

# Pando: Personal Volunteer Computing in Browsers

Erick Lavoie  
School of Computer Science  
McGill University  
Montreal, Quebec, Canada  
erick.lavoie@mail.mcgill.ca

Frederic Desprez  
Antenne Inria Giant  
INRIA Grenoble Rhône-Alpes  
Grenoble, France  
Frederic.Desprez@inria.fr

Laurie Hendren  
School of Computer Science  
McGill University  
Montreal, Quebec, Canada  
hendren@cs.mcgill.ca

Miguel Pupo Correia  
INESC-ID  
Lisboa, Portugal  
miguel.p.correia@tecnico.ulisboa.pt

## Abstract

The large penetration and continued growth in ownership of personal electronic devices represents a freely available and largely untapped source of computing power. Moreover, the large environmental and social impact of producing these devices suggests we should better use those that already exist. We aim to make these devices available for parallel computations to both scientists and other programmers of the general public, for their personal projects, and in the simplest way possible to program and to deploy. We named our approach to distributed computing *personal volunteer computing*.

We designed, implemented, and tested Pando, a new distributed computing tool based on a declarative concurrent programming model, organized around the pull-stream design pattern, and implemented using JavaScript, WebRTC, and WebSockets. This tool enables a dynamically varying number of failure-prone personal devices contributed by volunteers to parallelize the application of a function on a stream of values, by using the devices' browsers.

To illustrate Pando's capabilities, to show its benefits as well as its limitations, we implemented a variety of applications including crypto-currency mining, hyper-parameter optimization in machine learning, crowd computing, and open data processing and tested it using diverse devices we have accumulated over the years. Pando, both as a tool and a reference design, should therefore be a useful addition to the parallel toolbox of a multitude of users and a complementary approach to existing parallel and distributed computing alternatives.

## 1 Introduction

More than 1.5 billion cell phones were sold in the world in 2017 [21] and the computing power of the highest-end devices today rivals that of desktops and laptops [60]. They collectively represent an *immense source of largely untapped computing power*. At the same time, the large environmental and social impact of producing these devices suggests *we should better use those that already exist*. It is now common

to own many unused laptops, tablets, and cell phones, many of which could still make useful computing contributions.

Technical, financial, and administrative barriers in existing systems prevent a large part of the *general programmer population*, especially in humanities and developing countries, from using these resources for parallelizing the execution of many possible *personal projects*. This suggests a vast under-explored design space for various *personal tools* that use alternative computing resources. In parallel, the wide popularity of social applications makes it easier than ever for individual users to leverage their *personal social network* for help, but this possibility has little been used so far to meet their *computing needs*. *Personal devices*, *personal projects*, *personal tools*, and *personal social networks* form together a new *viewpoint* from which new tools may be developed, which we call *personal volunteer computing*.

The major challenges in designing *personal volunteer computing tools* for general programmers are analogous to those that have prompted the articulation of *intermediate technologies* [94] in the 1970s for promoting economic development in developing countries. Then, the limited access to high-speed reliable infrastructure, specialists, capital, and world-wide resources had promoted a development vision based on *local knowledge*, *local resources*, and *simple reliable designs* that may be implemented, maintained, and improved *by their users for their personal needs*. Translated to the design of distributed computing tools for today, the major challenges are to find simple designs that are applicable for a wide range of applications with minimum needs for hosted infrastructure or dedicated hardware, and explain them in a way to foster local appropriation and replication in many programming environments.

The *declarative concurrent* programming paradigm [101] greatly simplifies reasoning about concurrent processes: it abstracts the non-determinism in the execution by making it non-observable. This paradigm has already enjoyed great practical successes with the popular MapReduce [44] and Unix pipelining [64] programming models. Could it also be useful for building *personal volunteer computing tools*? This

paper answers positively through the design of Pando, a tool that enables a dynamically varying number of failure-prone personal devices contributed by volunteers to parallelize the application of a function on a stream of values by using the devices' browsers.

This paper makes the following contributions: (1) it introduces and articulates the personal volunteer computing viewpoint within the context of other parallel and distributed computing approaches; (2) it presents the design of Pando through both high-level design principles and a concrete working implementation, itself organized around the pull-stream design pattern and based on JavaScript [25], WebSockets [7], and WebRTC [17] to enable its execution inside browsers; (3) it presents our novel StreamLender abstraction that encapsulates the key properties of Pando's programming model necessary to distribute a single stream to a dynamically varying number of failure-prone processing devices; (4) it reports on the application of Pando to 7 applications, including crypto-currency mining, crowd computing, machine learning hyper-parameter optimization, and open data processing in combination with other peer-to-peer data distribution protocols; (5) it presents real-world measurements on the benefits that can be obtained showing that both Pando and personal volunteer computing are a useful and complementary addition to existing parallel and distributed computing approaches. The implementation of Pando is open source [32] and its individual components may be repurposed in many different applications.

The rest of this paper is organized as follows. We introduce personal volunteer computing by studying the limitations of other approaches for our intended context in Section 2. We present the overall design of Pando in Section 3. We provide the key properties and behaviour of the StreamLender abstraction in Section 4. We present the different applications in Section 5 and evaluate the benefits and limitations of parallelizing them in real-world deployments in Section 6. We compare the specificities of our design to related work in Section 7. We conclude with a brief recapitulation of the paper and future work in Section 8.

## 2 Personal Volunteer Computing

Personal volunteer computing aims to address the needs of scientists and programmers of the general population, for their *personal project*, using *personal tools*, leveraging their *personal devices* and those of their *personal social network*. In contrast, past parallel and distributed computing approaches have typically focused on addressing the most pressing needs of industry and research groups world-wide by producing efficient and scalable solutions that are unfortunately often too complex, expensive, and time consuming to use in a more *personal context*. We quickly survey the most representative approaches to highlight their limitations, which will motivate our design principles in the next section.

Previous parallelism approaches such as GPU programming and heterogeneous computing [78] have typically focused on maximizing the performance on newer hardware architectures. Acquiring the targeted devices may be too expensive for many use cases, the newer techniques may not work on the devices most people already own, and are usually sufficiently complicated to implement to require dedicated experts for the task. Most of the parallelism solutions are therefore better suited as reusable libraries and runtime systems rather than general-purpose coordination tools.

Cloud computing platforms [36, 40] introduce a *financial barrier* for those that do not have access to financial instruments, such as a bank account or a credit card. Existing open source cloud solutions such as Open Compute [6], Open Stack [4], and Hadoop [9] are designed for making available co-located dedicated devices to many users in a shared platform, with a complexity and resource requirements that limits their deployment on an ad hoc set of personal devices.

Edge [95] and gray [85] computing platforms extend existing clouds to use personal devices for computation and therefore inherit their financial and technical barriers for usage and deployment. Moreover, they require trust in the operator of the platforms that sensitive personal data from participating users will be correctly managed.

Grid computing platforms [51, 55, 56] are designed to make available devices from many collaborating organizations available to researchers with a unified interface. Access to the existing platforms requires administrative permissions, which are typically reserved to researchers from public institutions and not available to programmers from the general public. The tools themselves include facilities to manage complex access control policies that are not necessary when using personal devices for computations.

Previous peer-to-peer computing solutions [27, 46, 59, 66, 67, 82, 88, 99, 107] have typically focused on building global platforms, shared by many users. This introduces the need for load-balancing and multiplexing multiple tasks on the same set of participating devices, which complicates their implementation. Moreover, in a volunteer context, maintaining the system while no tasks are being executed has a cost in time and attention, with no immediate benefit, which disincentivises voluntary contributions.

Volunteer computing tools [30, 93] have historically focused on high-profile scientific projects with large computation needs. They require dedicated servers, a significant deployment effort, and are complicated to implement because of the necessity to deal with large scale issues, such as invalid results by malicious contributors. This limits their deployment on personal devices for smaller projects.

Previous browser-based computing solutions [48, 50, 102] have typically been developed as extensions of volunteer computing tools for similar tasks, inheriting their technical limitations. Moreover, the quick deprecation of past Web technologies, such as Java Applets on which the first tools

relied [37, 43, 52, 63, 79, 92], in combination with the limited funding available to maintain them resulted in many previous tools being now unusable.

The collective computing power and proliferation of personal devices has created an opportunity for *edge computing* and *peer-to-peer* systems to flourish. Both have aimed at building *global platforms* to answer the needs of all users within a single platform. There was however no clearly articulated approach to answer the needs of users at a *personal scale*, at the level of a single individual and their personal social network.

Compared to previous approaches, personal volunteer computing therefore strives to make tools for the parallel execution of workloads useful to the largest number of programmers by executing them on their personal devices. The major challenge in creating such tools is to make them useful for a wide range of applications while keeping them simple to program and to implement, within the limited resources available for their development, maintenance, and operation. These challenges had not previously been identified as a research direction; our first contribution with this paper is therefore to have articulated this *personal volunteer computing* approach.

### 3 Pando

Pando is the first tool explicitly designed for the purpose of personal volunteer computing. We first explain how to use it and its concrete benefits using one of our supported application (Section 3.1). We then articulate the design principles that enable those benefits (Section 3.2). We continue with a more detailed explanation of Pando’s programming model (Section 3.3) and finally present an overview of how it is implemented in a concrete system (Section 3.4).

#### 3.1 Usage Example

Suppose a user is working on a personal project involving a 3D animation, as illustrated in Figure 1, and the rendering uses *raytracing* [106], which is a computationally expensive technique. To accelerate the rendering of the entire animation, they want to *parallelize* the rendering of individual frames, while still obtaining them in the *correct order*.

