Tudor Ţiplea

Supporting Browser-based Machine Learning: Distributed Data Processing at the Network's Edge

UNIVERSITY OF CAMBRIDGE

Christ's College

A dissertation submitted to the University of Cambridge
in partial fulfilment of the requirements for
Computer Science Tripos, Part III

University of Cambridge
Department of Computer Science and Technology
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: tpt26@cam.ac.uk

1st June 2018

Declaration

I, Tudor Petru Țiplea of Christ's College, being a candidate for Part III of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date: 1st June 2018

Abstract

Because it can tell so much about nature and people, digital data is collected and analysed in immeasurable quantities. Processing this data often requires collections of resources typically organised in massive data centres, a paradigm known as cloud computing. However, this paradigm has certain limitations. Apart from the often prohibitive costs, cloud computing requires data centralisation, which could slow down real-time applications, or require exorbitant storage. Edge computing—a solution aiming to move computation to the network's edge—is regarded as a promising alternative, especially when tailored for Internet-of-Things deployment.

Aiming for more large-scale adoption, this project provides a proof of concept for edge computing support on an ubiquitous platform—the web-browser. This work is framed within an emerging `OCaml` ecosystem for data processing and machine learning applications. We explored options for `OCaml`-to-`JavaScript` compilation, and extended `Owl`, the main library in the ecosystem, guided by those findings. Next, we researched solutions for efficient data transmissions between browsers, then based on that, implemented a browser-compatible communication system analogous to TCP/IP network sockets. This system was later used to modify `Actor`, `Owl`'s distributed computing engine, making it deployable in the browser.

We demonstrated our work on `Owl` was successful, exemplifying the browser-deployed localised computing capabilities. The performance limitations of this part were analysed, and we suggest directions for optimisations based on empirical results. We also illustrated the accomplishment of browser-based distributed computing, again identifying limitations that must be overcome in the future for a complete solution.

Total word count: 11,847[1]

[1]This word count was computed using TEXcount

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank the following two people for the help they have given me throughout this dissertation:

- **Dr Richard Mortier**, for supervising this project, providing me with regular feedback and advice regarding the direction of this work.
- **Dr Liang Wang**, for co-supervising this project, and always being available and happy to help me with any difficulties I encountered while working on the two main libraries involved in this project.

1 Introduction

1.1 Background & Motivation

Increasingly many aspects of modern society are supported by a large variety of data. From personal communications and web analytics, to medical records and traffic information, almost every piece of information is valuable to someone, be it simply financially, or for research and grander humanitarian purposes. Incentivised by these benefits, significant effort has been invested during recent years in the research and development of techniques and tools capable of efficiently wringing as much value from the available data. The immensity of the data meant that much of this work was focused on distributed approaches for data processing and analysis. One famous example is *MapReduce* [1], a programming model designed for processing large datasets on clusters of computers. Perhaps the most popular implementation of *MapReduce* is the open-source *Apache Hadoop*[1] framework, which can deliver a reliable service on top of clusters of either commodity or higher-end hardware. Other similar projects are *Apache Storm*,[2] for real-time computation of streams of data, *Apache Spark*,[3] or *TensorFlow*,[4] an extremely popular library tailored for machine learning computations.

However, many of these projects rely on the computational power supplied by large clusters of high-end computers, which are rarely found outside the colossal data centres owned by wealthy tech companies. Even if a single computer can execute a data analysis task in a timely manner, most modern applications still require the processing capabilities of expensive machines with numerous powerful CPUs and GPUs. In many cases, these are prohibitively expensive for public research institutions, small companies, or individual programmers wishing to analyse large datasets. Besides steep costs, cloud infrastructures have other inherent issues affect-

[1] http://hadoop.apache.org
[2] http://storm.apache.org
[3] https://spark.apache.org
[4] https://www.tensorflow.org

ing everyone—many tasks that process user data must centralise it first, introducing latency hikes in real-time applications, burdensome storage requirements, and single-point-of-failure concerns.

Consequently, those drawbacks call for a shift towards alternative solutions which decentralise and distribute the computation towards the edge of the network. Known as Edge Computing, this class of solutions is seeing increasing popularity in research and industry, being regarded as a promising alternative to Cloud Computing. Speaking to its potential is the decision taken by Amazon, now reportedly the largest company by Cloud Computing revenue [2], to invest in the technology. Amazon introduced AWS Greengrass[5] as an extension to their cloud services, providing the ability to deploy applications on any Internet-of-Things (IoT) device supporting their SDK, even on local networks disconnected from the Internet. Microsoft, another large Cloud Computing provider, has also entered the Edge Computing market with the Microsoft Azure Stack service, which supports local processing followed by cloud-located aggregation. The growth of IoT will undoubtedly stoke the adoption of Edge Computing, especially because many applications, e.g. self-driving cars, cannot incur the cloud communication latency when taking crucial decisions.

But before the technology fully expands in the IoT domain, we must properly explore its capabilities in the personal device space. This is because collectively, these devices add up to massive amounts of computational capacity, which had been left mostly untapped for decentralised computing until recently. Glimpses of its potential started to shine part of the cryptocurrency phenomenon, which has gained much traction and attention in recent years, although wildly fluctuating between fad and genuine prospect. Bitcoin[6] emerged as the first decentralised digital currency, rewarding people in the network for verifying the blockchain transactions on their personal devices. Ethereum[7] expands on the idea, introducing smart contracts as building blocks for decentralised, distributed applications. Currently however, this type of platforms is still arguably obfuscated to ordinary developers, and faces a bumpy journey to large-scale adoption.

Unlike the projects enumerated so far, this project targets what is perhaps the most common computing platform—the web browser. Regardless of whether people use a laptop, desktop computer, tablet, or smartphone, and despite a large variety of operating systems, they have access to a web browser. Moreover, because most websites will continue to rely on JavaScript support to various degrees, at least

[5] https://aws.amazon.com/greengrass/

[6] https://bitcoin.org

[7] https://www.ethereum.org

for the foreseeable future, popular web browsers provide a JavaScript engine. Our goal is to support seamless web-browser deployment of data processing or ML applications. This draws inspiration from excellent ideas such as *CPDN*[8], a project relying on volunteers joining climate modelling distributed computations, or IBM's *World Community Grid*[9] effort, which aims to create a computational grid for scientific research, again relying on unused computing power donated for commendable purposes. However, these require specialised software running on user devices, which limits the number of people supporting the causes. Furthermore, many research teams are unable to implement such a system, while joining existing efforts is often a bureaucratic burden. Ideally, this project will fit into a larger open-source ecosystem that eliminates these concerns from the researchers' workflow, ensuring the development process is competitive with other data processing and machine learning frameworks, while keeping the distributed deployment as smooth as sharing a web link to anyone willing to volunteer the idle processing power of their personal devices.

An obvious solution is to implement a distributed machine learning and data processing library in `JavaScript`. However, such an idea has a major drawback—using `JavaScript` as the main programming language, which would affect both the library developer and the library user. The language is dynamically typed, and structuring the code into modules is difficult, which means development is error-prone and tedious. It also has an event-driven programming flow, which is unsuitable for machine learning applications. Overall, it is arguably an unappealing language, especially for researchers whose main domain of expertise is not Computer Science.

1.2 Objectives

The purpose of this project is to provide a proof of concept for an alternative solution, which can be attempted in two steps. Although it involves certain risks, this solution would meet most of the desired characteristics enumerated so far.

The first step involves exploring options for compilation to `JavaScript` of `Owl`, a numerical library for scientific computing written mostly in `OCaml`. `Owl` relies on external `C/C++` for its core functionalities, which cannot be compiled to `JavaScript`. Therefore, the main requirement is replacing these core parts with alternative implementations that would be compile-able to `JavaScript`. This carries a major risk, because it is hard to foresee and frame precisely to what extent `Owl` must be modified. However, if successful, this part of the project will allow developers to deploy the

[8] https://www.climateprediction.net
[9] https://www.worldcommunitygrid.org

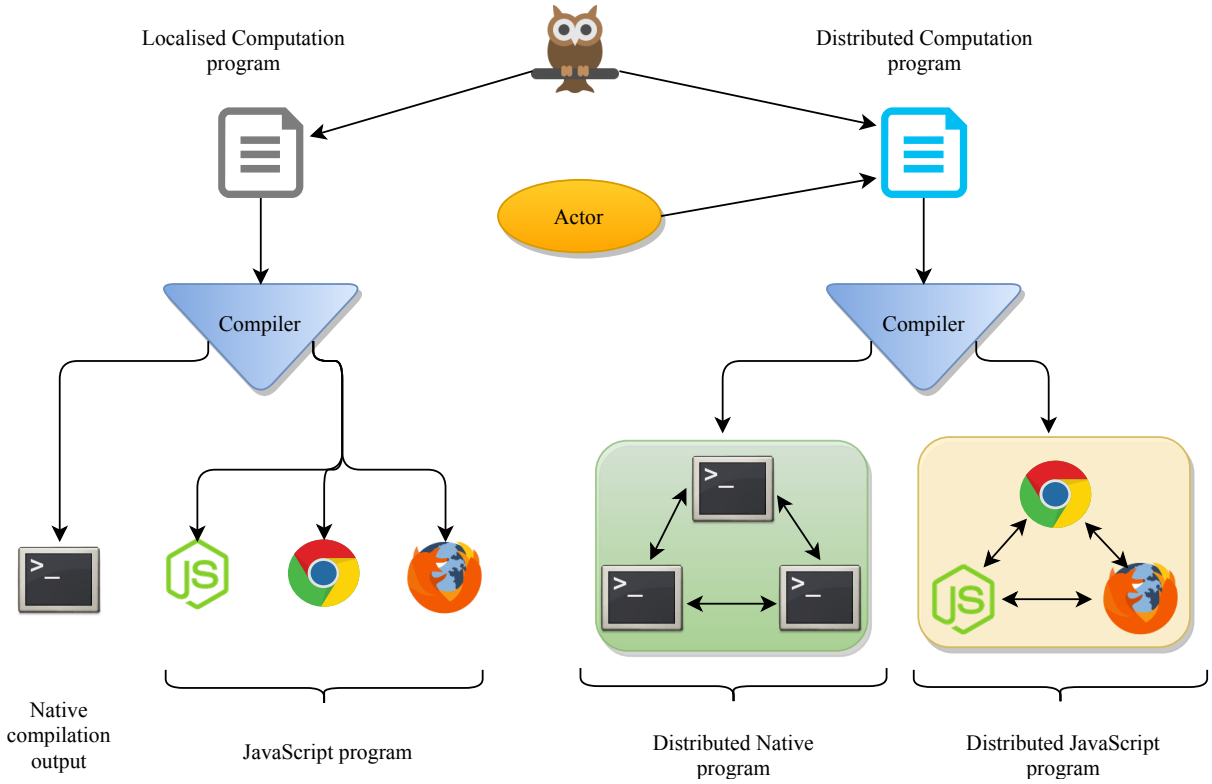


Figure 1.1: Deployment examples for `OCaml` programs using `Owl` and `Actor`.

same `Owl` program on multiple platforms—they can choose the most efficient option and run the it natively, or compile the program to `JavaScript` and deploy it on any `JavaScript` runtime, including web-browsers and `Node.js`. This unlocks various powerful development-deployment combinations—programmers could, for instance, efficiently train their machine learning models locally, using the native code, but perform inference on the end-users' web-browser, using the program's `JavaScript` version. This objective is discussed in depth in Chapter 2.

Following completion of the first part, `Owl` programs could run in the browser as long as they are localised to a single computation instance—they cannot be distributed and communicate with other programs. The second part of the project focuses on overcoming this limitation, aiming to support distributed browser computations, using `Owl` and `Actor`. `Actor` is another `OCaml` library, which serves as `Owl`'s distributed computing engine. Similarly to before, parts of `Actor` would be re-implemented such that they support compilation to `JavaScript`. However, this time, the task is hindered by the communication layer—any `OCaml` implementation would ultimately rely on standard network sockets, which do not have a direct browser analogue they can be compiled to. Consequently, in this portion of the project we research options for efficient communication between browsers, and how these could be combined with `Owl` and `Actor` for a complete solution to the issues enumerated so far. We present this half of the project in Chapter 3.

Facing multiple risks and uncertainties, this work scopes feasible implementations of the finished product, which would fit into a comprehensive ecosystem, together with `Actor` and `Owl`. Using this ecosystem, developers are advantaged by programming in `OCaml`, and can produce highly efficient native code. Furthermore, they can obtain `JavaScript` code from the same program, which can be deployed at the edge of the network. They can choose to smoothly distribute the computation, and with the novel composable `Zoo` system [4], they can piece together existing snippets of code to produce a complete data processing or machine learning pipeline, with minimal effort. In Figure 1.1 we present a few intended deployment options for applications developed inside this ecosystem.

1.3 Related Work

To the best of our knowledge, there is currently no framework that provides the complete set of features intended for the final ecosystem. Nevertheless, parts of our work are similar to existing systems and libraries. `Facebook Reason`[10] is a new syntax and toolchain inspired by `JavaScript`, but having `OCaml` as the backing language, which gives it a complete type system. `Reason` shares the abstract syntax tree with `OCaml`, which means it can be compiled to native code using the `OCaml` compiler. Also, similarly to our goals, `Reason` programs can be compiled to `JavaScript`, and has full inter-operability with the language, thanks to the `BuckleScript`[11] compiler. In fact, we carefully considered using `BuckleScript` in our project too, as the compiler from `OCaml` to `JavaScript` (see §2.2). However, `Reason` is not a complete replacement for our project—it is only an alternative to programming in `OCaml`. Developers wishing to create data processing applications in `Reason` would still have to find a scientific computing library providing the comprehensive set of features `Owl` does. The library would have to be implemented in `JavaScript`, in which case the ability to produce native code is lost, or should be compile-able to `JavaScript` using `BuckleScript`, alternative which we are not aware to exist. Similar impediments would hinder distributed deployments.

In terms of targeting `JavaScript` as a back-end for ML applications, *TensorFlow.js* is perhaps the framework most similar to our work. The developer can deploy and train ML models in the browser, with the help of *WebGL* accelerated kernels, which would not be available in the early versions of our project, but could be something supported in the future. However, *TensorFlow.js* has several disadvantages—`JavaScript` is

[10] https://reasonml.github.io
[11] https://bucklescript.github.io

also the development language, and although it can load pre-existing models defined using the `Python` API, developers are deprived of the benefits of a strongly-typed programming language. More importantly, the computation cannot be distributed between multiple browsers, and developers are limited to the `Python` API for native targets.

2

Owl Base

This chapter details the work invested into achieving our first goal—making it possible to compile programs using the `Owl` library to `JavaScript`, and to deploy them in web browsers. The name is given by the sub-library `Owl-Base`, which we isolated inside `Owl` for smooth compilation to `JavaScript`. We start by summarising the initial structure of the library, along with its most important features, and at the same time we identify which parts prevent immediate compilation to `JavaScript`. Then, we motivate our decision for which `OCaml-to-JavaScript` compiler to use, shortly exploring the alternatives. Finally, we elaborate on how `Owl` was modified to accomplish our objective, and explain our motivation for creating a sub-library inside `Owl`.

2.1 Initial design of `Owl`

`Owl` is an emerging `OCaml` library for scientific computing and machine learning, having a comprehensive set of features which makes it competitive with the more well-established libraries like *NumPy*[1] / *SciPy*[2] and *TensorFlow*. Many of those features can in fact be accessed in ways very similar to what programmers might expect after using the traditional libraries, which ensures the transition to `Owl` is quick and smooth. Table 2.1 illustrates this with some resemblances between the `NumPy` and the `Owl` API for N-d array and matrix operations.

At the core of `Owl` sits a solid implementation of N-dimensional (multi-dimensional) arrays, on top of which most of the other modules are built. These include a large swath of linear algebra operations, along with matrix-specific functions such as inverses and determinants—all accompanied by features such

[1] `http://www.numpy.org`
[2] `https://www.scipy.org`

as broadcasting, efficient implementations of sparse arrays, and powerful slicing operations. These are features which developers come to expect from competent scientific computing libraries. Furthermore, `Owl` also comes with extensive plotting functionalities.

Operation	Code	Library
Creating N-d arrays	`a = zeros(shape=(2,3), dtype=float)`	`NumPy`
	`let a = zeros [|2; 3|]`	`Owl`
Reshaping N-d arrays	`b = reshape(a, newshape=(4, 5))`	`NumPy`
	`let b = reshape a [|4; 5|]`	`Owl`
Indexing N-d arrays	`a[1][2]`	`NumPy`
	`a.%{[|1; 2|]}`	`Owl`
Slicing N-d arrays	`a[0:4:2, 6:-1:3]`	`NumPy`
	`a.${[[0;4;2]; [6;-1;3]]}`	`Owl`
Linear algebra on matrices	`a + b`	`NumPy`
	`a + b`	`Owl`
Matrix inverse	`linalg.inv(a)`	`NumPy`
	`Linagl.inv a`	`Owl`

Table 2.1: Resemblances between the `Owl` and the `NumPy` APIs illustrated on a few operations.

