Personal Volunteer Computing

Erick Lavoie

Advisor: Prof. Laurie Hendren

School of Computer Science
McGill University

This dissertation is submitted for the degree of

Doctor of Philosophy

November 2019
I dedicate this thesis to everyone world-wide that is actively finding ways of living that increase the health of their natural and social environments. May the field of computing be part of and amplify those initiatives.
Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text, acknowledgements, and the following three papers, whose material has been expanded.

*Personal Volunteer Computing*, published at Computing Frontier 2019 and co-authored with Laurie Hendren, is a summarized version of the Background (Chapter 2), Volunteering Scenarios on a Local Wi-Fi Network (Section 6.2), and Future Research Directions (Section 8.1). For this paper, I articulated the socio-technical analysis framework, performed literature reviews, did the experiments, and generated new ideas for future directions. Laurie helped improve the flow of ideas, focus the argument on the essentials, suggested the mobile experiments, and revise the methodology and analysis to make sure it was sound.

*Pando: Personal Volunteer Computing in Browsers*, accepted at Middleware 2019, co-authored with Laurie Hendren, Frederic Desprez, and Miguel Correia, is a summarized version of the Design of Pando (Chapter 3), the Pull-Stream Design Pattern (Section 4.1), the StreamLender presentation (Section 4.4), the Applications (Chapter 5), and the Volunteering Scenarios on a Wide Area Network (Section 6.3). For this paper, I designed and implemented Pando in JavaScript, I articulated the core properties of the programming model, I suggested a notation to describe the algorithms, I described the existing Pull-Stream Design pattern and its properties, I implemented all applications (except for the first version of Raytracer that was done by Umair Khan), and I performed and described all experiments on the Grid5000 and PlanetLab testbeds. Laurie helped improve the clarity of descriptions, the flow of the text, the use of examples, suggested categories of applications to implement, and helped structure the argument presentations over the various revisions of this paper. Fred and Miguel suggested references to existing work in distributed systems and helped improve the flow and quality of writings.

*Genet: A Quickly Scalable Fat-Tree Overlay for Personal Volunteer Computing using WebRTC*, published at SASO 2019 and co-authored with Laurie Hendren, Frederic Desprez,
and Miguel Correia, is a summarized version of Genet: Quickly Scalable Fat-Tree Overlay for WebRTC (Chapter 7). For this paper, I designed the overlay, implemented it in JavaScript for Pando, implemented the simulation tool for assessing the balance of trees as well as the webrtc-connectivity-testing tool for testing the connectivity of WebRTC between groups of participants, I performed all experiments, generated figures, did a first analysis of the results, and performed a literature review of existing work. Fred and Miguel suggested related works and Laurie, Fred, and Miguel helped revise the writings.

Erick Lavoie
November 2019
Acknowledgements

First and foremost, thanks to Quebec and Canadian tax payers who funded most of my research, either through scholarships, grants, or subsidized education fees. I would never have had the financial means to do it otherwise and this funding provided me with much needed free time to articulate the ideas in this dissertation. It is my hope that those ideas will provide direct and indirect benefits that are at least commensurate with the resources that supported my education and research training. Thanks also to the various funding bodies, such as the National Science and Engineering Research Council (NSERC) of Canada, and Fond de Recherche du Quebec – Nature et Technologies (FRQNT) for managing those funds so they can reach researchers and maintain a lively and world-class intellectual environment. While my funding came from those organisms, any opinions, findings, and conclusions or recommendations expressed in this dissertation do not necessarily reflect the views of NSERC or FRQNT. In a similar spirit and to ensure the French government continues to fund common infrastructure for research, I acknowledge that some experiments presented in this paper were carried out using the Grid’5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

Thanks to Umair Khan for creating the initial version of the Raytracing application example described in Section 3.1 and Section 5.1.2. Umair emailed me in July 2017, after having just graduated from Delhi Technological University, wanting to contribute to some of the research projects we were doing at McGill. His feedback and thirst for learning have helped improve Pando, and his volunteering initiatives have fuelled my motivation to continue working on the project.

Thanks to the many members of the McLab team at McGill throughout the years. In particular, thanks to Faiz Khan, Sujay Kathrotia, and Vincent Foley-Bourgon for their collaboration on our first JavaScript benchmarking paper that established the performance feasibility of Pando. Thanks to Faiz additionally for properly setting me on the "dating scene", which set things in motions that eventually led me to meeting the love of my life. Thanks to David Herrera and Hanfeng Chen for our collaboration, and much of the groundwork, on the second JavaScript benchmarking paper that showed the performance opportunity on mobile
devices was even more interesting than we previously thought. And thanks to everyone else in the team over the years that had the patience to listen to my ideas (and rants!) in lab presentation and informal discussions, these interactions have been key to turning the seeds of ideas into sprouts.

Thanks also for all the supporting staff at McGill who manage the administrative affairs and helped ensure our rooms were clean. It is quite easy to forget that lively intellectual environments depend much also on well-functioning bureaucracies and well-tended physical spaces. Many of them have duly performed their work even through the challenging budget cuts that we collectively experienced in the mid 2010s.

Thanks to Miguel Pupo Correia in providing helpful and timely feedback on papers we co-authored over the years, as well as the welcoming environment when I visited Lisbon. While Pando’s design evolved in a way that eventually did not require the improved version of the Kademlia design we published together, the work we did helped me understand the drawbacks in complexity of global persistent peer-to-peer platforms that prompted the articulation of personal volunteer computing and the later designs.

Thanks to many colleagues and friends at INRIA in Grenoble. In particular, thanks to Frédéric Desprez, Imma Pressegger, and the rest of the CORSE team at INRIA for providing a welcoming environment, first in hosting me for an internship in the summer of 2016, and then providing a working space and company during much of the writing process of this dissertation. Thanks to Antoine Hokayem for having taught me to put the technical contributions and core ideas ahead of my personal agenda in research papers, I am convinced it has made a difference in getting the latest version of the Personal Volunteer Computing paper accepted. Thanks also to Fabian, François, George, Antoine and everyone else for passing by regularly in my office and pulling me for beers and social activities, this was a much needed relief. I hope we will all be done with our dissertations soon! And thanks to the NanoD and OneAngstrom teams on the first floor, for their participation in the smartphone experiments of this dissertation and their company in sharing meals, breaks, "publishing" cookies and cakes. Your presence has made my work environment much more lively.

Thanks to many people from the Secure Scuttlebutt (SSB) evolving community. Thanks to Dominic Tarr who shared his contagious curiosity, insights, and for quickly "pulling" me in the community by trusting me with administrator privileges on the pull-stream organization as soon as I had produced new useful modules. Thanks to Bob Haugen for revising a rejected version of the Personal Volunteer Computing paper, which has helped keep my motivation level high at times when I had trouble finding the right audience and adequately communicating the insights behind the approach. Thanks to Alexander Cobleigh, aka ‘cblgh’, for voluntarily revising and commenting on a draft of this dissertation: knowing someone
else than my advisor and review committee has read my thesis voluntarily made the entire effort worthwhile! Thanks to Daan, the only SSB friend I can regularly meet in Grenoble for beers and Lego. Thanks to the rest of the community, too numerous to mention, for fostering a "radical" intellectual environment that inspired the articulation of the personal volunteer computing paradigm.

Thanks to the Soulakins, an informal group of friends in Grenoble and elsewhere in Europe sharing an interest in computers (or in people that have an interest in computers!), for providing a lively social environment. In particular, thanks to Marty for sharing a desk in his research lab at University of Grenoble in my very first of many visits in Grenoble. Thanks to Guillaume, aka ’ponce’, for the co-working session we had together. And thanks to everyone that volunteered for early experiments in connectivity and computing capabilities, showing once more that friendship relationships can indeed provide motivation for volunteer computing! Thanks also for the many reminders that I had gone way overboard in taking over six years to complete this dissertation!

Thanks to my parents and family for encouraging me to follow my curiosity and interests, however radical and uncertain they may seem, and however far they may lead me from them. Thanks to my brother, Pier-Luc Lavoie, for his infectious laugh and good spirit, our meetings in person and remotely are always uplifting to me. Thanks to my mother, Juliette Robichaud, for teaching me to be sensitive to other people’s pains and difficulties, for caring for those less fortunate than I am, and for placing the wellness of others well ahead of financial gains. This sensibility has motivated me to find systemic solutions to many problems of today. Thanks to my father, Richard Lavoie, for nurturing my thirst for knowledge and critical thinking, for having encouraged me early in life to always compare myself against the very best of what I could be capable of, and for teaching me to think slowly, deeply, and methodically. These character traits have provided a great foundation for the difficult journey of a PhD training. Thanks also for providing financial support until I could sustain myself with internships and scholarships, I have only been able to tackle work with uncertain financial returns because I did not have debts to pay back.

Thanks to Laurie, my advisor, for her support throughout the years, especially when I was going through periods of doubts. I have been able to stick to this project for six years, face numerous paper rejections, learn the ropes of academic writing, and keep enough focus to constantly make progress because of her steady rhythm and availability in times of need. I witnessed Laurie’s resilience in the face of a cancer diagnosis, her positive outlook through even the most grueling treatments, and her unwavering determination to keep being available to help her mother, supervise students, continue to publish, obtain grants, and pioneer new applications to improve the quality of life of other cancer patients. The ordeal has left me
with a profound respect and admiration for her character. I am sad that she could not read the final result of this dissertation before passing away, but I am proud that the lessons I have learned from her have carried me through the final writing stages. When editing, I can still hear her voice asking me to clarify, being more precise, introduce examples, and improve the writing flow, right before occasional bursts of laughing. I hope I will carry this voice for many years to come.

Thanks to Bettina, my "interim" advisor, for going through the last writing and administrative steps with me. It is never easy to carry someone else’s research and supervision roles, much less while carrying Department Chair duties. I have really appreciated the proactiveness in ensuring that all McLab students will be able to complete their programs, and the prompt and effective help in helping me complete this dissertation and my PhD program.

Thanks to Vanessa, the love of my life, for appearing on a Contact Improvisation dance floor on the 2nd of August 2015 and inviting me into our first dance. Over the last four years this dance has evolved into a lifelong commitment that I hope will carry us through many decades to come. Thanks for your presence throughout the years, in listening to my frustrations, excitements, doubts, and joys. Thanks also for your patience and support in dealing with so many mundane tasks while I was completing this dissertation (and battling with procrastination!). Your company is a source of deep joy and has dampen the worst moments of anxiety that I had to go through. May I be there for you also when you need it.
Abstract

In this dissertation, we articulate the new personal volunteer computing paradigm, that refines volunteer computing by focusing on solutions that are personal to the user on multiple dimensions: they target personal projects, they leverage the participation of volunteers from their personal social network, and they are built into personal tools that can be deployed on personal devices and can be easily maintained by a single developer.

We then present Pando, a new and first tool for personal volunteer computing written in and for JavaScript, that distributes the application of a function on a stream of inputs into the browsers of participating devices. Pando dynamically scales to new devices, gracefully tolerates sudden disconnections, and is easy to program because it is based on a declarative concurrent programming paradigm, in which the non-determinism of parallel executions is not observable by users.

We follow with a more detailed presentation of the implementation of Pando, based on the new Limiter, StreamLender, and DistributedMap abstractions. Our presentation uses a high-level notation, independent of JavaScript and that simplifies reasoning about concurrent aspects, to introduce all the algorithms that implement the Limiter, StreamLender, and DistributedMap abstractions. Because the concurrent aspects make StreamLender challenging to implement, even with a clear description, we present a run-time verification approach to ensure it is correct. The approach is easy to parallelize, with Pando for example, and quickly generates a large number of random executions to ensure a high-probability of correctness. The combination of clear descriptions and testing strategy should make Pando easy to reimplement in other programming environments.

We then present a large scope of applications that we implemented for Pando, based on existing libraries and examples. These applications represent various dataflow patterns and show Pando can be used not only for compute-intensive tasks but also for crowd-processing.

We then measure the throughput performance of these applications in three networking scenarios: (1) over a local-area Wi-Fi network, with personal laptop and smartphones, (2) over a virtual private network distributed throughout France, with the Grid5000 nodes, and (3) with a wide-area network throughout Europe on the Internet, with the PlanetLab EU nodes. We show personal devices are competitive in all scenarios, sometimes with older
devices competing with newer models, and other times with combinations of personal devices outcompeting remote server nodes. The flexible and easy support of all these scenarios is, to the best of our knowledge, a first in the volunteer computing literature.

We then present *Genet*, a new fat-tree overlay for WebRTC that enables Pando to overcome the limits of WebRTC in the number of connections, and showed the resulting combination of Pando and Genet to be able to *scale* to a thousand browsers in 30-55 seconds on local networks. Those results are possible because the design of Genet only uses *local information* to *deterministically* route the WebRTC connection messages, while ensuring the resulting tree is *probabilistically balanced*.

We conclude by outlining new exciting research directions that take into account the *limits to growth* our society is currently facing. Compared to the current trends in research, these new directions make smaller whole-system designs done by small teams viable again, but bring a stronger focus on leveraging abundant personal computing devices and taking into account the growing importance of efficiently using renewable electricity sources.
Résumé

Dans cette thèse, nous développons le nouveau paradigme de calcul bénévole personnalisé. Ce paradigme spécialise le calcul bénévole en mettant l’accent sur des solutions personnalisées à l’utilisateur ou l’utilisatrice qui ciblent simultanément: des projets personnels, la participation de bénévoles issus de cercles sociaux personnels, ainsi que l’utilisation d’outils personnels pouvant être déployés sur des appareils électroniques personnels et pouvant être facilement entretenus par un seul développeur.

Nous présentons ensuite Pando, un nouveau et premier outil pour le calcul bénévole personnalisé programmé en et pour JavaScript, qui distribue l’application d’une fonction sur chacune des entrées d’un flot de données dans les navigateurs web des appareils participants. Pando s’adapte dynamiquement à la connexion de nouveaux appareils, gère de manière transparente leur déconnexion soudaine, et est facile à programmer parce qu’il est organisé autour d’un paradigme de programmation concurrente déclarative, dans lequel le non-déterminisme des exécutions parallèles n’est pas observable par l’utilisateur.

Nous continuons avec une présentation plus détaillée du fonctionnement interne de Pando, basé sur les nouvelles abstractions Limiter, StreamLender, et DistributedMap. Notre présentation utilise une notation de haut-niveau, distincte de JavaScript et en rendant plus aisé la réflexion sur les aspects concurrents, pour introduire tous les algorithmes qui implémentent ces abstractions. Puisque les aspects concurrents de StreamLender le rendent difficiles à implémenter, même avec une description claire, nous présentons également une stratégie de vérification à l’exécution qui permet de rapidement découvrir les erreurs. Notre approche est facile à paralléiser, avec Pando par exemple, et génère rapidement un grand nombre d’exécutions aléatoires pour assurer, avec une forte probabilité, que StreamLender est correctement implémenté. L’utilisation combinée de notre description des algorithmes et de notre stratégie de test devrait permettre de réimplémenter Pando aisément dans d’autres environnements de programmation.

Nous présentons ensuite un large éventail d’applications que nous avons implémentées avec Pando, elles-mêmes basées sur des librairies et des exemples existants. Ces applications représentent diverses classes de flots de données et montrent que Pando peut être utilisé.
non seulement pour des applications de calcul intensif mais également pour coordonner des tâches effectuées par des humains.

Nous mesurons ensuite le débit de calcul de ces applications dans trois scénarios de réseautique: (1) en utilisant un réseau local sans fil connectant des ordinateurs portables et des téléphones intelligents, (2) en utilisant un réseau privé virtuel distribué entre plusieurs centres de calcul en France, composé de noeuds appartenant au projet Grid5000, et (3) dans un réseau à grande échelle sur Internet, reliant des noeuds de calcul du projet PlanetLab EU distribués au travers de l’Europe. Par ces tests, nous montrons que les appareils personnels sont compétitifs dans tous les scénarios, dans certains cas en montrant que certains appareils de générations précédentes peuvent fournir une puissance de calcul similaire à des appareils plus récents, et dans d’autres cas en montrant que plusieurs appareils personnels utilisés ensemble peuvent être plus performants que des serveurs distants. Au meilleur de nos connaissances, le support aisé de ces trois scénarios dans un même outil est une première dans la littérature sur le calcul bénévole.

Nous présentons ensuite Genet, un nouveau réseau logique pour WebRTC organisé autour d’un arbre à branches épaisses, qui permet à Pando de repousser les limites du nombre de connexions concurrentes supportées par WebRTC. Nous montrons que l’intégration de Genet dans Pando permet la mise à l’échelle en connectant un millier de navigateurs web en 30-55 secondes. Ces résultats sont possibles parce que Genet n’utilise que de l’information disponible localement pour (1) acheminer de manière déterministe les messages de connexion de WebRTC et (2) s’assurer que l’arbre reste équilibré de manière probabiliste.

Nous terminons en esquissant de nouvelles directions de recherche prometteuses qui tiennent compte des limites à la croissance auxquelles fait face notre société. En comparaison aux directions actuelles, ces nouvelles directions rendent de nouveau viable la conception de systèmes complets par de petites équipes, tout en mettant davantage l’accent sur l’utilisation de l’abondance d’appareils électroniques personnels existants et en tenant compte de l’importance grandissante d’utiliser efficacement des sources renouvelables d’électricité pour les alimenter.
# Table of contents

List of figures .............................................. xix
List of tables .............................................. xxiii
Nomenclature .............................................. xxv

1 Introduction .............................................. 1
   1.1 Challenges ........................................... 2
   1.2 Contributions ....................................... 3
   1.3 Organization ......................................... 4

2 Background .............................................. 7
   2.1 Major Paradigms ...................................... 7
   2.1.1 Cloud Computing ................................... 8
   2.1.2 Grid Computing .................................... 9
   2.1.3 Volunteer Computing ................................ 10
   2.1.4 Personal Volunteer Computing ....................... 11
   2.2 Related Approaches .................................. 12
   2.2.1 Emerging Cloud Refinements ....................... 13
   2.2.2 Decentralized Approaches ......................... 13
   2.2.3 Browser-Based Volunteer Computing ................ 15
   2.3 History of Desktop Grid and Volunteer Computing .... 16
   2.3.1 Workstations and Desktop Grid Computing ......... 16
   2.3.2 Personal Computers and Volunteer Computing ....... 17
   2.3.3 The Opportunity of Mobile Personal Computing .... 18
   2.3.4 Further Readings .................................. 18
   2.4 Summary .............................................. 19
# Design of Pando

3.1 Usage Example: Parallel Rendering of a 3D Animation .................................. 22
   3.1.1 Programming Interface ...................................................... 23
   3.1.2 Deployment ................................................................. 25
   3.1.3 Benefits ................................................................. 28
3.2 Design Principles .............................................................. 29
3.3 Programming Model .......................................................... 30
   3.3.1 Core Programming Model: Streaming Map ......................... 31
   3.3.2 Distributed Properties .................................................. 32
   3.3.3 Performance Properties ............................................... 33
3.4 Implementation Overview ..................................................... 34
   3.4.1 Web Technologies .................................................... 34
   3.4.2 Declarative Concurrency with Pull-Streams ...................... 35
   3.4.3 Architecture ............................................................ 37
   3.4.4 Potential Support for Other Distributed Execution Models .... 38
3.5 Determining Applications for which Pando is Beneficial .......... 40
3.6 Related Work ................................................................. 41
   3.6.1 Main Concurrency Paradigms ........................................ 41
   3.6.2 Declarative Concurrency .............................................. 42
   3.6.3 Stream Processing .................................................... 43
   3.6.4 Browser-Based Volunteer-Computing Implementations .......... 44
   3.6.5 Peer-to-Peer Computing in Browsers ............................... 45
   3.6.6 Ongoing Related Open Source Projects ......................... 46
   3.6.7 Emerging Technologies .............................................. 48
3.7 Summary ................................................................. 48

# Implementation of Pando

4.1 Pull-Stream Design Pattern ............................................... 51
   4.1.1 Benefits ............................................................... 55
   4.1.2 Callback Protocol Invariants ...................................... 60
   4.1.3 Complexity in Implementing Concurrent Modules ............. 62
4.2 Declarative Concurrent Notation ........................................ 64
   4.2.1 Event Notation ....................................................... 65
   4.2.2 Wait Synchronization .............................................. 67
   4.2.3 Explicit Thread Creation .......................................... 69
   4.2.4 First-order Logic Operators with a Reified History .......... 69
   4.2.5 Pull-Stream Syntactic Sugar ..................................... 70
## Table of contents

4.3 Limiter .......................... 73  
4.3.1 Definition ........................ 73  
4.3.2 Implementation ................. 73  
4.4 StreamLender ............................ 78  
4.4.1 Definition ........................ 78  
4.4.2 Implementation ................. 81  
4.5 DistributedMap .......................... 87  
4.5.1 Definition ........................ 87  
4.5.2 Implementation ................. 87  
4.6 Run-time Verification of StreamLender with Random Test Generation ... 90  
4.6.1 High-Level Description .......... 90  
4.6.2 Inter-leavings ...................... 91  
4.6.3 Random Selection of Next Action . 92  
4.6.4 Benefits and Limitations .......... 96  
4.7 Discussion .......................... 97  
4.8 Related Work .......................... 97  
4.9 Summary ............................ 98  

5 Applications 101  
5.1 Pipeline Processing ....................... 101  
5.1.1 Collatz ................................ 102  
5.1.2 Raytrace ............................ 103  
5.1.3 Arxiv ................................ 103  
5.1.4 StreamLender Random Testing .... 103  
5.1.5 Machine Learning Agent .......... 104  
5.1.6 Image Processing with HTTP Data Distribution ............. 106  
5.2 Synchronous Parallel Search for Crypto-Currency Mining ............. 108  
5.3 Stubborn Processing with Failure-Prone External Data Distribution .... 109  
5.3.1 Image Processing with WebTorrent Data Distribution .......... 110  
5.3.2 Image Processing with DAT Data Distribution ............. 111  
5.4 Summary ............................ 112  

6 Evaluation 113  
6.1 Measurement Methodology .............. 113  
6.1.1 Measuring the Performance of Individual Devices ............ 114  
6.1.2 Aggregating Metrics from Multiple Devices in Real-Time .... 116  
6.1.3 Measurement Infrastructure ......... 117
6.2 Volunteering Scenarios on a Local Wi-Fi Network .................................................. 119
  6.2.1 Personal Devices .................................................................................................. 119
  6.2.2 Smartphones of Colleagues on a Local Wi-Fi Network .................................. 121
6.3 Volunteering Scenarios on a Wide Area Network ..................................................... 122
  6.3.1 Updated Personal Devices on a Local-Area Network ...................................... 123
  6.3.2 Devices on a Virtual Private Network across France (Grid5000) .................. 123
  6.3.3 Devices on a Wide Area Network across Europe (PlanetLab EU) .............. 124
  6.3.4 Performance Results ......................................................................................... 125
6.4 Summary .................................................................................................................. 128

7 Genet: Quickly Scalable Fat-Tree Overlay for WebRTC .............................................. 131
  7.1 Design .................................................................................................................... 132
    7.1.1 Genet’s Node Definition .................................................................................. 134
    7.1.2 Usage and differences with the JavaScript implementation. ....................... 136
    7.1.3 Genet’s Node Implementation ....................................................................... 138
  7.2 Integration with Pando .......................................................................................... 143
    7.2.1 Scalable DistributedMap Definition ................................................................ 143
    7.2.2 Scalable DistributedMap Implementation .................................................... 145
  7.3 Evaluation .............................................................................................................. 149
    7.3.1 Depth with Probabilistic Balancing ............................................................... 149
    7.3.2 Bootstrap Latency When Scaling on Local Networks .................................. 151
    7.3.3 Throughput Ramp-up in the Collatz Application ......................................... 155
    7.3.4 WebRTC Connection Probability and Latency on the Internet .................. 157
  7.4 Discussion .............................................................................................................. 160
  7.5 Related Work ......................................................................................................... 160
  7.6 Summary ................................................................................................................. 162

8 Conclusion .................................................................................................................. 165
  8.1 Future Research Directions within the Limits to Growth ....................................... 166
    8.1.1 Potential Applications ................................................................................... 168
    8.1.2 Design Principles ......................................................................................... 169
    8.1.3 Research Topics ............................................................................................ 170

References ...................................................................................................................... 173

Appendix A Test Modules ................................................................................................. 191

Appendix B Application Implementations ........................................................................ 197
List of figures

3.1 Rotation animation .................................................. 23
3.2 JavaScript Programming Interface Example. ......................... 24
3.3 Unix Programming Interface Example. .............................. 24
3.4 Installation Example. .............................................. 25
3.5 Deployment Example. ............................................... 25
3.6 Initial State. ...................................................... 25
3.7 A Tablet Joined. .................................................. 26
3.8 The Tablet Rendered the First Image. ................................ 26
3.9 A Phone Joined. .................................................. 27
3.10 The Phone Rendered the Third Image. .............................. 27
3.11 The Tablet Crashed.................................................. 27
3.12 The Phone Rendered the Second Image. ............................ 28
3.13 Streaming Map. .................................................. 31
3.14 Architecture of Pando. ........................................... 39

4.1 Pull-Stream Design Pattern. ....................................... 52
4.2 Pull-Stream Source and Sink Examples. ......................... 53
4.3 Pull-Stream Transformer Example. ................................ 54
4.4 Pull-Stream Duplex Example. .................................... 54
4.5 Pull-Stream Composition. ......................................... 55
4.6 Pull-Stream Source, Sink, and Transformer in Python. ........ 56
4.7 Flow Regulation (in JavaScript). .................................. 57
4.8 Early-Termination Example using the Source of Figure 4.2. .... 57
4.9 Graceful Error Handling. ........................................... 58
4.10 Lazy Generation. .................................................. 59
4.11 Declarative Composability of Source, Sink and Transformer from Figures 4.2
        and 4.3. .......................................................... 59
4.12 Declarative Concurrent ParallelMap Example ..................... 61
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.13</td>
<td>Pull-Stream Merger</td>
<td>62</td>
</tr>
<tr>
<td>4.14</td>
<td>Pull-Stream Merger Implementation</td>
<td>63</td>
</tr>
<tr>
<td>4.15</td>
<td>Implicit Event Indexes and Coupled Event Handlers in JavaScript</td>
<td>66</td>
</tr>
<tr>
<td>4.16</td>
<td>Wait Primitive Example in JavaScript</td>
<td>68</td>
</tr>
<tr>
<td>4.17</td>
<td>Thread Creation in JavaScript</td>
<td>70</td>
</tr>
<tr>
<td>4.18</td>
<td>Reacting to Past Events in JavaScript</td>
<td>71</td>
</tr>
<tr>
<td>4.19</td>
<td>Creating and Connecting Pull-Stream Modules in JavaScript</td>
<td>71</td>
</tr>
<tr>
<td>4.20</td>
<td>Limiter Usage in JavaScript</td>
<td>74</td>
</tr>
<tr>
<td>4.21</td>
<td>Limiter Inputs and Outputs</td>
<td>75</td>
</tr>
<tr>
<td>4.22</td>
<td>StreamLender Usage in JavaScript</td>
<td>80</td>
</tr>
<tr>
<td>4.23</td>
<td>StreamLender inputs and outputs</td>
<td>81</td>
</tr>
<tr>
<td>4.24</td>
<td>DistributedMap Usage in JavaScript</td>
<td>88</td>
</tr>
<tr>
<td>4.25</td>
<td>StreamLender Inter-leaving Example</td>
<td>92</td>
</tr>
<tr>
<td>4.26</td>
<td>Test Transformer State Machines</td>
<td>94</td>
</tr>
<tr>
<td>5.1</td>
<td>Pipeline Processing Dataflow and Examples</td>
<td>101</td>
</tr>
<tr>
<td>5.2</td>
<td>Collaborative Filtering of Arxiv Abstracts</td>
<td>104</td>
</tr>
<tr>
<td>5.3</td>
<td>Machine Learning Agent Training</td>
<td>105</td>
</tr>
<tr>
<td>5.4</td>
<td>Image Processing</td>
<td>107</td>
</tr>
<tr>
<td>5.5</td>
<td>Synchronous Parallel Search Dataflow and Example</td>
<td>109</td>
</tr>
<tr>
<td>5.6</td>
<td>Stubborn Processing with External Data Distribution Dataflow</td>
<td>110</td>
</tr>
<tr>
<td>6.1</td>
<td>Raytracing Example Augmented with Measurements</td>
<td>114</td>
</tr>
<tr>
<td>6.2</td>
<td>iPhone SE Real-time Measurement Screenshot</td>
<td>115</td>
</tr>
<tr>
<td>6.3</td>
<td>Sampling Mismatch Problem</td>
<td>116</td>
</tr>
<tr>
<td>6.4</td>
<td>Raytracing Invocation with Logs</td>
<td>118</td>
</tr>
<tr>
<td>6.5</td>
<td>Screenshots of Real-time Reports with a Macbook Air 2011 and an iPhone SE</td>
<td>118</td>
</tr>
<tr>
<td>6.6</td>
<td>Historical Throughput on Mobile Devices</td>
<td>122</td>
</tr>
<tr>
<td>7.1</td>
<td>Genet’s WebRTC Bootstrap Sequence</td>
<td>133</td>
</tr>
<tr>
<td>7.2</td>
<td>Genet Node Usage in JavaScript</td>
<td>137</td>
</tr>
<tr>
<td>7.3</td>
<td>Scalable Pando</td>
<td>144</td>
</tr>
<tr>
<td>7.4</td>
<td>DistributedMap Usage in JavaScript</td>
<td>145</td>
</tr>
<tr>
<td>7.5</td>
<td>Number of Experiments with X% or Less of Nodes at Depth $\log(N) + 1$ over 1000 Repetitions and No Failures</td>
<td>150</td>
</tr>
<tr>
<td>7.6</td>
<td>Number of Experiments with X% or Less of Nodes at Depth $\log(N) + 1$ over 1000 Repetitions and a Failure Probability of 0.25 for Nodes to Fail</td>
<td>151</td>
</tr>
<tr>
<td>7.7</td>
<td>WebRTC Connection Latency Distribution Using Local Server</td>
<td>153</td>
</tr>
</tbody>
</table>
List of figures

7.8 Latency to Connect All Participants in the Fat-Tree on the Grid5000’s Grenoble Site over 5 Experiments. .......................... 154
7.9 Average Throughput on the Collatz Application on Grid5000. .................. 156
7.10 Latency to Reach Maximum Throughput on the Collatz Application on Grid5000. .................................................. 157
7.11 WebRTC Connection Latency Distribution over 380 Successful Connections between Nodes on the Grid5000’s Grenoble Site using a Remote Server. 158
7.12 Geographical Location of Participants. .............................................. 159
7.13 WebRTC Connection Latency Distribution between World-Wide Participants. 159

B.1 Collatz JavaScript Implementation. .................................................. 197
B.2 Simplified Arxiv JavaScript Implementation. ....................................... 198
B.3 StreamLender Random Testing Main Processing Function. ................ 199
B.4 Simplified Implementation of the Machine Learning Agent Processing Function. .................................................. 200
B.5 Simplified Implementation of Image Processing with HTTP Data Distribution. 201
B.6 Implementation of Crypto-Currency Mining. ................................... 202
List of tables

2.1 Comparison between the Major Paradigms and Personal Volunteer Computing. 12
2.2 Comparison between Edge, Gray, and Personal Volunteer Computing. 13
2.3 Comparison between Decentralized Approaches and Others. 14
2.4 Comparison between Previous Browser-Based Volunteer Computing Tools and Personal Volunteer Computing. 16
2.5 Comparison between Desktop Grid, Volunteer Computing and Personal Volunteer Computing. 19

3.1 Summary of the Properties of Pando’s Programming Model. 34
3.2 Comparison between Pando’s Implementation and Related Systems. 50

4.1 Pull-Stream Callback Protocol Events. 52
4.2 Pull-Stream Callback Protocol Events Examples in our Event Notation using the Merger Example of Figure 4.13. 65
4.3 Truth Table for Possible Actions on Test Transformer’s Input Depending on Active State. 95
4.4 Truth Table for Possible Actions on Test Transformer’s Output Depending on Active State. 95
4.5 Summary of the Pull-Stream Design Pattern Benefits. 99

5.1 Summary of Applications using Pipeline Processing. 102
5.2 Synchronous Parallel Search Example: Crypto-currency Mining. 109
5.3 Stubborn Processing Examples. 110

6.1 Average Throughput using a Combination of Personal Devices on Collatz, Crypto-Mining, and Random-Testing. 120
6.2 Average Throughput using a Combination of Personal Devices on Raytrace, Image-Processing, and MAgent. 120
6.3 Average Throughput of Cell Phones Volunteered by Colleagues. 121
6.4 Average Throughput using a Combination of Devices on a LAN, VPN, and WAN on Collatz, Crypto-Mining, and Random-Testing .......................... 126
6.5 Average Throughput using a Combination of Devices on a LAN, VPN, and WAN on RayTrace, Image-Processing, and MLAgent ............................. 127
# Nomenclature

**Acronyms / Abbreviations**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D</td>
<td>Three Dimensional</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ARM</td>
<td>Advance RISC Machine, i.e. CPU Architecture</td>
</tr>
<tr>
<td>CAD</td>
<td>Computer Assisted Design</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DHT</td>
<td>Distributed Hash Table</td>
</tr>
<tr>
<td>DP</td>
<td>Design Principle</td>
</tr>
<tr>
<td>FLOPS</td>
<td>Floating-Point Operations Per Second</td>
</tr>
<tr>
<td>GET</td>
<td>HTTP Request to Retrieve a File</td>
</tr>
<tr>
<td>GIF</td>
<td>Graphic Interchange Format</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Processing Unit</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>ICE</td>
<td>Interactive Connectivity Establishment</td>
</tr>
<tr>
<td>IPFS</td>
<td>InterPlanetary File System</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Translation</td>
</tr>
<tr>
<td>NPM</td>
<td>Node Package Manager</td>
</tr>
<tr>
<td>P2P</td>
<td>Peer-to-Peer</td>
</tr>
<tr>
<td>POST</td>
<td>HTTP Request to Publish a File</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computing</td>
</tr>
<tr>
<td>RNG</td>
<td>Random Number Generator</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Private Network</td>
</tr>
<tr>
<td>WAN</td>
<td>Wide Area Network</td>
</tr>
<tr>
<td>WebRTC</td>
<td>Web Real-Time Communication Protocol</td>
</tr>
<tr>
<td>WI-FI</td>
<td>(WIFI) Wireless Networking Protocol</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The large environmental and social impact of producing the latest computing devices suggests we should better use those that already exist. At the same time, the large penetration and continued growth in consumption of personal electronic devices represents a largely untapped resource of computing power. Over the last 20 years, computing developments in industry have increasingly focused on building global computing platforms, such Amazon Cloud [7], for a narrow set of globally-shared needs. Technical, financial, and administrative barriers in them prevent a large part of the general programmer population, especially in humanities and developing countries, from using these resources for the larger set of 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 not significantly 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 have named personal volunteer computing.

The major challenges in designing personal volunteer computing tools for the general public are analogous to those that have prompted the articulation of appropriate technologies\(^1\) [78] in the 1970s for promoting economic development in developing countries. Then, the limited access to high-speed reliable infrastructure, trained specialists, capital and worldwide 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 and explain them in a way to

\(^1\)Originally named intermediate technologies [177].
foster local appropriation and replication in many programming environments. Among all problems that could correspond to the previous challenges, this dissertation focuses on using a multitude of personal devices in parallel for a single task composed of an ordered sequence of independent inputs, an instance of distributed stream processing [37].

The declarative concurrent programming paradigm [199, Chapter 4] greatly simplifies reasoning about concurrent processes: it abstracts the non-determinism arising from multiple parallel threads executing at the same time by making it non-observable to the programmer. This paradigm has already enjoyed great practical successes with the popular MapReduce [49] and Unix pipelining [96] programming models. Could it also be useful for making personal volunteer computing tools easy to program and easier to implement? The present dissertation answers positively by showing how to leverage the growing abundance of computing resources present in the personal devices of volunteers in a simple, portable, and scalable way through: (1) the design and implementation of Pando, a personal volunteer computing tool that executes in Web browsers, supports a wide variety of applications that represent some of the needs of scientists and programmers in general, and can be deployed in local as well as global networks of devices, (2) a clear and complete explanation of the Limiter, StreamLender, and DistributedMap abstractions and their implementation that form the core of Pando, to ease their reimplementation in other programming environments, and (3) the Genet Fat-Tree Overlay that increases the total number of devices that can be connected to Pando over WebRTC on local networks.

1.1 Challenges

Three main challenges are addressed in this dissertation to leverage personal devices for personal volunteer computing.

Simplicity. Currently popular personal devices, such as smartphones, tablets, laptops, and desktops, have a wide range of technical capabilities, hardware-software configurations, and programming environments creating a combinatorial explosion of possible programming approaches. Many personal projects also have computation time requirements in the order of a few hours or days: tools that aim to accelerate those tasks shall therefore be easy enough to learn and deploy that they provide a performance improvement even when taking into account their learning curve. It is therefore important to make all personal devices usable in parallel for many possible tasks with a design that requires a minimum of programming and deployment efforts.

Portability. Users may be forced to use specific programming environments because their application depends on particular programming libraries that are not available elsewhere,
which may be the case if they rely on *numpy* in Python or any of the MATLAB-specific libraries. In addition, the quick deprecation of software technologies, as early as 5-10 years after they have been introduced as happened in the past for Web technologies, severely limits the lifespan of tools that depend on their associated application programming interfaces (APIs). To make Pando’s design as useful as possible today and tomorrow, it is therefore important to simplify the task for *other programmers* to reimplement Pando in their favourite programming environment. This in turn requires a clear and complete description of the core abstractions of the design and methods to test the correctness of their implementation.

*Scalability.* Resource limitations of devices and programming environments restrict the maximum number of concurrent connections on personal devices, therefore limiting the performance gains that are possible by using them in parallel with direct connections. To maximize the benefits in deploying personal tools, a quickly scalable connection scheme that can leverage all devices of a personal social network is necessary.

### 1.2 Contributions

We contribute to the development of *personal* volunteer computing in six ways.

First, we articulate the new *personal* volunteer computing viewpoint within the current major distributed computing approaches. We analyze the socio-technical context in which these approaches have been developed and identify an under-explored research area at the intersection of the personal projects, devices, tools, and social networks dimensions. We provide it as Background material in Chapter 2.

Second, we present the high-level design and JavaScript implementation of *Pando*, a new and first tool for personal volunteer computing that enables a single function to be applied in parallel to all values of a stream on a dynamically varying number of failure-prone volunteer devices. Pando is easy to program because we based its design on the declarative concurrent programming model: a user therefore does not need to reason about concurrency issues. We explain our choices to make the tool *simple* to use and to deploy in Chapter 3.

Third, we present in more detail the *StreamLender, Limiter* and *DistributedMap* abstractions that implement Pando. *StreamLender* in particular encapsulates the key distributed properties of Pando, independent of networking protocols or input-output libraries. All abstractions are based on the pull-stream design pattern, a functional programming pattern organized around a callback protocol that originated in the JavaScript open source community. We present the pull-stream design pattern in detail, for the first time in an academic publication. We then motivate the need for a higher level notation than JavaScript to present concurrent abstractions based on the pull-stream design pattern. We then present the al-
gorithms that implement the abstractions in a declarative concurrent notation we designed by inspiration from the language Oz [182], which supports the declarative concurrent programming model natively. We also provide a random testing strategy for StreamLender to ensure its implementation is correct. This level of detail should make the algorithms easier to understand and puts a stronger emphasis on replicating our design and implementation in other programming environments, current and future, than is currently typical in academic descriptions of distributed system designs. We present all in Chapter 4.

Fourth, we present 8 implemented examples of using Pando, representing a wide variety of potential applications that may be beneficial to scientists and other users of the general public. We also discuss how to quickly determine if Pando may be beneficial in some other applications. We present both in Chapter 5.

Fifth, we present a measurement methodology and infrastructure to quickly assess the performance of personal devices and improve the throughput of applications. We then show the throughput improvements provided by Pando in three volunteering scenarios: a local area network made of personal devices, a virtual private network distributed across France made of Grid5000 [17] nodes, and a wide-area network distributed across Europe made of Planet Lab EU [159] nodes. We present all in Chapter 6.

Sixth, we present the novel Genet Fat-Tree Overlay to quickly scale the performance obtainable with Pando with the number of available devices on local networks, which can scale up to a thousand browsers in 30-55s, enough to quickly leverage all machines in university departments or large organizations. We present the extended design of Pando with the Genet Overlay, as well as an empirical evaluation of its benefits on the Grid5000 testbed in Chapter 7.

1.3 Organization

The remainder of this dissertation is organized as follows. The personal volunteer computing paradigm is introduced by surveying major approaches to distributed computing in the next chapter. The high-level design of Pando is then presented in Chapter 3. The pull-stream design pattern, as well as the StreamLender and other pull-streams abstractions that compose Pando are presented in Chapter 4. Our implemented examples are given in Chapter 5. An evaluation of the performance of Pando in various scenarios is presented in Chapter 6. The Genet Fat-Tree Overlay that extends the design of Pando is presented in Chapter 7. We finally conclude and suggest possible directions for future work in Chapter 8.