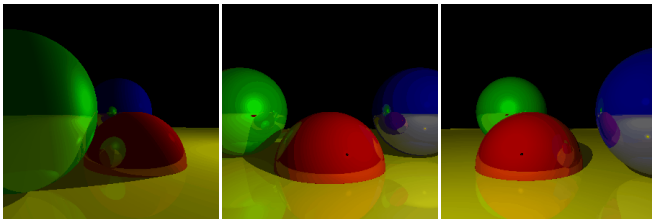


Figure 1. Rotation animation around a 3D scene.

If this were a professional project, our user could have relied on professional solutions [20, 26]. However, these are

often too expensive for personal projects and do not easily leverage the computing power of personal devices users already own. Instead, they can use Pando through a *simple programming interface* and a *quick deployment* solution.

#### 3.1.1 Programming Interface

Pando’s distribution of computation is organized around a *processing function* which is applied to a stream of *input values* to produce a stream of *outputs*. In this particular example, the processing function performs the *raytracing of the scene* from a particular *camera position* and outputs an *array of pixels*. The animation consists in a sequence of positions of the camera rotating around the scene.

Pando’s implementation parallelizes the execution of code in JavaScript by using the Web browsers of personal devices. To leverage those capabilities, a user writes a minimal amount of glue code to make the processing function compatible with Pando’s interface, as illustrated in Figure 2. In this example, the raytracing operation is provided by an external library, taken unmodified from the Web, which is first imported. Then a processing function using the required library is exposed on the module with the `’/pando/1.0.0’` property, which indicates it is intended for the first version of the Pando protocol. The function takes two inputs: `cameraPos`, the camera position for the current frame and `cb`, a callback to return the result. The body of the function first converts the camera position, which was received as a string, into a float value, then renders the scene. The pixels of the rendered image are then saved in a buffer, compressed with `gzip`, and output as a `base64` encoded string [2], which simplifies its transmission on the network.<sup>1</sup> The result is then returned to Pando through the callback `cb`. In case an error occurred in any of those steps, an error is caught then returned through the same callback.

```
1 // Import existing function
2 var render = require('raytracer')
3 // Import compressing module
4 var zlib = require('zlib')
5 module.exports['/pando/1.0.0'] = function(
6   cameraPos, cb) {
7   try {
8     var pixels = render(parseFloat(cameraPos))
9     cb(null, zlib.gzipSync(new Buffer(pixels)).
10      toString('base64'))
11   } catch (err) {
12     cb(err)
13   }
14 }
```

Figure 2. JavaScript programming interface example for rendering with raytracing.

<sup>1</sup>Those last three operations take a negligible amount of time compared to rendering the image.

The glue code should then be saved in a file, `render.js` in this example, and all library dependencies should be accessible using the Node Package Manager (NPM) conventions [23], typically in a `node_modules` sub-directory. Pando will automatically bundle all the dependencies on startup and adapt the code for the browser context by internally using `browserify` [13].

Pando is compatible with the Unix standard process interface, i.e. it can either receive its inputs on the *standard input* or as command-line arguments and it produces outputs on the *standard output*. In Figure 3, we connect Pando with other tools using bash scripting. The camera positions are provided as strings on the standard input by `generate-angles.js`, the rendered images are produced on the standard output as strings by Pando, and the assembly of the frames into a GIF animation is done by `gif-encoder.js`. All tools in the sequence are connected through Unix streams using the pipe operator (`|`). Pando could also be scripted from any other programming environment that supports the creation of Unix processes; the creation of inputs and the post-processing of outputs therefore need not be in JavaScript.

```
1 $ ./generate-angles.js | pando render.js --stdin |
  ./gif-encoder.js
2 Serving volunteer code at http://10.10.14.119:5000
```

**Figure 3.** Unix programming interface example for rendering inputs and processing outputs. After starting, Pando lists the URL necessary for deployment on the standard error.

### 3.1.2 Deployment

A user deploys Pando by starting it on the command-line<sup>2</sup>, as illustrated in Figure 3. Then they should wait for URL messages to appear. When displayed, those messages indicate that Pando is ready for other devices to join.

Upon joining, additional devices will process individual frames in parallel. In one possible example execution, illustrated in Figure 4, a tablet joins by opening the volunteer URL, then renders an image, then a faster phone joins, also renders an image, then the tablet crashes, and the phone takes over for the missing image. Communications happen over a choice of WebRTC [17], a recent peer-to-peer protocol for browsers, or WebSocket [7].

A user can invite friends to add their devices, even if they are outside the local network. To do so, the user deploys a small micro-server we built for Pando [33] on a platform that provides a public IP address, such as Heroku [22]. Being publicly accessible, the URL can then be shared to friends on existing social media. After opening the URL, a WebRTC connection will directly connect joining devices.

<sup>2</sup>After installing, ex: `npm install --global pando-computing` [31].

As illustrated in this deployment example, Pando *dynamically scaled* to accommodate the number of participating devices and *gracefully tolerated failures* with no particular programming effort from the user beyond specifying a function to process a single value. Moreover, the user did not need to (1) buy new devices, (2) create an account or obtain administrative permissions, (3) use financial instruments, (4) accommodate device specificities, or (5) wait for resources to be freed. The user could also (1) combine Pando with existing Unix tools, (2) use social media to request for help, and (3) know their data has only been shared between trusted devices.

## 3.2 Design Principles

The previous usage example provided significant benefits because we designed Pando around the following design principles (DPs), which we derived from the limitations of previous approaches (Section 2).

*Specific deployment* (DP1): the deployment of the tool that connects the different volunteers is specific to: (1) a single project, (2) a single known user with an existing social presence, either through the contacts of volunteers, or an identity in a social platform, and (3) the lifetime of the corresponding tasks, after which it shuts down.

*Compatible with a wide variety of existing personal devices* (DP2): the tool should leverage desktops, laptops, tablets, phones, embedded devices, and personal appliances that people already own.

*Easy to program* (DP3): the implementation of tasks should be done with a minimum of programming effort for use in a distributed setting. Ideally, it should be as easy to program in a distributed setting as in a local one.

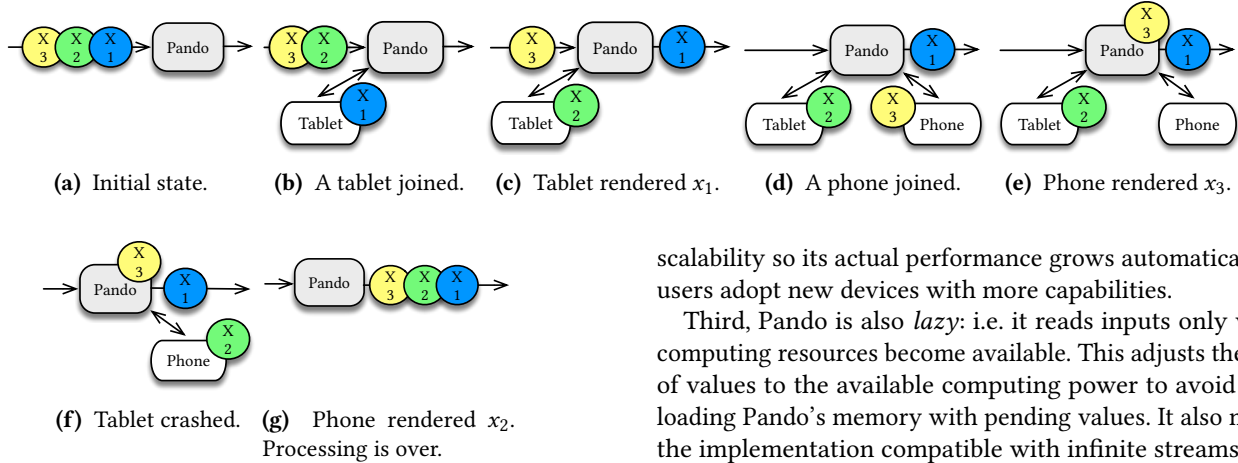
*Quick to deploy* (DP4): the tool should require little installation effort, should start processing quickly after launch, and then should dynamically scale up to benefit from help obtained through social networks.

*Composable and modular* (DP5): the tool should focus on coordinating contributing volunteers' devices but otherwise should rely on other tools and technologies for the rest of the needs of users. The core abstractions used in particular tools should be applicable to other uses. Tools should also combine with high-performance libraries, when available, to leverage the latest results of parallelism research without making the tools themselves more complicated.

## 3.3 Programming Model

In effect, Pando's programming model corresponds to a streaming version of the functional *map* operation: Pando applies a function  $f$  on a series of input values  $x_i$  to obtain a serie of results  $f(x_i)$ . Its implementation is free to process inputs in any order but outputs results in the order of their corresponding inputs.

We chose a streaming programming model because it is simple to program (DP3) yet powerful enough to coordinate



**Figure 4.** Deployment example.

the usage of multiple devices in parallel (DP2). The reason is that it belongs to the *declarative concurrency* paradigm [101] which *abstracts the non-determinism of executions by making it non-observable to the programmer*. In other words, a declarative concurrent program outputs the same result regardless of the order in which the various threads that compose the execution complete their tasks. That makes Pando as simple to program in a sequential setting with a single participating processor as for a parallel case with dozens. While it is implied by the definition of the *map* operation, it is worth noting that the *ordering* of outputs is important to preserve the *declarative concurrency* property; otherwise the relative speed of processors could influence the order of the results and make the non-determinism observable.

