". When host registered at broker, it received URL of jobs submitted by clients. Host downloaded an applet from a specified

URL and ran it within a browser until it left broker's site (in a followup [16], they specified that computation may be carried offline and the results could be uploaded after reconnect). Brokers also scheduled the tasks and managed fault-tolerance. To guide scheduling and mapping tasks and hosts, microbenchmarks were used to estimate host capabilities. It was envisaged that they would also verify the results from the hosts. In non-applet based implementation, broker would also map between binaries and compatible hosts.

Brokers initially were implemented as HTTP servers extended with servlets or CGI. In Javelin++ brokers were standalone Java applications, residing on long-living machines with regular internet access. In later versions distributed broker network was created. A primary broker was started by user using a configuration file. In case the primary broker was overloaded, it chose one of the connected hosts, preempted all applications running on that hosts and sent it its own broker code, causing the host to become a new secondary broker.

Javelin implemented few high-level programming primitives such as Linda-like tuple space and barriers, and small distributed shared memory (size of one *double* or *int*) in Javelin 2.0. To increase the speedup, they experimented with TCP sockets. They concluded that they have very high creation cost, but are on level magnitude faster than HTTP based communication. In Javelin++ RMI was used instead (which limited their browser base, as at that time, not all Java virtual machines shipped with browsers supported Java RMI).

By Javelin 2.0, name service was added to architecture: JavelinBNS (broker name service). Brokers would register at JavelinBNS, knowing the addresses of names serves from configuration file or finding them at runtime. Clients and hosts asked name servers for broker list, then used RMI-ping to find the best broker.

From the very beginning the Javelin team stressed that the tasks could be standalone programs. In time, their architecture evolved, allowing applications running in background as screensavers in Javelin++ and moving towards standalone Java programs in Javelin 2.0. In Javelin++ applets still could be used, provided that the users would use browsers supporting RMI-based communication. The move was motivated by new features in Java JDK, which allowed imposing security constraints on any Java application, causing applets to lose their main appeal.

The authors also discussed the possible economic models to be used. For example, they envisaged that when scientists would required more resources to carry the tests for their paper to be submitted to a conference with an approaching deadline, they would be willing to pay more for the resources [2].

The second most often mentioned first generation system is *Bayanihan* [68][69][70]. In Bayanihan's architecture, the central element consisted of two entities residing on the same node: HTTP server serving the Java classes and the Java application. The Java application created *problem objects* representing distributed computation, mapped one-to-one with *program objects*. Each program object created a *manager* and pool of data objects. For example, the *work manager* maintained work data pool and results pool, while *watch manager* monitored the work progress. In addition, *advocate objects* served as proxies to remote clients. Each object knew how to handle themselves: for example, work objects contained *doWork* method invoked by workers.

Volunteers could choose any problem from the problem table maintained by a work manager. Tasks were submitted eagerly to idle clients in round-robin fashion, eventually resulting in natural redundancy when there were more clients than jobs.

Clients could be applets or standalone Java applications. Each client was a chassis, containing an engine. Watcher clients used watch engine, which contacted a watch manager at the server side in order to investigate problem's statistics, computation progress etc. Worker clients would use work engine instead.

Workers had to invoke a remote method of a worker advocate, which then communicated with worker manager and sent back a work object. Worker client then called *doWork* method of the received work object. When the computation was finished, it called *sendDone* remote method of the work advocate. Result was then put into a result pool. After work pool was empty, worker could request another bunch of work objects; if a programmer wanted worker to to wait and synchronize with other workers, all he or she had to do was to reimplement the appropriate interface methods.

The most interesting feature of *POPCORN* [64] was its "capitalist" approach, visible even in the chosen nomenclature. The parties seeking for the resources were called *buyers*, machines voluntary contributing their resources were named *sellers*, and the server allowing their cooperation was called *market*.

Applets in POPCORN contained both the code and the data and were called *computelets*. Computelets were sent by buyers into the market. Sellers registered at the market, received computelets, executed them, and sent back the results to the market, which forwarded them to the buyer.

The jobs – *computation packets* – contained also information about their pricing and information how to handle different events, such as receiving results from sellers or premature termination. POPCORN provided rudimentary fault tolerance: it guaranteed that for every job, eventually either its *completed* or *failed* method would be called. Computelets losses were detected via timeouts. Programmers were responsible for result verification. There were no direct communication between the computelets.

In POPCORN, sellers were selling "java operations". Computelets invoke periodically high-priority thread – basically a microbenchmark – which computed few operations and reported the performance back to the marker. Sellers were paid by "popcoin". Each user had its account, and could spent popcoins on resources, or earn them by renting his or hers resources to other users. In theory those popcoins could be converted to real money.

Buyers offered their price for resources when advertising their computelets. Whenever a user wanted to sell his CPU time, he visited the page with an applet and clicked the appropriate "execute" button. There might be many potential buyers for his CPU time. The highest bidders were chosen, selected by vickrey auction (where buyers could specify their lowest/highest prices for the resources they want).

Additionally, the authors suggested that some users may become "publishers": they could create web pages with animated POPCORN logo. Visitors to this page inadvertently would execute applets executing computelets. The publishers would receive popcoins, hopefully trading interesting web content for visitors' CPU power.

The idea of brokers earning a profit from mediating between hosts selling their CPU power and clients wishing to buy the power was also discussed in *Unicorn* [66], the last BBVC solution based on Java applets we are aware of.

3.2. The second generation: Enter the JavaScript

While the first generation has shown that the browsers may be used for computation, its success was quite limited. For example, in tests carried by [29] to compute Mersenne primes (using Lucas-Lehmer primality test), during 9 days on average 4.7 hosts per hour participated in the project (maximum 15 hosts). Similarly, tests shown for Bayanihan were for 10 clients only [70]. Despite that, Fox et al predicted in 1997 that by 2007 web would be used as supercomputing platform [30], envisaging Java agents traveling through the net and spawning new slave agents to solve computational problems. Instead the idea by 2007 fell into oblivion, which provokes the question, why?

The performance could be only one part of the answer. In 1990s Java applets were quite fast compared to JavaScript – even in 2008, they were 10 to 100 times faster, spurring Klein and Spector to suggest [44] that their JS-based solution could be augmented by Java applets, for performance reasons – and yet nowadays they are virtually absent from the WWW. It was suggested that slow initialization of Java applets could be one of the reasons of eventual demise of the first-generation [61].

Another reason could be the inherent laziness of web users. First-generation often required users not just to click the links to start the applets, but – because of default browser settings – caused popup windows to appear, asking users to allow the applets to be run.

One contributing factor could be faster web application development when using JavaScript. Another, quite elusive and imponderable factor, is the perceived "coolness" of language. We speculate, based on our experiences and discussions with web developers, that Java have lost "coolness" factor, possibly contributing to the applet-based BBVC demise.

Whatever were the reasons, the idea of browser-based voluntary computing fell into the obscurity to the point that when it was reinvented in 2007, many authors were seemingly unaware of the existence of earlier proposals [9][44][45].

By second-generation browser-based voluntary computing solutions we understand approaches based on JavaScript, but without modern features such as thread support. While JavaScript was created already in 1997, it was AJAX (Asynchronous JavaScript and XML) [32] techniques which, starting in 2005, enabled development of new set of browser-based solutions. In second generation approaches a user can trigger the computation merely by visiting the page, this time without any popup window appearing (as it was often the case with applet-based approaches).

In 2007, three solutions based on JavaScript and AJAX appeared [9][44][45]. In each of them, JavaScript code embedded within HTML page was executed by browser when user visited the project site, using AJAX to communicate with server (to get tasks to execute and to return back the results), passing XML documents as request arguments.

On the conceptual level second-generation resembled earlier applet-based approaches. For example in proposal by Boldrin et al [9], there were clients (wokers) and server responsible for partitioning the problem, distributing the tasks, collecting and reassembling (i.e. aggregating) the results and keeping track of the clients' status.

Another solution proposed in 2007 was by Klein and Spector [44]. As far as we know, it was the first solution to implement genetic algorithms. They have implemented a PushScript, a JavaScript implementation of Push3 – a special-purpose language for genetic programming. Client-side implemented PushScript and few utility libraries. Users would

visit a web page, unwittingly running a client-side code: fetching the fitness tests from server, running them within PushScript engine and sending back results to server, and then – after a small delay – asking for new problem to solve. They implemented server-side code mainly in PHP, except Push3 engine which was in C++.

The third solution, *RABC* (Riken AJAX Browser Computing) came from Japan [45]. RABC aimed at creating simple environment SETI@Home-like projects. The distinguishing feature of this proposal was the fact that the kernel agent – JS code embedded within a HTML page – was trying to adapt its behavior (requesting tasks, returning the results) to network bandwidth.

The second proposal we know to use the genetic algorithms appeared in 2008 [58]. They used JSON instead of XML, hence the name *AGAJAJ* (Asynchronous Genetic Algorithm with JavaScript and JSON). AGAJAJ employed one central server. Volunteers visiting the server's page downloaded a code (written JavaScript) which fetched the parameters from the servers, ran several preset number of generation and then sent back the results (best individual according to fitness function) to the server. Server chose the best individual from all workers and sent it back for another round of generation. Eventually clients were shown the best results and were given chance to restart old experiment or start a new one.

The apparent performance problems with JavaScript led to create at least one Flash-based solution, Online Community Grid (OGC) [61]. Adobe Flash speed was comparable to Java applets. OGC envisaged webmasters putting a small flash widget on their sites. Users visiting the site for the first time would be able to choose whether to participate in one of available projects. Because work would be terminated whenever user would leave the site (by closing a tab, surfing to another page or closing the browser) the running processes were checkpointed every few seconds, in order to allow the work to be resumed when user would later return to the page.

We consider the second-generation to end at 2009, with two proof-of-concept implementations, of distributed hashing in [8] and of map-reduce by a blogger and web developer Ilya Grigorik [34]. In Grigorik's prototype worker node first makes a GET request to job-server, and receives an allocated unit of work with redirection to a prepared site containing mostly JavaScript code to execute. This code would contain typical map-reduce functions and then would emit the results back to the server by creating invisible form, filling it with the results, and submitting it.

Second-generation suffered from the JavaScript's lack of support for threads. For example, in [9] authors discussed at length how to re-engineer the programming loops in order for web pages to being responsive. Their solution was a creation of special function, which run only few iterations of the loop and then rescheduled itself for further future execution.

Another problem was poor performance of browser's JavaScript engines. In 2007, JavaScript was between 9.8 to 23.2 times slower (depending on the platform) than Java code for a simple mathematical task of random numbers generation [45]. Compared to C, the results were even worse: the browser-based JavaScript engine was between 20 to 200 times slower (depending on platform and browser) [44].

Developers also had to fight with JavaScript same-origin policy (where script from some address domain, e.g. put.poznan.pl, for security reason could not visit URLs from different domain, say github.pl – see Section 4.1). Finally, there had not existed standardized, mature and popular solutions for direct browser-to-browser communication, preventing any direct

communication between the tasks (bypassing the proxies). Fortunately, all those problems are being addressed by new emerging standards and technologies, leading to what we labeled third-generation BBVC.