Owl	TensorFlow
`input in_shape`	`a = placeholder(float32, in_shape)`
`|> lambda (fun x -> x / F 256.)`	`b = divide(a, 256)`
`|> conv2d [|5;5;1;32|] [|1;1|]`	`c = nn.conv2d(b, (5,5,1,32), (1,1))`
`~act_typ:Relu`	`d = nn.relu(c)`
`|> max_pool2d [|2;2|] [|2;2|]`	`e = layers.max_pooling2d(d, 2, 2)`
`|> dropout 0.1`	`f = layers.dropout(e, 0.1)`
`|> fully_connected 1024 ~act_typ:Relu`	`g = layers.dense(f, 1024, 'relu')`
`|> linear 10 ~act_typ:Softmax`	`h = layers.dense(g, 10, 'softmax')`
`|> get_network`	

Table 2.2: Comparison between the `Owl` and the `TensorFlow` definitions of a Convolutional Neural Network for classification on the MNIST dataset. Note that this omits the overhead of initialising the graph, session, and network weights, which is typically required in `TensorFlow`. Furthermore, `Owl` uses a dynamic graph to represent the structure of the neural network, while `TensorFlow` imposes a static graph restriction. Therefore, developers benefit from greater flexibility when constructing neural networks in `Owl`.

Algorithmic differentiation is another key component in `Owl`, providing the ability to precisely calculate derivatives instead of relying on approximations or purely symbolic methods. Built on top of this, `Owl` offers functionalities besides the conventional numerical methods enumerated so far. The *Regression* module is such an example, supporting, among others, *linear*, *exponential*, *ridge*, and *lasso* regression, most of them based on a stochastic gradient descent algorithm. Another example, perhaps the most exciting `Owl` feature, is the package of functions for machine learning and neural networks which comprises of a broad set of optimisers and network layers, including convolutional and recurrent structures. Even more intricate neural networks can be defined in a clear manner, in few lines of code. This is exemplified in Table 2.2, which presents side-by-side the code defining a Convolutional Neural Network for MNIST[3] classification in `Owl` and `TensorFlow`.

[3] http://yann.lecun.com/exdb/mnist/

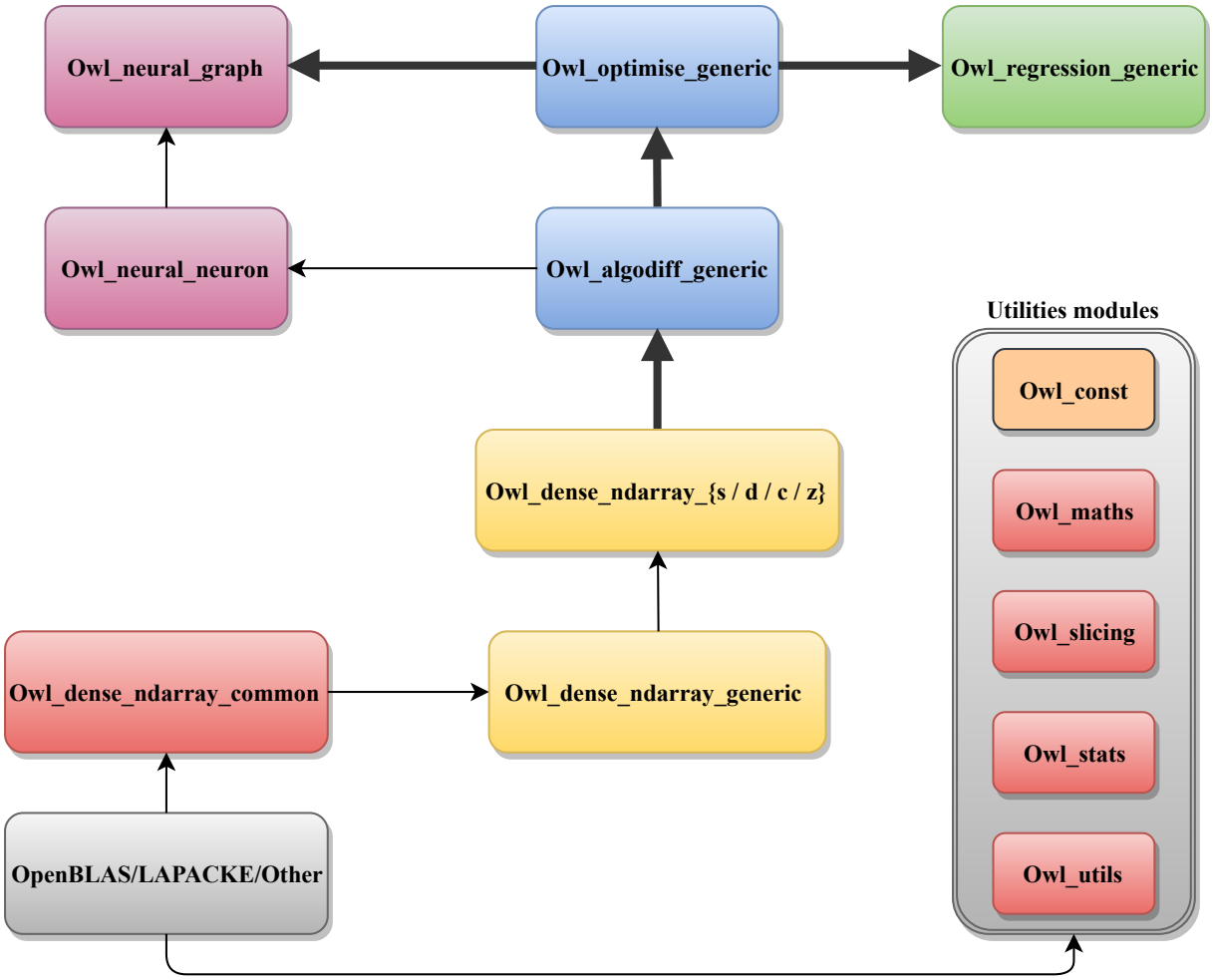


Figure 2.1: Simplified overview of `Owl`'s original structure. The thick arrows represent the main functor tree in the library, building up its features from the core *N*-d array implementations. Grey represents the external library dependencies which must be isolated, including *OpenBLAS* and *LAPACKE*. Red represents modules directly relying on those libraries, while the *Utilities modules* provide functions used pretty much everywhere in `Owl`.

In order to achieve the best performance, `Owl` relies on `OpenBLAS`[4] and `LAPACKE`[5] for efficient implementations of the crucial operations found at the library's core—for instance linear algebra functions, or *N*-d array operations such as indexing, or slicing. Unsuitably for our objectives, these libraries are already natively compiled, exposed to `Owl` through a `C` interface, which makes it impossible to compile them to `JavaScript`. Consequently, we must isolate these two dependencies and implement in pure `OCaml` the functions they provide. Of course, naively replacing them would lead to unsatisfactory performance whenever the developer targets native deployment—no `OCaml` implementation would be even remotely as efficient as the carefully-tuned

[4] https://github.com/xianyi/OpenBLAS
[5] http://www.netlib.org/lapack/lapacke.html

libraries they replace, especially one written in only the short amount of time that can be allocated for this secondary task. To preserve the superior performance of native code, we have to implement the pure OCaml substitutes as alternatives the developer can effortlessly switch between, depending on the desired output code.

A careful analysis of the library's design is required before restructuring can be planned. We proceed to describe several modules[6] which are crucial in the implementation of Owl. The higher-level features of Owl are built on top of core operations using a functor[7] tree. Figure 2.1 is a simplified view of how the modules interact, including an illustration of the functor tree.

***OpenBLAS*, *LAPACKE*, and other C dependencies**

These are the OCaml modules that provide an interface to the external C libraries, and thus prevent compilation to JavaScript. Anything relying on these will require restructuring.

Owl_const, Owl_maths, Owl_slicing, Owl_stats, and Owl_utils

Five modules implementing various utilities that are used almost everywhere in Owl—they collect mathematical constants (Owl_const), provide mathematical (Owl_maths) or statistical (Owl_stats) functions, implement slicing operations on N-d arrays (Owl_slicing), and other various utilities (Owl_utils) such as file manipulation or array iteration. Only one of the four, Owl_const, is implemented in pure OCaml—the rest rely on C code to various degrees, and must be seconded by pure OCaml alternatives.

Owl_dense_ndarray_common

This module provides type-safe interfaces to N-d array manipulation functions implemented in C. For example, it selects functions for equality testing, sorting, clipping by value, and element-wise addition, depending on the type of the N-d array—floating-point, 32-bit integer, complex, etc. It directly relies on the external libraries.

Owl_dense_ndarray_generic

Using Owl_dense_ndarray_common, this module implements the more complex polymorphic operations on N-d arrays, making up the bulk of the N-d array implementation in Owl. Some examples include reduction, mapping, or convolution operations.

[6] In OCaml, a module is a piece of code containing types, functions, values, and potentially other modules. They can be thought of as similar to C++ namespaces, but they are much more powerful.

[7] An OCaml functor can be seen as a function mapping modules to modules—they take a number of modules as input, and using those define a new module. For example, OCaml provides a Set functor which, given a module implementing total ordering for elements of type t, it produces a module representing sets of elements of type t.

`Owl_dense_ndarray_{s / d / c / z}`

These four modules specialise the previous module for each possible type of the elements in the N-d array—single-precision floating-point, double-precision floating-point, complex number, and integer (z) number, respectively. The library user will choose between these according to their requirements. All four conform to a common signature[8], which is important in the next modules.

`Owl_algodiff_generic`

This module is the lower-most functor in the `Owl` functor stack. It implements algorithmic differentiation of various precision, starting from the implementation of N-d arrays fed as input to the functor. For example, the `Owl_algodiff_generic` functor produces double-precision algorithmic differentiation when applied to `Owl_dense_ndarray_d`.

`Owl_optimise_generic`

The next functor in the stack builds upon algorithmic differentiation, preparing support for the regression and neural network features in `Owl`. It provides functions ranging from batch and regularisation handling, gradient computation algorithms, to model checkpointing and stopping conditions. After this module the functor stack splits into two branches.

`Owl_regression_generic`

The first branch implements `Owl`'s support for regression. This is the simpler of the two branches, because most of the support required for implementing regression is packaged in `Owl_optimise_generic`.

`Owl_neural_neuron` and `Owl_neural_graph`

Finally, the branch providing support for neural networks in `Owl` is split into two modules. The first, `Owl_neural_neuron` relies on `Owl_generic_algodiff` to implement functions required at the network layer level, from activations, dropouts, and embeddings, to recurrent structures and arbitrary differentiable functions. `Owl_neural_graph` then handles the infrastructure required to connect individual layers into a complete neural network.

This overview indicates that, because of `Owl`'s functor-based structure, we must focus most of our effort on writing alternative implementations for the N-d arrays. However, some additional work is required in rewriting and restructuring the utilities modules, and also ensuring the functors can also work with a pure `OCaml` implementation of N-d arrays.

[8] An `OCaml` signature represents an interface specification for a module.

2.2 Js_of_ocaml & BuckleScript

With some of the requirements fleshed out, we had to choose a suitable OCaml-to-JavaScript compiler, and determine exactly what its limitations were, before implementation commenced. We carefully analysed and experimented with two viable candidates—Js_of_ocaml[9] and BuckleScript[10]. Both compilers have almost no support for Unix or file-system operations, which is somewhat unsurprising considering the output code might run in a browser. In terms of producing JavaScript code, BuckleScript starts from the high-level raw-lambda representation produced by the OCaml compiler, while Js_of_ocaml begins with the low-level bytecode. This gives a minor advantage to BuckleScript, because the generated code will be more readable than the alternative. However, we assumed the typical Owl has no interest to read or modify the generated JavaScript code, so this did not weigh heavily in our consideration. Instead, several issues were identified with BuckleScript, which made us commit to Js_of_ocaml:

- At the time of our deliberations, the BuckleScript project was still very young, and its documentation was unsatisfactory or even missing, significantly slowing down our progress;
- It also had no support for Bigarray[11]—an OCaml standard library providing an efficient implementation of multi-dimensional arrays. Of course, an alternative was to rely on primitive arrays for the pure OCaml implementation of N-d arrays, but that was considered to introduce significant avoidable work, with no immediate benefits;
- BuckleScript mainly targets existing JavaScript workflows, for instance interoperating with the NPM[12] JavaScript package manager. Js_of_ocaml instead focuses on the OPAM[13] OCaml package manager, which is what Owl uses, and what we presume is part of the toolchain of a typical OCaml programmer using Owl.

Js_of_ocaml has two other advantages—it integrates well with Dune/Jbuilder[14], the build system used by Owl, and provides support for related OCaml libraries which will prove useful for our second objective (see §3.3).

[9] http://ocsigen.org/js_of_ocaml/
[10] https://bucklescript.github.io
[11] https://caml.inria.fr/pub/docs/manual-ocaml/libref/Bigarray.html
[12] https://www.npmjs.com
[13] https://opam.ocaml.org
[14] https://github.com/ocaml/dune

2.3 Pure `OCaml` utilities

Due to their prevalent use in the library, the first step was to implement pure `OCaml` alternatives to the utilities functions which were relying on external libraries. Besides the slicing functions, which were incorporated in the N-d array implementation (see §2.4), most functions that required work were from the `Owl_maths` and `Owl_stats` modules. In the mathematical module, the vast majority of functions were already implemented in `OCaml` standard libraries, with only a few functions which had to be computed using the following formulae:

$$\log_2 x = \frac{\log x}{\log 2} \tag{2.1}$$

$$\operatorname{arcsinh} x = \log(x + \sqrt{x^2 + 1}) \tag{2.2}$$

$$\operatorname{arccosh} x = \log(x + \sqrt{x^2 - 1}) \tag{2.3}$$

$$\operatorname{arctanh} x = \frac{1}{2} \log \frac{1 + x}{1 - x} \tag{2.4}$$

$$\operatorname{relu} x = \max(x, 0) \tag{2.5}$$

$$\operatorname{sigmoid} x = \frac{1}{1 + e^{-x}} \tag{2.6}$$

For the statistical module, `OCaml` provides a random number generator capable of producing standard uniformly distributed floating-point numbers $\mathrm{U} \sim \mathcal{U}(0,1)$. We used this to create the following random number generators:

- General uniformly distributed random number as:

$$X \sim \mathcal{U}(a,b) = a + (b - a)\mathrm{U}. \tag{2.7}$$

- Bernoulli random variable as:

$$X \sim \mathcal{B}er(p) = \begin{cases} 1, \text{ if } \mathrm{U} \leq p \\ 0, \text{ otherwise.} \end{cases} \tag{2.8}$$

- Random number distributed according to a Gaussian distribution with mean μ and variance σ^2, computed using the Box-Muller transform[15] as:

$$\begin{aligned} \mathrm{U}_n &\sim \mathcal{U}(0,1), \forall n \in N \\ Z_n &= \sqrt{-2 \log \mathrm{U}_n} \cos(2\pi \mathrm{U}_{n+1}) \\ Z_{n+1} &= \sqrt{-2 \log \mathrm{U}_n} \sin(2\pi \mathrm{U}_{n+1}) \\ X \sim \mathcal{N}(\mu_n, \sigma_n^2) &= \mu_n + \sigma_n Z_n. \end{aligned} \tag{2.9}$$

[15] http://mathworld.wolfram.com/Box-MullerTransformation.html

2.4 Pure OCaml N-d array

A significant amount of work was required to provide an alternative, pure OCaml implementation for Owl's N-d arrays. This is because many complex operations required higher in the functor stack are bundled in the N-d array module, in order to achieve the most efficient implementation for native targeting. Examples include convolution, backpropagation through convolutional layers, and reduction operations. Overall, more than 170 functions are exposed by the newly implemented N-d array module. In fairness, many of those share common steps which were collected as much as possible in generalised code, in order to reduce the implementation effort. Nevertheless, because of the sheer number of functions implemented, we can only present in the following a small selection of those considered to be of more interest.

As mentioned in §2.2, we use the Bigarray OCaml module for the underlying data structure in our implementation. Note that the implementation of certain functions in the new N-d array module is visibly not the most efficient possible. Optimising these functions was only a secondary objective which will be extensively explored in future iterations—our main objective was to demonstrate compiling Owl to JavaScript is possible with full functionality, albeit losing performance in some aspects. However, we were careful to efficiently access memory, where possible, by using efficient block memory functions such as blit or slice_left from the Bigarray module. These are also compiled by Js_of_ocaml to efficient operations manipulating views into arrays, rather than naïvely copying segments or iterating over them.

2.4.1 Convolution operations

Arguably the most complex operations in the new N-d array module, there are in total 21 functions dealing with convolution on N-d arrays. They implement standard convolution, max-pooling[16] , average-pooling[17] , as well as backward passes for those. The backward passes are used in the neural network backpropagation algorithm, calculating the gradient of some value with respect to either the input (for the three kinds of convolution), or the convolution kernel (for the standard convolution), given the gradient of the same value with respect to the output (see §2.4.1.2 for an explanation). Each comes in $1D$, $2D$, and $3D$ versions, depending on the dimensionality of the input. The $2D$ and $3D$ functions are very similar, with just

[16]Similar to standard convolution, but the kernel only describes the size of a window sweeping over the input. For each window position, the output value is the maximum input value overlapped by the window.

[17]Same as max-pooling, but the output value is given by the average of the elements overlapped by the window.

a small adjustment for the higher-dimensionality, while the $1D$ versions simply call the $2D$ functions after a reshaping of the input and kernel. Therefore, we focus on the $2D$ functions here.