This thesis makes contributions to 3 areas of knowledge: Chapter 3, 4, and 7 contribute a new design and implementation of a personal volunteer computing tool, Chapter 4 presents
new abstractions in the form of pull-stream modules, and Chapter 7 contributes to the design and implementation of quickly scalable distributed overlay networks. Since each of these knowledge areas are associated with different sub-fields of computer science, we have provided the related work at the end of the corresponding chapters.
Chapter 2

Background

Past distributed computing approaches have successfully addressed the most pressing needs of industry and research groups world-wide. However, comparatively less attention has been given to the needs of individuals of the general programmer population, for their personal projects, and using computing resources already freely available in their environment.

In this chapter, we position personal volunteer computing in the context of other past approaches to clearly establish its originality. We first identify the major approaches as of 2019 according to the number of devices they manage: cloud, grid, and volunteer computing (Sections 2.1). We present the dimensions along which they differ, forming separate paradigms of distributed computing. We summarize the findings in Table 2.1.

We also cover other approaches closely related to personal volunteer computing to better highlight their respective specificity: emerging cloud refinements, decentralized approaches and browser-based volunteer computing (Section 2.2). We summarize the findings in Table 2.2, 2.3, and 2.4.

We continue with a history of the development of desktop grid and volunteer computing approaches to highlight the interplay between their research developments and the larger evolution of the personal computing industry (Section 2.3). This suggests a clear trend towards a growing amount of computing capability available to individuals; yet we do not have corresponding established approaches to leverage them to their full potential.

We finally conclude this chapter by summarizing the original aspects of personal volunteer computing (Section 2.4) that can leverage these new opportunities.

2.1 Major Paradigms

Cloud, grid, and volunteer computing are major approaches of distributed computing that strongly differ in the socio-technical context of their development paradigm, including which
users they target, who provides the computing resources, who funds their development and operations, and how. The paradigms around which they are organized influence the number of computing devices available and the kinds of challenges that have to be addressed. We present each in turn and then introduce personal volunteer computing by contrast.

2.1.1 Cloud Computing

Cloud computing [14, 34] has emerged ten years ago as a market service that offers their customers on-demand computing resources with no initial capital investment and quick scalability to match variations in resource usage. For many businesses, it offers (1) lower capital risks associated with over- or under-provisioning their hardware infrastructure to match the demand on their services and (2) enables economies of scale by sharing the same hardware resources among multiple users, therefore increasing resource utilization. Clouds accelerated the growth of startups into global platforms, AirBnB [5] and Uber [195] being notable examples.

The devices that power a cloud are provided by a single company. The development and the management of the platform is funded by customers using the cloud services directly, by renting the computing resources, or indirectly, by using online services that are implemented with them. Most often, cloud providers receive funding from both cases: Google provides ad-supported search services and the AppEngine [71] platform; Amazon provides an online marketplace and the Amazon Elastic Cloud 2 (EC2) [7] platform. The exact number of devices cloud providers manage is considered a trade secret by some. Nonetheless, we estimate cloud providers may collectively manage in the order of millions of devices.

The operating costs that have a direct impact on the profitability of operating a cloud (ex: hardware acquisition, hardware and software management, power and cooling energy requirements) incentivize their efficient usage. Consequently, researchers develop strategies to build cloud infrastructure using commodity hardware, minimize resource consumption for given workloads, multiplex many concurrently running services on the same hardware to amortize the fixed costs of operation, and develop autonomic management strategies to minimize the involvement of humans. Companies also increasingly share designs both for hardware with the Open Compute Project [154] and software with Open Stack [155] to lower the development and maintenance costs. Additional challenges include the quality of service provided (ex: latency in provisioning resources, total amount of available computing power) and the accurate and efficient monitoring of resource usage for billing to ensure a customer only pays for what they use.

In their canonical form, clouds are limited in three ways. First, their billing infrastructure becomes a financial barrier for individuals and organizations that do not have access to
financial instruments, such as a bank account or a credit card. Second, their reliance on centrally-managed dedicated hardware incurs fixed costs that require a minimum price that may be inaccessible to many individuals and organizations. Third, their API requires access permissions which complicates their programming, in turn creating a higher technical barrier.

2.1.2 Grid Computing

Grid computing [66, 67] is an older but similar offering to cloud computing that makes computing resources belonging to different collaborating organizations available through a unified service. Grid computing has been named by analogy to the way the electric grid was initially built. As grid computing developed, it was anticipated to also exist as a commercial offering, but was replaced by cloud computing. The grid approach survives today as a scientific utility by providing computing resources to publicly funded organizations such as universities and research centres.

Grids are currently funded through the public spendings of governments and offered to researchers both to carry their research and train students in distributed computing. We estimate the potential number of devices available through public organizations to be in the order of millions, although specific grid projects have much lower offerings. Grid5000 [17] and PlanetLab [41] each currently boast offerings of about a thousand devices.

The main challenge with grids is to create technical infrastructure that interoperates with the various distinct administrative domains and organization policies that manage the computing resources while providing a unified interface to researchers.

Challenges in building grids are different than for clouds. There is no need for a billing infrastructure because in most cases both the users and computing infrastructure are paid from public spendings. Also, the focus is on collaboratively sharing the infrastructure between researchers rather than maximizing resource utilization because some research projects, such as performance studies, require a reserved access to the devices.

However, while grids are available to many public organizations, they are not available to the general public. Even for researchers, the administrative complexity in obtaining the necessary permissions may rule out many small-scale projects because the resulting gain is not worth the effort. Both issues raise administrative barriers.
2.1.3 Volunteer Computing

Volunteer computing [176, 11] leverages the personal devices of volunteers from the general public to perform computations. It is organized around a commons paradigm, both digital\(^1\), by sharing tools between many independent research teams, and physical, by enabling volunteers to contribute their computing resources to many projects. Volunteers do not receive financial benefits for their contributions but may receive public recognition in the form of computation points. Computations performed with volunteer computing sometimes represent global issues, such as climate prediction [43] or drug discovery for cancer [63]. Other times they simply capture the imagination of the general public, such as the search for extra-terrestrial intelligence (SETI@Home) [179].

The development of volunteer computing tools has been funded by governments through public research grants to provide researchers with supercomputing capacities at a much lower cost. It has the potential to leverage billions of personal devices although at the moment the current number of participating devices are in the order of a million. At the time of writing, its flagship project, BOINC [29], with 175,000 active volunteers managing 858,000 active computers, and a total combined computing power of 22.579 PetaFlops is one of the top five most powerful supercomputers in the world and has a fifth of the power of the most powerful one (Sunway TaihuLight), which boasts between 93 and 125 PetaFlops [191].

The major typical challenges to the volunteer computing approach concern the variability of capabilities of personal devices, the necessity to encourage and maintain volunteer engagement, and the automatic handling of volunteer’s unreliability [12, 148].

In contrast to cloud computing, an additional computation contribution incurs negligible costs to the researchers, therefore efficiently using the hardware is less of an issue. Also, financial transactions are replaced with computation points that are issued after successful work has been performed, which lowers the financial barrier to obtaining access to the computing devices. In contrast to grid computing, there are fewer administrative barriers to deploy the tools: a research team may buy its own server and use the tools freely to request support from the general public. Its major advantage for researchers, compared to both clouds and grids, is that the majority of the costs are supported by volunteers, which covers the acquisition, the operation, and the maintenance of the computing devices. However, it typically has a higher communication latency and more limited bandwidth available, making

\(^{1}\)"The digital commons are defined as information and knowledge resources that are collectively created and owned or shared between or among a community and that tend to be non-excludable, that is, be (generally freely) available to third parties. Thus, they are oriented to favor use and reuse, rather than to exchange as a commodity. Additionally, the community of people building them can intervene in the governing of their interaction processes and of their shared resources." (Fuster [68])
it better applicable to a lesser number of compute-intensive tasks with low communication requirements.

With 4 billion estimated Internet users [87], the current number of volunteers represents less than 0.005% of humanity. From a technical perspective, the number of smartphones that were sold in 2015 and 2016 is more than 3000 times the number of active computers in volunteer computing projects [151]. The approach has therefore not yet reached its full potential. We believe the complexity of the BIONC tools, that have been designed for researchers and large-scale projects, as well as the costs in acquiring and maintaining dedicated servers to run them are remaining technical and financial barriers that slows the widespread adoption of the paradigm.

2.1.4 Personal Volunteer Computing

*Personal* volunteer computing is a new volunteer computing approach we are proposing, that also follows the *commons* paradigm but focuses on the personal computation needs of programmers from the *general public*.

Its users may be, for example, scientists in research teams with low but significant computation needs, or individuals in developing countries with a personal smartphone and no access to other alternatives. The approach leverages the *trust* in its user’s personal social network, composed of *friends, family, and colleagues*, which lowers the possibility that volunteers would intentionally provide invalid results. Moreover, volunteers have an incentive to participate because of existing and implicit social relationships of reciprocity. So far, our work has been funded from research grants from governments but its potentially wide applicability could encourage the *general public to directly fund it for its own needs*. This approach has the potential to better leverage the *billions* of available personal devices [151] because users can obtain large amounts of computing power at low or no cost.

Personal volunteer computing also has challenges of its own: the wide diversity of programming environments and software/hardware combinations to support may require significant development efforts, the smaller running times of personal projects means that users need to be able to learn and deploy the tools quickly to benefit from the accelerated application, and there is no widespread existing funding model to sustain its growth. It is therefore significantly more important than for other approaches that the tools remain *simple* to use and to deploy to provide quick gains with low efforts. It is also important that the tools can be quickly *reimplemented* in many environments, current and future with clear and simple designs. Finally, the tools also need to *scale* to all the devices of the personal social network of its user to maximize their benefits.
Compared to cloud and grid computing, personal volunteer computing removes their financial and administrative barriers. Compared to volunteer computing, it drastically lowers its technical barriers and removes its financial barriers. Moreover, personal volunteer computing is the only approach that leverages the implicit social relationships of reciprocity to incentivize participation.

Similar to volunteer computing, personal volunteer computing is currently mostly applicable to compute-intensive tasks. Even so, personal volunteer computing has the opportunity to support a wider range of tasks because the communication latency between personal devices on a local network is significantly lower than that between remote devices connected over a wide area network, which volunteer computing tools typically target. The full extent of compatible applications is an open question.

<table>
<thead>
<tr>
<th></th>
<th>Cloud</th>
<th>Grid</th>
<th>Volunteer</th>
<th>Personal Volunteer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Paradigm</strong></td>
<td>Market service</td>
<td>Scientific utility</td>
<td>Commons</td>
<td>Commons</td>
</tr>
<tr>
<td><strong>Res. providers</strong></td>
<td>Single company</td>
<td>Multiple org.</td>
<td>General public</td>
<td>Friends,family</td>
</tr>
<tr>
<td><strong>Target users</strong></td>
<td>Customers</td>
<td>Researchers</td>
<td>Researchers</td>
<td>General public</td>
</tr>
<tr>
<td><strong>Funders</strong></td>
<td>Customers</td>
<td>Governments</td>
<td>Governments</td>
<td>General public?</td>
</tr>
<tr>
<td><strong>Nb of devices</strong></td>
<td>Millions</td>
<td>Millions</td>
<td>Billions</td>
<td>Billions</td>
</tr>
<tr>
<td><strong>Challenges</strong></td>
<td>Efficiency, monitoring, billing</td>
<td>Interoperability, unified interface, sharing</td>
<td>Variability, engagement, unreliability</td>
<td>Simplicity, portability, scalability</td>
</tr>
</tbody>
</table>

Table 2.1 Comparison between the Major Paradigms and Personal Volunteer Computing.

### 2.2 Related Approaches

Other approaches are related to personal volunteer computing by leveraging the increasing number and capabilities of personal devices but are different on other dimensions. Emerging edge and gray computing are refinements of the cloud approach, which we cover in the next section, decentralized computing is compatible with all the previous major paradigms, which we cover in Section 2.2.2, and browser-based volunteer computing is a specific refinement of volunteer computing using Web technologies, which we cover in Section 2.2.3.
2.2 Related Approaches

2.2.1 Emerging Cloud Refinements

The emerging edge and gray computing approaches are quite different in their motivation and trusted parties, which warrant a quick sketch to highlight the differences.

Both edge and gray computing are based on a cloud-hosted platform. Edge computing [180] performs computation tasks on the devices that directly interface with the real world, such as mobile phones and sensor networks. Gray computing [156, 157] does the same in web browsers by offloading tasks to visitors of websites. In both cases, the motivation is to provide better quality of service with lower latency and to lower the operation costs of cloud-hosted platforms.

However, when platform operators move computations to user devices they are actually shifting some of their costs to their users for the same service. Users also have to trust the operators that they will respect the privacy of their data.

Instead, in personal volunteer computing, the intention is to increase the computing power the general public can use for new applications. The direct exchange of data between participating devices provides better privacy guarantees by only requiring mutual trust between chosen friends and family members. For example, this could be particularly useful for applications that manipulate personal health data. We summarize the differences in Table 2.2.

<table>
<thead>
<tr>
<th></th>
<th>Edge &amp; Gray (Cloud)</th>
<th>Personal Volunteer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Motivation</strong></td>
<td>Lower latency, lower hosting costs</td>
<td>Increased capabilities (general public)</td>
</tr>
<tr>
<td><strong>Trusted Parties</strong></td>
<td>Platform operator</td>
<td>Friends, family</td>
</tr>
</tbody>
</table>

Table 2.2 Comparison between Edge, Gray, and Personal Volunteer Computing.

2.2.2 Decentralized Approaches

More established decentralized and peer-to-peer (P2P) approaches use personal devices both for the execution of tasks and their coordination. This distributes the usual responsibilities of servers in a central location to all the participating devices in the network using peer-to-peer algorithms. In turn, it makes the system more resilient to the failures of coordinators by eliminating their privileged position.

Decentralized approaches are not a different paradigm but a different implementation approach to tolerate additional failure modes and spread the loads differently. They are
Currently used to implement decentralized storage [88] and computation [83] through market services supported by crypto-currencies. They are also used to exchange files in commons directly between users [28].

Compared to cloud, grid, and volunteer computing, decentralized approaches remove the required trust from the operators of the platform/tool and the servers used. However, they usually still provide a globally shared platform and accordingly maintain global structured overlays [189, 1, 143, 77, 98, 99, 211, 171, 51] with corresponding maintenance and complexity challenges. When applied to volunteer computing, maintaining the platform while it is not actively used puts pressure on volunteers to keep it running. This costs energy, time, and attention, but provides no clear benefit. Moreover, the complexity in developing and maintaining such platforms requires dedicated specialists and ongoing recurrent resources.

Compared to decentralized approaches, volunteer computing tools are centralized in the sense that the coordination of computations is performed on a single device: volunteer computing uses a dedicated server while personal volunteer computing uses one of the user’s devices. However, contrary to decentralized platforms, different users create disjoint networks. This greatly simplifies the implementation of coordinators while providing independence from the failures of other users.

In contrast to decentralized approaches, personal volunteer computing leverages the existing mutual trust between friends and family. To recruit volunteers, existing social platforms are used instead of maintaining separate decentralized services. Both choices greatly reduce the complexity of the infrastructure needed so that the tool can be maintained by a single developer in their spare time. We summarize the differences in Table 2.3.

<table>
<thead>
<tr>
<th></th>
<th>Decentralized/P2P</th>
<th>Cloud/Grid</th>
<th>Volunteer</th>
<th>Personal Volunteer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design</strong></td>
<td>Global platform</td>
<td>Global Platform</td>
<td>Tool</td>
<td>Tool</td>
</tr>
<tr>
<td><strong>Trusted Parties</strong></td>
<td>P2P algorithms</td>
<td>Platform op.</td>
<td>Tool op.</td>
<td>Friends,family</td>
</tr>
<tr>
<td><strong>Coordination</strong></td>
<td>Distributed</td>
<td>Centralized</td>
<td>Centralized (disjoint)</td>
<td>Centralized (disjoint)</td>
</tr>
<tr>
<td><strong>Coordinator(s)</strong></td>
<td>All devices</td>
<td>Dedicated servers</td>
<td>Dedicated server</td>
<td>User device</td>
</tr>
</tbody>
</table>

Table 2.3 Comparison between Decentralized Approaches and Others.
2.2 Related Approaches

2.2.3 Browser-Based Volunteer Computing

Today, the Web and its browsers have become the de facto common development platform with a large number of users and developers as well as strong support from the biggest computing companies. Browser-based volunteer computing is a refinement of volunteer computing that uses Web browsers for performing computations on volunteer computers, with the main benefits of abstracting the heterogeneity of personal devices, providing a secure sandbox environment and increasingly close performance to native code [97, 81]. Different researchers have surveyed more than 40 articles published over the last two decades on browser-based volunteer computing and published their findings in 2016 [200], and 2017 [56]. From these surveys we noticed two trends.

First, the technologies available in the majority of Web browsers have a lifespan of about 5-10 years: the Java-based Web technologies used by the first platforms [39, 18, 95, 175, 138, 60] have been abandoned and replaced with JavaScript-based Web standards in the latest projects [128, 129, 133, 174, 73, 134, 133, 183, 2]. The first tools are abandoned because their dependencies are no longer maintained. Although a significant effort is put on backward compatibility in recent Web standards, the increasing complexity of the Web stack, which has an impact on the resource usage and energy footprint of devices, may eventually create an opportunity for clean-slate simpler stacks either developed within proprietary ecosystems or through public standards. There is no guarantee that these newer development platforms will maintain backward compatibility with the current Web. We could therefore see history repeat itself with the more recent published systems.

Second, as far as we know, the published systems have always been seen as an extension of the current paradigms but have failed to be used by high-profile projects. High-profile projects have large and long-lasting computing needs that are harder to reconcile with the more ephemeral lifetime of Web applications. They also often use specific languages and libraries that would require significant efforts to port to the Web.

To address both issues, personal volunteer computing targets new applications natively built for the Web, new users from the general public, with more modest but still significant computation needs, and aims for simplicity to enable the tools to be easily and quickly reimplemented in the ever-changing programming landscape. We advocate for a stronger focus on the design and key algorithms of volunteer computing tools, rather than on the tools themselves, and rely on other developers to port them to all applicable programming environments. We summarize the findings in Table 2.4.
Previous Browser-Based Volunteer | Personal Volunteer
---|---
Target Users | Researchers | General public
Goals | Feasibility, performance | Simplicity, portability
Contingence | Specific web technologies | Developers

Table 2.4 Comparison between Previous Browser-Based Volunteer Computing Tools and Personal Volunteer Computing.

2.3 History of Desktop Grid and Volunteer Computing

In this section, we survey the literature surrounding volunteer computing and its closely related cousin, desktop grid computing, to provide additional detail. While both approaches aim to use idle computing cycles, they differ in the resources they use. For both approaches, we present landmark publications and present their differences, focusing on the wider context of evolution of the computing industry over time to explain what enabled those approaches to appear and flourish. We then consider the new possibilities of personal mobile computing and notice a lack of corresponding approach(es) to fully leverage its possibilities, which further motivates the personal volunteer computing approach from a historical perspective.

2.3.1 Workstations and Desktop Grid Computing

*Desktop grid computing* is an extension of the grid computing paradigm to everyday desktop computers and workstations, and has a rich literature that spans more than two decades of research [58, 59]. Its inspiration can be traced back in 1982, to worm programs [181] that would spread to available workstations, in 1987 to early work on profiling workstation resources availability [141], and in 1988 to the Condor [123] system in which idle workstations were made available for computation with background tasks. Later systems include XtremWeb [60, 36] and its later incarnations to integrate it with other existing grid systems [79]. The goal of that line of research was to make the desktop grid as convenient to use as other grid or cloud computing infrastructure for *researchers*. There is a desktop grid federation website [86] that provides news on latest developments and an overview document of the approach for prospective organizations [45]. Recently, one of the main proponents of the approach, Gilles Fedak, has created the iEx.ec startup [83], to transfer the technology to emerging blockchain-based applications in a *decentralized cloud* organized around a market based on its own crypto-currency.
The desktop grid approach emerged in the mid 90s. Since the approach was conceived at the time most of the computers were workstations in various institutions, accordingly the approach was to "steal" the computing resources when these were idle. At the time, individuals could not volunteer their computers as in most cases they did not have administrative rights over them. The focus of the research was therefore to create various middlewares that would run in the background and make the computers available according to the policies of their owner organization.

2.3.2 Personal Computers and Volunteer Computing

It then took about a decade or so, leading to the end of the 1990s and beginning of 2000s, for the personal computer to become mainstream and be found in a majority of households. Individuals then started having ownership and administrative privileges over their machines and could use them for personal purposes. In addition, the spread of high-speed internet connections at home enabled those personal computers to be connected with fast links between one another and to distant servers. This created an opportunity for volunteer computing to develop. The BOINC project [10] seized the opportunity to empower a single researcher to tap in the resources owned by 100,000s of participants with a week of deployment work and an hour a week of maintenance. This was in contrast to the grid approach in which the management of the infrastructure is done by professionals and its access regulated. While the two approaches were different in the beginnings, over time there has been hybridization. Accordingly, the BOINC platform [29] may now be used for both volunteer and desktop grid computing.

Nonetheless, the literature surrounding volunteer computing comprises a number of interesting contributions in addition to Sarmenta’s thesis [176], which gave the approach its current name, the original BOINC paper [10], which alone boasts 2300 citations [42], and a later overview of volunteer computing from 2010 [11]. Nowadays, volunteer computing is still an active field of research with diverse contributions published in the last five years [53, 200, 56]. BOINC and related projects even have their own dedicated biannual conference, BOINC:FAST [30], with editions that have been held in 2013, 2015, 2017.

A search through ProQuest [162] for doctoral dissertations related to "volunteer computing" returns about 100 results. We were surprised to find only about 100 theses. Comparatively, a search through ProQuest for "cloud computing" returns over 2500 results over

---

2We queried for all English doctoral dissertations between 1990 and 2018 in computer science that mention "volunteer computing" as an entire expression.
the span of only 10 years\(^3\). A similar situation was also noticed in the introduction of Toth’s dissertation [192] 10 years ago, when comparing research in volunteer computing to grid computing. This suggests now, as it did then, that many research avenues within this paradigm are yet to be explored.

### 2.3.3 The Opportunity of Mobile Personal Computing

Compared to the end of the 90s and early 2000s, the ownership and usage of computers in 2019 are both more personal and social. Computing is more personal because individuals now own more devices collectively than organizations ever did: it is now common for people to own many devices such as desktops, laptops, tablets, phones, and smaller embedded devices such as a Raspberry Pi, directly or through their connected appliances. It is also common for people to use them both at and outside of work; some devices such as smartphones are even constantly used throughout the day. Computing is also more social because many mobile devices are used for personal communications that follow the trust and reciprocity relationships in informal social networks of people.

Other groups have noticed the opportunity offered by personal mobile computing. Tapparello et al. have produced a survey, up to 2015, of all volunteer computing systems targeted at mobile devices [184]. Closer to our aims, Pramanik et al. have studied the potential of using smartphones to provide supercomputer capabilities in India at a fraction of the cost of a regular supercomputer by leveraging the high-penetration rate of these devices compared to other kinds of computing devices [161]. However, none of the publications we have reviewed have suggested focusing on personal projects of programmers of the general public. Yet programmers of the general public comprise both the highest number of potential users and collectively own the greatest amount of computing power. Their collective computing needs may potentially dwarf any other applications\(^4\) and has therefore potential for significant societal impact. This opportunity is widely open for further academic research and personal volunteer computing may play a significant role in fulling the computing needs of users with resources they collectively own.

### 2.3.4 Further Readings

The curious reader may find a more detailed presentation of desktop grid and volunteer computing’s past and recent projects as well as a rich taxonomy to compare them in two

---

\(^3\)Similar to our previous search for volunteer computing, we queried for all English doctoral dissertations between 1990 and 2018 in computer science that mention “cloud computing” as an entire expression.

\(^4\)This corresponds to the concept of the long tail distribution popularized by Chris Anderson [9].
2.4 Summary

Compared to previously surveyed approaches, personal volunteer computing is original because it focuses on the combination of:

- **Personal projects.** Cloud computing exists to serve the business needs of their clients. Grid computing exists to serve research groups that work on long-lasting projects that are of interest to nations as a whole. Traditional volunteer computing appeals to researchers working on shared collective interests. Personal volunteer computing instead focuses on increasing the computing capabilities of individuals with typically smaller but still significant computing needs on new applications;

- **Personal volunteer network:** In traditional volunteer computing, anyone may participate, and while collectively most participants may generally be trusted, individual participants may not as they may try to game the system for fun or to obtain more computing credits than they actually deserve. In personal volunteer computing, friends, family, and colleagues contribute because they either care about the project or about the person doing it. The context of personal relationships between project initiator

<table>
<thead>
<tr>
<th></th>
<th>Desktop Grid</th>
<th>Volunteer</th>
<th>Personal Volunteer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Paradigm</strong></td>
<td>Scientific utility</td>
<td>Commons</td>
<td>Commons</td>
</tr>
<tr>
<td><strong>Res. providers</strong></td>
<td>Multiple org.</td>
<td>General public</td>
<td>Friends, family</td>
</tr>
<tr>
<td><strong>Target devices</strong></td>
<td>Workstations</td>
<td>Personal devices</td>
<td>Personal devices</td>
</tr>
<tr>
<td><strong>Target users</strong></td>
<td>Researchers</td>
<td>Researchers</td>
<td>General public</td>
</tr>
<tr>
<td><strong>Computing requirements</strong></td>
<td>Large</td>
<td>Large</td>
<td>Small, medium</td>
</tr>
</tbody>
</table>

Table 2.5 Comparison between Desktop Grid, Volunteer Computing and Personal Volunteer Computing.

recent surveys, published in 2012 [20] and 2016 [200]. Another interesting reference is Marosi’s thesis [127], in which he provides the most detailed comparison of the subtle differences between desktop grids and volunteer computing we have found. He follows with a formal model that captures the difference in their semantics and can be used to compare various middlewares systematically.
and volunteers *increases trust* and almost completely removes the need for security mechanisms against adversaries, which in turn simplifies the implementation of the tools;

- **Personal devices**: To increase the amount of available computing power, cloud and grid computing usually require acquiring new devices specifically for that purpose. Personal volunteer computing instead reuses devices that are already owned for other purposes. This property is shared with volunteer, edge and gray computing, as well as other decentralized approaches;

- **Personal tools**: In most cases, such as cloud, grid, and some volunteer computing platforms such as the World Community Grid [213], the infrastructure is most probably unsuitable to be deployed on the personal devices most people own. Even in other cases of volunteer and peer-to-peer computing, the effort to deploy and maintain the existing systems for personal devices may be too high for most casual potential users. In personal volunteer computing, the aim is to develop *abstractions and designs* that are *simple* enough to be *reimplemented* in many programming environments, present and future, by a single programmer and make the resulting tools quick and easy to deploy.

Compared to existing approaches, personal volunteer computing successfully lowers many barriers compared to existing platforms. It has a lower *technical barrier* by being simpler to use and to deploy. It has a lower *financial barrier* by not requiring money through financial instruments, such as a credit card or a bank account. It has a lower *administrative barrier* by not requiring users to obtain access permissions from third parties. In our opinion, personal volunteer computing represents a compelling and new distributed computing approach to uniquely benefit from the success of the personal computing industry in widely increasing the amount of computing power owned by the general public.
Chapter 3

Design of Pando

Personal volunteer computing provides a conceptual framework to develop new tools but leaves significant freedom in choosing different technical approaches to design them. In this chapter, we present Pando\(^1\), a new and first personal volunteer computing tool that can leverage 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. Pando is based on a *declarative concurrent* programming paradigm [199] which 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 [49] and Unix pipelining [96] programming models. We show for the first time it is also effective in personal volunteer computing tools.

Pando abstracts distribution but otherwise relies on existing toolchains. Programmers define the function to distribute, and the modules it depends on, following the current JavaScript programming idioms. Users can also easily combine Pando in Unix pipelines. Deployment on volunteers’ devices simply requires opening, in their browser, a URL provided by Pando on startup. Devices may join or quit at any time and Pando will transparently handle the changes. We present both the high-level design principles that guided the design and a concrete working implementation, itself organized around the pull-stream design pattern and based on JavaScript [91], WebSockets [206], and WebRTC [205] to enable its execution inside browsers. The implementation of Pando is open source [106]. Compared to other volunteer computing tools, we conceived Pando as a personal tool for quick and easy deployment rather than as a long-running server process. We also avoided the use of a

\(^{1}\text{Pando, ("I spread" in Latin), is also the name of the largest colony of genetically identical trees that share the same root system. It comprises 40,000 trunks (stems). It is believed to be 80,000 years old, making it one of the oldest living organisms [158].}
database and leveraged the heartbeat mechanism of WebSockets and WebRTC to simplify its implementation.

We first introduce the tool using the realistic example of parallelizing the rendering of individual frames of a 3D animation. This example is presented in the form of a tutorial aimed at new users with the benefits they obtain by using Pando (Section 3.1). We then explicitly present the underlying principles that guided our design (Section 3.2). We then delve into the streaming programming model (Section 3.3) that underlies Pando, and the clear and reusable distributed architecture (Section 3.4) we created to implement it. We then briefly discuss how to determine which applications could benefit from Pando (Section 3.5). We then compare our design to related work (Section 3.6) and we conclude with a summary of our contributions. Our JavaScript implementation of the design is freely accessible to anyone [106] and additional installation and usage details are provided in an evolving online handbook [105].

3.1 Usage Example: Parallel Rendering of a 3D Animation

Let’s say a user is working on an animation of a 3D scene, to illustrate a new model that was created using Computer-Assisted Design (CAD) tools. Photo-realistic rendering of those models requires computation-expensive techniques for each frame. A user could therefore benefit from rendering multiple frames in parallel. But regardless of the relative speed of the different processors, the rendered frames should be displayed in the correct order.

Figure 3.1 illustrates an animation in which a camera is rotating around a synthetic 3D scene composed of three reflective spheres, a reflective floor, and a light source. Each frame is rendered by using raytracing [13, 210], a technique in which the colour of each individual pixel of the final image is determined by simulating the trajectory of ray of lights in the scene. This technique is expensive: for a 400x400 pixels image, a small image by today’s standards, it takes between 1-2 seconds of processing on a MacBook Air 2011. An animation composed of a multitude of those images will take a total rendering time that is linear in the number of frames, which could result in multiple minutes or even hours of rendering if it were done sequentially.

There are of course professional solutions to parallelize rendering on clusters that are used by major animation studios [217, 48]. However, to the best of our knowledge, there are no offerings for hobbyists that enable them to use all the personal devices they already own to accelerate the rendering of animations they have created. Pando can help them do so, through a simple programming interface and a quick deployment solution, which we cover in the next two sections.
3.1 Usage Example: Parallel Rendering of a 3D Animation

Fig. 3.1 Rotation Animation around a 3D Scene.

3.1.1 Programming Interface

The distribution of a computation is organized around a processing function which needs to be applied to different input values to produce a sequence 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 camera positions that represents the camera rotating around the scene.

The current implementation of Pando parallelizes the execution of JavaScript code by using the Web browsers of personal devices. To leverage those capabilities, a user needs to write a minimal amount of glue code to make the processing function compatible with Pando’s interface, as illustrated in Figure 3.2. In this example, the raytracing operation is provided by an external library, which is first imported. Existing libraries need no modification: as long as they can be used to process a single value and only rely on operations available in a Web browser, they can be used with Pando. In fact, the open source raytracer library we use, was directly taken from the Web with no modification.

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, and output as a base64 encoded string [21], which simplifies its transmission on the network. Those last three operations take a negligible amount of time compared to rendering the image. 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.
Fig. 3.2 JavaScript Programming Interface Example for Rendering with Raytracing.

While not illustrated in this example, processing may also include asynchronous steps: it could rely on Web Workers, GPU processing, user input, or any other source of events that is external to the main thread of execution. If that were the case, a callback would have to be registered on the external source of events and the result would then be passed to Pando’s callback (cb).

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 [150], 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 [76].

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 3.3, 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.

Fig. 3.3 Unix Programming Interface Example for Rendering Inputs and Processing Outputs.

Pando will take care of handling the distribution and fault-tolerance aspects automatically with no further programming consideration from the user. Deployment steps to spread the computation on multiple devices are explained in the next section.
3.1.2 Deployment

Before deploying Pando, a user must first install it. The simplest option at the time of writing is to use NPM on the command-line, which will automatically install Pando and all its dependencies. The installation should take at most a few seconds on a fast Internet connection, or a few minutes on a slower one. For convenience, Pando can be installed with the `global` option to make its executable globally accessible:

```bash
$ npm install --global pando-computing
```

Fig. 3.4 Installation Example.

Once the installation is successful, a user deploys Pando by starting it on the command-line and then they should wait for URL messages to appear. When displayed, those messages indicate that Pando is ready to spread computations on other devices. Figure 3.5 illustrates the URLs obtained by executing the command of Figure 3.3. These URL messages are provided on the standard error to make them easily visible even if Pando is part of a bash processing pipeline or started within a sub-process. For this example, we start Pando with the additional `start-idle` option so that processing only starts after a first successful connection rather than having Pando immediately start processing values locally while waiting. This provides a clearer demarcation between the initial state and the actual start of processing.

```bash
$ ./generate-angles.js | pando render.js --stdin --start-idle | ./gif-encoder.js
```


Fig. 3.5 Deployment Example.

The system is then in the initial state, illustrated in Figure 3.6: there are a number of pending camera positions (i.e. X1, X2, and X3) to process but not yet any participating devices.

Fig. 3.6 Initial State.

The user may then open the volunteer code URL (i.e. http://10.10.14.119:5000) in the browser of another device connected to the local network. In this example, the user
opens the URL on a tablet as illustrated in Figure 3.7. It has the effect of downloading the processing code from Pando, establishing a WebRTC [205] connection for communication, sending one pending input from Pando to the tablet, and finally starting the processing of the value on the tablet.

![Diagram of tablet and Pando interaction]

Fig. 3.7 A Tablet Joined and Requested a Position to Process.

Once the tablet has completed the rendering of the first image, it returns the output to Pando, which outputs it. The tablet then automatically requests a new position to process, as illustrated in Figure 3.8.

![Diagram of tablet and Pando interaction with second position request]

Fig. 3.8 The Tablet Rendered the First Image and Requested the Second Position to Process.

While the tablet is processing, a phone may join also. The phone will request a pending position and start processing it, as illustrated in Figure 3.9. Both the phone and tablet are then processing inputs in parallel. The same mechanism can be used to leverage multiple cores on the same device. The URL simply has to be opened in multiple browser tabs simultaneously on the same device, one for each core.

Both devices may not have the same processing speed and inputs may not require a uniform amount of computation. This may make the phone return its image before the tablet does. In this case, Pando will retain the result until all other preceding results have been returned to preserve the original ordering, as illustrated in Figure 3.10.
3.1 Usage Example: Parallel Rendering of a 3D Animation

Fig. 3.9 A Phone Joined and Requested the Third Position to Process.

Fig. 3.10 The Phone Rendered the Third Image and Waits for a New Position to be Available.

Devices may fail and disconnect before having provided a result. Pando therefore remembers the inputs sent and will transparently send them again to another connected device in case of failure. For example, the tablet may crash before returning the second image. In that case, as illustrated in Figure 3.11, the phone will take over the second position and process it. To ensure processing devices stay available to process failed inputs from others, Pando keeps devices connected until all results have been returned, even if no inputs are available to process, as previously illustrated in Figure 3.10.

Fig. 3.11 The Tablet Crashed and the Phone Requested the Second Position.
Note that Pando trusts participating devices that if a result is returned, it came from a correct application of the processing function on the input. In the expected usage scenario, the devices added will be owned by the same user or by their friends and family who have no or limited incentives to corrupt the results.

After the phone has returned the third image, Pando outputs the second and third image in order, as illustrated in Figure 3.12. Pando then terminates and disconnects all remaining participating devices.

![Fig. 3.12 The Phone Rendered the Second Image and Submitted a Result. Pando then returned it after all pending outputs and disconnected the phone. Processing is now over.](image)

Volunteers can be invited to contribute their devices, even if the devices are outside the local network from which a user is working. To do so, the user deploys a small micro-server we built for Pando [107] on a platform that provides a public IP address, such as Heroku [80]. Given the URL of that micro-server on startup, Pando uses it to share the volunteer code and to receive the WebRTC connection requests. From a user’s perspective, the only difference compared to the scenario above is that the volunteer code URL provided by Pando on startup is the public address of the micro-server. Being publicly accessible, the URL can then be shared to friends and family on existing social media and communication platforms.

### 3.1.3 Benefits

Note how in the previous 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. Also, to reinforce the points made in Chapter 2, note everything that was not done by the user:

- no particular device needed to be bought to leverage those capabilities, devices already owned by the user could participate;

- no device needed to be registered beforehand, they could join or leave at any time without interrupting the work in progress;

---

2As explained in Section 3.4.1, the use of WebRTC provides automatic Network-Address Translation (NAT) traversal to connect devices across the network boundary.
3.2 Design Principles

- no account needed to be created on an online platform;
- no administrative permissions were required;
- no financial information needed to be provided;
- no background process needed to be maintained to keep the system running while not being used for a specific task;
- no device specificities needed to be accommodated during programming, the same code worked everywhere;
- no scheduling and waiting were needed for the resources to become available, if some devices were within reach they could participate.

Moreover:

- the user was free to combine Pando with tools built in various programming environments for generating inputs and post-processing outputs, as long as they are compatible with Unix processes;
- the user could use social media and communication platforms for requesting help, which requires significantly less effort than creating a marketing website, as is otherwise usually done for volunteer computing projects;
- the user could use the well-known programming syntax of JavaScript in a programming model that is already familiar, avoiding the need to learn potentially complicated operational semantics that parallel and distributed programming often require;
- the user’s data was only shared directly between trusted devices under the control of the user or that of their friends and family. It therefore removed data leak risks, either to other users or to the platform operator, otherwise associated with using cloud platforms or other kinds of globally-shared infrastructure.

In the next section, we take a step back from the previous example to explicit the general principles we used to design Pando and obtain those benefits.

3.2 Design Principles

The previous usage example provided significant benefits because we departed from the conventional wisdom that is used to design the most popular parallel and distributed computing
platforms of today, as explained in Chapter 2. Specifically, we looked at the limitations of previous approaches and we derived, as positive statements, the following *design principles* (DPs):

**DP1** The *deployment* of the tool 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;

**DP2** The tool is *compatible with a wide variety of existing personal devices* such as desktops, laptops, tablets, phones, embedded devices, and personal appliances that people already own;

**DP3** The implementation of tasks is done with a *minimum of programming effort*, as easy to program in a distributed setting as in a local one by using a simple programming model appropriate to the type of tasks targeted and *compatible with a wide range of applications*;

**DP4** The tool is *quick-to-deploy*: it requires little installation effort, has no administrative or financial barriers, starts processing quickly after launch, and then dynamically scales up to benefit from diffusion through social networks;

**DP5** The tool is *composable and modular*: (1) it focuses on coordinating volunteers’ devices but otherwise relies on other tools and technologies for the rest of the needs of users; (2) the core abstractions used in the implementation are usable in other dedicated applications; and (3) the tool can also be combined with high-performance libraries, when available, to leverage the latest results of parallelism research without additional usage complexity.

In the next two sections, we first provide more detail on the underlying programming model we have chosen to follow those principles (Section 3.3). We then provide a concrete and reusable architecture that connects modular components with communication technologies and can be implemented with current Web technologies (Section 3.4).

### 3.3 Programming Model

In this section, we abstract the specificities of the example of Section 3.1 to provide more detail on what makes Pando’s programming model both *powerful* and *easy-to-use*. We present the properties of the programming model in three sections: its core programming model
3.3 Programming Model

(Section 3.3.1), its distributed properties (Section 3.3.2), and its performance properties (Section 3.3.3).

### 3.3.1 Core Programming Model: Streaming Map

Pando’s core programming model corresponds to a set of inputs that is streaming to a processor that is free to process them in any order but must output the results in the same order as their corresponding inputs. The processor applies the function \( f \) on all input values \( x_i \) to obtain a result \( f(x_i) \) (Figure 3.13), which corresponds to the map operation of functional programming. The input \( x_i \) may be a complex object, used to pass multiple arguments, and this object may include functions, therefore enabling some inputs to be processed differently than others. An input may also refer to data that is managed by an external distribution protocol, which in turn may enable more efficient transfers or overcoming the message size limitations of communication channels, if any.

![Fig. 3.13 Streaming Map.](image)

We chose a streaming programming model because it is simple to program (DP3) yet powerful enough to coordinate the usage of multiple devices in parallel (DP2). The reason is that it belongs to the declarative concurrency paradigm [199, Chapter 4] 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 illustrated in Figure 3.13, 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, which may be useful for applications that process continuously arriving
values. We finally realized, while implementing multiple applications, that the model also
worked with problems requiring *feedback loops* of streaming values, in which the outputs
produced by Pando influence the number and values of inputs. Therefore, *unordered*, *ordered*,
and *feedback loop* for applications with both *finite* and *infinite* inputs can all be supported
with the streaming programming model, making Pando quite versatile and powerful. More
detail on example applications that illustrate all these cases are provided in Chapter 5.

### 3.3.2 Distributed Properties

We chose a number of additional distributed properties for Pando to make it easy to program
(DP3) and fast to deploy (DP4), which we present hereafter.

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

The potential number of participating devices is *unbounded*: as much as possible, Pando
should provide the illusion of infinite scalability. Of course, an implementation may exhaust
the physical resources of the devices on which it is executing but as much as possible
the design should strive for maximum scalability without *a priori* bounds on the number
of participating devices. This is similar in spirit to *garbage collection* [44] for managed
programming languages: the latter provides the illusion of infinite memory as long as the
necessary working memory is less than the available memory. In both cases, not having the
limitation baked in the design makes its scalability grow automatically as new devices with
more capabilities are adopted by volunteers.