3.3. The third generation: Fast and the furiously multithreading

We categorize solutions appearing after 2010 as the third generation of browser-based voluntary computing. One reason is that starting with Google's Chrome v8 in 2008, new JavaScript (JS) engines with just-in-time (JIT) compilers allowed JS code to rival Java [19]. When compared to Java, JavaScript was found to be slower only by about one third in math-intensive computation [60], though significant speed differences may exist between JS engines of different browsers [47].

However, the better performance is not the only characteristic of the post-2010 works, though it surely motivated the recent surge in popularity of BBVC. We will now enumerate the features common to many recent proposals, which – taken together – justify in our opinion grouping them all in a common "third-generation" category.

First, the creation of WebWorkers standard in 2010 finally allowed multithreading in JS applications. While absent from some works, WebWorkers might be safely assumed to be used by all recent third-generation solutions (e.g. [54][55][57][67][80]).

Second, in most of the proposals we've found, Node.js HTTP server is used [1][23][25][31][46][49][53][54][55][57][59][60][79][80], though few [22][50][67][71] approaches use Apache, one used lighttpd [8] and some [19][40] do not specify the server used. Node.js [73] is a JavaScript HTTP server, based on asynchronous I/O event model (with no threads), allowing nevertheless high performance and throughput.

Third, Servers often expose Restful interface [19][23][31][53][59][60][71][79], allowing anyone, in most basic form, to HTTP GET the tasks to execute and POST (or PUT) the results of the computation. Server-side code in third-generation is usually implemented in JavaScript (if the solution uses Node.js), though few [8][44][50][71] use PHP, and in one case [67] it is written in Python. The server implementations usually use opensource database backends, such as Redis [1][79], MongoDB [53][79], MySQL [8][40][50][67][79] (additionally CrowdCL [53] claims it can easily support MySQL) and postgresSQL [58][79]. QMachine [79] can use all of those backends plus sqlite and couchDB.

In third-generation, user typically loads a page with JavaScript, which then request work from the server using AJAX technique (sending GET HTTP request with XmlHTTPRequest method). The initial code may be only a stub which must first load an actual task code from the appropriate server. After a computation is finished, the results are sent back in form of XML (or JSON) document to server by AJAX, ideally by invoking POST or PUT HTTP method on some specified URL. Sometimes [23][46][49][57] the communication between worker node and servers uses WebSockets. Jobs in a number of solutions [1][47][50][57][67][79] use the Google's map-reduce concept [21].

We start our survey of third-generation by discussing *QMachine*, as it is arguably a state-of-the-art implementation [79].

There are three parts in QMachine's architecture: worker nodes (browsers), API server and a web server. The API server functions as job dispatcher/scheduler. It is essentially

Restful, accepting HTTP requests formatted in JSON. Using those requests one can GET job status, GET job or POST a new job to a server. It is implemented as a module of Node.js, with support for five different opensource databases: redis, mongoDB, sqlite, postgresSQL, couchDB. The web server implements presentation layer and serves web pages. Worker nodes have the usual role of executing the tasks.

To create jobs for QMachine, application developers have to reimplement the QM class, with *submit*, *map* and *mapreduce* functions. *Submit* allows a remote execution of single function on worker, while *map* is dedicated method describing invocation of many functions on set of data. Finaly, *mapreduce* function describes map-reduce [21] pattern. The *map* function may have set of input data URLs (in the example experiment, with twenty URLs describing genomes of different streptococcus pneumoniae).

QMachine uses CORS [76] (cross-origin sharing) in order to allow the code to import libraries and the data from third-party servers. The example job described in the paper get jmat library from Google Code, usmlibrary from GitHub and input data (genome for the computational experiment) from NCBI. Volunteers pull the task descriptions and start executing them after downloading the required code and data. Individual workers may download the data from different servers, using potentially different communication links – not just computation is partitioned, but also data transfer bandwidth. Results are sent back to the QMachine server. The job owner polls for the updates in the task status and eventually retrieves the results.

To assist programmers solving concurrency problems with asynchronous data transfer QMachine provides special library (quanah). In addition of JavaScript, they support also CoffeeScript (essentially, a pretty-ish language wrapper for JS).

The authors described the impressive experiment carrying DNA sequence analysis of bacteria genomes. It lasted 12 months, and involved 2100 different IPs from 87 countries issuing 2.2 million API calls to QMachine.

Another typical third-generation solution is MRJS [67], one of the first to use WebWorkers. MRJS is based on map-reduce paradigm. Computation is divided into two phases, requiring modicum synchronization between the workers.

Job owners submit jobs via simple web interface. Input data is divided into uniformly sized parts, called chunks. It is the job owner's responsibility to specify the number of chunk into which input data will be divided.

The workers first receive the task code and map chunks from JobServer with HTTP GET request. Results are sent back via HTTP POST. One chunk may be assigned to many workers, and majority-voting is used for result verification. JobServer aggregates the results from the workers and sends them reduce chunks in the second phase. The second phase may not proceed for a given job until server finishes the aggregation phase.

The input data (map chunks) may be stored on an external server. In such a case, address of input data is contained within a chunk. Because of JavaScript same-origin policy, JobServer must function as a proxy for external data server, forwarding data to workers when necessary. Only after finishing the tests authors discovered CORS standard, which would remove the need for JobServer functioning as a proxy.

JSMapReduce created by Langhans et al [50] (based on earlier Langhans bachelor thesis from 2012) shares many similarities to MRJS. In contrast to [67], the chunks are variable-sized, with more capable workers getting more data. Moreover, chunks are compressed

before being send. In addition, server functions also as a worker to avoid wasting its CPU cycles. Jobs can be submitted, initiated and queried about status by a simple java client.

Server consists from bridge, master and scheduler. Bridge is a standalone socket server written in PHP, interacting with java clients via simple text protocol. Master gets input from bridge, preprocess the jobs, creates tasks and puts them into a database. It also registers workers who pass some minimal performance requirements, manages immediate results and aggregates the final results. Scheduler authenticates workers and gives them registration number (later sent alongside all the requests), assign the tasks using FCFS policy, detects timeouts and handles redundancy. A server-push (comet) technique is used to maintain the live connection between the workers and servers.

Allowing JavaScript to run codes based on map-reduce pattern was also subject of work in [1]. However, in a presented prototype the code runs on Node.js runtime and not on browsers. Migrating into the browsers is the target of the authors' future work.

NodIO [59][60] was created as a framework for creating experiments solving problems using evolutionary/genetic algorithms. The project site's visitors see a HTML page showing graphs, links and information about an experiment, and running workers. Each worker gets unique identifier sent along each request to a server. Workers have their own “islands” with a variable-sized population. After each hundred generations, the best individuals are sent using HTTP PUT request to a server; afterwards, workers GET random individual from a server.

Evolutionary algorithms were also used in framework proposed by Duda et al [25], extended later to use the mobile devices [24]. An interesting feature of their approach was gzipping the code and data before sending it to workers.

In [57] authors investigated the possibility of using the browser-based voluntary computing in intelligence research. They propose *MLitB*: a prototype machine learning framework. MLitB was created to facilitate cooperation between researchers and to allow volunteers to contribute their CPU power to solve machine learning problems. The authors stated that currently the most of the machine learning libraries are written in highly domain-specific languages, reducing their popularity. They envisaged that with MLitB the machine learning may become ubiquitous, with widely available models that could be trained and executed in any browser. That would increase both the popularity of the machine learning (their use would require no specialized software) and the results reproducibility (models implemented in JavaScript would be free to be inspected, corrected and tested by anyone).

MLitB architecture have separate master and data servers. Master server may host many machine learning projects. Data server is a lightweight replacement for a image database and handles the data storage (namely the zipped images, as their use case is image classification). MLitB uses WebSockets for communication between servers and worker nodes, except the initial data exchange done via simple HTTP GET request – the reason is that the data is zipped, and authors state that zipped files are transferred faster over HTTP.

When browsers connect to a master server they run JavaScript-based *boss* worker. Users then choose the project in which they want to participate. That causes *slave* workers to run: threads to download the data, analyze the statistics, train the model etc. Workers receive more data that they actually need for training the models; the cached data is used when they receive new tasks. Trained models (neural networks) can be saved in and loaded from files formatted in JSON.

Model training is divided into the iterations. Each worker is told how long it should run, based on estimation of the network latency to the master. New worker nodes must wait for end of the iteration before they may join the computation. Unfortunately, in each iteration master waits for the slowest worker. Master, uploads and allocates new data initializes new workers, handles the reduce step, monitors latency, detects lost worker nodes via timeout and reallocates the data when necessary, and finally broadcasts new parameters.

Krupa et al [47] and their followup work Turek et al [74] proposed to create a search engine consisting of ensemble of browsers and two layers of servers. Browsers are responsible for lexical analysis and parsing, detection of URLs links in texts, processing the content and index building (e.g. of words in parsed pages). Servers distribute work. Each second layer server manages a subset of first layer servers, but in order for the system to be more fault tolerant, the second layer servers also know some randomly chosen servers managed by other second layer servers. That allows failover in case of failure of some second layer server.

In their proposal, search scripts are written first by experts in new formal language and then submitted to second layer servers, which rewrite them into JavaScript, containing definitions of search rules and implementation of the text processing algorithm. JS scripts are sent to the first layer servers. The work is further distributed based on addresses of pages to be searched: second layer servers manage their own range of overlapping URLs, those ranges are then partitioned amongst servers from the first layer, which further partitions them amongst the workers. Work distribution uses the data about workers' performance gathered by the first layer servers. Second layer servers then collect the results: pages meeting search criteria plus newly detected URLs and return them to job owner.

Comcute [12] and a followup work by another team, *Comcute.js* [22] are another proof-of-concept implementations. The architecture consists from comcute cores, responsible for starting and stopping the jobs, partitioning the data and aggregating the results, the dispatchers (called also proxies in some papers) controlled by one of the cores, responsible for assigning the jobs to the workers (browsers) and separating the core servers from direct interaction with the workers [10]. Based on a job owner description, cores may automatically divide input data and create an appropriate number of tasks [13][48]. A separate layer provides an entry point for job owners wishing to submit, delete or monitor their jobs. Somewhat surprisingly, in experimental evaluation of *Comcute.js* [22] the centralized architecture with one dispatcher worked the best.

The jobs submitted to comcute consist of data and JS code, and are divided into tasks (having the same code, but different input data). Dispatchers download JSON description of tasks from the core, dispatch the tasks to workers, and then send the results back to the core. Core aggregates the results and monitors the computation, though it's not clear to us whether the monitoring signifies anything more besides tracking the progress of the computation.

Capataz [54][55] is a library created to support voluntary computing. The applications must import capataz library and run within Node.js. Application can post a job to the server, which returns them a *Future* object. When workers return the results, the *Future* object is resolved allowing application to continue.

They suggested job bundling, where server sends the same code with many different sets of input data, speculating that browser's just-in-time compiler reuses the JavaScript code in case of job bundling, while it is not reusing the code when the same script is fetched several times in a row.

They have noted the limitations of the WebWorker's thread standard. For example, only the rendering thread have access to DOM structure of the web document.