We initially chose the *streaming map* programming model because it fits more problems than the *bag-of-tasks* model of typical volunteer computing problems, which usually have independent inputs with no ordering requirement. Some applications however, such as the sequence of images that compose the animation of our previous example (Section 3.1), do require a particular order. Problems with *unordered* inputs can be reduced to a streaming version simply by incrementally traversing the values in an arbitrary order, making the streaming model more general. The streaming version also enables working with an *infinite* number of values and applications requiring *feedback loops* (Section 5).

We also chose a number of additional distributed properties for Pando to make it easy to program (DP3) and fast to deploy (DP4).

First, participating devices may join *dynamically*, at any time during execution. Pando’s computing power will grow automatically. This removes the overhead of registering computing resources in advance and simplifies scaling for quick deployment.

Second, The potential number of participating devices is *unbounded*. Pando strives to provide the illusion of infinite

scalability so its actual performance grows automatically as users adopt new devices with more capabilities.

Third, Pando is also *lazy*: i.e. it reads inputs only when computing resources become available. This adjusts the flow of values to the available computing power to avoid overloading Pando’s memory with pending values. It also makes the implementation compatible with infinite streams with no additional implementation effort on our part. Users get support for laziness with no additional programming effort.

Last, Pando also *tolerates failures* of participating devices, making those failures transparent to the programmer. We chose a *crash-stop* failure mode<sup>3</sup>, in which participating devices will always faithfully carry their assigned task without deviating from their prescribed behaviour until they either suddenly crash or disconnect. This model corresponds to failures in which a browser tab, that executes computations, is suddenly closed or to a loss of network connectivity. In the presence of such failures, Pando guarantees *liveness*: once an input  $x_i$  has been read, if there are active participating devices, Pando will eventually provide  $f(x_i)$ .

The crash-stop failures of participating devices can be detected because we assume a *partially synchronous* execution<sup>4</sup>: most of the time, messages will be delivered within a specified *time bound*. This corresponds to the ability of communication channels such as TCP [1] and WebRTC [17] to suspect failures by failing to receive the acknowledgment of a *heartbeat* message within a specified time bound.

In terms of performance goals, we decided to focus on maximizing *throughput* with the additional following two properties. Pando distributes values to participating devices *conservatively*: a value is sent to at most one device for processing. The device will either produce a result or will crash, in which case the value will be sent to another device. This ensures participating devices process a maximum number of values simultaneously. Moreover, the rate at which values are submitted to participating devices *adapts* to their processing

<sup>3</sup>Failure modes can range from *crash-stop*, in which a process follows its instructions then may crash and stop sending messages forever, passing by *crash-recovery*, in which a process may fail then recover and try participating again, to *byzantine*, in which a process may deviate arbitrarily from its instructions including intentionally sending messages to hamper progress.

<sup>4</sup>Timing assumptions may range from *fully synchronous*, in which there is an upper time bound on message delivery, passing by *partially synchronous* [49], in which there is a time bound on message delivery that it will apply only *eventually* after an unknown delay, and culminating in *asynchronous*, in which there are no time bound on message delivery.

speed. Devices with a faster processing speed will receive more values to process, maximizing resource utilization.

This combination of programming model properties, which is summarized in Table 1, provides a powerful yet easy-to-use programming model as shown by the breath of applications categories supported (Section 5).

<b>Streaming Map</b>	$x_1, x_2, \dots \rightarrow f(x_1), f(x_2), \dots$
<b>Ordered</b>	Outputs provided in order.
<b>Dynamic</b>	New devices may join any time.
<b>Unbounded</b>	No <i>a priori</i> limit on participants nb.
<b>Lazy</b>	Inputs read when resources are avail.
<b>Fault-tolerant</b>	<i>Crash-stop</i> failures are tolerated.
<b>Conservative</b>	A single copy submitted at a time.
<b>Adaptive</b>	Faster devices receive more inputs.

**Table 1.** Summary of the programming model properties.

### 3.4 Implementation Overview

Our implementation was first based on our choice between available Web technologies (Section 3.4.1). We then organized it around a declarative concurrent paradigm to simplify both its usage and implementation effort (Section 3.4.2). We finally designed a reusable architecture by decomposing it into modules and communication technologies (Section 3.4.3).

#### 3.4.1 Technology Choices

We based our implementation on Web technologies because they are compatible with a wide number of personal devices, from smartphones and embedded devices to tablets, laptop, and desktops computers (DP2). In addition, they have a number of interesting characteristics. Virtual machines in modern browsers execute JavaScript at a speed within a factor of 3 of equivalent numerical code written in C [60, 65], making the performance sufficiently close to C to benefit from executing tasks inside multiple parallel Web pages. Browsers also provide a security sandbox that prevents code executing within a web page from tampering with the host operating system. *WebRTC* [17], enables direct communication between browsers, in many cases even in the presence of Network Address Translation (NAT), which removes the need for a server to relay all communications between the tool and the volunteers' devices. Links shared on social media platforms enable their users to quickly mobilize their social networks, opening the possibility to make volunteer computing projects go viral.

#### 3.4.2 Declarative Concurrency With Pull-Streams

Pando provides a *declarative concurrent* abstraction [101] of the parallel execution of the different participating processors (Section 3.3). Mainstream languages, such as JavaScript,

have not yet integrated features that make that style of programming widely accessible. We therefore instead based our design and implementation on the pull-stream design pattern [98], a functional code pattern that enables streaming modules to be built by following a simple callback protocol. It only requires support for higher-order functions from the base language. Implementations of abstractions built by following the pattern should therefore be straight-forward to port to many programming languages of today.

The pull-stream design pattern has originally been proposed by Dominic Tarr [98] as a simpler alternative to Node.js streams, that were plagued with design issues that had to be maintained for backward-compatibility. A community has grown around the pattern and more than a hundred modules have been contributed so far [15].

Perhaps, the simplest example of pull-stream modules is a source that lazily counts from 1 to  $n$ , connected to a sink that consumes all values and then stops, as illustrated in Figure 5. The callback protocol essentially consists in a request followed by an answer. The request may be used to ask for a value, abort the stream normally, or fail because of an error. Symmetrically, the answer may then produce a value, signify the end of the stream, or stop because of an error. A module may also both consume and produce values, in which case it can be used between a source and a sink. This is illustrated in Figure 6.

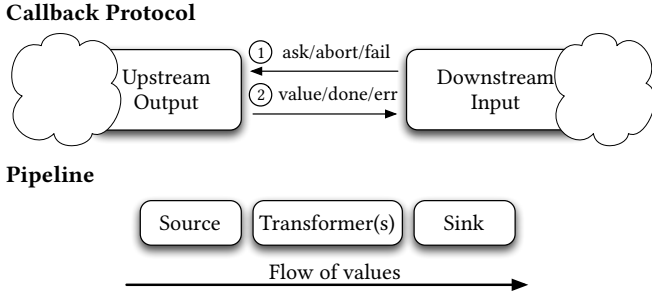
```

1 function source (n) {
2   var i = 1
3   return function output (abort, cb) {
4     if (abort)
5       return cb(abort, undefined)
6     else if (i<=n)
7       return cb(false, i++)
8     else
9       return cb(true, undefined)
10  }
11 }
12
13 function sink (request) {
14   request(false, function answer (done, v) {
15     if (done) return
16     else request(false, answer)
17   })
18 }
19
20 sink(source(10))
21 var pull = require('pull-stream')
22 pull(source(10), sink) // equivalent to line 20

```

**Figure 5.** Pull-stream example.

While the pattern does not simplify the task of implementing pull-stream modules, once implemented, the modules provide clear semantics and are easy to combine because



**Figure 6.** Pull-stream design pattern: callback protocol on top and pipeline of composable modules at the bottom.

they can provide declarative concurrent abstractions. Using the pull-stream design pattern therefore makes the rest of the implementation of Pando easier to reason about.

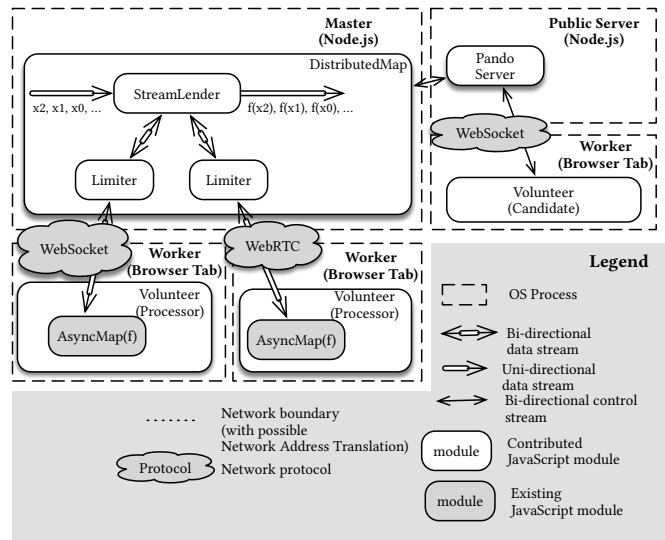
### 3.4.3 Architecture

The core modules of Pando and the way they are connected is illustrated in Figure 7. They work together to implement a *distributed map* that processes a stream of values  $x_i$  with a function  $f$ . Our implementation uses Node.js but could also work as a hosted Web application. Deployment consists in executing the tool on the command-line, which starts the Master process. HTTP connections from volunteers’ devices may then be made directly to the Master, if on the same local area network (not shown), or through a Public Server, if direct connectivity is not possible. The HTTP connection is used to obtain the Worker code including the  $f$  function and eventually establish either a WebSocket [7] or WebRTC [17] connection. The bootstrap of the WebRTC connection, which requires *signalling* of possible connection endpoints between peers, is done through a Public Server using a separate WebSocket connection. That connection closes after the WebRTC connection is established. Since signalling requires little resources, the Public Server could be executed on a small personal server such as a Raspberry Pi board [24] or the free tier of a cloud such as Heroku [22].