Pando is also *lazy*: i.e. it reads inputs only when computing resources become available
and demand some for processing. This adjusts the flow of values to the available computing
power to avoid overloading Pando’s memory with pending values. It also makes the imple-
mentation compatible with infinite streams with no additional implementation effort on our
part. The support of laziness does not require additional programming effort from users.

Pando also *tolerates failures* of participating devices, making those failures transparent
to the programmer. We chose a *crash-stop* failure mode\(^3\), 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

\(^3\)Failure modes can range from *crash-stop*, in which a process follow 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 at any time
including intentionally sending messages to hamper progress.
3.3 Programming Model

A browser tab, that executes computations, is suddenly closed or loses network connectivity. In practice it means that, given an input \( x_i \) and a function \( f \), participating devices will always either produce a result \( f(x_i) \) or stop answering forever to any kind of message. 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\(^4\): most of the time, messages will be delivered within a specified time bound. This corresponds to the ability of communication channels such as TCP [193] and WebRTC [205] to suspect failures by failing to receive the acknowledgment of a heartbeat message within a specified time bound.

3.3.3 Performance Properties

Our design principles did not mandate a particular performance profile, which we let open to suit various future applications with variations of our design. As a first step though, to better serve the applications we implemented, we decided to focus on maximizing throughput with the 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 speed. Devices with a faster processing speed will receive more values to process, maximizing resource utilization.

The combination of performance properties with the others mentioned previously, which is summarized in Table 3.1, provides a powerful yet easy-to-use programming model as will be shown by the breadth of applications categories supported in Chapter 5.

\(^4\)Timing assumptions may range from fully synchronous, in which there is an upper time bound on message delivery, passing by partially synchronous [54], 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.
### Table 3.1 Summary of the Properties of Pando’s Programming Model.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming Map</td>
<td>$x_1, x_2, \ldots \rightarrow f(x_1), f(x_2), \ldots$</td>
</tr>
<tr>
<td>Ordered</td>
<td>Outputs are provided in the order of the corresponding inputs.</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Participating devices may join at any time.</td>
</tr>
<tr>
<td>Unbounded</td>
<td>There are no <em>a priori</em> limit on the number of participating devices.</td>
</tr>
<tr>
<td>Lazy</td>
<td>Input values are read when computing resources become available.</td>
</tr>
<tr>
<td>Fault-tolerant</td>
<td><em>Crash-stop</em> failures of participating devices are tolerated.</td>
</tr>
<tr>
<td>Conservative</td>
<td>A single copy of a value to process is submitted at a time.</td>
</tr>
<tr>
<td>Adaptive</td>
<td>Faster participating devices receive more values to process.</td>
</tr>
</tbody>
</table>

3.4 Implementation Overview

In this section, we explain the technology choices we made and why (Section 3.4.1), the *pull-stream* design pattern we used to organize the implementation into individually reusable modules (Section 3.4.2), how the communication technologies and modules were organized in a reusable architecture (Section 3.4.3), and how the architecture could accommodate other distributed execution models (Section 3.4.4).

3.4.1 Web Technologies

Over the last decade, Web browsers and the standards on which they are based have become ubiquitous. We therefore decided to base our design on Web technologies because: (1) they are compatible with the largest number of personal devices (DP2), and (2) they are already known by a large number of programmers (DP3). Of those standards, we selected the following technologies which were the most relevant to our purpose.

*JavaScript* [91] has been designated the *assembly of the Web*\(^5\): it is the common denominator programming language on which all of current Web development depends because it natively runs in current browsers. Since the beginning of aggressive efforts at improving its execution performance that was started with the public release of Google Chrome in 2008 [89], JavaScript has eventually replaced other execution environments based on plugins [139], such Java Applets and Adobe Flash. In addition, the execution performance of JavaScript in modern Web browsers is within a factor of 3 of equivalent numerical code written in C [97, 81]. This makes the performance good enough to easily obtain speedups by running computations in parallel browser pages. Moreover, Web browsers also provide a security sandbox that prevents JavaScript code executing within a Web page from tampering with the host operating system. This protects the devices of volunteers from potential pro-

\(^5\)This has become an unattributed truism.
gramming mistakes. JavaScript has also one of the largest collection of reusable libraries built in common by the programming community and hosted online [150]. We therefore decided to use JavaScript both to implement Pando and to allow users to define their processing function.

**WebRTC** [205] enables direct, or peer-to-peer, communication between browsers without requiring the data to be relayed by a server. The major issue that WebRTC addresses is the establishment of a direct communication through the Network Address Translation (NAT) [92] performed by routers, that hides the IP address of a device on the local network. To work around the issue, WebRTC uses the ICE signalling protocol [194] that helps participants find a way to connect to one another by obtaining their public IP address from outside, using a TURN server such as those provided freely by Google, and testing various known ways to circumvent the NAT until one is found to be working. We chose to use WebRTC to exchange data between participating devices to ensure that even with communications reaching outside the local network, the data of a user will only be copied between participating trusted devices, providing privacy benefits.

**WebSocket** [206] provides the same benefits for Client-Server communication in a browser as TCP [193] is providing to native code: it enables two-way communication between both in a reliable communication channel. Before its introduction, Web applications had to perform active polling to simulate the same behaviour. We initially chose to use WebSocket for bootstrapping our other WebRTC communication channels because it was simpler than HTTP-based protocols. However, in the course of demonstrations in public spaces, we realized that the dependency of WebRTC on TURN servers could prevent connections to be established when an Internet connection was unreliable or inaccessible. We therefore later adapted the Pando design to also enable WebSocket connections for data exchanges, making the deployment simpler on local networks. Moreover, using WebSockets for data transfers has the added benefit that on virtual private networks, Pano does not reveal the local IP addresses of participants, further ensuring privacy.

Both **WebRTC** and **WebSocket** implement heartbeat messages that can suspect a failure and force a disconnection if no heartbeat message has been received after a certain time delay. This provides automatic detection of communication failures in distributed applications.

The next section explains how we made the implementation of Pando, in JavaScript, modular and easier to reason about.

### 3.4.2 Declarative Concurrency with Pull-Streams

While Pando provides a declarative concurrent programming model to its users, we could not directly and easily use the same model for its implementation because JavaScript, as
Design of Pando

many other mainstream programming languages, has not yet integrated primitives to support it. In fact, even if JavaScript has a single thread of execution, the conventional callback-based approach used to perform concurrent operations by waiting on multiple external sources of events is in fact a variation of *shared-state* concurrency [199, Chapter 8]. This happens because multiple callbacks can be allowed to modify variables that are shared. To avoid concurrency bugs, a JavaScript programmer therefore needs to be careful in ensuring that the execution result will be valid regardless of the order in which callbacks will be invoked.

Nonetheless, programming with *streams*, which is becoming quite popular to implement distributed applications, is a special case of declarative concurrency and helps removing the possibility of that kind of bugs. During our initial review of other existing distributed systems written in JavaScript, we stumbled upon the particularly elegant *pull-stream design pattern* [186], a functional approach to the definition of streaming modules that uses a structured callback protocol. This design pattern enables an application to be decomposed in independent but composable streaming modules. Moreover, the pull-stream design pattern only requires support for higher-order functions from the base language to be used, making it usable in almost all mainstream programming languages of 2019. We explain in more detail the pull-stream design pattern in Chapter 4.

By organizing the design around the pull-stream design pattern, we were able to encapsulate the properties of Pando, listed in Table 3.1, in our new *StreamLender* abstraction, whose behaviour is described in the next section. Within Pando, its behaviour is independent from the other input, output, monitoring, and communication aspects that are also performed. This makes StreamLender easy to reimplement in other programming environments. Moreover, in addition to making Pando modular (DP5), the pull-stream design pattern also enabled us to use some of the hundreds of pull-stream modules that have been contributed by the pull-stream community [164] for the non-core aspects.

The pull-stream design pattern also enabled the encapsulation of the concurrency issues within StreamLender, making the rest of the implementation of Pando easier to reason about because it could be considered sequential. The pull-stream design pattern therefore effectively provided the benefits of a declarative concurrent programming model. However, the correct implementation of concurrent pull-stream modules in JavaScript, especially that of StreamLender, still has significant challenges because, as for most JavaScript applications, it is also based on callbacks that mutate shared state. We address them and our solutions in Chapter 4.

The next section explains the various modules that compose Pando and how they are connected with the WebSocket and WebRTC communication channels.
3.4 Implementation Overview

### 3.4.3 Architecture

The core modules of Pando and the way they are connected is illustrated in Figure 3.14. 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 for the Master process and Web browsers for all the Workers, but could also work as a hosted Web application. Deployment consists in executing the tool on the command-line, which starts the Master process and an HTTP server. The HTTP server is used to serve the Volunteer code including the \( f \) function and eventually establish a connection to submit inputs and retrieve results.

Workers connect to the Master in two possible ways, either through WebSocket or WebRTC connections. If on a local network or a virtual private network, Workers retrieve the code from the HTTP server, using the URL provided by Pando on startup, and then connect directly through a WebSocket connection, shown on the left side of Figure 3.14. Otherwise, if direct connectivity is not possible, then Workers connect through WebRTC, as shown on the right side of Figure 3.14. The bootstrap of a WebRTC connection is done through a Public Server, which is a proxy server with a public IP address. This second possibility needs a preliminary step on startup: the user must supply the URL to the Public Server through the `host` command-line option to Pando. Pando then uploads the Volunteer code to the Public Server and establishes a persistent WebSocket connection. After the WebSocket connection is ready, Pando displays the Public Server URL instead of using its own local address. Later, when a new Worker opens that URL, the browser loads the Volunteer code, opens a temporary WebSocket connection to the Public Server, and creates a new WebRTC connection. The WebRTC connection generates signals that are sent to the Master through the Public Server. In response, the Master also creates a new WebRTC connection and replies with its own signals, also through the Public Server. The signals are used by both sides to open the WebRTC connection. If the connection is successful, the WebSocket connection between the Worker and the Pando Server is closed. Otherwise a timeout is raised and the Worker will try to connect again later, after a random delay. Since signalling requires few resources, the Public Server can be a small personal server such as a Raspberry Pi board [168] or the free tier of a cloud such as Heroku [80].

The pull-stream abstractions we designed and reused are shown as modules within the different processes, respectively in white and grey. The actual processing of values is done inside Workers using the existing `AsyncMap` [164] module that applies the function \( f \) on the different inputs. The core coordination is performed by our novel `StreamLender` abstraction, which creates multiple concurrent bi-directional sub-streams, one for each Worker. A sub-stream continuously borrows values from the input of StreamLender and returns results that
are eventually passed to StreamLender’s output. The sub-streams are dynamically created as Workers join.

For communication, 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 Limiter abstraction. The bound can be parameterized using the batch-size argument passed to Pando on startup. By increasing the batch-size, it is possible to hide the transmission latency to ensure inputs are sent to workers while they are busy processing. For example, suppose a single input takes 1 second to process, and the transmission takes 1 second in each direction. With a batch-size of 1, once a result is submitted by a Worker it would take 1 second of delay before the Master would send the next input, then one second for transmission before the computation would start again. Therefore the Worker would be actively busy only 1 second every 3 seconds, or only 33% of the time. By using a batch-size of 3 for the same example, the Worker would be busy 100% because there would always be an input ready to process while the others are in transmission. Modifying the batch-size increases the scope of applications that can benefit from a distribution strategy with Pando.

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. Chapter 4 provides more detail on the implementation of each abstraction: Limiter in Section 4.3, StreamLender in Section 4.4, and DistributedMap in Section 4.5.

### 3.4.4 Potential Support for Other Distributed Execution Models

Our architecture can potentially accommodate other distributed execution models, by changing the fault-tolerance and performance properties of the StreamLender abstraction. To support our argument, we explain the context in which our choice of distributed execution model is appropriate and we sketch potential alternatives for cases where it would not be.

Our current choice of the crash-stop distributed execution model is a compromise between performance, fault-tolerance, and implementation simplicity. Our choice is appropriate for cases where a user trusts that participating devices will produce a correct result but participants may disconnect from the network or close the computations at any time during
Fig. 3.14 Architecture of Pando.
execution. For other cases, such as situations where volunteers could intentionally submit invalid results, a stronger version of StreamLender would be necessary.

In that case, for example, a majority voting scheme [19] could be used: each input could be processed \( n \) times and the result would be accepted once \( \lfloor \frac{n}{2} \rfloor + 1 \) results obtained from different devices would be sufficiently similar.\(^6\) This would however divide the overall throughput by a factor of \( n \).

Our partially synchronous solution would also work in the less general synchronous execution model, in which the upper bound on message delivery time would be guaranteed at all times rather than only eventually. However, our solution is not applicable to the most general asynchronous situation, in which any message may be delayed for an arbitrary long time. This could arise if we were to use UDP [196] as transport without a failure detection mechanism such as those provided by TCP [193], WebSocket [206], and WebRTC [205]. In an asynchronous case, it would be impossible to detect whether a node has crashed or a message is simply delayed.

This problem could be handled with the following variation of eager scheduling [18] adapted for streams. In that strategy, StreamLender could split the incoming stream in batches of values such that once all values of a batch have been submitted, the first values for which no result has been obtained yet would be submitted a second time, etc. until all results have been obtained. Then StreamLender would move to the next batch. This approach would tolerate arbitrarily slow or unresponsive workers at the cost of reduced throughput.

Remarkably, the previous variations could be implemented only by changing some properties of StreamLender (and transport channels) while keeping the rest of the architecture the same, making our architecture potentially applicable in a range of distributed execution models.

3.5 Determining Applications for which Pando is Beneficial

Pando is not necessarily useful for all applications, even if they fit the streaming map programming model of Section 3.3. The main criteria is that the computation time for each input must be sufficiently large to amortize the distribution of data. We briefly sketch how to quickly determine which applications would benefit.

The application examples that will be explained in Chapter 5 show that the main JavaScript processing function is relatively easy to implement. The best way to determin-

\(^6\)Full equality for numerical problems involving floating point operations is not always possible.
mine whether Pando can be useful in a particular context is therefore to test how long a single input takes to process. The current implementation of Pando makes this easy by offering an option to test the processing of inputs without any data distribution, by passing the -local parameter on startup. We have found empirically that applications that take at least one second of processing per input are good candidates.

Once it has been established that each input takes a significant amount of processing time, the second test is to determine the data distribution time. To do so, Pando can simply be started again, this time omitting the -local option, and opening the volunteer code URL in a browser. The difference between the two tests indicates the cost of data distribution.

It is possible to augment the processing time for inputs by modifying applications to combine multiple values to process in a single input. In cases where it is not possible, the batch-size, which controls the maximum number of inputs a Limiter let through, can be augmented. With these two options, many applications that fit the streaming map programming model can benefit from distribution, with minimal programming efforts.

3.6 Related Work

To the best of our knowledge, Pando is the first tool for the explicit purpose of personal volunteer computing. Nonetheless, a number of other tools that are similar in some aspects have been created by others. In this section, we cover other work related to our design as well as other systems whose design is closest to ours.

3.6.1 Main Concurrency Paradigms

There are three main paradigms for concurrent programming: shared-state and message-passing concurrency are more expressive but harder to reason with, and declarative concurrency is less expressive but significantly easier to reason about. We briefly present the three paradigms’ key features.

On the one hand, in shared-state concurrency, multiple threads may access the same memory, with the Java [90] and CUDA\footnote{At the lowest level, CUDA uses memory fences to synchronize memory operations between threads that access the same memory location. Interestingly, the language also provides higher-level constructs based on streams to simplify programming.} [46] programming languages as typical examples. In that concurrency paradigm, the programmer uses atomic operations, that guarantee that a value read from memory has not been modified before a subsequent write occurred, either directly or through higher-level abstractions such as semaphores, locks, and monitors to synchronize access to memory between different threads. In message-passing concurrency,
multiple threads do not share the same memory but instead communicate by sending each other messages that are stored in message queues, with Erlang [55] and MPI [136] as typical examples. This paradigm mandates a style of programming in which memory is not shared. Nonetheless, both paradigms are equivalent in expressiveness because one can be implemented in terms of the primitives of the other [199] and therefore they can both exhibit concurrency bugs. Both are harder to debug in case of programming mistakes because bugs may only appear in some executions. These executions may be hard to reproduce because they depend on a particular inter-leaving of the participating threads.

On the other hand, declarative concurrency removes the possibility of concurrency bugs because it makes the non-determinism of a concurrent execution non-observable to the programmer: the result will always be the same regardless of the particular execution interleaving of the participating threads. For example, Pando guarantees that the result will be the stream \( f(x_0), f(x_1), \ldots \) regardless of the relative speed of the participating devices and whether some crashed during execution or not. Programmers therefore do not have to care about concurrency issues, making Pando significantly easier to use. In theory, declarative concurrency has expressiveness limitations regarding non-deterministic programs. For example, a server that listens to requests but does not know the source of the next one cannot be implemented in that paradigm [199]. However, in practice this limitation can be mitigated by selectively introducing non-determinism in limited ways while keeping much of the rest of the implementation declarative concurrent. For example, Pando encapsulates the non-determinism of which device is going to join next in StreamLender and keeps the rest of the implementation sequential.

### 3.6.2 Declarative Concurrency

Declarative concurrency has been studied in the context of dataflow programming, with languages such as Lucid [202] and Oz [182]. In the Oz language, the declarative programming model can be used directly to implement concurrent modules [199, 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.\(^8\)

While not widely known under this specific name, the declarative concurrency paradigm has nonetheless been experienced by a large number of programmers and researchers through the popular MapReduce [49] framework and Unix pipeline programming [96]. In effect, Pando implements the map operation of MapReduce; the other filtering and reduction phases

\(^8\) Further detail may be found in a pedagogical presentation of declarative concurrency using the Oz language in Chapter 4 of *Concepts, Techniques, and Models of Computer Programming* [199].
can be performed locally, if necessary, by chaining with other Unix tools, such as `grep` and `awk` for example.

Using declarative concurrency directly in our Pando implementation, in JavaScript, would have removed many potential sources of bugs. However, JavaScript, as many other mainstream programming languages, has not yet integrated features that make that style of programming widely accessible and easy. The current proposal for async/await in JavaScript [91, Section 6.2.3.1] offers implicit synchronization of asynchronous operations, which resembles the implicit data synchronization of single-assignment variables, but is more limited and cumbersome to use than the Oz single-assignment variables. We propose a slightly different and more expressive pseudo-code alternative in Chapter 4 that we found to be sufficient for expressing our algorithms and could serve as inspiration for future extensions.

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

### 3.6.3 Stream Processing


The streaming platforms mentioned previously are usually programmed using *dataflow graphs of computation* that combine multiple operators and complex data flows. These platforms then ensure their efficient and reliable execution on different targeted execution environments. This level of expressivity is not necessary for many personal projects and applications, such as those that we implemented in Chapter 5. To support our applications with a lower level of implementation complexity and make our design easier to reimplement in 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.
3.6.4 Browser-Based Volunteer-Computing Implementations

Fabisiak and al. [56] have proposed to group publications on browser-based volunteer computing implementations in three generations: the first generation [39, 6, 18, 175, 62, 147] was based on Java applets; the second generation [31, 100, 135, 24] used JavaScript instead but was somewhat limited by its performance; and the third generation [174, 52, 50, 169, 103, 129, 133, 126] fully emerged once performance issues were solved in multiple ways: JavaScript became competitive with C [97], WebWorkers [203], that did not interrupt the main thread, were introduced, and new technologies, such as WebCL [204], 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 [102, 119] that incorporates the latest communication technologies, such as WebSocket [206] and WebRTC [205], 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.\(^9\) However, the key difference of Pando is in our focus on the personal aspects of volunteer computing (Chapter 2) that led to specific design principles (DP1-DP5, Section 3.2) with various concrete impacts on its programming model, deployment strategy, and implementation choices.

Of the systems that have generic programming models, many focus on batch-processing [31, 100, 101, 50, 169, 102] 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 [24, 72, 174, 103, 133]. 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, Section 3.2), as are more often used in a personal context.

While some general purpose projects aim to deploy new global platforms [6, 39, 147, 175, 18, 101, 50, 103, 167, 2], sometimes on clouds [169, 119], 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, Section 3.2), 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 [167], (2) brokers/dispatchers/bridges to organize the tasks submitted [6, 39, 101, 50, 103, 2], (3)

\(^9\)As mentioned in the other sections, there are of course other peer-to-peer systems leveraging WebRTC for other uses [15, 51].
dynamic management of a set of managers [18], and (4) advocates [175] to represent clients in the server.

Many implementations are organized around a database [31, 101, 24, 174, 50, 169, 38]. 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 [100, 101, 135, 24, 174, 52, 50, 169, 103, 129, 38], to distribute inputs and collect results. Instead, and similar to newer projects [102, 119], 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, localizing the programming changes required to support different distributed execution models. These simplifications in turn hopefully make 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.

### 3.6.5 Peer-to-Peer Computing in Browsers

The server-centric model of Web technologies has historically limited the development of peer-to-peer Web applications. The recent introduction of WebRTC [205] has now opened the door to peer-to-peer systems that can execute in the browser, an opportunity that lead to the creation of many new peer-to-peer platforms [93, 209, 153, 201, 190, 82, 51].

Of all previously mentioned platforms, the closest to Pando is browserCloud.js [51] in its aim to provide a computation platform powered by the devices of participants. However, Pando’s implementation approach is quite different in its support for a single client, its overlay organization, its removal of maintenance overhead, and its absence of a need for a discovery mechanism. We cover those differences, hereafter, by contrast to browserCloud.js but the same points equally apply to other peer-to-peer systems.

BrowserCloud.js enables its participants to both contribute computing resources and submit tasks. Concurrent task submissions by multiple participants requires mechanisms to distribute the load to different subsets of workers. In contrast, each Pando user creates their own specific deployment rather than sharing the same global platform, which eliminates the need for mechanisms to identify subsets of free workers. We chose this approach because there are so many available personal devices that millions of independent networks are possible without overlap.

BrowserCloud.js organizes its participants in a structured overlay in the form of a ring to enable participants to efficiently submit tasks to and retrieve results from one another. The quality of services offered by the overlay is dependent on the reliability of the specific peers the user is connected to. For example, if the peers are unreliable (high level of churn), this
can introduce delays in connecting to a set of workers. In contrast, Pando does not need workers to communicate and therefore organizes them in a tree, forming a master-worker organization [175]. Furthermore, the availability of Pando only depends on the reliability of the master, which is under the direct control of the user. Failures of workers will only affect the latency in obtaining results, not in initiating the computations, and only for a specific user.

BrowserCloud.js requires maintenance and care from participants to keep the platform operational by making sure their device is still connected and contributing, even when no task is actively running. In contrast, Pando terminates when its task is over which removes the maintenance overhead when not in use.

BrowserCloud.js implements its own discovery service to enable clients to find free workers among the available participants. In contrast, Pando relies on existing social media to announce work to be performed and gather volunteers. We therefore benefit from their reliability, popularity, and the good reputation of public identities of their users. This in turn simplifies our implementation.

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.

### 3.6.6 Ongoing Related Open Source Projects

The following projects were all founded around 2013-2014, by a small clique of open source programmers that wanted to build decentralized alternatives to many centralized online services. In the meantime, each project has followed its own particular approach and fully blossomed into its own community of enthusiasts and contributors. In the most general sense, Pando is part of the same decentralization agenda so our first motivation for presenting them here is to show that this research field is quite active. Our other motivations are to mention what we reused from them, mention technical contributions we made in return, and show that their functionalities are complementary to those of Pando.

**WebTorrent** [207] is an implementation of BitTorrent\(^\text{10}\) for the Web, using WebRTC and JavaScript. It includes hybrid clients that can participate both in the BitTorrent and the WebTorrent network simultaneously. From the WebTorrent project, we reused two libraries: `simple-peer` [3] for WebRTC communication, and `simple-websocket` [4] for WebSocket communication.

\(^{10}\)BitTorrent [28] is a peer-to-peer protocol for immutable file distribution in which the bandwidth necessary to download files is simultaneously provided by other peers also interested in the same file.
3.6 Related Work

DAT [47] is a peer-to-peer data archival and distribution protocol, similar to BitTorrent in its distribution aspect while also enabling updates to data repositories by their original author. The main use case is for researchers to share and archive open datasets. It has a companion Beaker browser [22] itself based on Chromium [40], the open source foundation of Google’s Chrome browser. The Beaker browser natively supports the DAT protocol in addition to HTTP as well as tools for self-publication of personal websites in a peer-to-peer fashion. We experimented with using DAT for data distribution in one of our example applications in Chapter 5.

Secure-Scuttlebutt (SSB) [178] is a decentralized protocol for building social applications. It is based on replicated append-only chains of messages, authenticated by cryptographic signatures. Each user has their own chain of messages and there is no need for global consensus on their state. SSB instead provides eventual consistency through gossiping [26] between participants: participants exchange the latest messages of the common friends they follow. The replication of messages therefore follows the actual lines of trust between users: by default messages are not replicated beyond friends-of-friends, although this is configurable on a per-client basis to be looser or more stringent. The design is based on the Scuttlebutt11 [198] replication protocol that was originally designed for making the Amazon datastores fault-tolerant and performant in the presence of high-load. From the SSB project, we built upon the pull-stream design pattern [186] that is used internally. We also contributed new pull-stream modules [108, 110, 112, 111, 117], including an implementation of StreamLender [109]. Discussions with the SSB developer community greatly helped clarifying our articulation of personal volunteer computing (Chapter 2) as well as many properties that derived from using pull-streams.

The InterPlanetary File System (IPFS) [88] is a peer-to-peer distributed file system that aims to replace HTTP by providing better functionalities for archiving old versions of websites, by making the distribution of content faster, by removing central points of failures, and by supporting offline work. It works by using content-addressed links, typically derived from the hash of the data they represent, in combination with a BitTorrent-inspired data distribution protocol, as well as a distributed hash table (DHT) for name resolution and content discovery. Contribution of storage space for hosting IPFS data will be incentivized in the future through a peer-to-peer market based on Filecoin [61], a new crypto-currency developed by the same group. IPFS uses and supports the development of libp2p [121], an offshoot that became a modular library for developing peer-to-peer applications, started by David Dias, the developer of browserCloud.js [51].

11 Sea-slang for gossip.
Pando is complementary in enabling users to distribute their *computations* using personal devices, leveraging the other projects for communication and data storage needs. Together they suggest the possibility of a viable and important common research agenda that would greatly benefit from academic contributions in various related fields.

### 3.6.7 Emerging Technologies

In the course of this project, new technologies have emerged which could either improve the performance of applications built with Pando or benefit from new tools based on Pando’s design. We briefly present them to inspire future work.

WebAssembly [75] is a new low-level language for the web that strives to be safe, fast, portable, and compact and overcome the limitations of JavaScript as a compilation target. In numerical benchmarks, it was shown to have similar performance as C [81]. It should be possible to build fast libraries for the Web or as a compilation target for *processing functions* that Pando could then parallelize. Once enough Web browsers on personal devices support it, it should be directly usable with no modification to Pando to accelerate user workflows.

Julia [25, 94] is a new open source programming language for high-level dynamically-typed numerical programming, similar to Python/Numpy [152], Matlab [130], and R [166]. We believe the Pando design should be straightforward to port to any of them albeit with less guarantees in terms of security and portability to multiple personal devices compared to Web technologies. These languages may not provide an execution sandbox that is as well-tested (or available) as for Web browsers. They may also not run natively on multiple devices without the prior development of computing clients. Nonetheless, an implementation of Pando in these other languages could still be useful to use distributed local resources in parallel, such as idle machines in a research lab.

### 3.7 Summary

Pando is a first tool for personal volunteer computing to parallelize the application of a function on a stream of values by using personal devices’ browsers. It requires very little programming to leverage existing libraries and is quick to deploy on multiple cores and distributed personal devices. It *dynamically scales* to new devices and *gracefully tolerates the sudden disconnection* of participating devices. It is compatible with a wide number of existing personal devices people already own by executing its computations in their browser upon voluntary and explicit participation. Its design is simple by having a deployment that is *specific* to a single project, a single known user, and the lifetime of the tasks.
Pando’s use of a *declarative concurrent* programming model, in which the *non-determinism of the execution is not observable* and of which stream programming is a subset, shows that this paradigm is a great match for developing personal volunteer computing tools. Pando should be especially beneficial to programmers from the general public by being easy to reason about while enabling the transparent scaling of parallel computations on many cores and many devices. The choice of a programming model based on an *ordered streaming map*, that can scale *dynamically*, with no *a priori* bound on the number of participating devices, *lazily* reading input values and providing *fault-tolerance* has shown to be a great combination to simplify its programming while being useful for a wide range of applications. We also focused on *throughput* by *conservatively* sending a value for processing to a single device at a time and *adapting* the flow rate to keep the faster devices busy.

Our implementation of Pando is based on JavaScript, WebRTC, and WebSocket. It is organized around the *pull-stream design pattern* which only requires support for higher-order functions from the base language and enables the factorization of Pando into reusable and independent modules. Its greatest benefit has been to encapsulate the concurrency issues and the properties of Pando’s programming model in a new StreamLender abstraction, making the rest of the implementation essentially sequential and able to easily reuse existing pull-stream modules. Implementing correct declarative concurrent abstractions, such as StreamLender, is still a significant challenge that will be addressed in Chapter 4. The architecture of Pando, which connects WebRTC and WebSocket with the new Limiter, StreamLender, and DistributedMap modules as well as the existing AsyncMap module, should be relatively straightforward to reimplement in other programming environments. The scalability of the current architecture is however limited by the number of concurrent WebRTC connections that can be handled by a Web browser. This challenge will be addressed in Chapter 7. Variations in the fault-tolerance implementation of StreamLender would also enable the same design to work for distributed execution models other than the *crash-stop partially synchronous* model we have chosen, with no changes to the rest of the implementation.

Compared to existing streaming platforms, Pando strives to have a simpler implementation to be *easier to reimplement* in other programming environments: it uses a simpler programming model than other streaming platforms which otherwise support arbitrary graphs of computations. Its programming model is still sufficient to implement a wide variety of useful applications including those with an *infinite* number of *ordered* inputs and those based on *feedback loops*, as will be shown in Chapter 5. Compared to many browser-based voluntary computing platforms, its implementation foregoes the use of a database and instead handles concurrency issues with the StreamLender abstraction. Compared to peer-to-peer platforms, it focuses on being used as a *personal tool* by reusing the capabilities of other
existing tools to simplify its implementation, rather than providing all required services as a *self-sustaining global platform* able to operate independently. This makes Pando complementary to many emerging peer-to-peer projects that offer protocols for communication, data distribution, and social applications. In the future, Pando should also be able to leverage the upcoming WebAssembly for increased performance and be portable to Julia and other established numerical programming environment, such as Python/Numpy, Matlab, and R. The differences between Pando and other published solutions are summarized in Table 3.2.

<table>
<thead>
<tr>
<th>Programming Model</th>
<th>Stream-Process Platforms</th>
<th>Browser-Based Volunteer Comp.</th>
<th>BrowserCloud.js</th>
<th>Pando</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataflow graphs</td>
<td>Varied</td>
<td>Message-pass. concurrency</td>
<td>Decl. conc. w/ pull-streams</td>
<td></td>
</tr>
<tr>
<td>Map(-Reduce)</td>
<td>Java/JavaScript, Database</td>
<td>JavaScript</td>
<td>JavaScript</td>
<td></td>
</tr>
<tr>
<td>Varied</td>
<td>HTTP API</td>
<td>WebRTC</td>
<td>WebRTC, Websocket</td>
<td></td>
</tr>
<tr>
<td>Varied</td>
<td></td>
<td>WebRTC</td>
<td>Websocket</td>
<td></td>
</tr>
</tbody>
</table>

*Table 3.2 Comparison between Pando’s Implementation and Related Systems.*
Chapter 4

Implementation of Pando

As previously explained in the overview of Section 3.4.3, the core of the implementation of Pando is based on the Limiter, StreamLender, and DistributedMap abstractions. In this Chapter, we first present in more detail the pull-stream design pattern on which they are based (Section 4.1) and a declarative concurrent notation at a higher level than JavaScript to concisely explain their behaviour (Section 4.2). We then describe in more detail the Limiter (Section 4.3), StreamLender (Section 4.4), and DistributedMap (Section 4.5) abstractions as well as the algorithms that implement them. We then introduce a run-time verification method to test the correctness of an implementation of StreamLender (Section 4.6). We finish with a brief discussion of the benefits of our approach (Section 4.7), some related work (Section 4.8), and a summary (Section 4.9).

4.1 Pull-Stream Design Pattern

The pull-stream design pattern [186] is a functional code pattern that enables streaming modules to be built by following a simple callback protocol. The callback protocol and the types of modules that compose a streaming processing pipeline are illustrated in Figure 4.1. The pull-stream design pattern has originally been proposed by Dominic Tarr [186] 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 [164].

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. The different events that make up the callback protocol and their corresponding JavaScript function invocations are shown in Table 4.1.
Fig. 4.1 Pull-Stream Design Pattern. Callback protocol on top and pipeline of composable modules at the bottom.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Event</th>
<th>JavaScript Function Invocation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>ask</td>
<td>request(false, cb)</td>
<td>ask for a value</td>
</tr>
<tr>
<td>request</td>
<td>abort</td>
<td>request(true, cb)</td>
<td>abort the stream normally</td>
</tr>
<tr>
<td>request</td>
<td>fail</td>
<td>request(new Error(), cb)</td>
<td>abort the stream abnormally</td>
</tr>
<tr>
<td>answer</td>
<td>value</td>
<td>cb(false, v)</td>
<td>answer with the value 'v'</td>
</tr>
<tr>
<td>answer</td>
<td>done</td>
<td>cb(true, undefined)</td>
<td>no more values</td>
</tr>
<tr>
<td>answer</td>
<td>error</td>
<td>cb(new Error(), undefined)</td>
<td>an error occurred</td>
</tr>
</tbody>
</table>

Table 4.1 Pull-Stream Callback Protocol Events.
4.1 Pull-Stream Design Pattern

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 4.2.

```javascript
function source (n) {
    var i = 1
    return function output (abort, cb) {
        if (abort)
            return cb(abort, undefined)
        else if (i<=n)
            return cb(false, i++)
        else
            return cb(true, undefined)
    }
}

function sink (request) {
    request(false, function answer (done, v) {
        if (done) return
        else request(false, answer)
    })
}
sink(source(10))
```

Fig. 4.2 Pull-Stream Source and Sink Examples.

A module may also both consume and produce values, in which case it can be used between a source and a sink. The original documentation [186] call these modules through, but we prefer the term transformer. An example transformer that increments each value by 1 before passing it to the next transformer is illustrated in Figure 4.3.

Figure 4.4 shows a duplex module that also consumes and produces values, similar to a transformer, but is typically used to represent communication channels, such as a TCP connection. The definition illustrates the object form of the pull-stream pattern, that eases the use of the input and output in different pipelines or their connection in a loopback interface.

Once implemented, pull-stream modules provide clear semantics and are easy to combine, as illustrated in Figure 4.5. However, the construction of the pipeline is inverted compared to the flow of values, that goes from the source to the sink. Therefore, the pull-stream library [165] in JavaScript provides a convenience function, named pull, that takes a variable number of modules as arguments, in the same order as the values flow, and then constructs the pipeline in the proper order. Figure 4.5 shows a simplified version of pull that takes an array of modules in argument, rather than relying on the arguments object in JavaScript as other languages may not support function invocation with a variable number of arguments.
function transformer (request) {  // function form
    return function output (abort, cb) {
        request(abort, function answer (done, v) {
            if (done) cb(done, undefined)
            else cb(false, v+1)
        });
    }
}
// increments each value by 1
sink(transformer(source(10)))

var transformer = {  // object form
    input: function (request) {
        this._request = request
    },
    output: function (abort, cb) {
        this._request(abort, function answer (done, v) {
            if (done) cb(done, undefined)
            else cb(false, v+1)
        });
    }
}
// a pipeline can also be constructed in object form
transformer.input(source(10))
sink(transformer.output)

Fig. 4.3 Pull-Stream Transformer Example.

var duplex = {  // object form
    input: function (request) {
        // ex: connection to the input of a TCP channel
    },
    output: function (abort, cb) {
        // ex: connection to the output of a TCP channel
    }
}
// loopback pipeline, ex: echo channel
duplex.input(duplex.output)

Fig. 4.4 Pull-Stream Duplex Example.
4.1 Pull-Stream Design Pattern

4.1.1 Benefits

The pull-stream design pattern provides a combination of many interesting properties, which we illustrate with code examples.

**Portability**

The pull-stream design pattern only requires support for higher-order functions from the programming environment. As illustrated in Figure 4.6 with Python examples, implementations of abstractions built by following the pattern are therefore relatively straight-forward to reimplement in other popular programming languages of today.

**Flow regulation**

An upstream module (producer) and a downstream module (consumer) may both regulate the flow of values by respectively delaying the current answer and the next request. Both cases are shown in Figure 4.7.

**Early-termination**

As illustrated in Figure 4.8, the consumer may abort the stream early before the producer has generated all its values.

```javascript
// Increments each value by 2
sink(transformer(transformer(source(10)))) // using function form

// Connect modules in the order of flow
function pull (args) {
  var m = args[0]
  for (var i = 1; i < args.length; ++i) {
    m = args[i](m)
  }
  return m
}

pull([ // Also increments each value by 2
  source(10),
  transformer,
  transformer,
  sink
])
```

Fig. 4.5 Pull-Stream Composition.
def source(n):
    class context:  # Create a lexical context for closure variables
        i = 1

    def output(abort, cb):
        if abort: return cb(abort, None)
        elif (context.i <= n):
            j = context.i
            context.i += 1
            return cb(False, j)
        else: return cb(True, None)
    return output

def sink(request):
    def answer(done, v):
        if (done):
            print 'done'
        else:
            print v
        request(False, answer)
    request(False, answer)

def transformer(request):
    def output(abort, cb):
        def answer(done, v):
            if (done): cb(done, None)
            else: cb(False, v+1)
        request(abort, answer)
    return output

sink(transformer(source(10)))

Fig. 4.6 Pull-Stream Source, Sink, and Transformer in Python.
4.1 Pull-Stream Design Pattern

```javascript
function source (n) {
    var i = 1
    return function output (abort, cb) {
        setTimeout(function () { // the answer may be delayed
            if (abort) return cb(abort, undefined)
            else if (i<=n) return cb(false, i++)
            else return cb(true, undefined)
        }, Math.random()*1000) // 0-1000ms timeout delay
    }
}

function sink (request) {
    request(false, function answer (done, v) {
        if (done) return
        else {
            setTimeout(function () { // requests may also be delayed
                request(false, answer)
            }, Math.random()*1000) // 0-1000ms timeout delay
        }
    })
}
sink(source(10))
```

Fig. 4.7 Flow Regulation (in JavaScript).

```javascript
function print (done, v) {
    if (!done) console.log(v) // prints the value of 'v'
    else console.log('done') // prints 'done'
}
var request = source(10) // normally produces 10 values
request(false, print) // prints the first value: '1'
request(true, print) // aborts, prints 'done'
```

Fig. 4.8 Early-Termination Example using the Source of Figure 4.2.
Graceful error handling

As illustrated in Figure 4.9, any module may propagate an error and has an opportunity for cleaning its internal state after an error or the termination of the stream upstream or downstream. Moreover, since the callback protocol handles both normal and abnormal cases, the handling of errors can become transparent to a processing pipeline.

```javascript
function flaky () {
    return function output (abort, cb) {
        if (abort) return cb(abort, undefined)
        else {
            try {
                if (Math.random() > 0.9) throw new Error('Random error')
                else return cb(false, 'success')
            } catch (err) return cb(err, undefined)
        }
    }
}

function loop (done, v) {
    if (!done) { console.log(v); request(false, loop) }
    else if (done instanceof Error) console.log('error: ' + done)
    else console.log('done')
}

var request = flaky() // prints 'success', ..., 'error: Random error'
request(false, loop) // errors stay in the pipeline
```

Fig. 4.9 Graceful Error Handling.

Lazy generation

As illustrated in Figure 4.10, values are generated lazily therefore a source may produce infinitely many values. Pull-stream pipelines may therefore be used in long running systems with real-time updates.

Declarative composability

As illustrated in Figure 4.11, modules may be composed before the construction of the complete pipeline. This favours the creation of simple modules and their use in multiple libraries. Both the composition of modules and the construction of a pipeline are declarative: users don’t need to understand the callback protocol.
4.1 Pull-Stream Design Pattern

```javascript
function infinite () {
  var i = 1
  return function output (abort, cb) {
    if (abort) return cb(abort, undefined)
    else return cb(false, i++)
  }
}

function sink (request) {
  request(false, function answer (done, v) {
    if (done) return
    else {
      setTimeout(function () {
        request(false, answer)
      }, 1000) // Wait 1000 ms between requests
    }
  })
}

sink(infinite()) // run forever
```

Fig. 4.10 Lazy Generation.

```javascript
var countFrom2 = transformer(source(10)) // Src + Transf = Src
sink(countFrom2) // delayed pipeline construction

function partial (m1, m2) { // partial application when no Src
  return function (source) {
    return m2(m1(source))
  }
}

// Transf + Transf = Transf
var incBy2 = partial(transformer, transformer)
var sink2 = partial(sink, incBy2) // Sink + Trans = Sink
sink2(source(10)) // delayed pipeline construction
```

Fig. 4.11 Declarative Composability of Source, Sink and Transformer from Figures 4.2 and 4.3.
Declarative Concurrency

As illustrated in Figure 4.12 with a *ParallelMap* example, the implementation of modules may use parallelism to improve the overall throughput transparently, e.g. a transformer may request multiple values and process them in parallel before returning results. As long as the non-determinism of the implementation is non-observable, downstream modules shall be compatible with no extra effort.