2.4.1.1 Forward passes

Although Fast Fourier Transforms are perhaps the only answer to efficient convolution operations, they were considered superfluous to our current objectives due to their intricacy, instead opting for more naïve implementations. We took care, however, to optimise looping order as much as possible. For forward passes, we first iterate over the output and then the kernel, in order to be able to use an accumulator, and avoid multiple memory operations on the output N-d array. The precise input cell can be determined from the positions in the output and the kernel, the convolution strides, and the padding dimensions as follows:

$$\begin{aligned} \text{in}_{col} &= \text{out}_{col} \times \text{stride}_{col} + \text{kernel}_{col} - \text{pad}_{left} \\ \text{in}_{row} &= \text{out}_{row} \times \text{stride}_{row} + \text{kernel}_{row} - \text{pad}_{top}. \end{aligned}$$

Once the above are calculated, we can read the value from the input when the position is inside the limits, or use the value of 0 which is padded around the input when the position is outside. `Owl` supports two types of convolution padding—`VALID`, in which case no padding is added (and in_{col} and in_{row} will always be inside the limits), or `SAME`, which adds padding with 0 values around the input, such that the output has the same dimensions as the input dimensions divided by the strides. Mathematically, this gives the following formulae for the output width and left-padding, with output height and top-padding computed similarly:

$$\begin{aligned} \text{out}_{width} &= \begin{cases} \left\lceil \dfrac{\text{in}_{width} - \text{kernel}_{width} + 1}{\text{stride}_{col}} \right\rceil & \text{, if } \texttt{SAME} \text{ padding} \\ \left\lceil \dfrac{\text{in}_{width}}{\text{stride}_{col}} \right\rceil & \text{, if } \texttt{VALID} \text{ padding} \end{cases} \\ \text{pad}_{left} &= \max\left(\left\lfloor \frac{(\text{out}_{width} - 1) * \text{stride}_{col} + \text{kernel}_{width} - \text{input}_{width}}{2} \right\rfloor, 0 \right). \end{aligned}$$

Figure 2.2 presents a visualisation of the standard $2D$ convolution implementation in the new N-d array module. The $2D$ max-pooling and $2D$ average-pooling are very similar, with a few differences. Firstly, the output has the same number of channels

as the input, while the kernel is given by two integers representing the window size. This means we have one fewer loop, and instead of using a summing accumulator, we compute either the maximum or the average in the window.

2.4.1.2 Backward passes

We implement the backward convolution passes using a comparable approach. To illustrate this algorithm, consider the standard $2D$ convolution backward pass computing the gradient with respect to the input. For each position $\vec{i}$ in the input, we want to compute $\text{in}'_{\vec{i}} = \frac{\partial V}{\partial \text{in}_{\vec{i}}}$, for some value V, given $\text{out}'_{\vec{o}} = \frac{\partial V}{\partial \text{out}_{\vec{o}}}$ for all positions $\vec{o}$ in the convolution output. But in the forward pass we computed the output as:

$$\text{out}_{\vec{o}} = \sum_{\vec{k}} \text{kernel}_{\vec{k}} \times \text{in}_{\vec{i}} \tag{2.10}$$

where $\vec{k}$ iterates over the kernel, and $\vec{i}$ is the corresponding position in the input. Note that we assume no padding in this case, to simplify the explanation. Therefore, we can rewrite:

$$\begin{aligned}
\text{in}'_{\vec{i}} = \frac{\partial V}{\partial \text{in}_{\vec{i}}} &= \sum_{\vec{o}} \frac{\partial V}{\partial \text{out}_{\vec{o}}} \times \frac{\partial \text{out}_{\vec{o}}}{\partial \text{in}_{\vec{i}}} \\
&= \sum_{\vec{o}} \text{out}'_{\vec{o}} \times \frac{\partial \text{out}_{\vec{o}}}{\partial \text{in}_{\vec{i}}} \\
&= \sum_{\vec{o}} \text{out}'_{\vec{o}} \times \text{kernel}_{\vec{k}}
\end{aligned} \tag{2.11}$$

where $\vec{o}$ iterates over the output positions affected by $\text{in}_{\vec{i}}$, and $\vec{k}$ is the corresponding position in the kernel. This has a similar form to Equation (2.10), and is a convolution operation, with the input and output arrays swapped. We can indeed iterate over the kernel instead, and obtain:

$$\text{in}'_{\vec{i}} = \sum_{\vec{k}} \text{kernel}_{\vec{k}} \times \text{out}'_{\vec{o}} \tag{2.12}$$

Therefore, the implementation is very similar to that presented for the forward pass, with the appropriate changes to loop ordering. An analogous derivation leads to the same conclusion for the backward pass with respect to the kernel. A special case occurs for the max-pooling backward pass, where the sum operator is replaced by a

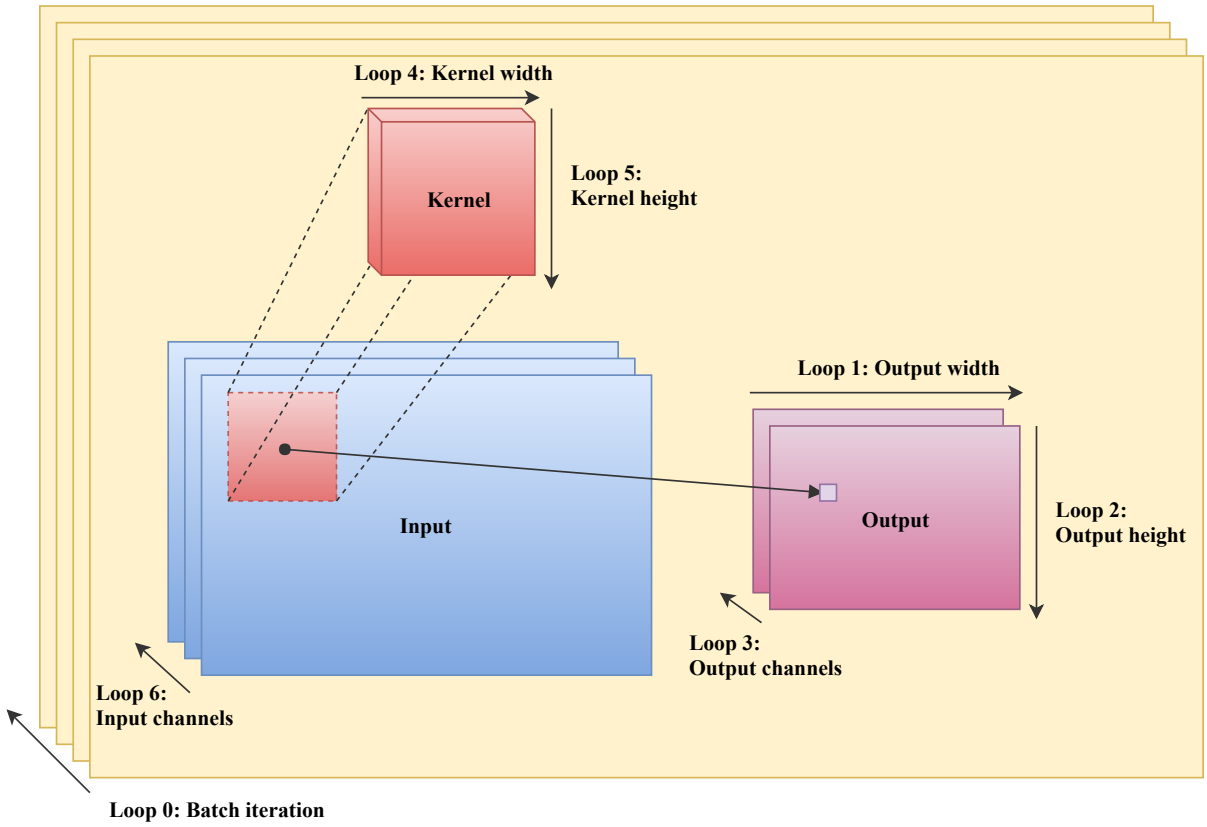


Figure 2.2: Illustration of standard 2*D* convolution implementation. Loop 0 is the outermost loop, while loop 6 is the innermost. Inside the body of loop 3 the exact position of the output cell is determined, and we can use an accumulator to compute its value while iterating over the kernel. The exact position in the input can be determined from the positions in the output and kernel, the convolution strides, and the padding type. Note that this guarantees only one memory access for each output cell. If iteration over input and output were to be swapped (iterate over the input first), then we cannot use an accumulator, and we must update each output cell multiple times, affecting performance for medium-to-large kernels.

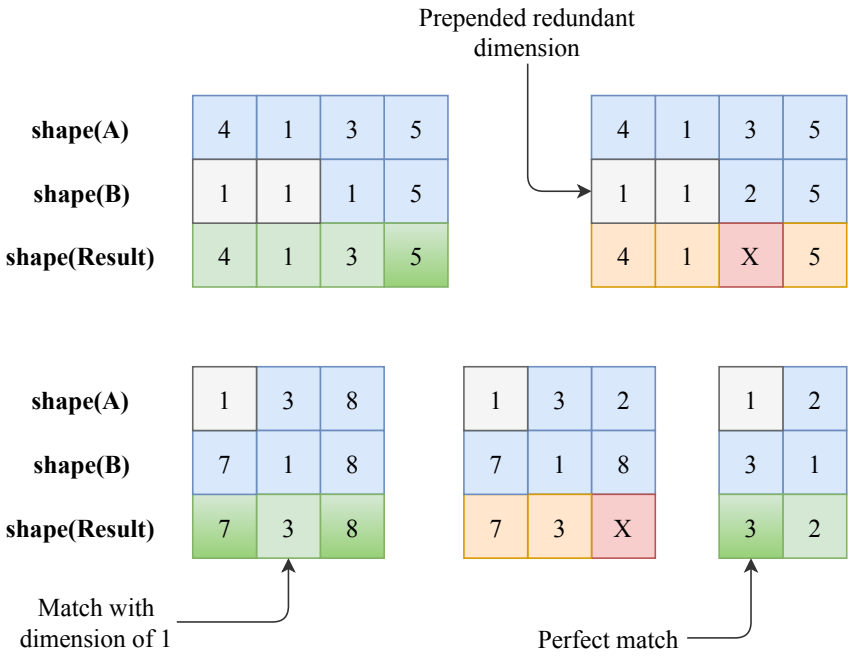


Figure 2.3: Broadcasting shape matching for a broadcasted operation between N-d arrays `A` and `B`. Blue represents the actual shape of the N-d arrays, with grey dimensions of 1 prepended if necessary. Red is mismatches which make the broadcasted operation fail.

max operator in the forward pass. However, adapting the algorithm for this case is trivial—in Equation (2.11) we can make the following replacement:

$$\frac{\partial \text{out}_{\vec{o}}}{\partial \text{in}_{\vec{i}}} = \begin{cases} 1, & \text{if out}_{\vec{o}} = \text{in}_{\vec{i}} \\ 0, & \text{otherwise.} \end{cases} \tag{2.13}$$

2.4.2 Broadcasting operations

Support for broadcasting operations is an important feature of any numerical library. The new N-d array module implements at least seven functions with broadcasting support, including addition, subtraction, multiplication, division, four-quadrant inverse tangent (`atan2`), and element-wise exponentiation. The behaviour is similar to that in `NumPy`—for a broadcasting operation on two N-d arrays, we compute their shapes, and determine whether they match. The shapes match if, starting from the right-most dimensions in both, each pair of dimensions match—both are equal, or one of them is 1. For N-d arrays of different ranks (number of dimensions), we reshape the one having lower rank with redundant prepended dimensions of 1. This process is illustrated in Figure 2.3, while Figure 2.4 demonstrates the broadcasting algorithm on an example.

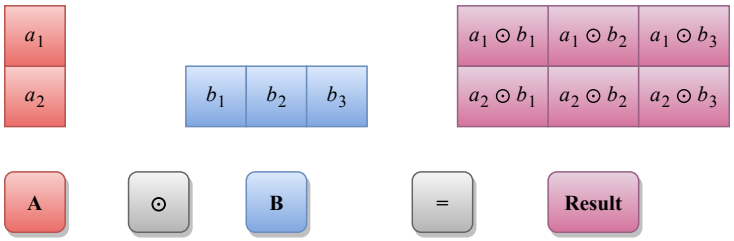


Figure 2.4: A demonstration of the broadcasting operation $\odot$ between N-d arrays A of shape $(1, 2)$, and B shape (3), giving a result of shape $(3, 2)$.

2.4.3 Matrix operations

Another important category of functions are those manipulating N-d arrays of rank two, i.e. on matrices. The exact implementation for many of them is not noteworthy, but it is important to point out the computational complexities of some. For instance, the matrix multiplication has worst-case complexity $O(n^3)$, worse than the fastest known algorithm for matrix multiplication. Matrix inversion is performed using the algorithm described in [3], which has the same worst-case complexity $O(n^3)$.

2.5 Isolating `Owl Base`

As mentioned in this chapter's introduction, once all the required modules were implemented, we proceeded by creating a sub-library inside `Owl`, called `Owl Base`. This sub-library isolates the pure `OCaml` modules from `Owl`. These modules do not rely on any external *C* libraries, nor do they interact with the file-system or `Unix` system[18]. `Owl Base` includes the newly implemented modules we described in previous sections, and covers most features offered by `Owl`. Most importantly, `Owl Base` can be fully compiled to `JavaScript`, which accomplishes our first objective.

There are two main reasons for which `Owl Base` was created. Firstly, `Owl` performs certain static initialisation routines such as random generator seeding, which are executed whenever the library is used in any program. These operations rely on external `C` libraries, which lead to runtime errors if the code is compiled to `JavaScript`, so they must be prevented from running in that case. Excluding them from `Owl Base` will accomplish the task when programs only use the sub-library instead of the rest of `Owl`. Of course, an alternative solution could have been refactoring `Owl` without isolating a sub-library. However, linking a smaller `Owl Base` is especially beneficial when deploying the `JavaScript` compilation output in a browser—the code will be much

[18]Except for functions obtaining the system time, which are compiled by `Js_of_ocaml` to `JavaScript` alternatives.

smaller and can be distributed more easily. This constitutes our second consideration for creating Owl Base.

3

Actor Pure

This chapter expands on the work from Chapter 2, presenting our progress towards support for distributed `Owl` computations running in the browser. We explore the `Actor` library, outline the requirements imposed by our objective, then describe how these are met. As in Chapter 2, we aim to eliminate *impure* dependencies from `Actor`, which cannot be compiled to `JavaScript`. Therefore, this new portion of the library is named `Actor Pure`.

3.1 Initial design of `Actor`

`Actor`[1] is a specialised distributed data processing framework, functioning as `Owl`'s distributed and parallel computing engine. The interaction between the two is designed for high composability and ease of use—distributed versions of core `Owl` modules can be obtained in just a few lines of code. For instance, the following single line of code creates a module for distributed training models for neural networks, using a parameter-server distribution model:

```
module M = Owl_neural_parallel.Make (Owl_neural_feedforward) (Actor_param)
```

`Actor` supports three distribution models—the Parameter-Server (PS) model used above, a Peer-to-Peer (P2P) model, and a Map-Reduce (MAPRE) model. We concentrated our efforts on adapting the Parameter-Server model for `JavaScript` compilation, because it is the most intuitive choice among the models available, especially for Machine Learning applications. In this model, workers perform training steps using local copies of the parameters, then send the computed gradients to the parameter-server. The parameter-server applies all updates to the global state, which the workers will fetch regularly to replace their local copies, thus ensuring

[1] https://github.com/owlbarn/actor

consistency between workers.

Nevertheless, our work lays a solid foundation for supporting `JavaScript` deployment for the other models. We do not foresee any significant architectural changes required—we just need to follow the recipe presented in §3.3 to make use of the new infrastructure. However, because models in `Actor` are implemented somewhat independently, and we aim for conciseness, we focus our description of `Actor`'s initial design on the PS model, which established the requirements guiding our implementation.

The foundation of the PS model is implemented as a distributed key-value store, supporting `get/set` operations. On top of this, `Actor` can build a typical PS API, with operations such as `register`, `push`, or `pull`. These operations are used by `Owl` to distribute data processing or machine learning training. There are three important modules in the PS implementation. First, the `Actor_paramserver` module represents, as the name suggests, the parameter-server which maintains in memory the global state of the parameters. The `Actor_paramclient` module relays certain operations to the parameter-server, which performs them locally, and for instance, for a `get` request, sends back the value. Finally, the `Actor_param` modules provides an opaque interface over the previous two—it automatically determines whether the current process is the parameter-server or a client, and acts accordingly for each operations. Any program using `Actor_param` is unaware whether it also performs the role of a PS, but it can expect `get` results to be up-to-date, and `set` operations to eventually commit.

Regardless of the distribution model used, `Actor` relies on two modules to create jobs and discover where they can be deployed. The first one is the `Actor_worker`, a program capable of starting jobs on the platform it runs. The `Actor_manager` is the second module, performing the role of a centralised organiser which coordinates and keeps track of existing jobs and workers. In practice, the manager is located at an address known by everyone. Whenever a worker starts on a platform, it registers itself with the manager, essentially signalling that the platform it runs on is available to execute jobs.

Suppose that once the manager and a few workers are running on a number of computers, the user wishes to deploy a distributed job defined with the help of `Owl` and `Actor`. The job executable, once started locally by the user, will contact the `Actor_manager`, asking to register and distribute itself wherever possible. At this point a number of things happen—first, the manager forwards the request to all the registered workers, which will then spawn a new process representing one instance of the distributed job. Note that, in this version of `Actor`, the job executable must be

available on all worker platforms targeted for deployment, before the job starts. The initial job started by the user becomes the job master, while the other instances are job workers. Finally, all instances will cooperatively perform the job, following the chosen distribution model. The entire process is illustrated in Figure 3.1, using the parameter-server model as an example. There, `Actor_param` automatically determines that the job master is the parameter-server, while the other instances are clients. However, the rest of the program is unaware of this.