The pull-stream abstractions we designed and reused are shown as modules within the different processes, respectively in white and grey. The core coordination is performed by our novel *StreamLender* abstraction (Section 4), which creates multiple concurrent bi-directional sub-streams, one for each worker. A sub-stream continuously borrows values from the input of *StreamLender* and return results that are eventually returned on its output. The sub-streams are dynamically created as Workers join. We use existing libraries that expose WebRTC and WebSocket channels as pull-streams. Since their implementation *eagerly reads all available values* on the sending side, we bound the total number of values that can be borrowed before a result is returned using our new *Limitier* module. The bound can be parameterized using an argument passed to Pando on startup. The

actual processing of values is done inside Workers using the existing *AsyncMap* [15] module that applies the function  $f$  on the different inputs.

Pando trivially enables parallel processing on multicore architectures on a single machine while enabling dynamically scaling up to other devices if necessary, making the tool useful in many contexts. Our design should also work with other technology choices, which could be mandated because users require specific libraries and technologies that are not available for the Web yet. For example, users may depend on specific numerical libraries available in Python/Numpy, MATLAB, or R. In that case, it should be straightforward to adapt the design by relying on TCP for communication and porting our modules to a different language.



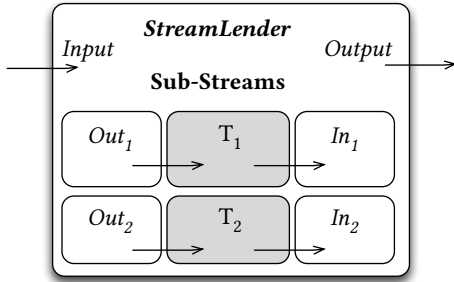
**Figure 7.** Architecture of Pando.

## 4 StreamLender

*StreamLender* is our novel abstraction that splits an input stream into multiple concurrent sub-streams and then merges back the results in a single output stream. The actual processing of the values is done using other transformer modules, as illustrated in Figure 8. We provide a usage example in Figure 9.

*StreamLender* encapsulates the *streaming, ordered, dynamic, fault-tolerant, conservative, and adaptive* properties of Pando’s programming model (Section 3.3), independently of a particular communication protocol or other input-output libraries. To the best of our knowledge, *StreamLender* is the first articulation of those properties in a reusable abstraction for distributed stream processing.

The complete and tested JavaScript implementation that we built and used in Pando is available as an independent pull-stream module [34]. The synchronization of events happening through callbacks initiated by multiple concurrent



**Figure 8.** StreamLender and its sub-streams. External transformer(s) modules connected to the sub-streams are greyed. They represent modules such as the Limiter of Figure 7.

```

1 var pull = require('pull-stream')
2 // StreamLender
3 var lender = require('pull-lend-stream')
4 var limit = require('pull-limit') // Limiter
5 pull(
6   pull.count(10),
7   lender,
8   pull.drain()
9 )
10 var duplex = ... // On webrtc connection opened
11 lender.lendStream(function (err, subStream) {
12   if (err) return
13   pull(
14     subStream.source, // output
15     limit(duplex),
16     subStream.sink    // input
17   )
18 })

```

**Figure 9.** StreamLender usage example.

streams was tricky to correctly implement and is rather cumbersome to decipher through the source code. We therefore derived a more readable pseudo-code version that uses explicit waiting primitives and events that correspond to the invocation of callbacks to help reimplementations, available in the supplemental material. As a sample, Algorithm 1 shows how the requests made on a sub-stream output are answered, either with a value from another sub-stream that failed, a new value requested on the StreamLender *Input*, or a *done* if no more values are left to process. The ordering and synchronization of outputs is done through a priority queue ordered by the index of the values in the stream.

## 5 Applications

Pando can be applied to a wide range of applications. In this section, we present some examples according to their dataflow pattern, i.e. how data flows between Pando and

**Algorithm 1** Sub-stream output ask request.

```

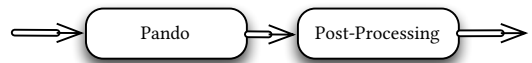
1: upon  $Out_i:ask\langle\rangle$ 
2:   if  $failed \neq \emptyset$  then
3:     ANSWERWITHFAILEDVALUE( $Out_i$ )
4:   else if  $Input$  has terminated ( $done$  or  $err$ ) then
5:     WAITONOTHERS( $Out_i$ )
6:   else ▷ Lazily read a new value
7:     trigger  $Input:ask\langle\rangle$ 
8:     wait  $Input$  answer
9:     if  $answer = Input:value\langle v\rangle$  then
10:      remember  $v$ 
11:      trigger  $Out_i:value\langle v\rangle$ 
12:     else
13:      WAITONOTHERS( $Out_i$ )
14:
15: procedure ANSWERWITHFAILEDVALUE( $Out_i$ )
16:   let  $v$  be the oldest value of  $failed$ 
17:   remember  $v$ 
18:    $failed \leftarrow failed \setminus \{v\}$ 
19:   trigger  $Out_i:value\langle v\rangle$ 
20: procedure WAITONOTHERS( $Out_i$ )
21:   wait until last result received or  $failed \neq \emptyset$ 
22:   if last result received then
23:     trigger  $Out_i:done\langle\rangle$ 
24:   else
25:     ANSWERWITHFAILEDVALUE( $Out_i$ )

```

other tools and protocols. We implemented each application using components built as separate Unix tools but the same components could be implemented as pull-stream modules and combined into a single application as well, either as a standalone webpage or a smartphone application. We summarize key aspects of each application.

### 5.1 Pipeline Processing

*Pipeline processing* is a sequence of independent processing stages applied to a stream of inputs, as illustrated in Figure 10. Traditional *bag-of-tasks* problems, typically associated with volunteer computing, can also be solved with this approach, by listing each individual task in sequence.



App.	Inputs	Pando	Post
Collatz	Ints	Nb of steps	Max
Raytrace	Camera pos.	Raytracing	Anim. gif
Arxiv	Meta-info	Human tagging	None
SL test	RNG seeds	Rand. exec.	Monitor fail.
ML agent	Hyperparams	Simulation	None

**Figure 10.** Pipeline processing dataflow and examples.



This approach is straight-forward to use with Pando and easiest to combine with other Unix tools. We implemented five applications that show diverse use cases. *Collatz* implements the Collatz Conjecture [16], an ongoing BOINC project, to find an integer that results in the largest number of computation steps. Our implementation was compiled from Matlab to JavaScript using the Matjuice compiler [14, 54] and then adapted to use a BigNumber library. Other languages with a JavaScript compiler may therefore benefit from Pando without having to implement a distribution strategy. *Raytrace* distributes the rendering of individual frames of a 3D animation and assembles them in an animated gif (Section 3.1). A similar strategy could be useful to integrate in open source animation tools for artists that do not have access to a rendering farm. *Arxiv* distributes the tagging of interesting papers to a group of collaborators, a form of *crowdprocessing*, by using the browser as a user interface rather than a processing environment. A similar approach could be used to quickly launch an online rescue search using satellite or aerial images in times of disasters. *StreamLender test* performs random executions of StreamLender to find cases where the invariants of the pull-stream protocol are violated. It helped us fix three bugs in corner cases that were not found with manually written tests and then scale up the testing strategy to perform millions of executions quickly without finding errors, increasing confidence that our implementation is correct. *Machine learning agent* searches for the optimal learning rate, an hyperparameter, that helps an autonomous agent in a simulated environment quickly learn sequences of steps that result in rewards. This approach could be beneficial to train deep neural networks in browsers. In this particular example, the training phase is interactive: the user can see the behaviour of the agent as it is learning and early-abort a particular hyper-parameter case if the agent fails to learn, a form a *hybrid human-machine learning collaboration*.

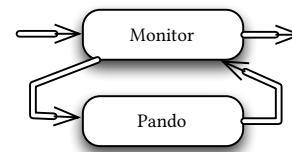
## 5.2 Synchronous Parallel Search

The structure of *blockchains* in crypto-currencies such as Bitcoin [81] mandates a *synchronous parallel search* organization: all miners compete to find a random value, or *nonce*, such that the hash of the nonce and the block of transactions combined is inferior to a difficulty threshold, itself controlling the probability of finding a nonce in the first place. Once a valid nonce has been found, the list of blocks is extended, and all miners move on to work on the next block.

In the case of Bitcoin, there is no upper bound on the amount of computational power required to mine the next block because the difficulty is automatically adjusted such that the time between each successful block is roughly ten minutes. The increasing difficulty, and therefore computational requirements to mine a new block, makes it increasingly costly for malicious actors to generate a fork of the chain of blocks at arbitrary places, preserving the integrity of

the longest chain of blocks. This results in a global consensus on the history of transactions.

A synchronous parallel search introduces a *feedback loop* in the flow of data, as illustrated in Figure 11, because the next input to process is determined by the last valid result obtained. In our implementation, a monitor therefore lazily provides *mining attempts* to Pando, including the current block and a range of integers to test. It generates as many as there are participating workers. Each worker tests all integers in the range and answers either with a valid nonce or a failure and then requests a new mining attempt. The monitor keeps providing new mining attempts until a valid nonce is found and then moves on to the next block. In this example, both the list of inputs, as blocks, and the computational requirements are potentially infinite, making a lazy streaming approach quite natural to use.