Fault-tolerance

As will be explained in Section 4.1.3 and illustrated shortly in Figure 4.14, some errors can be handled within the protocol without terminating the streaming pipeline. This is useful when a module is connected to multiple modules, some of which may fail or terminate earlier than others.

4.1.2 Callback Protocol Invariants

So far, our presentation of the pull-stream design pattern has been driven by examples and relied on the ability of developers to generalize beyond them. This is however not sufficient to provide guarantees of interoperability with other modules. Moreover, the current documentation on pull-streams [165] is not explicit on the invariants expected on *correct* pull-stream modules. In practice, by inspecting the behaviour of commonly used modules and discussing with the developers, we have established the following invariants. A compliant pull-stream module therefore must:

1. not perform additional requests after having received a *done* or *error* as an answer, or after having made a request that terminates the upstream module, with an *abort* or *fail* request;

2. ensure every request received is eventually followed by an answer;

3. ensure every answer callback is invoked only once;

4. wait for the previous answer to be received before making a new request, unless the module is aborting or failing;

5. ensure the order invocation of answer callbacks happens in the order in which their corresponding requests were made;

6. ensure termination answers and requests are correctly propagated.
4.1 Pull-Stream Design Pattern

function drain (cb) {
  var buf = []
  return function (request) {
    request(false, function next (done, v) {
      if (done) return cb(buf)
      else {
        buf.push(v)
        request(false, next)
      }
    })
  }
}

function parallelMap (f) {
  return function (request) {
    var results = null, count = 0, done = false, k = 0
    return function output (abort, cb) {
      if (results === null) {
        // Read all values from upstream in a buffer
        drain(function done (buf) {
          if (buf.length === 0) return cb(true, undefined)
          // Apply 'f' on all values in parallel
          results = []
          buf.forEach(function (x, i) {
            f(x, function (y) {
              results[i] = y
              count++
              // Wait for all values to be processed
              if (count === buf.length) {
                return cb(false, results[k++])
              }
            })
          })(request)
          // Return remaining results
        })
        // No more results to return
        else if (k < results.length) return cb(false, results[k++])
        // No more results to return
        else return cb(true, undefined)
      }
    }
    // usage
    pull([source(10),
      // Simulate a processing of at least one second, with random
duration between invocations
      parallelMap(function (x, cb) { setTimeout(function () { cb(x*x) },
        1000 + Math.random()*1000) },
      sink
    ])
}

Fig. 4.12 Declarative Concurrency ParallelMap Example. It is seamlessly used in combination with the Source, Sink and Pull from Figures 4.2 and 4.5. While every of the 10 invocations of the function f may take between 1000 and 2000 ms, the total processing takes at most ≈2000 ms.
These invariants will be necessary later to understand some implementation decisions and we have strived to ensure our own modules follow those invariants. To help other developers ensure compliance, we have implemented pull-stream modules that can check at run-time the compliance with the previous invariants [116].

4.1.3 Complexity in Implementing Concurrent Modules

Pull-stream modules are straight-forward to implement as long as they form a sequential pipeline, i.e. only a single value is produced, transformed, and consumed at a time. The implementation of concurrent modules, i.e. modules that may produce or consume values with multiple streams forming parallel pipelines, however becomes harder to specify and reason about, as might be hinted by the ParallelMap example of Figure 4.12. As an additional example, Figure 4.13 illustrates a Merger transformer that creates a single stream of values from two concurrent sources in a round-robin fashion, i.e. taking alternatively one value on each input stream until the second is exhausted and continuing with the values of the first until it is also exhausted. One of the two streams might fail during processing; nonetheless, Merger will continue requesting values from the other after the error happened, an example of fault-tolerance.

![Merger Diagram](image)

**Fig. 4.13 Merger Example with Two Input Streams.**

A possible JavaScript implementation of the Merger module is shown in Figure 4.14. Compared to other module implementations we have shown so far, there are two main sources of additional complexity that come from its concurrent behaviour. First, it is now necessary to synchronize on multiple events in certain cases: for example the answer to an abort is provided downstream only once all input streams are done (line 9). Second, much of the complexity in the implementation comes from the necessity to handle all possible termination conditions. The normal case of round-robin between the two inputs only requires 6 lines of code (line 3, 21, 37-40); most of the remaining 45 lines of code are dedicated to handling termination.

Similar to the implementation of the Merger module, an efficient JavaScript implementation of concurrent modules generally uses internal variables and boolean conditions to
function merger (request1, request2) {
    return function output (abort, cb) {
        var next = 1, done1 = false, done2 = false
        function wait (index, cb) {
            return function answer (done, v) {
                if (index === 1) done1 = done
                else done2 = done
                // wait for both to be done before returning
                if (done1 && done2) cb(true, undefined)
            }
        }
        function nextValue (index, cb) {
            return function answer (done, v) {
                index === 1 ? done1 = done : done2 = done
                if (done1 && done2) { // both done, we are done
                    cb(true, undefined)
                } else if (index === 1 && done1 && !done2) {
                    request2(false, nextValue(2, cb)) // 1 done, try 2
                } else if (index === 2 && !done1 && done2) {
                    request1(false, nextValue(1, cb)) // 2 done, try 1
                } else cb(false, v) // not done, return value
            }
        }
        return function output (abort, cb) {
            if (abort) {
                if (done1 && done2) cb(true, undefined)
                if (!done1) request1(abort, wait(1, cb))
                if (!done2) request2(abort, wait(2, cb))
            } else {
                if (done1 && done2) {
                    cb(true, undefined)
                } else if (done1 && !done2) {
                    request2(false, nextValue(2, cb))
                } else if (!done1 && done2) {
                    request1(false, nextValue(1, cb))
                } else if (next === 1) { // round-robin
                    next = 2; request1(false, nextValue(1, cb))
                } else {
                    next = 1; request2(false, nextValue(2, cb))
                }
            }
        }
    }
}

// usage
pull(merger(source(10), source(2)), sink)

Fig. 4.14 Pull-Stream Merger Implementation.
form an implicit state machine. However, for more complicated modules, the resulting behaviour can be hard to understand and assess for correctness by reading the JavaScript source code alone. For example, the StreamLender implementation supports a dynamically varying number of concurrent sub-streams and therefore the number of possible states grows exponentially with the number of active streams. Moreover, the resulting implementation requires around 400 lines of JavaScript code [109], which is quite cumbersome to explain in a clear and concise manner.

To simplify the presentation of the implementation of the Limiter, StreamLender, and DistributedMap abstractions that make up Pando, we therefore first present, in the next section, a notation to make that complexity more manageable.

### 4.2 Declarative Concurrent Notation

In this section, we present a declarative concurrent notation, at a higher-level than JavaScript, that relies on events happening at the interface of modules. We then use that notation in the next sections as pseudo-code to present all the core algorithms of Pando.

The main insight behind our choice is that both function invocations, such as invoking request(false, cb), within a single process and the reception of messages on communication channels between multiple processes in a distributed algorithm, are the basic events that trigger behaviours in implementations.\(^1\) Therefore, the notation typically used to specify distributed algorithms [35] can be adapted to describe modules executing within the same process through a well-defined callback protocol. This insight is somewhat obvious in retrospective, since callbacks in JavaScript are also called event handlers. The main and less obvious benefit of using event handlers is that the presentation of algorithms can now decouple the callback protocol requests from their answers, which, as we shall see, is especially beneficial when an answer shall be triggered after and only once multiple events have occurred.

Our notation leverages the declarative concurrent programming model with a number of primitive operations that are otherwise not part of JavaScript and were inspired by the Oz kernel language semantics and primitives [182]. Our notation includes: (1) an index parameter and explicit cases to decouple the different events of the pull-stream protocol, (2) a wait synchronization primitive with implicit concurrent execution of event handlers, (3) explicit thread creation blocks, (4) first-order logic operators in combination with a reified

\(^1\)It is our understanding that the same analogy was used to implement the first object-oriented systems, with the exchange of messages happening between objects rather than modules. Remote procedure calls as a source of distributed events also shows the equivalence.
history of past events that happened at the interface of a module, and (5) some syntactic sugar to create and connect pull-stream modules. We cover the five in turn with examples of the JavaScript code patterns they abstract.

### 4.2.1 Event Notation

Our first primitive is the notation for events in the format \( \text{Origin: name(parameters)} \). \text{Origin} can be a reference to another module or process and helps distinguish between multiple modules, when one concurrent module is connected to many others. In practice, the origin is the callback interface through which it is connected with other modules. The \text{name} corresponds to the specific event that occurred. As previously listed in Table 4.1, there are three possible request events (\text{ask}, \text{abort}, and \text{fail}) as well as three possible answer events (\text{value}, \text{done}, and \text{error}) when referring to pull-stream events. The \text{parameters} serve two purposes. First, it is sometimes necessary to show the sequence in which events from the same origin happened, and distinguish between each. We therefore use an \text{index parameter} that denotes the position of the event in the stream, e.g. \text{ask}_1 happened before \text{fail}_2. Second, some events may carry additional values, such as an error \text{err}. In these cases, the additional values are written after the index parameter. Some examples are illustrated in Table 4.2 corresponding to some events that may happen on the \text{Merger} module example of Figure 4.13.

The use of those events in algorithms is illustrated in Algorithm 1. In the example, requests for \text{inputs} are initiated internally with a \text{trigger} and answers are obtained from another module with the upon event handler. The equivalent JavaScript implementation is given in Figure 4.15.

<table>
<thead>
<tr>
<th>Origin</th>
<th>Event</th>
<th>Request Index</th>
<th>Event Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input1</td>
<td>ask</td>
<td>1</td>
<td>\text{Input1:ask}(1)</td>
</tr>
<tr>
<td>Input2</td>
<td>abort</td>
<td>2</td>
<td>\text{Input2:abort}(2)</td>
</tr>
<tr>
<td>Output</td>
<td>fail</td>
<td>(i)</td>
<td>\text{Output:fail}(i, \text{err})</td>
</tr>
<tr>
<td>Input1</td>
<td>value</td>
<td>1</td>
<td>\text{Input1:value}(1, \text{v})</td>
</tr>
<tr>
<td>Input2</td>
<td>done</td>
<td>2</td>
<td>\text{Input2:done}(2)</td>
</tr>
<tr>
<td>Output</td>
<td>error</td>
<td>(j)</td>
<td>\text{Output:error}(j, \text{err})</td>
</tr>
</tbody>
</table>

Table 4.2 Pull-Stream Callback Protocol Events Examples in our Event Notation using the \text{Merger} Example of Figure 4.13.

Representing pull-stream events this way has three main advantages: (1) it makes the correspondence between a request and its answer explicit in the event signature by sharing
Algorithm 1 Pull-stream events for streaming values between modules

1: upon started ▷ Requests
2: trigger Input:ask<i> ▷ Request the i-th value
3: trigger Input:abort<i> ▷ Abort the stream normally
4: trigger Input:fail<i, err> ▷ Abort the stream with an error err
5:
6: upon Input:value<i,v> ▷ Answers
7: or Input:done<i> ▷ Obtained the i-th value
8: or Input:error<i, err> ▷ Stream done: no more values
9: continue ▷ Stream failed with error err
10:

```javascript
var input = ...

function started () {
  input(false, answer) // ask<i>
  input(true, answer) // abort<i>
  input(new Error(), answer) // fail<i, err>
}

function answer (done, v) {
  if (done == false) ... // value<i,v>
  else if (done == true) ... // done<i>
  else if (done instanceof Error) ... // error<i, err>
}
```

Fig. 4.15 Implicit Event Indexes and Coupled Event Handlers in JavaScript
the same index; this makes it easier to refer to either of them as data dependencies for synchronization using the wait primitive; (2) it decouples the various cases so they can all be expressed at the same level in the implementation rather than nested within the function that handles the request or answer; (3) it decouples the request and answer, which avoids the bookkeeping required to manage answer callbacks correctly (as will be illustrated shortly, in Figure 4.16).

### 4.2.2 Wait Synchronization

Our next primitive is a wait operation that blocks until one or all of possibly many events have happened. These events may represent state changes within a module or events happening at its interface. Both possibilities are illustrated in Algorithm 2.

**Algorithm 2 Wait primitive**

1. `wait Origin: event(parameters) or internal condition or ...`
   - Cedes control and resumes after at least one of the conditions is satisfied
2. `wait Origin: event(parameters) and internal condition and ...`
   - Cedes control and resumes after all conditions are satisfied

An example is provided in Algorithm 3 and the equivalent implementation in JavaScript is given in Figure 4.16. When implemented in JavaScript, the wait primitive requires a non-local translation: every source of events that may change the condition state needs to check if conditions are now satisfied and then make sure the continuation following the wait, as a function handler, is executed only once. Moreover, since the answer callbacks are passed as an argument to requests in the pull-stream callback protocol, they need to be stored for later execution if they cannot be executed right away.

The wait operations are used in combination with an implicit *inter-leaving* concurrent semantics [199, Section 4.1] for event handlers: each event handler executes until completion or the next wait operation. In the latter case, it will resume execution after the condition on its wait is satisfied. A handler can never be pre-empted, it must cooperatively cede the flow of control to other threads, by completing or waiting, for them to make progress. However, when multiple threads are ready, they may do so in any possible order, as illustrated in Algorithm 4.

In existing JavaScript frameworks, APIs for event handlers usually store multiple callbacks that depend on the same event in a queue and later execute them in the order they were added (first-in first-out). However, when depending on input-output, network, or random timing events the order in which callbacks are registered may change between different executions. So the programming model of our algorithms follows the more general model
Algorithm 3 Wait Primitive Example

1: \( n \leftarrow 0 \)
2: \( \text{cancelled} \leftarrow \text{false} \)
3: 
4: \textbf{upon} Output:ask\( (i) \)
5: \( \text{wait} n > 2 \) \textbf{or} \( \text{cancelled} = \text{true} \)
6: \textbf{trigger} Output:done\( (i) \)
7: 
8: \textbf{upon} :inc\( () \)
9: \( n \leftarrow n + 1 \)
10: 
11: \textbf{upon} :cancel\( () \)
12: \( \text{cancelled} \leftarrow \text{true} \)
13: 

```
var n = 0
var cancelled = false
var answer = null
var done = false

function waitContinuation() {
  if (done || answer === null) return
  else if (n > 2 || cancelled === true) {
    done = true
    answer(true) // done<i>
  }
}

function inc () {
  n++
  waitContinuation()
}

function cancel () {
  cancelled = true
  waitContinuation()
}

function output (abort, _answer) {
  if (abort === false) { // ask<i>
    answer = _answer
    waitContinuation()
  } else {
    ...
  }
}
```

Fig. 4.16 Wait Primitive Example in JavaScript
in which threads may execute in any order. The `wait` primitive essentially serves to restricting the order such that it follows the resolution of *data dependencies*. This follows the dataflow programming model, which simplifies reasoning about correctness in a concurrent setting [199].

**Algorithm 4** Concurrent event handlers

1: **upon** started
2:   
3:       **wait** internal condition c  \(\triangleright\) Cedes execution control
4:       **trigger** `Internal::event1()`
5:  \(\triangleright\) May happen before or after `event2`
6:  
7:   **upon** internal condition c
8:     
9:        **trigger** `Internal::event2()`
10:    \(\triangleright\) May happen before or after `event1`

### 4.2.3 Explicit Thread Creation

Our next primitive enables the creation of *explicit threads*, in contrast to the implicit threads of the JavaScript event handlers. A new thread is specified with the `thread` keyword, as illustrated in Algorithm 5. The thread is actually *created* when the corresponding `thread` block is executed. It is then scheduled for execution after the creator thread starts waiting or has completed.

The corresponding JavaScript implementation is illustrated in Figure 4.17. This example uses the global callback queue of JavaScript with `setImmediate` because the thread does not depend on any data.

**Algorithm 5** Explicit thread creation

1: **upon** started
2:     **thread**
3:       **thread started**  \(\triangleright\) Starts executing after `done`
4:     
5:       **done**

### 4.2.4 First-order Logic Operators with a Reified History

Our next primitive enables reasoning over past events. To that end, we use the existential (\(\exists\)) quantifier from first-order logic and a reified history of the past, as illustrated in Algorithm 6.
Fig. 4.17 Thread Creation in JavaScript

Triggering or handling an event implicitly adds it to the set of past events, as shown explicitly in line 2 and 6.

Another option to reason over past events would have been to represent those events as state changes to local variables but doing so makes the dependency to external events harder to track, as shown in the equivalent JavaScript implementation of Figure 4.18. For presenting algorithms we therefore prefer referring directly to events. Note that for conciseness, in most cases, we use events as conditions without explicitly mentioning the past set, as in line 9. These forms are equivalent to the explicit notation.

**Algorithm 6** Reasoning about past events

1. \textbf{upon} \textit{started}  
2. \textit{past} $\leftarrow$ \textit{past} $\cup$ \{Input:abort\}(I) \hspace{1cm} \textgreater{} Implicit before trigger  
3. \textbf{trigger} Input:abort(I) \hspace{1cm} \textgreater{} Ask for a value from another pull-stream module  
4. \textbf{upon} Input:done(I) \hspace{1cm} \textgreater{} Received an answer  
5. \textit{past} $\leftarrow$ \textit{past} $\cup$ \{Input:done\}(I) \hspace{1cm} \textgreater{} Implicit after upon  
6. \textbf{if} \hspace{1cm} $\exists_i$ Input:abort\(\langle i \rangle \in \textit{past}$ \hspace{1cm} \textgreater{} Explicit past  
7. \hspace{1cm} ...  
8. \hspace{1cm} ...  
9. \hspace{1cm} $\exists_i$ Input:abort\(\langle i \rangle \) then \hspace{1cm} \textgreater{} Equivalent implicit past in conditional  
10. \hspace{1cm} ...  
11. \hspace{1cm} ...

**4.2.5 Pull-Stream Syntactic Sugar**

We finally use syntactic sugar for expressing the creation of pull-stream modules, with the create keyword, and their later connection together in a processing pipeline, with the $\Rightarrow$ operator. Both are illustrated in Algorithm 7 and their equivalent implementation in JavaScript is given in Figure 4.19.
4.2 Declarative Concurrent Notation

```javascript
var input = ...  
var aborted = false  
var done = false  
function started () {
  aborted = true  
  input(true, answer)  
}

function answer (_done, v) {
  if (_done === true) {
    done = true  
    if (aborted) ...
  }
}
```

Fig. 4.18 Reacting to Past Events in JavaScript

**Algorithm 7** Creating and connecting pull-stream modules

1: upon started
2: create Source Source
3: create Sink Sink
4: create Transformer Trans
5: create Duplex Out, In
6: Src ⇒ Trans ⇒ Sink ▷ Pipeline
7: Out ⇒ In ▷ Loopback Duplex

```javascript
function started () {
  function source (abort, answer) { ... }  
  function sink (input) { ... }  
  function transformer (input) {
    return function source (abort, answer) { ... } // eq. output
  }
  var duplex = { // object form
    input: function (source) { ... },  
    output: function source (abort, answer) { ... }
  }
  pull([source, transformer, sink])  
  pull([duplex.output, duplex.input])
}
```

Fig. 4.19 Creating and Connecting Pull-Stream Modules in JavaScript. Uses the pull definition of Figure 4.5.
The rest of the notation and operations we use is standard, of which a complete exposition has been done by Cachin et al. [35]. With the previous definitions behind, we are therefore ready to present the Limiter, StreamLender, and DistributedMap abstractions of Pando.
4.3 Limiter

A Limiter encapsulates another duplex or transformer module to bound the number of input values that are allowed to be requested before the corresponding outputs are returned. We use it in Pando to regulate the flow of values through WebSocket or WebRTC duplex channels because the pull-stream libraries for both protocols eagerly read all available values. Without the Limiter, one duplex connection to a single volunteer would prevent other connected volunteers to obtain values to process. By using the Limiter, values are still eagerly read but by no more than a bounded number which leaves the others for other volunteers. With a limit larger than 1, a limiter will allow an initial batch of values to quickly go through and then will adaptively let new values through at the rate at which results are produced.

The next two sections provide first a detailed definition and then an implementation that fulfills the definition.

4.3.1 Definition

The Limiter abstraction (Abstraction 1) can be used both with duplex channels and transformers because they share the same interface, i.e. they both have an input and an output that follow the pull-stream protocol. On instantiation, a non-limited pull-stream transformer or duplex channel is passed as argument and a new limited version of the same module is returned. Requests made by the encapsulated non-limited module are delayed until the number of requested values that have no corresponding output is less than $n$. It therefore bounds how eager a module can be. Moreover, the same mechanism also provides flow-control based on the rate at which a module outputs values. That limits the rate at which a module may request inputs to its processing speed. The limit on the number of values $n$ is set at the moment the module is instantiated and it can be updated dynamically after a processing pipeline has started, by using the updateLimit request. The abstraction assumes that the encapsulated module correctly follows the pull-stream protocol and returns exactly one result per value provided, otherwise its behaviour is undefined. Figure 4.20 shows how the JavaScript implementation of the Limiter can be used, with an explanation of each operation in the comments.

4.3.2 Implementation

The implementation of Limiter is illustrated in Figure 4.21. The non-limited module with input NLInput and output NLOutput is encapsulated by a limited module with input Input and output Output. The index of requests and answers on Output and Input are not correlated
Abstraction 1 Limiter Definition

Parameters

*NonLimited*: transformer or duplex channel

*n*: (optional) initial limit, defaults to 1

Returns

*Limited*: transformer or duplex channel

Requests

*Limited*: `updateLimit(n)`: Update the number of values *limited* can request to *n*

Properties

1. *Bounded Eagerness*: Delays *Limited* request for a value until the total number of values in process by *NonLimited* (requested but not output) is less than *n*.

2. *Output-based Flow-Control*: After the limit *n* is reached, the rate at which inputs are requested by *Limited* is bounded by the rate at which *NonLimited* produces results.

3. *Dynamic*: The limit *n* can be updated during execution.

Expectations


2. *No Duplication*: *NonLimited* returns exactly one result per value provided.

```javascript
var limiter = require('pull-limit') // Limiter library
var pull = require('pull-stream') // Pull-stream library
var duplex = require('websocket')(/* init */) // Duplex channel
var limited = limiter(duplex, 1), // Limit to one value at a time
    pull() // Create a pull-stream pipeline
        pull.count(10), // Source that counts from 0 to 10
        limited, // Limited duplex channel
        pull.drain() // Sink that requests all results
limited.updateLimit(2) // Dynamically update the limit to 2
```

Fig. 4.20 Limiter Usage in JavaScript
and are therefore represented by two different indices $i$ and $j$. The presentation of the implementation is split in two parts, the first part for the outputs and the second part for the inputs.

![Diagram of Limiter Inputs and Outputs](image)

**Fig. 4.21 Limiter Inputs and Outputs.** The *NonLimited* module in gray is external to the implementation.

In Part 1 (Algorithm 8), we first provide the definitions of the different inputs and outputs as well as the two local variables $nLimit$, to track the current limit, and $k$ to track the number of values currently processed by *NonLimited*. The rest of the implementation essentially propagates the requests between *Output* and *NLOutput* in one direction as well as the answers between *NLOutput* and *Output* in the other direction, while decreasing the number of values processed once a value is output (line 21).

In Part 2 (Algorithm 9), we show the corresponding propagation of input requests from *NLInput* to *Input*, as well as the propagation of input answers from *Input* to *NLInput*. The core of the abstraction is implemented starting from line 30. When an ask request is made then a check is made to see if the limit has been reached, which may have happened for two reasons: either $k$, the number of values requested, has reached the limit $nLimit$ or $nLimit$ has decreased to or below $k$. In both cases, the ask is delayed until either the limit is now over $k$ or *NonLimited* makes another request to terminate the stream normally (abort) or abnormally (fail). If the stream is not terminated $k$ is incremented and a new ask is triggered. Otherwise, the stream is terminated upstream by triggering abort. To maintain the invariant that every request receives an answer, a second done answer on *NLInput* is triggered (line 39, the first answer done will be triggered from line 51 or line 54).

The implementation illustrates how, even if the pull-stream protocol may seem simple at first, proper handling of error conditions in all cases is non-trivial and represents much of the complexity in implementing correct modules.

---

2We use $i$ on the output, which comes before $j$ in the alphabet, because every request on the input has necessarily been preceded by a request on the output.
Algorithm 8 Limiter Implementation Part 1 (Outputs)

1: Limited input is Input
2: Limited output is Output
3: NonLimited input is NLInput
4: NonLimited output is NLOutput
5:
6: \( n \text{Limit} \leftarrow n \) if defined, 1 otherwise \( \triangleright \) Current limit
7: \( k \leftarrow 0 \) \( \triangleright \) Current number of values being processed
8: upon Limited: updateLimit\( \langle n \rangle \)
9: \( n \text{Limit} \leftarrow n \)
10:
11: upon Output: ask\( \langle i \rangle \) \( \triangleright \) Pass-through the requests on Limited to NonLimited
12: \( \text{trigger NLOutput: ask}\langle i \rangle \)
13:
14: upon Output: abort\( \langle i \rangle \)
15: \( \text{trigger NLOutput: abort}\langle i \rangle \)
16:
17: upon Output: fail\( \langle i, \text{err} \rangle \)
18: \( \text{trigger NLOutput: fail}\langle i, \text{err} \rangle \)
19:
20: upon NLOutput: value\( \langle i, v \rangle \) \( \triangleright \) Idem for answers from NonLimited to Limited
21: \( k \leftarrow k - 1 \)
22: \( \text{trigger Output: value}\langle i, v \rangle \)
23:
24: upon NLOutput: done\( \langle i \rangle \)
25: \( \text{trigger Output: done}\langle i \rangle \)
26:
27: upon NLOutput: error\( \langle i, \text{err} \rangle \)
28: \( \text{trigger Output: error}\langle i, \text{err} \rangle \)
29:
4.3 Limiter

Algorithm 9 Limiter Implementation Part 2 (Inputs)

30: \textbf{upon} \texttt{NLInput:ask}(j) \quad \triangleright \text{Pass the requests from } \textit{NonLimited} \text{ to } \textit{Limited}
31: \quad \textbf{if} \; k \geq nLimit \; \textbf{then}
32: \quad \quad \textbf{wait} \; k < nLimit \; \textbf{or} \; \texttt{NLInput:abort}(j+1) \; \textbf{or} \; \texttt{NLInput:fail}(j+1, err)
33: \quad \quad \textbf{if} \; \texttt{NLInput:abort}(j+1) \notin \text{ past } \textbf{and} \; \texttt{NLInput:fail}(j+1, err) \notin \text{ past} \; \textbf{then}
34: \quad \quad \quad k \leftarrow k + 1
35: \quad \quad \quad \textbf{trigger} \; \texttt{Input:ask}(j)
36: \quad \quad \textbf{else}
37: \quad \quad \quad \textbf{trigger} \; \texttt{Input:abort}(j)
38: \quad \quad \quad \textbf{wait} \; \texttt{NLInput:done}(j) \; \textbf{or} \; \texttt{NLInput:error}(j, err)
39: \quad \quad \quad \textbf{trigger} \; \texttt{NLInput:done}(j+1)
40: \quad
41: \textbf{upon} \; \texttt{NLInput:abort}(j)
42: \quad \textbf{trigger} \; \texttt{Input:abort}(j)
43: \quad
44: \textbf{upon} \; \texttt{NLInput:fail}(j, err)
45: \quad \textbf{trigger} \; \texttt{Input:fail}(j, err)
46: \quad
47: \textbf{upon} \; \texttt{Input:value}(j,v) \quad \triangleright \text{Pass the answers from } \textit{NonLimited} \text{ to } \textit{Limited}
48: \quad \textbf{trigger} \; \texttt{NLInput:value}(j,v)
49: \quad
50: \textbf{upon} \; \texttt{Input:done}(j)
51: \quad \textbf{trigger} \; \texttt{NLInput:done}(j)
52: \quad
53: \textbf{upon} \; \texttt{Input:error}(j, err)
54: \quad \textbf{trigger} \; \texttt{NLInput:error}(j, err)
55: \quad
4.4 StreamLender

A StreamLender is a transformer that enables values from a single stream to be continuously borrowed by multiple concurrent sub-streams for processing. Sub-streams borrow values, process them, and return results to the original stream. We use it in Pando to enable concurrent processing by multiple volunteers. This is the core algorithm of Pando that provides the key properties of the programming model of Section 3.3.

4.4.1 Definition

The StreamLender (Abstraction 2) is instantiated with no parameter and returns a transformer. It may then be connected in a pull-stream pipeline. At any time, a new sub-stream may be instantiated with lendStream that returns a duplex channel. That duplex channel may be connected with another transformer to process values or connected to another duplex channel to send the values elsewhere. The abstraction provides a combination of interesting properties. We comment on each of unbounded, dynamic, lazy, conservative, fault-tolerant, ordered and adaptive in turn.

The number of sub-streams that can be created is unbounded: as long as the process executing StreamLender has enough memory available, it can coordinate an arbitrary high number of sub-streams. This gives a maximum flexibility in using the abstraction while making no assumption on the resources of the process that executes it.

The sub-streams may be created dynamically, at any time during execution, regardless of the number or the state of other sub-streams.

The StreamLender transformer itself is lazy, it does not request values on its Input unless a sub-stream has first requested one. This follows the common behaviour of pull-stream modules and simplifies working with infinite streams.

A value is conservatively lent to at most one sub-stream at a time until that sub-stream either provides a result or fails. This maximizes throughput when the probability of failure of sub-streams is low as is the case with Pando.

The StreamLender transformer is fault-tolerant: even if individual sub-streams may fail, as long as at least one of them continues making progress, all results should eventually be provided. Fault-tolerance is necessary for distributed processing as volunteers may leave at any time, intentionally or because of a communication channel failure.

The results are ordered, with the same order as the corresponding input values. This is a more general property than being unordered: applications that do not require results to be ordered should still be correct in cases where the order is preserved but the converse is not
Abstraction 2 StreamLender Definition

Returns

StreamLender: transformer with input Input and output Output

Requests

StreamLender: lendStream() returns Duplex $S_k O$, $S_k I$ which represents the sub-stream $S_k$ with output $S_k O$ (with values flowing from StreamLender to $S_k$) and input $S_k I$ (with values flowing from $S_k$ to StreamLender). $S_k O$ and $S_k I$ may be connected to a transformer or another duplex channel.

Properties

1. Unbounded: There is no upper bound on the number of sub-streams created by lendStream.
2. Dynamic: A new concurrent sub-stream may be created at anytime by calling lendStream.
3. Lazy: Each value requested on StreamLender’s Input has been preceded by a sub-stream $S_k$ requesting a value on $S_k O$.
4. Conservative: A value on Input is lent to at most one sub-stream at a time.
5. Fault-tolerant: As long as at least one sub-stream is correct and making progress, all values provided to Input should eventually have a corresponding result produced on Output even if individual sub-streams fail (done or error before producing all their results for the values they were lent).
6. Ordered: All results on Output are produced in the order of the corresponding values provided to Input.
7. Adaptive: Sub-streams that request values faster receive more of them.

Expectations

1. Well-behaved: All sub-streams $S_k$ correctly follow the pull-stream protocol.
2. Live: When StreamLender requests a value through $S_k I$, the sub-stream eventually either returns a value or terminates (done or error).
3. No Duplication: $S_k I$ obtains exactly one result per value provided by $S_k O$.
4. Ordered: All $S_k I$ results are obtained in the order of the corresponding values provided by $S_k O$. 
true. This increases latency in some cases compared to an unordered version as results that are already computed will be held until all previous results have been returned.

Sub-streams receive values at a rate that is proportional to the rate at which they ask for them. StreamLender therefore allows a rate that is adapted to the processing rate of individual sub-streams. This is in contrast to round-robin or other equal schemes in which every sub-stream would receive values at the same rate. A non-adaptive version would in effect make all sub-streams be limited by the rate of the slowest.

This combination of properties has turned out to be sufficient and quite useful for Pando. Its usage in JavaScript is illustrated in Figure 4.22.

```javascript
var pull = require('pull-stream') // Pull-stream library
var limit = require('pull-limit') // Limiter library
var duplex = limit(require('websocket')/* init */) // Limited duplex channel
var lender = require('pull-lend-stream') // StreamLender library
pull(      // Create a pull-stream pipeline
  pull.count(10), // Source that counts from 0 to 10
  lender,
  pull.drain() // Sink that requests all results
)

lender.lendStream(function (err, subStream) {
  if (err) return
  pull(
    subStream,
    duplex, // Remote processing
    subStream
  )
})

lender.lendStream(function (err, subStream) {
  if (err) return
  pull(
    subStream,
    pull.map(function (x, cb) { // Local processing
      cb(null, x*x)
    }),
    subStream
  )
})
```

Fig. 4.22 StreamLender Usage in JavaScript. Compared to the interface of Abstraction 2, lendStream returns an extra err argument in case a sub-stream with no values to process would be created, to avoid creating the other objects for nothing.
4.4.2 Implementation

The implementation of StreamLender is illustrated in Figure 4.23. Instantiating the abstraction provides a StreamLender transformer module with an Input and an Output. Multiple lendStream requests instantiate multiple sub-streams, which can be connected to an external transformer $T_k$, as shown in the figure, or a duplex channel (not shown). Each sub-stream is itself a duplex channel composed of an output $S_kO$, that provides values from Input to the sub-stream, and an input $S_kI$ that obtains the results from $T_k$. Input and Output are not directly correlated and have different indices $j$ and $i$ for their respective requests and answers: as many values from Input will be read as requested by all sub-streams regardless of the number of requests made on Output. Likewise, results are only provided on Output once sub-streams produce results for $S_kI$. A single sub-stream may request an unbounded number of values before producing a result, its output and input are therefore also decorrelated. Correspondingly, we respectively use indices $n$ and $m$ to refer to specific requests and answers. The presentation of the implementation is split in four parts: the first part for the Output, the second to explain the creation of sub-streams and requesting new values on Input, the third for helper procedures, and the fourth to serialize the requests created by concurrent sub-streams.

![Fig. 4.23 StreamLender Inputs and Outputs. The transformers/duplexes ($T_i$) in the sub-streams in gray are external to the implementation.](image)

In Part 1 (Algorithm 10), we first give definitions for Input and Output as well as global data structures and variables: results is an array that is used to reorder results and to synchronize the output of StreamLender with the results computed by the sub-streams, failed is a set of values for which a sub-stream failed before producing a result, $k$ is the number of
sub-streams created and is used as an index into each, and $j$ is the index of the last request made on $Input$. The first event handler (starting at line 7) provides results or a termination condition depending on the conditions of $Input$. The second event handler (starting at line 20) handles the early termination: it first aborts the input, waits for its answer and then confirms the termination after having waited for the last pending answer, to ensure they have all been received.

**Algorithm 10 StreamLender Implementation Part 1 (Outputs)**

1. `StreamLender` input is $Input$  
2. `StreamLender` output is $Output$  
3. $results \leftarrow \emptyset$  
4. $failed \leftarrow \emptyset$  
5. $k \leftarrow 0$  
6. $j \leftarrow 0$  
7. **upon** $Output$ : ask($i$)  
   8. \quad \text{wait until } results[i] \text{ is bound or } \exists j' \text{ Input: } done(j') \text{ or } \exists j', \text{err} \text{ Input: } error(j', \text{err})  
   9. \quad \text{if } \exists j, \text{err} \text{ Input: } error(j', \text{err}) \text{ then}  
      10. \quad \quad \text{trigger } Output: \text{error}(i, \text{err})  
   11. \quad \text{else if } \exists j' \text{ Input: } done(j') \text{ then}  
      12. \quad \quad \text{if } i < j' \text{ then}  
      13. \quad \quad \quad \text{wait until } results[i] \text{ is bound}  
      14. \quad \quad \quad \text{trigger } Output: \text{value}(i, results[i])  
      15. \quad \quad \text{else}  
      16. \quad \quad \quad \text{trigger } Output: \text{done}(i)  
      17. \quad \text{else}  
      18. \quad \quad \text{trigger } Output: \text{value}(i, results[i])  
   19.  
20. **upon** $Output$ : abort($i$) \text{ or } $Output$ : error($i$, err)  
21. \quad j \leftarrow j + 1; j' \leftarrow j  
22. \quad \text{Ensure the index does not vary while waiting}  
23. \quad \text{trigger } Input: \text{abort}(j')  
24. \quad \text{wait until } Input: \text{done}(j') \text{ or } \exists_{err'} \text{ Input: } error(j', err')  
25. \quad \text{if } i > 1 \text{ then}  
      26. \quad \quad \text{Answers must be returned in order}  
      27. \quad \quad \quad \text{wait until } Output: \text{value}(i - 1, results[i - 1]) \text{ or } Output: \text{done}(i - 1)  
      28. \quad \quad \quad \text{or } \exists_{err''} \text{ Output: } error(i - 1, err'')  
      29. \quad \quad \quad \text{trigger } Output: \text{done}(i)  

In Part 2 (Algorithm 11), we provide the core of the implementation of StreamLender: the creation of sub-streams as well as how values flow in and results flow out. The implementation uses two helper functions that are defined later in Part 3. Invoking `lendStream` creates a duplex channel for the sub-stream with specific handlers. Each sub-stream remembers values
that are being processed to reassign them to another sub-stream in case of failure. When a sub-stream asks for a value (line 34), one of three things happen. In the first case, if there are failed values, the oldest will be used. In the second case, if there are no more inputs, the sub-stream will wait for all results to be produced by other sub-streams so that if another one fails before being done, it can carry over. In the third case, it asks for a new value from Input, which may either provide a new one or complete, similar to the second case. The rest of the implementation is relatively straight-forward and explained in the comments.

In Part 3 (Algorithm 12) we provide the missing helpers from Part 2. WaitOnOthers ensures all results have been obtained before returning because until so, sub-streams may still fail. In case of failure, it reassigns the failed value to another sub-stream. RecoverFailedValues makes a sub-stream’s values for which no result was produced available again for processing by other sub-streams.

The implementation provided so far for StreamLender violates invariant 5 of the pull-stream protocol (Section 4.1.2) because it allows concurrent ask requests to be made. We factorize the serialization into its own Serializer algorithm in Part 4 (Algorithm 13). The implementation is mostly straight-forward: ask starting from the second are delayed until an answer has been provided on Output, itself preceded by an answer on Input. However, once Input has terminated with done or error, there should be no more requests, because of invariant (1) (Section 4.1.2). Therefore an answer is provided on Output without making a request on Input. Last, all answers have to be provided in the same order as the requests, therefore the answer for a given request waits for the immediately preceding answer to have been provided, which transitively ensures that all previous requests have been answered.