3.2 Requirements analysis

Using this overview of `Actor`'s design we can pinpoint which aspects must be adapted in order to achieve our second objective—support for distributed deployment of `Owl` & `Actor` programs on `JavaScript` platforms. Standard `Actor` relies on `OCaml` bindings for the open-source `ZeroMQ` library[2] for efficient and reliable communication between the `Actor` managers, workers, and jobs running on various computers. This is an external dependency not compilable to `JavaScript`, and must be replaced. The official `ZeroMQ` project also provides `JavaScript` bindings, but unfortunately these only support `Node.js`, meaning we would lose the ability to deploy in the browser. We considered using third-party `JavaScript` bindings for `ZeroMQ`, but the only candidate project we identified is outdated and inefficient—it relies on a server to relay messages between browsers, which would be a major bottleneck in ML applications where data and parameters are heavily exchanged. Therefore, the main challenge becomes implementing an efficient `JavaScript` communication layer, and inserting it into `Actor` as a replacement for `ZeroMQ`. Because we wish not to disrupt the `Actor` implementation excessively, we intend to make the replacement as similar to `ZeroMQ` as possible. This challenge is tackled in §3.3,§3.4, and §3.5.

Our second major challenge is prompted by the `Actor_worker` module, which relies on the `Unix` `fork` and `execv` commands to spawn new jobs when needed. As we intend to run programs on any operating system in the browser, we must create a mechanism through which workers can start jobs with a list of arguments (like command-line arguments in `Unix`). This is discussed in §3.6.

[2] http://zeromq.org

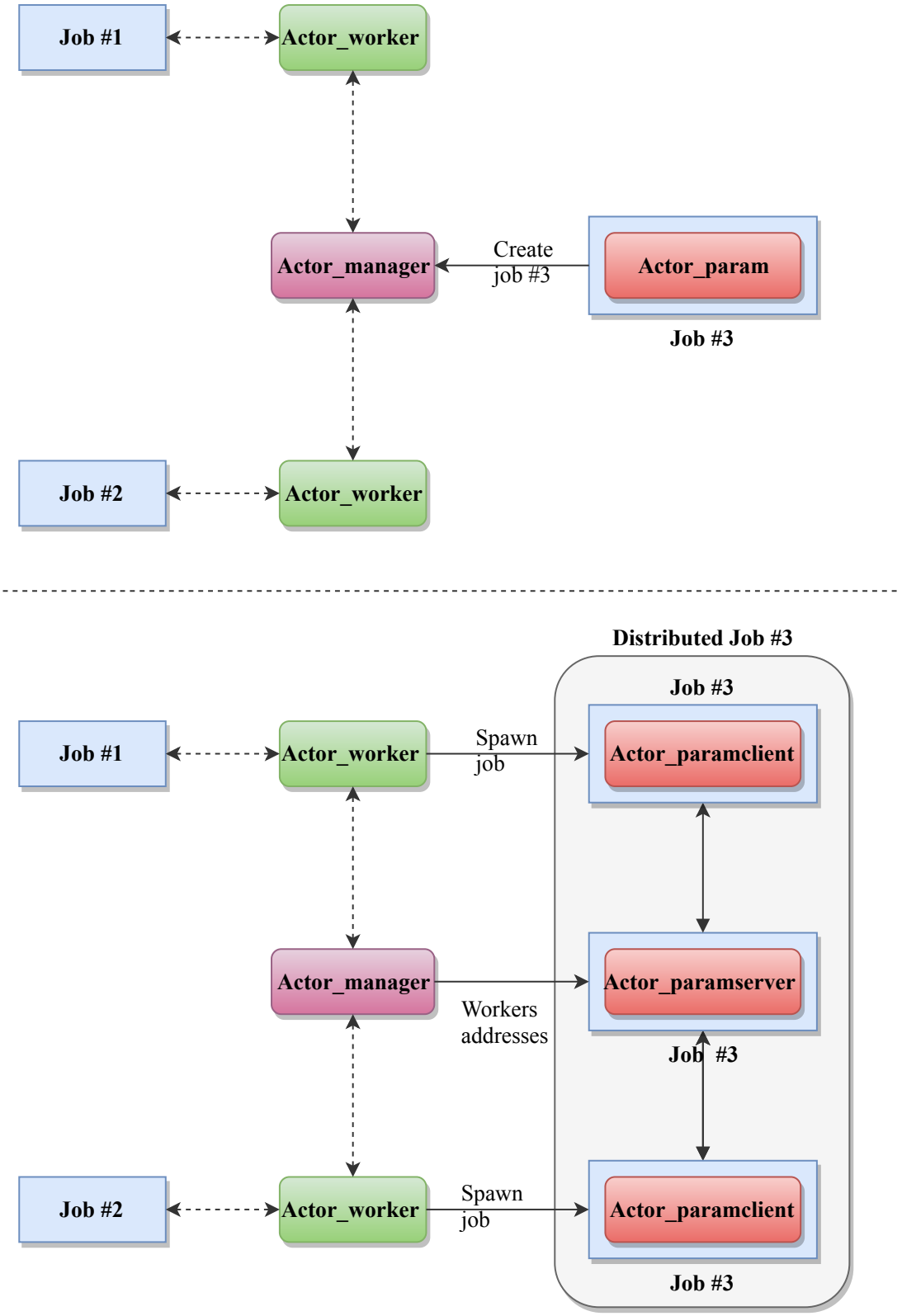


Figure 3.1: Deploying distributed jobs using the parameter-server distribution model from `Actor`. In the first step (upper half), the user executes the job master, which will send a job registration request to the `Actor_manager`. This request will be forwarded to the `Actor_workers`, which will spawn new instances of the job (lower half). The master and the spawned jobs will then cooperate according to the parameter-server model. `Actor_param` automatically determines whether each instance is the PS or just a client, but hides this from the rest of the program.

3.3 From blocking to non-blocking communication

Multiple operations in `Actor` are based on blocking communication primitives. For instance, an `Actor_paramclient get` operation blocks until the client receives the response from the PS. However, `JavaScript` is event-driven, and, from the perspective of the programmer, single-threaded[3]. This makes it impossible to implement a truly blocking communication layer in `JavaScript`, causing a significant hurdle in our plans. We investigated a number of possible solutions, including the following:

- Implement the communication layer using the `async/await` construct[4] available in modern `JavaScript` standards, which would camouflage the asynchronous nature of `JavaScript`. Unfortunately, `Js_of_ocaml` only produces code compliant to older `JavaScript` standards, which cannot use the `async/await` constructs. We could circumvent this by writing a transpiler which adapts the code produced by `Js_of_ocaml` to use the `async/await` construct, but this remedy is inelegant and risky.

- Expose a callback-based interface from the communication layer, and modify `Actor` to a continuation-passing style (CPS) implementation for communication operations. However, any simple function call changed to a CPS call will make the caller also require a CPS argument, requirement which will cascade down the call stack[5] and necessitate extensive effort to fully adapt `Actor`.

Fortunately, the previous idea can be improved using `Lwt`, a collection of `OCaml` libraries for asynchronous programming with promises. `Lwt`, like `Js_of_ocaml`, is part of the larger `Ocsigen`[6] framework, meaning the two sub-projects have a high degree of inter-operability. A `Lwt` promise is a placeholder for a value not yet computed, in which case the promise is in a *pending* state. When the corresponding value is computed, the promise enters a final *resolved* state, and the value can be extracted. Instead, if the computation fails, the promise commits to a *rejected* state, exposing an exception detailing why the computation failed. In general, the type `k Lwt.t` represents a promises which can be resolved with a value of type `k`. To illustrate this mechanism, Table 3.1 presents a program which synchronously reads an integer, doubles it, then prints the result, alongside its asynchronous alternative using `Lwt` promises.

[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await

[5] Consider `g() = (f()) + 10` calling `f() = read()`, where `read` is blocking. If we change `read` to CPS, we also need to change `f`, then `g`, leading to `f cont = read (λx.cont x)` and `g cont = f (λx.cont (x + 10))`.

[6] http://ocsigen.org

Blocking I/O	Non-blocking I/O

`val read_blocking : unit → int`
`val print_blocking : int → unit`
`----------`
 	
`let (x : int) = read_blocking() in`
`let (result : unit) =`
`  print_blocking (x + x)`
	`val read_lwt : unit → int Lwt.t`
	`val print_lwt : int → unit Lwt.t`
	`val Lwt.bind : α Lwt.t →`
	`  (α → β Lwt.t) → β Lwt.t`
	`----------`
	`let (x_lwt : int Lwt.t) = read_lwt() in`
	`let (result : unit Lwt.t) =`
	`  Lwt.bind x_lwt (fun (x : int) →`
	`  print_lwt (x + x))`

Table 3.1: Blocking to non-blocking I/O using `Lwt`. Type annotations are added for clarity. Note the result on the right-hand side is a unit promise, which will be resolved after the result is printed.

A callback can be attached to a promise using `Lwt.bind`. When the promise is *resolved* with a value, the callback is executed with that value as argument. The callback can perform other asynchronous operations, and return a final promise. `Lwt.bind` returns a third promise, which is fulfilled when the previous two promises are. This entire process is very similar to how monads operate.

The earlier example prompts some concerns regarding code readability and the effort required to introduce `Lwt` into `Actor`. Because of `Lwt.bind`, the code using the value of a promise must be refactored into a λ-function, which easily becomes unwieldy when multiple such cases are sequenced. Luckily, `Lwt` provides an `Ocaml` syntax extension that eliminates this issue. This extension provides constructs such as `let%lwt` instead of `let`, removing the need for `Lwt.bind` and λ-functions. This is a huge simplification, and one of the main reasons we chose `Lwt` over CPS or other approaches.

We proceeded to wrap synchronous `ZeroMQ` functions into `Lwt` promises, thus creating pseudo-asynchronous functions. This means sending/receiving messages remains a blocking operation while we use `ZeroMQ`, but this will be replaced in the next phases with a truly asynchronous module. At this point `Actor` should perform exactly the same, thus we can use the existing tests to ensure this refactoring did not introduce bugs. Table 3.2 exemplifies the refactoring on a snippet of `Actor` code, highlighting the cleanness and ease of using `Lwt` with the syntax extension.

Blocking messaging

```
val ZMQ.send_sync : msg_t → unit
val ZMQ.recv_sync : unit → msg_t
val value_from_msg : msg_t → value_t
----------------------------------------
let get k =
  let k' = stringify k in
  ZMQ.send_sync (PS_Get, k');
  let msg = ZMQ.recv_sync () in
  let v = value_from_msg msg in
  m
----------------------------------------
val get : key_t → value_t
```

Pseudo-asynchronous messaging

```
val ZMQ_Lwt.send : msg_t → unit Lwt.t
val ZMQ_Lwt.recv : unit → msg_t Lwt.t
val value_from_msg : msg_t → value_t
----------------------------------------
let get k =
  let k' = stringify k in
  ZMQ_Lwt.send (PS_Get, k') ;%lwt
  let%lwt msg = ZMQ_Lwt.recv () in
  let v = value_from_msg msg in
  Lwt.return m
----------------------------------------
val get : key_t → value_t Lwt.t
```

Table 3.2: Refactoring `Actor` to use asynchronous messaging with `Lwt`. The `get` function fetches the value associated with a key from the PS. The `let%lwt` construct is a syntactic-sugar replacement for `Lwt.bind`, while `;%lwt` is a similar replacement specialised for unit promises. The refactored function is asynchronous, returning a promise which resolves to the value when received from the PS. `ZMQ_Lwt` is the naïve wrapper over `ZeroMQ`, which will be replaced. Note we renamed functions and omitted certain arguments on both sides, such that the example is clearer.

3.4 Peer Communication Layer

This section presents the Peer Communication Layer (PCL)—the `JavaScript` foundation for the desired `ZeroMQ` replacement. We had to implement at least some part of the replacement in `JavaScript`, due to the required browser support discussed in §3.2. However, we limited this as much as possible and aimed for high modularity, because programming in `OCaml` means cleaner and less error-prone code.

Our intention with PCL is to provide capabilities very similar to TCP/IP sockets. We name these socket abstractions `PCLSockets`. Same as with network sockets, the user is able to *bind* a local (i.e. corresponding to this process) `PCLSocket` to listen on, and *connect* to a remote (i.e. corresponding to a different process) `PCLSocket`. Furthermore, similarly to TCP, messages can be sent/received *reliably* and *in-order* between local and remote `PCLSockets`. Figure 3.2 illustrates this comparability. The replacement for `ZeroMQ` can then be implemented on top of `PCLSockets` just like `ZeroMQ` is built on top of TCP/IP sockets.

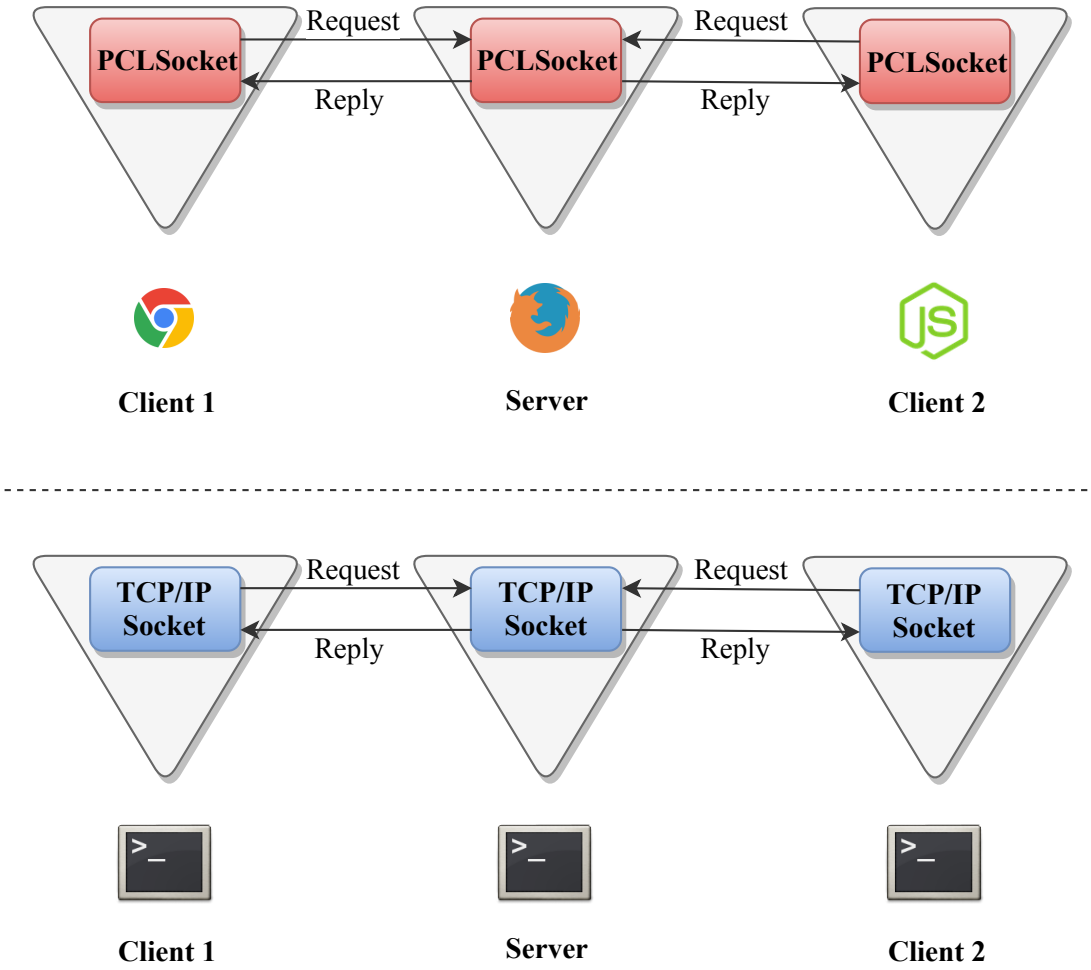


Figure 3.2: Example use of `PCLSockets` as a browser alternative for TCP/IP sockets. Programs running in the browser can communicate with each other using `PCLSockets`, just as they would natively with TCP/IP sockets.

3.4.1 WebRTC

The feasibility of our plans relied on efficient communication between browsers. We researched options, but there are in fact not many protocols with the desired properties available. `WebSocket`[7] was a candidate protocol, but browsers cannot initiate connections, and most importantly, a server is needed to relay messages between browsers. This is not a scalable solution in the long run—the server would be a significant bottleneck for P2P distributed applications with high network traffic.

Fortunately, `WebRTC`[8] meets our requirements. An open-source project, the `WebRTC` protocol supports a multitude of browsers and platforms, including *Chrome*, *Firefox*, *Opera*, *Android*, and *iOS*. It provides direct, browser-to-browser Real-Time

[7] https://tools.ietf.org/html/rfc6455
[8] https://webrtc.org

Communications capabilities, for audio, video, and data. WebRTC might also require a relay TURN[9] server, when the peers are unable to communicate directly from behind a *Network Address Translation* device. Nonetheless, this rarely happens, and it is impossible to avoid using a relay in that case anyway.

We decided to use the WebRTC RTCDataChannel as a *reliable* and *ordered* transport primitive for sending data between two PCLSockets. However, setting up the channel is an intricate process, having certain requirements, so we will describe it gradually.

3.4.2 Signalling channel

Perhaps counterintuitively, a prior connection is required between two peers in order to set up the RTCDataChannel, because the process necessitates exchanging data such as network reachability information. We used the Socket.io[10] library to implement this signalling channel. The library is distributed under the MIT license[11], uses the WebSocket protocol, and it allows us to build bi-directional communication between the peers and a central server. This server is a simple message proxy, ensuring peers can reach each other without WebRTC. Note that this is used exclusively for opening and keeping the RTCDataChannel alive, using a small number of tiny messages—it does not introduce the bottleneck discussed in §3.4.1, because the bulk of the P2P communication is over the direct RTCDataChannel.