In the most common architecture employing Web Workers and Node.js it is cumbersome to provide good data flow between workers and servers. We found an open source project named Pasture.js [38] which simplifies this task. It provides a safe and comfortable kind of remote procedure call. In pasture.js, the server side code is called "Shepherd", the browser's main JavaScript code is called "Sheepdog" and a Web Worker threads are called "Sheep". Those 3 roles are then divided into modules. Programmer can easily use *talk().emit* method to call a certain method in any module from within any module, eg. executing *talk().emit('sheep.tea.waitor','do','addtobill',["name"],true)* will invoke a *waitor* function in a Web Workers module called "tea". It enables an easy communication between the server and the browser's worker threads, allowing to inspect and change each other's state.

Modern personal computers are equipped with fast graphic cards, and WebCL proposed standard [41] will allow web applications to use them. We know about two works making use of the GPU acceleration: WeevilScout [19] and CrowdCL [53].

WeevilScout [19] uses a WebCL code wrapped in JavaScript. New jobs submitted to WeevilScout can be written in JavaScript. Server transforms the code to a form which can be used in browser-based computation. During submission, job owners provide parameters (for example, many different matrices for matrix product code) and the server automatically generates jobs for all parameter combinations. Jobs are then put into the folders which are available as a web resource. Worker nodes periodically poll the server for new jobs, and they also ask for new jobs after finishing the old one. Authors claim that because scripts are cached on the client-side, running multiple jobs differing only by parameters will result in script being downloaded only once and then shared between many jobs.

Except WeevilScout, the only other solution we know to use GPU is *CrowdCL* [53]. Their contribution includes creating JavaScript framework, including KernelContext object abstraction to simplify creating OpenCL applications, starting applications on GPU and communication with the device. In order to use CrowdCL, programmers have to instantiate JS class and define at least *run* method. This class is passed to CrowdClient class constructor, which then runs the class in the background and sends back the results to a server, using HTTP POST method. The interaction with the GPU is enabled by allowing the programmer to define handlers for a number of events. Developers can also specify a method to be run after getting each request. Another distinguishing feature is the ability to batch several results to reduce the communication between clients and servers.

The economy of BBVC was discussed in [80]. For example, some cloud providers charge for downloading data, but not for uploading to the cloud. Because of that, the authors suggested that workers should not download the data from job dispatcher/scheduler, but rather from a separate content delivery network. Similarly, the results should be uploaded directly to the data server in the cloud instead of being returned to job dispatcher/scheduler. An interesting insight is that the value of the computation cycles “stolen” from the users may be ultimately lower than the costs of additional bandwidth and compute power demanded from the web servers. The authors proposed a model showing when BBVC would be cost-effective.

The problem of JavaScript performance gap with regard to native code was addressed by two works by Kajitani et al [39][40] (the original proposal is a technical report written

in Japanese). They used a rarely used extensions, NaCL (Native Client) and PNaCL (Portable Native Client), The former allows native codes to be run by browsers and the latter first requires compilation into LLVM bytecodes, which then can be run at near-native speed. Unfortunately, the future of both extensions is doubtful, as Mozilla has no plans to support them and Google, the original proponent, seems to stop further development[4]. The authors have compiled Gnu Mathematic Library into format compatible with PNaCL, potentially enabling the use of existing scientific code base without the rewriting it into JavaScript. Unfortunately, they've shown that the performance is still up to 9 times slower than the original native codes.

BrowserCloud.js [23] is a browser-based computation grid. It uses emscripten to generate very fast JavaScript code, indexedDB for storing data at the browser's side, and WebRTC [27] for direct browser-to-browser communication.

The most distinguishing feature of browserCloud is its decentralized nature. The main servers are used only as a randez-vous points between the browsers. After new clients learns about other clients, it uses WebRTC for direct communication with those clients. New clients register at signaling server, receive unique identifiers. Other peers are then notified about then existence of new peer.

Signaling server holds the network state in its memory, which allows it to generate new unique identifiers. Messages between the peers are routed via Chord routing. Distributed Hash Table (DHT) holds which peer is responsible for routing message destined to a particular peer. In addition, peers have so called “fingers” i.e. direct connections to randomly chosen other peers.

We conclude our survey by briefly mentioning an interesting BBVC project, called crowdprocess.com; it was referenced by [54]. It was supposedly a company which embedded JS code in webmasters websites; unfortunately, the project seems to be inactive, and its main website redirects to james.finance, the site definitely not related to BBVC.

4. Issues in browser-based voluntary computing

Browser-based voluntary computing is still relatively unpopular. In this section we identify commonly recurring problems in browser-based voluntary computing which may contribute to its lack of wider acceptance, and how those problems were or could be solved. Since many problems are common to all voluntary computing projects, when discussing the possible solutions we will also sometimes mention non-browser based approaches. In addition, we discuss the fault tolerance issues in Section 4.6 and in Section 4.7 we present the scheduling policies employed in BBVC.

4.1. Limitations imposed by browser's security model

The browser provides a sandboxed execution environment for the tasks distributed by the BBVC platform, greatly simplifying security and safety management. Faulty code usually (at least, in absence of errors in browser's code) should cause just unresponsiveness of a

4 http://developer.chrome.com/native-client/nacl-and-pnacl

given web page. Moreover, browser may terminate long-running scripts. Nevertheless, the advantages of this model come also with some drawbacks.

Same-origin policy is the most important restriction imposed by browser on a script code. A script originating at some domain may only access web pages having the same origin, i.e. having the same port, protocol and URL address. By default, requests to other domains are forbidden by browser for security reasons. However, this restriction could be circumvented by enabling CORS [76] on the server that owns the resources that the browser requests. The configuration on the server's side involves setting the header property Access-Control-Allow-Origin to the name of the domain from which the requests are allowed. QMachine used Allow-origin headers to get past CORS. MRJS on the other hand, used proxy server due to their initial lack of familiarity with CORS [67]. In addition, scripts executing in different tabs of the same browser, may communicate via postMessage/onMessage mechanism, even if they reside on different domains. Another technique to allow cooperating scripts to communicate, as long as they run in the windows of the same browser, is setting domain property of the loaded document to the same value.

Sandbox execution prevents accessing user's personal data from within JavaScript. Although it can store data in different structures of browser's memory (indexedDB, cookies, LocalStorage) – the amount of available storage is limited (for example, about few megabytes in the LocalStorage). This limits the size of data processed by workers.

There could be some restrictions of using JavaScript built-in functions from within Web Workers, window.crypto.getRandomValues() for instance (cryptographically random values generator) due to not having a 'window' object in Worker's scope.

In first generation of BBVC, it was suggested that restrictions of browser's security model could be circumvented by downloading standalone Java applications or using signed applets [6][68]. Alternatively, users could be asked to switch off applets network communication restrictions (for example, in order to establish a network connection to a third-party network site) [29].

The browser restrictions do have some effect on the job code structure, but usually can be circumvented. Same-origin policy prevents tasks from calling methods of tasks executed by a different browser, but tasks can communicate indirectly via central server (Section 4.2). Limitations on size mean code's data structures must have a reasonable size. Finally, browser prevents JavaScript to run Web Workers with a code that is not sourced from a separate JavaScript file with a domain identical to parent's domain. To evade this restriction one can create an artificial file visible only by a browser, by creating BLOB object, filling it with JS code and passing it as a parameter to a JavaScript method createObjectURL.

The use of plugins may help alleviate the mentioned restrictions. Unfortunately, it requires the active participation of the users (installation of the plugin), which, in turn, may discourage some of the potential participants.

4.2. Task-to-task communication

Communication between tasks usually is possible only when they reside on the same worker node, via JavaScript's methods postMessage and onMessage. Communication between different worker nodes is hard because of browser's security model. One solution is to route messages via the central server [7][16]. In [6], applets use RMI for direct applet-to-applet

communication. Applet first passes its reference to a broker (called “initiator”) which passes the RMI reference to all other applets participating in the same session. As described earlier, browserCloud.js [23] uses WebRTC [27] to achieve the same goal. Central server is used only to register browser handles, and message routing is done by the browsers themselves. There exist already libraries such as webGC [11] (based on WebRTC) which could be used to implement direct task-to-task communication. Unfortunately, WebRTC is still not fully supported by all browsers[5].

Lack of widely accepted and universally available mechanism enabling browsers to communicate constrains the design choices for algorithms developed for BBVC, and is one of the factors, even if minor, limiting the wider acceptance of BBVC. Support for direct task-to-task communication is limited in the papers we reviewed.

4.3. Recruiting and keeping the volunteers

Voluntary computing, including browser-based voluntary computing, ultimately cannot exist without recruiting users willing to contribute their machines' resources for a public cause. One have to both attract the users to a project and to keep them interested. Since those problems are shared by both BBVC and voluntary computing in general, the offered solutions should be shared too. In this section we do not concentrate on BBVC only and we discuss the papers analyzing the problem from the perspective of voluntary computing in general.

The great advantage of browser-based voluntary computing is its potential to recruit users from a broader audience than is typical for traditional projects. For example, BOINC seem to attract only particular type of users – 96.5% of BOINC users were identified as male, 83% of them were labeled as advanced users. Almost no children, women or elders participated in BOINC projects, which stands in stark contrast to a typical internet user profile [18].

The most basic way to get computing power is to pay users or to force them to participate [68], effectively meaning that the voluntary computing becomes voluntary in name only. New volunteers could be recruited by sending email invitations and blog posts [45][58], social media [19][83] or by being reported upon in popular new sites [79]. To attract new users, a flash widget used by OGC [61] allowed to inspect the news fetched from Yahoo Top Stories feed, and in theory could be used to allow social interaction or playing games. Users could run parasitic games, paying with their CPU time by allowing some computation job running in background [2]. This concept was tested in [50] where users played simple snake game, while in the background workers were doing useful work. The experiment was quite successful: an average game lasted about 2 minutes and impact of background computation on users' experience was not perceptible. None of the users rejected the idea of contributing their CPU cycles to the computation.

Some problems may be more interesting to users than others. Fox et al [30] identified environmental protection, weather forecasting and economic growth projection as kind of problems which potentially may attract many volunteers. Whatever the problem is, its

5 http://iswebrtcreadyyet.com/ Accessed on 9.01.2017

results should be freely accessible as both an ethical way to thank the public and as a way to popularize the project and attract more volunteers [59].

After users are recruited, it's important to keep them motivated to contribute. The average user contributes less than 5% of their compute power, and median and mean participation time of users is between 2 to 3 months [26]. It was suggested that lottery with monetary awards could be used to motivate users [68]. Awards could be printed diplomas diplomas for top contributors as in Einstein@Home [36]. Users could get credits which could be then used to pay for accessing web resources [29][64]. However, even purely virtual credit system providing nothing more except the reputation can be very successful to attract and keep the users [3][36]. To increase the reward for users, credits should be assigned as soon as possible, with publicly visible rankings [3]. As social interaction is important for many users, they should be able to create teams, to create their own profiles and to browse profiles belonging to other users [3]. Message boards with reputation system based on credit and seniority (time spent on boards) may be helpful in motivating users [3]. Unfortunately, in BBVC users will usually contribute anonymously, even though some proposals require users to register before contributing (e.g. Unicorn [66]).