We suggest implementing Serializer as a separate pull-stream module. This module can then be combined upstream of the rest of the StreamLender implementation using the composition of pull-stream modules (Figure 4.5) to maintain the invariants of the protocol. Having separate modules will ease debugging, increase readability of the source code, and facilitate correctness reasoning.
Algorithm 11 StreamLender Implementation Part 2 (Sub-streams and Input)

29: upon StreamLender:\lendStream{} \\
30: \hspace{1em} k \leftarrow k + 1 \quad \triangleright \text{Sub-stream index} \\
31: \hspace{1em} create Duplex S_k O, S_k I \quad \triangleright S_k O \text{ provides values and } S_k I \text{ reads results} \\
32: \hspace{1em} n \leftarrow 0 \quad \triangleright \text{Index of } S_k I \text{ requests} \\
33: \hspace{1em} remembered \leftarrow \emptyset \quad \triangleright \text{Inputs with no result yet} \\
34: upon S_k O: \ask(m) \\
35: \hspace{1em} if failed is not \emptyset then \quad \triangleright \text{Take the input of another failed sub-stream} \\
36: \hspace{2em} (j', v) \text{ be the oldest of } \failed \text{ (smallest } j') \\
37: \hspace{2em} remembered \leftarrow remembered \cup \{(j', m, v)\} \\
38: \hspace{2em} failed \leftarrow failed \setminus \{(j', v)\} \\
39: \hspace{2em} trigger S_k O: \value(m, v) \\
40: else if \exists j' Input: done(j') \text{ or } \exists j' Input: error(j', err) then \\
41: \hspace{2em} WAITONOTHERS(remembered, S_k O, m, j') \\
42: else \\
43: \hspace{1em} j \leftarrow j + 1; j' \leftarrow j \quad \triangleright \text{Ensure the index does not vary while waiting} \\
44: \hspace{2em} trigger Input: \ask(j') \quad \triangleright \text{Request the next value from } Input \\
45: \hspace{2em} wait until Input: value(j', v) \text{ or } Input: done(j') \text{ or } Input: error(j', err) \\
46: if Input: value(j', v) then \\
47: \hspace{2em} remembered \leftarrow remembered \cup \{(j', m, v)\} \\
48: \hspace{2em} trigger S_k O: \value(m, v) \\
49: else \\
50: \hspace{2em} WAITONOTHERS(remembered, S_k O, m, j') \\
51: thread \quad \triangleright \text{Pull the sub-stream values until done} \\
52: \hspace{1em} while \exists n S_k I: done(n) \text{ and } \exists n, err S_k I: error(n, err) \text{ do} \\
53: \hspace{2em} n \leftarrow n + 1 \quad \triangleright \text{Request index} \\
54: \hspace{2em} trigger S_k I: \ask(n) \\
55: \hspace{2em} wait until \exists v S_k I: value(n, v) \text{ or } S_k I: done(n) \text{ or } \exists err S_k I: error(n, err) \\
56: \hspace{n} upon S_k I: \value(n, v) \quad \triangleright \text{Prepare results for output} \\
57: \hspace{2em} if \exists (i, n, v) \in remembered then \\
58: \hspace{3em} remembered \leftarrow remembered \setminus \{(i, n, v)\} \\
59: \hspace{3em} results[i] \leftarrow v \\
60: \hspace{2em} upon S_k I: done(n) \text{ or } S_k I: error(n, err) \quad \triangleright \text{Handle failures} \\
61: \hspace{2em} \text{RECOVERFAILEDVALUES(remembered)} \\
62: \hspace{2em} upon or S_k O: abort(m) \text{ or } S_k O: fail(m, err) \\
63: \hspace{2em} \text{RECOVERFAILEDVALUES(remembered)} \\
64: \hspace{2em} trigger S_k O: done(m) \\
65: return Duplex S_k O, S_k I
Algorithm 12 StreamLender Implementation Part 3 ( Helpers for Synchronization on Termination and Failure-Handling)

72: **procedure** \( \text{WAIT\_ON\_OTHERS} \) (\( \text{remembered,} S_k O, m, j' \))
73: \begin{align*}
\text{wait until } & \text{result}[j' - 1] \text{ is bound or } \exists f, v \in \text{failed} \\
\text{if } & \text{result}[j' - 1] \text{ is bound } \text{then} \quad \triangleright \text{All results in} \\
\text{trigger } & S_k O: \text{done}(m) \\
\text{else} & \quad \triangleright \text{Another sub-stream failed for value at index } f \\
\text{remember} & \text{remembered} \leftarrow \text{remembered} \cup \{(f, m, v)\} \quad \triangleright \text{Process the failed value} \\
\text{failed} & \leftarrow \text{failed} \setminus \{(f, v)\} \\
\text{trigger } & S_k O: \text{value}(m, v)
\end{align*}
80:

81: **procedure** \( \text{RECOVER\_FAILED\_VALUES} \) (\( \text{remembered} \))
82: \( \text{remembered}' \leftarrow \text{remembered} \)
83: \( \text{remembered} \leftarrow \emptyset \)
84: \begin{align*}
\text{for all } & (i, m, v) \text{ in } \text{remembered}' \text{ do} \\
\text{failed} & \leftarrow \text{failed} \cup \{(i, v)\}
\end{align*}
85:
Algorithm 13 StreamLender Implementation Part 4 (Serializer Transformer)

86: Serializer input is Input  ▷ Definitions
87: Serializer output is Output  ▷ To be connected to StreamLender’s Input
88: upon Output:ask\(i\)
89:  if \(i = 1\) then
90:      trigger Input:ask\(i\)
91:  else
92:      wait until \(\exists_v\) Output:value\((i - 1, v)\) or Output:done\((i - 1)\)
93:          or \(\exists_{err}\) Output:err\((i - 1, err)\)
94:      if Output:done\((i - 1)\) then
95:          trigger Output:done\(i\)
96:          else if \(\exists_{err}\) Output:err\((i - 1, err)\) then
97:              trigger Output:err\((i - 1, err)\)
98:          else
99:              trigger Input:ask\(i\)

100:
101: upon Output:abort\(i\)
102:    trigger Input:abort\(i\)
103:
104: upon Output:fail\((i, err)\)
105:    trigger Input:fail\((i, err)\)
106:
107: upon Input:value\((i, v)\)
108:    trigger Output:value\((i, v)\)
109:
110: upon Input:done\(i\)
111:    trigger Output:done\(i\)
112:
113: upon Input:err\((i, err)\)
114:    trigger Output:err\((i, err)\)
115:
4.5 DistributedMap

The DistributedMap abstraction combines the Limiter and StreamLender abstractions to enable the application of a function on a stream of values using the computing capabilities of a distributed set of volunteers. In the rest of this section, we provide the definition of DistributedMap (Section 4.5.1) and its implementation (Section 4.5.2).

4.5.1 Definition

The DistributedMap (Abstraction 3) is instantiated with a processing function \( F \) to apply to all values of the stream and a BatchSize to determine the maximum number of values to send per available processor. It returns a pull-stream transformer. It has a number of properties derived from those of the Limiter (Abstraction 1) and StreamLender (Abstraction 2). Those properties directly correspond to those of the two other abstractions, with an additional capability balancing.

The capability balancing is provided by the combination of StreamLender’s adaptivity and Limiter’s output-based flow control. The adaptivity allows faster sub-streams (faster processors) to pull more values, which removes the need for a push-based load-balancing heuristic. But if it were unbounded, a single greedy sub-stream will starve other sub-streams. So this is restricted with Limiter and parameterized with the BatchSize. After the limit of values has been reached, the Limiter ensures that subsequent values are sent at the same rate as the outputs are produced. This limits the input rate without any extra monitoring mechanism, simply by observing when outputs are produced. The combination of the two is simple and powerful since it is sufficient to adapt the flow of values to match the capabilities of the processors. The throughput of the entire system can then be maximized by choosing a BatchSize such that the fastest processors have enough work to be kept busy. Its value should also be no more than strictly necessary in order to minimize the memory consumption in StreamLender, because it remembers all values concurrently being processed for fault-tolerance.

The usage of DistributedMap in JavaScript is illustrated in Figure 7.4. Its interface is similar to the abstraction definition.

4.5.2 Implementation

The implementation is provided in two parts, the first part (Algorithm 14) executes on the Master process and the second part (Algorithm 15) executes in Worker processes on the same physical machine or on a separate device. In our JavaScript implementation [106], the
**Abstraction 3** DistributedMap Definition

**Parameters**

\( F \): Processing function to apply to all values

\( BatchSize \): Maximum number of values to send per processor

**Returns**

\( DistributedMap \): transformer with input \( Input \) and output \( Output \)

**Properties**

1. \( Unbounded \): There is no upper bound on the number of processing nodes (implementations of protocols such as WebRTC impose a practical limit).

2. \( Dynamic \): Processing nodes may join, leave, or crash-stop at any time.

3. \( Lazy \): Inputs are requested in proportion to the number of currently available processors. The flow rate of newer values sent is proportional to the flow rate of the results received.

4. \( Conservative \): An input is processed by at most one processing node at a time.

5. \( Fault-tolerant \): As long as at least one processing node is correct and keep making progress, all inputs should eventually have a corresponding result even if other processing nodes fail (done or error before producing all their results).

6. \( Ordered \): All results are provided in the order of the corresponding inputs.

7. \( Capability-balanced \): Faster processors receive more values to process.

```javascript
var f = function(x, cb) { cb(null, x*x) }
var mapper = distributedMap(f, { batchSize: 1 })
pull([
    pull.count(10),
    mapper,
    pull.drain()
])
mapper.on('url', function(err, url) {
    // Display URL to later open in a browser
})
```

Fig. 4.24 \emph{DistributedMap} Usage in JavaScript.
Master process is a Node.js commandline tool and the Worker processes execute in browser tabs (Section 3.4.3).

**Algorithm 14** DistributedMap Implementation (Master Process)

1: *DistributedMap* input is *Input* \[\text{Definitions}\]
2: *DistributedMap* output is *Output*
3: serve *F* on *Pando-Server*
4: display URL
5: listen for volunteer connections
6: *Lender* ← *StreamLender*()
7: *Input* ⇒ *Lender* ⇒ *Output*
8:
9: **upon** *DistributedMap::volunteerConnected*(duplex)
10: *limited* ← *Limiter*(duplex, *BatchSize*)
11: *SI,SO* ← *Lender::lendStream*()
12: (*SO* ⇒ *limited* ⇒ *SI*)
13:

**Algorithm 15** DistributedMap Volunteer Implementation (Worker Processes)

1: retrieve *F* from *Pando-Server*
2: open duplex channel to *DistributedMap* Master
3:
4: **upon** *Volunteer::connected*(duplex)
5: *Output*, *Input* ← duplex
6: *LocalStream* ← (*Input* ⇒ *AsyncMap*(F) ⇒ *Output*)
7:

The implementation of the Master and Workers essentially consists in connecting the Limiter and StreamLender to the communication channels. The Worker also uses an *AsyncMap* transformer, which applies the function *F* to every value of the stream. The implementation of AsyncMap is straightforward: it is an implementation of the function *map* operator in a pull-stream module; an implementation is provided in the Appendix.

The *F* and *BatchSize* values are communicated through the *Pando-Server*. Many different mechanisms are possible; in our implementation we use the HTTP protocol with JavaScript files created at the time *DistributedMap* is instantiated.

Notice how the error handling is implicit on the Master process: if a sub-stream fails because the duplex communication channel has been cut, the heartbeat mechanism of the latter will eventually detect the failure and will raise an error. This error will happen in the callback protocol and cause the termination of the sub-stream. StreamLender will then reallocate the unprocessed values to another sub-stream.
4.6 Run-time Verification of StreamLender with Random Test Generation

Even with a clear understanding of the pull-stream callback protocol and the behaviour of StreamLender, it is still challenging to ensure that an implementation of StreamLender is correct. In this section, we present the run-time verification strategy [57] that we designed and implemented to make it easier. This strategy helped us remove bugs that resulted in StreamLender not making progress, because some combinations of events were handled incorrectly and did not lead to an expected event, or did not follow the pull-stream protocol correctly, by generating events that should not have happened. Our approach did not require external tooling, like other approaches use [57], it was implemented only using pull-stream modules and a few custom algorithms.

4.6.1 High-Level Description

Our testing strategy is based on a step-by-step execution of StreamLender in a controlled pipeline. At every step of the execution, we have explicit control of actions that trigger StreamLender events. A step in the execution consists in executing one of the currently possible actions, then waiting for StreamLender to process it, and then picking the next possible action based on events generated by StreamLender during processing. An execution of a test consists in a sequence of those steps until no more events are possible or an execution error has been found.

We use a number of testing modules to implement a pipeline: a Source connected to the StreamLender input, a Sink connected to its output, and a dynamic number of Transformers connected in the middle of each of the sub-streams (Figure 4.23). The Source, Sink, and Transformer modules are implemented to externally control the events they would normally trigger: in our JavaScript implementation [115] these are implemented with function calls. This way, when connected to StreamLender, any event that would have originated externally to StreamLender, such as asking a value on StreamLender’s output from the Sink, is only triggered by an explicit function invocation. This is in contrast to the examples of Section 4.1 in which the pipeline was driven by a loop internal to the Sink and an automatic reaction of modules to callback events.

During execution, the pull-stream callback protocol invariants (Section 4.1.2) are checked to ensure they are maintained throughout the execution on every interface of StreamLender. To do so, we insert, between StreamLender and the Source, Sink, and Transformer modules, additional transformers modules that report an error if one of the invariants has been violated.
By using those checkers, if no error has been raised, it therefore means the execution was correct.

Four execution scenarios are possible, one of them is chosen randomly before the execution starts. All scenarios are parameterized by a \textit{count} of the number of values that should either be produced by the Source or read by the Sink. In the first scenario, the normal and most common execution case is simulated: the Source produces up to \textit{count} values, from 1 to \textit{count}, after which it terminates normally with a \textit{done} event. The Sink reads values until it receives a done. The second scenario is identical to the first, except that the Source will terminate with an \textit{error} instead of a done. In the third scenario, the Source produces an infinite number of values but the Sink will \textit{abort} after having received \textit{count} values. The fourth scenario is identical to the third, except the Sink will \textit{fail} with an error rather than aborting.

\subsection*{4.6.2 Inter-leavings}

The testing modules are then used by the test execution loop to generate \textit{inter-leavings}, i.e. sequences of events that are generated by multiple modules executing concurrently. At each step, the execution algorithm chooses the next event to generate. The list of valid events is determined by the pull-stream callback protocol. In reaction to one of these events, StreamLender may generate one or multiple events, that will then influence the valid events available from the testing modules. In effect, the universe of possible inter-leavings form a tree rooted in the initial state of StreamLender with inter-leavings being paths from the root to the leaves. The goal of the testing strategy is to cover as many branches as possible.

A partial execution of an inter-leaving is illustrated in Figure 4.25. Time is flowing from left to right and discrete events are represented as black dots. Each thread of execution has its own timeline, represented as a solid arrow. Following the single-threaded execution model of JavaScript with callbacks, threads cooperatively take turns to execute a single step then cede control to other threads: their executions are \textit{inter-leaved}. Discrete events therefore never happen at the same time and never align vertically. Events that have been triggered externally are in black, the first one of which is the asking for a value from StreamLender’s output (StreamLender: \textit{Output}:\textit{ask}(1) ). Events that have been triggered automatically by the implementation of StreamLender are in grey. The causal link between externally triggered events and StreamLender generated events is illustrated with dashed arrows.

In this particular example, StreamLender is first created. The Sink asks a value on StreamLender’s output. Then a first sub-stream is created with \texttt{lendStream}, which immediately triggers an \texttt{ask} from its output. This is followed by the creation of a second sub-stream with \texttt{lendStream} again, which also triggers an \texttt{ask} from the second sub-stream output. Then
the first sub-stream asks for a value, which triggers an ask on the input of StreamLender, that is received by the Source. The Source does not immediately answer. Meanwhile, the second sub-stream also asks for a value, which is implicitly queued. The Source then returns a value, that goes to the StreamLender Input and is finally passed to the Input of the first sub-stream. The pending request from the second sub-stream triggers a second ask from StreamLender Input which eventually gets a value from the Source. This value is passed to the second sub-stream. Meanwhile, the first sub-stream has processed its first value and has output a result, which gets output on StreamLender’s output as a value, etc.

![Fig. 4.25 StreamLender Inter-leaving Example.](image)

### 4.6.3 Random Selection of Next Action

The choice of which inter-leaving to execute is made randomly, by selecting at each step, one of the possible actions to trigger. The choices are made with a pseudo-random number generator, parameterized by a seed: passing the same seed will therefore result in the same execution. Selecting a specific action involves one or multiple random choices between multiple available alternatives. At the highest level, a first choice is made on which module to operate: the Source, the Sink, StreamLender, and one of the Transformers in the sub-streams. Only those with available actions are included in the random choice. If the Source or Sink
are selected, only one action is available for each, which is determined by the execution scenario and the current value count. If the StreamLender is selected, a new sub-stream will be created. A parallelism factor limits the number of parallel sub-streams that may be active at any time: after the limit is reached, lendStream won’t be appear in the choices until one of the sub-streams terminates. If the Transformers are selected, then a second random choice is made to select a particular sub-stream and possible action. This is the most complicated part of the execution so we will delve in a little more detail.

Sub-stream Transformers, that simulate the behaviour of remote processors, are implemented with two mostly independent state machines, one for the transformer Input, that receives new values to process, and one for the transformer Output, that returns the processed values to StreamLender. Since StreamLender does not place a limitation on the number of values that may be requested by a sub-stream, the Input can request values faster than the Output returns them (while there are values available). However, the Output can only return values after they have been obtained by the Input, so there is a dependency from the Output to the Input. In addition, the Input shall stop requesting values after the Output has received an abort or fail. There is therefore also a dependency from the Input to the Output. In order to make the matter easier to reason about, we present separately the state machine behaviour of both Input and Output, that implements the pull-stream callback protocol, from the truth tables that determine which actions are possible in any given state of the transformer.

The state machines for the Transformer’s Input and Output are illustrated in Figure 4.26. They are composed of four states and seven transitions based on the six possible events of the pull-stream callback protocol. The ready state is active when the module can take an action. The waiting state is active when the module is waiting for an external event. The terminating state is active when the module is in the process of terminating and the terminated state is active when the module has finished processing. The Input module starts in the ready state because it is the one initiating requests for inputs on the sub-stream. The Output module starts in the waiting state because it first needs a request from the sub-stream to return a value or terminate the stream. The normal sequence on the Input is that the Input will ask a value and waits. When it receives a value (which is stored as a pending value to output later), it goes back into the ready state for the next value. Similarly, in the normal case, the Output waits for an ask, after which it becomes ready. Then if there is a pending output, it returns the value, then wait again. If no value is pending, the Output keeps waiting until a new value is received by the Input. This continues on the Input and Output until there are no more values (done) or an error happens (error) after which the Input is terminated. Meanwhile, the Output will return values until both the Input is terminated and there are no more pending outputs, after which the next time it is ready it will answer done and also
move to the *terminated* state. When both the Input and Output are terminated the sub-stream is removed from the active sub-streams. In some cases, it is also possible for the sub-stream to be aborted from the Output, in case StreamLender is aborted early, or from the Input, if a network error breaks the communication. When this happens, either Input or Output becomes *terminating*. After that the Input will receive a confirmation of termination, i.e. done or error, and will be *terminated*. Similarly, the Output will explicitly confirm the termination with a done or error after which it is *terminated* too. To make things clearer, Figure 4.26 shows the actions that are triggered by the module in bold and those that triggered externally in a regular font.

![Test Transformer State Machines](image)

**Fig. 4.26 Test Transformer State Machines.** Events in bold are actions explicitly triggered by the test runner, events in regular font are automatically triggered by StreamLender.

The state of the Input and Output, and sometimes the number of pending output values, then serve to determine which actions are possible in the current step for sub-streams. Table 4.3 shows the truth table for determining possible actions on the Input, for which there are four cases depending on the Input’s state and the Output state. In the first and most common case, in which the Input is *ready* and the Output is either *ready* or *waiting*, the Input may either ask for a value or terminate normally (abort) or abnormally (fail). In the second case, when the Input is *waiting*, because there is a pending ask, and the Output is either *ready* or *waiting* too, because of the invariants of the callback protocol (Section 4.1.2), no more than a single value can be asked at a time. Therefore it is only possible to terminate with an abort or a fail. The same is true for the third case, when the Input is *ready* or *waiting* and the Output is *terminating* or *terminated*: the output has been terminated and is therefore not able to answer with values anymore so no more values should be asked. In the fourth case, whenever the Input is *terminating* or is *terminated*, regardless of the state of the Output, no more requests are possible.
4.6 Run-time Verification of StreamLender with Random Test Generation

<table>
<thead>
<tr>
<th>Input.state</th>
<th>Output.state</th>
<th>ask</th>
<th>abort</th>
<th>fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>ready</td>
<td>ready/waiting</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>waiting</td>
<td>ready/waiting</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>ready/waiting</td>
<td>terminating/terminated</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>terminating/terminated</td>
<td>——</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 4.3 Truth Table for Possible Actions on Test Transformer’s Input Depending on Active State. ‘——’ means all possible states. Possible actions are bolded for emphasis.

The truth table for the Output, shown in Table 4.4, is more complicated because it needs to take into account the disponibility of pending output values, in addition to the Input and Output states. In the first and common case, if the Output is ready, i.e. has received an ask request while previously been waiting and there are pending values, regardless of the state of the Input that value can be provided. Moreover, as long as there are available values to return, the Output will not terminate the stream to ensure they flow through. We could have added an additional case to distinguish between the Input being terminated because of an error to close the stream faster but that would have brought unnecessary complexity for little gain in our application, so we avoided it. In the second case, while there are no pending values and unless the Input is terminated, the Output cannot act. In the third case, if the Input is terminated however, no more values will come in therefore the Output can terminate with a done or an error. In all remaining combinations, for the fourth, fifth, and seventh cases, the Output cannot act because it is either waiting for a request, is terminating but the Input is not terminated, or both the Input and Output are terminated and there is nothing left to do. In the sixth case, while the Output is terminating, i.e. because it has been previously aborted, and the Input is now terminated, it can now terminate itself with done or error.

<table>
<thead>
<tr>
<th>Output.state</th>
<th>size(pendingValues)</th>
<th>Input.state</th>
<th>value</th>
<th>done</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>ready</td>
<td>&gt; 0</td>
<td>—</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>ready</td>
<td>0</td>
<td>ready/waiting/terminating</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>ready</td>
<td>0</td>
<td>terminated</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>waiting</td>
<td>—</td>
<td>—</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>terminating</td>
<td>—</td>
<td>ready/waiting/terminating</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>terminating</td>
<td>—</td>
<td>terminated</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>terminated</td>
<td>—</td>
<td>terminated</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 4.4 Truth Table for Possible Actions on Test Transformer’s Output Depending on Active State. ‘——’ means all possible states. Possible actions are bolded for emphasis.

While the state machines and truth table determine what action are possible in sub-streams in any given step, if we were to give equal probability to each possible action, this would
make the normal cases quite unlikely because there are many more failure conditions than normal conditions. Therefore, execution scenarios would not often end up with multiple parallel sub-streams, which is what we are ultimately trying to simulate, because each would fail too fast. We therefore bias the selection of actions for the normal and common cases, i.e. Input:ask, Output:value, and Output:done, to a 90% probability. In effect, this means that on any given step there is a 10% failure rate. In practice, this led to many cases generated that exhibit parallel sub-streams while making failures sufficiently frequent to make us confident in the level of coverage. In retrospect, the bias is somewhat arbitrary but turned out sufficient to catch most bugs that affected StreamLender’s concurrent behaviour. This completes the important aspects of action selection for all testing modules. We provide the algorithms that implement the modules and execution scenarios according to the previous description in Appendix A.

### 4.6.4 Benefits and Limitations

We have covered, in the previous explanation, how a single test execution is constructed. To debug StreamLender, we ran a large number of those executions until we found some that failed (and obtained the seed that generated the specific execution). We then repeated the same execution a number of times with the generating seed while fixing the bugs in the StreamLender implementation until the execution would be correct. We kept doing so three or four times until no error was found for a large number of tests. Our method being probabilistic, the larger the number of successful test cases, the higher we can be confident that the StreamLender’s implementation is correct.

This run-time verification strategy stress-tests StreamLender more efficiently than if we had relied on running the entire Pando tool: we avoid the setup and teardown time of the rest of the tool on each test case and we can tune the generation algorithm to more effectively search the space of possible execution inter-leavings in executions that would otherwise be hard to control, much less replicate. Moreover, a random generation approach is easier to parallelize than a systematic enumeration of all possible cases. This strategy is therefore better compatible with Pando: we can therefore use Pando to test its own correctness by running the tests on many personal devices in parallel, and therefore have included it in our example Applications (Chapter 5).

The limitation of a random strategy however is that it provides no guarantee that all cases have been covered, i.e. that all branches of the tree have been explored. The correctness is therefore only probable, although the level of probability can be made arbitrarily high by increasing the number of test cases generated. In practice, it turned out to be sufficient. The reliability of Pando after having fixed the bugs discovered with this testing strategy became
high enough that we never noticed problems in all our subsequent performance tests. Since each performance test is executed with a new scenario, we are fairly confident, given the large number of execution so far, that our testing strategy is effective and that our JavaScript StreamLender’s implementation is robust.

### 4.7 Discussion

This completes the presentation of Pando’s core abstractions and means to test them. Collectively, they are the main contributions of the design of Pando and correspond to the core behaviour that could be ported to other programming environments. Everything else should be relatively straight-forward to figure out with documentation on libraries and communication standards available.

Moreover, the main goal we had in choosing a notation to describe the key abstractions of Pando’s implementation, was to reduce as much as possible the accidental complexity that came from initially using JavaScript to implement the algorithms. This way we could distill their essence and make it easier for others to reimplement them in other languages. We believe we have succeeded both subjectively, because the presentation felt simpler and clearer to us, but also objectively, as the complete implementation of the `Limiter`, `StreamLender`, and `DistributedMap` abstractions is shorter by at least a factor of two compared to their JavaScript implementations [110, 109, 106].

### 4.8 Related Work

The inspiration for presenting the key aspects of Pando’s implementation as abstractions, determined by their interface and properties, and followed by the algorithms that implement them, came from Cachin et al. [35]’s textbook presentation of distributed algorithms. Our notation is also greatly inspired by theirs.

We are not aware of any other extensive description of abstractions and algorithms for volunteer computing. To the best of our knowledge we are the first to articulate and implement the StreamLender abstraction. Moreover, the existing publications focus on the high-level design (ex: BOINC [10]) and implementations details that are specific to some programming languages (ex: object-oriented Java in Bayanihan [175]). Moreover, compared to the execution assumptions in Cachin et al.’s book [35], in which the number of processes that execute the algorithm is known \textit{a priori}, our \textit{dynamic} setting in which volunteer’s devices may join at any time and therefore the number of sub-stream can be arbitrarily large, is more challenging. However, since the process on which StreamLender is executing is assumed
Implementation of Pando

to be correct, it is easier to implement fault-tolerance than it is when the algorithm runs on multiple processes and it is not known beforehand which are going to correctly execute.

Dominic Tarr has previously applied a run-time verification testing strategy for the *pull-many* pull-stream module [185], which combines multiple concurrent sources of values into a single stream, based on random testing. However, the assumed invariants of the pull-stream callback protocol had not been explicitly listed and the run-time verification approach had not been documented before. We have independently developed our own approach, which may share some similarities, and hopefully our more complete explanation should make the technique more accessible to other developers of pull-stream modules.

### 4.9 Summary

The pull-stream design pattern has a number of beneficial properties, summarized in Table 4.5, that we leverage to organize the implementation of Pando. We have shown that the pull-stream design pattern only depends on higher-order function and that it could equally easily be used in Python, making the abstractions designed around this pattern easily portable to other programming languages. However, the implementation of concurrent modules is still difficult both to implement and describe clearly.

We therefore presented a declarative concurrent notation, at a higher level than JavaScript, inspired by the declarative concurrent subset of the Oz programming language [199] and an existing notation used to describe concurrent algorithms [35]. This choice made the presentation of the algorithms that implement the *Limiter, StreamLender, and DistributedMap* abstractions clear and shorter than their JavaScript implementation, by at least halving the number of lines of code required to describe them. In addition, we have presented the interface and properties that help decompose the Pando implementation in well-defined modules and favours the reuse of the abstractions in other contexts. To the best of our knowledge, this is the first time such an effort has been made to describe abstractions useful for volunteer computing.

The most complex of the three modules, StreamLender, is still challenging to implement correctly, even with a clear description of its properties and behaviour. We therefore described a run-time verification testing strategy based on random test generation that helps find implementation bugs. The choice of a random test generation strategy enabled the parallel execution of a large number of tests, using Pando to test itself, to quickly find bugs and ensure a high probability of correctness. By applying this run-time verification strategy to our own JavaScript implementation, we have removed a number of bugs, in which some events that should have been triggered were not and some events that should not have been
### Property Definition

**Portability**  The implementation only requires higher-level functions and is therefore easy to reimplement in many popular programming languages of 2019.

**Flow-regulation**  The producer and consumer of values can both slow down the rate to their processing capabilities.

**Early-termination**  The consumer may abort the stream early before the producer has generated all its values.

**Graceful error handling**  Any module may propagate an error, and have an opportunity for cleanup after an error happened. The handling of errors is transparent to users of the pull-stream modules.

**Lazy generation**  Values are generated lazily, i.e. as they are requested. Therefore infinite streams can be produced and long running systems with real-time updates are supported.

**Declarative composable**  Users can assemble pipelines of modules without needing to understand the callback protocols on which pull-streams are based. Pairs that combine Source, Sink, and Transformers modules may be assembled prior to having a complete pipeline.

**Declarative concurrency**  The implementation of modules may use parallelism to improve the overall throughput. As long as the non-determinism is non-observable, downstream modules, sequential or parallel, are compatible with no extra effort.

**Fault-tolerance**  A module may recover from errors transparently when it is concurrently connected to multiple other modules that are failure prone.

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>Definition</td>
</tr>
<tr>
<td>Portability</td>
<td>The implementation only requires higher-level functions and is therefore easy to reimplement in many popular programming languages of 2019.</td>
</tr>
<tr>
<td>Flow-regulation</td>
<td>The producer and consumer of values can both slow down the rate to their processing capabilities.</td>
</tr>
<tr>
<td>Early-termination</td>
<td>The consumer may abort the stream early before the producer has generated all its values.</td>
</tr>
<tr>
<td>Graceful error handling</td>
<td>Any module may propagate an error, and have an opportunity for cleanup after an error happened. The handling of errors is transparent to users of the pull-stream modules.</td>
</tr>
<tr>
<td>Lazy generation</td>
<td>Values are generated lazily, i.e. as they are requested. Therefore infinite streams can be produced and long running systems with real-time updates are supported.</td>
</tr>
<tr>
<td>Declarative composable</td>
<td>Users can assemble pipelines of modules without needing to understand the callback protocols on which pull-streams are based. Pairs that combine Source, Sink, and Transformers modules may be assembled prior to having a complete pipeline.</td>
</tr>
<tr>
<td>Declarative concurrency</td>
<td>The implementation of modules may use parallelism to improve the overall throughput. As long as the non-determinism is non-observable, downstream modules, sequential or parallel, are compatible with no extra effort.</td>
</tr>
<tr>
<td>Fault-tolerance</td>
<td>A module may recover from errors transparently when it is concurrently connected to multiple other modules that are failure prone.</td>
</tr>
</tbody>
</table>

Table 4.5 Summary of the Pull-Stream Design Pattern Benefits.
were. This improved the reliability of the implementation enough that we never experienced bugs afterwards in our performance tests. A similar strategy had previously been used by Dominic Tarr for the implementation of the pull-many [185] implementation but had not been clearly described in detail. Our approach is the first detailed description of its application to the implementation of concurrent pull-stream modules and shall therefore make it easier for other developers of pull-stream modules to apply in the future.

The combined description of the pull-stream design pattern, the Limiter, StreamLender, and DistributedMap abstractions and the run-time verification strategy for StreamLender shall enable others to easily reimplement the design of Pando in other programming languages, making the design more resilient to technological evolution and faster to spread in other programming environments.
Chapter 5

Applications

Pando can be applied to a wide range of applications. In this Chapter, we present nine example applications regrouped according to their dataflow pattern, i.e. how data flows between Pando and other tools and protocols connected to it: pipeline processing (Section 5.1), synchronous parallel search (Section 5.2), and stubborn processing (Section 5.3). 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. For each application, we provide a description and the testing parameters that we used for the performance evaluation of Chapter 6. The full JavaScript source code for all applications is available in the online Pando handbook [105]. We finally end the chapter with a summary (Section 5.4).

5.1 Pipeline Processing

Pipeline processing is a sequence of independent processing stages applied to a stream of inputs, as illustrated in Figure 5.1. Traditional bag-of-tasks problems, typically associated with volunteer computing, can also be solved with this approach, by listing each individual task in sequence.

![Pipeline Processing Dataflow and Examples](image)

Fig. 5.1 Pipeline Processing Dataflow and Examples.
Table 5.1 Summary of Applications using Pipeline Processing.

<table>
<thead>
<tr>
<th>Application</th>
<th>Inputs</th>
<th>Pando</th>
<th>Post-Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collatz</td>
<td>Ints</td>
<td>Nb of steps</td>
<td>Max filter</td>
</tr>
<tr>
<td>Raytrace</td>
<td>Camera positions</td>
<td>Raytracing</td>
<td>Animated GIF</td>
</tr>
<tr>
<td>Arxiv</td>
<td>Meta-info</td>
<td>Manual tag</td>
<td>None</td>
</tr>
<tr>
<td>StreamLender random testing</td>
<td>RNG seeds</td>
<td>Inter-leaving</td>
<td>Failure monitoring</td>
</tr>
<tr>
<td>Machine learning agent</td>
<td>Hyperparams</td>
<td>Simulation</td>
<td>None</td>
</tr>
<tr>
<td>Image processing (HTTP distr.)</td>
<td>Landsat-8 images</td>
<td>Blur filter</td>
<td>None</td>
</tr>
</tbody>
</table>

Pipeline processing is straight-forward to use with Pando and easily compatible with the streaming paradigm of Unix tools, making distributed processing easy to integrate in custom Unix workflows. We implemented six applications that show diverse use cases, summarized in Table 5.1. We present each application separately in the next sections.

5.1.1 Collatz

Collatz implements the Collatz Conjecture [188], an ongoing BOINC project, to find an integer that results in the largest number of computation steps. Our first implementation was compiled from Matlab to JavaScript using the Matjuice compiler [65, 64]. However, because it used regular integers which limited the size of integers that could be tested, we manually adapted it to use a BigNumber library to be able to test integers at least as big as $3,572,582,862,843,521,162,235$, the integer that results in the largest number of steps as of July 2019 [124]. For reference, the resulting JavaScript implementation is given in Figure B.1.

The implementation follows easily from the formulation of the problem, with its core computation implemented in the `collatz` function. However, since testing a single integer is significantly faster than the time it takes to transmit the value to a Worker, the Pando function tests a range of integers and returns the largest number of steps from that range, with the corresponding interval.

By default, we use a range of 175 integers, which at the time of writing results on a processing time of approximately 1 second on a MacBook Air 2011. For tests on faster machines, we have sometimes increased the range to increase the processing time.

This example shows that other languages with a JavaScript compiler may therefore be distributed without having to implement a distribution strategy, and that Pando also works with existing volunteer computing tasks.
5.1.2 Raytrace

*Raytrace* distributes the rendering of individual frames of a 3D animation and assembles them in an animated gif, which we have already described in detail as a motivating example in Section 3.1.

The gif-encoder has limitations that prevent sending images that are too big (> 65kB) on the standard input, when connected to Pando. It also does not handle flow control properly when reading the same input from a file. The image size is therefore artificially limited to 300x300 pixels to operate within those limitations.

For tests, the number of individual frames in the animation is scaled up with the number of cores. Each image takes roughly one second of processing on a Macbook Air 2011 and a hundred images make a full rotation around the scene. When scaling up the number of frames beyond a hundred, the animation therefore makes several rotations.

This example suggests a similar strategy could be useful in open source animation tools for artists that do not have access to a rendering farm.

5.1.3 Arxiv

*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. Each volunteer is shown a different abstract to classify as "relevant" or "irrelevant" and Pando returns a stream of decisions regarding each abstract. An example of the user interface is illustrated in Figure 5.2. For reference, a simplified version of the JavaScript implementation, that only displays the summary of the paper without the authors, title, etc., is shown in Figure B.2.

We did not use this application in our performance evaluation because the "computation step" is actually performed by a human. It was therefore not comparable with the other tests and did not depend on the performance of the underlying machine. The example shows however that Pando can be applied beyond CPU-intensive tasks, and can actively involve humans in the processing step.

Other examples where this might also be beneficial are online rescue search using satellite or aerial images in times of disasters, quick surveys on current topics, etc.

5.1.4 StreamLender Random Testing

*StreamLender random testing* performs random executions of StreamLender to find cases where the invariants of the pull-stream protocol are violated. We have already presented the
algorithm in detail in Section 4.6. For reference, the JavaScript main processing function is shown in Figure B.3. The actual JavaScript implementation of the interleaving function is available online [115].

For our tests, each input generates 300 executions to ensure a processing time per input of around 1 second on a MacBook Air 2011. We generated enough test inputs to keep all workers busy for at least 5 minutes. Each execution also ran with a new random seed: this increases the probability that StreamLender is correct with every new performance test.

The same approach could be used to do test coverage on other critical software components. We could, for example, test the concurrent behaviour of the Secure-Scuttlebutt implementation, a protocol for decentralized social applications [178].

5.1.5 Machine Learning Agent

*Machine learning agent* searches for the optimal learning rate, a hyperparameter, that helps an autonomous agent in a simulated environment quickly learn sequences of steps that result in rewards. In this particular example, the training phase is interactive: the user can see the behaviour of the agent as it is learning and can early-abort a particular hyper-parameter case if the agent fails to learn, a form a *hybrid human-machine learning collaboration*. The
user interface, which shows an ongoing simulation and the interaction buttons, is shown in Figure 5.3.

![Figure 5.3](image)

Fig. 5.3 Machine Learning Agent Training.

The actual training of the agent is implemented with a variant of reinforcement learning [137] combined with experience replay [122]: the agent accumulates memories of sequences of actions and the rewards obtained in the past; at every step of the execution, it optimizes its own behaviour to select sequences of steps that maximize its rewards. The source code for the simulation and the training of the agent was taken from the ConvNetJS
demo application by Andrej Karpathy [104]. In the environment shown in Figure 5.3, the agent senses its environment through nine "rays" in front of it that indicate the position of a wall, toxic food in the form of green circles, and good food in the form of red circles. Toxic and good food randomly respawn over time after they are eaten. Over time the agent learns to explore its environment to find good food, while avoiding walls and toxic food. A simplified implementation of the main processing function is shown in Figure B.4. It shows both the use of a button to reject the current execution, and the automatic execution of the simulation for a pre-defined time interval.

With the right hyper-parameters, the learning agent typically takes 15 minutes to converge to an effective strategy. That was rather long for performance testing, so we instead ran each instance for 55 seconds: this was short enough to make the tests convenient, yet long enough to spend a significant amount of time in the CPU-bound phase of the execution to cover for the initial warm-up phase that is not.

Other machine learning tasks that require hyper-parameter optimization could also be compatible: artificial life, cellular automata, deep neural networks for image classification, etc.

5.1.6 Image Processing with HTTP Data Distribution

Image processing blurs the images from the open satellite dataset [173], using a gaussian filter [125], illustrated in Figure 5.4. Since images may be large (multi-megabytes) and may run into the limitations of some of the libraries we are using, we combine Pando with a minimal HTTP server to transfer the images in and out of the browser. Input image files are sent from a static file server to a participating browser through HTTP. Pando sends only the URL of the input image to a participant, the browser actually loads the image through a GET request. The browser then processes the image and sends the result through a POST request. After the POST succeeded, the result’s meta-information is sent to Pando which outputs it on the standard output. A simplified implementation is shown in Figure B.5.

Since the result meta-information is not submitted back to Pando unless the POST succeeded, we have the guarantee that once it has been received by Pando the result has been saved on the file system. Compared to the DAT and WebTorrent versions that will be explained in Section 5.3, this version removes the need for fault-tolerance in the transmission of data.

Each image takes 3-10 seconds of processing on a MacBook Air 2011. When testing with a large number of processors, we increased the number of images to process by taking several copies of each image, enough to keep the processors busy for at least 5 minutes.
Fig. 5.4 Image Processing.
Other kinds of image processing would be possible such as georectification of shore-based photography [32]: a camera on the ground takes pictures, which are later post-processed to reproject the pixels and interpolate missing values to obtain images equivalent to aerial photography, at a fraction of the cost.

5.2 Synchronous Parallel Search for Crypto-Currency Mining

The structure of blockchains in crypto-currencies such as Bitcoin [142] 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. For reference, a simplified implementation of the mining function is shown in Figure B.6.

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 5.5, 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.

For our tests, we have chosen a difficulty level that results in blocks being successfully mined every 1-10 seconds. This saturates the CPU as well as if we had chosen a higher difficulty level because it does not affect the rate at which hashing attempts are made. Our results should therefore generalize to higher difficulty levels.
5.3 Stubborn Processing with Failure-Prone External Data Distribution

In addition to the HTTP version of Section 5.1, we implemented two additional versions of distributed blurring of the Landsat-8 open satellite dataset [173]: one distributing the data with the DAT protocol [47], itself accessible in the Beaker browser [22], a fork of Chromium [40], and another that uses WebTorrent [207] running in browsers that support WebRTC. We briefly explain the common mechanism to both, then provide more detail on the specificities of each.

In both cases, managing data outside of Pando introduces an additional failure mode due to the *asynchronous* transmission of results: 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

<table>
<thead>
<tr>
<th>Application</th>
<th>Inputs</th>
<th>Monitor</th>
<th>Pando</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crypto-currency Mining</td>
<td>Blocks</td>
<td>Block + Range of Test Values</td>
<td>Mine nonce</td>
</tr>
</tbody>
</table>

Table 5.2 Synchronous Parallel Search Example: Crypto-currency Mining.

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 [140]. There may therefore be potential applications in mining those emerging crypto-currencies with Pando to support charities and fund open source software.
Table 5.3 Stubborn Processing Examples.

been factored into our new stubborn pull-stream module [117] which can be combined with sharing and downloading modules that are specific to a particular protocol, as illustrated in Figure 5.6.

![Fig. 5.6 Stubborn Processing with External Data Distribution Dataflow.](image)

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.

5.3.1 Image Processing with WebTorrent Data Distribution

WebTorrent [207] is a JavaScript implementation of the BitTorrent protocol [28]. Our implementation of Image Processing with WebTorrent shares input images with a client named webtorrent-hybrid [208], which runs in Node.js. The images are then loaded and processed in a browser through the webtorrent[207] library. Finally the results are transferred back and saved on the filesystem. The WebTorrent library for browsers relies on WebRTC, we have successfully tested it with Firefox and Chrome.

Our implementation seeds all input data from within the same torrent, which, of all approaches we tested, made the initialization the fastest. Still, the creation of a torrent with webtorrent-hybrid can take 5-8s in Node.js. Downloading a torrent is significantly faster if some of the open trackers are used to seed the torrent and download it with a magnet link (ex: wss://tracker.openwebtorrent.com). Presumably this is because they remove the need for a lookup through the DHT which itself requires WebRTC connections. Moreover, creating many torrents in the browser (ex: 7) can exhaust the number of concurrent WebRTC
connections allowed. It may also exhaust the maximum number of listeners used by event emitter internally by WebTorrent.