In a short overview, we implemented the signalling channel to operate as follows. The signalling server keeps track of all connected peers, as well as which PCLSocket corresponds to which peer. A peer can register (bind) a PCLSocket with the signalling server using a unique name. This imitates how an IP address and a port number identifies a network socket. Peers can send messages for a certain PCLSocket name to the server, which will forward them to the appropriate peer. The signalling server queues messages sent towards unknown sockets, which are relayed when the socket is registered. Figure 3.3 depicts the signalling channel used in a simplified scenario.

[9] https://tools.ietf.org/html/rfc5766
[10] https://socket.io
[11] https://opensource.org/licenses/MIT

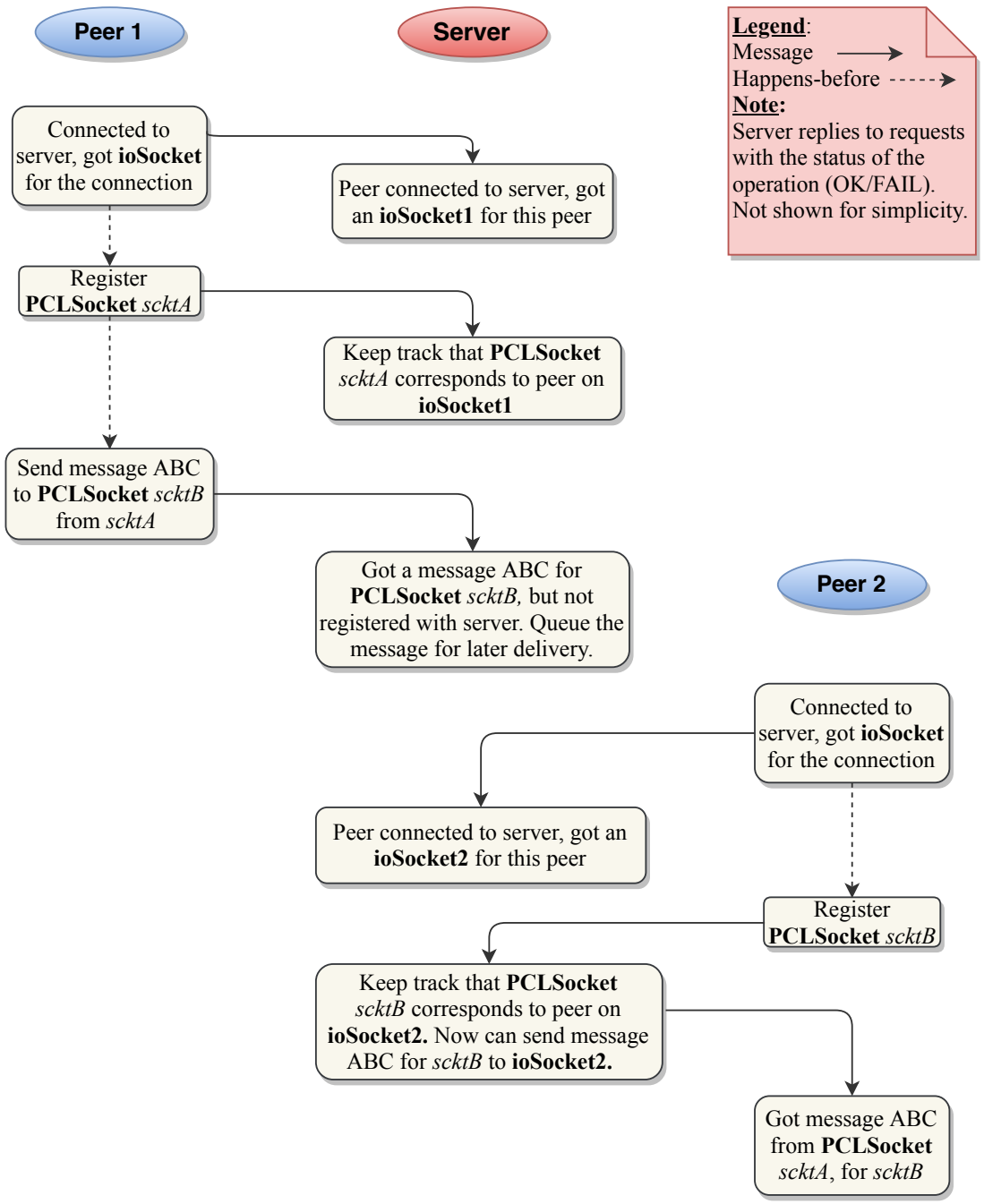


Figure 3.3: Flowchart illustrating how the signalling channel server operates. Time flows downwards. The `Socket.io` library uses a `ioSocket` to represents the connection between a client and the server. Messages over the signalling channel, including server requests, have replies sent back with the status of the operation (e.g. error message on failure, or acknowledgement on success).

3.4.3 Setting up PCLSocket-to-PCLSocket connections

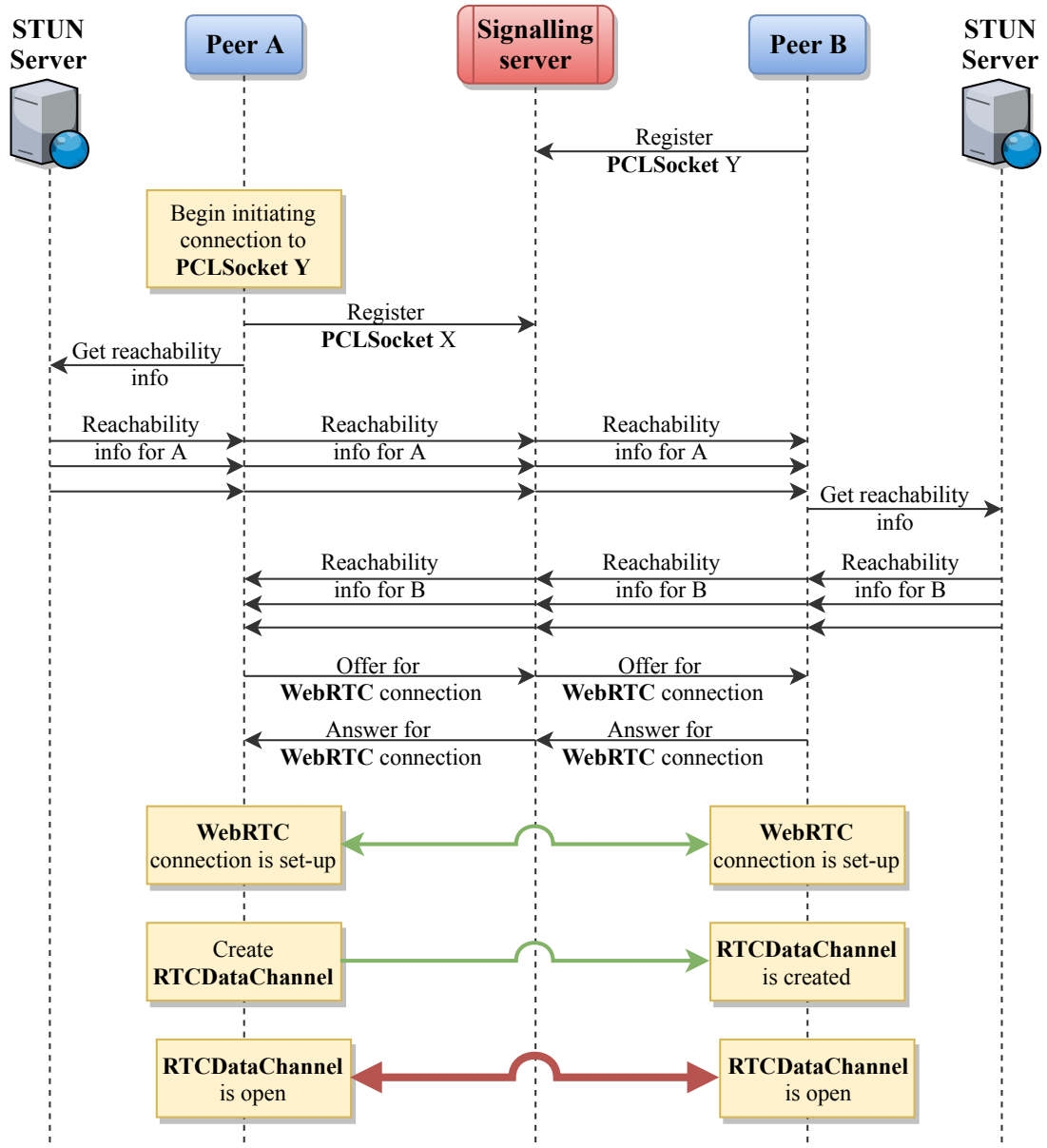


Figure 3.4: Initiating an `RTCDataChannel` between two `PCLSockets`. Green arrows represent communication between *A* and *B* performed by `WebRTC` behind-the-scenes. The red arrow is the open `RTCDataChannel` between `PCLSockets` *X* and *Y*.

As mentioned in §3.4.1, we consider the `RTCDataChannel` as analogous to a TCP connection. When a peer *A connects* to a remote `PCLSocket` registered by peer *B*, the layer first binds a local `PCLSocket` to listen on, then sets up the `RTCDataChannel` between the two. The process begins with both peers exchanging network reachability information, which is queried from a STUN server[12]. A handshake protocol performed through the signalling channel follows that, at the end of which the

[12]The STUN server is used by an endpoint to determine the IP and port number allocated to it. See `https://tools.ietf.org/html/rfc5389` for more details.

`RTCDataChannel` is open. Figure 3.4 presents an event trace diagram of the entire procedure.

3.4.4 The PCL `JavaScript` API

We now present the *Peer Communication Layer* `JavaScript` API. The PCL implementation is highly asynchronous, relying on `JavaScript Promises`[13] to ensure the order of certain operations. However, to support clean inter-operability with compiled `OCaml` code, we replace promises with callbacks in the API. Here, functions take two additional arguments, and always return void immediately. The first is an `on_success_callback`, called with the result of the operation when successful. The second argument is an `on_failure_callback`, which is called when the operation fails, with a description of the failure. The most important API operations are:

Start PCL

Description: Connects to the signalling server and initialises the layer. Must be called before the PCL is used for other operations.
Arguments:

- `signalling_server_url`—The address of the signalling server.

Result: A unique string identifier for this peer.

Bind `PCLSocket`

Description: Registers a `PCLSocket` with the peer and the signalling server. Must be called before accepting connections on the socket. Will fail if the `PCLSocket` with the given address is already registered.
Arguments:

- `socket_id`—Unique identifier for the *local* `PCLSocket`.
- `on_msg_callback`—Function which will be called whenever a message for this socket is received.
- `on_connection_to_callback`—Function which will be called whenever a remote `PCLSocket` connects to (or disconnects from) this socket.

Result: None.

Deallocate `PCLSocket`

Description: Unregister an already bound `PCLSocket` with the peer and the signalling server.
Arguments:

- `socket_id`—The identifier for the *local* `PCLSocket`.

[13] See the Language Specification: https://tc39.github.io/ecma262/#sec-promise-objects

Result: None.

Connect to `PCLSocket`

Description: Set up an `RTCDataChannel` to a remote `PCLSocket`. It will first bind a local `PCLSocket` with an unused-before identifier. The `RTCDataChannel` will be a bridge between the two `PCLSockets`.
Arguments:

- `remote_socket_id`—Identifier for the *remote* `PCLSocket`.
- `on_msg_callback`—Same as in **bind**, for the *local* `PCLSocket`.
- `on_connection_to_callback`—Same as in **bind**, for the *local* `PCLSocket`.

Result: Identifier of the *local* `PCLSocket` that was bound and is connected to the *remote*.

Send message to `PCLSocket`

Description: Sends a message from a local `PCLSocket` to a remote `PCLSocket`, over the `RTCDataChannel` set up between the two. The two sockets must be connected as a result of the **connect** operation, performed by either this peer or the destination peer.
Arguments:

- `local_socket_id`—Identifier for the *local* `PCLSocket`.
- `remote_socket_id`—Identifier for the *remote* `PCLSocket`.
- `message`—The message to send.

Result: None.

3.4.5 PCL `OCaml` bindings

PCL relied on `WebRTC` and `Socket.io` until now, so the core had to be implemented in `JavaScript`. However, because the PCL `JavaScript` API was designed to be clean and self-contained, we can continue the rest of the implementation in `OCaml`. We use some `Js_of_ocaml` features to provide `OCaml` bindings for the PCL API. This API hides behind opaque types the following: local and remote `PCLSockets`, messages, and failure explanations. This is done because all those elements are actually strings, which might be confusing to the programmer. This way, the type checker will ensure calls to the API do not mix up the arguments. Listing 3.1 contains part of the `OCaml` signature for these bindings.

3.4.6 PCL `Lwt`

The naïve `OCaml` bindings in §3.4.5 are somewhat cumbersome due to the additional `on_success_callback` and `on_failure_callback` arguments required for each call.

```
(** Opaque data types for various arguments.
  * The OCaml type checker will ensure arguments are not mixed up. *)
type msg_t           (* type of a message *)
type local_sckt_t    (* type of a local PCLSocket *)
type remote_sckt_t   (* type of a remote PCLSocket *)
type fail_reason_t   (* type of a failure reason *)

(** Conversion functions between string and the four types above.
  * Not shown here. *)

(** A failure callback is passed a reason for the failure. *)
type fail_callback_t = fail_reason_t → unit

(** A callback called when a message is received. It takes the source of the message
  * (remote socket), the destination (local socket), and the message. *)
type on_msg_callback_t = remote_sckt_t → local_sckt_t → msg_t → unit

(** A callback for when a remote socket is either connected to or
  * disconnected from the local socket *)
type on_connection_to_callback_t = local_sckt_t → remote_sckt_t → bool → unit

(** The on_success_callback for a Connect operation receives the identifier of the
  * local PCLSocket that was bound and is connected to the remote. *)
type on_success_connect_callback_t = local_sckt_t → unit

(** This is the "Connect to PCLSocket" operation from § 3.4.4. *)
val pcl_connect_to_address : remote_sckt_t → on_msg_callback_t
                           → on_connection_to_callback_t
                           → on_success_connect_callback_t
                           → fail_callback_t → unit
```

Listing 3.1: Snippet of the signature of the PCL API `OCaml` bindings. This contains the type definitions, as well as the signature of the connect function. The other functions are omitted for brevity.

More importantly, as discussed in §3.3, any asynchronous behaviour must eventually be wrapped inside Lwt promises.[14], so this is done now. Figure 3.5 exemplifies how we create Lwt alternatives to the callback-based PCL functions, while Listing 3.2 presents the cleaner signature of the PCL Lwt-based API.

```
(** Start PCL *)
val promise_start_comm_layer : string → string Lwt.t

(** Bind PCLSocket *)
val promise_bind_address : local_sckt_t → on_msg_callback_t
                         → on_connection_to_callback_t → unit Lwt.t

(** Deallocate PCLSocket *)
val promise_deallocate_address : local_sckt_t → unit Lwt.t

(** Connect to PCLSocket *)
val promise_connect_to_address : remote_sckt_t → on_msg_callback_t
                               → on_connection_to_callback_t → local_sckt_t Lwt.t

(** Send message to PCLSocket *)
val promise_send_msg : local_sckt_t → remote_sckt_t → msg_t → unit Lwt.t
```

Listing 3.2: Signature of the PCL Lwt-wrapped API functions. These operations are essentially as explained in §3.4.4.

[14]One might sensibly wonder why JavaScript promises were replaced with callbacks, only to replace those again with Lwt promises. This layer of indirection was required because Js_of_ocaml does not support direct conversion between JavaScript and Lwt promises.