Users may try to cheat, trying to claim credits for fake results [68]. In [2] it was suggested that program specific checksum could be computed by tasks to indicate they were actually run. To compute just and correct credit amount assigned to users after computation, small microbenchmarks might be first sent to a user to estimate his/hers machine computing capabilities [83]. For example, the BOINC client run benchmarks to estimate user' CPU power. Credit assigned to user is based on results of elapsed CPU time of correct results [3], and credits may be awarded to users based on their provided CPU power, storage and network.

Finding out why users contribute their time is an interesting research topic. Volunteers in BOINC were asked to state the reasons for their participation [65]. From 1500 users invited, 274 sent back useable responses. Users who were interested in seeing publishable results contributed more than others, but that motivation was not very important to most users. Personal values explained why users joined the projects, but not explained their contribution level. Reputation was ranked as very important. It was suggested that projects wanting to increase the users' contribution should aim to increase social interactions and should maximize the exposure to user achievement with minimal effort from the user – for example, instead of going to project website to find out who has contributed the most to the project, users could be informed by small widgets in the running application . Many users' contribute most at the beginning, and their participation level goes down in time. Team affiliation helps to keep contribution level high. While those results were based on self-selected and non-representative user sample, they are nevertheless valuable, as many projects might want to create base “core” user groups (to proselytize, provide skeleton computing power source and maybe even actively participate in project) which we speculate would consist in users with profile similar to those participating in BOINC projects.

The potential of BBVC to attract many volunteers has not materialized in practice yet. That means that despite no technical knowledge is required from users in order to participate in BBVC, the *accessability* challenge is not fully met. What's more baffling, the modern BBVC platforms we know do not employ the well known strategies developed for non-browser based voluntary computing platforms to keep users contributing (described

above). Most of the proposals we reviewed either no not report the number of attracted volunteers, or admits that the number is in the range of less than few hundreds. Even QMachine's results seem less impressive when remembering that those were collected over a period of almost a year, for many different projects. Therefore, the problem of recruiting the users is the most important challenge faced by browser-based platforms. We think choosing the right audience and combining them with methods mentioned in this section may be a key to a success of future BBVC platforms.

4.4. Trust and security issues

The issues of trust in voluntary computing are twofold. Firstly, BBVC solution must prevent job owners from abusing the trust of the volunteering users, otherwise it could become a platform for distribution of malicious software [51]. A malicious job owner could try to submit trojan horses, viruses, install backdoors, or try to use volunteers' computers to create denial of service attacks on internet services [26]. Volunteers must trust the applications that it does what it claims, that it won't attack their privacy and harm their system [26]. Secondly, users must be prevented from sabotaging the computation. Moreover, if the solution rewards the users for contributing their power, users could falsely claim the credit for the job they had not completed.

We stress that same problems would appear even without malicious intentions from volunteers and job owners. Software bugs may inadvertently harm users' machines, and faulty hardware or software errors on users' machines may be indistinguishable from intentionally submitted wrong results.

To deal with the first issue, the job owners' software can be checked manually and/or tested before being accepted for distribution on a BBVC platform. For example, XtremWeb allows submissions only from designated repository of trusted jobs. The jobs are accepted only from trusted institutions, tested first on set of dedicated users, and finally uses pair of public/private keys is used to ensure code integrity [28]. Unfortunately, sometimes data in distributed computation could be sensitive, and would have to be encrypted while residing on hosts' side [2]. Similarly, application code could be obfuscated [68][8]. We think, however, that if researchers want to use free resources provided by the volunteers, they should reciprocate with sharing their code and results, especially since it may building in build users' trust [59].

Several solutions to the problem of fake results submitted by users were proposed in the literature. Redundant computation is most often used. The canonically correct results may be then chosen by a simple majority voting [48][69][80]. The discrepancies between the results sometimes may sometimes be expected, and may depend on used compiler, architecture, or operating system. Because of that, scheduler in BOINC used homogenous redundancy, sending tasks of a given job only to machines with the same configuration [3]. Voting may be improved by credibility-based weighting, where trusted users' results are given more consideration than novice users [26][49]. Application-specific function may compare the results and choose the canonical, correct version [3][53]. To make sabotageur's work harder, the scheduler may limit number of tasks sent to a single host [3].

In [68] a “spot-checking” technique was proposed, where server would occasionally sent a job to client knowing a correct result in advance. Computation of some software

checksum would be ideal for both spot-checking and to ensure users actually run the software they received [2][28]. If the result from the client would differ from the expected, the client would be blacklisted, and the previous results (if any) received from that client would be discarded (backtracked).

Dividing the users into several classes (e.g. anonymous users, registered users, veteran participants) and differentiating the level of trust accordingly, may help with some of the issues mentioned here. Disabling anonymous job submission or requiring basic code review will prevent BBVC platform from becoming malware distribution center.

Finally, we want to mention the inherent ethical issues in running JavaScript code without notifying user [8]. In theory, Youtube could easily abuse the trust of its users by offloading data processing to their browsers, creating a supercomputer easily reaching the first place of the top500 list [80]. An earlier described solution with “publisher” in POPCORN de facto could be an example of such parasitic computing [64]. Nothing would prevent a platform from running scripts without users' knowledge. This problem is, however, not limited to BBVC. We would also note that asking for users confirmation would either irritate most of them, if one would use pop-up windows, or limit the users' participation if users would have to actually click on some checkbox or button to activate the scripts. We think that simple banner informing the users about running scripts, with links to more information and possibility to turn off the scripts (remembering user" decision for the future) would provide fair balance between need to recruit the users and yet enabling them to have a final decision to say whether they want to participate in a BBVC project.

4.5. **JavaScript limitations.**

The use of JavaScript may seem to address the *programmability* challenge mentioned in Section 2. Unfortunately, the language still has its limitations.

A severe limitation of JavaScript harming its use in scientific computation is the fact that, due to its floating point boundaries, it does not handle large numbers well (Table 2). The JavaScript floating number object is allowed to have precision of maximum 16 digits. It doesn’t matter if it’s zero before the comma and 16 digits after or 8 digits before comma and 8 digits after – the number of the digits must be equal to 16. However, there are libraries that extend the number precision, with a disadvantage of calculation slowdown.

The JavaScript is not designed as a general-purpose language and there are few ready scientific libraries available. Adding ready-to-use components implementing commonly appearing problems (e.g. matrix multiplication) is necessary to address *programmability* challenge.

Table 2. Accuracy of multiplication in JavaScript [84]

Multiplication	Result
10000000000000000.7777777777777777*1	10000000000000000.0
1000000000000000.7777777777777777*1	1000000000000000.8
100000000000000.7777777777777777*1	100000000000000.78
...	...
10.7777777777777777*1	10.777777777777779
1.7777777777777777*1	1.7777777777777777
1.7777777777777777*1	1.7777777777777777
1.77777777777777*1	1.77777777777777

Problems with JS performance may limit the scalability of the applications dedicated to the BBVC. JavaScript codes may be substantially sped up using asm.js [35][6]. Since asm.js is a subset of JavaScript, code using it should run on most if not all browsers, though older versions may encounter minor problems. An emscripten compiler [81] is able to convert LLVM bytecodes into very fast JavaScript using asm.js, rivaling even a native code in speed [23][80][41]. Therefore, any language which can be compiled into LLVM bytecodes can be converted into fast JavaScript code using this tool. While several works mention possibility of using the emscripten, we are aware of only one actually using it [23]. Scientific code may in future benefit enormously from technologies such as simd.js [37] allowing JavaScript to use accelerated vector operations. Finally, the recently proposed WebAssembly[7] standard may solve the performance problem.

All of the mentioned limitations, lack of ready-to-use components, JS limitations and performance issues limit the acceptance of the BBVC.

4.6. Fault tolerance

The are several kinds of faults which can affect distributed computation in BBVC (a *fault tolerance* challenge from Section 3). Worker nodes may fail, users may close the system, the browser, or just the tab running the scripts. All those events mean the task sent to the worker node is lost, and they all can be eventually detected by a job scheduler, for example by using timeouts or by subscribing to *disconnect* event with WebSocket technology. The lost tasks can be then reallocated by the job scheduler to another worker node [9][57][83]. Assigning tasks in round-robin fashion may result in natural redundancy and achieving natural resistance to worker node failures [50][55][63][67][70]. Task re-allocation has positive side-effect of ensuring the computation is not blocked by slowest worker nodes [7][68]. Dealing with faults might be left to the developer, who may choose to ignore lost results [64].

6 http://www.2ality.com/2013/02/asm-js.html
7 http://webassembly.org/

A mirror problem is when worker node loses connection to the job scheduler. One solution is to retry on lost connections (when POSTing the results), increasing the delay after each timeout [54][55]. Lost connection may signify the server crash. One solution is to continue the execution offline for a certain amount of time – e.g. 10 minutes, and cache the results in local storage. If the connection returns, the results kept in the cache could be send to server and be cleared afterward on the client side [19]. The second solution is to request work from to another server until connection will be reestablished with the old one.

4.7. Job and task scheduling

Typically, in BBVC worker requests the work from the job scheduler/dispatcher. Job schedules must ensure that each job is eventually completed, that workers are not overloaded and, ideally, that no worker is ever idle. Known scheduling algorithms are not always applicable because of high downtime and unpredictability of node behavior [26] (the *adaptability* challenge mentioned in Section 3). The naive policies do not take the participant history into the account. Those are FCFS, locality assignment (tasks are sent to worker who already has the needed data) and random assignment (grouping workers based on their availability and power). Knowledge based policies may take into account past behavior, for example by allocating less data to slow clients [57]. The most popular policy is to submit tasks eagerly to idle workers in a round-robin fashion [68]. Usually, workers request for another task as soon as they complete the previous one [19][67]. More complicated scheduler was considered for Comcute [5],.

In Javelin++ [62] two scheduling policies were proposed, both based on work-stealing (where when host is idle, it “steals” task from other hosts). Both policies require that tasks can be easily divided into smaller chunks. In tree-based scheduling policy, hosts are managed in hierarchical structure, and they steal tasks from a deterministic set consisting of host's parent and children. In probabilistic policy, hosts steal tasks randomly from other tasks. A tree manager prevents the tree to become broken in case of hosts' failures, restructuring tree when necessary. Both policies seemed to affect the performance in a similar manner.

Based on our review, there is still a place for developing new scheduling policies, which would take into account the hosts capabilities and, especially, which would try to reduce the uncertainty in host behavior (for example, by trying to predict the behavior or by imposing some constraints).

5. Poznan Open Voluntary Computing Platform (PovoCop)

In the previous section we have identified several areas for improvement. First and the most important challenge is attracting the users. To overcome this challenge we suggest targeting the right audience and employing strategies developed for other voluntary computing platforms, such as keeping reputation scores. Developing new scheduling policies taking into account hosts past histories and their characteristic is another interesting field of research. Another important issue is addressing the *programmability* challenge by easing

the development for new applications. Finally, increasing the efficiency of the platform should be another priority.