App.	Inputs	Monitor	Pando
Crypto-curr.	Blocks	Block + Range	Mine nonce

**Figure 11.** Synchronous parallel search dataflow and example.

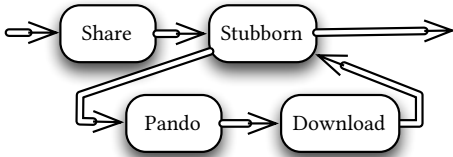
Bitcoin miners nowadays use dedicated hardware that is several orders of magnitude faster than the performance that can be achieved with an equivalent implementation executing in JavaScript. There is therefore limited practicality in mining Bitcoins in browsers, even with the gains obtained by parallelizing the task. Nonetheless, *proof-of-work* algorithms have been designed to work better on regular CPUs [80]. There may therefore be potential applications in mining those emerging crypto-currencies with Pando to support charities and fund open source software.

## 5.3 Stubborn Processing with Failure-Prone External Data Distribution

Our focus in the design of Pando has been on the coordination aspects of distributing work. Otherwise we rely on external data distribution protocols for managing large data. We implemented distributed blurring of Landsat-8 open satellite dataset [90] by distributing the data with the DAT protocol [19], itself accessible in the Beaker browser [18], a fork of Chromium [3]. A similar approach could also work with WebTorrent [10] running in browsers that support WebRTC.

Managing data outside of Pando introduces an additional failure mode: it is possible to receive a successful result but the worker may still crash before the results' data have been

fully downloaded. To address the issue our application outputs a result only after a successful download. Otherwise, the input is resubmitted for computation. The monitoring to implement that feedback loop has been factored into our new *stubborn* pull-stream module [35] which can be combined with sharing and downloading modules that are specific to a particular protocol, as illustrated in Figure 12.



App.	Inputs	Share/Down.	Pando
Batch proc.	Landsat-8 imgs	DAT protocol	Blur filter

**Figure 12.** Stubborn processing with external data distribution dataflow and example.

This use of Pando could be especially appropriate in cases where there is a growing availability of open datasets combined with limited funding and resources available to process them, as is the case for many citizen initiatives.

## 6 Evaluation

Pando enables many personal devices to jointly execute a task in parallel. However, the speedup benefits depend on their actual capabilities. The difference in performance between older and newer devices is sometimes sufficiently large that it obviates the contributions of the former. In this section, we therefore first quantify the performance available in common and less common devices to select the most useful ones. We then quantify the performance we obtained in using them in parallel with Pando to show there are indeed benefits in our approach, both in improving on the performance available on a single device and by using a combination of other, possibly older, devices as a replacement for a recent and more powerful device.

From the applications in Section 5 we used *collatz*, *cryptomining*, *random-testing*, and *raytracer*. We left out *arxiv* because it performs no computation and *batch-processing* of photos because the Beaker browser we rely on is not supported on mobiles and requires an explicit user interaction to save the resulting image making it harder to be consistent between experiments. All applications are compute-bound, as is typical of volunteer computing, and we evaluate their average throughput over a six minute execution.

We selected a diverse set of devices from our own personal collection and that of our friends. For phones, we evaluated the iPhone 4S (2 cores 1.0 Ghz ARM 32-bit), released in 2011, the Samsung Galaxy S4 (4 cores 1.9 Ghz ARMv7), released in 2013, and the iPhone SE (2 cores 1.85 Ghz ARMv8 64-bit),

released in 2016. For laptops, we evaluated a Macbook Air mid-2011 (2 cores i7 1.8 Ghz x86 64-bit), the Novena [12], a linux laptop based on a Freescale iMX6 CPU (4 cores 1.2 Ghz ARMv7 32-bit) produced in a small batch in 2015, an Asus Windows laptop based on a Pentium N3540 (4 cores 2.16 Ghz x86 64-bit) processor, and a Macbook Pro 2016 (4 cores i5 2.9 Ghz x86 64-bit).

For browsers, we used the native ones on the phones (Safari and Samsung) and Firefox (63.0.1 on x86 and 60.3.0 ESR on ARM) on laptops for consistency and because it is the fastest on numerical benchmarks [60].

We used Pando version 0.14.0 [32] with the version of application examples in Pando’s handbook [31] at commit 6e8d0335.

We first compare the performance of the MacBook Air mid-2011, the main work computer of the first author, to a single core of other devices to determine which are beneficial to accelerate tasks, with the results listed in Table 2. For phones, the performance of an iPhone 4S and a Samsung Galaxy S4 was too small to be significant; we therefore left them out of further tests. The Novena and Asus laptops performed closer to the Macbook Air, and when using 4 cores would approach or surpass that of a single core on the Macbook Air. The iPhone SE had similar or better performance than the Macbook Air in 3 applications. The Macbook Pro had almost a factor of 2 better performance. We therefore used those last four devices in two combinations to determine the benefits of using them with Pando.

Using these devices together with Pando brings performance benefits. We have chosen the first set of devices to have similar performance levels and span three different operating systems and two different instruction sets: we combined 1 worker on each of the Macbook Air and iPhone SE, to leave one core for Pando’s master on the Macbook Air and avoid pauses in computation introduced by iOS when a page is in the background, with 4 workers on each of the Novena and Asus laptops. This resulted in a 2.88-3.80 speedup, about twice faster than using two workers on the Macbook Air (not shown). For the second set of devices, we added the Macbook Pro with 4 workers. This approximately doubled the performance making the performance of the first set of devices about the same as that of the Macbook Pro.

Using Pando with multiple and diverse devices therefore definitely provides a benefit over only using the cores on the Macbook Air. Moreover, the performance obtained with the Novena, Asus, Macbook Air, and iPhone SE together also shows that these devices can be used in combination as an alternative to the Macbook Pro for a similar level of performance.

Moreover, Table 2 also yields a few interesting illustrations of the evolution of computing performance over multiple generations of devices and trends for the future. First, even if the Macbook Air and Macbook Pro have been released five years apart, the latter is only about twice faster on a single

	Collatz <i>BigNum/s</i>	Ratio	Crypto- Mining <i>Hashes/s</i>	Ratio	Random- Testing <i>Tests/s</i>	Ratio	Raytracer <i>Frames/s</i>	Ratio
<b>Single Worker</b>								
(1) iPhone 4S	15.0	0.07x	6,078	0.09x	36	0.08x	0.04	0.03x
(2) Samsung Galaxy S4	18.4	0.09x	12,615	0.18x	61	0.13x	0.11	0.07x
(3) Novena Laptop (Linux ARM)	34.3	0.16x	9,867	0.14x	72	0.15x	0,22	0.13x
(4) Asus Laptop (Windows x86)	85.5	0.41x	26,275	0.38x	214	0.45x	0.68	0.41x
(5) <i>Macbook Air mid-2011</i>	<i>210.2</i>	<i>1.00x</i>	<i>69,180</i>	<i>1.00x</i>	<i>475</i>	<i>1.00x</i>	<i>1,65</i>	<i>1.00x</i>
(6) iPhone SE	319.1	1.52x	55,246	0.80x	499	1.05x	1.72	1.04x
(7) Macbook Pro 2016	404.8	1.93x	129,933	1.88x	788	1.66x	2.84	1.72x
<b>Multiple Workers</b>								
4x(3) + 4x(4) + 1x(5) + 1x(6)	798.0	3.80x	202,606	2.93x	1438	3.03x	4.76	2.88x
4x(3) + 4x(4) + 1x(5) + 1x(6) + 4x(7)	1636.2	7.78x	418,344	6.05x	3274	6.90x	8.80	5.33x

**Table 2.** Average throughput with Pando’s master running on a Macbook Air 2011 and workers on various devices.

core, providing an example of the slow scaling of single-core performance in the last years. And second, the performance difference between the iPhone 4S and the Macbook Air, both released in 2011, is larger than that of the iPhone SE and the Macbook Pro, both released in 2016, which now are within a factor of two of each other. It therefore seems we can expect smartphones in the next years to have performance increasingly close to our work devices, which could make them increasingly interesting for accelerating distributed computations.

## 7 Related Work

As far as we know, Pando is the first tool explicitly designed for the purpose of *personal* volunteer computing. In this section we provide more detail on the *declarative concurrency* work it was inspired from and other systems that share similar technology choices. While Pando shares some technology choices with previous platforms, *it combines them for different aims*. Pando therefore represents a novel volunteer computing tool for a personal technology environment in which individuals and their social network own multiple increasingly powerful personal devices

### 7.1 Declarative Concurrency

Declarative concurrency has been studied in the context of dataflow programming, with languages such as Lucid [104] and Oz [96]. In the Oz language, the declarative programming model can be used directly to implement concurrent modules [101, Chapter 4]; it is based on using single-assignment variables that enable multiple threads to implicitly synchronize on the availability of data, on top of which higher-level abstractions such as streams can be built. The declarative concurrency paradigm has also been experienced by a large number of programmers and researchers through the popular MapReduce [44] framework and Unix pipeline programming [64]. In effect, Pando implements the *map* operation of

MapReduce; the other filtering and reduction phases can be performed locally, if necessary, by chaining with other Unix tools, such as `grep` and `awk` for example.

JavaScript, as many other mainstream programming languages, has not yet integrated features that make declarative concurrency widely accessible and easy, with good declarative concurrency primitives. We therefore instead based our design and implementation on the pull-stream design pattern (Section 3.4.2).

As far as we know, we are the first to develop and document systematic abstractions for volunteer computing using the declarative concurrent paradigm.