Nonetheless, the latency in loading a torrent can be quite high, even when the data is transferred between processes on the same machine. It can take tens of seconds and sometimes a few minutes before a torrent is resolved, the data is transferred, and the data is finally usable. Firefox 63.0.3 seemed slower at that than Chrome 71.0, possibly because establishing a WebRTC connection is slower. Moreover, in both cases, when a new torrent is created within the browser to transfer the result back, in some of our tests, it could take several minutes before the torrent was downloaded on the filesystem. The high and random latency for transferring data with WebTorrent currently makes it unsuitable for performance testing with Pando.

5.3.2 Image Processing with DAT Data Distribution

DAT [47] is a peer-to-peer protocol organized around mutable repositories of data, called archives, originally created for the archival and distribution of large scientific datasets. DAT has since grown to encompass a larger number of use cases. Its data distribution strategy, which disseminates updates to other copies of a repository, is similar to BitTorrent and WebTorrent. However, among other differentiating capabilities, DAT supports updating a repository with new versions by the original creator. At the time of writing, this protocol is only supported on Node.js and in the Beaker Browser [22] but we expect it should get more traction in the next years.

We use DAT with Pando to transfer images in and out of the browser. To transfer images in, a client executing natively with Node.js, first shares all the images in a single DAT archive. Then each Worker executing in the Beaker Browser downloads its input image by reading the corresponding file from the DAT repository. To transfer the results out, each Worker creates a new archive, adds the result to it, and then returns the URL of the corresponding file through Pando. The pull-stubborn module attempts a download of the result, and if successful, outputs the path to the result on the file system. Otherwise, it re-submits the input for re-computation by a different worker.

There were two impediments in using DAT with Pando. First, the protocol is only supported natively within the Beaker Browser, which prevented us from executing the application on mobile devices. Second, for security reasons, the Beaker Browser asks explicitly the user for permission to create a new DAT archive within a Web page. Since, we create one for every result, that quickly became cumbersome to run the tests. So while the performance of the DAT protocol was significantly better than that of WebTorrent, presumably
because it does not use WebRTC to establish the peer-to-peer connections, its lack of support and its security model made the resulting application not really useful.

Because both the WebTorrent and DAT versions had issues, we resorted to the HTTP version for Image Processing for our performance tests, which was both fast and easy to use. Nonetheless, the issues for both WebTorrent and DAT should be addressable so it would be worth revisit these use cases later.

5.4 Summary

We have presented nine application examples, Collatz, Raytrace, Arxiv, Stream Lender Random Testing, Machine Learning Agent, Crypto-currency Mining, and three different versions of Image Processing that all rely on a different external protocol for data distribution. These applications were regrouped according to their flow of data, in three families: pipeline processing, synchronous parallel search, and stubborn processing, which in the first case had the data flow linearly between multiple modules and in the two others implied some form of feedback loop that influenced which inputs are submitted to workers. The example applications covered many compute-intensive tasks but also hinted at the possibility of using Pando for crowdprocessing, in which the computation step is actually performed by a volunteer. We have also provided many suggestions of additional potential applications that would be similar to our examples. Together these examples show a large scope and variety of applications for Pando. Finally, we provide a few main processing functions, often simplified, as reference in Appendix B. The full source code for all applications can be found in the online Pando handbook [105].
Chapter 6

Evaluation

The previous chapters have explained how to implement Pando as well as a number of realistic example applications. In this chapter, we quantify the performance benefits offered by Pando in a number of real-world scenarios. We first present our measurement methodology and its implementation (Section 6.1). We then show concrete performance benefits on the example applications of Chapter 5, deployed on multiple devices both on a local Wi-Fi network (Section 6.2) and on wide-area networks (Section 6.3). We conclude with a summary of our contributions (Section 6.4). Further detail that might be useful to replicate our experiments, including the instrumented applications, are available in our online handbook [105].

6.1 Measurement Methodology

Our measurement approach enables the quick assessment of the throughput performance of individual devices as well as the aggregation and comparison of the results globally. This in turn enables quick tuning of applications to maximize performance and the identification of the best and worst performing devices to focus deployment on the most useful ones. Our measurement infrastructure provides real-time updates on the contributions of each device, which increases the engagement of volunteers, and easy logging for later offline analysis, to share and analyze performance results.

We first present how we measure throughput performance on each device (Section 6.1.1), we then explain how we aggregate the results (Section 6.1.2), and finally show how we implemented those strategies in Pando (Section 6.1.3).
6.1.1 Measuring the Performance of Individual Devices

Each device measures its own throughput performance by recording, for each input, the number of items processed, as well as the time spent performing computations and data transfers. The device then reports the cumulated measurements and derived metrics at regular intervals to Pando. Since the determination of metrics is specific to each application, a developer needs to manually augment the processing function with a few additional time measurements and a single call to a Pando library function, as illustrated in Figure 6.1.

```javascript
var render = require('raytracer') // import existing function
var zlib = require('zlib') // import compressing module
var pando = require('pando-computing') // import Pando library

module.exports['/pando/1.0.0'] = function(cameraPos, cb) {
  try {
    var startTime = new Date()
    var pixels = render(parseFloat(cameraPos))
    pando.report({
      cpuTime: new Date() - startTime, // computing time (ms)
      dataTransferTime: 0, // data transfer time (ms)
      nbItems: 1, // number of items processed
      units: 'Images' // nature of items processed
    })
  cb(null, zlib.gzipSync(new Buffer(pixels)).toString('base64'))
  } catch (err) {
    cb(err)
  }
}
```

Fig. 6.1 Raytracing Example Augmented with Measurements.

Behind the scene, the report function accumulates the time measurements and the number of items processed. To bound the number of messages sent for monitoring the system and limit bandwidth utilization, we configured the reports to be sent at regular pre-defined intervals. The default reporting interval is three seconds but can be modified through Pando’s reporting-interval parameter on startup. Periodically, every time the reporting interval has elapsed, various metrics are computed: the average throughput in items per second and the percentage of time spent performing computation and data transfer. The current results are kept in memory to compute statistics (average, standard-deviation, maximum, and minimum, standard-deviation) since a device was first connected. The metrics and current statistics are then displayed in the web page executing Pando, as illustrated with a screenshot in Figure 6.2.
The reporting interval is actually a lower time bound. The interval between reports may actually be larger in practice because some part of the computation executing in the main thread of execution could delay the execution of the reporting function.\footnote{JavaScript is single-threaded. Application developers may choose to use WebWorkers\cite{WebWorkers} to mitigate the issue.} To compute the various metrics, we therefore use the actual time elapsed between the current and the last report.

A worker may potentially spend significant time outside of the computation and data transfer times that are explicitly measured. The extra overhead may come from the libraries used, and operating system or network operations that are not strictly incurred by an application’s JavaScript implementation. In practice however, the exact source of overhead need not be determined: when the CPU load, i.e. \(\frac{\text{cpuTime}}{\text{elapsedTimeSinceLastReport}}\), is \(> 95 - 99\%\) the majority of the time, the overhead of using Pando is actually sufficiently low to be negligible.

By monitoring the CPU load and data transfer loads on the device, a developer can quickly determine whether to increase the amount of computation per input to increase CPU utilization. And once CPU load is sufficiently high, a developer may also optimize the application implementation to maximize the throughput.

\begin{figure}[h!]
\centering
\includegraphics[width=\textwidth]{fig6_2.png}
\caption{iPhone SE Real-time Measurement Screenshot after Disconnection.}
\end{figure}
6.1.2 Aggregating Metrics from Multiple Devices in Real-Time

Since individual devices report at potentially variable intervals, aggregating metrics from multiple devices leads to inexact results that can potentially over- or under-estimate the correct value. The issue essentially comes from the mismatch in sampling rates. Let’s illustrate the problem with an example, graphically represented in Figure 6.3.

Suppose we have two devices \( A \) and \( B \), with \( A \) having a processing speed that is twice that of \( B \). Moreover, let’s assume that the reporting interval, \( t \), is roughly equal to the time it takes for \( A \) to process a single value. The time it will take for \( B \) to process a value will therefore be roughly \( 2t \). \( A \) will display throughputs oscillating around \( \frac{1}{t} \) item and will refresh them every \( t \). \( B \) will display throughputs oscillating around \( \frac{0.5}{t} \) item and will report them every \( 2t \). Both throughput reports are accurate from the respective points of view of \( A \) and \( B \).

Now also suppose that there is a monitor process \( M \) that receives the reports and displays them to the user. \( M \) refreshes the throughputs in real-time, every \( t \) time interval. From a start at 0, \( M \) will make a first report at time \( t \) with only the throughput of \( A \), because it does not yet have received \( B \)’s report. Then at time \( 2t \), \( M \) must decide how to compute \( B \)’s throughput. We have identified three possible options, all with their benefits and drawbacks.

First, \( M \) can simply report the throughput submitted by \( B \), \( 0.5 \frac{\text{item}}{t} \) in this example. This will accurately represent the combined throughput of \( A \) and \( B \) in the interval \([t, 2t]\). However, this under-estimates the actual throughput of \( B \) on the interval \([0, t]\). The reports therefore cannot be used to accurately compute the total number of items processed. In this example, by using the reports between \([0, 2t]\), \( M \) would accurately compute 2 items processed by \( A \), but only 0.5 items for \( B \) instead of the actual 1 item that was processed.
Second, $M$ can instead compute the throughputs \textit{relative to its own time reference} rather than that of $A$ and $B$. $M$ can use the \textit{number of items} processed since the last report, rather than the self-reported throughput of $A$ and $B$, and derive the throughputs from its own sampling interval. Using the same example as before, $M$ would compute again a throughput of $1 \frac{\text{item}}{t}$ for $A$ both between $[0, t]$ and $[t, 2t]$. But $M$ would compute a throughput of $1 \frac{\text{item}}{t}$ for $B$ on the interval $[t, 2t]$. This would accurately represent the total number of values processed on the interval $[0, 2t]$ but it over-estimates the actual throughput of $B$ on the interval $[t, 2t]$.

Third, $M$ can \textit{retroactively update the previous reports} for $B$. In this example, at time $t$ the report for $B$ would be 0, but at time $2t$, $M$ would update the report to $0.5 \frac{\text{item}}{t}$ after the reception of $B$’s report. This option accurately represents $B$’s throughput but may not actually be practical if the real-time measurements are streamed to other tools.

We used the first option in our real-time reporting infrastructure: it is the simplest to implement and while not exact, it provides a lower-bound on the actual performance of the devices. In practice however, we simply mitigated the entire issue by choosing a reporting interval that was sufficiently large to contain the processing time of a single input on the slowest device.

\subsection{6.1.3 Measurement Infrastructure}

Our implementation is separated in three components: the \textit{data collection}, the \textit{logging and offline analysis} tools, and the \textit{real-time display}.

For \textit{collecting data}, we implemented the previous measurement methodology for Workers connected over both WebSocket and WebRTC (Section 3.4.3). When the Worker is connected over a WebSocket connection, the reporting library, previously shown in Figure 6.1, transparently opens an additional WebSocket connection to the Master process, through a separate HTTP route, to send the reports. When the Worker is connected over WebRTC, there is already a second connection opened for managing the potential growth of a tree overlay (Chapter 7). This connection is reused for submitting reports.

For \textit{logging data and offline analysis}, we provide periodic aggregate reports on all currently connected devices through the Unix standard error of Pando. By default, the reports are inactive, but they can be activated by setting the \texttt{DEBUG} environment variable to "pando-computing:monitoring" when invoking Pando. The reports can then be redirected into a file using the Unix standard error redirection operator ($2>$\texttt{FILE}) for logging. A similar strategy is used to log the throughput as measured by tools connected to the output of Pando. We show both in Figure 6.4, in an example invocation that we used for measuring the performance of the Raytracer application. We then use the logs, in a separate analysis step, to compute the average throughput of individual devices and their total. For individual devices,
we add the total number of items processed and divide by the sum of all report intervals. For the total throughput of all devices together, we add the individual throughputs.

```bash
./generate-angles --positions-nb=5000 | DEBUG='pando-computing: monitoring' pando raytracer.js --stdin --start-idle --batch-size=2 2>$TESTDIR/devices.txt | DEBUG='throughput*' ./monitor 2>$TESTDIR /output.txt >/dev/null
```

Fig. 6.4 Raytracing Invocation with Logs.

For real-time display, we simply aggregate the device metrics, as explained in Section 6.1.2 and periodically send the updates to a separate monitoring Web page for visualization. The URL of this Web page is provided by Pando on startup. From these reports, we generate a number of figures using the plotly JavaScript library [160], shown in Figure 6.5.

![Absolute Throughputs](image1)

(a) Absolute Throughputs.

![Relative Throughputs](image2)

(b) Relative Throughputs.

![CPU, Data Transfer, and Other Loads](image3)

(c) CPU, Data Transfer, and Other Loads.

![Historical Throughputs](image4)

(d) Historical Throughputs.

Fig. 6.5 Screenshots of Real-time Reports with a Macbook Air 2011 and an iPhone SE.
6.2 Volunteering Scenarios on a Local Wi-Fi Network

In this section, we show that Pando is useful to leverage our personal devices at home, but also the smartphones of our friends and colleagues at work. We expect these two scenarios to be the most common use cases for Pando for a majority of users. For both experiments, we have ensured 95-99% CPU utilization on all application examples to measure the full computing potential of the participating devices. We used version 0.17.2 of Pando for the tests and all devices connected over the WebSocket protocol. We used the version of benchmarks at commit 12164ee69b of the pando-handbook [105].

6.2.1 Personal Devices

We used the Collatz, Crypto-Mining, StreamLender Random Testing, Raytrace, Image-Processing, and Machine Learning Agent applications. For all applications, we hide transmission delays by sending values to process in batches of two, this way network delays for one value happen while the other is processed.

Table 6.2 shows the combination of devices we gathered from those we have accumulated over the years. The oldest is the iPhone 4S (2 cores 1.0 Ghz ARM 32-bit), released in 2011, and the two newest are the iPhone SE (2 cores 1.85 Ghz ARMv8 64-bit), released in 2016, and the Macbook Pro 2016 (4 cores i5 2.9 Ghz x86 64-bit). In between, we also have the Novena [149], 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 Air mid-2011 (2 cores i7 1.8 Ghz x86 64-bit). We used Firefox (64.0 on x86 and 60.3.0 ESR on ARM) on laptops for consistency and because it is the fastest on numerical benchmarks [81] and on the iPhones we used Safari.

We noticed that the number of concurrent browser tabs that provided the maximum performance was less than the number of cores of many devices, possibly because some shared resources of the CPUs were saturated or because the OS or the browser reserved other cores for different services. We therefore chose the minimum number of cores that provided the maximum performance, which we mention beside the device name in Table 6.2. The performance when using a single core was roughly equal to the ratio of the throughput obtained divided by the number of cores mentioned. We also reserve one core on the MacBook Air 2011 to execute Pando’s Master process which coordinates communication with other devices, leaving the other cores for computations.

A few results are worth discussing. First, the performance of the iPhone 4S was too low on some benchmarks to be included. On the others we noticed that the iPhone SE brings a
significant performance improvement, between 3x and 21x. This shows that not all older phones may provide a significant contribution on modern tasks. Second, the iPhone 4S and the Macbook Air 2011 are of the same generation, similar to the iPhone SE and the Macbook Pro. The performance gap between each pair, when taking the performance on a single core, has significantly reduced; it was between 3.3x and 14x in 2011 and dropped to between 1.3x and 2.1x in 2016. Note that on the image processing application, the Macbook Pro is surprisingly slower; using Safari on the Macbook Pro instead of Firefox makes it faster again, the difference can therefore be attributed to the difference in optimizations performed by browsers. Third, combining all other devices provides a performance level comparable to that of the Macbook Pro, which means that we could at least double the overall throughput of the applications by leveraging other devices we have access to, making them quite useful.

<table>
<thead>
<tr>
<th>Device (cores)</th>
<th>Collatz</th>
<th>Crypto-Mining</th>
<th>Random-Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bignum/s</td>
<td>Hashes/s</td>
<td>Tests/s</td>
</tr>
<tr>
<td><strong>iPhone 4S</strong> (1)</td>
<td>15.55</td>
<td>13,951</td>
<td>54.22</td>
</tr>
<tr>
<td><strong>Novena</strong> (2)</td>
<td>63.56</td>
<td>16,326</td>
<td>150.46</td>
</tr>
<tr>
<td><strong>Asus Laptop</strong> (3)</td>
<td>254.08</td>
<td>59,877</td>
<td>617.40</td>
</tr>
<tr>
<td><strong>MBAir 2011</strong> (1)</td>
<td>218.92</td>
<td>56,906</td>
<td>551.18</td>
</tr>
<tr>
<td><strong>iPhone SE</strong> (1)</td>
<td>314.86</td>
<td>46,849</td>
<td>498.65</td>
</tr>
<tr>
<td><strong>MBPro 2016</strong> (2)</td>
<td>814.48</td>
<td>199,917</td>
<td>1816.23</td>
</tr>
<tr>
<td><strong>All</strong></td>
<td>1681.45</td>
<td>393,826</td>
<td>3688.15</td>
</tr>
</tbody>
</table>

Table 6.1 Average Throughput using a Combination of Personal Devices on Collatz, Crypto-Mining, and Random-Testing.

<table>
<thead>
<tr>
<th>Device (cores)</th>
<th>Raytrace</th>
<th>Image-Processing</th>
<th>MLAGent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Frames/s</td>
<td>%</td>
<td>Images/s</td>
</tr>
<tr>
<td><strong>iPhone 4S</strong> (1)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>Novena</strong> (2)</td>
<td>0.34</td>
<td>3.2</td>
<td>0.03</td>
</tr>
<tr>
<td><strong>Asus Laptop</strong> (3)</td>
<td>1.88</td>
<td>17.6</td>
<td>0.08</td>
</tr>
<tr>
<td><strong>MBAir 2011</strong> (1)</td>
<td>1.47</td>
<td>13.7</td>
<td>0.04</td>
</tr>
<tr>
<td><strong>iPhone SE</strong> (1)</td>
<td>1.69</td>
<td>15.8</td>
<td>0.23</td>
</tr>
<tr>
<td><strong>MBPro 2016</strong> (2)</td>
<td>5.33</td>
<td>49.8</td>
<td>0.14</td>
</tr>
<tr>
<td><strong>All</strong></td>
<td>10.70</td>
<td>100.0</td>
<td>0.53</td>
</tr>
</tbody>
</table>

Table 6.2 Average Throughput using a Combination of Personal Devices on Raytrace, Image-Processing, and MLAGent.
6.2 Volunteering Scenarios on a Local Wi-Fi Network

6.2.2 Smartphones of Colleagues on a Local Wi-Fi Network

We invited 12 of our colleagues at work to volunteer their phones. Arguably, the sample of phones is representative of phones owned by the general public. We repeated the Random-Testing experiment, but omitted the other applications to avoid draining the phones’ batteries (and the patience of our colleagues!). The list of phones and their average throughput over the entire experiment is provided in Table 6.3. Figure 6.6 shows historical throughput on the same experiment, which illustrates the variability in amount of contributed throughput and the combined throughput of all phones. The exact specification of each device is rather tedious to list and of limited interest since the whole experiment would be rather hard to replicate. There are still a few points worth mentioning.

<table>
<thead>
<tr>
<th>Device</th>
<th>Tests/s</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPhone SE</td>
<td>443.46</td>
<td>18.70</td>
</tr>
<tr>
<td>Huawei P10 lite 2017</td>
<td>364.99</td>
<td>15.39</td>
</tr>
<tr>
<td>Samsung Galaxy S7</td>
<td>304.64</td>
<td>12.84</td>
</tr>
<tr>
<td>Xiaomi redmi note 6 pro</td>
<td>291.03</td>
<td>12.27</td>
</tr>
<tr>
<td>LG G6 H870 2017</td>
<td>260.17</td>
<td>10.97</td>
</tr>
<tr>
<td>Lenovo P2a42 2016</td>
<td>171.26</td>
<td>7.22</td>
</tr>
<tr>
<td>Wileyfox Storm 2016</td>
<td>128.89</td>
<td>5.43</td>
</tr>
<tr>
<td>Honor</td>
<td>125.29</td>
<td>5.28</td>
</tr>
<tr>
<td>Zenfone 3</td>
<td>100.58</td>
<td>4.24</td>
</tr>
<tr>
<td>Samsung A3 2016</td>
<td>90.58</td>
<td>3.82</td>
</tr>
<tr>
<td>Zenfone 2</td>
<td>55.81</td>
<td>2.35</td>
</tr>
<tr>
<td>Huawei P10 lite 2017 (2)</td>
<td>35.13</td>
<td>1.48</td>
</tr>
<tr>
<td><strong>All</strong></td>
<td><strong>2371.82</strong></td>
<td><strong>100.00</strong></td>
</tr>
</tbody>
</table>

Table 6.3 Average Throughput of Cell Phones Volunteered by Colleagues.

First, the range of performance is significant, the slowest device, the Zenfone 2, is 8 times slower than the fastest of the lot, the iPhone SE. Second, it may be possible that some of them had been using energy saving modes, the iPhone SE was connected over a usb cable while the other were all running from their batteries. This is certainly the case for the second Huawei phone, which locked during the experiment and went into low power mode, explaining the 10x difference with the other identical device. Third, the overall performance of all devices combined is higher than that of the Macbook Pro, showing that asking your colleagues for help may be a valid substitute for a faster machine on some applications. And finally, the implementation of the application used only a single core on each device, it may be possible
to reach a factor of 2-4 better performance in the future by leveraging parallel libraries or WebWorkers [203].

The last experiment shows that while older cell phones, such an iPhone 4S, may contribute an insignificant amount of computing power, the combined computing power of a dozen more recent cellphones can outperform a top-of-line laptop of only two years ago. Moreover, the first experiment results provide additional empirical evidence for the decreasing performance gap between cellphones and laptop computers, opening the door for using them for significant computations in the future, not only alone but also in combination with others.

### 6.3 Volunteering Scenarios on a Wide Area Network

In this section, and in complement to the previous results, we evaluate the performance of Pando in two additional deployment scenarios: a France-wide state-of-the-art computing grid, Grid5000 [17], connected over a virtual private network (VPN) that is similar to a large organization computing infrastructure, and a wide-area network (WAN) deployment with computing devices distributed throughout Europe on PlanetLab EU [159] that is similar to deployments on the devices of a distributed volunteer community.
These experiments were made in a separate setting with the more recent 0.17.14 version of Pando [106] and the application examples in Pando’s handbook [105], at commit c5247923. The only major difference compared to the experiments of the previous section, is that the image used for Raytrace was smaller to avoid a limitation on the size of individual WebRTC messages in the simple-peer [3] library we use for managing WebRTC connections. The consequence is that throughput results, in this evaluation, shall be larger for the same devices, running the same browser, on the same network. Otherwise, the applications were the same and we measured the throughput for at least five minutes of execution, which diminished the impact of some input requiring more computation than others. Also, the HTTP version of Image Processing access a file server from the local network, we therefore do not provide throughput results on the WAN case.

Because the version of Pando and the applications were more recent, we redid the experiment with the personal devices to make them comparable with the VPN and WAN experiments. In the next three sections, we list the specifications of the devices and the network that we used for the experiments.

### 6.3.1 Updated Personal Devices on a Local-Area Network

We selected the personal devices of Section 6.2.1 but we omitted the slower iPhone 4S because it previously only contributed a small throughput. The devices had the following software configurations: the iPhone SE was executing iOS 12.1, and Safari; the Macbook Air mid-2011 executed MacOS 10.13.6 and Firefox 66.0.5 64-bit; the Novena [149] executed Debian Linux 8, and Firefox 60.3.0esr 32-bit; the Asus Windows laptop executed Windows 10 version 1803 and Firefox 66.0.5 64-bit, and the Macbook Pro 2016 executed MacOS 10.14.1 and Firefox 63.0.1 64-bit.

The MacBook Air executed the Master process of Pando and was connected to the other personal devices through a Wifi network. We used a batch-size of 2, effectively enabling one input to be transferred while the other is being processed.

### 6.3.2 Devices on a Virtual Private Network across France (Grid5000)

We selected one node for each of the 8 participating Grid5000 clusters, themselves distributed between major cities in France along the INRIA network. Each cluster has multiple models, each with a unique name that facilitates replication. We list them by model name (ex: dahu) followed by the cluster site where they are hosted (ex: grenoble), as well as their technical characteristics. They all use different versions of Debian Linux 4.9.x 64-bit and as a browser,
Chrome version 73.0.3683.121, through the Electron 5.0.1 environment. The characteristics of each node follow.

*dahu.grenoble*, acquired in 2018, and comprised of 32 nodes, each having 2 CPUs Intel Xeon Gold 6130, 16 cores/CPU, 192GB RAM, 223GB SSD, 447GB SSD, 3726GB HDD, and 10Gbps ethernet.

*chetemy.lille*, acquired in 2016, and comprised of 15 nodes having 2 CPUs Intel Xeon E5-2630 v4, 10 cores/CPU, 256 GiB RAM, 300 GB HDD, and 10 Gbps ethernet.

*petitprince.luxembourg*, acquired in 2013, and comprised of 15 nodes having 2 CPUs Intel Xeon E5-2630L, 6 cores/CPU, 32GB RAM, 232GB HDD, and 10Gbps ethernet.

*nova.lyon*, acquired in 2016, comprised of 23 nodes having 2 CPUs Intel Xeon E5-2620 v4, 8 cores/CPU, 64GB RAM, 557GB HDD, and 10Gbps ethernet.

*grisou.nancy*, acquired in 2016, comprised of 51 nodes having 2 CPUs Intel Xeon E5-2630 v3, 8 cores/CPU, 128GB RAM, 2x558GB HDD, and 10Gbps ethernet.

*ecotype.nantes*, acquired in 2017, comprised of 48 nodes having 2 CPUs Intel Xeon E5-2630L v4, 10 cores/CPU, 128GB RAM, 372GB SSD, and 10Gbps ethernet.

*paravance.rennes*, acquired in 2015, comprised of 72 nodes having 2 CPUs Intel Xeon E5-2630 v3, 8 cores/CPU, 128GB RAM, 2x558GB HDD, and 10Gbps ethernet.

*uvb.sophia*, acquired in 2011, comprised of 44 nodes having 2 CPUs Intel Xeon X5670, 6 cores/CPU, 96GB RAM, 232GB HDD, and having 1 Gbps ethernet.

We measured the performance on a single core on a single node per cluster. The results should scale linearly with additional nodes but less than linearly when using more than one core per node, because there is increasing contention for CPU resources when increasing the number of cores used in parallel. The Master process of Pando was executing on one core of the MacBook Air 2011, mentioned in the personal devices experiment and the connections between the Master process and the remote devices were made using the WebSocket protocol through a VPN connection. The MacBook Air was itself connected to the Internet through the Wifi network of INRIA and to the Grid5000 nodes through a VPN access. We used a batch-size of 2, effectively enabling one input to be transferred while the other is being processed.

### 6.3.3 Devices on a Wide Area Network across Europe (PlanetLab EU)

We selected seven nodes among the PlanetLab EU nodes that are still working and used one core per node. For each node, we used Chrome version 69.0.3497.128 through the Electron 4.1.3 environment. The characteristics of each node follow.
6.3 Volunteering Scenarios on a Wide Area Network

\textit{cse-yellow.cse.chalmers.se} Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz with 6 cores/CPU, 512MB RAM, Fedora Core Linux, version 25, kernel 4.13.16 64-bit, and having 10 Gbps ethernet.

\textit{mars.planetlab.haw-hamburg.de} Intel(R) Xeon(R) CPU L5520 @ 2.27GHz with 4 cores/CPU, 512MB RAM, Fedora Core Linux, version 25, kernel 4.13.16 64-bit, and having 10 Gbps ethernet.

\textit{ple42.planet-lab.eu} Intel(R) Westmere E56xx/L56xx/X56xx (Nehalem-C) with 1 core/CPU, 512MB RAM, Fedora Core Linux, version 25, kernel 4.13.16 64-bit, and having 10 Gpbs ethernet.

\textit{onelab2.pl.sophia.inria.fr} Intel(R) Xeon(R) CPU E5506 @ 2.13GHz with 4 cores/CPU, 512MB RAM, Fedora Core Linux, version 25, kernel 4.11.9 64-bit, and having 10 Gpbs ethernet.

\textit{planet2.elte.hu} Intel(R) Core(TM)2 Duo CPU E6850 @ 3.00GHz with 2 cores/CPU, 512MB RAM, Fedora Core Linux, version 25, kernel 4.8.6 64-bit, and having 10 Gpbs ethernet.

\textit{planet4.cs.huji.ac.il} Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz with 10 cores/CPU, 512MB RAM, Fedora Core Linux, version 25, kernel 4.11.9 64-bit, and having 10 Gpbs ethernet.

\textit{ple4.cesnet.cz} Intel(R) Xeon(R) CPU X3363 @ 2.83GHz with 4 cores/CPU, 512MB RAM, Fedora Core Linux, version 25, kernel 4.11.9 64-bit, and having 10 Gpbs ethernet.

We measured the performance on a single core on a single node per cluster. Similar to the VPN experiment, the Master process of Pando was executing on one core of the MacBook Air 2011. However, the connections between the Master process and the remote devices were made using the WebRTC protocol. The MacBook Air was itself connected to the Internet through the Wifi network of INRIA. We used a batch-size of 4, effectively enabling up to three additional inputs to be transferred while the first is being processed.

6.3.4 Performance Results

The experiment results are shown in Table 6.4 and 6.5. We highlight here interesting results derived from the data presented.

\textit{Pando can take advantage of computing devices, whether available on a LAN, a VPN, or a WAN.} We could use the same tool to execute the applications in parallel on the multiple cores of a single personal device, on multiple personal devices at the same time, on a state-of-the-art grid infrastructure distributed throughput France, or on a distributed set of devices connected to the Internet. In all cases, there was a performance benefit in using all those devices in parallel that improves significantly on the performance that would have been obtained
### Table 6.4 Average Throughput using a Combination of Devices on a LAN, VPN, and WAN on Collatz, Crypto-Mining, and Random-Testing.

<table>
<thead>
<tr>
<th>Device Type</th>
<th>LAN: Personal Devices (cores)</th>
<th>VPN: Grid5000 Nodes (cores)</th>
<th>WAN: PlanetLab EU Nodes (cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Collatz</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bignums/s</td>
<td>2209.65</td>
<td>3823.51</td>
<td>1845.52</td>
</tr>
<tr>
<td>%</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td><strong>Crypto-Mining</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hashes/s</td>
<td>378,672</td>
<td>1,534,102</td>
<td>717,485</td>
</tr>
<tr>
<td>%</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td><strong>Random-Testing</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tests/s</td>
<td>3603.70</td>
<td>7559.93</td>
<td>3985.04</td>
</tr>
<tr>
<td>%</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
</tbody>
</table>

- **LAN: Personal Devices**
  - Novena (2) - Bignums/s: 121.85, Hashes/s: 16,185, Tests/s: 142.84
  - Asus Laptop (3) - Bignums/s: 490.45, Hashes/s: 59,895, Tests/s: 622.64
  - MBAir 2011 (1) - Bignums/s: 215.58, Hashes/s: 58,693, Tests/s: 526.82
  - iPhone SE (1) - Bignums/s: 336.18, Hashes/s: 42,720, Tests/s: 509.64
  - MBPro 2016 (2) - Bignums/s: 1045.58, Hashes/s: 201,178, Tests/s: 1801.76

- **VPN: Grid5000 Nodes**
  - dahu.grenoble (1) - Bignums/s: 642.04, Hashes/s: 230,061, Tests/s: 1341.77
  - chetemy.lille (1) - Bignums/s: 524.71, Hashes/s: 206,195, Tests/s: 975.58
  - petitprince.luxembourg (1) - Bignums/s: 261.36, Hashes/s: 136,189, Tests/s: 631.83
  - nova.lyon (1) - Bignums/s: 521.35, Hashes/s: 199,901, Tests/s: 982.16
  - grisou.nancy (1) - Bignums/s: 541.53, Hashes/s: 216,932, Tests/s: 1026.26
  - ecotype.nantes (1) - Bignums/s: 479.07, Hashes/s: 187,668, Tests/s: 939.07
  - paravance.rennes (1) - Bignums/s: 535.72, Hashes/s: 215,096, Tests/s: 1021.99
  - uvb.sophia (1) - Bignums/s: 317.73, Hashes/s: 142,061, Tests/s: 641.26

- **WAN: PlanetLab EU Nodes**
  - cse-yellow.cse.chalmers.se (1) - Bignums/s: 470.49, Hashes/s: 162,173, Tests/s: 996.89
  - mars.planetlab.haw-hamburg.de (1) - Bignums/s: 225.38, Hashes/s: 93,189, Tests/s: 428.30
  - ple42.planet-lab.eu (1) - Bignums/s: 210.15, Hashes/s: 82,297, Tests/s: 444.35
  - onelab2.pl.sopha.inria.fr (1) - Bignums/s: 201.43, Hashes/s: 95,609, Tests/s: 459.66
  - planet2.elte.hu (1) - Bignums/s: 216.42, Hashes/s: 85,927, Tests/s: 505.04
  - planet4.cs.huji.ac.il (1) - Bignums/s: 298.42, Hashes/s: 112,363, Tests/s: 651.54
  - ple1.cesnet.cz (1) - Bignums/s: 223.22, Hashes/s: 85,927, Tests/s: 499.27
### 6.3 Volunteering Scenarios on a Wide Area Network

Table 6.5 Average Throughput using a Combination of Devices on a LAN, VPN, and WAN on RayTrace, Image-Processing, and MLAGent.

<table>
<thead>
<tr>
<th>LAN: Personal Devices (cores)</th>
<th>Raytrace Frames/s</th>
<th>Raytrace %</th>
<th>Image-Process. Img./s</th>
<th>Image-Process. %</th>
<th>MLAGent Steps/s</th>
<th>MLAGent %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Novena (2)</td>
<td>0.66</td>
<td>3.5</td>
<td>0.04</td>
<td>5.3</td>
<td>51.74</td>
<td>10.7</td>
</tr>
<tr>
<td>Asus Laptop (3)</td>
<td>3.63</td>
<td>19.1</td>
<td>0.10</td>
<td>13.3</td>
<td>112.59</td>
<td>23.2</td>
</tr>
<tr>
<td>MBAir 2011 (1)</td>
<td>2.94</td>
<td>15.5</td>
<td>0.06</td>
<td>9.0</td>
<td>68.81</td>
<td>14.2</td>
</tr>
<tr>
<td>iPhone SE (1)</td>
<td>2.90</td>
<td>15.3</td>
<td>0.33</td>
<td>45.9</td>
<td>60.24</td>
<td>12.4</td>
</tr>
<tr>
<td>MBPro 2016 (2)</td>
<td>8.81</td>
<td>46.6</td>
<td>0.19</td>
<td>26.5</td>
<td>191.51</td>
<td>39.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN: Grid5000 Nodes (cores)</th>
<th>Raytrace Frames/s</th>
<th>Raytrace %</th>
<th>Image-Process. Img./s</th>
<th>Image-Process. %</th>
<th>MLAGent Steps/s</th>
<th>MLAGent %</th>
</tr>
</thead>
<tbody>
<tr>
<td>dahu.grenoble (1)</td>
<td>3.12</td>
<td>19.0</td>
<td>0.44</td>
<td>16.1</td>
<td>219.18</td>
<td>16.6</td>
</tr>
<tr>
<td>chetemy.lille (1)</td>
<td>2.04</td>
<td>12.5</td>
<td>0.37</td>
<td>13.6</td>
<td>167.03</td>
<td>12.6</td>
</tr>
<tr>
<td>petitprince.luxembourg (1)</td>
<td>1.47</td>
<td>9.0</td>
<td>0.27</td>
<td>9.7</td>
<td>124.00</td>
<td>9.4</td>
</tr>
<tr>
<td>nova.lyon (1)</td>
<td>1.95</td>
<td>11.9</td>
<td>0.34</td>
<td>12.4</td>
<td>164.57</td>
<td>12.4</td>
</tr>
<tr>
<td>grisou.nancy (1)</td>
<td>2.17</td>
<td>13.2</td>
<td>0.36</td>
<td>13.1</td>
<td>176.12</td>
<td>13.3</td>
</tr>
<tr>
<td>ecotype.nantes (1)</td>
<td>1.86</td>
<td>11.4</td>
<td>0.33</td>
<td>12.1</td>
<td>162.25</td>
<td>12.3</td>
</tr>
<tr>
<td>paravance.rennes (1)</td>
<td>2.19</td>
<td>13.4</td>
<td>0.35</td>
<td>12.8</td>
<td>176.41</td>
<td>13.3</td>
</tr>
<tr>
<td>uvb.sophia (1)</td>
<td>1.57</td>
<td>9.6</td>
<td>0.28</td>
<td>10.2</td>
<td>133.88</td>
<td>10.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WAN: PlanetLab EU Nodes (cores)</th>
<th>Raytrace Frames/s</th>
<th>Raytrace %</th>
<th>Image-Process. Img./s</th>
<th>Image-Process. %</th>
<th>MLAGent Steps/s</th>
<th>MLAGent %</th>
</tr>
</thead>
<tbody>
<tr>
<td>cse-yellow,cse.chalmers.se (1)</td>
<td>0.74</td>
<td>15.5</td>
<td>—</td>
<td>—</td>
<td>148.85</td>
<td>20.8</td>
</tr>
<tr>
<td>mars.planetlab</td>
<td>0.64</td>
<td>13.6</td>
<td>—</td>
<td>—</td>
<td>78.66</td>
<td>11.0</td>
</tr>
<tr>
<td>.haw-hamburg.de (1)</td>
<td>0.54</td>
<td>11.3</td>
<td>—</td>
<td>—</td>
<td>81.17</td>
<td>11.4</td>
</tr>
<tr>
<td>ple42.planet-lab.eu (1)</td>
<td>0.68</td>
<td>14.3</td>
<td>—</td>
<td>—</td>
<td>83.57</td>
<td>11.7</td>
</tr>
<tr>
<td>onelab2.pl.sophia.inria.fr (1)</td>
<td>0.73</td>
<td>15.4</td>
<td>—</td>
<td>—</td>
<td>99.75</td>
<td>14.0</td>
</tr>
<tr>
<td>planet2.elte.hu (1)</td>
<td>0.77</td>
<td>16.1</td>
<td>—</td>
<td>—</td>
<td>119.62</td>
<td>16.7</td>
</tr>
<tr>
<td>planet4.cs.huji.ac.il (1)</td>
<td>0.65</td>
<td>13.8</td>
<td>—</td>
<td>—</td>
<td>102.76</td>
<td>14.4</td>
</tr>
</tbody>
</table>
otherwise on a single personal device. That flexibility, for example, enables leveraging the fastest computing devices available with a minimum of effort: in our experiments, these were the Grid5000 nodes.

The throughput impact of network latency can be minimized for computation-bound applications, if large enough batches of inputs are used. For the LAN and VPN experiments, we used input batches of size 2 and for the PlanetLab experiments, we used input batches of 4. These were sufficiently large to compensate for the transmission delay of inputs, even in the case of image-processing where 168kb images were sent for processing through a different channel. Obviously, those results hold only as long the ratio between computation time and data transfer time is sufficiently large. Nonetheless, it shows that for applications for which this holds, the option of sending inputs in batches provided by Pando is sufficient to hide the network latency.

A single core from personal devices of 2016 sometimes provide higher throughput than older servers. On Collatz, the iPhone SE outperforms the uvb.sophia from Grid5000 and almost all PlanetLab server nodes. This is true in more cases when comparing the throughput of a single core on the MBPro 2016 with the performance of a few Grid5000 nodes and many PlanetLab nodes. It therefore means that, sometimes, it may be better to leverage many personal devices than relying on older and remote server nodes.

2-5 cores on recent personal devices can outperform the fastest server core. It therefore means that asking 2-5 friends with recent smartphones or laptops, such as the iPhone SE or the Macbook Pro 2016, to participate with Pando can replace renting a high-end server core in remote data centres. While this seems rather impractical if the devices are powered by their battery, the use of portable solar panels, for example, can remove the problem during sunny days.

The previous experiments therefore show that using Pando, a user can leverage spare computing capacity either in local or remote personal devices, that batching inputs is sufficient to hide network latency, and that the computing power available in personal devices is quite significant, even compared to state-of-the-art grid infrastructure. To the best of our knowledge, Pando is the first tool for volunteer computing that can leverage computing resources in all these scenarios within the same tool.

6.4 Summary

In this Chapter, we have first explained how we measured the throughput performance on individual results, through a reporting library that tracks in real-time the throughput, CPU utilization, and data transfer times. At regular intervals, this library displays the results on
the device, and reports the samples to the Pando’s Master process. The results from multiple devices are then aggregated for display on a monitoring Web page and logging for later analysis. We used both capabilities to run the following experiments.

First, we measured the performance of the applications of Chapter 5 on personal devices we have accumulated over the years, that cover the Linux, Mac, and Windows operating systems, the Intel and ARM processor architectures, in laptops and smartphones. These experiments showed that Pando enabled devices from 2011 to 2016 to provide a combined computing power equivalent to a MacBook Pro 2016, showing there is still value in leveraging them for computations. They also highlighted that the performance gap between an Apple phone and a laptop has diminished from 2011 to 2016, showing that smartphones are becoming increasingly competitive as computation devices. We confirmed that intuition by measuring the combined performance of 12 smartphones provided by our colleagues at work: this resulted in a higher total throughput than that of a MacBook Pro from 2016. These two experiments showed the value in leveraging personal devices on a local area network, as a replacement for higher-end devices.