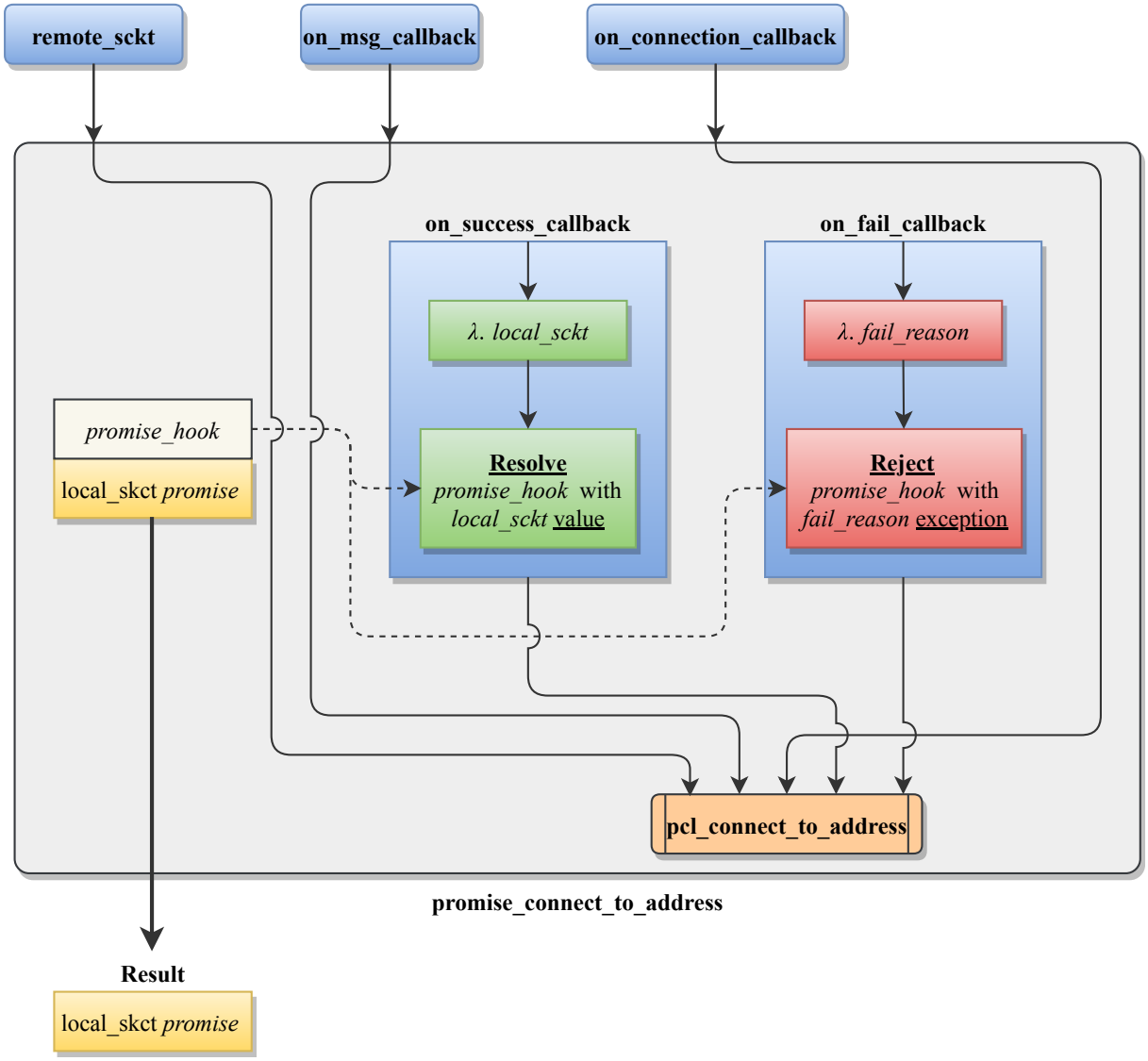


Figure 3.5: Wrapping a PCL asynchronous function with callbacks into a `Lwt`-based alternative. Blue are the arguments passed to the original `pcl_lwt_connect_to_address` function. Inside the new function, we create a new *pending* promise for the local socket, with its corresponding hook. This hook is packed into success & failure callbacks. The `on_success_callback`, if called, will *resolve* the promise with the local socket value. The `on_fail_callback` *rejects* the promise with an exception wrapping the failure reason. The `promise_connect_to_address` function immediately returns the *pending* promise.

3.5 OMQJS

By itself, the Peer Communication Layer is a breakthrough—it implements a communication system analogous to network sockets, which can be used by programs running in a web-browser. However, ZeroMQ provides additional features to standard network sockets, so replacing it in Actor requires further work. This section presents what this replacement supports, and how it is implemented on top of PCL. Perhaps unimaginatively, we named this JavaScript-based replacement OMQJS.

3.5.1 OMQSocket

Similarly to ZeroMQ's ZeroMQ_Socket, the core construct in OMQJS is the OMQSocket. Roughly speaking, an OMQSocket is a higher-level endpoint which can listen on multiple local PCLSockets, and be connected to multiple remote PCLSockets. This is akin to a ZeroMQ_Socket and multiple network sockets. The similarity is illustrated in Figure 3.6.

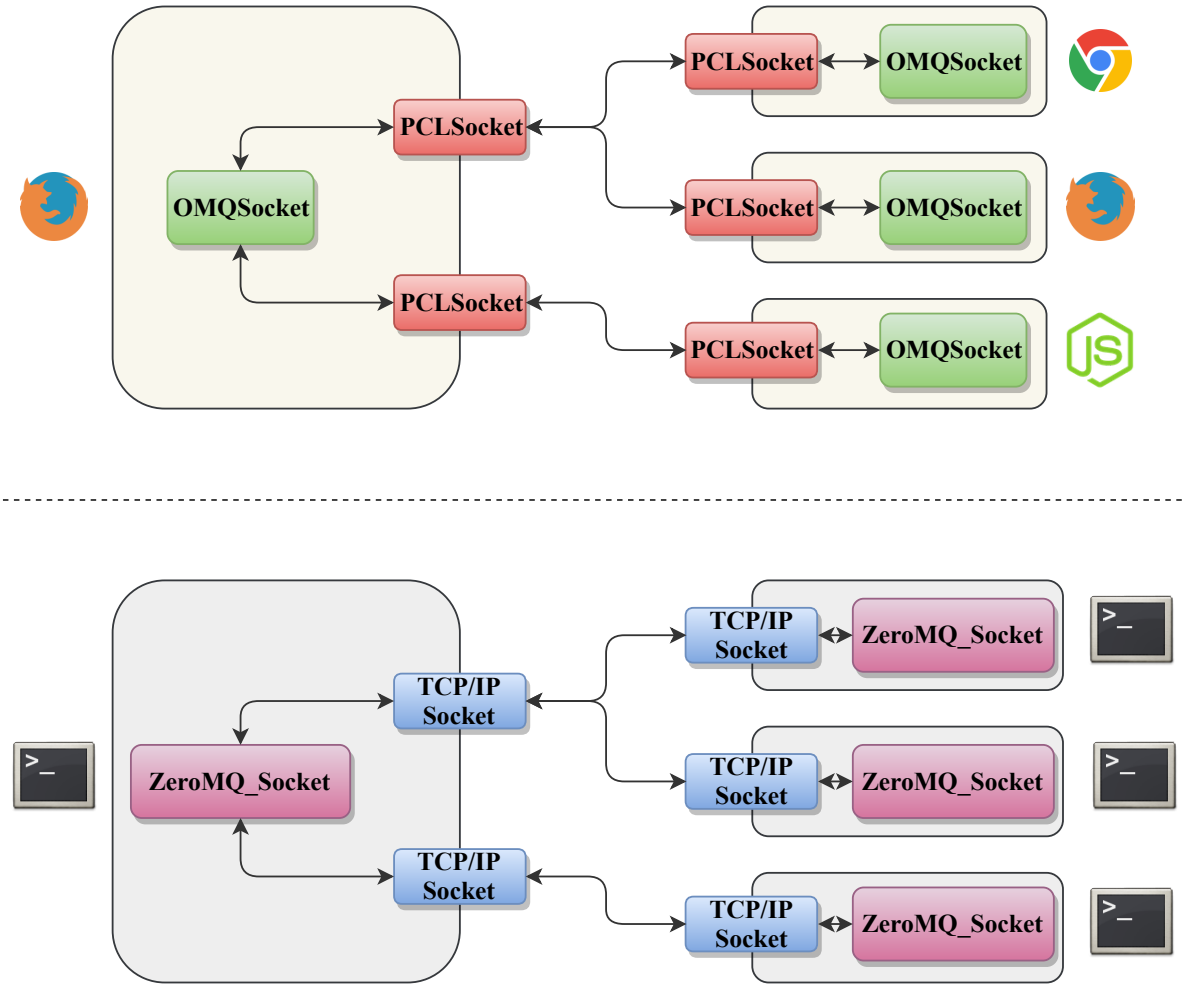


Figure 3.6: Example usage of OMQSockets and ZeroMQ_Sockets.

The behaviour of the OMQSocket depends to a large degree on its *kind*. Although ZeroMQ supports at least nine kinds of ZeroMQ_Sockets, for now we only replicate those four used by Actor. We briefly describe these in the following, while Table 3.3 compares certain characteristics of the socket. Their behaviour is mostly the same as in ZeroMQ, with the exception of a couple of differences which are highlighted.

REQ OMQSocket

This socket is typically used by a client to send requests to servers, and receive replies. It is restricted to an alternating pattern of send request, receive reply.

REP OMQSocket

This is analogous to REQ, on the server side. It allows the server to reply to one request at the time.

DEALER OMQSocket

This improves the REQ socket, allowing for multiple requests to be sent at any time. Replies can arrive in any order, so the application is responsible to match replies to requests, if necessary.

ROUTER OMQSocket

This enhances the REP socket, allowing the server to process multiple requests at a time. To allow this, the socket maintains a hashtable, mapping *opaque socket identifiers* (OMQ_ID) to remote sockets. For each receive call, the application receives the message and the OMQ_ID of the source client. After it processes the request, the application passes the reply alongside the corresponding OMQ_ID to the socket, which will use the hashtable and OMQ_ID to *route* the reply to the appropriate client. This process is sketched in Figure 3.7.

Kind	Send/Receive pattern	Message routing strategy	
		Incoming	Outgoing
REQ	send, receive, send, ...	last peer	random*
REP	receive, send, receive, ...	fair-queued	last peer
DEALER	unrestricted	fair-queued	random*
ROUTER	unrestricted	fair-queued	routed

Table 3.3: Comparison between OMQSocket kinds. Last peer for REQ means the only accepted reply is from the server the earlier request was sent to. Last peer for REP means the socket replies to the client which sent the request. Note: random* will be changed to round-robin in the future, as it is in ZeroMQ.

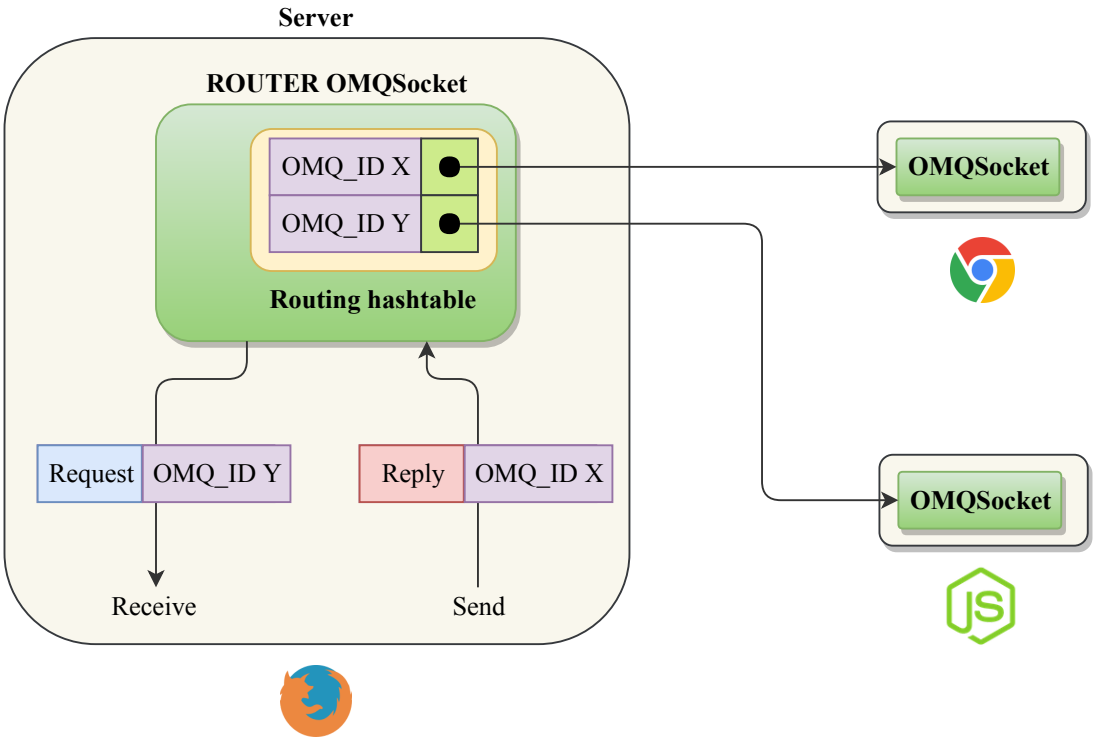


Figure 3.7: Illustration of `ROUTER OMQSocket` routing replies using a hashtable.

3.5.2 Simulating blocking operations

A few operations provided by a `ZeroMQ_Socket` might block in certain cases. For instance, a *receive* performed on a `REQ ZeroMQ_Socket` blocks until the server replies. We want `OMQJS` to simulate this behaviour using `Lwt` promises. PCL `Lwt` only considers some of the possible scenarios. For instance, it can simulate blocking sends only between already-connected `PCLSockets`. But `OMQJS` must consider cases such as when send is blocked because the `OMQSocket` is not connected to any remotes, or receive is blocked because no messages are available. Listing 3.3 illustrates how the latter scenario is handled.

```
(** The type of a promise hook. For instance, a hook of type int prhook_t corresponds
  * to a promise which can be resolved with an integer value. *)
type α prhook_t = α Lwt.u
(** The state of the socket. *)
type sckt_state_t =
    NotBlocked
    (* Blocked waiting to send an outstanding message. *)
  | BlockedSend of (* similar hooks, but details irrelevant here *)
    (* Blocked waiting to receive a message *)
  | BlockedRecv of msg_t prhook_t

(** Inside the OMQSocket.receive_message function *)
  match socket.incoming_msg_queue.pop_front() with
      (* Already have a message available, return it wrapped in a promise *)
      Some msg → return create_promise_resolved_with_value(msg)
    | None → begin (* Must block to wait for a message *)
        (* Create a new promise for the message, and its corresponding hook *)
        (promise, promise_hook) = new_promise_with_hook();
        socket.state ← BlockedRecv(promise_hook); (* Socket enters a blocked state *)
        return promise (* Immediately return the promise for the message *)
      end

(** Inside the on_msg_callback function passed to PCL when binding a PCLSocket.
  * This is called by PCL whenever a new message is received *)
  match socket.state with
      BlockedRecv(promise_hook) → begin (* The socket is blocked waiting for a message *)
        resolve_promise_with_value(promise_hook, msg); (* Resolve the promise *)
        socket.state ← NotBlocked (* The socket is not blocked anymore *)
      end
      (* If the socket is not blocked waiting for a message, queue the message *)
    | _ → socket.incoming_msg_queue.push(msg)
```

Listing 3.3: Pseudocode illustrating how the `OMQSocket` handles blocking *receive message* operations. The `OMQSocket` maintains a queue of incoming messages. When the application asks for a message, the socket checks the queue. If any messages are available, one is immediately returned. Otherwise, the `OMQSocket` enters a blocked state, returns a *pending* promise, and remembers the corresponding promise hook. In parallel, the `OMQSocket` passes an `on_msg_callback` to PCL when it binds a local `PCLSocket`. This callback is called by PCL whenever a new message is received, and it checks the socket state. If the `OMQSocket` is waiting for a message, the callback resolves the *pending* promise, and unblocks the `OMQSocket`. Otherwise, the callback simply adds the message to the queue. Note this omits many details, including the fact that operations have timeouts. If a timeout occurs before the promise is resolved, the socket leaves the blocked state, and the promise is rejected.

3.5.3 The OMQJS API

```
(** Polymorphic variant type denoting the possible kinds of OMQSockets. *)
type omq_sckt_kind = [`REP | `REQ | `DEALER | `ROUTER]
(*** -------------- OMQSocket module -------- *)
(** Type of an OMQSocket. Can only be one of the four REQ, REP, DEALER, ROUTER *)
type 'a omq_socket_t

(** Send a message. NOTE: Does not work on ROUTER sockets. *)
val send_msg : ?block:bool → [<`DEALER | `REP | `REQ] omq_socket_t
               → omq_msg_t → unit Lwt.t
(** Receive the message. NOTE: Does not work on ROUTER sockets. *)
val recv_msg : [<`DEALER | `REP | `REQ] omq_socket_t → omq_msg_t Lwt.t

(** Route a message using an omq_id. NOTE: Only works on a ROUTER socket. *)
val send_msg_with_id : [`ROUTER] omq_socket_t → omq_id_t → omq_msg_t → unit Lwt.t
(** Receive the message and the omq_id. NOTE: Only works on ROUTER sockets *)
val recv_msg_with_id : [`ROUTER] omq_socket_t → (omq_id_t * omq_msg_t) Lwt.t

(** Bind a local PCLSocket *)
val bind_local : [<`DEALER | `REP | `REQ | `ROUTER] omq_socket_t
                 → local_sckt_t → unit Lwt.t
(** Connect to a remote address *)
val connect_to_remote : [<`DEALER | `REP | `REQ | `ROUTER] omq_socket_t
                        → remote_sckt_t → local_sckt_t Lwt.t

(*** -------------- OMQContext module -------- *)
type omq_context_t

(** Create the OMQContext, which also starts PCL.
  * It takes the signalling server address as argument *)
val create : string → (string * omq_context_t) Lwt.t
val terminate : omq_context_t → unit
(** Create the four kinds of OMQSockets *)
val create_rep_socket : omq_context_t → [`REP] omq_socket_t
val create_req_socket : omq_context_t → [`REQ] omq_socket_t
val create_dealer_socket : omq_context_t → [`DEALER] omq_socket_t
val create_router_socket : omq_context_t → [`ROUTER] omq_socket_t
```

Listing 3.4: Part of the OMQJS API.

The other important module in the OMQJS implementation is the OMQContext. This starts the PCL, and creates, keeps track of, and closes OMQSockets. The most important elements of the API are shown in Listing 3.4. Note the implementation uses polymorphic variant types[15] to denote the OMQSocket kind. This ensures the

[15] http://caml.inria.fr/pub/docs/manual-ocaml-400/manual006.html

OCaml type-checker will detect any inappropriate uses of a ROUTER OMQSocket. For instance, a program which sends a message without an OMQ_ID through a ROUTER OMQSocket would not compile.

This API provides alternatives for all the ZeroMQ functions used in Actor. Following the work described in §3.3, we entirely remove the ZeroMQ dependency from PS Actor, and effortlessly switch to using OMQJS.