### 7.2 Stream Processing

*Stream processing* has been widely adopted as a programming model for *scalable distributed stream processing* [41], for general purpose programming on CPUs [58], for distributed GPU programming [108], and for Web-based peer-to-peer computing based on the WebRTC [17], WebSockets [7], and ZeroMQ [11] protocols. Those platforms are programmed using *dataflow graphs of computation* that combine multiple operators and complex data flows. They then ensure an efficient and reliable execution on different targeted execution environments. This level of expressivity is not necessary for many personal projects and applications (Section 5). To support our applications with a lower level of implementation complexity and make our design easier to port to other programming environments, Pando therefore concentrates on distributing the computation that is applied in a single stage of the streaming pipeline with the *map* operation. Everything else is performed locally by leveraging other tools.

### 7.3 Browser-Based Volunteer Computing

Fabisiak and al. [50] have surveyed more than 45 different browser-based volunteer computing systems developed over more than two decades. They grouped the publications in

three generations, that followed the evolution of Web technologies: the first generation [29, 37, 43, 53, 83, 92] was based on Java applets; the second generation [38, 39, 68, 77] used JavaScript instead but was somewhat limited by its performance; and the third generation [45, 47, 71, 73, 74, 76, 87, 91] fully emerged once performance issues were solved in multiple ways: JavaScript became competitive with C [65], WebWorkers [5], that did not interrupt the main thread, were introduced, and new technologies, such as WebCL [8], were proposed to increase the performance beyond what is possible on a single thread of execution on the CPU.

We further sub-divide Fabisiak and al.'s third generation into an explicit fourth [70, 72] that incorporates the latest communication technologies, such as WebSocket [7] and WebRTC [17], because they make fault-tolerance easier. Pando could be grouped with the fourth generation of systems and, as far as we know, is the first to leverage WebRTC for the explicit goal of volunteer computing. However, the key difference of Pando is in our focus on the *personal* aspects of volunteer computing (Section 2) that led to specific design principles (DPs of Section 3.2) with the following concrete impacts on its programming model, deployment strategy, and implementation choices.

Of the systems that have generic *programming models*, many focus on *batch-processing* [39, 45, 68–70, 87] as typically happens in high-profile long-running applications, sometimes reusing, in the browser, the MapReduce programming model that has been successful in data centers [38, 57, 71, 76, 91]. In contrast, by using a streaming model, Pando enables different and more personal applications by supporting *infinite streams* and *feedback loops*. This simplifies the combination of Pando with existing Unix tools and other programming environments (DP5).

While some general purpose projects aim to *deploy* new *global platforms* [28, 29, 37, 43, 45, 69, 71, 83, 86, 92], sometimes on *clouds* [72, 87], we have chosen to prioritize local deployments for personal uses. Pando also supports cloud platforms, if necessary for connectivity, but our common use cases do not require them. Moreover, by having a deployment that is *specific* to a single user and project (DP1), the implementation is simplified. That removes the need for solutions such as: (1) access restrictions in the form of *random URLs* to segregate the computations of different concurrent users [86], (2) *brokers/dispatchers/bridges* to organize the tasks submitted [28, 29, 43, 45, 69, 71], (3) dynamic management of a set of *managers* [37], and (4) *advocates* [92] to represent clients in the server.

Many *implementations* are organized around a database [38, 39, 42, 45, 69, 87, 91]. Pando's implementation instead encapsulates concurrency aspects in the StreamLender abstraction, removing the need for a database library. Other implementations are organized around a request-response API based on HTTP [38, 42, 45, 47, 68, 69, 71, 74, 77, 87, 91], to distribute inputs and collect results. Instead, and similar to newer

projects [70, 72], Pando communicates through WebRTC and WebSocket. In our case, the heartbeat mechanism of both protocols enabled our design to encapsulate the fault-tolerance strategy within StreamLender. These simplifications in turn hopefully makes it more likely that other programmers will adapt the design for embedding in other applications or to reimplement as standalone tools for different programming environments.

### 7.3.1 Peer-to-Peer Computing

*Peer-to-peer computing*, in which participating devices provide resources and help coordinate the services that are used, has a rich literature [27, 59, 66, 67, 75, 82, 88, 89, 97, 99, 107]. However, the server-centric model of Web technologies has historically limited the development of peer-to-peer Web platforms and applications. The recent introduction of WebRTC [17] removed that limitation which led to the creation of many new ones [46, 61, 62, 84, 100, 103, 105].

Of all previously mentioned systems, the closest to Pando is *browserCloud.js* [46] in its aim to provide a computation platform powered by the devices of participants. However, Pando's implementation approach is quite different and simpler because a deployment is restricted to a single client, its overlay organization need not make workers communicate with one another, it does not require maintenance when not in use for specific tasks, and removes the need for a discovery algorithm by instead relying on existing social media platforms. In our view, these differences come from a difference in application context. Using BrowserCloud.js's approach, and that of other peer-to-peer systems, is better to create *globally-shared self-sustaining platforms*. Ours is better to quickly obtain a working *personal tool* when a dependency on other tools and platforms is acceptable.

## 8 Conclusion

In this paper, we introduced the *personal volunteer computing* approach and derived, from the limitations of other approaches to parallel and distributed computing in a personal context, new design principles. We then introduced, along those principles, the design of Pando, a new and first tool for personal volunteer computing that enables a dynamically varying number of failure-prone personal devices contributed by volunteers to parallelize the application of a function on a stream of values using the devices' browsers. In doing so, we have explained how the declarative concurrent model made its programming simple and how the pull-stream design pattern was used to decompose its implementation in reusable modules. We then provided more detail about the properties and implementation of the new StreamLender abstraction that performs the core coordination work within Pando, which, by virtue of being independent of particular communication protocols or input-output libraries, should be easy to reimplement in many other programming

environments by following a complete description available as supplemental material. We followed with a presentation of a wide variety of novel applications organized along different dataflow patterns that showed Pando was useful on a wide number of existing and emerging use cases. We completed with an evaluation of Pando’s benefits in a real-world setting and showed throughput speedups on the previous applications using commonly available devices. The entire paper made the case that Pando, both as a tool and a reference design, should be a useful addition to the parallel toolbox of a multitude of users. In the coming years, *personal volunteer computing* could finally help scientists and programmers of the general public, for many new personal applications, more effectively tap into the increasing ubiquity of powerful mobile devices world-wide.

## References

- [1] 1981. Transmission Control Protocol. <https://www.ietf.org/rfc/rfc793.txt>. (1981). [Online; accessed 16-October-2018].
- [2] 2003. The Base16, Base32, and Base64 Data Encodings. <https://tools.ietf.org/html/rfc3548>. (2003). [Online; accessed 16-October-2018].
- [3] 2008. The Chromium Projects. <https://www.chromium.org/>. (2008). [Online; accessed 12-October-2018].
- [4] 2010. Open Stack. (2010). <https://www.openstack.org/> [Online; accessed 9-November-2018].
- [5] 2010. Web Workers. <https://w3c.github.io/workers/>. (2010). [Online; accessed 26-October-2018].
- [6] 2011. Open Compute Project. (2011). <http://www.opencompute.org/> [Online; accessed 9-November-2018].
- [7] 2011. The WebSocket Protocol. <https://tools.ietf.org/html/rfc6455>. (2011). [Online; accessed 13-February-2017].
- [8] 2011. WebCL: Heterogeneous parallel computing in HTML5 web browsers. <https://www.khronos.org/webcl/>. (2011). [Online; accessed 26-October-2018].
- [9] 2012. Apache Hadoop. [hadoop.apache.org](http://hadoop.apache.org). (2012). [Online; accessed 17-April-2017].
- [10] 2013. WebTorrent. <https://webtorrent.io/>. (2013). [Online; accessed 17-April-2017].
- [11] 2014. ZeroMQ. <http://zeromq.org/>. (2014). [Online; accessed 12-October-2018].
- [12] 2015. Novena Main Page. [https://www.kosagi.com/w/index.php?title=Novena\\_Main\\_Page](https://www.kosagi.com/w/index.php?title=Novena_Main_Page). (2015). [Online; accessed 9-November-2018].
- [13] 2017. Browserify. <http://browserify.org/>. (2017). [Online; accessed 15-April-2017].
- [14] 2017. Matjuice Repository. <https://github.com/sable/matjuice>. (2017). [Online; accessed 13-February-2017].
- [15] 2017. Pull-Stream Module List. <https://pull-stream.github.io/>. (2017). [Online; accessed 25-July-2017].
- [16] 2017. The Collatz Conjecture. <http://boinc.thesonntags.com/collatz/>. (2017). [Online; accessed 14-February-2017].
- [17] 2017. WebRTC 1.0: Real-time Communication Between Browsers. <https://www.w3.org/TR/webrtc/>. (2017). [Online; accessed 05-April-2017].
- [18] 2018. Beaker Browser. (2018). <https://beakerbrowser.com/>
- [19] 2018. The Dat project. (2018). <https://datproject.org/>
- [20] 2018. Deadline Compute Management System. <https://deadline.thinkboxsoftware.com/>. (2018). [Online; accessed 16-October-2018].
- [21] 2018. Gartner Says Worldwide Sales of Smartphones Recorded First Ever Decline During the Fourth Quarter of 2017. <https://www.gartner.com/en/newsroom/>. (2018). [Online; accessed 9-November-2018].
- [22] 2018. Heroku. (2018). <https://www.heroku.com/> [Online; accessed 17-November-2017].
- [23] 2018. Node Package Manager. (2018). <https://www.npmjs.com/> [Online; accessed 17-November-2017].
- [24] 2018. Raspberry Pi. (2018). <https://www.raspberrypi.org/>
- [25] 2018. Standard ECMA-262. <https://www.ecma-international.org/publications/standards/Ecma-262.htm>. (2018). [Online; accessed 16-October-2018].
- [26] 2018. Zinc Render. <https://www.zyncrender.com/>. (2018). [Online; accessed 16-October-2018].
- [27] Nabil Abdennadher and Regis Boesch. 2005. Towards a Peer-to-Peer Platform for High Performance Computing. In *Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*. IEEE, 8–pp. DOI : <http://dx.doi.org/10.1109/HPCASIA.2005.98>
- [28] Leila Abidi, Christophe Cérin, Gilles Fedak, and Haiwu He. 2015. Towards an Environment for doing Data Science that runs in Browsers. In *Proceedings of the International Conference on Smart City/Social-Com/SustainCom (SmartCity)*. IEEE, 662–667. DOI : <http://dx.doi.org/10.1109/SmartCity.2015.145>
- [29] Albert D Alexandrov, Maximilian Ibel, Klaus E Schauer, and Chris J Scheiman. 1997. SuperWeb: Towards a global web-based parallel computing infrastructure. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*. IEEE, 100–106.
- [30] David P. Anderson. 2010. Volunteer Computing: The Ultimate Cloud. *Crossroads* 16, 3 (March 2010), 7–10. DOI : <http://dx.doi.org/10.1145/1734160.1734164>
- [31] Anonymous. 2017. Pando Handbook. (2017). Link\_removed. [Online; accessed 17-November-2017].
- [32] Anonymous. 2017. Pando Repository. (2017). Link\_removed [Online; accessed 17-November-2017].
- [33] Anonymous. 2017. Pando Server. (2017). Link\_removed [Online; accessed 17-November-2017].
- [34] Anonymous. 2017. Pull-LendStream Implementation. Link\_removed. (2017). [Online; accessed 15-April-2017].
- [35] Anonymous. 2018. Pull-Stubborn Implementation. Link\_removed. (2018). [Online; accessed 28-October-2018].
- [36] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, and others. 2009. *Above the clouds: A Berkeley view of cloud computing*. Technical Report. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley.
- [37] Arash Baratloo, Mehmet Karaul, Zvi M Kedem, and Peter Wijkoff. 1999. Charlotte: Metacomputing on the Web. *Future Generation Computer Systems* 15, 5 (1999), 559 – 570. DOI : [http://dx.doi.org/10.1016/S0167-739X\(99\)00009-6](http://dx.doi.org/10.1016/S0167-739X(99)00009-6)
- [38] Kevin Berry. 2009. Distributed and Grid Computing via the Browser. In *Proceedings of the 3rd Villanova University Undergraduate Computer Science Research Symposium (CSRS 2009)*.
- [39] Fabio Boldrin, Chiara Taddia, and Gianluca Mazzini. 2007. Distributed computing through web browser. In *Vehicular Technology Conference, 2007. VTC-2007 Fall. 2007 IEEE 66th*. IEEE, 2020–2024.
- [40] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems* 25, 6 (2009), 599–616. DOI : <http://dx.doi.org/10.1016/j.future.2008.12.001>
- [41] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B Zdonik. 2003. Scalable Distributed Stream Processing.. In *CIDR*, Vol. 3. 257–268.
- [42] Pawel Chorazyk, Aleksander Byrski, Kamil Pietak, Marek Kisiel-Dorohinicki, and Wojciech Turek. 2017. Volunteer computing in a