In this section we present our plans for a platform which, in a completed form, hopefully will solve all of the problems and challenges mentioned above, allowing users to volunteer their computing power to solve the scientific problems. PovoCop will consist of set of servers maintained by our team and set of recruiting sites. Any web site could become recruiting site by injecting small code into the pages they serve. Researchers would upload their codes into our platform. Whenever users (volunteers) will visit one of the recruiting sites, task will be automatically fetched from our infrastructure. Computed results will be then aggregated, verified and presented to the researchers submitting the jobs (and, possibly, any other interested party). Currently PovoCop is in an early stage of work, though we have a working prototype.

5.1. The motivation

Why BBVC is still relatively unpopular? In Section 4.5 we have pointed to the performance issues and lack of ready-to-use JS components as one factor. We conjecture, however, that there is another reason the scientific community seemed to be hesitant to embrace the possibilities offered by BBVC. Simply, the community has access to clouds, clusters or established non-browser based voluntary computing platform such as BOINC. Therefore, we motivate our work by choosing to target an another audience. Nowadays it is not uncommon for amateurs attempting to solve problems which not so long ago were thought to require professional teams working on supercomputers or grids. Such amateurs usually have modest technical knowledge, sufficient to maintain and configure a blog. Sometimes they are PhD students or scientists, who would want to quickly check some hunch or experiment with data, while lacking access for larger computing infrastructure. While our completed solution should be universal enough to be used by everyone, at least initially we will target this class of users. That will allow the platform to mature and then, we hope, it will attract the more "serious" users.

We envisage that the potential users would insert simple JavaScript code to their blog software (possibly just a single line <*script src*=...>). Whenever their blog would be visited, a script will run, showing some basic information on the sidebar and connecting to infrastructure we intend to establish. As blog authors usually form cliques, some of them could be recruited to mobilize the computing power and to proselytize the solution.

Software will be also provided for so called "power users" who would be able to run their own infrastructure based on Povocop. Therefore, after PovoCop would gain a popularity amongst amateurs, professional users would have easy way to adopt the platform for their own uses. Eventually we would want all those infrastructures to be able to share the harvested computing power.

5.2. Architecture

The architecture we are developing consists of five main entities (0). We will now describe all the entities and their roles in our architecture, both in completed future version and in the proof-of-concept prototype implementation.

Job owner (e) is a person creating the JavaScript code to be executed by voluntary users. We plan to offer job owners the capabilities to describe their jobs, e.g. marking whether they want the results of their jobs to be publicly available, or whether the results are repeatable (for example, results depending on *Math.rand()* function will not be repeatable).

Every task will be a part of a job that could be too much time-consuming for one computer to complete. Every task is going to be handled by one Web Worker in user's browser. This means, that some users may execute four task at once, and some may execute eight of them, depending on number of cores in their computer's CPU. Currently, division into tasks is the job's owner responsibility. Tasks should be short functions accepting parametrized input data. The requirement of functions being short is necessary, as a browser may interrupt long-running scripts. In order to ease the development, our platform will offer the possibility to divide the jobs automatically into tasks, based on the job's input data [20].

Recruiter (b) is HTTP server with its own content (hopefully valued by users) and injected JS script provided by PovoCop. Users connecting to Recruiter will become *drafted*: their browser will load and execute the tasks, becoming a *Plantation* (a). In our initial implementation users cannot choose the project in which they want to participate. In the final version users will be provided with short information, where they will be able to choose the project (with one being pre-chosen by default) or refuse the participation. Users will have the possibility to register themselves, therefore notifying the platform they are willing to donate more of their resources to the platform. We will also provide a special browser plugin for users wanting to participate even more. Therefore, users could start by participating anonymously in PovoCop, while the platform would gently encourage them towards more active role.

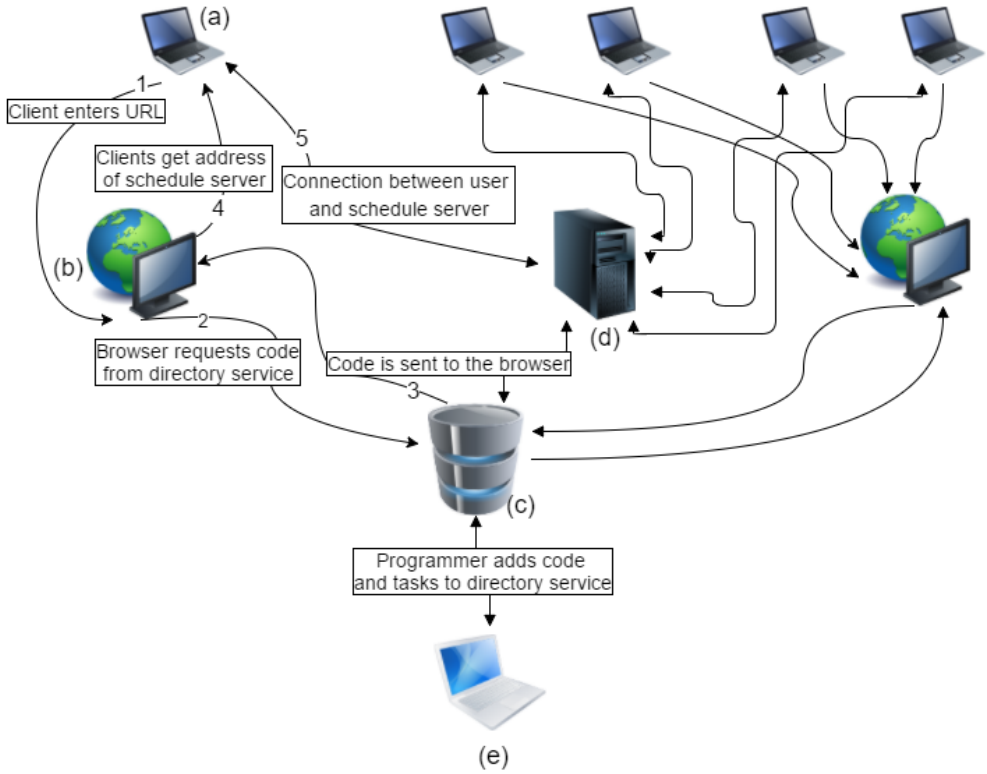
Plantation (a) is a browser with Web Workers. Worker is a single thread, the code executor. Its speed will depend on browser's type (e.g. Mozilla Firefox was the fastest code executor during our tests), the number of CPU cores, operating system and system load. Plantation connects the *Directory* service to get initial code and then cooperates with *Scheduler* to get data, post results and receive more tasks. Plantation will maintain local pool of tasks currently executed or scheduled locally for execution. For anonymous and registered users, the Plantation will stop to execute as soon as the user will leave the Recruiter's website. However, with the browser plugin mentioned above, the Plantation will still run even after leaving the website.

Directory service (c) is a server with a database of jobs to execute (code, number of iterations, network address of scheduler). Worker will get the code and address from directory server and job owner will upload jobs or add tasks to current jobs in directory service – there will be a web interface (currently in tests) serving both GET requests for the code retrieval (for workers) and POST requests with code submission (for job owners).

In initial implementation we have only one Directory Service. This, obviously, creates the usual problems with single-point-of-failure. Moreover, single server may, in theory, be overburdened with work. Other authors reported that this rarely became an issue with the number of users we hope we will recruit in the initial phase of development [22]. In future

we can either try a master-slave replication with fail-over in case of primary replica failure, or we could exploit the natural load balancing offered by DNS, by offering several servers operating under the same DNS name.

Scheduler (d) is a server that gives tasks (input data for the code) to workers, and



receives the computing results from them. Along with workers it is the main actor in our system. In addition, it monitors if the workers are still computing tasks. In future it will verify the correctness of the results sent by users. It stores the results in directory service. One scheduler can maintain many jobs; ideally, jobs maintained by one scheduler should share the code and differ only in parameters (i.e. input data). In addition, jobs sharing the code should not be maintained by different schedulers. In the current implementation we have only one Scheduler, but we are currently working on adding more. The required functionality should be finished in the next version of the prototype. We want to develop new scheduling policies, taking into account the fact that some Plantations will be more reliable and predictable than other.

Figure 1. Architecture of PovoCop

Users visit Recruiter's site, executing scripts which load the code from the Directory. Directory also provides user with Scheduler's address, assigning Plantations to Schedulers in round-robin fashion. Plantation then gets input data from Scheduler (being assigned several tasks), executes the task and tries to post back the results, in return receiving another task. If Plantation loses the connection to Scheduler, it tries to reconnect. In future, it will then contact the Directory which will provide it with address, of another Scheduler. Since the tasks are mapped to particular schedulers, when connection to Scheduler A is lost and Plantation switches to Scheduler B, all tasks from Scheduler A become *orphaned*. In the next version of the prototype, the orphaned tasks will be terminated. We could continue to execute them, ideally with lower priority – unfortunately JavaScript do not offer functionality to control workers' priorities.

In future, Schedulers will periodically inform the Directory about the number of tasks awaiting assignment. Directory will then assign workers based on that information. In addition, in future we plan to add the possibility of downloading the input data for the job from external data server, instead of from Scheduler.

5.3. The prototype

We have implemented a proof-of-concept prototype of our architecture. Here we will shortly discuss the implementation details of the working prototype. Currently we are working on a new version, which should be ready during 2017. We have also another team exploring possibilities offered by a different architecture design and investigating alternatives to the tools we have used, with a first prototype scheduled to ship in February 2017.

Except Recruiter (which could be based on any HTTP server), all other servers will be based on Node.js – which will allow us to write whole code in one language (JavaScript), hopefully easing the development and maintenance. We are considering the possibility of implementing some required functionality (e.g. a parser for a job specification language) as C-based modules for Node.js, for performance reasons.

JavaScript was the obvious choice to implement the platform. This script language is now (2016) used by 93.6% of websites[8]. Its latest version is ECMAScript 2016, released at June 17, 2016. Moreover, JavaScript is the only language available at every browser. JavaScript runs on specific engines such as V8 implemented in Google Chrome browser or SpiderMonkey implemented in Mozilla Firefox. These engines don't allow JavaScript to make any malicious functions, protecting user's data on the computer and disallowing scripts to infect user's computer with viruses.

To make HTML page a part of PovoCop, the website owner must simply inject *<script src="..."/>* tag into the page (manually in our prototype). The script's address will point into .js file on a Directory server. Directory provides the browser with a script and an IP address of Scheduler server. Browser connects to the Scheduler with Web Socket technology to get input data. As discussed in Section 4.1, server reconfiguration is required to allow browser to send requests to different domains. After configuring "Access-Control-Allow-Origin"

8 W3Techs: Usage of client-side programming languages for websites, 4.07.2016, http://w3techs.com/technologies/overview/client_side_language/all

header on both Scheduler and Directory, we can enable browser to access the resources on either server.

The initial script uses a quick benchmark to estimate the number of CPU cores of client's browser and the single core's speed. The Scheduler can use this knowledge to send more tasks to multicore computers or prioritize clients with faster CPU's when deciding on tasks distribution. Benchmark computation will be used in the future as part of spot-checking technique, to ensure user actually runs the tasks and returned correct results.