Second, we measured the performance of the same applications, this time executing on Grid5000 machines distributed throughout France connected over a Virtual Private Network, which is similar to the network infrastructure of any other large organization with many desktop devices, as well as PlanetLab nodes distributed throughout Europe and connected over the Internet, which is similar to a distributed group of volunteers connected over the Internet. These two experiments showed that Pando can be beneficial in both cases and that the additional network latency can be hidden by increasing the number of inputs that are batched together. Moreover, the performance of 2-5 personal devices has been shown to be competitive not only with the older servers that are part of PlanetLab but also to the state-of-the-art servers that are part of Grid5000.

To the best of our knowledge, Pando is the first volunteer computing tool that has been shown to be beneficial and quickly deployed on devices, personal or professional, connected over local and wide-area networks, either through a virtual private network or the Internet.
Chapter 7

Genet: Quickly Scalable Fat-Tree Overlay for WebRTC

Current browser implementations limit the number of concurrent WebRTC connections to a single source to 256 [131]. In practice, this number is even more limited: the overhead of maintaining connections becomes significant beyond 70 concurrent connections in some libraries.\(^1\) This limits the total number of participants.

We increased the total number of participants by using a *fat-tree overlay*. In a fat-tree [120], processors are located on the leaves and internal nodes relay data for all their children; each new layer in the tree increases the number of possible connections exponentially. The main benefit in the context of the Web is to remove the need to relay data on dedicated servers by employing intermediate nodes in a fat-tree as relays.

Existing work on fat-trees [120, 170, 70, 27] has not so far provided solutions for quick scaling. The key issues are to quickly distribute the newer participants among the existing leaves and quickly route, through the fat-tree, the multiple messages generated by WebRTC to open a new connection. To address both, we propose a novel routing scheme, which we call Genet,\(^2\) that only requires *local information* to route connection messages: this eliminates the latency that would otherwise be incurred by waiting for the status from other parts of the tree. The destination for messages is derived from the hash value of the combined identifiers of the source and the current routing node, providing two properties. First, the scheme *deterministically routes* multiple messages sent by a new participant to the same leaf node. Second, the scheme ensures *probabilistic balancing* of newer connections between all the children to keep the tree balanced. This design is especially suited to the context of

\(^1\)Such as the electron-webrtc [23] library for Node.js.

\(^2\)Clonal colony of plants in which all individuals share the same genetic material. We expect the design, if it is successful, to eventually form a dynamic forest of overlays on the Web.
compute-intensive applications that leverage volunteers’ devices because users tend to add local devices first before asking for help from others; the devices in the first layer of the tree will therefore also benefit from the largest available bandwidth.

In this Chapter, we present the design of Genet (Section 7.1), we explain how we integrated it in Pando to increase Pando’s scalability (Section 7.2), we evaluate the balancing, scaling, and connectivity potential of the Design (Section 7.3), we discuss the results (Section 7.4), and we compare it to related work (Section 7.5). We finally conclude with a summary of the Chapter (Section 7.6).

7.1 Design

Our Genet overlay organizes participants in a tree to increase the number of concurrent connections that can be made to a single origin, while bounding the number of concurrently active WebRTC connections each participant maintains. To establish a WebRTC connection, participants exchange signals, or possible connection endpoints, with one another to determine how to connect through the Network-Address Translation (NAT) schemes used by routers. The ICE signalling protocol [194] used by WebRTC uses a trickle mode in which signals are sent as they are discovered. This reduces the latency to open the connection compared to waiting for all endpoints to be identified. The trickle mode generates multiple messages that need to be routed through the tree to exactly the same destination node. Moreover, to minimize the latency and make the tree grow quickly, the depth of nodes should be minimized by making the number of children in sibling sub-trees similar.

Our solution solves both problems while requiring only information available locally in each node. Each node maintains a list of at most ChildrenLimit children, a deployment parameter with a default of 10. Children are added in that list in the order in which they connect and keep the same index until they either disconnect or crash. As illustrated in Figure 7.1, when a new participant joins the tree, the candidate first opens a WebSocket channel to the Relay Server and creates a random identifier id (Step 1). It then sends multiple join requests that each contain its identifier (origin) and one of the WebRTC ICE signals to the Root (Master) node (Step 2). From there, each node has two choices. In the first case, if it has less children than ChildrenLimit, it assigns the candidate to one of its children and attempts to open a WebRTC connection using the candidate’s signals. During the opening, it will generate signals of its own that are sent as replies to the candidate through the Relay Server (Step 3). Signals are exchanged by both parties until a direct WebRTC connection is established, after which the WebSocket connection of the candidate is terminated (Step 4). In the second case (not illustrated), the node delegates the requests to one of its children. If the
WebRTC connection to the child is not yet open, the requests are held until the connection is established and then forwarded.

Each node makes routing decisions for delegation by taking the origin identifier, \(\text{xor}\)ed with the node’s identifier \(id\), the hash of the result is taken, and then the numerical index of the child in the children list is computed by taking the modulo \(\text{ChildrenLimit}\):

\[
\text{childIndex} = \text{hash}(\text{originId} \text{ xor} \text{ nodeId}) \text{ mod} \text{ChildrenLimit}
\]

![Genet’s WebRTC Bootstrap Sequence](image)

Fig. 7.1 Genet’s WebRTC Bootstrap Sequence. The steps are marked with numbered diamonds.

The \(\text{xor}\) of the originId and nodeId is not strictly necessary, a concatenation of the bits of both identifiers could work also. The advantage of the \(\text{xor}\) function is to provide a result with the same number of bits as the identifiers, which may be useful when all operations need to be performed in fixed-width registers.

This routing scheme has three interesting properties. First, routing is deterministic: requests from the same origin are routed to the same child at every step of the tree. Second, the choice of a good hash function ensures probabilistic balancing of newer connections between the children. Third, by using only information locally available in each node, the routing decisions are quick to make because they don’t need global information about the tree, which enables a quick scaling of the tree on startup.
In some cases, nodes may fail and suddenly disconnect during execution. In those cases, their children, once they have detected the failure, will in turn disconnect their own children (if they have some) and all disconnected nodes will try to reconnect to the root. In other cases, the WebRTC connection may fail to successfully open. Then, the parent node will remove the potential candidate after a configurable timeout, with a default value of 60 seconds. If there were any pending requests that were held for the failed candidate, the current implementation will continue holding them until a new candidate joins and successfully connects\(^3\), after which the pending requests will be delegated.

When deploying the scheme on a local network, it is possible to combine in the same process the Root node and the Relay server. On a wide-area network however, it is important that the Relay Server has a publicly-facing IP address to enable direct WebSocket connections.

Our implementation performs a routing optimization to accelerate the exchange of messages: to reply to signals, a node opens a direct WebSocket connection to the Relay server. Then if a candidate receives the first reply-signal before having submitted all its own signals, the candidate will use the origin of the reply as a destination for all subsequent signals. This optimization therefore skips some of the routing steps for the late signals. It is however not necessary, another variation that minimizes the number of WebSocket connections to the Relay Server by routing all replies through the Root would also work. We have made our JavaScript implementations of both the Genet algorithm \([114]\) and the relay server \([113]\) available as reusable libraries for Node.js and the browser.

### 7.1.1 Genet’s Node Definition

We provide the interface and properties of a Genet Node in Abstraction 4. The abstraction is instantiated with a *ChildrenLimit* and a *ReportingInterval* and returns a new node, not yet connected to a tree. It may either be used as the root of a new tree, by invoking `becomeRoot`, or connected to an existing tree by invoking `join` with a reference to the `Root` of that tree. A node may leave at any time by invoking `leave`. After joining, `parentConnected` will eventually be invoked which provides a duplex channel to communicate with the parent. This channel has its own error reporting mechanisms that is compatible with *pull-streams* modules. Each time a new child becomes connected, `childConnected` is invoked, with a symmetric duplex channel to communicate with the child, also compatible with pull-streams. The current number of leaf nodes is reported through `leafNumber` every `ReportingInterval`, the `id` used refers to either the node itself or any of its children. The properties follow from the design of the previous section.

\(^3\)A future version could optimize this case by promoting one of the pending requests to a candidate right away.
Abstraction 4: Node Definition

Parameters

*ChildrenLimit*: Maximum number of children on each node.

*ReportingInterval*: Time interval between two status reports.

Returns

*Node*: An *Genet* node, not yet connected to the tree.

Requests

*Node*: `becomeRoot()` : Become the root of the tree. Implicitly obtains identifier *Node.id*.

*Node*: `join(Root)` : Join the tree from Root. Is not invoked on the Root. Implicitly obtains identifier *Node.id*.

*Node*: `leave()` : Gracefully leave the tree.

Indications

*Node*: `parentConnected(id,duplex)` : The parent node is now connected. Messages can be sent to and received from the parent with *duplex*, compatible with pull-streams. Is never triggered on the root node.

*Node*: `childConnected(id,duplex)` : Child *id* is now connected. Messages can be sent to and received from child *id* with *duplex*, compatible with pull-streams.

*Node*: `leafNumber(id,n)` : reports the current number of leaf nodes of *id* every *ReportingInterval*. *id* is either *Node.id* or one of its children *id*.

Properties

1. *Dynamic*: Children may join, leave, or crash-stop at any time.

2. *Bounded-Degree*: Each node may have at most *ChildrenLimit* children. Beyond that limit, newer nodes are grafted onto children.

3. *Probabilistic Balancing*: The tree is balanced using a hash function applied to the random identifier of joining candidates. On average each children of a node has a similar number of children.

4. *Quick Bootstrap through Deterministic Routing of Join Requests*: Join requests are deterministically delegated through the tree using only the identifier of *node* and of a candidate for quick routing (which is needed by WebRTC because information on how to connect is sent in multiple messages).
7.1.2 Usage and differences with the JavaScript implementation.

Its usage in JavaScript is illustrated in Figure 7.2. There are two major differences and some minor differences compared to the abstraction definition. First, in the current implementation of Pando, the reporting of the number of leaves is done outside of the `webrtc-tree-overlay` library that implements the abstraction, by opening an additional WebRTC channel. This was done because the original design goal for the library was to have a flexible minimal library for creating a spanning tree that could be used for many different applications. The two functionalities have been fused in our presentation to simplify the implementation of the Node in the next section. Second, the `Root` is not directly accessible with a reference; communications need to go through the publicly accessible Pando-Server. This is abstracted in the implementation of the next section to make it simpler. Finally, the interface has minor differences compared to Abstraction 4 to make it more idiomatic to the language: the Pando-Server is passed as an argument to the constructor and the `id` of a duplex channel is stored on the channel object itself rather than passed as an additional parameter. The implementation also includes an additional security mechanism: the `becomeRoot` method requires an additional 'secret' to prevent connections by unauthorized nodes on the public Pando-Server.
7.1 Design

// On the root
var pandoServer = require('webrtc-bootstrap')('server hostname or ip:port')
var Node = require('webrtc-tree-overlay')

// becomeRoot requires an additional secret string to prevent unauthorized connections
var root = new Node(pandoServer).becomeRoot('secret')
root.on('child-connect', function (duplex) {
  // id on the duplex channel rather than a separate parameter
  duplex.id
  duplex.send('ping')
  duplex.on('data', function (data) {
    console.log(data)
  })
})

// On a child node
var pandoServer = ...
var Node = ...

// Root is implicitly known by the Pando-Server
var node = new Node(pandoServer).join()
node.on('parent-connect', function (duplex) {
  duplex.on('data', function (data) {
    console.log(data)
    duplex.send('pong')
  })
})
node.on('child-connect', function (duplex) {
  duplex.send('ping')
})

Fig. 7.2 Genet Node Usage in JavaScript. There are differences compared to Abstraction 4, which are explained in Section 7.1.2.
7.1.3 Genet’s Node Implementation

The implementation that follows is a simplified version of the actual implementation to make the presentation shorter and clearer. In the actual implementation, communications between the nodes go through two different mechanisms: when a node is not yet connected to the tree, communications happen over WebSockets by using the Pando-Server as a relay. This is used to bootstrap a direct WebRTC connection that replaces the WebSocket relay after it has been established. We call this first WebRTC connection, a control channel because it is used for managing the overlay. This mechanism is implicit in our presentation: nodes can send and receive messages simply by holding a reference to one another. The implementation still shows how the WebRTC signals are propagated to open a duplex channel but it illustrates it for the data channel that is made available by the abstraction (duplex), rather than the control channel that is first opened by the actual implementation and then used for opening the data channel. However, the process is the same so it should be easy to transpose. The implementation of a Node is split in four parts: the first part for the initialization, the second for the protocol by which a node joins the tree, the third for reporting on the number of leaf nodes, and the fourth for handling termination and failures.

In Part 1 (Algorithm 16) the internal data structures and the timer for reports are first initialized. A node then has a choice of becoming root with becomeRoot or joining an existing tree with join. It self-assigns an id in both cases. becomeRoot does nothing else. It is shown here as a placeholder: in the actual implementation it opens a connection to Pando-Server to connect the root to a publicly accessible server. join uses a reference to a Root node to join its tree: it creates a new WebRTC peer connection which will generate signals. Those signals trickle over time and are sent to the Root with joinRequests as soon as they are generated. They will be used by a node in the tree, which will become the parent, to open a duplex channel. Once the channel is ready it is made available through the parentConnected indication.

Part 2 (Algorithm 17) shows the deterministic routing and probabilistic balancing protocol for joining. Starting from the Root, when a node receives a joinRequest it has three choices. First, if the signal has been sent from a Candidate node that is already in the process of joining the current Node, it is passed to the corresponding WebRTC duplex channel. Otherwise, if there are still available slots for a new child, it is added to the internal data structures. A new duplex channel is created with the first received signal. The other endpoint signals generated by this side of the channel are passed by to the Candidate with a joinReply and once the duplex channel is ready, it is made available with the childConnected indication. Thirdly, if there are no more slots available for children, the joinRequest is delegated to one of the children with a deterministic computation. The last piece of the implementation
Algorithm 16 Node Implementation Part 1 (Initialization)

1: \(\text{Parent} \leftarrow \text{null}\) \hspace{1cm} \(\triangleright \) Reference to the Parent Node
2: \(\text{parentId} \leftarrow \text{null}\) \hspace{1cm} \(\triangleright \) Parent’s Id
3: \(\text{parentDuplex} \leftarrow \text{null}\) \hspace{1cm} \(\triangleright \) Parent WebRTC data channel
4: \(\text{child} \leftarrow []\) \hspace{1cm} \(\triangleright \) Array of (ChildNode, Id)
5: \(\text{childDuplex} \leftarrow []\) \hspace{1cm} \(\triangleright \) Array of children’s WebRTC data channels
6: \(\text{childrenNumber} \leftarrow 0\)
7: \(\text{leafNumber} \leftarrow []\) \hspace{1cm} \(\triangleright \) Array of children’s number of leaf nodes
8: \textbf{trigger} \(\text{Timer}: \text{setInterval}(\text{ReportingInterval})\)
9:
10: \textbf{upon} \(\text{Node}: \text{becomeRoot}()\)
11: \hspace{1cm} \(\text{id} \leftarrow \text{random}()\)
12:
13: \textbf{upon} \(\text{Node}: \text{join}(\text{Root})\)
14: \hspace{1cm} \(\text{id} \leftarrow \text{random}()\)
15: \hspace{1cm} \(\text{parentDuplex} \leftarrow \text{new WebRTC peer connection}\)
16: \hspace{1cm} \textbf{thread}
17: \hspace{1cm} \textbf{for all} signal \textbf{in} parentDuplex’s signals \textbf{do}
18: \hspace{1.5cm} \textbf{trigger} \(\text{Root}: \text{joinRequest}(\text{id, signal, Node})\)
19: \hspace{1cm} \textbf{upon} parentDuplex opened
20: \hspace{1.5cm} \textbf{trigger} \(\text{Node}: \text{parentConnected}(\text{id, parentDuplex})\)
21:
22:
23:
initializes the Parent information on a reply and gives the signals to the opening duplex channel.

**Algorithm 17** Node Implementation Part 2 (Join Protocol)

```
24: upon Node: joinRequest(origin, signal, Candidate)  ▷ Try new signal
25:    if (Candidate, origin) ∈ child then  
26:        give signal to childDuplex[origin]  
27:    else if childrenNumber < ChildrenLimit then  ▷ New candidate
28:        childrenNumber ← childrenNumber + 1
29:        childIndex ← next free index from 1 to ChildrenLimit
30:        child[childIndex] ← (Candidate, origin)
31:        leafNumber[origin] ← 1
32:        childDuplex[origin] ← new WebRTC peer connection
33:        give signal to childDuplex[origin]
34:        upon childDuplex[origin] opened
35:            trigger Node: childConnected(origin, childDuplex[origin])
36:    thread
37:       for all signal in childDuplex[origin]'s signals do
38:           trigger Candidate: joinReply(id, signal, Node)
39:    else  ▷ Delegate to a child
40:        delegateIndex ← (hash(id ∧ origin)%ChildrenLimit) + 1
41:        (Delegate, id) ← child[delegateIndex]
42:        trigger Delegate: joinRequest(origin, signal, Candidate)
43:    upon Node: joinReply(id, signal, ParentNode)
44:        if Parent = null then
45:            Parent ← ParentNode
46:            parentId ← id
47:            give signal to parentDuplex
```

Part 3 (Algorithm 18) reports the number of leaf nodes. Every tick of the Timer, the number is reported both from the abstraction and to the Parent node. If Node has no children, it is therefore a leaf and a count of 1 is reported. Otherwise, it is a middle node and the sum of the counts that were last received for all the children is reported.

Part 4 (Algorithm 19) shows termination and failure handling. When a Node leaves or encounters an error, it disconnects from its Parent, empties the internal data structures, destroys all currently open channels and stops its timer for reporting. However, if the failure is suspected on a child, the child is removed but the rest continues. The same logic applies.
whether the failure is detected on a Node or on its WebRTC duplex channel. If the parent of a node wrongly suspects a node has failed, the node will still stop and recursively disconnect all its children. The reconnection, if necessary, has to be handled by the user of the abstraction by creating a new instance.
Algorithm 19 Node Implementation Part 4 (Termination and Failure Handling)

65: upon Node:leave()
66: Parent ← null
67: parentId ← null
68: destroy parentDuplex
69: leafNumber ← []
70: for all (ChildNode,origin) in child do
71: REMOVECHILD(ChildNode)
72: trigger Timer:stop()
73:
74: procedure REMOVECHILD(ChildNode)
75: if ∃_id(node,id) ∈ child then
76: trigger ChildNode:leave()
77: remove the entry with value (node,id) from child
78: remove the entry with index id from leafNumber
79: childrenNumber ← childrenNumber - 1
80: destroy childDuplex[id]
81:
82: upon FailureDetector:Suspect(TreeNode)
83: if Parent = TreeNode then
84: trigger Node:leave()
85: else
86: REMOVECHILD(TreeNode)
87:
88: upon parentDuplex closed, failed, or connection timeout
89: trigger Node:leave()
90:
91: upon childDuplex[id] closed, failed, or connection timeout
92: ∀(ChildNode,id') from child such that id' = id, REMOVECHILD(ChildNode)
93:
7.2 Integration with Pando

We implemented a scalable version of Pando based on the previous implementation of Genet, written in JavaScript [114]. When a new browser window, executing on the device, successfully connects, it first joins as a leaf in the fat-tree and computes results, therefore acting as a processor. When additional browser windows join beyond the ChildrenLimit of the root, the extras connect to the current leaves. The leaves then stop computing and instead start relaying data and results, becoming coordinators. The process repeats at every level of the tree with new devices joining. We have successfully tested a thousand participants (Section 7.3) but the design should allow potentially millions of devices to connect in a single overlay, the limiting factors being the bandwidth available on the root node and the number of concurrent connections supported by the Relay Server, which determine the joining rate.

The implementation of Pando using the Genet overlay follows a recursive structure. Our scalable implementation re-uses the StreamLender abstraction (Section 4.4) on intermediary nodes of the fat-tree, as illustrated in Figure 7.3, enabling failures to be handled in the parent of a failing node. Intermediary nodes may also fail. In that case, the parent node will handle the failure by re-submitting the values in a different sub-tree.

As in the original implementation of Pando, we also regulate the flow of values to the WebRTC channels by using the Limiter module. However, in contrast to the previous static configuration, the limit is dynamically adjusted using the periodic reports on the number of children in the sub-tree to adjust the flow to a growing or shrinking fat-tree.

Using the previous design, the fat-tree overlay enables larger total throughput for Pando while providing a quick speed of deployment.

7.2.1 Scalable DistributedMap Definition

The Scalable DistributedMap (Abstraction 5) is instantiated with the Root of the tree overlay, a processing function $F$ to apply to all values of the stream, and a BatchSize to determine the maximum number of values to send per available processor. It returns a pull-stream transformer. It has a number of properties derived from those of the Limiter (Abstraction 1), StreamLender (Abstraction 2), and the Node (Abstraction 4).

The properties are the same as for the DistributedMap (Section 4.5.1) but have been augmented with the additional properties that are provided by the Genet Fat-Tree Overlay. The bounded connections derives directly from the bounded degree of Genet nodes. The logarithmic latency derives directly from the probabilistic balancing of the tree. A balanced tree will have a logarithmic height in relation to the number of nodes in the entire tree and
Fig. 7.3 Scalable Pando.
therefore the communication latency in number of steps between the root and the leaves will also be logarithmic. The quick bootstrap comes from the quick bootstrap of a Genet node.

The usage of *Scalable DistributedMap* in JavaScript is illustrated in Figure 7.4. Its interface is similar to the abstraction definition.

```javascript
var pull = require('pull-stream')
var distMap = require('pando-computing')
var root = ... // Create a Node
var f = function (x, cb) { cb(null, x*x) }
var mapper = distMap(f, { root: root, batchSize: 1 })
pull(
  pull.count(10),
  mapper,
  pull.drain()
)
mapper.on('url', function (err, url) {
  // Open url in a browser
})
```

Fig. 7.4 *DistributedMap* Usage in JavaScript.

### 7.2.2 Scalable DistributedMap Implementation

The implementation is provided in two parts, the first part (Abstraction 14) executes on the Master process and the second part (Abstraction 15) executes in Worker processes on the same physical machine or on a separate device. In our implementation, the Master process is a Node.js commandline tool and the Worker processes execute in browser tabs (Section 3.4.3).

Both implementations are almost identical. They instantiate a *StreamLender* to borrow values from the stream and connect it with their inputs and outputs. On the Master process the input and outputs correspond to those of the transformer. In volunteers, they correspond to the input and output of a *duplex channel* that transitively connects them to the Master.

Both implementations process values locally. On the Master, the processing starts as soon as the abstraction is instantiated. On the volunteers, it starts as soon as they are connected to their parent (coordinator). They both use the *AsyncMap* module of the pull-stream core library [164] to process the values.

Both implementations change their behaviour after a child connects to them. They pause local processing of values and instantiate a new sub-stream to distribute values and collect results from the newly connected child.\[^4\]

[^4]: This can be implemented as a pull-stream module.
**Abstraction 5** Scalable DistributedMap Definition

**Parameters**

- **Root**: Root node of the tree overlay.
- **F**: Processing function to apply to all values.
- **BatchSize**: Maximum number of values to send per processor.
- **ChildrenLimit**: Maximum number of children on each node.
- **ReportingInterval**: Time interval between two status reports.

**Returns**

- **ScalableDistributedMap**: transformer with input *Input* and output *Output*

**Properties**

1. **Unbounded**: There is no upper bound on the number of processing nodes.
2. **Dynamic**: Processing nodes may join, leave, or crash-stop at any time.
3. **Lazy**: Inputs are requested in proportion to the number of currently available processors. The flow rate of newer values sent is proportional to the flow rate of the results received.
4. **Conservative**: An input is processed by at most one processing node at a time.
5. **Fault-tolerant**: As long as at least one processing node is correct and keep making progress, all inputs should eventually have a corresponding result even if other processing nodes fail (done or error before producing all their results).
6. **Ordered**: All results are provided in the order of the corresponding inputs.
7. **Capability-balanced**: Faster processors receive more values to process (as long as *BatchSize* is large enough to keep them busy).
8. **Bounded connections**: The maximum number of connections on each processing node is bounded.
9. **Logarithmic latency**: The communication latency in number of steps between the processors and *DistributedMap* is logarithmic in the number of processors.
10. **Quick Bootstrap**: The processing tree can grow quickly when new nodes join.
Algorithm 20 Scalable DistributedMap Implementation (Master Process)

1: DistributedMap input is Input \hfill \triangleright \text{Definitions}
2: DistributedMap output is Output
3: Root $\leftarrow$ Node(ChildrenLimit, ReportingInterval).becomeRoot()
4: serve $F$, BatchSize and Root on Pando-Server
5: Lender $\leftarrow$ StreamLender() \hfill \triangleright \text{Start as a Processor}
6: limit $\leftarrow \emptyset$
7: Input $\Rightarrow$ Lender $\Rightarrow$ Output
8: $S_1I, S_1O \leftarrow$ Lender.lendStream()
9: LocalStream $\leftarrow (S_1O \Rightarrow AsyncMap(F) \Rightarrow S_1I)$ \hfill \triangleright \text{Use existing map module}

10:
11: \textbf{upon} Root.childConnected($id, duplex$) \hfill \triangleright \text{Become a Coordinator}
12: \hspace{1em} pause LocalStream
13: \hspace{1em} limited $\leftarrow$ Limiter(duplex)
14: \hspace{1em} $S_1I, S_1O \leftarrow$ Lender.lendStream()
15: \hspace{1em} $(S_1O \Rightarrow limited \Rightarrow S_1I)$
16: \hspace{1em} limit[$id$] $\leftarrow$ limited

17:
18: \textbf{upon} Root.leafNumber($id, n$)
19: \hspace{1em} \textbf{if} $id = $ Root.$id$ \textbf{and} $n = 0$ \textbf{then}
20: \hspace{2em} resume LocalStream \hfill \triangleright \text{Resume being a Processor}
21: \hspace{1em} \textbf{else if} limit[$id$] \textbf{exists} \textbf{then}
22: \hspace{2em} trigger limit[$id$]:updateLimit(BatchSize * $n$) \hfill \triangleright \text{Adapt flow}

23:
Algorithm 21 ScalableVolunteer Implementation (Worker Processes)

1: retrieve $F$, BatchSize, and Root from Pando-Server
2: Volunteer $\leftarrow$ Node(ChildrenLimit, ReportingInterval)
3: Lender $\leftarrow$ StreamLender() ▷ Prepare Processor
4: limit $\leftarrow$ []
5: LocalStream $\leftarrow$ EmptyStream
6: trigger Volunteer:join(Root)
7: 
8: upon Volunteer:parentConnected(id,duplex) ▷ Start as a Processor
9:    Output,Input $\leftarrow$ duplex
10:   Output $\Rightarrow$ Lender $\Rightarrow$ Input
11:   $S_1I,S_1O \leftarrow$ Lender.lendStream()
12:   LocalStream $\leftarrow (S_1O \Rightarrow \text{AsyncMap}(F) \Rightarrow S_1I)$
13: 
14: upon Volunteer:childConnected(id,duplex) ▷ Become a Coordinator
15:    pause LocalStream
16:    limited $\leftarrow$ Limiter(duplex)
17:    $S_1I,S_1O \leftarrow$ Lender.lendStream()
18:    $(S_1O \Rightarrow \text{limited} \Rightarrow S_1I)$
19:    limit[id] $\leftarrow$ limited
20: 
21: upon Volunteer:leafNumber(id,n)
22: if id $=$ Volunteer.id and n $= 0$ then
23:    resume LocalStream ▷ Resume being a Processor
24: else if limit[id] exists then
25:    trigger limit[id]:updateLimit(BatchSize * n) ▷ Adapt flow
26:
the flow rate on Limiter is adjusted accordingly. And if all children disconnect, either by gracefully leaving or suddenly crashing, local processing is resumed. Fault-tolerance is implicit in the pull-stream protocol used by the duplex channels. The only difference between the two implementations is that the Master has a connection to the Root of the tree while the volunteers join the tree from a Node.

The F, BatchSize, and Root references are communicated through the Pando-Server. Many different mechanisms are possible; in our implementation we use the HTTP protocol with JavaScript files created at the time DistributedMap is instantiated.

This completes the presentation of the scalable abstractions enabled by Genet.

7.3 Evaluation

In the next sections, we evaluate the behaviour of the design, both in simulation over a large number of experiments, and in real-world deployments, over a smaller number of experiments.

7.3.1 Depth with Probabilistic Balancing

We first study the impact of choosing a probabilistic balancing scheme on the depth of the fat-tree under various levels of failure, because the depth has a direct impact on the latency of communication between the root and the leaf nodes.

How deep is the fat-tree?

$N$ nodes in a perfectly balanced tree are at depth less or equal to $\lfloor \log(N) \rfloor$. Because they are distributed randomly in our fat-tree, a certain percentage of nodes are deeper. To quantify the percentage of nodes that may be affected, we simulated the construction of the tree with nodes with random identifiers that join one after the other, assuming all nodes do not crash. We then counted the number of nodes in the extra levels, and repeated the experiment a thousand times.

Over a thousand experiments, we observed no nodes two levels deeper, which while possible in theory is in practice extremely unlikely. The proportion of nodes at depth $\log(N) + 1$ varied between experiments. The results are shown in Figure 7.5, as a cumulative distribution function for various sizes of trees, to provide both intuitions about the average behaviour and the maximum cases. Our results show that in a majority of experiments ($\geq 700$), 8% or less nodes are on the extra level of the tree, regardless of the number of nodes in the tree. They also show that in all cases, 17% or less of nodes were in the extra level.
Moreover, the larger the tree, the closer all experiments get to around 7.5% of nodes in the extra level. Therefore, in all experiments, \( \approx 83\% \) of nodes are located no deeper than they would have been if the tree had been fully balanced.

![Fig. 7.5 Number of Experiments with X% or Less of Nodes at Depth \( \log(N) + 1 \) over 1000 Repetitions and No Failures.](image)

**Do failures make it deeper?**

In practice, a certain number of nodes *will fail* and force their children to reconnect. To quantify the impact, we construct a tree as in the previous experiment but then disconnect a certain percentage of nodes, then let all nodes reconnect through the root. We then count the number of nodes at deeper levels than \( \log(N) \). We performed a thousand experiments for trees of size 10, 100, 1000, and 10000 under various probabilities of failure (\( F \)) from 0 to 1.

Over a thousand experiments, we observed no nodes at depth \( \log(N) + 2 \). In all cases, the failures did not affect the percentage of nodes at depth \( \log(N) + 1 \). Results with a 25% probability of failure are shown in Figure 7.6; the results are the same for other levels of probability. Failures therefore do not change the distribution of nodes through the tree.

Our probabilistic balancing scheme therefore achieves equivalent depth as a deterministic algorithm for at least 83% of nodes, in the presence of failures or not. For larger trees, of a
7.3 Evaluation

Fig. 7.6 Number of Experiments with X% or Less of Nodes at Depth $\log(N) + 1$ over 1000 Repetitions and a Failure Probability of 0.25 for Nodes to Fail.

thousand nodes or more, this percentage increases to 92.5%: *scale therefore increases the effectiveness of the scheme* (up to a limit).

### 7.3.2 Bootstrap Latency When Scaling on Local Networks

How quickly does the Genet fat-tree scale in practice? We first measure the latency in establishing a WebRTC connection as a baseline and then measure the added overhead of our scheme to connect all nodes, as a function of the size of the fat-tree, with fat-trees of size 10 to a 1000 nodes. We performed these measurements on Grid5000 [17] because it is representative of deployments on a local area network, such as those of a university or a large organization, the infrastructure was accessible to us, it facilitates the replication of experiments, and it can easily scale the number of participating nodes.

For our experiments, we used Grid5000 nodes from the Grenoble site, that has two models of nodes. The first model (dahu) are based on the Dell PowerEdge C6420 that uses Intel Xeon Gold 6130 CPUs (Skylake, 2.10GHz, 2 CPUs/node, 16 cores/CPU), have 192GB of memory and are connected by 10 and 100 Gbps network links. The second model (yeti) is based on the Dell PowerEdge R940 also with Intel Xeon Gold 6130 CPUs (Skylake, 2.10GHz, 4 CPUs/node, 16 cores/CPU), have 768GB of memory and are also connected by 10 and 100 GBps network links. The exact distribution of nodes for experiments is chosen randomly between the two models, based on availability (because other experiments
concurrently run at the same time on other machines), in our case it was almost always the dahu nodes that were used, with an occasional yeti node in the mix. All our throughput experiments were also made with these nodes.

**How long does it take to establish a single WebRTC connection?**

While individual nodes on Grid5000 enjoy a sub-millisecond latency, the ping between nodes is typically between 0.1 and 0.2 ms, establishing a WebRTC connection is significantly slower. As explained in Section 7.1, each participant in a connection first starts listing potential connection end-points that can enable Network-Address Translation, also contacting STUN servers in the process. For example, the Google STUN server we use (stun.l.google.com) has an average ping latency of 35ms. The endpoints need to be exchanged between participants through a relay and finally, multiple connections are tried from both sides until one is found to work. Between nodes on a local network, the connection can be established earlier because some of the endpoints will use the local IP address and therefore a connection on the local area network can be established before the other endpoints discovered by the STUN server are received. Nonetheless, the messages exchanged with the relay server significantly adds to the delay.

In our tests, the relay server was running on the local network and connected 20 browser windows on 10 Grid5000 nodes forming a fully connected clique, each window opening a connection to every other window. All connection attempts succeeded. For all connections, we measured the latency between creating the connection and a confirmation message sent through the connection, which corresponds to the time it takes before data starts flowing through the connection. We used webrtc-connection-testing [118] version 4.0.0, an open source tool we built for this task. The results are shown in Figure 7.7. We observed connection latencies less than 1000ms, with 95.5% of connections taking less than 500ms. Of the 363 connections that took less than 500ms, 41 took less than 100ms, 112 took between 100ms and 200ms, 132 took between 200ms and 300ms, and 78 took between 400ms and 500ms (not shown on the figure).

Therefore, even if Grid5000 nodes have sub-millisecond ping latencies, establishing a WebRTC connection is slow and typically takes three to four orders of magnitude longer, up to 1000ms. As the implementation of WebRTC is part of the implementation of the browser, any overlay design executing in JavaScript is subjected to this constraint and will be fully deployed at the speed at which the slowest connections are established.
7.3 Evaluation

How long does it take to connect all nodes in a fat-tree?

Fully deploying a fat-tree, in which nodes are organized in multiple layers, shall therefore take at least time that is proportional to the depth of the tree and the time it takes to establish the slowest connections. In this section and the next, we show it is the case in practice.

We used Pando version 0.17.9 [106] for our tests, which implements the design of Section 7.2. The fat-tree is used to distribute inputs to processors and retrieve back results. We used a fat-tree of degree 10, which means that each node has at most 10 children and each layer of the tree will have a multiple of 10 participants in it. We used a test application that waits for 1 second then returns the square of the input value for a number of reasons. This removes the impact of potential differences of CPUs speeds, making it easier to determine when all processors are connected and producing results because the overall throughput, in values per seconds, is equal to the number of participating processors (on the leaves of the fat-tree). In turn, this means our measurements really represent the coordination overhead of the entire system. And finally, the time it takes to reach the full throughput represents the bootstrapping latency.

We deployed the fat-tree on 10 different Grid5000 nodes on the Grenoble’s site by progressively increasing the number of browser windows executing on each node, with 1, 5, 10, 25, 50, and 100 windows. We repeated each experiment five times. Each node was connected with a one second delay after the previous, e.g. the first node opens its browser windows after a 1-second delay, the second with a 2-seconds delay, etc. up to 10 seconds for the last node. While opening a large number of browser windows at the same time (each executing in their own operating system process) from the same node worked fine, launching browsers at the same time from multiple nodes led to connection errors, which prompted
the addition of an artificial delay between Grid5000 nodes. The rate of connection for 10 browser windows is therefore 1 browser window per second for 10 seconds, while the rate of connection for 1000 browser windows is 100 browser windows per second for 10 seconds. We have not found the exact reason for this issue. Should the issue be solved in future versions, we expect our connection delay to decrease accordingly. The performance results we report are therefore conservative: i.e. it is possible the design could perform even better than what we report.

As the fat-tree is deploying, periodic status updates are sent from the leaves to the root node to report on the current state of the fat-tree. We used an interval of 3 seconds between reports, therefore the state of the fat-tree may be known at the earliest 3 seconds after having changed. This means the latency that we report in the next figures represent an upper bound on the actual latency to connect the nodes. We measured the time it takes until all browser windows were counted as children in the tree, both the coordinator and processor nodes.\(^5\) We then also measured the throughput of squared values at the output of Pando, also by sampling at intervals of 3 seconds. We measured the time it takes until the throughput corresponds to the number of leaves in the fat-tree, as reported by the previous reporting mechanism. The throughput measurements are independent from the reporting strategy, and the latency measured really represents the time observed by a user of Pando until full throughput is achieved. Both results are shown in Figure 7.8.

---

\(^5\)The reporting mechanism is distinct from the calculation of the number of leaves presented in the previous sections. Every reporting interval, a node reports on the number of children connected to itself. The intermediate node is counted by its parent, therefore the root report on the number of children accurately represents the total number of browser tabs connected.
For 10 browser windows, it typically takes about 15 seconds to connect them all in the first layer of the fat-tree. This is about 5 seconds longer than the 10 seconds required to open all browser windows; the additional delay can be explained by a 1-2 second delay before Pando’s server is ready after start-up, the maximum latency of 1000ms to establish the slowest connections, as reported in the previous experiment, and the reporting interval, i.e. we learned of that last connection 5 reports after Pando was started. The maximum throughput is first observed one sample later, at 18 seconds, as shown by the ‘Output Latency’ curve. As the size of the tree increases, so does its depth and therefore the latency to fully connect the fat-tree. At 100, 250, and 500 browser windows, it takes about 18 seconds to fully connect the children, while at 1000 browser windows it takes 24 seconds. The latency to reach maximum throughput follows accordingly by one or two samples, 21 seconds for 50 and 100 browser windows, 24 seconds for 250 and 500 browser windows, and 28 seconds for 1000 browser windows. The variation between experiments also grows larger as the size of the tree grows, we measured a latency of up to 54 seconds both for connecting children and reaching maximum throughput in a single experiment. We can therefore conclude that it typically takes 30 seconds to connect all nodes in our WebRTC fat-tree and sometimes up to a minute on the Grid5000 testbed.

7.3.3 Throughput Ramp-up in the Collatz Application

Now that we have established a typical latency of 30 seconds to reach maximum throughput, does the fat-tree, when used with Pando, behave in the same way when the leaf nodes are actually performing computations? We answer that question by taking one representative application of volunteer computing, the Collatz application (Section 5.1.1). For the purpose of measuring the scaling behaviour, the single core performance is not critical, a faster implementation shall increase our throughput measurements by a constant factor, which would be obtained by better using the CPU, while not affecting much the scaling behaviour, which is instead due to the coordination performed by the fat-tree.

Studying the throughput scaling behaviour on actual applications is complicated by the fact that all tasks do not take the same amount of time. The throughput at the output of Pando can vary both because nodes join or because tasks are temporarily faster or slower. Moreover, as our fat-tree design probabilistically balances the tree, the actual number of leaves that are processing inputs varies between experiments. It is therefore harder to determine when all connected nodes have started contributing to computations. We therefore first measured the average throughput with a given number of nodes, and let the deployment compute for at least three minutes. We then took the average throughput measured after all the nodes were connected and counted the number of participating processors (leaves). We measured for 10
Grid5000 nodes, with 1, 16, and 32 browser windows per node, the last being the maximum number of cores available on the machines of the Grenoble site. The results are shown in Figure 7.9.

![Average Throughput on the Collatz Application on Grid5000.](image)

As the number of browser windows increases, so does the average throughput, showing a clear benefit to scaling the number of participating cores. However, the results are not quite linear. This is actually not due to the fat-tree design but contention for CPU resources on the same machine. We did a quick second experiment with 10 browser windows on a single machine and we obtained \( \approx 480 \text{BigNums/(s \cdot node)} \) rather than the \( \approx 560 \text{BigNums/(s \cdot node)} \) we obtained with 10 browser windows on 10 nodes.

We then used the previous average throughput as a target to determine the time it takes before all cores are actually contributing results, when deployed with the fat-tree overlay. We therefore measured the time it takes until the output throughput reaches the average throughput measured previously, adjusted for the actual number of participating processors (leaf nodes in the fat-tree). We used the same methodology as in Section 7.3.2. The results are shown in Figure 7.10.

The results are consistent with the previous results, even slightly better probably because of the uncertainty added by the 3 seconds sampling interval. In this case again, reaching maximum throughput typically takes 15 seconds with 10 nodes (\( \approx 10 \) cores), 18 seconds with 160 nodes (\( \approx 110 \) cores), and 21 seconds with 320 nodes (\( \approx 220 \) cores).
Fig. 7.10 Latency to Reach Maximum Throughput on the Collatz Application on Grid5000.

### 7.3.4 WebRTC Connection Probability and Latency on the Internet

The previous results show that our WebRTC fat-tree design is effective in quickly deploying a large number of nodes on a local area network and provided a methodology for systematically studying their performance for volunteer computing applications, both of which had never been done before. In this section we provide some additional intuitions about how a deployment that targets the Internet should be adapted and show how the tools we built for the previous experiments can be used in that setting to motivate future works.

