A Installing Pando from Sources

Pando can be installed from source in Linux or MacOS. This option has the least memory footprint and is the main intended use case. In most cases, this should work without extra effort. You will first need a working installation of Node.js and the Node Package Manager (NPM).

A.1 Installing Node.js and NPM

We suggest to first install the Node Version Manager (NVM)[5]:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0
     .34.0/install.sh | bash
```

This enables installing Node.js and the Node Package Manager (NPM) with the following command:

```
nvm install node
```

This method does not rely on the package manager of a Linux distribution and makes it easy to keep up-to-date with the latest developments. You can then install Pando from source with one of the following two options.

A.2 Using the Node Package Manager

Pando can be installed globally with the following command:

```
npm install --global pando-computing
```

This provides a globally accessible pando executable. The various options are accessible with:

```
pando --help
```

A.3 Using the Sources on GitHub

The latest version, and its dependencies, can be installed from the GitHub sources with:

```
git clone git@github.com:elavoie/pando-computing
cd pando-computing
npm install
```

We suggest then linking, which will provide a globally accessible pando executable:

```
cd pando-computing
npm link
```

B Installing the Example Applications

The example applications are available in the `pando-handbook`'s Git repository. You can install them from GitHub:

```
git clone git@github.com:elavoie/pando-handbook
```

Each example is a library following the module format used by Node.js. For example, you can try the raytracing application from Section 2.1 by:

```
cd pando-handbook/examples/raytracer
npm install # install dependencies
./generate-angles | pando raytracer.js --stdin | ./write-
     current-frame
```

Some application names are slightly different from the names used in this paper: *StreamLender-Testing* is `random-testing`, *Raytrace* is `raytracer`, *Image Processing* is `photo-batch-processing`, and *MLAgent-Training* is `rlnetwork`. More detail on each application is given in the corresponding `README.md` files.

[5]"https://github.com/nvm-sh/nvm#installation"

C Using the Pre-configured Docker Image

For convenience, we also provide a Docker image[6] that contains: (1) Pando; (2) the example applications with convenience scripts that log the experiment data, (3) the data from the experiments mentioned in the paper, and (4) an analysis script that computes the average throughput for all devices from the logs. This should make it easy to quickly test our applications with new devices and compare new results against those of the paper.

C.1 Retrieving the Image

The image is available on Docker Hub, you can retrieve it using the docker command-line tool:

```
docker pull elavoie/pando-middleware19
```

C.2 Running the Examples of the Pando-Handbook

You can run the Image to create a Container in the following way. It defaults to a squaring example, that waits for one second before returning the square of the input numbers:

```
docker run -it -p 5000:5000 -p 5001:5001 -p 8080:8080
     elavoie/pando-middleware19
```

You can select one of the seven other example applications among the following choices: `arxiv`, `collatz`, `crypto-mining`, `raytracer`, `random-testing`, `photo-batch-processing`, `rlnetwork`. For example, to run the `raytracer`:

```
docker run -it -p 5000:5000 -p 5001:5001 -p 8080:8080
     elavoie/pando-middleware19 /bin/bash
./raytracer
```

The `-it` option starts an interactive session. The `-p` option maps the ports from inside the container to the ports of `localhost`. Some of the examples rely on those specific ports to work. Port `5000` is used to serve volunteer code, port `5001` is used to serve a monitoring page that provides real-time report on the performance of all devices, and port `8080` is used by the `photo-batch-processing` to serve images with an HTTP server.

Running the `square` example should print:

```
starting square, will print 1**2, 2**2, ...
Serving volunteer code at http://172.17.0.2:5000
Serving monitoring page at http://172.17.0.2:5001
```

The URL provided on startup is *from inside the container* (Pando has no way of knowing the external IP address of a Docker container): to connect from a different machine you will need to use the IP address *of the machine running the container.*

C.3 (Optional) Rebuilding the Image from Sources

For future reference, the image can be rebuilt from the GitHub repository, which contains a `Dockerfile`:

```
cd pando-computing
docker build -t elavoie/pando-middleware19 .
```

Doing this can help quickly testing with a more recent version of Pando and the `pando-handbook`.

[6]https://hub.docker.com/r/elavoie/pando-middleware19

D Deploying Pando on Devices

From the machine running the container, you can use http://localhost:5000. Open the URL in a Web browser, then type a device name, then 'save'. The HTTP server might take a few seconds before being accessible, so be a little patient. The square example should then output "1,4,9,...". You can stop the computation at any time by doing CTRL-C.

From other devices, you will need the IP address of the machine running the container. You can then open http://<ip-address>:5000.

By default, a deployment will use a WebSocket to communicate with Pando's Master process. This is equivalent to using the following url: http://localhost:5000/#protocol=websocket;.

Alternatively, WebRTC can be used by setting `protocol` to `webrtc`: http://localhost:5000/#protocol=webrtc;.

E Monitoring Deployments

You can monitor the deployment by opening the monitoring page at http://localhost:5001. The monitoring page provides real-time updates on the performance of all devices, including a historical graph of the combined performance of all devices. This page is handy to determine whether some devices are too slow, whether their CPU is fully utilized.

F Replicating the Experiments

The scripts in `/usr/src/pando-handbook/middleware-2019/run` (default working directory), all log their performance results in the `results` directory. The convention is `results/<app>/<datetime>`. The scripts include the same parameters that we used for executing our experiments. You can also change the parameters passed to Pando, e.g. `batch-size`, by modifying the scripts.

We suggest letting an experiment run for 5 minutes to average the throughputs, to amortize the variation in processing time of inputs. You can stop an experiment at any time with CTRL-C.

After the experiment is over, you can extract the average throughput from `devices.txt` generated by the runs in the `results` directory. For example, to extract the throughputs from a run of `raytracer` you can do:

```
cat results/raytracer/2019-09-07T21\:58\:00/devices.txt |
    ../analysis/throughputs.js
```

This should output, for example:

```
MacBook Air 2011 1.23 34.21\%
iPhone SE 2.37 65.79\%
Total: 3.61 items/s
```

We gathered the results presented in Table 2 with the same methodology. The data we gathered for this paper is available under `/usr/src/pando-handbook/middleware-2019/experiments`.

G WebRTC Gotchas and Limitations

WebRTC is notoriously finicking. Your firewall settings, the configuration of your local network, the router, and the specific combination of browsers that you use can influence whether a peer-to-peer connection will succeed. So here is a checklist of things to look for:

- make sure your firewall setting will let inbound traffic through;
- if you are using LittleSnitch[7] to monitor and filter outbound traffic, make sure the traffic is let through with an appropriate rule or a manual notification;
- test WebRTC connectivity with the WebRTC Troubleshooter[8].

[7]https://www.obdev.at/products/littlesnitch/index.html

The current version of Pando relies on `electron-webrtc` to establish WebRTC connections between the Master process and the Workers: this library opens an instance of Chromium in the background and communicates with it over an inter-process communication library to leverage its WebRTC implementation. Unfortunately, this library is unmaintained and the Chromium version it uses is getting old. At the time of writing this appendix, and in contrast to the moment when we performed the experiments, Pando can no longer connect locally over WebRTC to a tab on a more recent version of Chrome.

To diagnose connectivity issues, we built a WebRTC connection testing tool[9] that replicates the setup of Pando. You can install it by:

```
git clone https://github.com/elavoie/webrtc-connection-
    testing.git
cd webrtc-connectivity-testing
npm install
npm start
```

You can then connect a participant by running the command-line client:

```
cd webrtc-connectivity-testing
bin/participant http://localhost:8080
```

This client uses the same setup as the Pando Master process. You can then connect a second participant, that simulates a Worker, by opening Chrome at the same address. If a line appears between both participants in the "Current Connectivity" then the connection was successful and Pando can be used with WebRTC. For convenience, the server has already been deployed on Heroku[10].

Should anyone be interested in improving Pando, we would suggest putting the management of WebRTC connections inside the monitoring page, that itself connects to the Master process via a WebSocket. That way the dependency on `electron-webrtc` could be removed and all WebRTC connections would happen between recent browsers.

H Reusing the StreamLender and Limiter Abstractions outside Pando

The *StreamLender* and *Limiter* abstractions are available as pull-stream modules in JavaScript. These are available in independent Git repositories with minimal dependencies[11][12]. They are also available under `/usr/src/pando-computing/node_modules` in the image for Docker: look for `pull-lend-stream`, and `pull-limit`. Both modules should be useful to build new applications based on the design of Pando and solve the most complicated part of the implementation.

I Reimplementing the Abstractions in other Languages

A full description of *StreamLender*, *Limiter*, and *DistributedMap* is available in the corresponding PhD thesis[13]. This should be convenient to help a reimplementation in a different language than JavaScript.

[8]https://test.webrtc.org/
[9]https://github.com/elavoie/webrtc-connection-testing
[10]http://webrtc-connection-testing.herokuapp.com
[11]https://github.com/elavoie/pull-lend-stream
[12]https://github.com/elavoie/pull-limit
[13]Erick Lavoie, 2019, *Personal Volunteer Computing*, McGill University