- scalable lightweight web-based environment. *Computer Assisted Methods in Engineering and Science* 24, 1 (2017), 17–40.
- [43] Bernd O. Christiansen, Peter Cappello, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauer, and Daniel Wu. 1997. Javelin: Internet-based parallel computing using Java. *Concurrency: Practice and Experience* 9, 11 (1997), 1139–1160. DOI: [http://dx.doi.org/10.1002/\(SICI\)1096-9128\(199711\)9:11<1139::AID-CPE349>3.0.CO;2-K](http://dx.doi.org/10.1002/(SICI)1096-9128(199711)9:11<1139::AID-CPE349>3.0.CO;2-K)
- [44] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
- [45] Roman Dębski, Tomasz Krupa, and Przemysław Majewski. 2013. ComcuteJS: A web browser based platform for large-scale computations. *Computer Science* 14 (2013).
- [46] David Dias and Luís Veiga. 2018. BrowserCloud.js - A federated community cloud served by a P2P overlay network on top of the web platform. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18)*. ACM, New York, NY, USA, 2175–2184. DOI: <http://dx.doi.org/10.1145/3167132.3167366>
- [47] Jerzy Duda and Wojciech Dłubacz. 2012. Distributed Evolutionary Computing System Based on Web Browsers with JavaScript. In *International Workshop on Applied Parallel Computing*. Springer, 183–191.
- [48] Muhammad Nouman Durrani and Jawwad A Shamsi. 2014. Volunteer computing: requirements, challenges, and solutions. *Journal of Network and Computer Applications* 39 (2014), 369–380. DOI: <http://dx.doi.org/10.1016/j.jnca.2013.07.006>
- [49] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (April 1988), 288–323. DOI: <http://dx.doi.org/10.1145/42282.42283>
- [50] Tomasz Fabisiak and Arkadiusz Danilecki. 2017. Browser-based harnessing of voluntary computational power. *Foundations of Computing and Decision Sciences* 42, 1 (2017), 3–42. DOI: <http://dx.doi.org/10.1515/fcds-2017-0001>
- [51] Gilles Fedak. 2015. *Contributions to Desktop Grid Computing*. Habilitation à diriger des recherches. Ecole Normale Supérieure de Lyon. <https://hal.inria.fr/tel-01158462>
- [52] Gilles Fedak, Cécile Germain, Vincent Neri, and Franck Cappello. 2001. XtremWeb: A Generic Global Computing System. In *First IEEE/ACM International Symposium on Cluster Computing and the Grid (CC-GRD'2001)*. IEEE, 582–587. DOI: <http://dx.doi.org/10.1109/CCGRID.2001.923246>
- [53] David Finkel, Craig E Wills, Brian Brennan, and Chris Brennan. 1999. Dtriblets: Java-based distributed computing on the Web. *Internet Research* 9, 1 (1999), 35–40.
- [54] Vincent Foley-Bourgon and Laurie Hendren. 2016. Efficiently Implementing the Copy Semantics of MATLAB’s Arrays in JavaScript. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS 2016)*. ACM, New York, NY, USA, 72–83. DOI: <http://dx.doi.org/10.1145/2989225.2989235>
- [55] Ian Foster, Carl Kesselman, and Steven Tuecke. 2001. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *The International Journal of High Performance Computing Applications* 15, 3 (2001), 200–222. DOI: <http://dx.doi.org/10.1177/109434200101500302>
- [56] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. 2008. Cloud Computing and Grid Computing 360-degree Compared. In *Grid Computing Environments Workshop (GCE'08)*. IEEE, 1–10. DOI: <http://dx.doi.org/10.1109/GCE.2008.4738445>
- [57] Ilya Grigorik. 2009. Collaborative Map-Reduce in the Browser. <https://www.igvita.com/2009/03/03/collaborative-map-reduce-in-the-browser/>. (2009). [Online; accessed 16-October-2018].
- [58] Jayanth Gummaraju and Mendel Rosenblum. 2005. Stream Programming on General-Purpose Processors. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)*. IEEE Computer Society, Washington, DC, USA, 343–354. DOI: <http://dx.doi.org/10.1109/MICRO.2005.32>
- [59] Andrew B Harrison. 2008. *Peer-to-grid computing: Spanning diverse service-oriented architectures*. Ph.D. Dissertation. Cardiff University (United Kingdom). <https://search.proquest.com/openview/69ebafc7df184c92a437b66ee04345ee/>
- [60] David Herrera, Hanfeng Chen, Erick Lavoie, and Laurie Hendren. 2018. Numerical Computing on the Web: Benchmarking for the Future. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2018)*. ACM, New York, NY, USA, 88–100. DOI: <http://dx.doi.org/10.1145/3276945.3276968>
- [61] Yonghao Hu, Zhaohui Chen, Xiaojun Liu, Fei Huang, and Jinyuan Jia. 2017. WebTorrent Based Fine-grained P2P Transmission of Large-scale WebVR Indoor Scenes. In *Proceedings of the 22nd International Conference on 3D Web Technology (Web3D '17)*. ACM, New York, NY, USA, Article 7, 8 pages. DOI: <http://dx.doi.org/10.1145/3055624.3075944>
- [62] Alan B. Johnston and Daniel C. Burnett. 2012. *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web*. Digital Codex LLC, USA.
- [63] Mehmet Karaul. 1998. *Metacomputing and resource allocation on the world wide web*. Ph.D. Dissertation.
- [64] Brian W. Kernighan and Rob Pike. 1983. *The UNIX Programming Environment*. Prentice Hall Professional Technical Reference.
- [65] Faiz Khan, Vincent Foley-Bourgon, Sujay Kathrotia, Erick Lavoie, and Laurie Hendren. 2014. 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, 91–102. DOI: <http://dx.doi.org/10.1145/2661088.2661090>
- [66] Jik-Soo Kim. 2009. *Decentralized and scalable resource management for desktop grids*. Ph.D. Dissertation. University of Maryland, College Park. <http://hdl.handle.net/1903/9259>
- [67] Jik-Soo Kim, Beomseok Nam, and Alan Sussman. 2014. Scalable and effective peer-to-peer desktop grid system. *Cluster Computing* 17, 4 (2014), 1185–1201. DOI: <http://dx.doi.org/10.1007/s10586-014-0390-z>
- [68] Jon Klein and Lee Spector. 2007. Unwitting distributed genetic programming via asynchronous JavaScript and XML. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, 1628–1635.
- [69] Fumikazu Konishi, Manabu Ishii, Shingo Ohki, Ryo UESTU, and Akihiko Konagaya. 2007. RABC: A conceptual design of pervasive infrastructure for browser computing based on AJAX technologies. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*. IEEE, 661–672.
- [70] Makoto Kuhara, Noriki Amano, Kan Watanabe, Yasuyuki Nogami, and Masaru Fukushi. 2014. A peer-to-peer communication function among Web browsers for Web-based Volunteer Computing. In *Communications and Information Technologies (ISCIT), 2014 14th International Symposium on*. IEEE, 383–387.
- [71] Philipp Langhans, Christoph Wieser, and François Bry. 2013. Crowdsourcing MapReduce: JSMAPReduce. In *Proceedings of the 22nd International Conference on World Wide Web*. ACM, 253–256.
- [72] Guillaume Leclerc, Joshua E. Auerbach, Giovanni Iacca, and Dario Floreano. 2016. The Seamless Peer and Cloud Evolution Framework. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016 (GECCO '16)*. ACM, New York, NY, USA, 821–828. DOI: <http://dx.doi.org/10.1145/2908812.2908886>
- [73] Tommy MacWilliam and Cris Cecka. 2013. CrowdCL: Web-based volunteer computing with WebCL. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. IEEE, 1–6.
- [74] Gonzalo J Martinez and Leonardo Val. 2015. Capataz: a framework for distributing algorithms via the World Wide Web. *CLEI Electronic Journal* 18, 02 (2015), 2. [http://www.scielo.edu.uy/scielo.php?script=sci\\_arttext&pid=S0717-50002015000200002&lng=en&nrm=iso](http://www.scielo.edu.uy/scielo.php?script=sci_arttext&pid=S0717-50002015000200002&lng=en&nrm=iso)
- [75] Petar Maymounkov and David Mazieres. 2002. Kademia: A peer-to-peer information system based on the xor metric. In *International*