3.6 Job Spawning Layer

As discussed in §3.2, deployment of Actor programs in the browser has a second requirement—creating a tool enabling Actor_workers to spawn jobs, which happens in two steps. First, the job master (this is defined in §3.1 and Figure 3.1, it is the first instance of a PS job) must determine the job name, and send it to Actor_workers, alongside other job arguments. In native-deployed Actor, this is the executable name, extracted from the command-line arguments. In the second step, Actor_workers start the job locally. When running on Unix, this is done using fork and execv. We describe here our browser-supported alternative job spawning mechanism. Similarly to PCL, we provide OCaml bindings for JSL, obtaining the simple API displayed in Listing 3.6.

```
<p id="actor_job_name">test_job</p> <!-- This is the job name JSL extracts -->
<!-- jQuery library -->
<script id="jquery" src="link/to/jquery/library"></script>
<!-- Socket.io library -->
<script id="io_socket_library" src="link/to/socket/io/library"></script>
<!-- The WebRTC configuration for PCL -->
<script id="pcl_config" src="link/to/pcl/config"></script>
<!-- The JS PCL implementation -->
<script id="pcl_script" src="link/to/pcl/script"></script>
<!-- The JSL implementation -->
<script id="jsl_layer" src="link/to/jsl/script"></script>
<!-- The script for the actual job -->
<script id="actor_job_script" src="link/to/main/script/of/the/job"></script>
```

Listing 3.5: Part of the standardised HTML file describing a browser-deployed Actor Pure program.

Browser-deployed Actor Pure programs are specified by HTML files containing a series of JavaScript files, for multiple reasons. Firstly, they rely on external libraries like Socket.io, which must be loaded from external sources. Secondly, it is inefficient

to copy the JavaScript PCL implementation inside every program, when it can be cached by the browser. Therefore, we standardise the HTML file, and always specify the job name inside. This is illustrated in Listing 3.5. JSL can explore the HTML document using jQuery[16], and extract the job name.

For the second part, in order to spawn a new job given its name, JSL simply creates a button on the Actor_worker page, linking to the HTML file describing the job. If the button is clicked, a new tab (or window) is opened, which loads and starts the job. To pass arguments to the new job, JSL first transforms the argument list to a URL encoded string using JSON. This string is appended to the job link in a query string format. JSL can parse this string on the job side, and extract the arguments. For example, the following link starts the job named job_one, having the list ["test_arg"] as its arguments:

http://preconfigured.base.link/**job_one**.html?*args*=***%5B%22test_arg%22%5D***

```
(** Get the current job name *)
val jsl_get_job_name : unit -> string

(** Get the sys args. As customary, the very first one is the job name. *)
val jsl_get_sysargs : unit -> string array

(** Spawn the job with this name, while passing those arguments. *)
val jsl_spawn_job_with_args : string -> string array -> unit
```

Listing 3.6: The JSL OCaml API.

[16] https://jquery.com

4

Evaluation

In this chapter, we review the outcomes of the work presented in Chapters 2 and 3, and we assess to what degree these achieve the objectives we aimed for in Chapter 1. We continue to discuss the two main pieces of work separately, similarly to how they were introduced in the previous two chapters. Therefore, §4.1 evaluates the modifications we made to `Owl`, while §4.2 examines the extensions which lead to `Actor Pure`.

4.1 `Owl` Base

Our first main objective was providing support for developers to write machine learning and data processing programs in `OCaml`, then smoothly compile them to `JavaScript` and deploy them in a browser or on `Node.js`. Achieving this goal required, among other things, implementing in pure `OCaml` a considerably-sized module of functions operating on multi-dimensional arrays, and significantly restructuring the `Owl` numerical library. We employ three criteria in evaluating whether this work accomplishes our objective:

- Firstly, we prove users can write `OCaml` programs using `Owl`, compile them to `JavaScript`, then deploy them in the browser.
- Secondly, we demonstrate this process is painless, the developers' productivity is not disrupted, but in fact increased—they can write a single program which can be deployed on both native and `JavaScript` platforms with minimal effort.
- Finally, although it was beyond the scope of this project, we discuss the effectiveness of `JavaScript` deployment, identifying performance limitations that must be addressed in the future.

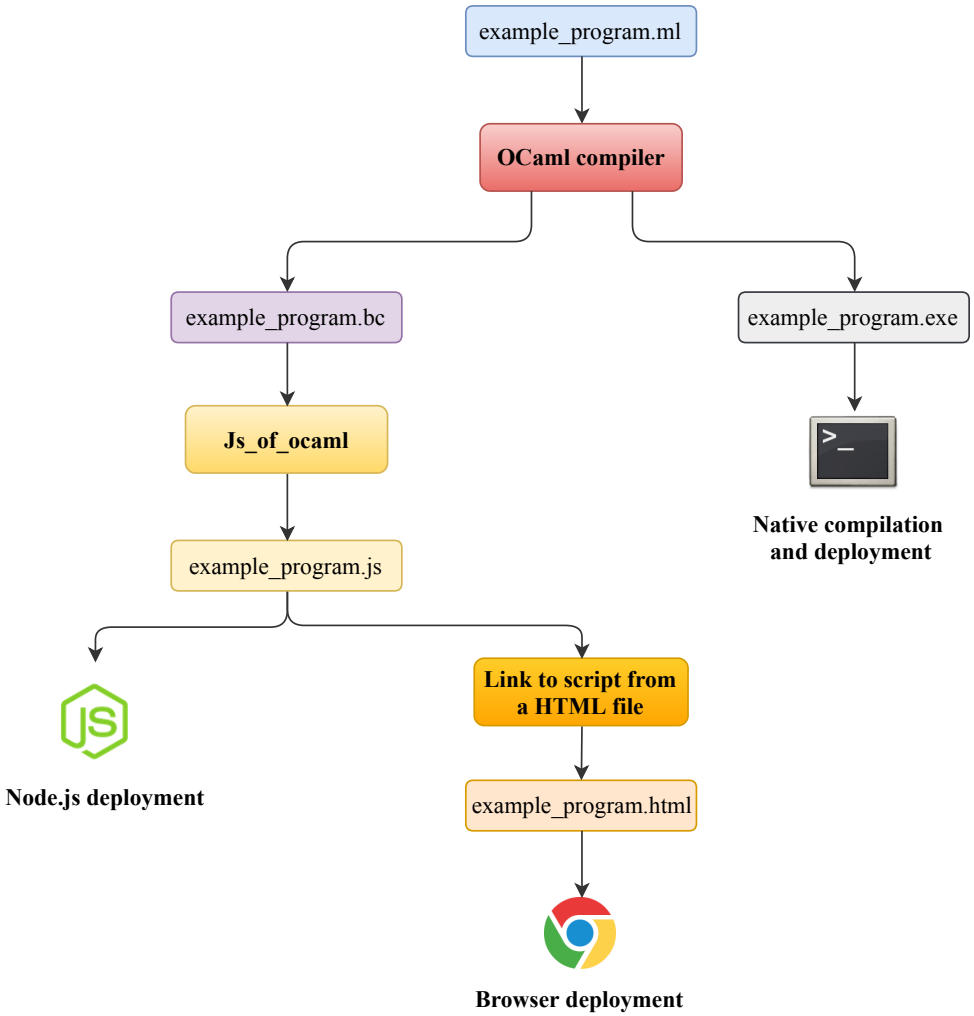


Figure 4.1: Compilation of `Owl` programs to `JavaScript`.

4.1.1 Compilation to `JavaScript` & cross-platform deployment

To allow compilation to `JavaScript` and browser deployment, developers must meet a number of requirements when writing `Owl`-based programs:

- Cannot use any `Owl` functionalities which are outside of `Owl Base`, because those are designed for native compilation, and might rely on external `C` libraries. Furthermore, developers must be careful to only link `Owl Base`, as `Owl` itself performs certain unsupported operations when linked. The reason for this requirement is explained in more detail in §2.5.

- Be careful not to use any `OCaml` modules or libraries which are not supported by `Js_of_ocaml`. Most notable unsupported or only partially-supported modules are the following standard `OCaml` modules—the `Unix` module, from which only time-related functions are supported, the `Thread` module, and the `Sys` module. However, this is arguably a foreseeable requirement—developers would not expect threading or access to `Unix` functions when aiming for browser deployment.

- Make sure to also compile the programs to OCaml bytecode, because that is what Js_of_ocaml expects.

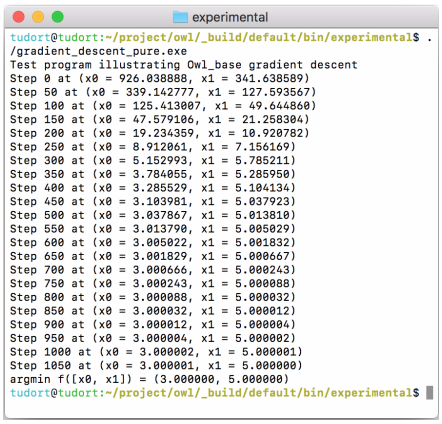
If these requirements are met, compiling to JavaScript is straightforward—the developer only has to pass the bytecode produced by the OCaml compiler as input to Js_of_ocaml. This can be done on the command-line, for instance `js_of_ocaml example_program.bc` producing a JavaScript file corresponding to the OCaml program `example_program.ml`. However, if the developer uses the popular Dune (JBuilder) build system (which Owl also uses), this has Js_of_ocaml support which can automatically produce the JavaScript file, alongside the native and bytecode compilation output. The JavaScript program can easily be executed in Node.js, but deploying it in the browser requires one additional step—linking to the script from an HTML file the browser can open. Although we provide a tool which does this automatically for testing purposes, developers have different requirements regarding code distributions, so they will most likely handle this straightforward step themselves. We illustrate this process in Figure 4.1.

To demonstrate the cross-platform deployment process we implemented a simple program which uses Owl's regression features to compute:

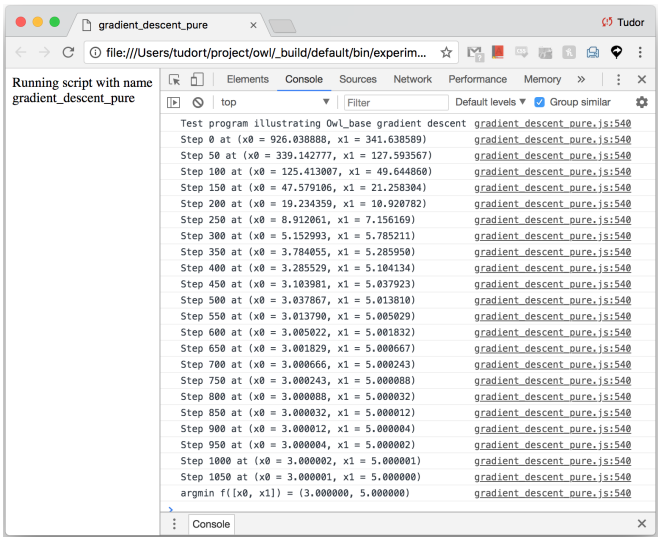
$$\underset{\vec{x}}{\operatorname{argmin}} \left\| \vec{x} - \begin{bmatrix} 3 \\ 5 \end{bmatrix} \right\|_2^2$$

where $\vec{x}$ is a vector of length 2 represented using an Owl N-d array. The program starts with random values in $\vec{x}$, then uses gradient descent to find the optimal vector. Figure 4.2 illustrates this program running on multiple platforms.

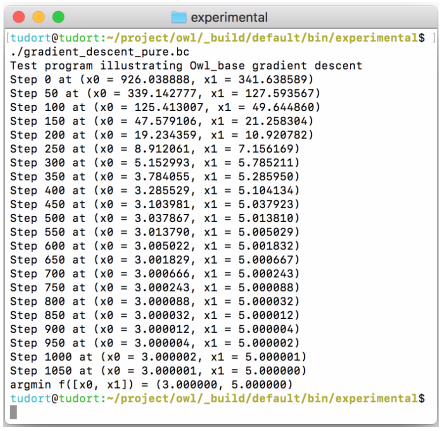
So far, this provides evidence that, with respect to our first criterion, the project successfully accomplishes the objective—developers can write Owl programs in OCaml, but compile them to JavaScript and execute them in the browser. Even more so, the same program can be deployed natively, or in Node.js. Yet, this only partially satisfies the second criterion, because in what was shown so far, the program is limited to using the pure OCaml implementation of N-d arrays, even when deployed on native platforms. This implementation is naturally expected (confirmed in §4.1.3) to be slower for certain operations than the original N-d array implementation, which relies on very efficient C libraries for linear algebra operations. Therefore, if indeed limited to only using Owl Base, targeting multiple platforms with the same program would be a significant drawback, and the project would fail to satisfy the second criterion. However, this is actually not the case.



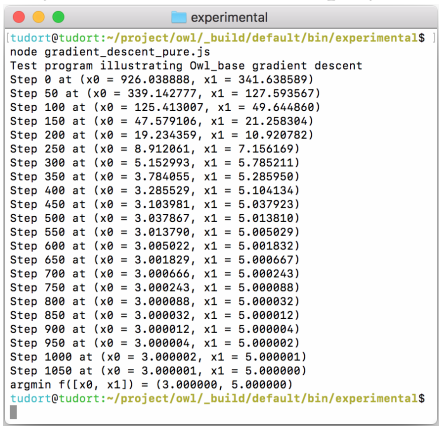
(a) Native deployment.



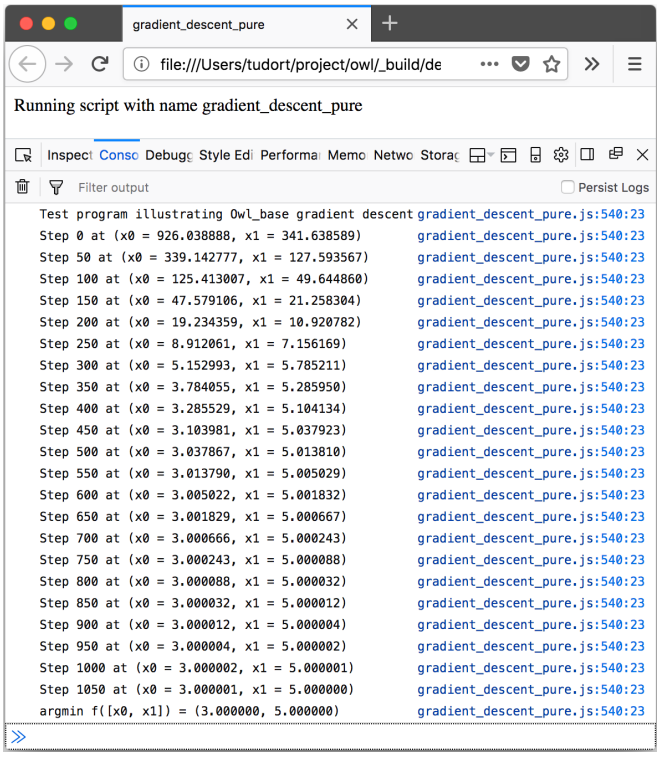
(b) Deployment in Google Chrome.



(c) Bytecode executable deployment.



(e) `Node.js` deployment.



(d) Deployment in Mozilla Firefox.

Figure 4.2: The same gradient descent program implemented using `Owl` is executed on multiple platforms, including two popular web browsers.

We designed our work such that all N-d array implementations conform to the same signature, i.e. they expose identical interfaces. This means developers can switch between the two N-d array implementations, depending on the targeted platform, using a simple programming pattern—packaging the program into a functor. The developer continues to assume it can only access `Owl Base` when writing the code, but it can feed in the more efficient original N-d arrays when compiling for native deployment. This pattern is illustrated in Listing 4.1 using a fabricated example, but we actually utilised it heavily when implementing unit tests (§4.1.2) and benchmarks (§4.1.3).

```
(** In the "main.ml" file, this is the functor creating the actual program,
  * once applied to an actual N-d array implementation.
  * NOTE: This cannot use any Owl function outside Owl Base,
  * because it has to be compile-able to JS. *)
module MakeProgram(NdarrayPlaceholder : Owl_types.Ndarray_Basic) = struct
  (** This is how the program would be implemented originally,
    * but instead of using a specific N-d array implementation,
    * the code will use the functor argument NdarrayPlaceholder *)
   let a = NdarrayPlaceholder.zeros [|1; 2|] in
   let b = NdarrayPlaceholder.ones [|1; 2|] in
   let s = NdarrayPlaceholder.add a b in
   NdarrayPlaceholder.print s
 (** ----- end of the 'original' program -----  *)
end
(** ----- end of "main.ml" ----- *)

(** In "main_pure.ml" file, the "single-line" program which can be compiled to JS.
  * Uses single-precision pure OCaml N-d array. *)
include Main.MakeProgram(Owl_base_dense_ndarray.S)
(** ----- end of "main_pure.ml" ----- *)

(** In "main_native.ml" file, the "single-line" program intended for native deployment.
  * Uses original, efficient implementation of single-precision N-d arrays *)
include Main.MakeProgram(Owl_dense_ndarray.S)
(** ----- end of "main_native.ml" ----- *)
```

Listing 4.1: Programming pattern for efficient cross-platform deployment of `Owl` programs. This is a simple fabricated example, with a single main file. Of course, more complicated programs spread across many files can be implemented, as long as the main functor-based entry point remains exposed.

4.1.2 Unit tests

Thorough testing is a core pillar of good software engineering practices, making it expected but also so standard in many large-scale projects that it can often be omitted from the corresponding write-ups. In this case however, we briefly discuss it explicitly for two reasons. Firstly, this highlights the versatility of the pattern presented in Listing 4.1, thus standing as evidence of the project's success with respect to the multi-platform support objective. Secondly, unit testing serves a crucial additional purpose in our work—it monitors that the cross-platform support does not lead to platform-specific behaviour in `Owl` programs. The library now has two almost entirely independent N-d array implementations, which must perform identically for our objective to be accomplished.