The browser receives code from Directory Service (server written in Node.js) and tasks from Scheduler (also written in Node.js) and creates a Web Worker with the received code and arguments for each task. When the task is done, the Worker emits result to the main browser scope via postMessage/onMessage JavaScript mechanism. Result is then re-emitted to Scheduler via WebSockets. Scheduler saves the result into the database. Scheduler periodically sends PUT requests to Directory Service with the number of currently connected clients and number of tasks left to do 100% of the Job – this data will be used in the next version of the prototype.

The tasks to execute and the results of computing should be automatically sent to/from server without page reload. For the required communication between workers and servers we decided to use WebSockets. Compared to basic HTTP requests (with AJAX), WebSockets do not need headers for each message. Another benefit is the server's ability to send data to any client at any time. Before that technology, a browser had to send a request to server in order to get some data (using techniques such as reverse AJAX/Comet). Many structures can be sent over Web Sockets – String, JSON, Number, Boolean. Reliability of the data is ensured by underlying TCP protocol. WebSocket allows two way communication – server can send message to browser at any moment. Instead of standard WebSockets browser implementation, we use Socket.io extension. Our choice was dictated by additional features provided by Socket.io, such as long pooling or broadcasting.

In order to execute many threads we use WebWorkers. It allows us to run it in a separate runtime. It is safe against XSS attacks (injecting malicious JavaScript code to the WebSites without having permission, i.e. in blog comments) or CSRF (sending requests to other websites using the cookie for logged in users on these websites).

We need a database to store code, results, scope of tasks arguments, and job arguments. We choose noSQL database MongoDB because of it's scalability, ease of use and integration with Node.js with frameworks like Mongoose.

Every Job must have the code, the IP address of scheduler and the scope of tasks to do inserted into the Data Dictionary's database. Job owner must provide an input data for tasks, the interval of arguments to be sent to clients and the code to be executed by them. There has to be a Scheduler working on public IP address. Job owner has to provide an optional function that verifies results, and a non-optional function that divides input arguments to clients.

Jobs posted by job owners may contain unintended errors, e.g. the code may end up in never ending loop. The working version of the prototype has no built-in defenses against such behavior. In the next version, however, we will use Node.js' ability to run code as a child process in the background. Code will be first run as a small task with parameters on server side. If the code won't end after a certain time, child process will be killed and the code will not be inserted into the database.

We are in process of developing a simple browser plugin to augment our platform. Users can install this plugin, which will contact the Directory automatically and ask for tasks to execute whenever user will run a browser, without the requirement of visiting any web site first.

The prototype's code is freely available at https://github.com/Vatras/BrowserComputing.

5.4. Experimental evaluation

To check the capability of our prototype, we decided to test 2 simple algorithms. We ran the tests on 13 computers with Inter(R) Quad Q9650 3.00 GHz CPU, 8GB RAM based on Windows 7 operating system. The browser we used was Google Chrome.

The first algorithm was PI estimation using Monte Carlo technique. We chose this problem because of low communication and small size of passed messages and because the technique ensures computational task independence.

Figure 2 shows results of the experiment. The distributed code achieved relatively good approximation of PI, much earlier than the sequential version. Part of the success was better randomization because of distributed random numbers generation. Sequential version of the same algorithm, implemented in C, could not stabilize around comparable PI approximation even after several minutes, compared to few seconds in case of distributed version. The low number of message passed was another key to high performance of this algorithm.

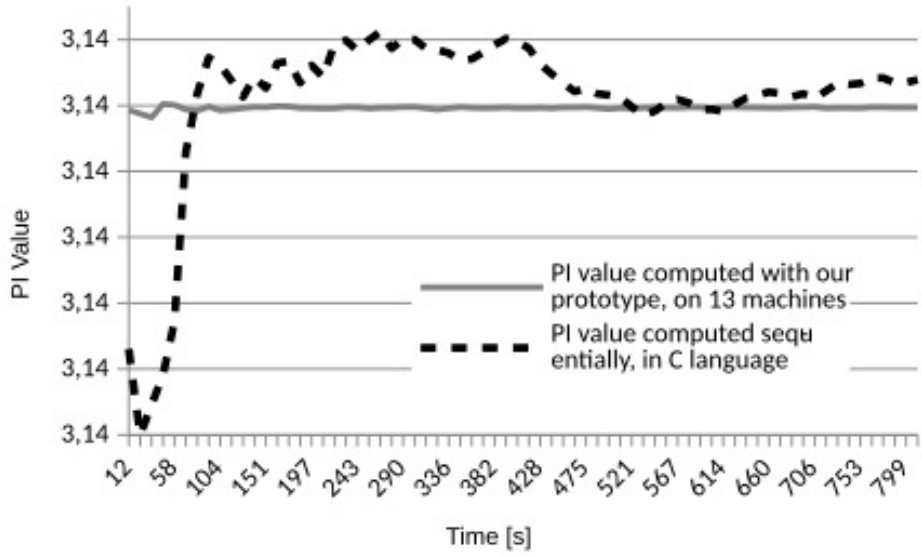


Figure 2. A graph comparing the value of PI, computed by sequential C code and by using our prototype with 13 computers [84]

As the second algorithm we evaluated the matrix multiplication. We chose this problem to check the abilities of JavaScript in simple math calculations. We also wanted to see the impact of heavy message passing on overall time. In experiments every node received two columns as arrays, multiplied them, sending the results afterward. In this case sequential version was faster than the distributed. The second experiment (Figure 3) revealed that it's not the computing speed, but rather the network bandwidth is crucial in running algorithms with heavy input or output messages – the time the browser spent downloading the matrices was higher than time actually spent on computation. We saw similar results in the third experiment (results not shown) with distributed file sort, when distributed algorithm performance was worse in order of magnitude, mostly because of high communication costs.

We conclude our prototype is better suited to the algorithms downloading input data only once, doing basic calculations and sending results back to the server using lightweight messages.

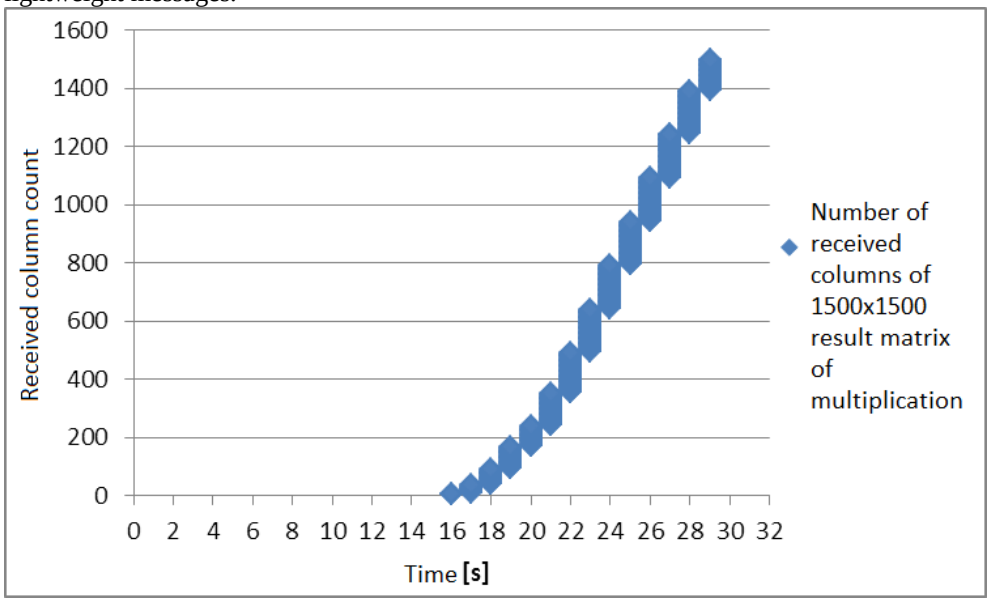


Figure 3. Graph showing moments of receiving succeeding columns of matrix multiplication of size 1500 x 1500 – for 13 computers using our prototype[84]

6. Related work

One of the first successful large-scale attempts at voluntary computing was GIMPS (Great Internet Mersenne Prime Search) [82], started in January 1996. Until January 2016, fifteen previously unknown prime numbers were found[9], and total average computing power of

9 "GIMPS Project Discovers Largest Known Prime Number: $2^{74,207,281}$-1", http://www.mersenne.org/primes/?press=M74207281

GIMPS project amounted to almost 300TFlops[10] (as of June 2016). Another early precursor was distributed.net [51], created in 1997 to answer the RSA Secret-key challenge[11]. The third, and probably the most known, was Seti@Home project, a search for extraterrestial radio signals of artificial origin, oficially started in 1999.

Based on experiences gathered during Seti@Home project, a team from Berkeley university created BOINC (Berkeley Open Infrastructure for Network Computing) [3]. BOINC is a common infrastructure for distributed voluntary computing. Currently there are at least 57 projects based on BOINC, with close to half million volunteers and more than nine hundred thousands computers, contributing on average 10.5 PetaFLOPS. The largest BOINC project is still Seti@Home, with 1.6 million users, while the smallest project, BMG@Home attracted just 170 volunteers[12]. Other popular projects are Predictor@Home (study of protein behavior), Folding@Home (protein folding), climateprediction.net, climate@home (study of climate changes) and many more. Almost half of compute power is dedicated to top five BOINC projects [75].

BOINC applications can be screensavers, window services or standalone application. To decrease the amount of effort in developing new BOINC applications and their maintenance, a virtual machine approach was implemented [36]. First, the application is compiled for a particular virtual machine. Later, a virtual machine image is created (virtualization-friendly linux plus the instantiation data). Finally, users must install a manager responsible for downloading, unwrapping and running virtual machine images. A lot of effort was put into assuring the last step would be as easy as possible.

Another project with similar goals is XtremWeb [28]. Anyone can set up a project using XtremWeb, starting his or hers own web page. The only requirement is that the "collaborators" must contribute some of their leftover computing power to original XtremWeb.

In XtremWeb, worker initiates the connection with the main server and receives list of servers providing the jobs, as well as how to contact to those servers (i.e what protocol they use and on which port they listen). Clients request work matching their specification. This specification contains description of client's operating system and architecture, and specifies what binaries already are in client's possession. Server answers by sending back the job and address to which client should upload the results. Client periodically pings the server to assure the server its still alive. Clients which do no respond for a long time are considered dead. Job assigned to dead clients is rescheduled. Clients are written in Java, using an interface allowing to call system calls (in C). XtremWeb contains dispatcher which is responsible to choosing the jobs (for example, based on their priority) which are then send to scheduler responsible for scheduling their execution.

The EDGeS project [75] aimed to join computing resources of institutional grids (in this case, from EGEE – Enabling Grids for E-sciencE) and desktop grids, i.e. XtremWeb and BOINC. The ultimate goal was to create special "bridges" enabling grids to act as clients for BOINC and XtremWeb-based projects, and also to allow using BOINC and XtremWeb infrastructure for executing jobs submitted to EGEE grids.