The previous results already show that the latency in establishing WebRTC connections is a significant factor in the overall latency of deploying a fat-tree, because even on a fast local network, a connection can take up to 1000 ms to be established. We tested two additional settings, one in which the relay server for exchanging the connection endpoints is located outside the local network and a second in which the participants are distributed across the planet.

We show the results of the first experiment in Figure 7.11 when establishing connections between browser windows executing on Grid5000, but relying on a remote server located in Paris, France\(^6\), for relaying signalling messages. The ping latency from Grenoble to that server takes 40ms on average and ranges from 13ms to 150ms, about 130-1500 times higher than between nodes within Grid5000. All connections succeeded also in this case. We observed similar results as for the experiment with a local server but with greater variability, with some connections taking between 1000ms and up to 16s to be established. Among the fastest established connections, 16 took less than 100ms, 123 took between 100ms and

---

\(^6\)Running on Amazon Cloud.
200ms, 82 took between 200ms and 300ms, 31 took between 300ms and 400ms, 29 took between 400ms and 500ms, 36 took 500ms to 600ms, and 17 took 600ms to 700ms (not shown in the figure), and together account for 89% of all latency results. Compared to using a local server, 22.6% less connections take less than 500ms and almost three times more take between 500ms and 1000ms. Connections therefore have additional latency as well as greater variability, as would be expected from messages routed on the Internet.

Fig. 7.11 WebRTC Connection Latency Distribution over 380 Successful Connections between Nodes on the Grid5000’s Grenoble Site using a Remote Server.

For the second experiment, we asked 20 participants randomly selected among Mechanical Turk [8] workers to open a web page that tested their WebRTC connectivity to other participants and the experimenter. Out of the 21 participants, including the experimenter, 17 chose to voluntarily disclose their location using the geolocation API of their browser. The world-wide distribution of participants is shown in Figure 7.12. Between all participants that were connected to the relay server at the same time, 398 WebRTC connection attempts were made, out of which 194 succeeded, for a success ratio of 48.7%. This shows, unsurprisingly, that random connections between participants do not always succeed. However, contrary to our initial expectations, almost half of the connections succeeded.

The latency in establishing connections, as shown in Figure 7.13, has more variation compared with the local and remote server experiments on Grid5000. Except for one result, all other connections took at least 500ms to be established and most results are well-distributed between 500ms and 8500ms. Compared to Figure 7.7 and Figure 7.11, it is therefore more typical for a participant to take several seconds to be connected.\footnote{This experiment used the previous 3.0.2 version of the webrtc-connection-testing tool [118], which can introduce an additional connection delay because participants that are already connected receive notifications of newer participants only every 5 seconds. This was fixed in version 4.0.0 (which was used in the previous}
7.3 Evaluation

Fig. 7.12 Geographical Location of Participants. Only those that accepted to share their location are displayed.

Fig. 7.13 WebRTC Connection Latency Distribution between World-Wide Participants. The figure shows a distribution of 194 successful connections.
Supposing our results generalize, which shall be validated in larger settings, this means that choosing a random Internet participant for connection shall lead to a successful connection almost half the time. However, this also highlights the need for mechanisms to tolerate failures of initial connections. One possible solution would be to first test for connectivity before deploying the fat-tree, which unfortunately would lead to a higher connection latency. A second possible solution would be to attempt multiple random connections when a participant joins.

### 7.4 Discussion

The Genet Fat-Tree overlay could be applied to other problems than volunteer computing. The most promising seems to bootstrap other overlay networks built with WebRTC. It could, for example, implement a peer sampling protocol, such as Spray [146], and the initial bootstrap could be made fast by having new nodes join multiple nodes in the tree, forming a mesh that could then progressively converge to an efficient topology. The quick scaling ability of the design we have presented is therefore complementary to potential refinements based on existing overlay designs.

### 7.5 Related Work

**Fat-tree topologies** have originally been proposed to provide high-bandwidth communication between nodes in computing clusters while minimizing the cost of switching hardware [120]. Fat-trees derive their name from the increasing bandwidth requirements on edges closer to the root because they relay traffic for all children in the underlying sub-tree. Fat-trees were later also adopted explicitly or implicitly in *overlay networks*, in which nodes connected using Internet protocols are organized in logical networks for efficient communication, to provide, for example, multicast communication [216, 27].

Extensive work on tree overlays for multicast applications has been done since the 90s [216], in which the same data is disseminated from a single source to tens and up to millions of participants. Typical applications of volunteer computing have different data transfer patterns because each participant receives a different sub-set of data. BOINC submits the same computation to a small number of participants (at least three) until a majority experiments) to send notifications as soon as participants are connected. Because we could not assemble again the same set of participants to repeat the Internet deployment, we present the original results. While the average latency could possibly be lower, we would expect to still observe an increased variability of the latencies and values of at least a few seconds.
agrees [187], while the current version of Pando does not use redundancy because the code is executed on trusted devices. In addition, in both cases, each participant will return different results to the root.

To the best of our knowledge, we are the first to propose a fat-tree overlay for scaling volunteer computing applications that supports an infinite number of inputs and provides a decentralized scheme for allocating nodes in the tree. ATLAS [16]’s tree of managers organized around work-stealing is perhaps the oldest documented scheme that relies on a tree for scalability but little details about the implementation were provided and the actual implementation was tested with only 8 machines. Javelin++ [145] relies on a tree structure to implement a distributed work-stealing scheduler but the scheme relied on tasks being finite and the position of a new node in the tree is computed from the root. Bayanihan [175] conceived a tree of servers that maps to the underlying network topology when the bandwidth on the link to a single server is insufficient, but to the best of our knowledge the scheme was never implemented. Connection decisions in our scheme do not require global information about the tree, yet they ensure probabilistic balancing and guarantee the routing of multiple connection messages to the same leaf node.

BOINC [10] currently supports hundreds of thousands of participants but relies on a dedicated server with sufficient resources and an interaction pattern that is tailored to long running computations. Volunteers obtain the task to perform and transmit the results in two different remote procedure calls. Participant failures are detected with a soft limit on the expected time to completion, which therefore requires an estimate that is application dependent. Our design is tailored to shorter running tasks and instead relies on the heartbeat mechanism provided by WebRTC to detect the failure of a participant. Moreover, by relying on WebRTC to scale up the number of concurrent connections, we can support at least a thousand participants with no investment in dedicated hardware nor renting of hosted resources.

Compared to other published volunteer computing tools, we are the first to have successfully tested with a thousand participants and the first to use WebRTC to connect participants in a fat-tree overlay. Most published volunteer computing tools [6, 18, 39, 175, 52, 129, 169, 102, 119] were tested with less than a hundred of participants. Some of the most recent have been tested with more than a hundred participants [135, 103, 133] and even up to 400 concurrent participants [50]. But the largest internet deployments of custom tools [135, 103, 133] have so far reached a hundred concurrent participants [133].

WebRTC [205] has been used in the design of other kinds of overlay networks, including content delivery [172], real-time collaboration [197], and virtual reality [82]. Kuhara and al. [102] have proposed a service to share files for volunteer computing but they tested their
system on a single machine. BrowserCloud.js [51], is the only other distributed computing platform we are aware of that also uses WebRTC as an overlay. Contrary to our design, it is organized around a distributed hash table rather than a tree, and tasks are pushed from the submitting peer to available workers rather than being pulled by workers as they become free. The implementation of browserCloud.js has been tested on 10-25 browsers on a single machine, which provides little information about the speed at which their overlay can scale in deployments on a local network. Spray [146] is a peer sampling implementation that also uses WebRTC and they also tested on the Grid5000 testbed, with up to 600 hundred concurrent browsers. However, their experiments limit the rate at which participants join to 1 per 5 seconds. It therefore takes 50 minutes for the 600 browsers to join. In a similar setup, our fat-tree overlay deploys on a thousand browsers in 20-55 seconds.

7.6 Summary

WebRTC enables browsers to exchange data directly but the number of possible concurrent connections to a single source is limited. We overcame the limitation by organizing participants in a fat-tree overlay: when the maximum number of connections of a tree node is reached, the new participants connect to the node’s children. Our design quickly scales when a large number of participants join in a short amount of time, by relying on a novel scheme that only requires local information to route connection messages: the destination is derived from the hash value of the combined identifiers of the message’s source and of the node that is holding the message. The scheme provides deterministic routing of a sequence of connection messages from a single source and probabilistic balancing of newer connections among the leaves.

To show the probabilistic balancing scheme is useful, we measured the depth of nodes and found that at least 83% of nodes have the same depth as they would have in a deterministic scheme; this percentage grows to 92.5% as the tree grows larger and is independent of the failure level of nodes if they reconnect through the root after a disconnection. To show the design can quickly scale, we measured the time required for all participants to become connected within a fat-tree overlay fully implemented and tested in Pando. We succeeded in connecting a thousand browser windows in 22-55 seconds on a local network. These results show that the design is quite useful for quick deployments on local networks, such as those in a university department or a large organization. Additional preliminary measurements of connectivity probability and latency for WebRTC on Internet deployments show that further refinements of the design in an Internet setting shall include tolerance to failures of initial
connections, perhaps by initiating multiple connections upon joining, and tolerating initial connection latencies of up to 9-16 seconds.

Compared to previous work on fat-trees, we are the first to (1) propose a deterministic routing scheme for connection messages to quickly grow a fat-tree overlay when a large number of participants join in a short amount of time, (2) implement such a design with WebRTC to overcome the limit on the number of connections, and (3) apply the idea to dispatch work and retrieve results in a volunteer computing tool, using participants for data distribution rather than a dedicated server.
In this dissertation, we have first articulated the new personal volunteer computing paradigm, that refines volunteer computing by focusing on the combination of personal projects, personal volunteer network, personal devices, and personal tools.

We then presented Pando, a new and first tool for personal volunteer computing, that distributes the application of a function on a stream of inputs into the browsers of participating devices. Pando dynamically scales to new devices, gracefully tolerates sudden disconnections, and is easy to program because it is based on a declarative concurrent programming paradigm, in which the non-determinism of the execution is not observable by users.

We followed with a more detailed presentation of the implementation of Pando, based on the new Limiter, StreamLender, and DistributedMap abstractions, themselves organized around the pull-stream design pattern. Our presentation used a high-level notation, independent of JavaScript and simpler to reason with, to present all the algorithms that implement the Limiter, StreamLender, and DistributedMap abstractions. Because the concurrent aspects make StreamLender challenging to implement, even with a clear description, we have presented a run-time verification approach to ensure it correctly follows the invariants of the pull-stream callback protocol. The approach is easy to parallelize, with Pando for example, and quickly generates a large number of random executions to ensure a high-probability of correctness. The combination of clear descriptions and testing strategy should make Pando easy to reimplement in other programming environments.

We then presented a large scope of applications that we implemented for Pando, based on existing libraries and examples, including Collatz, Raytrace, Arxiv, StreamLender Random Testing, Machine Learning Agent, Crypto-Mining, and three different versions of Image Processing using the HTTP, the WebTorrent, and the DAT protocols for data distribution. These applications represent various dataflow patterns, including pipeline processing, synchronous...
parallel search, and stubborn processing, and showed Pando could be used not only for compute-intensive tasks but also for crowd-processing.

We then measured the throughput performance of these applications in three networking scenarios: (1) over a local-area Wi-Fi network, with personal laptop and smartphones, (2) over a virtual private network distributed throughout France, with the Grid5000 nodes, and (3) with a wide-area network on the Internet, with the PlanetLab EU nodes. We showed personal devices to be competitive in all scenarios, sometimes with older devices competing with newer models, and other times with combinations of personal devices outcompeting remote server nodes. The flexible and easy support of all these scenarios is, to the best of our knowledge, a first in the volunteer computing literature.

We finally presented Genet, a new fat-tree overlay for WebRTC that enables Pando to overcome the limits of WebRTC in the number of connections, and showed the resulting combination of Pando and Genet to be able to scale to a thousand browsers in 30-55 seconds on local networks. Those results were possible because the design of Genet only uses local information to deterministically route the WebRTC connection messages, while ensuring the resulting tree is probabilistically balanced.

These results provide a nice foundation for further performance tuning, the support of additional programming models, and the extension of fault-tolerance in more adversarial settings. Beyond those incremental advances, we now briefly sketch more radical future research directions that takes into account the limits to growth our society is currently facing.

### 8.1 Future Research Directions within the Limits to Growth

*The Limits to Growth* [132] is a study, published in 1972, commissioned by the Club of Rome, and funded by the Volkswagen Foundation, in which a leading team of researchers at MIT used the nascent capabilities of computer simulations to model various scenarios that could help predict the future evolution of our industrial civilization. They captured in their model the interactions between "population growth, accelerating industrialization, widespread malnutrition, depletion of non-renewable resources, and environmental degradation". They then generated multiple scenarios, depending on how we would deal with limits, to anticipate their global effects. While heavily criticized at the time, their business-as-usual scenario has closely followed the evolution of our society in the last half-century and their analysis has therefore recently regained widespread support. The most unsettling aspect of that scenario however, is the prediction of "a sudden and uncontrollable decline in both population and industrial capacity" within our lifetimes, possibly as close as within one or two decades from today. In other words, the seemingly endless improvements in our quality of life of the last
century is headed towards a wall, which we shall reach soon, because the physical, social, and environmental limits cannot sustain the infinite growth on which they were predicated.

The implications on the computing industry, since it has been a major driver of industrial growth since the publication of the report, could have been foreseen. Consequently, the priorities of research in computer science and engineering should at least have included contingency plans. However, we believe that computer scientists collectively missed the urgency of the situation. Maybe we are so used to work in virtual environments that we are shielded from many aspects of the real world. Maybe also that our formal and informal training has focused on a combination of physics, engineering, economics, and mathematics, and these for half a century have been the most relevant disciplines to predict the future of computing. These disciplines however typically do not address social and environmental limitations.

We are currently facing limits, and while the computing research field has already started dealing with energy efficiency because it started having a significant economic cost in industry, we believe that the limits that become increasingly relevant today are social and environmental. For example, the surveillance infrastructure required to power advertisement-based online services as well as the environmental pollution caused by electronic waste are now regularly making the news. And the constantly increasing prices of the newer and faster devices, faster than inflation, is putting them out of reach for an increasing number of people.

The predicament that we are facing now cannot be solved by finding new strategies to maintain growth, as these at best only delay an eventual collapse. We will actually have to drastically reduce the energy and resource usage that our computing infrastructure requires if we want to maintain their widespread benefits in the face of impeding resource contractions. So we propose to design new generations of systems to be as affordable as possible to keep computing technologies relevant for sectors with less, no, or even negative economic growth. The key advantage of personal volunteer computing to that end is that it can provide computing services by leveraging existing personal devices that have already been paid for, can be operated at low cost, and can easily be switched to use renewable and decentralized sources of electricity.

In the next sections, we first review concrete potential future applications from existing citizen science initiatives. We then articulate a set of design principles that would be consistent with providing computing services within the social and environmental limits we are facing. We then sketch what potential research topics may be pursued within different sub-fields of computer science.
8.1.1 Potential Applications

The unifying theme behind all the applications we envision is that they increase capabilities at the community level by using that same community’s resources rather than distant computing infrastructure and associated supporting resources. We start from applications that may directly contribute to ongoing initiatives and then generalize abstract dimensions that may guide future investigations and associated technical solutions.

One direction with potential short-term gains is to support existing citizen science initiatives. Some Public Lab [163] projects rely on near-infrared imagery to determine plant health [85, 144] and rely on software processing pipelines [84]. Pictures are often processed at home, due to the amount of processing required. Using Pando would enable that processing to happen in the field with the volunteers’ cellphones collectively. Another example, Zooniverse [218] leverages the abilities of volunteers to perform classification, pattern-matching, annotation, and transcription tasks that may provide useful data for researcher as is, or after training with machine learning algorithms. Pando could make the effort more social by coordinating the efforts of volunteers working in the same room on their cellphones to perform the different tasks, similar to our Arxiv example.

The applications we used in Chapter 5, except for Arxiv, all used the devices for automatic processing, synchronously to complete tasks faster, and in the vicinity of its user. Each choice is one possibility along a different dimension. We briefly sketch other possibilities along the same dimensions.

Nature of Computing. Automatic computations are performed strictly on machines. Human computations require the input from a human to make a decision, identify features, classify elements, or process information, such as those performed by Zooniverse volunteers. In between, hybrid computations may blend both for a better result. An exciting prospect for personal volunteer computing is to use data generated by a community and train machine learning algorithms to act as a collective memory specific to that community. For example, a foraging application, that could help identify and track useful plants and mushrooms and optimized for a local region, could be quite valuable to spread the skills quickly while preventing the appropriation of the knowledge by external parties.

Locality in time. Synchronous processing has all computing resources working at the same time to complete a task as quickly as possible. Asynchronous processing decouples the work performed by computing resources, which may instead contribute when they are most available. This would enable, for example, processing infrastructure powered by renewable energy in which the nodes work when there is sufficient energy available but otherwise power down. This could work well in combination with an asynchronous messaging infrastructure such as Secure Scuttlebutt [178].
Locality in space. Colocated computations happen with all computing resources in a close physical space. Widely distributed computations happen with computing resources spread over a larger area and potentially connected by routing infrastructure, such as the Internet. The latter may happen for online communities of interests. We can imagine, for example, a community-specific archival service, similar to Archive.org, that would only keep track of resources that were linked by community discussions related to specific interests. It would use the computing power and storage of participants to archive the content at links, build indexes of resources, and spread the load among members sharing the same interests.

All the applications proposed previously address a community’s needs using the computing resources it already owns. We believe there are many more interesting applications that can be built along the same lines and these have the potential to be several orders of magnitude more affordable in infrastructure investment and energy usage than current alternatives.

8.1.2 Design Principles

We propose the following design principles for a new generation of personal volunteer computing systems that work with existing devices and are cheap to operate.

Local first. Instantly available high-bandwidth communication and scalable computing power and storage planet-wide require significant infrastructure investment and use enough energy to power small countries. In contrast, systems keeping information and computation local for services that may be offered without global connectivity will make the services more resilient to potential outages and energy shortages, and will bring down energy usage in intermediate routing nodes and data centres by several orders of magnitude.

Leverage humans and their communities first, then augment their capabilities. Volunteers can achieve gigabytes/s by carrying portable hard-drives; they can quickly provide access to dozens of CPUs in a few minutes by lending their devices’ capabilities; they can help organize information and maintain devices in working order; and they can build supporting infrastructure for providing energy and communication. Moreover, effective human communities have a high inherent level of trust and solidarity that can help simplify the designs of systems. The total amount of voluntary work volunteers can bring to keep their community functioning is a large resource to leverage. Volunteers may still benefit from automation to decrease the most tedious aspects of their work, in this case the infrastructure augments their capabilities rather than replacing them.

Simplify and decentralize hardware-software infrastructure. By leveraging humans for many tasks, such as explicitly making their devices available when requested by a friend and therefore doing away with discovery mechanisms relying on persistent connectivity,
we can simplify the design of systems to the point that they can be maintained by one or a few developers part-time. In turn, this increases their accessibility to sectors with restricted available resources. Moreover, by decentralizing the coordination and building infrastructure at the community level, the services provided will be more resilient to global outages.

**Leverage the full usable lifetime of existing personal devices.** The usable lifetime of our personal devices, potentially up to several decades, is significantly longer than their average replacement rate and constitutes another major resource to be exploited for next generation systems. This can provide infrastructure to resource-constrained communities and keep the devices from landfills for longer.

**Optimize for energy use and local energy sources.** Lowering the energy usage drastically reduces the operating costs. There are many potential optimization opportunities for applications that are not latency or throughput critical. This may allow, for example, to create systems whose computation activity follow the natural cycles of energy availability from solar panels or wind turbines. This would in turn lower the need for expensive energy storage, or backup infrastructure powered by non-renewable energy, therefore further lowering the infrastructure and maintenance costs as well as their environmental footprint.

Taken together, these principles really represent an exciting new paradigm of research that considers both unexploited opportunities of today and the hard constraints on energy and resources that may appear in a near future for many sectors of our society.

### 8.1.3 Research Topics

In this section, we briefly sketch some of the promising avenues to be explored in existing disciplines of computing and other related fields when designing new systems built along the previous principles.

**Programming Languages**

To enable a single, or a few developers, to maintain and extend the required software infrastructure favours a renewal of the tradition of self-hosted programming systems, such as Oberon [212], Smalltalk [69], and Forth [33]. Newer efforts could aim at creating a full environment for community applications based around personal volunteer computing.

**New Algorithms**

We envision the following three new kinds of algorithms to be implemented for personal volunteer computing systems.
Social algorithms would be partly or completely be performed by cooperative members of communities, while being augmented by capabilities of interactivity, long-term faithful storage, and automatic cryptographic validation brought by the computers.

Subjective algorithms would index, process, and organize the data produced by local communities according to the interest of each user, from their point-of-view, without access to global information. This could provide similar services as the current recommendation and personalization services of today without requiring a third-party to access private data.

Natural algorithms would be long running and energy-aware so they behave in accordance to the natural cycles in which the computing systems are embedded.

System Design

For many older devices, the maintenance and support from the original manufacturer is discontinued earlier than the end of their useful lifetime. For example, Apple stops supporting updates for older devices six years after they have been released. As such there is a need for minimal operating systems that can be developed and maintained by a community inheriting legacy devices long after their manufacturer stopped their support. The resulting community support may need to extend to multiple decades. The designs should also take energy availability into consideration for task scheduling, persistence, and performance management.

Communication and Sensor Networks

Old mobile phones may be used to design ad hoc mesh communication networks. The resulting system should be self-configuring with minimal expert knowledge from volunteers deploying them. New routing algorithms should also take into account energy availability and efficiency.

Energy Engineering

Design of small-scale energy storage and production in the 5-10W range that can be built with local, abundant, and inexpensive materials by volunteers. These could be based on various technologies, including sterling engines using water or oil for thermal storage, thermo-electric effects by combining different metals and heating them with the sun, small wind or water turbine built with salvaged electric motors, etc. These will help bring the operation costs close to zero.

The previous research directions, compared to the current trends in research, make smaller whole-system designs done by small teams viable again. There may therefore be valuable
insights to dig back from the 80s and 90s literature and to refresh with the benefit of insights from the two to three decades that followed, including the growing importance of energy management.

As a closing note, the contributions of this dissertation on personal volunteer computing have been a first necessary but small step in research directions with a wide variety of interesting and critical research challenges. We hope members of the computing research community will welcome those directions and join us in our investigations.
References


References


References


Appendix A

Test Modules

Algorithm 22 TestSrc Actions

1: sourced ← 0 ▷ Number of values produced by TestSrc
2: function HASNEXTINPUT
3: \hspace{1em} return \(\exists TestSrc:\text{ask}(sourced + 1) \lor \exists TestSrc:\text{abort}(i) \lor \exists TestSrc:\text{fail}(i)\)
4: function NEXTINPUT
5: \hspace{1em} if \(\exists TestSrc:\text{abort}(i) \lor \exists TestSrc:\text{fail}(i)\) then
6: \hspace{2em} if \(\exists TestSrc:\text{ask}(i - 1) \land \nexists TestSrc:\text{value}(i - 1, v) \land \nexists TestSrc:\text{done}(i - 1)\)
7: \hspace{3em} and \(\nexists TestSrc:\text{error}(i - 1, err)\) then ▷ Non-answered previous ask?
8: \hspace{2em} trigger TestSrc:\text{done}(i-1) ▷ Close the stream
9: \hspace{1em} else if sourced < count then ▷ Answer with the next value
10: \hspace{2em} sourced ← sourced + 1
11: \hspace{1em} trigger TestSrc:\text{value}(sourced, sourced)
12: \hspace{1em} else if normalSequence then ▷ Terminate the stream
13: \hspace{2em} if normalTermination then
14: \hspace{3em} trigger TestSrc:\text{done}(sourced+1)
15: \hspace{2em} else
16: \hspace{3em} trigger TestSrc:\text{error}(sourced+1, err)
17: \hspace{1em} else ▷ Generate as many values as requested (infinite stream)
18: \hspace{2em} sourced ← sourced + 1
19: \hspace{1em} trigger TestSrc:\text{value}(sourced, sourced)
Algorithm 23 TestSink Actions
1:  \( \text{sinked} \leftarrow 0 \)  \( \triangleright \) Number of values received by TestSink
2:  \( \text{TestSink}.\text{out puts} \leftarrow [] \)
3:  \textbf{function} \text{HASNEXTOUTPUT}
4:  \textbf{return} \( \exists i. \text{TestSink}:\text{done}(i) \) \textbf{and} \( \exists i,\text{err} \text{TestSink}:\text{error}(i,\text{err}) \)
5:  \textbf{function} \text{NEXTOUTPUT}
6:  \textbf{if} \text{normalSequence} \textbf{then}  \( \triangleright \) Request as many values as available
7:    \( \text{sinked} \leftarrow \text{sinked} + 1 \)
8:    \textbf{trigger} \text{TestSink}:\text{ask}(\text{sinked})
9:  \textbf{else if } \text{sinked} < \text{count} \textbf{then}  \( \triangleright \) Request values up to count
10:   \( \text{sinked} \leftarrow \text{sinked} + 1 \)
11:  \textbf{trigger} \text{TestSink}:\text{ask}(\text{sinked})
12:  \textbf{else if } \text{normalTermination} \textbf{then}  \( \triangleright \) Terminate normally
13:  \textbf{trigger} \text{TestSink}:\text{abort}(\text{sinked} + 1)
14:  \textbf{else}  \( \triangleright \) Terminate abnormally
15:  \textbf{trigger} \text{TestSink}:\text{fail}(\text{sinked} + 1, \text{err})
16:  \textbf{upon} \text{TestSink}:\text{value}(i,v)
17:  \text{TestSink}.\text{out puts} \leftarrow \text{concatenate}(\text{TestSink}.\text{out puts}, [v])
18:  

Algorithm 24 Test Transformer Input: State Machine Transitions
1:  \textbf{upon} \text{Input}:\text{ask}(m)  \( \triangleright \) Triggered by Test
2:    \textbf{if} \text{Input.state} = "ready" \textbf{then}
3:      \text{Input.state} \leftarrow "waiting"
4:    
5:  \textbf{upon} \text{Input}:\text{abort}(m) \textbf{or} \text{Input}:\text{fail}(m,\text{err})  \( \triangleright \) Triggered by Test
6:    \textbf{if} \text{Input.state} = "ready" \textbf{or} \text{Input.state} = "waiting" \textbf{then}
7:      \text{Input.state} \leftarrow "terminating"
8:  
9:  \textbf{upon} \text{Input}:\text{value}(m, v)  \( \triangleright \) Triggered by StreamLender
10:  \textbf{if} \text{Input.state} = "waiting" \textbf{then}
11:    \text{Input.state} \leftarrow "ready"
12:  
13:  \textbf{upon} \text{Input}:\text{done}(m) \textbf{or} \text{Input}:\text{error}(m,\text{err})  \( \triangleright \) Triggered by StreamLender
14:  \textbf{if} \text{Input.state} = "waiting" \textbf{or} \text{Input.state} = "terminating" \textbf{then}
15:    \text{Input.state} \leftarrow "done"
16:  

Algorithm 25 Test Transformer Output: State Machine Transitions

1: upon \(\text{Output:ask}(n)\) \(\triangleright\) Triggered by StreamLender
2: \hspace{1em} if \(\text{Output.state} = \text{waiting}\) then
3: \hspace{2em} \text{Output.state} \leftarrow \text{ready}
4: 
5: upon \(\text{Output:abort}(n)\) or \(\text{Output:fail}(n, \text{err})\) \(\triangleright\) Triggered by StreamLender
6: \hspace{1em} if \(\text{Output.state} = \text{"waiting"}\) or \(\text{Output.state} = \text{"ready"}\) then
7: \hspace{2em} if \(\exists \text{Output:ask}(n-1)\) and \(\not\exists \text{Output:value}(n-1,v)\) and \(\not\exists \text{Output:done}(n-1)\)
8: \hspace{3em} \text{trigger} \(\text{Output:done}(n-1)\) \(\triangleright\) Terminate pending ask
9: \hspace{1em} \text{Output.state} \leftarrow \text{"terminating"}
10: 
11: upon \(\text{Output:value}(n, v)\) \(\triangleright\) Triggered by Test
12: \hspace{1em} if \(\text{Output.state} = \text{"ready"}\) then
13: \hspace{2em} \text{Output.state} \leftarrow \text{"waiting"}
14: 
15: upon \(\text{Output:done}(n)\) or \(\text{Output:error}(n,\text{err})\) \(\triangleright\) Triggered by Test
16: \hspace{1em} if \(\text{Output.state} = \text{"ready"}\) or \(\text{Output.state} = \text{"terminating"}\) then
17: \hspace{2em} \text{Output.state} \leftarrow \text{"done"}
18:
**Algorithm 26 Inter-leaving**

1: Inputs:
2: \( seed \) \hspace{1cm} \(\triangleright\) Pseudo-random number generator (prng) initial state
3: \( count \) \hspace{1cm} \(\triangleright\) Nb of values in input or output
4: \( parallelism \) \hspace{1cm} \(\triangleright\) Maximum number of sub-streams that can be active at the same time
5: \( maxSS \) \hspace{1cm} \(\triangleright\) Maximum total number of sub-streams that can be created
6: 
7: Random.seed = seed \hspace{1cm} \(\triangleright\) Initialize the prng state, \(0 \leq \text{Random}() < 1\)
8: \( activeSS \leftarrow \emptyset \) \hspace{1cm} \(\triangleright\) References to Transformers in active sub-streams
9: 
10: \( normalSequence \leftarrow \text{Random}() > 0.5 \) \hspace{1cm} \(\triangleright\) Terminates from the source? Or the sink?
11: \( normalTermination \leftarrow \text{Random}() > 0.5 \) \hspace{1cm} \(\triangleright\) Terminates normally? Or with an error?
12: 
13: create TestSource TestSrc, TestSink TestSink, StreamLender SL
14: create ProtocolChecker \( PC1, PC2 \)
15: Pipeline \(\leftarrow (\text{TestSrc} \Rightarrow PC1 \Rightarrow SL \Rightarrow PC2 \Rightarrow \text{TestSink})\)
16: 
17: function PossibleActions
18: \( choices \leftarrow \emptyset \)
19: \hspace{1cm} if HasNextInput() then
20: \hspace{2cm} choices \leftarrow choices \cup \{1\}
21: \hspace{1cm} if HasNextOutput() then
22: \hspace{2cm} choices \leftarrow choices \cup \{2\}
23: \hspace{1cm} if SL can create sub-streams then
24: \hspace{2cm} choices \leftarrow choices \cup \{3\}
25: \hspace{1cm} if activeSS \neq \emptyset then
26: \hspace{2cm} choices \leftarrow choices \cup \{4\}
27: \hspace{1cm} return choices
28: 
29: choices \leftarrow PossibleActions()
30: \hspace{1cm} while choices \neq \emptyset do
31: \hspace{2cm} PickNextAction(choices)
32: \hspace{1cm} choices \leftarrow PossibleActions()
33: 
34: if TestSink.outpus = Range(count) and PC1.errors = \emptyset and PC2.errors = \emptyset then
35: \hspace{1cm} print "test succeeded" \hspace{1cm} \(\triangleright\) Ex: \(\text{Range}(0) = [], \text{Range}(1) = [1]\), etc.
36: \hspace{1cm} else
37: \hspace{2cm} print "test failed, with seed:" + seed
Algorithm 27 PickNextAction

1: function PickNextAction(choices)  ▶ Set of numbers representing possible actions
2:  choice ← Random(choices)   ▶ Pick one, with equal probability for each.
3:  if choice = 1 then
4:     NextInput()
5:  else if choice = 2 then
6:     NextOutput()
7:  else if choice = 3 then
8:     CreateSubStream()
9:  else if choice = 4 then
10:    PickNextSubStreamAction(activeSS)

Algorithm 28 Sub-Streams Creation

1: totalSS ← 0
2: function CanCreateSubStream
3:   return \( i \) TestSink:done(i) and \( i, err \) TestSink:error(i, err) and size(activeSS) < parallelism
4:
5: function CreateSubStream  ▶ Optional, avoids infinite loop in case of bugs
6:   if totalSS > maxSS then
7:      Throw Exception("Max number of sub-streams reached, seed: " + seed)
8: 9:   totalSS ← totalSS + 1   ▶ Create the sub-stream
10:  SkO, SkI ← SL.lendStream()
11:  create Sink Input, Source Output  ▶ Test Transformer
12:  Input.state ← "ready", Output.state ← "waiting"  ▶ State machines init.
13:  Output.values ← []  ▶ Queue of values to output
14:  Input.m ← 0, Output.n ← 0  ▶ Event indexes
15:  activeSS ← activeSS \( \bigcup \) \{(Input, Output)\}
16:  create ProtocolChecker SSPC1, SSPC2
17:  SkO \( \Rightarrow \) SSPC1 \( \Rightarrow \) Input, Output \( \Rightarrow \) SSPC2 \( \Rightarrow \) SkI
18: 19: upon Input:value(m, v)  ▶ Enqueue values for later output
20:    Output.values.enqueue(v)
21:
22: upon Input.state = "done" and Output.state = "done"  ▶ Remove sub-stream
23:    totalSS ← totalSS - 1
24:    activeSS ← activeSS \( \setminus \) \{(Input, Output)\}
25:
Algorithm 29 Sub-Streams Actions

1: function PICKNEXTSUBSTREAMACTION(activeSS)
2:     (Input, Output) ← Random(activeSS)
3:     choicesIn ← InputTT(Input.state, Output.state)  \(\triangleright\) Table 4.3
4:     choicesOut ← OutputTT(Output.state, size(Output.values), Input.state)  \(\triangleright\) Table 4.4
5:     choices ← choicesIn \(\cup\) choicesOut
6:     correct ← Random() < 0.9  \(\triangleright\) 90% bias towards a correct execution
7:     if ("ask" \(\in\) choices or "value" \(\in\) choices) and correct then  \(\triangleright\) Pick next action
8:         action ← Random(\{'ask", "value"\} \(\cap\) choices)
9:     else if "done" \(\in\) choices and correct then
10:         action ← "done"
11:     else
12:         action ← Random(\{"ask", "value"\})
13:     if action \(\in\) \{"value", "abort", "fail"\} then  \(\triangleright\) Increment the event counter
14:         Input.m ← Input.m + 1
15:     else
16:         Output.n ← Output.n + 1
17:     if action = ask then  \(\triangleright\) Trigger the corresponding event
18:         trigger Input:ask\langle Input.m \rangle
19:     else if action = abort then
20:         trigger Input:abort\langle Input.m \rangle
21:     else if action = fail then
22:         trigger Input:fail\langle Input.m, err \rangle
23:     else if action = value then
24:         trigger Output:value\langle Output.n, Output.values.dequeue() \rangle
25:     else if action = done then
26:         trigger Output:done\langle Output.n \rangle
27:     else if action = error then
28:         trigger Output:error\langle Output.n, err \rangle
29:     else  \(\triangleright\) Do nothing
30:         trigger Output:error\langle Output.n, err \rangle
31:     end if
Appendix B

Application Implementations

```javascript
var Big = require('bignumber.js')

function collatz (n) {
    n = new Big(n) // convert to BigNum if not already
    var y = 0
    while (n.gt(1)) { // n > 1
        if (n.mod(2).eq(0)) { // n % 2 = 0
            n = n.div(2) // n = n/2
        } else {
            n = n.times(3).plus(1) // n = 3*n + 1
        }
        y = y + 1
    }
    return y
}

module.exports["/pando/1.0.0"] = function (x, cb) {
    var interval = JSON.parse(x)
    var start = new Big(interval.start)
    var range = new Big(interval.range)
    var limit = start.plus(range)
    var largest = 0, bignums = 0
    for (var i = start; i.lt(limit); i = i.plus(1)) {
        var r = collatz(i)
        if (r > largest) largest = r
        bignums++
    }
    interval.largest = largest
    cb(null, JSON.stringify(interval))
}
```

Fig. B.1 Collatz JavaScript Implementation.
```javascript
var summary = null
var relevantBtn = null
var irrelevantBtn = null

function createVisualization () {
    summary = document.createElement('p')
    relevantBtn = document.createElement('button')
    relevantBtn.innerText = 'Relevant'
    relevantBtn.setAttribute('onclick', 'relevant()')
    irrelevantBtn = document.createElement('button')
    irrelevantBtn.innerText = 'Irrelevant'
    irrelevantBtn.setAttribute('onclick', 'irrelevant()')
    var box = document.getElementById('visualization')
    box.appendChild(relevantBtn)
    box.appendChild(irrelevantBtn)
    box.appendChild(summary)
}

var _cb = null

global.relevant = function () {
    if (_cb && id) {
        var link = id.innerText
        _cb(null, JSON.stringify({
            link: link,
            relevant: true
        }))
    }
}

global.irrelevant = function () {
    if (_cb && id) {
        var link = id.innerText
        _cb(null, JSON.stringify({
            link: link,
            relevant: false
        }))
    }
}

module.exports['/pando/1.0.0'] = function (x, cb) {
    var entry = JSON.parse(x)
    summary.innerText = entry.content
    _cb = cb
}
```

Fig. B.2 Simplified Arxiv JavaScript Implementation.
```javascript
var interleaving = require('pull-lend-stream-random-tester')

module.exports['/pando/1.0.0'] = function (x, cb) {
  var options = JSON.parse(x)
  for (var i = 0; i < options.executions; ++i) {
    options.seed = Math.round(Math.random() * Math.pow(2, 31))
    try {
      interleaving(options)
    } catch (e) {
      return cb(null, JSON.stringify({
        options: options,
        success: false
      }));
    }
  }
  delete options.seed
  cb(null, JSON.stringify({
    options: options,
    success: true
  }));
}
```

Fig. B.3 StreamLender Random Testing Main Processing Function.
```javascript
var startTime = new Date()
var hyperparams = null
var current_interval_id = null
var simulation = null

// User can manually reject an execution that is performing poorly
var rejectBtn = document.createElement('button');
rejectBtn.setAttribute('onclick', 'abort(false)');
rejectBtn.innerHTML = 'Reject';

function tick() {
    simulation.tick();
    if (simulation.clock % 50 === 0) draw(); // Draw every 50 steps

    var currentTime = new Date()
    var trainingTime = currentTime - startTime
    if (trainingTime >= hyperparams['training-ms']) {
        abort(true)
    }
}

function run() {
    window.clearInterval(current_interval_id);
    current_interval_id = setInterval(tick, 0);
}

function abort(s) {
    var result = JSON.stringify({
        hyperparams: hyperparams,
        accepted: s === true,
        ticks: w.clock,
        rewards: JSON.stringify(reward_graph)
    })
    window.clearInterval(current_interval_id);
    cb(null, result)
}

module.exports['/pando/1.0.0'] = function (x, cb) {
    createVisualization() // Provided by ML Library
    hyperparams = JSON.parse(x)
    _cb = cb
    simulation = new World(700, 500); // Provided by ML Library
    simulation.agents = [new Agent(hyperparams)]; // Provided by ML Library
    startTime = new Date()
    run(); // Provided by ML Library
}
```

Fig. B.4 Simplified Implementation of the Machine Learning Agent Processing Function.
```javascript
var luminance = require('luminance')
var getPixels = require('get-pixels')
var savePixels = require('save-pixels')
var blur = require('ndarray-gaussian-filter')

var url = info.baseUrl + '/' + info.entity + '/' + info.preview
beforeImg.src = url
beforeImg.onload = function () {
    // Process
    getPixels(url, 'image/jpg', function (err, image) {
        if (err) return cb(err)

        image = luminance(image)
        blur(image, 5)

        // Save the result
        savePixels(image, 'canvas').toBlob(function (resultBlob) {
            // Transfer the data
            var url = info.result.baseUrl
            var filepath = info.result.entity + '/' + info.result.preview

            var form = new FormData()
            form.append('file', resultBlob, filepath)

            var xhr = new XMLHttpRequest()
            xhr.open('POST', url, true)

            xhr.onload = function () {
                if (xhr.status === 200) cb(null, JSON.stringify(info))
                else throw new Error('transfer error')
            }

            xhr.send(form)
        }, 'image/jpg')
    })
}
```

Fig. B.5 Simplified Implementation of Image Processing with HTTP Data Distribution.
```
var shajs = require('sha.js');
var log = require('debug')('miner');
var pando = require('pando-computing');

// Bitcoin minimum target: '00000000

// Bitcoin maximum target: '

module.exports['/pando/1.0.0'] = function (x, cb) {
  /*
   * {
   *   block: String,
   *   target: String (hex),
   *   range: Number (Number of nonces),
   *   seed: Number (starting nonce)
   * }
   */

  var job = JSON.parse(x);
  var block = new Buffer(job.block);
  var packedTarget = new Buffer(job.target, 'hex');
  // packedTarget format: 0x00000000
  // target: 0x00000000 * (2**8*(0x00 - 3))
  var target = new Buffer(32).fill(0);
  var data = packedTarget.slice(1)
  target.fill(data, offset, offset + 3)

  var result = {
    block: job.block,
    target: job.target,
    success: false, // Boolean
    attempts: 0, // Integer
    nonce: null // null or String
  }

  var nonce = job.seed
  while (nonce < job.seed + job.range) {
    var hash = shajs('sha256').update(nonce.toString()).update(block)
      .digest()
    result.attempts++
    if (target.compare(hash) >= 0) {
      result.success = true
      result.nonce = nonce
      return cb(null, JSON.stringify(result))
    }
    nonce++
  }

  // None found
  return cb(null, JSON.stringify(result))
};
```

Fig. B.6 Implementation of Crypto-Currency Mining.