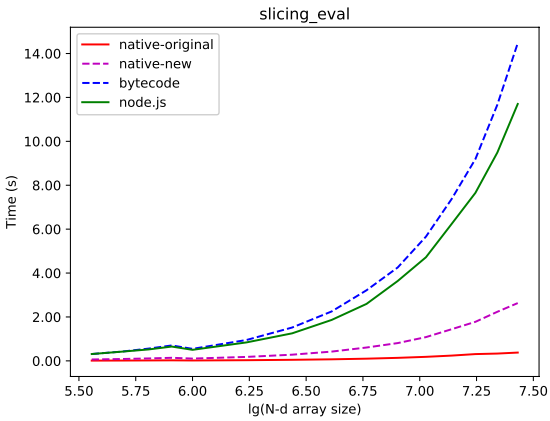
In order to establish this verification layer, we selected most of the unit tests relevant to the original N-d array implementation, and adapted them using the pattern in Listing 4.1. These tests explicitly cover significant features in the N-d array module, including slicing and convolution operations, but indirectly test many other functions. Therefore, because the tests are essentially the same, but are applied on different implementations, this acts as a control mechanism that could detect any functional discrepancies between those implementations. Obviously, as all testing, this can only serve as a warning system, and is not a guarantee of equivalence.

4.1.3 Benchmarks

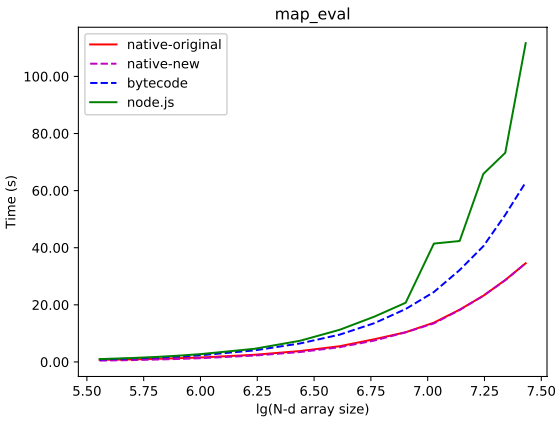
Due to high risks and difficulty of accurately estimating the amount of changes required to adapt the libraries to suit our requirements, the project objectives have focused from the beginning on providing proof-of-concept implementations rather than fully-fledged, production-ready software. The analysis of our `Owl` work has so far shown the project was highly successful in that regard—we accomplished our goal to create a working prototype of a system providing support for `OCaml`-to-`JavaScript` compilation of `Owl` programs, and almost seamless cross-platform deployment. Nevertheless, evaluation would not be complete without scoping out what the performance limitations are, and thus suggesting directions for further efforts aiming to improve and build upon this project.

For this purpose, we measured and compared the performances of the original and the new N-d array implementations, focusing on operations which sit at the core of the N-d array module. Our methodology involved writing the benchmarks as functors, like in Listing 4.1, to ensure the operations tested are identical for both modules. We then compile the original N-d array module benchmarks natively,

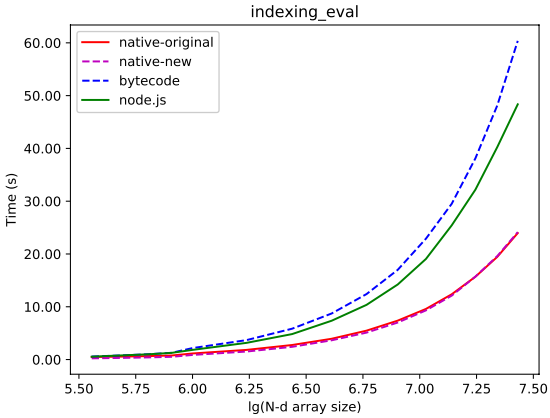
while the benchmarks for the new module are compiled to three versions—bytecode, native, and JavaScript. Compiling both modules for native execution lets us identify differences between the implementations themselves, rather than discrepancies correlated with the execution platform. Each benchmark is executed with N-d arrays of increasing size and number of dimensions. Each measurement is taken 5 times, then the average execution times are plotted. We discuss here the most instructive results.



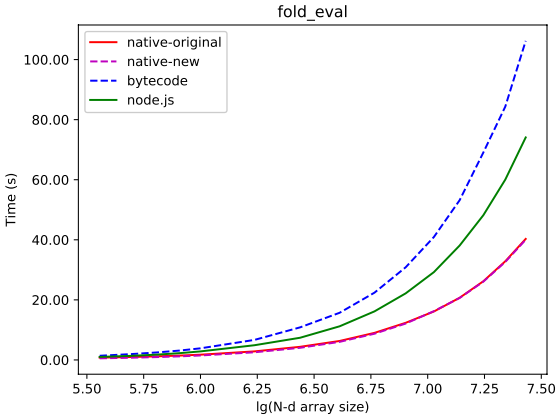
(a) Evaluation of slicing operations.



(b) Evaluation of the map function.



(c) Evaluation of indexing operations.



(d) Evaluation of the fold function.

Figure 4.3: Evaluation of a selection of N-d array operations in Owl. The original N-d array implementation compiled natively is denoted "native-original". The other three lines represent the new N-d array modules, deployed as a native executable ("native-new"), bytecode executable ("bytecode"), and compiled to JavaScript and deployed in Node.js. The horizontal axis contains the log_{10} of N-d array size the benchmark program operates on.

The first thing to note is that fold, indexing, and map operations share the same pure OCaml implementation in both N-d array modules. This immediately explains

why the native deployments perform exactly the same in Figures 4.3b to 4.3d. Slicing, on the other hand, is performed using external libraries in the original module, which explains why, in Figure 4.3a, the native deployment of this operation is much faster for the original implementation than the new one. This confirms the expectations that functions which were fully re-implemented in pure `OCaml` are less efficient than the original versions, and therefore, they will require optimisations in the future.

In all benchmarks, the bytecode and `JavaScript` versions become much slower as the N-d array increases in size. This is not unforeseen however—multi-platform versions will almost always trade performance for more deployment options, and this is no exception. Nonetheless, we can aim to reduce this gap in the future, especially when targeting `JavaScript` deployment. We could optimise the `OCaml` implementation keeping in mind the limitations of `JavaScript`, or we can try to replace `Js_of_ocaml` with an alternative such as `BuckleScript`.

Deployment	Mixed benchmark	Memory allocation benchmark
Native	1.28 ± 0.02 s	0.43 ± 0.05 s
Bytecode	1.85 ± 0.03 s	0.47 ± 0.05 s
`JavaScript`	7.88 ± 0.14 s	14.72 ± 0.04 s

Table 4.1: Comparison memory allocation impact on `JavaScript` code performance. Both benchmark programs are compiled natively, to bytecode, and to `JavaScript` using `Js_of_ocaml`. Each measurement is repeated 5 times, we report the mean execution time and the standard deviation, in seconds. The mixed benchmark program allocates a `BigArray` containing about 3×10^7 elements, and then iterates over it, summing up its contents. The second benchmark only allocates a `Bigarray` about 10 times larger, and does *nothing* else. The difference between bytecode and `JavaScript` performance is significantly larger in the second benchmark, indicating memory allocation considerably hinders performance. Note that neither program uses `Owl`, only the standard `OCaml Bigarray` module.

This brings us to arguably the most interesting result—except for the `map` benchmark, `JavaScript` code is slightly faster than its bytecode alternative. Perhaps surprising at first, `Js_of_ocaml`'s documentation confirms that in many cases, the `JavaScript` compilation output runs faster than the corresponding bytecode[1]. Immediately, this raises the question why `map` is different, with the `JavaScript` code performing comparatively much worse as the N-d array grows. We attribute this to the more considerable memory allocation performed by `map` than the other operations—`map` outputs a second N-d array of the same size as the input, which is

[1] http://ocsigen.org/js_of_ocaml/3.1.0/manual/performances

much larger than the additional slice potentially allocated by slicing operations, or a dimensionality-reduced N-d array produced by `fold`. `Js_of_ocaml`'s documentation justifies this theory, declaring that memory allocation might be less efficient. An additional measurement we performed provides further support for this theory, as argued in Table 4.1. Therefore, this observation gives some direction for further work aiming to improve the performance of resulting `JavaScript` code—optimisations should focus on reducing memory allocation, perhaps aiming to reuse memory as much as possible.

4.2 Actor Pure

The second major objective of this project was to implement a proof of concept solution for distributing computations in the web-browser, using the `Actor` library. We demonstrate here that we have overcome the crucial challenges entailed by this goal, and we have a functional prototype. Nevertheless, we identify an issue which requires solving before the project can support fully-fledged distributed ML or data processing applications. However, this problem is attainable, and we suggest a possible solution.

To illustrate a working prototype, we implemented a trivial job which uses the `Actor Pure` parameter-server module to create a few random integer key-value pairs, and distribute them across all job instances. Each instance then increments the value for each key, and sends it back to the PS. The job does not perform anything useful, but it allows us to demonstrate a few crucial achievements:

- `Actor Pure` can be compiled to `JavaScript`;
- The `JavaScript` versions of the main modules in `Actor Pure` can run in the browser;
- More importantly, they can communicate with each other directly, using `WebRTC`;
- And finally, `Actor_pure_workers` can spawn instances of new jobs locally, when necessary.

It is almost impossible to convincingly portray the series of events which prove those claims in a limited number of still images, but we nevertheless present screenshots of a few of those events in Figure 4.4.

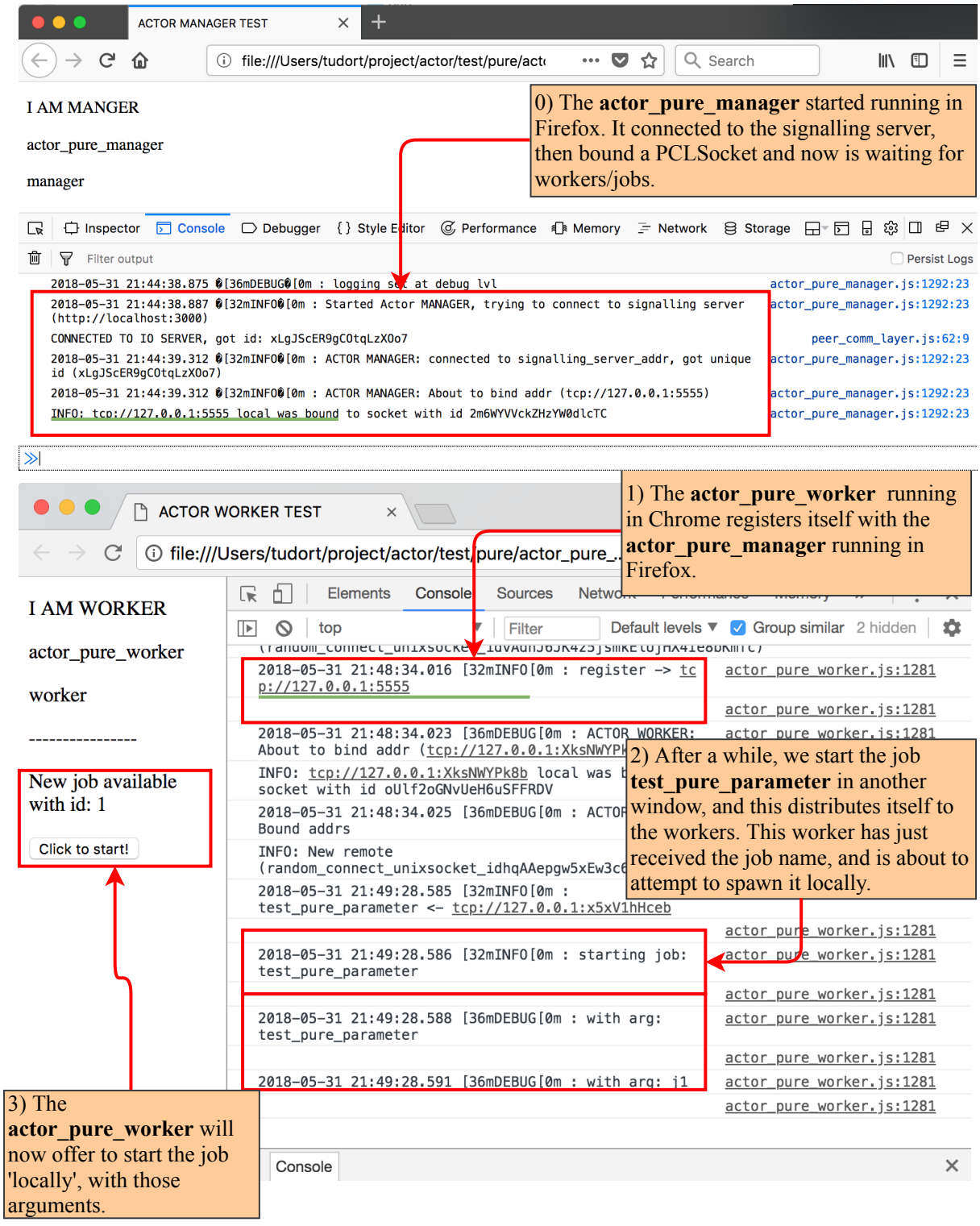


Figure 4.4: Illustration of `Actor Pure` modules executing and communicating with each other in Google Chrome and Mozilla Firefox.

4.2.1 Serialisation issues

Regrettably, we cannot yet demonstrate our Actor Pure work on a distributed neural network training example. Everything proceeds as expected until the parameter server attempts to serialise the data structure representing the neural network, such that it can be sent over to the clients. We identified the fault might be caused by the difference in JavaScript strings and OCaml strings—the latter are a mutable arrays of bytes, while JavaScript uses constant arrays of UTF-16 code points. Js_of_ocaml explicitly distinguishes these two, maintaining a special tag alongside OCaml strings in the compiled code, which explicitly identifies the type of the object. For some reason, JSON is unable to maintain this tag information when serialising the OCaml string representation. Therefore, OCaml strings must be *explicitly* converted to JavaScript strings *before* serialisation, then converted back after deserialisation. However, detecting such strings before the serialisation call is almost impossible, especially if they are "hidden" as leaves in complex data structures (such as those representing a neural network in Owl). Ultimately, this problem arises because the serialisation call is not *type safe*.

This issue was arguably difficult to anticipate, and we were unable to fix it yet because of time limitations. Nonetheless, we have already identified a potential solution—the OCaml Deriving library[2] would allow us to modify Owl and Actor such that serialisation becomes type-safe. We already confirmed Deriving is supported by Js_of_ocaml, and we intend to test this solution soon. Even with this issue, we consider this part of the project to also be successful—we have demonstrated that Actor can be deployed in the browser, the communication systems we implemented is functional, and so is the job spawning layer. Therefore, support for distributed computing in the browser is attainable following this design, and thus the proof of concept is delivered.

[2] https://github.com/ocsigen/deriving

5 Conclusion

5.1 Results

This project has successfully produced a proof-of-concept implementation for its ambitious overarching goal—support for distributed machine learning and data processing deployed at the edge of the network, in web-browsers. In doing so, the project has accomplished the following:

- Introduced the motivation for Edge Computing, its benefits, and some possible applications (Sections 1.1 and 1.2).
- Discussed related work, highlighting significant advantages and disadvantages in comparison to our the intended result (§1.3).
- Identified and compared options for `OCaml`-to-`JavaScript` compilation (§2.2).
- Introduced the `Owl` numerical library and described its original design (§2.1).
- Overall, implemented and restructured parts of `Owl` to support compilation to `JavaScript` (Chapter 2).
- Demonstrated the success of these modifications, and identified possible directions for optimisation (§4.1).
- Introduced the `Actor` library for distributed computations (§3.1).
- Modified `Actor` to support asynchronous communications (§3.3).
- Implemented a communication system analogous to TCP/IP network sockets (§3.4) for the web-browser, and a browser-compatible `ZeroMQ` replacement based on that (§3.5).
- Overall, adapted `Actor` to support browser deployment (Chapter 3). Demonstrated this is a successful proof of concept, but suggested further work which is required for a complete solution (Figure 4.3).

5.2 Further work

Although it is complete with respect to its main objectives, the project unlocks many options for further improvements, which were not explored because of time constraints. Many of these are introduced where relevant in this dissertation, but we also collect them here for completeness:

- Some of the functions in `Owl Base` have inefficient implementations, and should be optimised. Once that becomes a main objective, a thorough evaluation comparing `Owl` to frameworks such as `TensorFlow` would be valuable.
- Exploring the performance of `BuckleScript` as an alternative to `Js_of_ocaml` would be interesting. However, this requires removing the `Bigarray` dependency in `Owl Base`, and implementing everything using standard `OCaml` arrays.
- Implement type-safe serialisation in `Owl` and `Actor` using the `Deriving` library, as explained in §4.2.1.
- The `Job Spawning Layer` (§3.6) employs a somewhat primitive and unsafe solution. In the future, passing job arguments through cross-origin communication[1] might be a better solution.
- Compile `Owl Base` to micro-kernels using MirageOS[2].
- Explore the possibility of using `WebGL`-accelerated kernels when deploying in the browser.

[1] https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage
[2] https://mirage.io

Bibliography

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[2] Bob Evans. *Amazon To Become #1 In Cloud-Computing Revenue By Beating IBM's $17 Billion*, 2018 (accessed May 18, 2018). https://www.forbes.com/sites/bobevans1/2018/01/26/amazon-to-become-1-in-cloud-computing-revenue-by-beating-ibms-17-billion.

[3] Ahmad Farooq and Khan Hamid. An efficient and simple algorithm for matrix inversion. *Int. J. Technol. Diffus.*, 1(1):20–27, January 2010.

[4] J. Zhao, T. Tiplea, R. Mortier, J. Crowcroft, and L. Wang. Data Analytics Service Composition and Deployment on Edge Devices. *ArXiv e-prints*, April 2018.