10 "PrimeNet Activity Summary 2016-06-24 15:00 UTC", http://www.mersenne.org/primenet/

11 "distributed.net History & Timeline", http://www.distributed.net/History

12 Retrieved at June the 23 2016 from http://boinc.berkeley.edu/

The existing surveys on voluntary computing in general [26][78], but the surveys we are familiar with dedicate a very limited space for browser-based solutions and describe only the most popular proposals.

A concept related to voluntary computing is crowd computing or "citizen science" (leveraging human abilities for scientific purposes) where humans participate in solving scientific problems. While they contribute their computing resources (by running applications provided by the project teams), the key difference is that they contribute also their skills and knowledge.

There are still problems where a human can relatively quickly find the solution, or good approximation of the solution, while there are no known algorithms or they are slow. One example is pattern recognition. Zooniverse [72] exploits unique human abilities in pattern recognition by encouraging volunteers to participate in projects involving classification of the galaxies, finding exoplanets or classifying wild bees captured by motion cameras in jungle near Serengeti. In 2014 900.000 volunteers participated in 20 projects, and 59 scholarly articles were published with Zooniverse findings.

A related concept are Games With A Purpose (GWAP) or Human Computing Games. In [17] it was shown that when NP-complete graph-related problems are presented as task of determining whether some puzzle is solveable (or not), human players can to find a solution within minutes (when algorithmic approach required hours or even days). In one case novice player devised a novel strategy which eluded an expert researcher (and his three students) studying this problem for three years. Using games has advantage of appealing to larger user base: about 71% of people playing puzzle games are women, and 37% are people in their 40s [18].

FoldIt was descendant of Rosetta@Home – the participants in that project were often frustrated that they were seeing an obvious solution, but were not able to refine the results found by algorithms [43]. In FoldIt players manipulate protein structures using interactive visual tools. Players can create micro-programs, called recipes (a name most likely chosen to sound as non-technical as possible), which can be edited and shared, optimized collectively by society of FoldIt players (though recipes can be private to their creator and/or his team). Some of the recipes are comparable to algorithms created by experts. Within 3 and half month, 721 players run 5488 unique recipes. 568 players wrote 5202 recipes. In January 2013, 2200 players were active.

The input of volunteers to solving scientific problems is much appreciated. "FoldIt players" are credited as coauthors or at least two scientific papers [42][43]. For example, within ten days players were able to find a workable 3D structure of Mason-Pfizer monkey virus M-PMV retroviral protease – a problem for which there was no solution for previous 15 years. As retroviral proteases have critical role in virus proliferation and maturation, this potentially can help design anti-retroviral drugs, including AIDS drugs.

Browser-based approaches may also be used to create distributed file system [49][46], "stealing" disk space from users.

7. Conclusions

We have surveyed three generations of browser-based voluntary computing, discussing their commonalities and differences. We have presented the issues shared by all BBVC (and

sometimes also non-browser-based voluntary computing) projects and the solutions proposed in the literature. Finally, we discussed architecture of BBVC platform we are working on, and we have shown the working prototype of the platform.

Our prototype deals well with the algorithms which do not require large sets of input data. Tasks sent to the browsers should be as short as possible, because server will receive the results of more computations in case of loosing connection with the browser. We have found out one unexpected result of distributed computing: using different computers provides much better randomization in results of invocations of Math.rand() function. When we get the results they are really random – which is seen, for instance, in PI generation using Monte Carlo method. Our architecture could utilize this finding to create the distributed pseudo-random generators.

In future, we intend to implement all the missing elements of our architecture. We want to create JavaScript implementation of 4mix algorithm (used to model population as a mixture of four reference populations) and test it on set of blogs. Removing single-point-of-failure elements and balancing the load is another important task for our future work. Designing new scheduling policies and solving issues related user recruitment, result verification, implementing cheating-resistant user credit system and fault tolerance are other important challenges.

References

[1] Abidi L., Cérin C., Fedak G., He H., Towards an Environment for doing Data Science that runs in Browsers. In: *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*, 662-667

[2] Alexandrov A.D., Ibel M., Schauser K.E, Scheiman C. J., SuperWeb: towards a global Web-based parallel computing infrastructure. In: *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, 1997, 100-106

[3] Anderson D.P., BOINC: a system for public-resource computing and storage. In: *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, 2004, 4-10.

[4] Anderson D.P., Volunteer computing: the ultimate cloud. *Crossroads* 16, 3 (March 2010), 7-10

[5] Balicki J., Bieliński T., Korłub W., Paluszak J., Volunteer Computing System Comcute with Smart Scheduler, *Applications of Information Systems in Engineering and Bioscienc*e, *Proceeding of SEPADS'14 – 13th International Conference on Software Engineering, Parallel and Distributed System*, 2014, 54-60

[6] Baratloo A., Karaul M., Holger K., Kedem Z.M, An Infrastructure for Network Computing with Java Applets. *Concurrency Practice and Experience*, **10,** 11-13, 1998, 1029-1041

[7] Baratloo A., Karaul M., Kedem Z. M., Wijckoff P., Charlotte: Metacomputing on the Web, *Future Generation Computer Systems*, **15**, 5-6, 1999, 559-570

[8] Berry K., Distributed and Grid Computing via the Browser, In *Proceedings of the 3rd Villanova University Undergraduate Computer Science Research Symposium (CSRS 2009)*, Villanova University, 2009, Retrieved: http://www.csc.villanova.edu/~tway/courses/csc3990/f2009/csrs2009/Kevin_Berry_Grid_Computing_CSRS_2009.pdf

[9] Boldrin F., Taddia C., Mazzini G., Distributed Computing Through Web Browser. *2007 IEEE 66th Vehicular Technology Conference*, Baltimore, MD, 2007, 2020-2024

[10] Brudło P., Foundations of Grid Processing for the Comcute System, in: Balicki J., Krawczyk H., Nawarecki E. (eds.), *Grid and Volunteer Computing*, Gdańsk University of Technology Press, Gdańsk, 2012, 33-39

[11] Carvajal-Gómez R., Frey D., Simonin M., Kermarrec A. M., WebGC Gossiping on Browsers Without a Server. In *Web Information Systems Engineering–WISE 2015, Springer International Publishing*, 332-336

[12] Czarnul P., Kuchta J., Matuszak M., Parallel Computations in the Volunteer–Based Comcute System, in: Goos G., Hartmanis J., van Leeuwen *Parallel processing and Applied Mathematics, Lecture Notes in Computer Science*, **8384**, Berlin, Heidelberg, Springer-Verlag, 2014, 261–271

[13] Czarnul P., On configurability of Distributed Volunteer-based Computing in the Comcute system, in: Balicki J., Krawczyk H., Nawarecki E. (eds.), *Grid and Volunteer Computing*, Gdańsk University of Technology Press, Gdańsk, 2012, 53-69

[14] Liu Ch., White R.W., Dumais S., Understanding web browsing behaviors through Weibull analysis of dwell time. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval* (SIGIR '10). ACM, New York, NY, USA, 2010, 379-386. DOI=http://dx.doi.org/10.1145/1835449.1835513

[15] Charalampidis I., Berzano D., Blomer J., Buncic P., Ganis G., Meusel R., Segal, B. CernVM WebAPI-Controlling Virtual Machines from the Web. In *Journal of Physics: Conference Series*, **664**, 2, 2015, 022010, IOP Publishing

[16] Christiansen B. O., Cappello P., Ionescu M. F., Neary, M. O., Schauser, K. E., Wu, D., Javelin: Internet-based parallel computing using Java. *Concurrency: Pract. Exper.*, **9**, 1997, 1139–1160

[17] Cusack C., Largent J., Alfuth R., Klask K.. Online games as social-computational systems for solving np-complete problems. In: *Meaningful play*, 2010

[18] Cusack C., Martens C., Mutreja P., Volunteer computing using casual games. In: *Proceedings of Future Play 2006 International Conference on the Future of Game Design and Technology*, October 2006, 1-8

[19] Cushing R., Putra G.H.H., Koulouzis S., Belloum A., Bubak M., de Laat C., Distributed Computing on an Ensemble of Browsers. *IEEE Internet Computing*, **17**, 5, Sept.-Oct., 2013, 54-61

[20] Danilecki A., Fabisiak T., Kaszubowski M., Job Description Language for a Browser-based Computing Plaftorm: A Preliminary Report, Accepted for the *9th Asian Conference on Intelligent Information and Database Systems*, April 2017, Kanazawa, Japan

[21] Dean J., Ghemawat S., MapReduce: simplified data processing on large clusters. *Communications ACM* , **51**, 1, January 2008, 107-113

[22] Dębski R., Krupa T., Majewski P., Comcutejs: A Web Browser Based Platform For Large-Scale Computations. *Computer Science (AGH)*, **14**, 1, 2013, 143-152
[23] Dias D., browserCloud. js-A federated community cloud served by a P2P overlay network on top of the web platform, Master Thesis, Tecnico Lisboa, 2015.
[24] Dłubacz W., Duda J., Distributed evolutionary computing system capable to use mobile devices. In: Matiaško K., Lieskovský A., Mokryš M. (eds.), *Proceedings in Conference of Informatics and Management Science*, **2**, 1, 2013, 393-396
[25] Duda J., Dłubacz W., Distributed evolutionary computing system based on Web browsers with JavaScript, In: Öster P., Manninen P. (eds.), *Applied parallel and scientific computing, Lecture Notes in Computer Science*, **7782**, Berlin, Heidelberg, Springer-Verlag, 2013, 183–191
[26] Durrani M. N., Shamsi J.A., Volunteer computing: requirements, challenges, and solutions, *Journal of Network and Computer Applications*, **39**, March 2014, 369-380
[27] Ericsson A.B.,Burnett D.C., Jennings C., Naranayan A., Aboba B., WebRTC 1.0: Real Time Communication Between Browsers, *World Wide Web Consortium (W3C) Working Draft*, Retrieved: https://www.w3.org/TR/webrtc/, August 2016
[28] Fedak G., Germain C., Neri V., Cappello F., XtremWeb:A Generic global computing systems. In: *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRD'2001)*, Australia, 2001, 582-287
[29] Finkel D., Wills C.E., Amorin K., Covati A., Lee M., An applet-based approach to large-scale distributed computing. In: *Proceedings of the International Network Conference*, Plymouth, United Kingdom, July 2000, 175-182
[30] Fox G. C., Furmanski W., PETAOPS and EXAOPS: Supercomputing on the Web. *Internet Computing, IEEE*, **1**, 2, 1997, 38-46
[31] Gallacher T., A browser based distributed computing platform utilising idle cpu cycles on visitors computers when browsing. Retrieved: https://github.com/tomgco/dis.io, 2012
[32] Gareth J.J., AJAX: A New Approach to Web Applications. http://adaptivepath.org/ideas/ajax-new-approach-web-applications/, February 2005
[33] Główny Urząd Statystyczny, Społeczeństwo informacyjne w Polsce, http://stat.gov.pl/download/gfx/portalinformacyjny/pl/defaultaktualnosci/5497/2/5/1/spoleczenstwo_informacyjne_w_polsce_2015_-_notatka.pdf
[34] Grigorik I., Collaborative Map-Reduce in the Browser. Retrieved: https://www.igvita.com/2009/03/03/collaborative-map-reduce-in-the-browser/, March the 3rd, 2009
[35] Herman, D., Wagner, L., Zakai, A.,asm. js: Working Draft 17 March 2013. Retrieved at: *asmjs.org/spec/latest.*
[36] Høimyr N., Marquina M., Asp T., Jones P., Gonzalez, A., Field, L., Towards a Production Volunteer Computing Infrastructure for HEP. In *Journal of Physics: Conference Series*, **664**, 2, 2015, 022023. IOP Publishing
[37] Jibaja, I., Jensen, P., Hu, N., Haghighat, M. R., McCutchan, J., Gohman, D., McKinley, K. S., Vector Parallelism in JavaScript: Language and compiler support for SIMD. In P*roceedings of 2015 International Conference on Parallel Architecture and Compilation (PACT)*, October 2015, 407-418
[38] Jonker M., Web browser based distributed computing FOSS project. Retrieved: http://www.meetup.com/limerick-open-source/messages/77136940/, May 24, 2015