- Workshop on Peer-to-Peer Systems*. Springer, 53–65.
- [76] Edward Meeds, Remco Hendriks, Said Al Faraby, Magiel Bruntink, and Max Welling. 2015. MLitB: machine learning in the browser. *PeerJ Computer Science* 1, e11 (2015). DOI: <http://dx.doi.org/10.7717/peerj-cs.11>
- [77] Juan Julián Merelo-Guervós, Pedro A Castillo, Juan Luis Jiménez Laredo, A Mora Garcia, and Alberto Prieto. 2008. Asynchronous distributed genetic algorithms with Javascript and JSON. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence)*. IEEE Congress on. IEEE, 1372–1379.
- [78] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.* 47, 4, Article 69 (July 2015), 35 pages. DOI: <http://dx.doi.org/10.1145/2788396>
- [79] John P Morrison, James J Kennedy, and David A Power. 2001. Webcom: A Web Based Volunteer Computer. *The Journal of Supercomputing* 18, 1 (2001), 47–61. DOI: <http://dx.doi.org/10.1023/A:1008163024500>
- [80] Ujan Mukhopadhyay, Anthony Skjellum, Oluwakemi Hambolu, Jon Oakley, Lu Yu, and Richard Brooks. 2016. A brief survey of cryptocurrency systems. In *Privacy, Security and Trust (PST), 2016 14th Annual Conference on*. IEEE, 745–752.
- [81] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).
- [82] Sagnik Nandy. 2005. *Large scale autonomous computing systems*. Ph.D. Dissertation. UC San Diego. <https://escholarship.org/uc/item/3s96x9qc>
- [83] Noam Nisan, Shmulik London, Oded Regev, and Noam Camiel. 1998. Globally distributed computation over the internet—the popcorn project. In *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on*. IEEE, 592–601.
- [84] J. K. Nurminen, A. J. R. Meyn, E. Jalonen, Y. Raivio, and R. Garc asa Marrero. 2013. P2P media streaming with HTML5 and WebRTC. In *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 63–64. DOI: <http://dx.doi.org/10.1109/INFCOMW.2013.6970739>
- [85] Yao Pan. 2017. *Gray Computing: A Framework for Distributed Computing with Web Browsers*. Ph.D. Dissertation. Vanderbilt University. <http://etd.library.vanderbilt.edu/available/etd-11192017-220210/>
- [86] Sean R. Wilkinson and Jonas S. Almeida. 2014. QMachine: commodity supercomputing in web browsers. *BMC Bioinformatics* (2014), 1–1. DOI: <http://dx.doi.org/10.1186/1471-2105-15-176>
- [87] Cushing Reginald, Ganeshwara Putra, Spiros Koulouzis, Adam Beloum, Marian Bubak, and Cees de Laat. 2013. Distributed Computing on an Ensemble of Browsers. *IEEE Internet Computing* 17, 5 (Sept. 2013), 54–61. DOI: <http://dx.doi.org/10.1109/MIC.2013.3>
- [88] Andrew Rosen. 2016. *Towards a Framework for DHT Distributed Computing*. Ph.D. Dissertation. Georgia State University. [https://scholarworks.gsu.edu/cs\\_diss/107](https://scholarworks.gsu.edu/cs_diss/107)
- [89] Antony Rowstron and Peter Druschel. 2001. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 329–350.
- [90] David P Roy, MA Wulder, Thomas R Loveland, CE Woodcock, RG Allen, MC Anderson, D Helder, JR Irons, DM Johnson, R Kennedy, and others. 2014. Landsat-8: Science and product vision for terrestrial global change research. *Remote sensing of Environment* 145 (2014), 154–172.
- [91] Sandy Ryza and Tom Wall. 2010. MRJS: A JavaScript MapReduce Framework for Web Browsers. (2010). <http://www.cs.brown.edu/courses/csci2950-u/f11/papers/mrjs.pdf>
- [92] Luis FG Sarmenta and Satoshi Hirano. 1999. Bayanihan: Building and Studying Web-Based Volunteer Computing Systems using Java. *Future Generation Computer Systems* 15, 5-6 (1999), 675–686.
- [93] Luis Francisco Gumar Sarmenta. 2001. *Volunteer computing*. Ph.D. Dissertation. Massachusetts Institute of Technology. <http://hdl.handle.net/1721.1/16773>
- [94] Ernst Friedrich Schumacher. 2011. *Small is beautiful: A study of economics as if people mattered*. Random House.
- [95] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *Internet of Things Journal* 3, 5 (2016), 637–646. DOI: <http://dx.doi.org/10.1109/JIOT.2016.2579198>
- [96] Gert Smolka. 1995. The Oz programming model. In *Computer science today*. Springer, 324–343.
- [97] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review* 31, 4 (2001), 149–160.
- [98] Dominic Tarr. 2016. Pull Streams. <http://dominictarr.com/post/145135293917/history-of-streams>. (2016). [Online; accessed 7-February-2017].
- [99] Niklas Therning and Lars Bengtsson. 2005. Jalapeno: Decentralized Grid Computing Using Peer-to-peer Technology. In *Proceedings of the 2Nd Conference on Computing Frontiers (CF '05)*. ACM, New York, NY, USA, 59–65. DOI: <http://dx.doi.org/10.1145/1062261.1062274>
- [100] N. Tindall and A. Harwood. 2015. Peer-to-peer between browsers: cyclon protocol over WebRTC. In *2015 IEEE International Conference on Peer-to-Peer Computing (P2P)*. 1–5. DOI: <http://dx.doi.org/10.1109/P2P.2015.7328517>
- [101] Peter Van-Roy and Seif Haridi. 2004. *Concepts, Techniques, and Models of Computer Programming*. MIT Press.
- [102] Neelanarayanan Venkataraman. 2016. A Survey on Desktop Grid Systems-Research Gap. In *Proceedings of the 3rd International Symposium on Big Data and Cloud Computing Challenges (ISBCC)*, V. Vijayakumar and V. Neelanarayanan (Eds.), Vol. 49. Springer International Publishing, 183–212. DOI: [http://dx.doi.org/10.1007/978-3-319-30348-2\\_16](http://dx.doi.org/10.1007/978-3-319-30348-2_16)
- [103] C. Vogt, M. J. Werner, and T. C. Schmidt. 2013. Leveraging WebRTC for P2P content distribution in web browsers. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*. 1–2. DOI: <http://dx.doi.org/10.1109/ICNP.2013.6733637>
- [104] William W Wadge and Edward A Ashcroft. 1985. *LUCID, the dataflow programming language*. Vol. 303. Academic Press London.
- [105] M. J. Werner, C. Vogt, and T. C. Schmidt. 2014. Let Our Browsers Socialize: Building User-Centric Content Communities on WebRTC. In *2014 IEEE 34th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 37–44. DOI: <http://dx.doi.org/10.1109/ICDCSW.2014.35>
- [106] Turner Whitted. 1980. An Improved Illumination Model for Shaded Display. *Commun. ACM* 23, 6 (June 1980), 343–349. DOI: <http://dx.doi.org/10.1145/358876.358882>
- [107] Dany Wilson. 2015. *Architecture for a Fully Decentralized Peer-to-Peer Collaborative Computing Platform*. Ph.D. Dissertation. Université d’Ottawa/University of Ottawa. DOI: <http://dx.doi.org/10.20381/ruor-4170>
- [108] Shinichi Yamagiwa and Leonel Sousa. 2007. Design and implementation of a stream-based distributed computing platform using graphics processing units. In *Proceedings of the 4th international conference on Computing frontiers*. ACM, 197–204.