[39] Kajitani S., Nogami Y., Fukushi M., Amano N., A performance evaluation of Web-based volunteer computing using applications with GMP, In: *2015 IEEE International Conference on Consumer Electronics - Taiwan (ICCE-TW),*, Taipei, 2015, 41-42

[40] Kajitani S., Nogami Y., Miyoshi S., Austin T., Volunteer Computing for Solving an Elliptic Curve Discrete Logarithm Problem, In: *2015 Third International Symposium on Computing and Networking (CANDAR)*, Sapporo, pp. 122-126

[41] Khan F., Foley-Bourgon V., Kathrotia S., Lavoie E., Using JavaScript and WebCL for numerical computations: a comparative study of native and web technologies. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS'14)*, ACM, New York, NY, USA, 2014, 91-102

[42] Khatib F., Cooper S., Tyka M. D., Xu K., Makedon I., Popović Z., FoldIt Players, Algorithm discovery by protein folding game players. *Proceedings of the National Academy of Sciences*, *108*(47), 2011, 18949-18953

[43] Khatib F., DiMaio F., Foldit Contenders Group, Foldit Void Crushers Group, Cooper S., Kazmierczyk M., Baker, D., Crystal structure of a monomeric retroviral protease solved by protein folding game players. *Nature Structural & Molecular Biology*, **18**, 10, 2011, 1175–1177

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

[45] Konishi F., Ohki S., Konagaya A., Umestu R., Ishii M., RABC: A conceptual design of pervasive infrastructure for browser computing based on AJAX technologies. In *Seventh IEEE International Symposium on Cluster Computing and the Grid, 2007, CCGRID 2007*, IEEE, 2007, 661-672

[46] Kruliš M., Falt Z., Zavoral F., Exploiting HTML5 Technologies for Distributed Parasitic Web Storage. *Databases, Texts*, 2014, 71.

[47] Krupa T., Majewski P., Kowalczyk B., Turek W., On-Demand Web Search Using Browser-Based Volunteer Computing. In: *Proceedings of 6th International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, Palermo, 2012, 184-190

[48] Kuchta P., in: Balicki J., Data partitioning and Task Management in the Clustered Server Layer of the Volunteer-based Computation System, in: Krawczyk H., Nawarecki E. (eds.), *Grid and Volunteer Computing*, Gdańsk University of Technology Press, Gdańsk, 2012, 40-52

[49] Kuhara M., Amano N., Watanabe K., Nogami Y., Fukushi M., A peer-to-peer communication function among Web browsers for Web-based Volunteer Computing, *Communications and Information Technologies (ISCIT), 2014 14th International Symposium on*, Incheon, 2014, 383-387

[50] Langhans, P., Wieser, C., Bry, F. Crowdsourcing MapReduce: JSMapReduce. In: *Proceedings of the 22nd international conference on World Wide Web companion*, International World Wide Web Conferences Steering Committee, May 2013, 253-256

[51] Lawton G., Distributed net applications create virtual supercomputers. *Computer*, 2000, **6**, 16-20

[52] Luther A., Buyya R., Ranjan R., Vanugopal S., Alchemi: A. NET-based Enterprise Grid Computing System. In: *International Conference on Internet Computing*, 2005, 269-278

[53] MacWilliam T., Cecka C., CrowdCL: Web-based volunteer computing with WebCL/ *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, Waltham, MA, 2013, 1-6.

[54] Martínez G., Val L., Implementing crossplatform distributed algorithms using standard web technologies, *Computing Conference (CLEI), 2014 XL Latin American*, Montevideo, 2014, 1-8

[55] Martinez G., Val L., Capataz - a framework for distributing algorithms via the World Wide Web. *CLEI Electronic Journal*, **18**, 2, 2015, 1

[56] Masiejczyk, J. and Balicki, J. and Korłub, W. and Paluszak J., Mersenne number finding and Collatz hypothesis verification in the Comcute grid system, in: Krawczyk H., Nawarecki E. (eds.), *Grid and Volunteer Computing*, Gdańsk University of Technology Press, Gdańsk, 2012, 148-162

[57] Meeds E., Hendriks R., Al Faraby S., Bruntink M., Welling M., MlitB:Machine Learning in the Browser, *PeerJ Computer Science*, **1**, July 2015, e11

[58] Merelo-Guervos J.J., P. A. Castillo, J. L. J. Laredo, A. Mora Garcia and A. Prieto, Asynchronous distributed genetic algorithms with Javascript and JSON. *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, Hong Kong, 2008, pp. 1372-1379

[59] Merelo-Guervos J.J, Garcia-Sanchez P., Designing and Modeling a Browser-Based Distributed Evolutionary Computation System. In: *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference*. ACM, 2015,1117-1124

[60] Merelo-Guervos J.J., Valdez M.G., Castillo P.A., Sanchez P.G., de las Cuevas P., Rico N., NodIO, a JavaScript framework for volunteer-based evolutionary algorithms: first results, Retrieved: http://arxiv.org/abs/1601.0160, 2016

[61] Miller D.M., The Online Community Grid: Volunteer Grid Computing with the Web Browser, Undergraduate Thesis, Georgia Institute of Technology, 2010, Retrieved at: http://hdl.handle.net/1853/33477

[62] Neary M. O., Brydon S. P., Kmiec P., Rollins S., Cappello P., Javelin++: scalability issues in global computing. In: *Proceedings of the ACM 1999 conference on Java Grande*, ACM, June 1999, 171-180

[63] Neary M. O., Phipps A., Richman S., Cappello P., Javelin 2.0: Java-based parallel computing on the Internet. In *Euro-Par 2000 Parallel Processing*, Springer Berlin Heidelberg, August 2000, 1231-1238

[64] Nisan N., London S., Regev O., Camiel N., Globally distributed computation over the Internet - the POPCORN project, In: *Proceedings of 18th International Conference on Distributed Computing Systems*, Amsterdam, 1998, 592-601

[65] Nov O., Anderson D.P., Arazy O., Volunteer computing: A model of the factors determining contribution to community-based scientific research. In: *Proceedings of the 19th International Conference on World Wide Web, WWW '10*. 2010. 741-750

[66] Ong T. M., Lim T. M., Lee B. S., Yeo C. K., Unicorn: Voluntary Computing Over Interner, *SIGOPS Oper. Syst. Rev.*, **36**, 2, 2002, 36-51

[67] Ryza S., Wall T., *MRJS:* A JavaScript MapReduce Framework for Web Browsers. Retrieved: *http://www.cs.brown.edu/courses/csci2950-u/f11/papers/mrjs.pdf* (2010).
[68] Sarmenta L.F.G., Bayanihan: Web-based volunteer computing using Java. In: *Worldwide Computing and Its Applications—WWCA'98*. Springer Berlin Heidelberg, 1998, 444-461
[69] Sarmenta L.F.G., Hirano S., Bayanihan: building and studying web-based volunteer computing systems using Java. *Future Generation Computer Systems*, **15**, 5–6, October 1999, 675-686
[70] Sarmenta L.F.G., An adaptive, fault-tolerant implementation of BSP for Java-based volunteer computing systems. In: *Parallel and Distributed Processing*. Springer Berlin Heidelberg, 1999, 763-780
[71] Simonarson S., Browser Based Distributed Computing, Retrieved: https://www.tjhsst.edu/~rlatimer/techlab10/SimonarsonPaperQ1-09.pdf, 2010
[72] Simpson R., Page K.R., De Roure D., Zooniverse: observing the world's largest citizen science platform. In *Proceedings of the 23rd International Conference on World Wide Web* (WWW '14 Companion). ACM, New York, NY, USA, 2014, 1049-1054
[73] Tilkov S., Vinoski S.,Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, **14**, 6, November 2010, 80-83
[74] Turek, W., Nawarecki, E., Dobrowolski G., Krupa T., Majewski P., Web Pages Content Analysis Using Browser-based Volunteer Computing. *Computer Science*, **14,**2, 2013, 215
[75] Urbah E., Kacsuk P., Farkas Z., Gilles F., Kecskemeti G., Lodygensky O., Marosi A., Balaton Z., Caillat G., Gombas G., Kornafeld A., Kovacs J., He H., Lovas R., EDGeS: Bridging EGEE to BOINC and XtremWeb. *Journal of Grid Computing*, **7**, 3, 2009, 335-354
[76] Van Kesteren A., Cross-origin resource sharing. *World Wide Web Consortium (W3C) Recommendation*, Retrieved: https://www.w3.org/TR/cors/, January 2014
[77] Vanhelsuw L., Create Your Own Supercomputer with Java, *JavaWorld Online Magazine*, Jan 1997, Retrieved: http://www.javaworld.com/jw-01-1997/jw-01-dampp.html
[78] Venkataraman, N., A Survey on Desktop Grid Systems-Research Gap. In *Proceedings of the 3rd International Symposium on Big Data and Cloud Computing Challenges (ISBCC–16')* , Springer International Publishing, 2016,183-212
[79] Wilkinson S. R., Almeida J. S., QMachine: commodity supercomputing in web browsers. *BMC bioinformatics*, **15**, 1, 2014, 1
[80] Yao P., White J., Sun Y., Gray J.. Gray computing: an analysis of computing with background JavaScript tasks. In: *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, **1**, IEEE Press, Piscataway, NJ, USA, 2015, 167-177
[81] Zakai A., Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion* (OOPSLA '11). ACM, New York, NY, USA, 2011, 301-312. DOI=http://dx.doi.org/10.1145/2048147.2048224
[82] Ziegler G. M., The great prime number record races. *Notices of the AMS*, **51**, 4, 2004, 414-416

[83] Zorrilla M., Martin A., Tamayo I., Aginako N., Olaizola I. G., Web Browser-Based Social Distributed Computing Platform Applied to Image Analysis. In: *Proceedings of 2013 Third International Conference on Cloud and Green Computing (CGC)*, Karlsruhe, 2013, 389-396

[84] Fabisiak T., Web Browser Based Distributed Computing, Master Thesis, Poznan University of Technology, 2016.

Received 26.08.2016, Accepted 19.01.2017